DERS KİTABI

İşletim Sistemleri

Sercan Külcü

Giresun Üniversitesi · Bilgisayar Mühendisliği Bölümü

İşletim Sistemleri

Sercan Külcü
Giresun Üniversitesi, Bilgisayar Mühendisliği Bölümü
Giresun, Türkiye

Bu kitap, şu adreste yayımlanan ders notlarının basılı hâlidir:
`sercankulcu.github.io/teaching/operating-systems/`

Sunumlar ve çalışma soruları o sayfadadır. Bölümlerin kodu ve laboratuvar çalışmaları `github.com/sercankulcu/operating-systems-labs`, Java örnekleri `github.com/sercankulcu/operating-systems-java` deposundadır.

© Sercan Külcü. Tüm hakları saklıdır.

Notların 14 Kasım 2026 tarihli hâlinden üretilmiştir.

Önsöz

İşletim sistemi donanım ile uygulamalar arasında duran yazılımdır: işlemci zamanını, belleği, depolamayı ve aygıtları yönetir, programlara bunları güvenle ve paylaşarak kullanabilecekleri soyutlamalar sunar. Bu kitap, Giresun Üniversitesi Bilgisayar Mühendisliği Bölümü'nde verilen İşletim Sistemleri dersinin notlarından oluşur ve bölümleri dersteki sırayla izler. Örnekler C ile, dört bölümde ayrıca Java ile yazılmıştır.

Kitap dört kısımdır. Kısımlar sırasıyla şunlardır: **Temeller** (Bölüm 1–2), **Süreç Yönetimi** (Bölüm 3–7), **Bellek ve Depolama** (Bölüm 8–11) ve **İleri Konular** (Bölüm 12–15).

Her bölüm aynı düzeni izler. Başta öğrenme çıktıları yer alır. Biçimsel tanımlar, teoremler ve çözülmüş örnekler adlandırılmış kutulardadır. *Sık yapılan hata* kutuları, öğrencilerin en sık düştüğü yanılgıları gösterir. Kod izlemeleri, bir programın ya da algoritmanın adım adım nasıl çalıştığını gösterir. Bölüm; bir özet, bir kavram listesi, cevaplarıyla birlikte *Kendini Dene* soruları, ileri okuma önerileri ve kaynakçayla kapanır. Kitabın sonundaki terim sözlüğü, her İngilizce terim için notlarda kullanılan tek Türkçe karşılığı verir.

Bölümlerin kodu ve laboratuvar çalışmaları derse eşlik eden `github.com/sercankulcu/operating-systems-labs`, Java örnekleri `github.com/sercankulcu/operating-systems-java` deposundadır; ilgili klasörlerin adı her bölümün başında yazılıdır. Sunumlar ve çalışma soruları ders sayfasındadır.

Sercan Külcü
Giresun

İçindekiler

Dersin Haritası 1

KISIM I · TEMELLER

1 Giriş 4

1.1 İşletim sistemi nedir? 5
1.2 Konum ve modlar 10
1.3 İşletim sisteminin rolleri 15
1.4 Tarihçe 27
1.5 Donanım 36
1.6 G/Ç yöntemleri 41
1.7 İşletim sistemi çeşitleri 43
1.8 Temel kavramlar 46
1.9 Özet 50
1.10 Kavram listesi 51
1.11 Kendini Dene 52
1.12 İleri okuma 54
1.13 Kaynaklar 55

2 İşletim Sistemi Yapıları 56

2.1 Yapı ve hizmetler 57
2.2 Temel kavramlar 60
2.3 Koruma 69
2.4 Sistem çağrıları 73
2.5 POSIX ve Win32 çağrıları 85
2.6 Çekirdek yapıları 93
2.7 Sanal makineler ve çok işlemcili sistemler 107
2.8 Bağlama, yükleme ve açılış 110
2.9 Özet 113
2.10 Kavram listesi 114
2.11 Kendini Dene 114
2.12 İleri okuma 117
2.13 Kaynaklar 117

KISIM II · SÜREÇ YÖNETİMİ

3 Süreçler 120

3.1 Süreç kavramı 121
3.2 Süreç oluşturma 129
3.3 Süreç sonlandırma 144
3.4 Süreç durumları ve PCB 148
3.5 Çizelgeleme altyapısı 155

3.6 Süreçler arası iletişim 166
3.7 Mobil sistemlerde çoklu görev 176
3.8 Özet 177
3.9 Kavram listesi 178
3.10 Kendini Dene 179
3.11 İleri okuma 182
3.12 Kaynaklar 182

4 İş Parçacıkları 183

4.1 Temel kavramlar 184
4.2 Çok çekirdekli programlama 197
4.3 İş parçacığı modelleri 204
4.4 İş parçacığı kütüphaneleri 214
4.5 Örtük iş parçacığı 224
4.6 İş parçacığı sorunları 232
4.7 Süreç mi, iş parçacığı mı? 242
4.8 Özet 243
4.9 Kavram listesi 244
4.10 Kendini Dene 244
4.11 İleri okuma 246
4.12 Kaynaklar 246

5 CPU Çizelgeleme 247

5.1 Temel kavramlar 248
5.2 Temel algoritmalar 257
5.3 Round Robin 272
5.4 Çok düzeyli kuyruklar 279
5.5 Orantılı pay 284
5.6 Çok işlemcili çizelgeleme 289
5.7 Gerçek zamanlı çizelgeleme 295
5.8 İşletim sistemi örnekleri 303
5.9 Algoritmaları değerlendirme 309
5.10 Özet 312
5.11 Kavram listesi 313
5.12 Kendini Dene 314
5.13 İleri okuma 316
5.14 Kaynaklar 316

6 Senkronizasyon 318

6.1 Yarış durumu 319
6.2 Kritik bölge problemi 325
6.3 Yazılım yaklaşımları 328
6.4 Bellek modeli 335
6.5 Donanım desteği 344
6.6 Kilitler 352
6.7 Semaforlar 360

6.8 Klasik problemler 365
6.9 Monitörler 376
6.10 POSIX ve Java 385
6.11 Canlılık 391
6.12 Kilitsiz yapılara giriş 394
6.13 Özet 397
6.14 Kavram listesi 398
6.15 Kendini Dene 398
6.16 İleri okuma 400
6.17 Kaynaklar 401

7 Kilitlenme 402

7.1 Kilitlenme ve kaynaklar 403
7.2 Kilitlenmenin dört koşulu 409
7.3 Kilitlenmenin modellenmesi 413
7.4 Başa çıkma stratejileri 422
7.5 Tespit ve kurtarma 423
7.6 Kilitlenmeden kaçınma 437
7.7 Kilitlenmeyi önleme 453
7.8 Diğer sorunlar 462
7.9 Pratikte kilitlenme 468
7.10 Özet 473
7.11 Kavram listesi 474
7.12 Kendini Dene 475
7.13 İleri okuma 477
7.14 Kaynaklar 478

KISIM III · BELLEK VE DEPOLAMA

8 Bellek Yönetimi 480

8.1 Bellek donanımı 481
8.2 Adres alanı 489
8.3 Bağlama ve yükleme 497
8.4 Takas 502
8.5 Bitişik tahsis 504
8.6 Boş alan yönetimi 512
8.7 Sayfalama 517
8.8 TLB 526
8.9 Sayfa tablosu yapıları 531
8.10 Kesimleme 539
8.11 Gerçek mimariler ve uygulama 544
8.12 Özet 546
8.13 Kavram listesi 547
8.14 Kendini Dene 548
8.15 İleri okuma 550
8.16 Kaynaklar 550

9 Sanal Bellek 551

9.1 Sanal belleğe giriş 552
9.2 Talebe bağlı sayfalama 556
9.3 Performans: etkin erişim süresi 565
9.4 Yazınca kopyalama 568
9.5 Bellek eşlemeli dosyalar 571
9.6 Sayfa yer değiştirme 574
9.7 Çalışma kümesi 594
9.8 Algoritmaların karşılaştırması 599
9.9 Çerçeve tahsisi 601
9.10 Thrashing 604
9.11 Tasarım konuları 608
9.12 Çekirdek belleği 615
9.13 Gerçek sistemler 620
9.14 Özet 623
9.15 Kavram listesi 624
9.16 Kendini Dene 624
9.17 İleri okuma 626
9.18 Kaynaklar 627

10 Dosya Sistemleri 628

10.1 Neden dosya sistemi? 629
10.2 Dosya kavramı 630
10.3 Dizinler 641
10.4 Koruma 649
10.5 Disk düzeni 651
10.6 Tahsis yöntemleri 654
10.7 Dizinlerin gerçeklenmesi 665
10.8 Boş alan yönetimi 667
10.9 Performans 673
10.10 Tutarlılık ve kurtarma 678
10.11 VFS ve ağ dosya sistemi 687
10.12 Özet 689
10.13 Kavram listesi 690
10.14 Kendini Dene 691
10.15 İleri okuma 693
10.16 Kaynaklar 693

11 Giriş/Çıkış 694

11.1 G/Ç sistemine genel bakış 695
11.2 G/Ç donanımı 698
11.3 Programlanmış G/Ç 702
11.4 Kesmeler 705
11.5 Doğrudan bellek erişimi (DMA) 712
11.6 G/Ç yazılımı 717
11.7 G/Ç arayüzleri 726

11.8 Saatler ve diğer aygıtlar 730
11.9 Diskler 735
11.10 Disk kolu çizelgeleme 739
11.11 SSD ve NVMe 752
11.12 RAID ve kararlı depolama 757
11.13 Özet 765
11.14 Kavram listesi 766
11.15 Kendini Dene 767
11.16 İleri okuma 769
11.17 Kaynaklar 769

KISIM IV · İLERİ KONULAR

12 Çok İşlemcili Sistemler 771

12.1 Neden çok işlemci? 772
12.2 Çok işlemcili donanım 774
12.3 Çok işlemcili işletim sistemi türleri 790
12.4 Senkronizasyon 794
12.5 Çok işlemcili çizelgeleme 802
12.6 Çoklu bilgisayarlar 806
12.7 Sanallaştırma 819
12.8 Dağıtık sistemler 825
12.9 Özet 837
12.10 Kavram listesi 838
12.11 Kendini Dene 838
12.12 İleri okuma 840
12.13 Kaynaklar 841

13 Güvenlik 842

13.1 Güvenlik hedefleri ve tehditler 843
13.2 Kriptografi 851
13.3 Koruma 863
13.4 Biçimsel güvenlik modelleri 874
13.5 Kimlik doğrulama 881
13.6 Yazılım açıkları 890
13.7 İçeriden saldırılar 898
13.8 Kötü amaçlı yazılımlar 899
13.9 Savunma ve yalıtım 904
13.10 Donanım düzeyinde güvenlik 908
13.11 Özet 910
13.12 Kavram listesi 911
13.13 Kendini Dene 912
13.14 İleri okuma 914
13.15 Kaynaklar 914

14 İşletim Sistemi Tasarımı 916

14.1 Tasarımın zorlukları ve hedefleri 917
14.2 Arayüz tasarımı 919
14.3 Sistem yapısı 927
14.4 Tasarım ilkeleri 934
14.5 Yararlı teknikler 943
14.6 Performans 950
14.7 Proje yönetimi 963
14.8 Test ve hata ayıklama 965
14.9 İşletim sistemlerinde eğilimler 977
14.10 Özet 980
14.11 Kavram listesi 981
14.12 Kendini Dene 982
14.13 İleri okuma 984
14.14 Kaynaklar 984

15 Linux 985

15.1 Linux'un kökleri 986
15.2 Arayüzler ve kabuk 989
15.3 Çekirdeğin yapısı 994
15.4 Süreçler ve iş parçacıkları 997
15.5 Sinyaller 1002
15.6 Çizelgeleme 1006
15.7 Önyükleme 1010
15.8 Bellek yönetimi 1011
15.9 Giriş/çıkış 1021
15.10 Dosya sistemleri 1025
15.11 Güvenlik 1035
15.12 Android 1042
15.13 Dersin kavramları Linux'ta 1043
15.14 Özet 1044
15.15 Kavram listesi 1045
15.16 Kendini Dene 1046
15.17 İleri okuma 1048
15.18 Kaynaklar 1048

EKLER

Ek A Terim Sözlüğü 1049

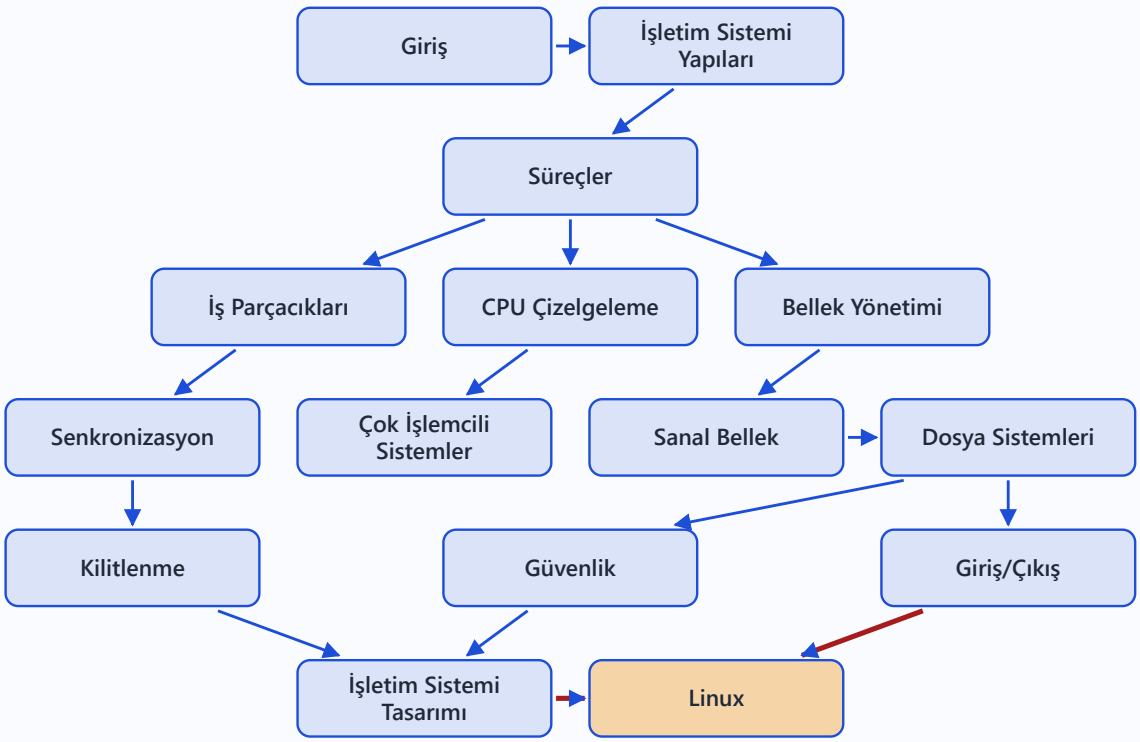
Ek B İşletim Sistemleri İçin Ön Bilgiler 1060

Ek C İşletim Sistemlerinin Öncüleri 1066

İŞLETİM SİSTEMLERİ

Dersin Haritası

Bu sayfa dersin bütün bölümlerini tek bir haritada birleştirir. Her bölüm bir düğümdür; bir ok, sonraki konunun önceki konunun hangi fikrine dayandığını gösterir. Derse başlamadan önce okunursa konuların nereye gittiğini, ders sırasında okunursa bulunulan yeri gösterir.



Şekil 1: Dersin haritası. Oklar, bir konunun hangi konunun üzerine kurulduğunu gösterir. Bütün yollar Süreçler'den geçer ve Linux'ta birleşir.

Şekil 1 konular arasındaki bağları gösterir. Her düğüm bir bölümdür; bir ok, sonraki konunun önceki-nin hangi fikrine dayandığını belirtir. Haritada yalnızca en yakın bağlar çizilmiştir; bir bölümün dayandığı bütün bölümler Tablo 1'dedir.

Konu (bölüm)	Dayandığı konu	Bağ
Giriş (1)	—	İşletim sisteminin iki rolü: genişletilmiş makine ve kaynak yöneticisi; çekirdek ve kullanıcı modu.
İşletim Sistemi Yapıları (2)	Giriş	Hizmetler sistem çağrılarıyla sunulur: süreç, adres alanı, dosya ve boru kavramları.

Konu (bölüm)	Dayandığı konu	Bağ
Süreçler (3)	İşletim Sistemi Yapıları	Çalışan program: fork, exec ve wait; süreç durumları ve süreç kontrol bloğu.
İş Parçacıkları (4)	Süreçler	Aynı adres alanında birden çok yürütme akışı; kullanıcı ve çekirdek düzeyi iş parçacıkları.
CPU Çizelgeleme (5)	Süreçler	Hazır kuyruğundaki süreçlerden hangisi işlemciyi alır: FCFS, SJF, Round Robin ve geri beslemeli kuyruklar.
Senkronizasyon (6)	İş Parçacıkları	Paylaşılan veriye eş zamanlı erişim yarış durumu doğurur: kritik bölge, mutex ve semafor.
Kilitlenme (7)	Senkronizasyon	Kilitleri yanlış sırada almak kilitlenmeye götürür: dört koşul, bankacı algoritması.
Bellek Yönetimi (8)	İşletim Sistemi Yapıları, Süreçler	Her sürecin adres alanı: mantıksal ve fiziksel adres, bitişik tahsis, sayfalama.
Sanal Bellek (9)	Bellek Yönetimi	Sayfalamanın talebe bağlı biçimi; sayfa hatası ve sayfa yer değiştirme algoritmaları.
Dosya Sistemleri (10)	İşletim Sistemi Yapıları, Sanal Bellek	Dosya kavramının diskteki karşılığı; bellek eşlemeli dosyalar ve sayfa önbelleği sanal belleği kullanır.
Giriş/Çıkış (11)	Süreçler, Dosya Sistemleri	Dosya sisteminin altındaki katman: aygıt sürücüleri, kesmeler ve DMA; bloke olan süreç bekleme kuyruğuna geçer.
Çok İşlemcili Sistemler (12)	CPU Çizelgeleme, Senkronizasyon	Birden çok işlemcide çizelgeleme ve senkronizasyon: önbellek tutarlılığı, spinlock'ların ölçeklenmesi.
Güvenlik (13)	İşletim Sistemi Yapıları, Dosya Sistemleri	Çekirdek ve kullanıcı modundan dosya izinlerine kadar koruma; erişim denetim listeleri ve yetenekler.
İşletim Sistemi Tasarımı (14)	Kilitlenme, Çok İşlemcili Sistemler, Güvenlik	Önceki bölümlerdeki seçimlerin ortak ilkeleri: mekanizma ile politikanın ayrılması, ödünleşimler.
Linux (15)	Giriş/Çıkış, Çok İşlemcili Sistemler, İşletim Sistemi Tasarımı	Bütün kavramlar gerçek bir sistemde: task_struct, EEVDF çizelgeleyicisi, sanal adres alanı, VFS.

Tablo 1: Haritadaki bağlar.

Haritanın ana omurgası şöyle okunabilir: **süreç → paylaşım → bellek → depolama → gerçek sistem**. Süreç kavramı dersin merkezidir; ondan üç dal çıkar. İşlemci dalı çizelgelemeyle, eş zamanlılık dalı iş parçacıkları, senkronizasyon ve kilitlenmeyle, bellek dalı sayfalama ve sanal bellekle ilerler. Sanal bellek dosya sistemine, dosya sistemi de G/Ç katmanına bağlanır. Çok işlemcili sistemler çizelgeleme ile senkronizasyonu birlikte yeniden sorar. Tasarım bölümü bu seçimlerin ortak ilkelerini toplar, Linux bölümü de hepsini çalışan bir çekirdekte gösterir.

KISIM I

Temeller

Bölüm 1–2

KISIM I · BÖLÜM 1

Giriş

Bilgisayarda çalışan her program, kendisinden daha alt düzeydeki bir yazılıma güvenir: işletim sistemine. Tarayıcı bir dosyayı diske yazarken diskin markasını bilmez. Oyun ekrana çizerken ekran kartının yazmaçlarına dokunmaz. Derleyici çalışırken başka programların belleğini bozamaz. Bu kolaylığın ve korumanın hepsini işletim sistemi sağlar. Not, işletim sisteminin donanım ile uygulamalar arasındaki yerini ve iki temel rolünü anlatarak başlar: donanımı kullanışlı soyutlamalarla sunan genişletilmiş makine ve kaynakları paylaştıran yönetici. Ardından çekirdek ve kullanıcı modunu, bu iki mod arasındaki denetimli geçişi, işletim sistemlerinin tarihsel gelişimini ve bu gelişimi yönlendiren donanım özelliklerini işler. Son kesimler G/Ç yöntemlerine, işletim sistemi çeşitlerine, politika ile mekanizma ayrımına ve bir C programının çalışan bir sürece dönüşmesine ayrılmıştır.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum01` · **Laboratuvar:** `github.com/sercankulcu/operating-systems-labs/lab01`

ÖĞRENME ÇIKTILARI

Tanım ve konum
İşletim sisteminin donanım ile uygulamalar arasındaki yerini ve iki temel rolünü (genişletilmiş makine, kaynak yöneticisi) açıklamak.

Koruma
Çekirdek modu ile kullanıcı modunu ve aralarındaki denetimli geçişi (sistem çağrısı, kesme, istisna) açıklamak.

Tarihsel gelişim
Toplu iş, çoklu programlama ve zaman paylaşımı fikirlerinin hangi sorunları çözdüğünü anlatmak; iki işin zaman çizelgesini çizip CPU kullanımını hesaplamak.

Donanım temelleri
İşlemci ve yazmaçların, bellek hiyerarşisinin ve çok çekirdekli yongaların işletim sistemini nasıl etkilediğini yorumlamak.

G/Ç yöntemleri
Yoklama, kesme ve DMA arasındaki temel farkı söylemek, CPU yükünü hesaplamak; kesme ile tuzağı ayırmak.

Kavramlar
İşletim sistemi çeşitlerini, politika–mekanizma ayrımını ve programın çalıştırılabilir dosyaya dönüşmesini örneklerle açıklamak.

1.1 İşletim sistemi nedir?

"İşletim sistemi" sözü günlük dilde çoğu zaman ekranda görülen masaüstüyle karışır. Başlat menüsü, dosya gezgini, duvar kâğıdı ve simgeler akla gelir. Oysa bunlar işletim sisteminin yalnız görünen yüzüdür, hatta çoğu sıradan birer programdır. Asıl iş görünmeyen yerde yapılır: hangi programın ne zaman çalışacağına, belleğin hangi parçasını kimin kullanacağına, diske giden isteklerin hangi sırayla işleneceğine karar veren yazılım katmanında. Bu kesim, o katmanın ne olduğunu ve neden var olduğunu anlatır.

1.1.1 Neden bir ara katman gerekir?

Bugünkü sıradan bir dizüstü bilgisayarın içine bakalım. Çok çekirdekli bir işlemci, gigabaytlarca bellek, bir SSD, kablosuz ve kablolu ağ kartları, bir grafik işlemcisi (GPU), klavye, dokunmatik yüzey, kamera, mikrofon, hoparlör ve birkaç USB bağlantısı vardır. Bu aygıtların her biri başka bir üreticiden gelir ve her birinin kendine özgü bir denetim yöntemi vardır. Bir SSD'ye veri yazmak için komut kuyrukları hazırlanır, blok adresleri hesaplanır, hata kodları denetlenir. Bir ağ kartından paket almak için halka biçimli tamponlar kurulur ve kesmeler ayarlanır.

Her uygulama geliştiricisinin bu ayrıntıların hepsini öğrenmesi ve kendi programında doğrudan yazması düşünülemez. Böyle olsaydı bir metin düzenleyicinin, piyasadaki bütün disk modelleri için ayrı kod içermesi gerekirdi. Üstelik aynı anda çalışan iki program aynı diske kendi bildiği gibi yazmaya kalkışsa diskteki veri birbirine karışırdı. Demek ki iki ayrı ihtiyaç vardır: donanımın karmaşıklığını gizleyen *ortak bir arayüz* ve aynı donanımı kullanan programlar arasında *düzen ve koruma*. İşletim sistemi bu iki ihtiyaca verilen yanıttır.

> **Tanım 1.1**
>
> **İşletim sistemi** (*operating system*, kısaca İS), bilgisayarın donanımını yöneten ve uygulamalara basit, güvenli ve ortak bir arayüz sunan yazılım katmanıdır.

Tanımdaki üç sıfatın her biri ayrı bir işe karşılık gelir. *Basit* arayüz, uygulamanın "şu dosyaya şu baytları yaz" demesinin yetmesidir; diskin içyapısı İS'nin sorunudur. *Güvenli* arayüz, bir programın hatasının ya da kötü niyetinin başka programlara ve sistemin bütününe zarar verememesidir. *Ortak* arayüz ise aynı programın farklı donanımlarda değişmeden çalışabilmesidir: bir C programı hangi SSD takılı olursa olsun aynı `write` çağrısıyla dosyaya yazar.

Bugün çok farklı ortamlarda çok farklı işletim sistemleri çalışır. Masaüstünde Windows 11, macOS, Ubuntu ya da Fedora gibi Linux dağıtımları ve ChromeOS yaygındır. Telefon ve tabletlerde Android, iOS/iPadOS ve HarmonyOS kullanılır. Sunucu ve bulut tarafında Linux açık farkla öndedir; Windows Server ve FreeBSD de yer bulur. Otomobillerin, ev aletlerinin ve nesnelerin interneti (*Internet of Things*, IoT) aygıtlarının içinde ise Zephyr, FreeRTOS, Contiki-NG, VxWorks ve QNX gibi küçük, özel amaçlı sistemler çalışır. Bu sistemlerin boyutları ve hedefleri birbirinden çok farklıdır, ama hepsi aynı iki temel sorunu çözer.

1.1.2 Çekirdek ve geri kalan yazılım

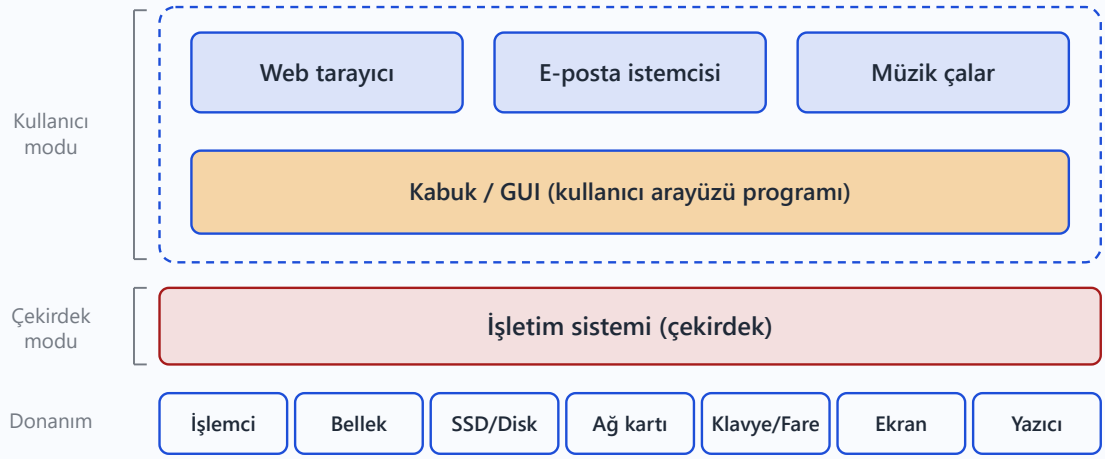
Bir işletim sistemi kurulduğunda diske binlerce dosya yazılır: sürücüler, kütüphaneler, yardımcı programlar, yazı tipleri, ayar araçları, masaüstü ortamı. Bunların hepsini "işletim sistemi" saymak, terimi neredeyse anlamsız kılar. Bu yüzden ders kitapları dar ve geniş iki anlam ayırır.

> **Tanım 1.2**
>
> **Çekirdek** (*kernel*), işletim sisteminin bilgisayar açık kaldığı sürece bellekte duran ve işlemcinin ayrıcalıklı modunda çalışan kısmıdır. Dar anlamda "işletim sistemi" denince çekirdek kastedilir.

Çekirdek; süreçleri yaratır ve çizelgeler, belleği dağıtır, dosya sistemlerini ve aygıt sürücülerini barındırır, kesmeleri karşılar. Geri kalan yazılımın büyük kısmı ise çekirdeğin üstünde, sıradan programlar gibi çalışır. Kullanıcının bilgisayarla konuştuğu iki arayüz de bu ikinci gruptadır: **kabuk** (*shell*), yani komut satırı, ve **GUI** (*graphical user interface*: grafik kullanıcı arayüzü). İkisinin ayrıntısına aşağıda dönülecek; burada önemli olan, ikisinin de çekirdeğin parçası olmamasıdır.

1.1.3 Bilgisayar sisteminin bileşenleri

Bir bilgisayar sistemini katmanlar hâlinde düşünmek, kimin neyi yaptığını görmenin en kolay yoludur. Şekil 1.1 bu katmanları alttan üste gösterir.



Şekil 1.1: Bilgisayar sisteminin katmanları. En altta donanım (işlemci, bellek, SSD/disk, ağ kartı, klavye/fare, ekran, yazıcı) bulunur. Üstünde çekirdek modunda çalışan işletim sistemi yer alır. En üstte, kesikli çerçevenin içinde, kullanıcı modunda çalışan programlar vardır: web tarayıcı, e-posta istemcisi, müzik çalar ve onlarla aynı düzeyde kabuk ya da GUI.

En alttaki **donanım** katmanı işlemciyi, belleği, depolama aygıtlarını ve giriş/çıkış (G/Ç) aygıtlarını içerir. Bunlar fiziksel bileşenlerdir; kendi başlarına yalnız çok basit komutları anlar. Hemen üstteki **işletim sistemi**, donanımı doğrudan denetleyen tek yazılımdır ve işlemcinin ayrıcalıklı modunda çalışır. En üstteki katmanda ise geri kalan her şey bulunur: kullanıcının çalıştırdığı uygulamalar ve onlarla aynı düzeyde duran kabuk ile GUI.

Şekildeki ayrım yazılım dünyasını ikiye böler. Çekirdek modunda çalışan yazılım işletim sistemidir. Kullanıcı modunda çalışan her şey, kabuk ve GUI bile, sıradan bir kullanıcı programıdır. Bu ayrımın neden bu kadar önemli olduğu, modları anlatan 1.2.2'de açıklanır.

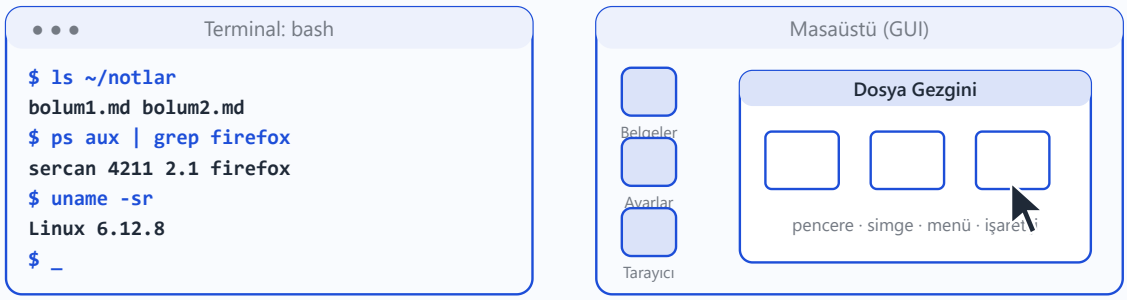
> **Sık yapılan hata**
>
> Masaüstünü, pencereleri ya da komut satırını işletim sisteminin kendisi sanmak. Bir Linux sunucusunda hiç GUI yoktur, ama işletim sistemi tam olarak çalışır. Aynı Linux çekirdeğinin üstünde GNOME, KDE Plasma ya da yalnız bir terminal bulunabilir. Görünen arayüz değişir, çekirdek aynı kalır.

1.1.4 Kabuk ve GUI

Kullanıcı bir işi yaptırmak istediğinde bunu ya bir komut yazarak ya da fareyle bir simgeye tıklayarak bildirir. Bu iki yolun her birini ayrı bir program karşılar.

Tanım 1.3

Kabuk, kullanıcının yazdığı komutları okuyan, yorumlayan ve ilgili programları başlatan komut satırı arayüzüdür (*command-line interface*, CLI). **GUI** ise pencere, simge, menü ve işaretçiden (*windows, icons, menus, pointer*: WIMP) oluşan görsel etkileşim ortamıdır.



Şekil 1.2: Solda bir terminal penceresinde bash kabuğu: `ls ~/notlar` iki dosya listeler, `ps aux | grep firefox` çalışan Firefox sürecini bulur, `uname -sr` çekirdek sürümünü (Linux 6.12.8) yazar. Sağda pencere, simge, menü ve işaretçiden oluşan bir masaüstü (GUI).

Yaygın kabuklar arasında Linux ve macOS'ta bash, zsh ve fish; Windows'ta PowerShell ve eski cmd.exe sayılabilir. GUI tarafında Windows Gezgini, macOS Finder ve Linux'taki GNOME ile KDE Plasma ortamları bilinir. Şekil 1.2, aynı bilgisayarda iki arayüzün yan yana çalışabildiğini gösterir.

Kabuk da GUI de çekirdeğin parçası değildir. İkisi de kullanıcı modunda çalışan programlardır ve çekirdekten bir hizmet istediklerinde, örneğin bir dosyayı açmak ya da yeni bir program başlatmak için, *sistem çağrısı* yapar. Bunun pratik sonuçları vardır. Kabuk değiştirilebilir: bash'ten zsh'e geçmek için çekirdeğe dokunmak gerekmez. Aynı anda birden fazla kabuk açılabilir. Bir kabuk çökerse yalnız o pencere kapanır, sistem çalışmaya devam eder. Kullanıcılar arayüzlerini kişiselleştirebilir, çünkü değiştirdikleri şey yalnız bir programdır.

1.1.5 Kabuk mu, GUI mi?

İki arayüz arasında bir yarış yoktur; her biri farklı işlerde öne çıkar. Kabuğun en büyük gücü **otomasyondur**. Komutlar bir betik (*script*) dosyasına yazılınca aynı iş binlerce kez, hiç tıklama yapmadan tekrarlanabilir. İkinci gücü, küçük araçları birleştirebilmesidir. Bir komutun çıktısı **boru hattıyla** (*pipeline*) bir sonrakinin girdisine bağlanır:

```
$ ps aux | grep ssh | wc -l
```

Bu satırda `ps aux` çalışan bütün süreçleri listeler. `grep ssh` bu listeden içinde “ssh” geçen satırları seçer. `wc -l` ise kalan satırları sayar. Üç basit aracın birleşimi “içinde ssh geçen kaç süreç satırı var?” sorusunu tek satırda yanıtlar. Kabuk az kaynak harcar ve SSH bağlantısıyla uzaktaki bir sunucuyu yö-

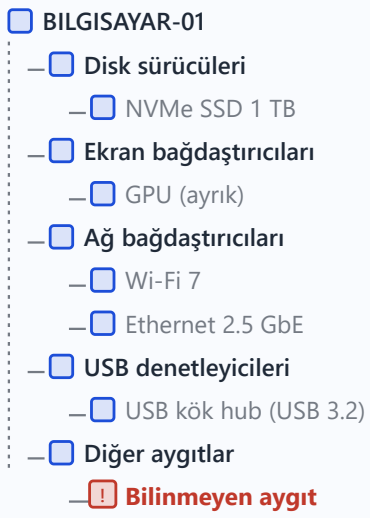
netmenin doğal yoludur. Belirli görevleri hızlı ve kesin biçimde yapar.

GUI'nin gücü ise **keşfedilebilirliktir**. Kullanıcı komut ezberlemek zorunda kalmaz; menülere bakarak neler yapılabileceğini görür. Fotoğraf, harita ya da tasarım gibi görsel içerikle çalışmak için doğal ortamdır. Geniş bir kullanıcı kitlesine ulaşır ve ekran okuyucu gibi erişilebilirlik araçlarıyla birlikte çalışır. Dokunmatik ve sesli arayüzler de bu ailenin üyeleridir.

Pratikte ikisi birbirini tamamlar: günlük işler için GUI, tekrarlı ve uzaktan yapılan işler için kabuk. Her ikisi de aynı çekirdek hizmetlerini kullanır. Çekirdek açısından bir dosyanın kabuktaki `cp` komutuyla mı, yoksa gezginde sürükle-bırak ile mi kopyalandığı fark etmez. Her iki durumda da çekirdeğe ulaşan şey, aynı türden sistem çağrılarıdır.

1.1.6 Aygıt Yöneticisi

İşletim sisteminin donanımı nasıl gördüğünü izlemenin somut bir yolu, Windows'taki **Aygıt Yöneticisi** aracıdır. Bu araç, bilgisayara bağlı donanım aygıtlarını ve onları süren yazılımları kategoriler hâlinde bir ağaçta listeler. Komut satırından `devmgmt.msc` yazılarak açılır; Microsoft Yönetim Konsolu (MMC) için yazılmış bir eklentidir.



Şekil 1.3: Aygıt Yöneticisi'nin ağaç görünümü. BILGISAYAR-01 altında disk sürücüleri (NVMe SSD 1 TB), ekran bağdaştırıcıları (ayrık GPU), ağ bağdaştırıcıları (Wi-Fi 7, Ethernet 2.5 GbE) ve USB denetleyicileri listelenir. "Diğer aygıtlar" altındaki bilinmeyen aygıt, sürücüsü bulunamadığı için uyarı simgesiyle işaretlenmiştir.

Aygıt Yöneticisi her aygıtın türünü, üreticisini, sürücü sürümünü ve kullandığı kaynakları (hangi kesme hattını, hangi bellek aralığını) gösterir. Buradan sürücü güncellenebilir ya da önceki sürüme geri alınabilir, bir aygıt devre dışı bırakılabilir. Sorunlu aygıtlar Şekil 1.3'teki gibi uyarı simgesiyle işaretlenir.

Burada dikkat edilmesi gereken nokta, aracın kendisinin işi yapmadığıdır. Aygıtları algılayan, onlara kaynak atayan ve uygun sürücüyü yükleyen, çekirdekteki Tak-Çalıştır (*Plug and Play*, PnP) yöneticisi ve aygıt sürücüleridir. Aygıt Yöneticisi yalnız onların tuttuğu bilgiye açılan bir penceredir. Linux'ta aynı bilgi `lspci` (PCI aygıtları), `lsusb` (USB aygıtları), `lsblk` (blok aygıtları) ve `lsmod` (yüklü çekirdek modülleri) komutlarıyla okunur. `/sys` dizinindeki dosyalar ve aygıt olaylarını yöneten `udev` hizmeti de aynı bilgiye açılır.

1.2 Konum ve modlar

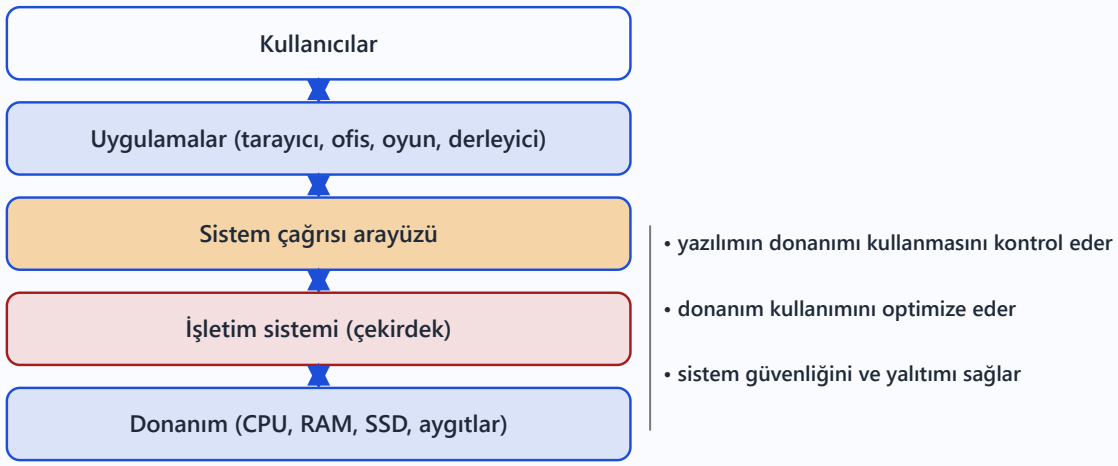
İşletim sisteminin "donanım ile uygulamalar arasında" durduğunu söylemek kolaydır. Asıl soru, bu aradalığın nasıl korunduğudur. Bir uygulama neden diske kendisi yazamaz? Kötü niyetli bir program işletim sistemini neden atlayamaz? Yanıt yazılımda değil, işlemcinin kendisindedir. Bu kesim önce işletim sisteminin konumunu, sonra bu konumu güvence altına alan donanım özelliğini, yani işlemci modlarını anlatır.

1.2.1 İşletim sisteminin konumu

Donanım fiziksel bileşenlerin, yani CPU'nun, RAM'in ve diskin adıdır. Yazılım ise bilgisayarın yapacağı işleri tarif eden koddur. İşletim sistemi bu ikisi arasındaki arayüzdür. Uygulamalar donanıma doğrudan erişmez; İS'nin sunduğu **sistem çağrıları** üzerinden erişir.

Tanım 1.4

Sistem çağrısı (*system call*), bir kullanıcı programının çekirdekten hizmet istemek için kullandığı denetimli giriş noktasıdır. Dosya açmak, okumak, yazmak, yeni süreç yaratmak ve bellek istemek gibi ayrıcalık gerektiren işlerin hepsi sistem çağrısıyla yapılır.



Şekil 1.4: İşletim sisteminin konumu. Kullanıcılar uygulamaları (tarayıcı, ofis, oyun, derleyici) kullanır; uygulamalar sistem çağrısı arayüzünden geçerek işletim sistemine ulaşır; işletim sistemi donanımı (CPU, RAM, SSD, ay-

gıtlar) denetler. Sağdaki notlar İS'nin üç görevini sıralar: yazılımın donanımı kullanmasını kontrol eder, donanım kullanımını en iyi duruma getirir, sistem güvenliğini ve yalıtımı sağlar.

Şekil 1.4 beş katmanı üst üste gösterir. Dikkat çeken katman, uygulamalar ile işletim sistemi arasındaki ince "sistem çağrısı arayüzü"dür. Bu arayüz, iki dünya arasındaki tek kapıdır. Uygulamanın İS'den isteyebileceği her şey bu kapının listesinde yazılıdır; listede olmayan bir işi uygulama yaptıramaz. Linux'ta bu listede birkaç yüz çağrı bulunur ve her birinin bir numarası vardır.

Şeklin sağındaki üç madde, işletim sisteminin bu konumdan yaptığı işleri özetler. *Kontrol*: hangi programın hangi donanıma ne zaman ve ne kadar erişeceğine İS karar verir. *En iyileme*: donanımın boşta kalmamasını, işlerin makul sürede bitmesini sağlar. *Güvenlik ve yalıtım*: programların birbirine ve sisteme zarar vermesini engeller.

1.2.2 Çekirdek modu ve kullanıcı modu

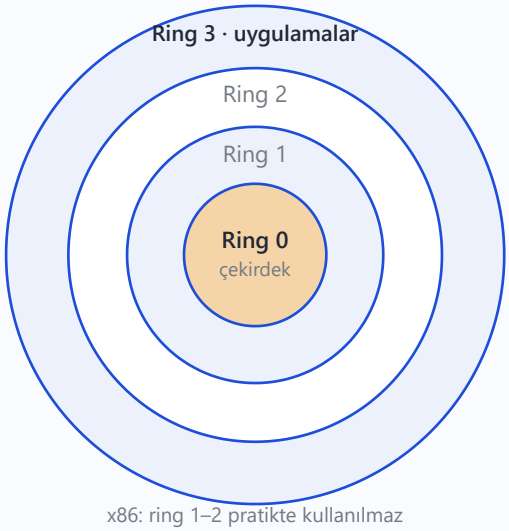
Bir programın donanıma doğrudan erişmesini yalnız "erişmemesi gerektiğini söyleyerek" engellemek mümkün değildir. Program ne de olsa işlemcinin üzerinde çalışır ve işlemci kendisine verilen her komutu yürütür. Engel, işlemcinin içine yerleştirilmiş bir donanım özelliğiyle kurulur.

Tanım 1.5

İşlemcide o anki ayrıcalık düzeyini gösteren bir **mod biti** (ya da birkaç bitlik ayrıcalık düzeyi) bulunur. **Çekirdek modunda** (*kernel mode*) bütün komutlar ve bütün bellek kullanılabilir. **Kullanıcı modunda** (*user mode*) ayrıcalıklı komutlar yasaktır; böyle bir komut denendiğinde işlemci komutu yürütmez, bir **istisna** üretir ve denetimi çekirdeğe verir.

Ayrıcalıklı komutlar, sistemin bütününü etkileyebilecek komutlardır. G/Ç aygıtlarına doğrudan erişmek, kesmeleri kapatmak, sayfa tablosunu (yani hangi sanal adresin hangi fiziksel belleğe karşılık geldiğini) değiştirmek bu gruptadır. Bunlardan biri kullanıcı modunda denendiğinde işlemci bir **tuzak** (*trap*) üretir: o anki komutu bırakır, çekirdek moduna geçer ve çekirdeğin önceden belirlediği işleyiciye atlar. Çekirdek de çoğunlukla bu programı sonlandırır. Sistemin geri kalanı bundan etkilenmez.

Farklı işlemci ailelerinde bu düzeyler farklı adlar taşır. x86 mimarisinde dört **halka** (*ring*) vardır: ring 0 en ayrıcalıklı, ring 3 en kısıtlı düzeydir. ARMv8 mimarisinde istisna düzeyleri EL0'dan EL3'e kadar sıralanır; uygulamalar EL0'da, çekirdek EL1'de çalışır. RISC-V'de ise U (kullanıcı), S (denetçi) ve M (makine) modları bulunur.



Şekil 1.5: x86 ayrıcalık halkaları. Merkezdeki ring 0'da çekirdek, en dıştaki ring 3'te uygulamalar çalışır. Ring 1 ve ring 2 pratikte kullanılmaz; bugünkü işletim sistemleri yalnız iki düzeyle yetinir.

Şekil 1.5, işletim sistemlerinin dört düzeyin yalnız ikisini kullandığını gösterir. Bunun nedeni taşınabilirliktir: Linux ve Windows pek çok işlemci ailesinde çalışır ve bunların ortak paydası iki düzeydir. Kabuk ve GUI, kullanıcı modundaki yazılımın en alt basamağında durur; onların bile ring 0'a erişimi yoktur.

Kullanıcı modundan çekirdek moduna geçmenin yalnız üç denetimli yolu vardır:

- **Sistem çağrısı**: program bilerek, özel bir komutla (x86-64'te `syscall`) çekirdekten hizmet ister.
- **Kesme** (*interrupt*): bir aygıt, örneğin zamanlayıcı ya da ağ kartı, işlemciye dışarıdan sinyal gönderir.
- **İstisna** (*exception*): yürütülen komutun kendisi bir sorun çıkarır; sıfıra bölme, geçersiz adrese erişim ya da kullanıcı modunda ayrıcalıklı komut gibi.

Üç yolun ortak özelliği, programın *nereye* atlanacağını seçememesidir. Atlanacak adresler, çekirdeğin açılışta doldurduğu bir tabloda yazılıdır. Program yalnız "çekirdeğe gitmek istiyorum" diyebilir; çekirdeğin hangi noktasından içeri gireceğini belirleyemez. Koruma tam olarak buna dayanır.

Sık yapılan hata

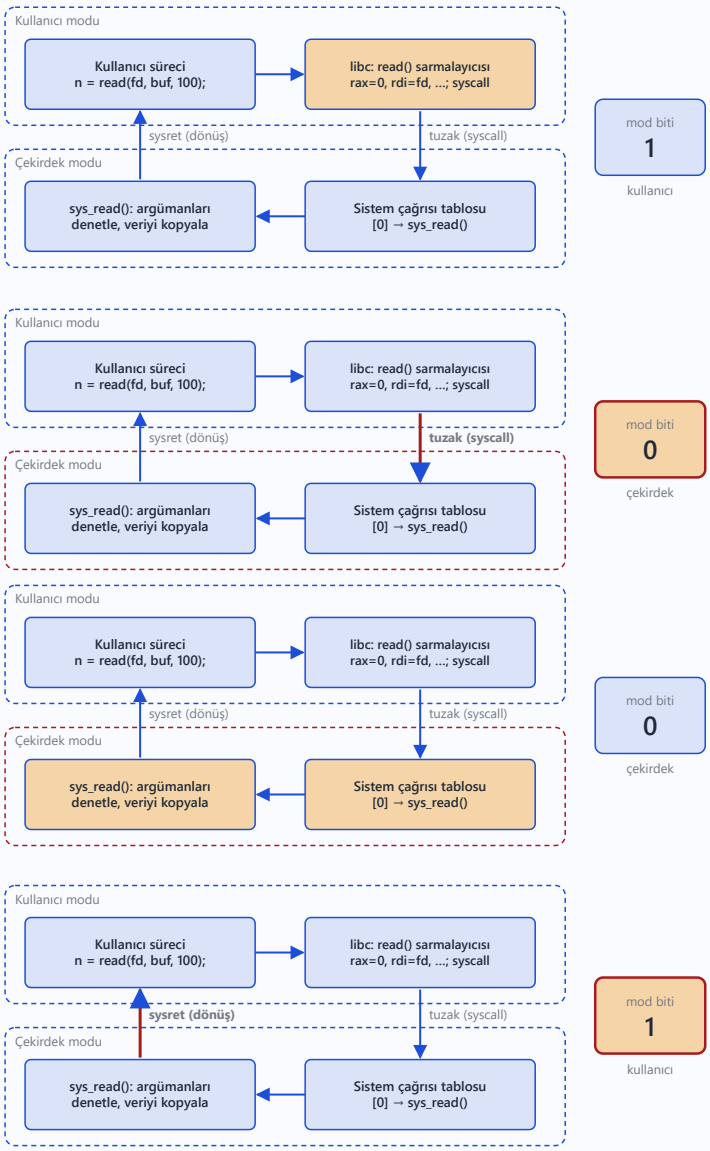
Çekirdek modunu "yönetici (root ya da Administrator) hesabıyla çalışmak" ile karıştırmak. Yönetici hesabıyla açılan bir program da kullanıcı modunda çalışır; yalnız çekirdek ona daha çok izin verir. Örneğin her dosyayı açabilir. Ama bu programın her işi yine sistem çağrısıyla yapılır ve programın kendisi asla ring 0'da çalışmaz.

1.2.3 Mod geçişi adım adım: read()

Mod geçişini somutlaştırmak için bir C programının dosyadan okuma yaptığı tek satırı izleyelim:

```
char buf[100];
int n = read(fd, buf, 100);   /* n: okunan bayt sayısı */
```

Programcı açısından bu, sıradan bir fonksiyon çağrısıdır. Gerçekte ise bu satır, kullanıcı modundan çekirdek moduna gidip geri dönen beş adımlık bir yolculuk başlatır. Aşağıdaki değerler x86-64 mimarisinde çalışan Linux içindir.



Şekil 1.6: `read()` sistem çağrısında mod geçişi (soldan sağa, yukarıdan aşağı). (a) libc sarmalayıcısı çağrı numarasını ve argümanları yazmaçlara koyar; mod biti 1, yani kullanıcı modu. (b) `syscall` komutu tuzak üretir; mod biti 0 olur. (c) Çekirdek, sistem çağrısı tablosunun 0. girdisinden `sys_read()`'i bulur, argümanları denetler ve veriyi kopyalar. (d) `sysret` mod bitini yeniden 1 yapar ve süreç kaldığı yerden devam eder.

1. **Başlangıç.** Süreç kullanıcı modunda çalışıyor (mod biti = 1). Bir dosyadan okumak ister, ama diske kendisi erişemez; disk denetleyicisinin yazmaçlarına dokunan komutlar ayrıcalıklıdır.
2. **Sarmalayıcı.** Programın çağırdığı `read()` aslında C kütüphanesindeki (libc) küçük bir *sarmalayıcı* fonksiyondur. Bu fonksiyon çağrı numarasını `rax` yazmacına koyar (x86-64 Linux'ta `read`'in numarası 0'dır), argümanları da sırayla `rdi`, `rsi`, `rdx` yazmaçlarına yerleştirir: `fd`, `buf`, 100. Ardından `syscall` komutunu yürütür.
3. **Tuzak.** `syscall` komutuyla donanım üç şey yapar: mod bitini 0 yapar, dönüş adresini kaydeder ve çekirdeğin sabit giriş noktasına atlar. Süreç nereye atlanacağını seçemez; giriş noktasını çekirdek açılışta bir yazmaca yazmıştır.
4. **Çekirdekte iş.** Çekirdek, `rax`'taki numarayla *sistem çağrısı tablosuna* bakar ve 0. girdide `sys_read()`'i bulur. Bu fonksiyon önce argümanları doğrular: `fd` bu sürecin açık bir dosyası mı, `buf` adresi gerçekten sürecin kendi belleğinde mi? Denetimler geçerse veriyi dosyadan sürecin tamponuna kopyalar.
5. **Dönüş.** Sonuç, yani okunan bayt sayısı, `rax`'a yazılır. `sysret` komutu mod bitini yeniden 1 yapar ve süreç `read()`'ten sonraki komuttan devam eder. Programın gözünde `n` değişkeni okunan bayt sayısını almıştır.

Dördüncü adımdaki doğrulama, korumanın kalbidir. Kötü niyetli bir program `buf` olarak çekirdeğin kendi belleğinin adresini verebilir. Çekirdek bu adresi denetlemeden kullansaydı, dosyadaki veriyi kendi yapılarının üstüne yazar ve sistemi ele geçirmeye açık hâle gelirdi. Bu yüzden çekirdek, kullanıcıdan gelen hiçbir değere güvenmez.

Örnek 1.1

Aşağıdaki işlemlerden hangileri kullanıcı modundan çekirdek moduna geçişe yol açar? (a) `int x = a + b;` (b) `read(fd, buf, 100);` (c) bir komutun geçersiz bir adrese erişmesi (d) `strlen(s);` (e) zamanlayıcının kesme üretmesi.

Çözüm. (a) Toplama, yazmaçlar ve ALU üzerinde yapılan sıradan bir işlemdir; mod değişmez. (b) Sistem çağrısıdır; libc sarmalayıcısı `syscall` komutuyla bilerek tuzak üretir, geçiş olur. (c) Komutun kendisinden doğan bir istisnadır; işlemci çekirdeğin işleyicisine atlar, geçiş olur. (d) `strlen`, bellekteki karakterleri sıfır baytına kadar sayan bir kütüphane fonksiyonudur; tamamen kullanıcı modunda kalır. (e) Dışarıdan gelen bir donanım kesmesidir; geçiş olur. Sonuç: (b), (c) ve (e) çekirdeğe geçer; bunlar sırasıyla sistem çağrısı, istisna ve kesmedir.

Sık yapılan hata

Her kütüphane fonksiyonunun çekirdeğe gittiğini sanmak. `strlen`, `memcpy`, `sqrt` gibi fonksiyonlar libc'de olsalar da yalnız kullanıcı belleği ve yazmaçlarla çalışır. Çekirdeğe yalnız `read`, `write`, `open`, `fork` gibi sistem çağrısı sarmalayıcıları gider. `printf` ise ikisinin karışımıdır: biçimlendirmeyi kullanıcı modunda yapar, sonunda `write` sistem çağrısıyla çıktıyı

gönderir.

1.2.4 Fiziksel aygıtlar, mikro mimari, makine dili

İşletim sisteminin altında da katmanlar vardır. Bunları ayırmak, İS'nin işini sınırlamak için yararlıdır.



Şekil 1.7: Altı katmanlı görünüm. Donanım tarafında alttan üste fiziksel aygıtlar, mikro mimari ve makine dili (ISA); yazılım tarafında işletim sistemi, sistem programları (derleyici, editör, kabuk) ve uygulama programları (banka, rezervasyon, tarayıcı).

- **Fiziksel aygıtlar**: yongalar, bellek modülleri, SSD ve G/Ç aygıtları; elektronik bileşenlerin kendisi.
- **Mikro mimari**: işlemcinin iç tasarımı. Veri yolu, yazmaçlar, aritmetik-mantık birimi (ALU), önbellekler ve denetim birimi bu düzeydedir.
- **Makine dili**: donanımın doğrudan yürüttüğü ikili komutlar. Bir işlemci ailesinin komut kümesine **komut kümesi mimarisi** (*instruction set architecture*, ISA) denir. x86-64, ARM64 ve RISC-V bugünün başlıca ISA'larıdır.

İşletim sisteminin bu katmanlardaki işi, sürücüler aracılığıyla üst katmanın isteklerini aygıtın anlayacağı düşük düzeyli komutlara çevirmektir. Bir sınırı da iyi bilmek gerekir: programı makine diline *derleyici* çevirir, işletim sistemi bunu yapmaz. Derleyici, Şekil 1.7'deki gibi sistem programları arasında yer alan sıradan bir kullanıcı programıdır. İşletim sistemi derlenmiş programı belleğe yükler, çalıştırır ve donanıma erişimine aracılık eder.

1.3 İşletim sisteminin rolleri

İşletim sistemine iki ayrı açıdan bakılabilir. Yukarıdan, yani uygulamanın gözünden bakıldığında İS, donanımın çirkin ayrıntılarını gizleyen ve yerine kullanışlı kavramlar koyan bir katmandır. Aşağıdan, yani donanımın gözünden bakıldığında ise aynı işlemciyi, aynı belleği ve aynı aygıtları isteyen pek çok

program arasında hakemlik yapan bir yöneticidir. Ders kitapları bu iki bakışı *genişletilmiş makine* ve *kaynak yöneticisi* diye adlandırır.

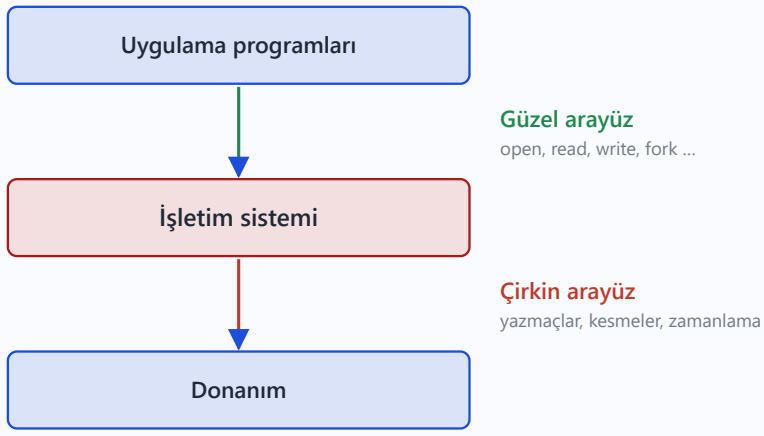
1.3.1 Genişletilmiş makine

Çıplak donanımın sunduğu arayüz, programcı için hem karmaşık hem de kırılgandır. Bir SSD'ye tek bir blok yazmak için önce NVMe komut kuyrukları hazırlanır ve mantıksal blok adresi (*logical block address*, LBA) hesaplanır. Komut kuyruğa konur ve denetleyici uyarılır. Ardından tamamlanma kuyruğu izlenir ve olası hata kodları yorumlanır. Uygulamanın bunların hiçbirini bilmesi gerekmez; yalnız şunu der:

```
write(fd, "Merhaba", 7);
```

Aradaki farkı kapatan işletim sistemidir. Tanenbaum bu rolü özlü bir cümleyle anlatır: işletim sistemi *çirkin donanımı güzel soyutlamalara* dönüştürür.

> **Tanım 1.6**
>
> **Soyutlama** (*abstraction*), bir şeyin gereksiz ayrıntılarını gizleyip yalnız işe yarayan özelliklerini gösteren basitleştirilmiş modeldir. İşletim sistemi, donanım üzerine soyutlamalar kurduğu için **genişletilmiş makine** (*extended machine*) olarak görülür: programcıya çıplak donanımdan daha kullanışlı, "genişletilmiş" bir bilgisayar sunar.



Şekil 1.8: Genişletilmiş makine. Uygulama programları işletim sisteminin sunduğu güzel arayüzü (`open`, `read`, `write`, `fork` ...) görür. İşletim sistemi ise donanımın çirkin arayüzüyle (yazmaçlar, kesmeler, zamanlama) uğraşır.

Genişletilmiş makine rolünün üç sonucu vardır. Birincisi *kolaylıktır*: donanımın yetenekleri kullanılabilir hâle gelir, karmaşık ayrıntıları gizlenir. İkincisi *korumadır*: uygulamalar donanıma doğrudan erişemediği için birbirinin işini bozamaz. Üçüncüsü *taşınabilirliktir*: SSD değişse de, hatta disk yerine ağ üzerinden bir depolama kullanılsa da `write` çağrısı aynı kalır.

1.3.2 Soyutlamalar ve roller

İşletim sisteminin kurduğu soyutlamaların her biri belirli bir donanım parçasına karşılık gelir. Tablo 1.1 başlıca eşleşmeleri verir.

Donanım	İS soyutlaması
İşlemci (CPU)	Süreç ve iş parçacığı
Bellek (RAM)	Adres alanı (sanal bellek)
Depolama (SSD/disk)	Dosya ve dizin
Ağ kartı	Soket
Ekran, klavye	Pencere, terminal

Tablo 1.1: Donanım bileşenleri ve işletim sisteminin onlar için kurduğu soyutlamalar.

Tablodaki her soyutlama, donanımın *sınırlı ve paylaşılan* bir kaynağını her programa "kendine ait" gibi gösterir. Tek bir işlemci vardır, ama her süreç işlemciyi yalnız kendisi kullanıyormuş gibi yazılır. Tek bir fiziksel bellek vardır, ama her süreç 0 adresinden başlayan kendi adres alanını görür. Diskte milyonlarca blok vardır, ama program yalnız bir dosya adı bilir. Ağ kartı tektir, ama her bağlantının kendi soketi vardır.

Bu soyutlamaların çevresinde dört ayrı rol ayırt edilir. **Donanım tasarımcısı** ISA'yı, aygıtları ve aygıtların arayüzlerini tasarlar. **Çekirdek geliştiricisi** soyutlamaları bu donanımın üzerinde gerçekler; süreç, sanal bellek ve dosya sistemi onun ürünüdür. **Uygulama geliştiricisi** soyutlamaları sistem çağrıları ve uygulama programlama arayüzleri (API) üzerinden kullanır. **Son kullanıcı** ise yalnız uygulamaları ve arayüzü görür. Bu dersin büyük kısmı, ikinci rolün işini anlamaya ayrılmıştır.

1.3.3 Bir write() çağrısının yolculuğu

Soyutlamanın neyi gizlediğini görmek için `write(fd, "Merhaba", 7)` çağrısını uygulamadan SSD'ye kadar izleyelim. Bu çağrı, programın açtığı dosyaya 7 baytlık "Merhaba" dizgisini yazmak ister.



Şekil 1.9: Bir `write()` çağrısının katmanları. Kullanıcı modunda uygulama ve C kütüphanesi; çekirdek modunda sistem çağrısı arayüzü, sanal dosya sistemi (VFS), dosya sistemi (ext4, NTFS, APFS), sayfa önbelleği ve blok katmanı, aygıt sürücüsü (NVMe); en altta SSD denetleyicisi. Son adımda sürücü ile SSD denetleyicisi vurgulanmıştır.

1. **Uygulama** yalnız üç şey verir: bir dosya tanımlayıcısı (`fd`), bir tampon ve bir uzunluk. Diskin türünü, hatta verinin yerel diske mi ağa mı gideceğini bilmez.
2. **libc sarmalayıcısı** argümanları yazmaçlara koyar ve `syscall` ile çekirdeğe geçer; tıpkı 1.2.3'teki `read()` gibi.
3. **Sanal dosya sistemi** (*virtual file system*, VFS), `fd`'yi sürecin açık dosya tablosundan bir dosya nesnesine ve oradan dosyanın i-node'una çözer. VFS, bütün dosya sistemleri için ortak olan katmandır.
4. **Dosya sistemi** (ext4, NTFS, APFS …), dosyadaki konumu, yani bağıl konumu, diskteki blok adresine çevirir. Dosya büyüyorsa yeni blok ayırır.
5. **Sayfa önbelleği**: veri çoğunlukla önce bellekteki sayfa önbelleğine yazılır ve `write()` hemen döner. Sayfa "kirli" olarak işaretlenir; diske yazma daha sonra yapılır. Verinin hemen kalıcı olması isteniyorsa program `fsync()` çağırır.
6. **Aygıt sürücüsü** bir NVMe yazma komutu hazırlar. SSD denetleyicisi veriyi doğrudan bellek erişimiyle (DMA) bellekten alır, flash sayfasına yazar ve iş bitince bir kesme üretir.

Yedi baytlık bir yazma için altı katman çalışmıştır. Bunların yalnız ilki uygulamanın kodudur. Beşinci adımdaki ayrıntı da önemlidir: `write()` döndüğünde veri büyük olasılıkla henüz diskte değildir. Elektrik o anda kesilirse son yazılanlar kaybolabilir. Bu yüzden veritabanları gibi güvenilirlik isteyen programlar `fsync()`'i bilinçli olarak kullanır.

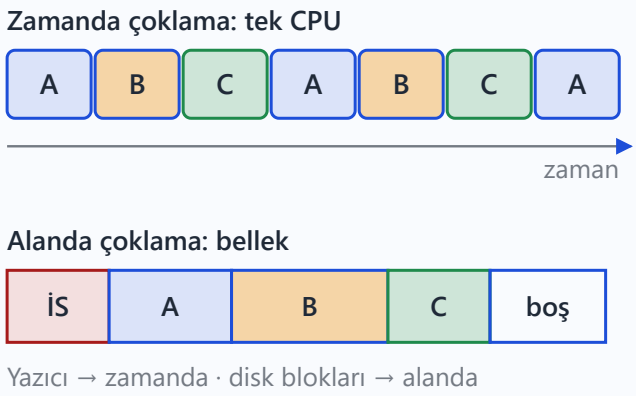
1.3.4 Kaynak yöneticisi

Aşağıdan bakıldığında işletim sisteminin görevi farklı görünür. Birden çok program aynı anda çalışmak ister; hepsinin işlemciye, belleğe, diske ve ağa ihtiyacı vardır. İşletim sistemi bu kaynakları programlar arasında dağıtır ve her programın payını diğerlerine karşı korur.

Tanım 1.7

Kaynak yöneticisi olarak işletim sistemi, CPU, bellek ve G/Ç aygıtları gibi kaynakları birden çok programa dağıtır ve korur. Hedefi, kaynakları *verimli* kullanmak ve programlar arasında *adil* paylaştırmaktır.

Kaynaklar iki biçimde paylaştırılır. **Zamanda paylaştırmada** (*time multiplexing*) programlar kaynağı sırayla kullanır: önce biri, sonra öteki. **Alanda paylaştırmada** (*space multiplexing*) kaynak parçalara bölünür ve her program kendi parçasını aynı anda kullanır.



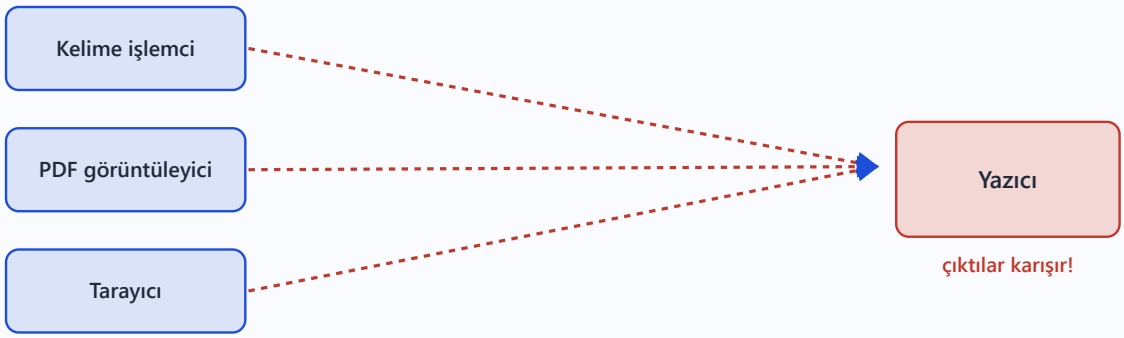
Şekil 1.10: Üstte zamanda paylaştırma: tek CPU'yu A, B, C süreçleri sırayla kullanır. Altta alanda paylaştırma: bellek İS, A, B, C ve boş bölge olarak bölünmüştür. Yazıcı zamanda, disk blokları alanda paylaştırılır.

Hangi kaynağın hangi biçimde paylaştırılacağı, kaynağın bölünüp bölünemeyeceğine bağlıdır. İşlemci aynı anda yalnız bir komut akışını yürütür; bölünemez, zamanda paylaştırılır. Bellek ise bölünebilir: aynı anda birkaç program bellekte durur ve her biri kendi bölgesini kullanır. Disk alanı da bölünür, her dosyaya ayrı bloklar verilir; ama diske giden okuma ve yazma istekleri sırayla işlendiği için disk ayrıca zamanda da paylaştırılır. Yazıcı ise bölünemez; yarım sayfasını bir programa, yarısını ötekine vermenin anlamı yoktur.

Kaynak yöneticisi bakışı, işletim sistemi tasarımının temel sorularını doğurur. Birçok program aynı anda yazdırmak isterse ne olur? Her sürecin ne kadar kaynak kullandığı nasıl ölçülür? Kaynaklar çoğaldığında, örneğin çekirdek sayısı arttığında, adalet ve verimlilik nasıl sağlanır? Sonraki alt kesimler bu soruları sırayla ele alır.

1.3.5 Aynı anda yazdırma: biriktirme

Bir ofiste üç kişinin neredeyse aynı anda belge yazdırdığını düşünelim. Biri kelime işlemciden bir rapor, biri PDF görüntüleyiciden bir fotoğraf, biri de tarayıcıdan bir not gönderir. Yazıcı tektir. Üç program da yazıcıya doğrudan yazabilseydi, sayfalar satır satır birbirine karışırdı: raporun ilk satırının altında fotoğrafın bir parçası, onun altında notun ilk cümlesi basılırdı.

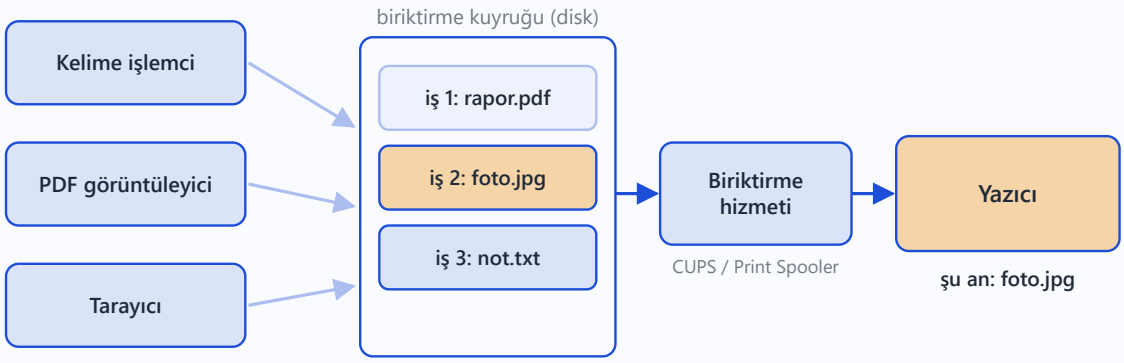


Şekil 1.11: Biriktirme olmadan: kelime işlemci, PDF görüntüleyici ve tarayıcı aynı anda tek yazıcıya yazmaya çalışır ve çıktılar karışır.

İşletim sisteminin çözümü, yazıcıyı tek bir sürecin tekeline vermektir.

Tanım 1.8

Biriktirme (*spooling*), yavaş ve paylaşılamayan bir aygıta giden işlerin önce diskteki bir kuyruğa yazılması, sonra tek bir hizmet sürecinin bu işleri aygıta sırayla göndermesidir.



Şekil 1.12: Biriktirmeyle yazdırma. Üç program işlerini diskteki biriktirme kuyruğuna yazar (iş 1: rapor.pdf, iş 2: foto.jpg, iş 3: not.txt). Yazıcıyı yalnız biriktirme hizmeti (Linux'ta CUPS, Windows'ta Print Spooler) kullanır. İş 1 bitmiş, şu an iş 2 (foto.jpg) basılıyor.

Şekil 1.12 işleyişi gösterir. Programlar işlerini yazıcıya değil, işletim sisteminin diskteki biriktirme kuyruğuna yazar ve hemen kendi işlerine döner. Yazıcıyı kullanan tek süreç biriktirme hizmetidir: Linux'ta CUPS, Windows'ta Print Spooler. Karışıklık bu tekellikle önlenir, çünkü aynı anda yazıcıya yalnız bir iş gider. Hizmet işleri genellikle geliş sırasıyla (*first-come, first-served*, FCFS) gönderir. Yönetici bir işe öncelik verebilir ya da bir işi duraklatabilir.

Biriktirmenin ikinci kazancı zamandır. Yazıcı, işlemciye göre son derece yavaştır; bir sayfanın basılması saniyeler sürer. Programlar yazıcının bitirmesini beklemek zorunda kalsaydı, kullanıcı her yazdırmada dakikalarca donmuş bir pencereye bakardı. Biriktirme sayesinde yavaş yazıcı çalışırken programlar ve işlemci başka işler yapar. G/Ç ile hesaplama aynı anda ilerler, yani *örtüşür*. Bu fikir işletim sistemlerinin tarihinde tekrar tekrar karşımıza çıkar.

1.3.6 Kod izleme: biriktirme kuyruğu

Biriktirmenin özü küçük bir C koduyla modellenebilir. Kuyruk 4 yuvalı bir dizidir; programlar `ekle()`, biriktirme hizmeti ise `biriktirme_hizmeti()` fonksiyonunu çağırır.

```
char *kuyruk[4];
int giris = 0, cikis = 0, adet = 0;
void ekle(char *dosya) {   /* programlar */
  kuyruk[giris] = dosya;
  giris = (giris + 1) % 4;
  adet++;
}
void biriktirme_hizmeti(void) {
  while (adet > 0) {
    yazici_gonder(kuyruk[cikis]);
    kuyruk[cikis] = NULL;   /* yuva boşalır */
    cikis = (cikis + 1) % 4;
    adet--;
  }
}
```

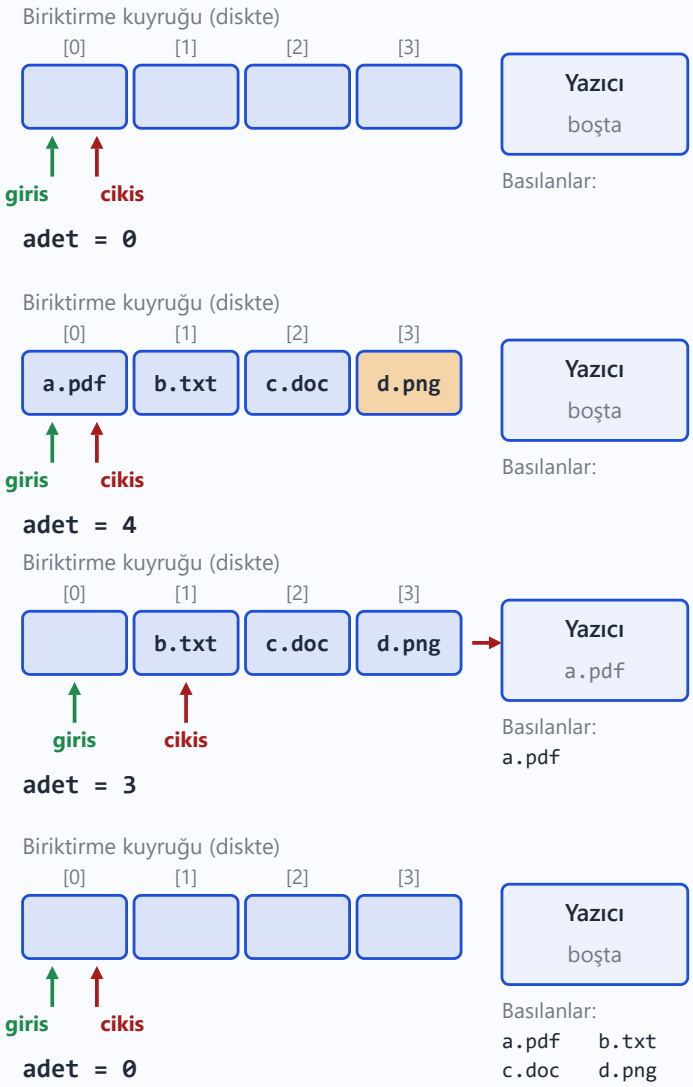
Kod satır satır şöyle okunur. İlk satır, dosya adlarını tutacak 4 elemanlı bir işaretçi dizisi tanımlar. İkinci satırdaki üç sayaç kuyruğun durumunu tutar: `giris` sıradaki boş yuvanın indisidir, `cikis` basılacak en eski işin indisidir, `adet` ise kuyruktaki iş sayısıdır. `ekle()` fonksiyonu dosya adını `giris` yuvasına yazar, `giris`'i bir ilerletir ve `adet`'i artırır. İlerletmede kullanılan `% 4` işlemi kuyruğu **dairesel** yapar: 3. yuvadan sonra 0. yuvaya dönülür. `biriktirme_hizmeti()` ise kuyrukta iş kaldıkça en eski işi yazıcıya gönderir, gönderilen yuvaya `NULL` yazarak onu boşaltır, `cikis`'i aynı biçimde ilerletir ve `adet`'i azaltır.

Dört programın sırayla a.pdf, b.txt, c.doc ve d.png gönderdiğini, ardından biriktirme hizmetinin çalıştığını varsayalım. Tablo 1.2 her çağrıdan sonra değişkenlerin ve kuyruğun durumunu verir. Kuyruk sütununda "·" boş yuvayı gösterir.

Adım	İşlem	`giris`	`cikis`	`adet`	Kuyruk [0] [1] [2] [3]	Basılanlar
0	başlangıç	0	0	0	· · · ·	—
1	`ekle("a.pdf")`	1	0	1	a.pdf · · ·	—
2	`ekle("b.txt")`	2	0	2	a.pdf b.txt · ·	—
3	`ekle("c.doc")`	3	0	3	a.pdf b.txt c.doc ·	—

Adım	İşlem	`giris`	`cikis`	`adet`	Kuyruk [0] [1] [2] [3]	Basılanlar
4	`ekle("d.png")`	0	0	4	a.pdf b.txt c.doc d.png	—
5	gönder (1. tur)	0	1	3	· b.txt c.doc d.png	a.pdf
6	gönder (2. tur)	0	2	2	· · c.doc d.png	a.pdf, b.txt
7	gönder (3. tur)	0	3	1	· · · d.png	a.pdf, b.txt, c.doc
8	gönder (4. tur)	0	0	0	· · · ·	a.pdf, b.txt, c.doc, d.png

Tablo 1.2: Biriktirme kuyruğunun izlemesi.



Şekil 1.13: İzlemenin dört anı. (a) Kuyruk boş, `giris` ve `cikis` 0. yuvayı gösterir. (b) Dört `ekle`'den sonra kuyruk dolu, `adet = 4`; `giris` başa sarmıştır. (c) a.pdf yazıcıya gitmiş, `cikis = 1`, `adet = 3`. (d) Dört iş basılmış, kuyruk yeniden boş, yazıcı boşta.

Tablonun iki satırı özel dikkat ister. Dördüncü adımda `giris = (3 + 1) % 4 = 0` olur: indis dizinin başına *sarar*. Bu anda `giris` ile `cikis` eşittir, ama kuyruk boş değil doludur. İki indisin eşitliği tek başına boş ile doluyu ayırt edemez; `adet` sayacı tam bu yüzden tutulur. Sekizinci adımda `cikis` de

0'a sarar ve `adet = 0` olduğu için döngü biter. İşler geliş sırasıyla, yani FCFS düzeninde basılmıştır. Yazıcıyı yalnız hizmet kullandığı için çıktılar karışmamıştır.

Yuvayı boşaltan `kuyruk[cikis] = NULL;` satırı, kuyruğun doğru çalışması için zorunlu değildir: hangi yuvaların dolu olduğunu `cikis` ve `adet` sayaçları zaten belirler. Satır, izlemeyi okunur kılar ve basılmış bir işin adının dizide unutulup kalmasını önler. Tablonun beşinci adımından itibaren boşalan yuvaların "·" görünmesi bu satırın sonucudur.

Maliyet. `ekle()` üç atama yapar; kuyruktaki iş sayısından bağımsız olarak sabit sürede, yani *O(1)*'de biter. `biriktirme_hizmeti()` kuyrukta *n* iş varsa döngüyü *n* kez döner: *O(n)*. Kuyruğun kendisi sabit, 4 yuvalık bir alan kaplar.

> **Sık yapılan hata**
>
> Bu kodu gerçek bir sistemde olduğu gibi kullanmak. İki eksik vardır. Birincisi, `ekle()` kuyruğun dolu olup olmadığını denetlemez; beşinci bir iş gelirse en eski işin üzerine yazar. İkincisi, programlar ve hizmet ayrı süreçlerdir ve aynı anda çalışabilir. `adet++` ile `adet--` tek bir komut değil, "oku, değiştir, yaz" dizisidir; araya öteki taraf girerse bir güncelleme kaybolur. Bu duruma yarış durumu denir. Gerçek biriktirme kuyrukları bu yüzden bir kilitle korunur.

1.3.7 Kaynak kullanımını ölçmek

Kaynakları adil dağıtmak için önce kimin ne kadar kullandığını bilmek gerekir. İşletim sistemi bu bilgiyi her süreç için ayrı ayrı tutar. Tablo 1.3 üç temel kaynak için ölçütleri ve onları gösteren araçları özetler.

Kaynak	Ölçüt	Linux	Windows
CPU	Kullanıcı ve çekirdek modunda geçen CPU zamanı, kullanım yüzdesi	`/proc/[pid]/stat`, `top`, `time`	Görev Yöneticisi, Performans İzleyicisi
Bellek	Yerleşik küme (RSS), sanal boyut, çalışma kümesi; ayırma ve serbest bırakma örüntüsü	`/proc/[pid]/status`, `smem`	"Çalışma kümesi" sütunu
G/Ç	Okunan ve yazılan bayt, sistem çağrısı ve G/Ç isteği sayısı	`/proc/[pid]/io`, `iotop`, `strace -c`	Kaynak İzleyicisi

Tablo 1.3: Kaynak kullanımı ölçütleri ve araçları.

CPU zamanının kullanıcı ve çekirdek modu diye ikiye ayrılması anlamlıdır. Bir program hesap yaparken zamanını kullanıcı modunda geçirir. Çok sayıda küçük dosya okuyan bir program ise zamanının büyük kısmını çekirdekte, sistem çağrılarının içinde geçirir. Linux'ta `time` komutu bu iki değeri `user` ve `sys` adıyla ayrı ayrı yazdırır.

Bellek için en sık bakılan değer **yerleşik küme boyutudur** (*resident set size*, RSS): sürecin o anda fiziksel bellekte bulunan sayfalarının toplamı. Sanal boyut ise sürecin ayırdığı bütün adres alanıdır ve RSS'ten çok büyük olabilir, çünkü ayrılan her sayfa gerçekten belleğe getirilmez. Linux'ta bu değerler

`/proc` dizinindeki dosyalardan okunur:

```
$ grep -E 'VmRSS|Threads' /proc/4211/status
VmRSS:    412360 kB
Threads:  57
```

Burada 4211 numaralı süreç (önceki terminal örneğindeki Firefox süreci) için iki satır seçilmiştir. `VmRSS` sürecin 412.360 kB'lık kısmının fiziksel bellekte olduğunu, `Threads` ise sürecin 57 iş parçacığı içerdiğini söyler. `/proc` diskte gerçekten var olan bir dizin değildir. Çekirdek bu dosyaların içeriğini her okumada o anki değerlerden üretir.

Bu ölçümlerin kaynağı çekirdektir. Çekirdek her bağlam değiştirmede, her sistem çağrısında ve her sayfa hatasında süreç başına tuttuğu sayaçları günceller. Görev Yöneticisi ya da `top` gibi araçlar bu sayaçları yalnız okur ve gösterir.

1.3.8 Adalet ve verimlilik

Kaynakların çoğalması, yani çekirdek sayısının ve bellek miktarının artması, başarımı ve tepki süresini iyileştirir. Ama aynı zamanda kaynakları adil ve verimli paylaştırmayı zorlaştırır. Dört çekirdekli bir makinede hangi sürecin hangi çekirdekte çalışacağı, tek çekirdekli bir makinedeki "sıradaki kim?" sorusundan çok daha karmaşıktır. İşletim sistemleri bu sorunu dört araçla ele alır.

- **Çizelgeleme**: hangi süreç ne zaman ve hangi çekirdekte çalışacak? Bu kararı veren çekirdek bileşenine **çizelgeleyici** (*scheduler*) denir. Linux'ta uzun yıllar CFS (*Completely Fair Scheduler*) kullanıldı; Linux 6.6 sürümüyle (2023) yerini EEVDF çizelgeleyicisi aldı.
- **Önceliklendirme**: bazı süreçlere öteki süreçlerden daha çok pay verilir. Linux'ta `nice` değerleri, Windows'ta öncelik sınıfları ve her iki sistemde de gerçek zamanlı sınıflar bu iş içindir.
- **Kotalar ve sınırlar**: bir sürecin ya da süreç grubunun kullanabileceği kaynağa üst sınır konur. Linux'ta cgroups v2 (konteynerlerin temeli), `ulimit` ve disk kotaları; Windows'ta Job nesneleri bu amaçla kullanılır.
- **Ödünleşim**: verim, tepki süresi ve adalet aynı anda en iyi duruma getirilemez. Biri iyileşirken öteki kötüleşebilir.

Son madde üç kavrama dayanır. **Verim** (*throughput*), birim zamanda biten iş sayısıdır. **Tepki süresi** (*response time*), bir isteğin yapılmasından sistemin ilk yanıtına kadar geçen süredir. **Adalet** ise her sürecin hak ettiği payı alıp almadığıdır. Örneğin süreçler arasında sık geçiş yapmak tepki süresini kısaltır, çünkü hiçbir süreç uzun süre beklemez. Ama her geçişin bir maliyeti vardır; geçiş sayısı arttıkça işlemcinin asıl işe ayırdığı zaman, dolayısıyla verim azalır.

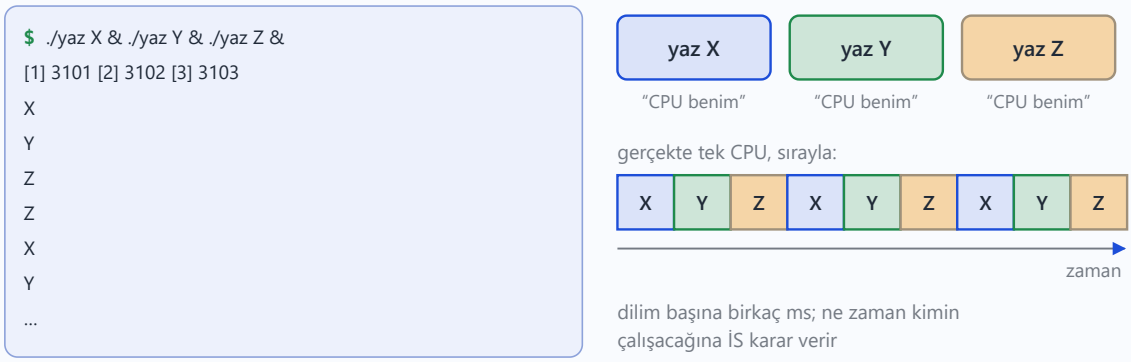
Doğru denge iş yüküne bağlıdır. Bir sunucuda öncelik verimdir: saatte işlenen istek sayısı en çok olmalıdır. Bir masaüstü bilgisayarda öncelik tepki süresidir: fare tıklamasına anında yanıt gelmelidir. Gerçek zamanlı bir sistemde ise öncelik, işlerin son tarihlerinden önce bitmesidir.

1.3.9 Üç büyük fikir: dört küçük deney

Arpaci-Dusseau'ların *Operating Systems: Three Easy Pieces* (OSTEP) kitabı, işletim sistemi konularını üç büyük fikir çevresinde toplar: **sanallaştırma**, **eş zamanlılık** ve **kalıcılık**. Her fikir, Linux üzerinde çalıştırılan küçük bir deneyle somutlaştırılabilir. Sanallaştırma işlemci ve bellek için ayrı ayrı gösterildiğinden deney sayısı dörttür.

> **Tanım 1.9**
>
> **Sanallaştırma** (*virtualization*), tek bir fiziksel kaynağı her programa kendine ait, ayrı bir kaynak varmış gibi gösterme tekniğidir. İşletim sistemi işlemciyi ve belleği sanallaştırır.

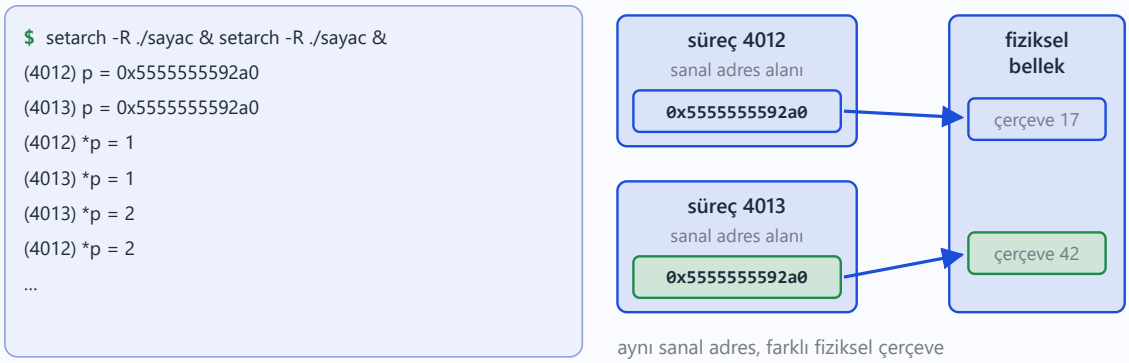
Deney 1: işlemciyi sanallaştırma. `yaz` adlı program sonsuz bir döngüde kendisine verilen harfi ekrana yazar. Tek çekirdekli bir makinede üç kopyası arka planda başlatılır:



Şekil 1.14: İşlemciyi sanallaştırma. `./yaz X & ./yaz Y & ./yaz Z &` komutu 3101, 3102, 3103 numaralı üç süreç başlatır; çıktıda X, Y, Z harfleri karışık sırayla görünür. Her program "CPU benim" sanır; gerçekte tek CPU sırayla X, Y, Z süreçlerine verilir. Her dilim birkaç milisaniyedir ve kimin ne zaman çalışacağına İS karar verir.

Üç program da kendi işini kesintisiz yapıyormuş gibi görünür; çıktıda üç harf karışık sırayla belirir. Oysa makinede tek çekirdek vardır. İşletim sistemi işlemciyi birkaç milisaniyelik dilimler hâlinde sırayla üç sürece verir. Geçişler insan gözünün fark edemeyeceği kadar sık olduğu için her biri kendi işlemcisi varmış gibi davranır. Burada iki ayrı soru vardır. Sıradaki dilimin kime verileceği bir *politika* sorusudur ve çizelgeleme algoritmasıyla yanıtlanır. Geçişin nasıl yapılacağı ise bir *mekanizma* sorusudur ve bağlam değiştirmeyle yanıtlanır. İki kavramın ayrımı 1.8.1'de işlenir.

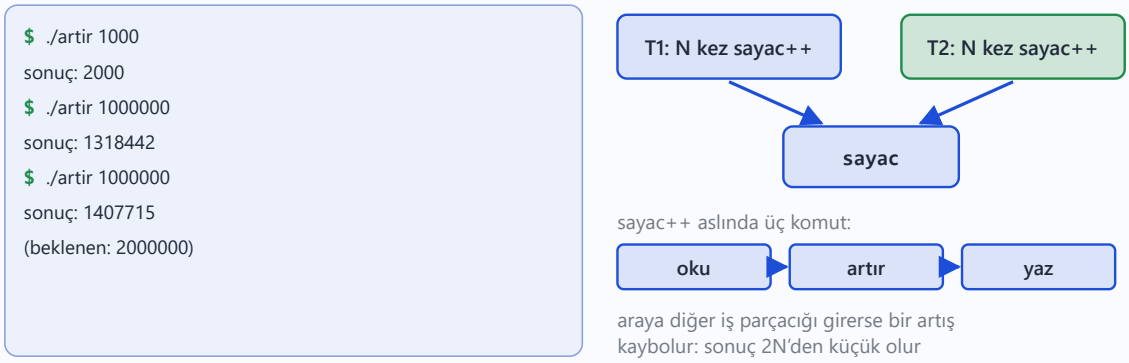
Deney 2: belleği sanallaştırma. `sayac` adlı program bir tam sayı için bellek ayırır, bu belleğin adresini yazdırır, sonra döngüde değeri artırıp yazdırır. İki kopyası aynı anda çalıştırılır:



Şekil 1.15: Belleği sanallaştırma. 4012 ve 4013 numaralı süreçler aynı adresi (`0x5555555592a0`) yazdırır, ama her birinin değeri birbirinden bağımsız 1, 2, ... diye artar. İki süreç aynı sanal adresi kullanır; İS bu adresleri farklı fiziksel çerçevelere (17 ve 42) eşler.

İki süreç de aynı adresi yazdırır. Bellek gerçekten paylaşılsaydı, iki süreç aynı tam sayıyı artırır ve değerler 1, 2, 3, 4 diye ortak ilerlerdi. Oysa her süreçte değer 1'den başlayıp bağımsız artar. Açıklama şudur: programların gördüğü adres **sanal adrestir**. Her süreç kendi adres alanını görür ve işletim sistemi, donanımın yardımıyla, aynı sanal adresi her süreç için farklı bir fiziksel bellek çerçevesine eşler. Deneyde `setarch -R` komutu kullanılmıştır; bu komut adres alanı yerleşiminin rastgeleleştirilmesini (*address space layout randomization*, ASLR) kapatır. ASLR açık kalsaydı adresler her çalıştırmada değişir ve aynılık görülmezdi.

Deney 3: eş zamanlılık. `artir` programı iki iş parçacığı başlatır. Her iş parçacığı ortak bir `sayac` değişkenini *N* kez artırır. Beklenen sonuç *2N*'dir.



Şekil 1.16: Eş zamanlılık. `./artir 1000` doğru sonucu (2000) verir. `./artir 1000000` ise 2.000.000 yerine bir çalıştırmada 1.318.442, ötekinde 1.407.715 verir. T1 ve T2 aynı `sayac` 'ı artırır; `sayac++` aslında oku, artır, yaz biçiminde üç komuttur ve araya öteki iş parçacığı girerse bir artış kaybolur.

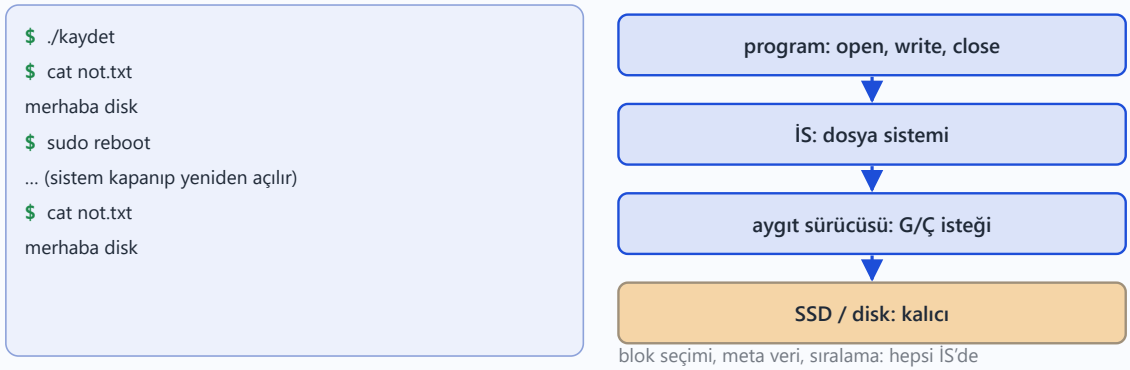
N = 1000 için sonuç doğru görünür, çünkü iş parçacıkları o kadar kısa sürer ki birbirleriyle pek karşılaşmazlar. *N* bir milyona çıkınca sonuç *2N*'den küçük olur ve her çalıştırmada başka bir değer çıkar. Nedeni, `sayac++` ifadesinin *atomik* olmamasıdır: makine düzeyinde değeri bellekten yazmaca

okuma, yazmaçta artırma ve geri yazma diye üç ayrı komuttur. İki iş parçacığı aynı eski değeri okur, ikisi de bir artırır ve ikisi de aynı yeni değeri yazarsa iki artıştan biri kaybolur. Paylaşılan veriye eş zamanlı erişimin doğru yönetilmesi, işletim sistemlerinin en zor konularından biridir.

Tanım 1.10

Eş zamanlılık (*concurrency*), birden çok işin zaman içinde iç içe ilerlemesidir. İşler tek bir işlemcide sırayla dilimler alarak ilerleyebilir; birden çok çekirdekte gerçekten aynı anda çalışmaları ise **paralellik** (*parallelism*) diye adlandırılır.

Deney 4: kalıcılık. `kaydet` programı `not.txt` dosyasına "merhaba disk" yazar. Bilgisayar yeniden başlatıldıktan sonra dosya yine okunur:



Şekil 1.17: Kalıcılık. `./kaydet` çalıştırılır, `cat not.txt` "merhaba disk" yazar; `sudo reboot` ile sistem yeniden açıldıktan sonra da aynı içerik okunur. Program yalnız `open`, `write`, `close` çağırır; dosya sistemi, aygıt sürücüsü ve G/Ç isteği İS'nin içindedir; veri SSD ya da diskte kalıcıdır.

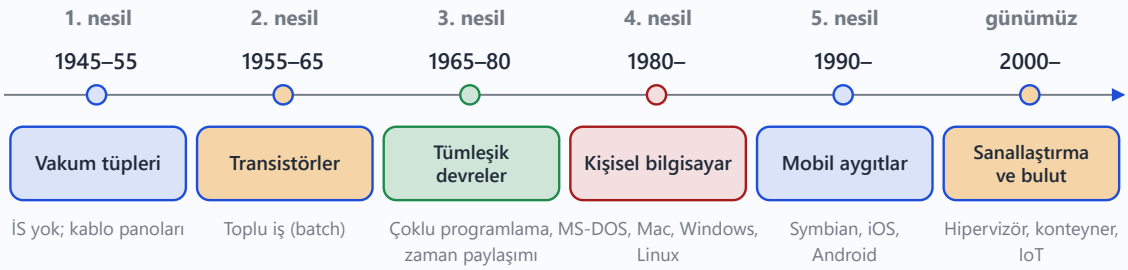
Ana bellek (DRAM) *uçucudur*: güç kesilince içeriği silinir. Verinin yeniden başlatmadan sonra da kalması isteniyorsa, dosya sistemi aracılığıyla kalıcı bir aygıta yazılır. Programın yaptığı tek şey `open`, `write` ve `close` çağırmaktır. Verinin hangi bloğa yazılacağı, dosyanın adı, boyutu ve izinleri gibi meta verilerin nerede tutulacağı ve aygıtla nasıl konuşulacağı işletim sisteminin işidir.

1.4 Tarihçe

Bugünkü işletim sistemlerinde bulunan pek çok fikir, belirli bir dönemin belirli bir sorununa çözüm olarak doğmuştur. Biriktirme, çoklu programlama ve zaman paylaşımı birer moda değil, pahalı bir kaynağı boşa harcamamanın yollarıdır. Tarihi bilmek, bu fikirlerin neden var olduğunu anlamayı kolaylaştırır.

1.4.1 Nesiller

Tanenbaum, bilgisayarların ve işletim sistemlerinin gelişimini donanım teknolojisindeki sıçramalara göre nesillere ayırır.



Şekil 1.18: İşletim sistemlerinin nesilleri. 1. nesil (1945–55) vakum tüpleri, İS yok, kablo panoları; 2. nesil (1955–65) transistörler ve toplu iş; 3. nesil (1965–80) tümleşik devreler, çoklu programlama ve zaman paylaşımı; 4. nesil (1980–) kişisel bilgisayar, MS-DOS, Mac, Windows, Linux; 5. nesil (1990–) mobil aygıtlar, Symbian, iOS, Android. Son kutu (2000–, sanallaştırma ve bulut: hipervizör, konteyner, IoT) Tanenbaum'un nesillerinden biri değildir; bugünkü sistemleri anlamak için eklenmiştir.

Her nesil yeni bir elektronik teknolojiyle başlar: vakum tüpü, transistör, tümleşik devre, mikroişlemci. Teknoloji değiştikçe makinelerin boyu, fiyatı ve güvenilirliği değişir. Değişmeyen ise her nesilde yeniden sorulan şu sorudur: pahalı olan kaynağı nasıl daha iyi kullanırız? Bu kaynak ilk nesillerde işlemcinin kendisiydi. Fiyatlar düştükçe pahalı kaynak bellek ve en sonunda *insan zamanı* oldu.

1.4.2 1. ve 2. nesil

Birinci nesil (1945–55) vakum tüpleriyle yapılan makinelerin dönemidir. ENIAC (1945) gibi bu makineler bir odayı doldururdu; yavaştı ve tüpler sık yandığı için sık arızalanırdı. Aynı küçük mühendis grubu makineyi tasarlar, kurar, programlar ve bakımını yapardı. Programlama makine diliyle ya da kablo panolarındaki bağlantıları değiştirerek yapılırdı; 1950'lerin başında delikli kartlar kullanılmaya başlandı. Programlama dili de işletim sistemi de yoktu. Makineler çoğunlukla sayısal tablolar hesaplamak için kullanılırdı.

İkinci nesil (1955–65) transistörle başladı. Transistör tüpün yerini aldı; makineler küçüldü, güvenilir hâle geldi ve daha az enerji harcadı. Bu dönemin ana bilgisayarları klimalı odalarda dururdu ve fiyatları milyonlarca dolardı. Programcı ile makine arasına bir **operatör** girdi: programcı kart destesini operatöre teslim eder, operatör işi çalıştırır, sonucu basılı kâğıt olarak geri verirdi.

Bu düzenin büyük bir kaybı vardı. Bir iş bitince operatör sıradaki işin kartlarını okutup derleyiciyi yükleyene kadar milyonluk makine boş beklerdi. Çözüm **toplu iş** oldu.

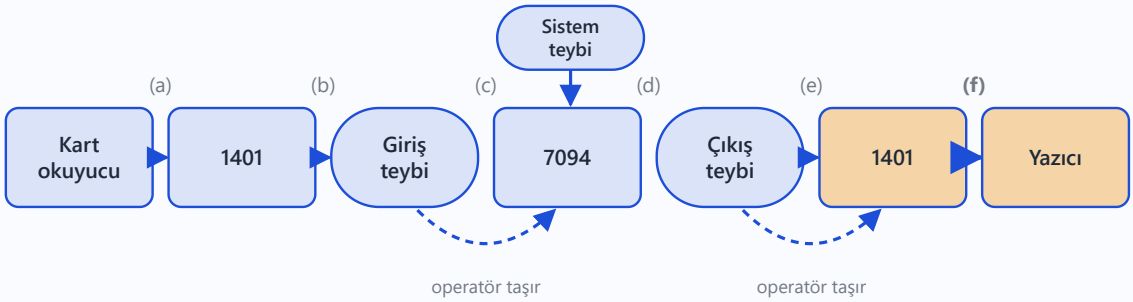
Tanım 1.11

Toplu iş sistemi (*batch system*), işlerin önceden bir araya toplandığı ve küçük bir denetim programının, yani **monitörün**, bu işleri insan müdahalesi olmadan sırayla çalıştırdığı sistemdir.

Toplu işin amacı, işler arasındaki el ile geçiş süresini ortadan kaldırmaktı. İşler bir manyetik teybe toplanır, monitör onları sırayla ve otomatik olarak çalıştırırdı; operatör yalnız teypleri taşırdı. Ama iki sınır kaldı. Bellekte aynı anda tek iş bulunduğu için iş G/Ç yaparken işlemci boşta beklerdi. Programcı da işi teslim ettikten ancak saatler sonra sonuç alabiliyordu; etkileşim yoktu. Toplu iş, etkileşimli ve gerçek zamanlı kullanıma uygun değildi.

1.4.3 Erken bir toplu iş sistemi

İkinci neslin tipik bir kurulumu, iki ayrı IBM makinesini birlikte kullanırdı: ucuz bir IBM 1401 ve pahalı bir IBM 7094.



Şekil 1.19: Erken bir toplu iş sistemi. (a) Kart okuyucu, (b) 1401 kartları giriş teybine yazar, (c) operatör giriş teybini 7094'e taşır, (d) 7094 sistem teybindeki monitörle işleri hesaplar ve sonuçları çıkış teybine yazar, (e) operatör çıkış teybini 1401'e taşır, (f) 1401 sonuçları yazıcıdan basar.

1. (a) Programcılar delikli kart destelerini 1401'in kart okuyucusuna getirir.
2. (b) Ucuz 1401, bir grup işin kartlarını okuyup giriş teybine yazar. Yavaş kart okuma işi böylece pahalı makineden ayrılır.
3. (c) Operatör dolan giriş teybini 7094'e taşır.
4. (d) 7094, sistem teybindeki monitör programının denetiminde işleri sırayla hesaplar ve çıktıları çıkış teybine yazar.
5. (e) Operatör çıkış teybini yeniden 1401'e taşır.
6. (f) 1401 çıktıları yazıcıdan basar. Bu işe *çevrim dışı yazdırma* denirdi.

Bu düzenin mantığı açıktır: pahalı 7094 hiç kart okumak ya da yazdırmakla uğraşmaz. Teypten okumak karttan okumaktan çok daha hızlıdır, dolayısıyla pahalı makinenin zamanı hesaplamaya kalır. Yavaş G/Ç işleri ucuz makineye devredilmiştir. Aynı ilke, biraz farklı bir biçimde, bugünkü biriktirmede de yaşar.

1.4.4 IBM 1401, 7094 ve System/360

Bu dönemin üç makinesi, işletim sistemlerinin evriminde ayrı birer basamaktır. Tablo 1.4 onları karşılaştırır.

Makine	Yıl	Özellik
IBM 1401	1959	Ticari veri işleme; karakter tabanlı, görece ucuz. Kart okuma, teybe kopyalama ve yazdırma gibi G/Ç işlerinde de kullanıldı. 1960'ların en yaygın bilgisayarlarından biriydi.
IBM 7094	1962	Bilimsel hesaplama; 36 bit sözcüklü ve 1401'den tamamen farklı bir mimari. IBSYS ve FMS altında toplu iş çalıştırdı. MIT'de değiştirilmiş bir 7094 üzerinde CTSS ile zaman paylaşımı denendi.
System/360	1964	Küçükten büyüğe aynı mimariyi paylaşan bir makine ailesi. İşletim sistemi OS/360: çoklu programlama ve biriktirme içeren, milyonlarca satırlık, hatalarla dolu dev bir proje.

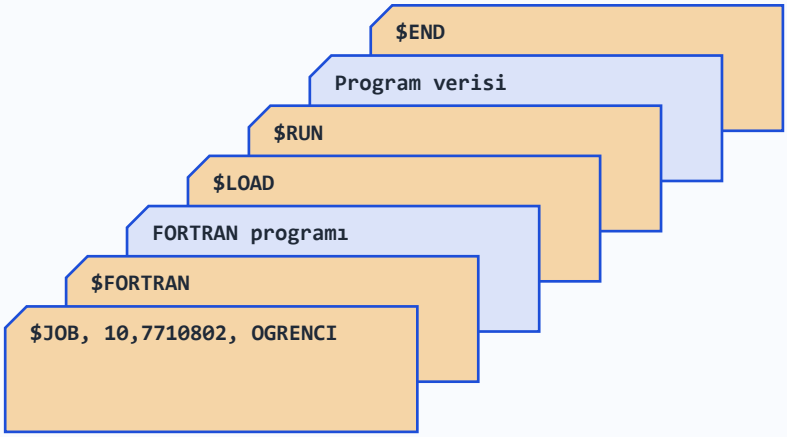
Tablo 1.4: Üç IBM makinesi.

Tablodaki iki ayrıntı sık karıştırılır. Birincisi, 1401 ve 7094 OS/360 altında çalışmazdı; OS/360, 1964'te duyurulan System/360 ailesi için yazılmıştı. İkincisi, 7094, 1401'in güçlü bir sürümü değildi; onu tamamlayan, bambaşka mimarili ayrı bir makineydi. İki makinenin birbirinin programını çalıştıramaması, IBM'i System/360'ta tek mimariye dayanan bir aile kurmaya itti. Bu aile fikri, bir programın küçük makinede yazılıp büyük makinede değiştirilmeden çalışabilmesi anlamına geliyordu ve bugünkü x86 ya da ARM ailelerinin de temelidir.

OS/360'ın geliştirilmesi yazılım mühendisliğinin klasik örneklerinden biri oldu. Projenin yöneticilerinden Fred Brooks, deneyimlerini *The Mythical Man-Month* kitabında anlattı. Kitaptaki en ünlü gözlem, geciken bir yazılım projesine yeni insan eklemenin projeyi daha da geciktirdiğidir.

1.4.5 Tipik bir FMS işi

FMS, IBM 709 ve 7094 için yazılmış bir toplu iş monitörüdür; adı *Fortran Monitor System*'in kısaltmasıdır. Bir iş, monitöre ne yapacağını söyleyen özel kartlarla çevrelenmiş bir kart destesiydi.



Şekil 1.20: Tipik bir FMS işinin kart destesi. Öndeki ilk karttan arkaya doğru: `$JOB` (en çok çalışma süresi, hesap numarası, programcı adı), `$FORTRAN`, FORTRAN programı, `$LOAD`, `$RUN`, program verisi ve `$END`.

$ ile başlayan kartlara **denetim kartları** denir. Monitör desteyi sırayla okur ve her denetim kartında bir iş yapar. `$JOB` yeni bir işi başlatır ve hesabını açar. `$FORTRAN` derleyiciyi yükler ve ardından gelen kartları derletir. `$LOAD` derlenen programı belleğe yükler. `$RUN` programı çalıştırır ve sonraki kartları onun verisi olarak verir. `$END` işi bitirir. Bu kartlar, bugünkü kabuk betiklerinin ve IBM'in iş denetim dili JCL'nin atasıdır. Bir kabuk betiği de aynı biçimde “derle, bağla, çalıştır” adımlarını sırayla yazan bir komut listesidir.

Sık yapılan hata

FMS kısaltmasını fabrika otomasyonundaki Esnek Üretim Sistemi (*Flexible Manufacturing System*) ile karıştırmak. Buradaki FMS, FORTRAN Monitor System'dir; iki kavram arasında hiç-bir bağ yoktur.

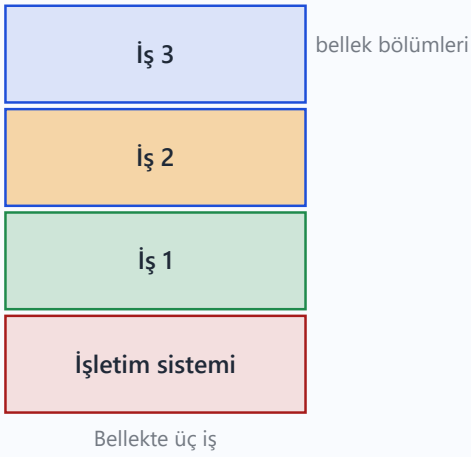
1.4.6 3. nesil: tümleşik devreler ve çoklu programlama

Üçüncü nesil (1965–80), tümleşik devrelerin dönemidir. System/360 gibi uyumlu aileler ortaya çıktı; makineler daha küçük, daha ucuz ve daha güvenilir oldu. Bu nesil, işletim sistemlerine üç kalıcı fikir kazandırdı.

Birinci fikir **çoklu programlamadır**.

Tanım 1.12

Çoklu programlama (*multiprogramming*), belleği bölümlere ayırıp her bölümde ayrı bir iş tutma ve bir iş G/Ç beklerken işlemciyi hemen bellekteki başka bir işe verme tekniğidir.



Şekil 1.21: Çoklu programlamada bellek dört bölüme ayrılmıştır: en altta işletim sistemi, üstünde iş 1, iş 2 ve iş 3.

İkinci nesilde bellekte tek iş bulunuyordu ve iş teypten okuma yaparken işlemci bekliyordu. Bilimsel hesaplamada bu bekleme küçüktü; ama ticari veri işlemede işlerin zamanının büyük kısmı G/Ç ile geçiyordu ve işlemci zamanının çoğu boşa gidiyordu. Çoklu programlamada ise Şekil 1.21'deki gibi bellekte birkaç iş birlikte durur. Biri G/Ç beklerken işlemci başka birini çalıştırır.

Tek işlemcili bir makinede işler gerçekten aynı anda çalışmaz; işlemci bir anda yalnız birini yürütür. İşler zaman içinde iç içe ilerler. Bu, gerçek paralellik değil, eş zamanlılıktır. Bellekte birden çok iş bulunması yeni bir sorun da getirir: bir iş, hatası yüzünden ya da kasten, ötekinin belleğine yazmamalıdır. Bu yüzden çoklu programlama, belleği koruyan donanım olmadan güvenle yapılamaz.

İkinci fikir **biriktirmedir**. Kartlar artık ayrı bir makine ve teyp aracılığıyla değil, doğrudan diske okunuyordu. Bir iş bitince sıradaki iş diskten belleğe yüklenirdi. 1401 ile teyp taşıma düzeni böylece ortadan kalktı.

Üçüncü fikir **zaman paylaşımıdır**.

Tanım 1.13

Zaman paylaşımı (*time sharing*), çok sayıda kullanıcının kendi terminalinden aynı makineyi etkileşimli olarak kullanabilmesi için işlemcinin kısa zaman dilimleriyle süreçler arasında döndürülmesidir. Bir sürecin dilimi dolunca, G/Ç yapmasa bile, zamanlayıcı kesmesiyle işlemci elinden alınır.

Çoklu programlama ile zaman paylaşımı arasındaki fark ince ama önemlidir. Çoklu programlamada işlemci bir işten ancak o iş beklemeye girince alınır; hiç G/Ç yapmayan bir iş işlemciyi saatlerce tutabilir. Zaman paylaşımında ise donanım zamanlayıcısı düzenli aralıklarla kesme üretir ve her sürecin dilimi sınırlıdır. Amaç verimden çok, her kullanıcıya kısa bir tepki süresi sağlamaktır. Programcı artık işini teslim edip saatlerce beklemez; komutunu yazar ve saniyeler içinde yanıt alır.

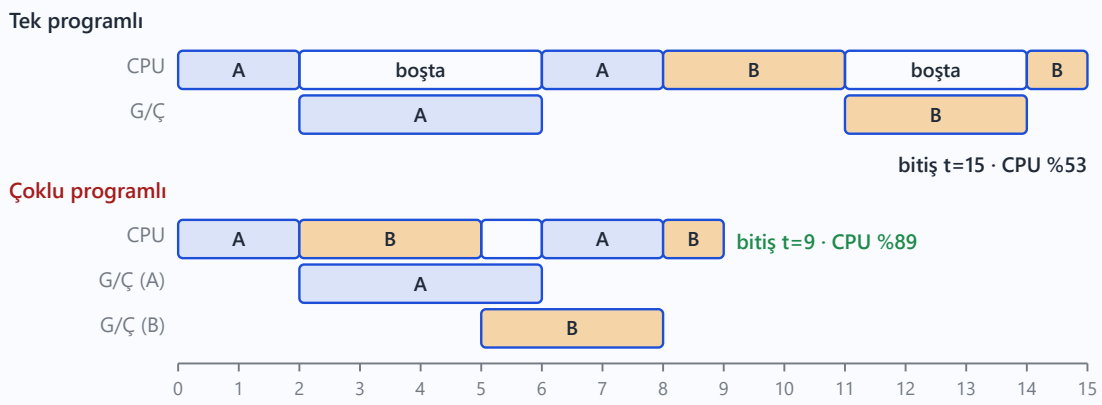
Zaman paylaşımının soyağacı bugünkü sistemlere kadar uzanır: MIT'de geliştirilen CTSS (1961), ardından çok iddialı MULTICS projesi, MULTICS'ten ayrılan Bell Labs araştırmacılarının yazdığı UNIX (1969). UNIX'ten Berkeley'deki BSD sürümleri doğdu. Farklı UNIX türevlerinin birbirleriyle uyumlu olması için POSIX standardı yazıldı. Andrew Tanenbaum öğretim amacıyla MINIX'i geliştirdi; Linus Torvalds da 1991'de MINIX'ten esinlenerek Linux çekirdeğini yazdı.

1.4.7 Çoklu programlama zaman çizelgesi

Çoklu programlamanın kazancını sayıyla görmek için iki işi tek işlemcili bir makinede çalıştıralım. Süreler birim zaman cinsindendir:

- **A** işi: 2 birim CPU, 4 birim G/Ç, 2 birim CPU.
- **B** işi: 3 birim CPU, 3 birim G/Ç, 1 birim CPU.

İki işin toplam CPU ihtiyacı *2 + 2 + 3 + 1 = 8* birimdir. Her işin G/Ç'si kendi aygıtında yapılır, yani A'nın G/Ç'si ile B'nin G/Ç'si birbirini beklemez.



Şekil 1.22: Tek programlı ve çoklu programlı zaman çizelgeleri. Üstte tek programlı sistem: A 0–2, CPU boşta 2–6, A 6–8, B 8–11, CPU boşta 11–14, B 14–15; bitiş *t = 15*, CPU kullanımı %53. Altta çoklu programlı sistem: A 0–2, B 2–5, boşta 5–6, A 6–8, B 8–9; A'nın G/Ç'si 2–6, B'nin G/Ç'si 5–8; bitiş *t = 9*, CPU kullanımı %89.

Örnek 1.2

Tek programlı sistem. Bellekte tek iş bulunur; B, ancak A tamamen bitince başlar.

1. *t = 0–2*: A CPU kullanır.
2. *t = 2–6*: A G/Ç yapar. Bellekte başka iş olmadığı için CPU 4 birim boşta kalır.
3. *t = 6–8*: A son CPU dilimini kullanır ve biter.
4. *t = 8–11*: B CPU kullanır.
5. *t = 11–14*: B G/Ç yapar; CPU 3 birim boşta.
6. *t = 14–15*: B son CPU dilimini kullanır ve biter.

Toplam süre 15 birim, CPU'nun çalıştığı süre 8 birimdir. CPU kullanımı *8 / 15 ≈ 0,53*, yani yaklaşık %53'tür. Boşta geçen 7 birim, tam olarak iki işin G/Ç sürelerinin toplamıdır (*4 + 3*).

Örnek 1.3

Çoklu programlı sistem. İki iş de bellektedir; CPU boşalınca hazır olan işe verilir.

1. *t = 0–2*: A CPU kullanır; B hazır olarak bekler.
2. *t = 2*: A G/Ç'ye gider (2–6). CPU boş kalmaz, hemen B'ye verilir: B *t = 2–5* arasında çalışır. A'nın G/Ç'si bu sırada arka planda sürer.
3. *t = 5*: B G/Ç'ye gider (5–8). A'nın G/Ç'si henüz bitmemiştir; hazır iş yoktur ve CPU yalnız *t = 5–6* arasında boşta kalır.
4. *t = 6*: A'nın G/Ç'si biter; A son CPU dilimini *t = 6–8* arasında kullanır ve biter.
5. *t = 8*: B'nin G/Ç'si tam bu anda biter; B son dilimini *t = 8–9* arasında kullanır ve biter.

İki iş 15 yerine 9 birimde bitmiştir. CPU kullanımı *8 / 9 ≈ 0,89*, yani yaklaşık %89'dur. Boşta geçen tek birim, iki işin aynı anda G/Ç beklediği *t = 5–6* aralığıdır.

İki örnek arasındaki farkın kaynağı tektir: çoklu programlamada CPU ile G/Ç *örtüşür*. Tek programlı sistemde A'nın 4 birimlik G/Ç'si boyunca CPU boş beklerken, çoklu programlı sistemde bu sürenin 3 birimi B'nin hesabıyla doldurulmuştur. İşlerin toplam CPU ihtiyacı değişmemiş, yalnız beklemeler başka işlerin hesabıyla örtülmüştür. Bu yüzden çoklu programlamanın kazancı, işlerin G/Ç oranı yükseldikçe artar. Hiç G/Ç yapmayan iki iş için kazanç sıfırdır.

Sık yapılan hata

Çoklu programlamanın işleri "daha hızlı çalıştırdığını" düşünmek. Her işin kendi CPU süresi aynı kalır; A yine 4, B yine 4 birim CPU kullanır. Kısalan şey, bütün işlerin bitmesi için geçen toplam süredir. Örnekte A her iki sistemde de *t = 8*'de biter; kazanç tamamen B'nin 15 yerine 9'da bitmesinden gelir. A'nın G/Ç'si 2 birim sürseydi, *t = 4*'te bitecekti. CPU o sırada B'de olduğu için A *t = 5*'e kadar bekler ve *t = 7*'de biterdi; tek başına çalışsa *t = 6*'da bitecekti. Çoklu programlama toplam işi hızlandırır, tek bir işin bitişini ise geciktirebilir.

1.4.8 4. nesil: kişisel bilgisayarlar

Dördüncü nesil (1980–), binlerce transistörü tek bir yongaya sığdıran çok büyük ölçekli tümleşik devrelerle, yani mikroişlemcilerle başladı. Bilgisayar ilk kez bir kişinin masasına sığacak kadar küçüldü ve bir kişinin alabileceği kadar ucuzladı. Tablo 1.5 bu dönemin işletim sistemi kilometre taşlarını sıralar.

Yıl	Gelişme
1974	Gary Kildall'ın Intel 8080 için yazdığı CP/M: mikrobilgisayarlar için disk tabanlı ilk yaygın İS.
1981	IBM PC ve 86-DOS'tan türetilen MS-DOS; BASIC yorumlayıcısı ROM'da.
1973–84	Xerox PARC'ta geliştirilen GUI fikirleri, Apple Lisa (1983) ve Macintosh (1984) ile kullanıcı dostu arayüze dönüştü.
1985–2000	Windows 1.0 … 3.x, MS-DOS üzerinde çalışan bir grafik ortamdı; Windows 95/98/ME de hâlâ MS-DOS tabanlıydı.
1993–	Sıfırdan yazılan Windows NT; ardından 2000, XP, 7, 10 ve 11. Bugünkü Windows'un çekirdeği NT'dir.
1991–	Linus Torvalds'ın Linux çekirdeği; 2001'de UNIX tabanlı Mac OS X (Darwin/XNU), bugünkü macOS'un atası.

Tablo 1.5: Kişisel bilgisayar döneminin kilometre taşları.

Bu dönemde işletim sisteminin hedefi değişti. Önceki nesillerde makine pahalı, insan zamanı ucuzdu; işletim sistemi makineyi bir an bile boş bırakmamaya çalışırdı. Fiyatların hızla düşmesi bilgisayarı herkese açınca denge tersine döndü. Artık asıl değerli olan kullanıcının zamanıydı ve İS'nin hedefi "makineyi verimli kullan"dan "kullanıcıya kolaylık sağla"ya kaydı. GUI'nin yükselişi bu kaymanın sonucudur.

MS-DOS ile Windows NT arasındaki geçiş de öğreticidir. MS-DOS tek görevli bir sistemdi ve bellek koruması yoktu; bir programın hatası bütün sistemi çökertebilirdi. Windows 95/98/ME bu temelin üstüne kurulduğu için kararsızlıklarıyla tanındı. Windows NT ise baştan koruma, çoklu görev ve çok kullanıcılı çalışma düşünülerek yazıldı. 2001'de XP ile birlikte ev kullanıcıları da NT çekirdeğine geçti.

1.4.9 5. nesil: mobil aygıtlar ve günümüz

Beşinci nesil (1990–) mobil aygıtların dönemidir. 1990'larda PDA'lar ve Palm OS, 2000'lerde Symbian ve BlackBerry OS yaygındı. 2007'de iPhone ile iOS, 2008'de Linux çekirdeği üzerine kurulu Android çıktı. Microsoft'un Windows Phone'u 2010'da piyasaya girdi, 2017'de geliştirilmesi durduruldu. Bugün pazarı Android ve iOS paylaşır. Mobil işletim sistemlerinde dokunmatik arayüz, pil yönetimi ve uygulamaların hangi verilere erişebileceğini belirleyen izin modeli öne çıkar.

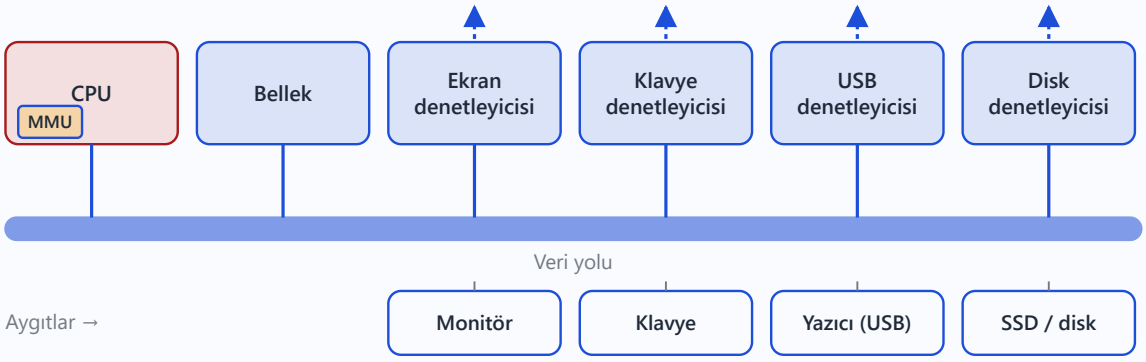
Mobil dünyanın yanında dört eğilim bugünkü işletim sistemlerini biçimlendirir:

- **Sanallaştırma ve bulut**: **hipervizörler** (KVM, Hyper-V, Xen) tek bir fiziksel sunucuda çok sayıda sanal makine çalıştırır. Bulut hizmetlerinin temeli budur.
- **Konteynerler**: Docker ve Kubernetes gibi araçlar, aynı çekirdeği paylaşan ama birbirinden yalıtılmış ortamlar kurar. Yalıtım Linux'un ad alanları (*namespaces*) ve cgroups özellikleriyle sağlanır.
- **Heterojen donanım**: bir yongada CPU, GPU ve yapay zekâ işlemleri için NPU birlikte bulunur; işlemcilerde büyük ve küçük çekirdekler bir arada kullanılır.
- **Güvenlik**: bellek güvenliğini dil düzeyinde sağlayan diller çekirdeğe giriyor. Linux 6.1'den beri çekirdekte Rust desteği vardır.

1.5 Donanım

İşletim sistemi donanımı yönettiği için donanımın özellikleri İS tasarımını doğrudan biçimlendirir. Yazmaçların sayısı bağlam değiştirmenin maliyetini, bellek düzeyleri arasındaki hız farkı çizelgeleme kararlarını, çekirdek sayısı eş zamanlılık sorunlarının boyutunu belirler. Bu kesim, işletim sistemi açısından önemli olan donanım özelliklerini özetler.

1.5.1 Kişisel bilgisayarın bileşenleri



Şekil 1.23: Kişisel bilgisayarın bileşenleri. CPU (içinde MMU), bellek ve ekran, klavye, USB ve disk denetleyicileri ortak bir veri yoluna bağlıdır. Denetleyicilerin altında monitör, klavye, USB yazıcı ve SSD/disk aygıtları bulunur.

Şekil 1.23 basitleştirilmiş bir modeldir. İşlemci, bellek ve G/Ç aygıtları bir **veri yolu** (*bus*) üzerinden haberleşir. Her aygıtın bir **denetleyicisi** vardır: monitörü ekran denetleyicisi, diski disk denetleyicisi sürer. İşlemci aygıtla doğrudan değil, onun denetleyicisiyle konuşur.

İşlemcinin içinde görülen **MMU** (*memory management unit*: bellek yönetim birimi), programların kullandığı sanal adresleri fiziksel adreslere çeviren donanımdır. 1.3.9'daki ikinci deneyde iki sürecin aynı sanal adresinin farklı fiziksel çerçevelere eşlenebilmesi bu birim sayesindedir. Çeviri kurallarını, yani sayfa tablolarını işletim sistemi yazar; çeviriyi her bellek erişiminde MMU yapar.

Gerçek sistemlerde tek bir ortak hat yerine PCIe gibi noktadan noktaya bağlantılar kullanılır; her aygıt işlemciye kendi hattıyla bağlanır. Ama kavramsal model aynı kalır: işlemci, bellek ve aygıt denetleyicileri birbirine bağlıdır ve işletim sistemi denetleyicilerle konuşarak aygıtları yönetir.

1.5.2 İşlemci ve yazmaçlar

İşlemci bilgisayarın "beynidir": bütün komutları o yürütür. Çalışması basit bir döngüye dayanır.

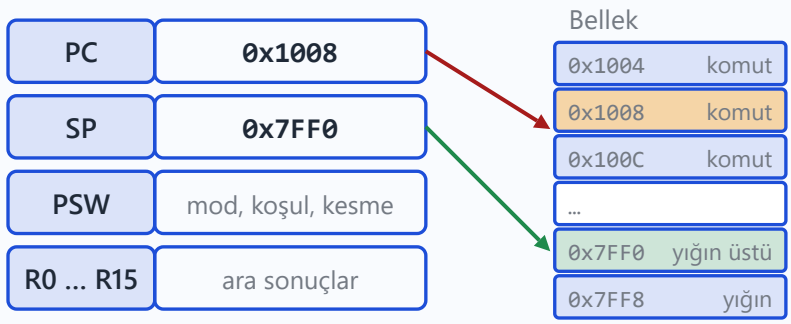
Tanım 1.14

İşlemcinin temel çalışma biçimi **getir–çöz–yürüt döngüsüdür** (*fetch–decode–execute cycle*): sıradaki komut bellekten getirilir, ne yapılacağı çözülür ve komut yürütülür. Ardından bir sonraki komuta geçilir.

Bellek işlemciye göre yavaş olduğu için değişkenler ve ara sonuçlar işlemcinin içindeki küçük ve çok hızlı saklama birimlerinde, yani **yazmaçlarda** (*register*) tutulur. Veriyi bellekten yazmaca taşıyan komuta *yükle*, yazmaçtan belleğe taşıyan komuta *sakla* denir. Genel amaçlı yazmaçların yanında, işlemcinin durumunu tutan özel yazmaçlar vardır.

Yazmaç	x86-64	ARM64
Program sayacı: sıradaki komutun adresi	`RIP`	`PC`
Yığın işaretçisi: yığının üstü	`RSP`	`SP`
PSW: koşul bitleri, mod, kesme izni	`RFLAGS`	`PSTATE`
Genel amaçlı yazmaç sayısı	16	31

Tablo 1.6: Başlıca yazmaçlar ve iki mimarideki adları.



Şekil 1.24: Yazmaçlar ve bellek. PC (`0x1008`) sıradaki komutu, SP (`0x7FF0`) yığının üstünü gösterir. PSW mod, koşul ve kesme bitlerini; R0 … R15 ara sonuçları tutar.

Program sayacı (*program counter*, PC) sıradaki komutun adresini tutar; her komuttan sonra ilerler, atlama komutlarında yeni adresi alır. **Yığın işaretçisi** (*stack pointer*, SP), fonksiyon çağrılarında yerel değişkenlerin ve dönüş adreslerinin tutulduğu yığının üstünü gösterir. **Program durum sözcüğü** (*program status word*, PSW) ise son karşılaştırmanın sonucunu tutan koşul bitlerini, işlemcinin modunu (çekirdek ya da kullanıcı) ve kesmelerin açık olup olmadığını içerir. 1.2.2'deki mod biti de bu yazmaçtadır.

Her işlemci ailesinin kendi komut kümesi vardır. x86-64 için derlenmiş bir program ARM işlemcide doğrudan çalışmaz; ya yeniden derlenmesi ya da bir çevirici katmandan geçmesi gerekir. İşlemcinin başarımı ise tek bir sayıyla ifade edilemez: saat hızı, çekirdek sayısı, önbellek boyutu, komutların iş hattında kaç aşamada işlendiği ve bellek bant genişliği birlikte etkilidir.

Yazmaçların işletim sistemi için asıl önemi **bağlam değiştirmedir**.

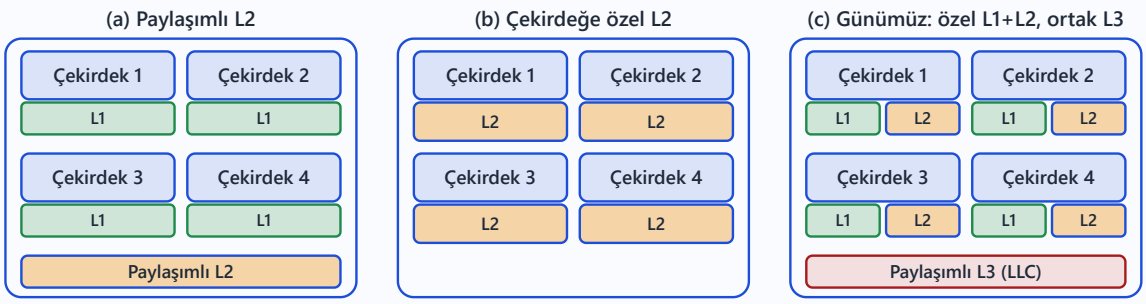
Tanım 1.15

Bağlam değiştirme (*context switch*), işlemcinin bir süreci bırakıp başka bir süreci çalıştırmaya geçmesidir. İşletim sistemi bunu yaparken bırakılan sürecin bütün yazmaçlarını belleğe kaydeder ve sıradaki sürecin daha önce kaydedilmiş yazmaçlarını geri yükler.

Kaydetmenin neden zorunlu olduğu kolayca görülür. İşlemcide tek bir yazmaç takımı vardır. Sıradaki süreç çalışmaya başlayınca bu yazmaçların hepsini kendi değerleriyle ezer. Kesilen süreç daha sonra kaldığı yerden sürecekse, aynı PC'ye, aynı SP'ye ve aynı ara değerlere dönmesi gerekir. Tek bir yazmaç bile eksik kaydedilse, süreç döndüğünde yanlış bir değerle hesaba devam eder ve hata nedeni bulunamayacak kadar gizli kalır.

1.5.3 Çok çekirdekli yongalar

Saat hızını artırmak 2000'lerin ortasında ısı sınırına dayandı. Bunun üzerine üreticiler tek bir hızlı işlemci yerine aynı yongaya birden çok işlemci koymaya başladı. Bugün telefonlarda bile sekiz ya da daha çok çekirdek bulunur.



Şekil 1.25: Dört çekirdekli yongalarda önbellek düzenleri. (a) Her çekirdeğin kendi L1'i ve paylaşımlı bir L2. (b) Her çekirdeğe özel L2. (c) Günümüzün yaygın düzeni: her çekirdeğe özel L1 ve L2, bütün çekirdeklerin paylaştığı L3 (son düzey önbellek, LLC).

Çok çekirdekli yongalarda üç ayrı fikir bir arada bulunur:

- **Çok çekirdek**: bir yongada birden çok bağımsız işlemci bulunur. İşletim sistemi her birini ayrı bir işlemci olarak görür ve her birine ayrı süreç çizelgeler.
- **Eş zamanlı çoklu iş parçacığı** (*simultaneous multithreading*, SMT; Intel'deki ticari adı Hyper-Threading): bir çekirdek iki donanım iş parçacığını aynı anda çalıştırır. İşletim sistemine iki "mantıksal CPU" görünür, ama bu ikisi aynı çekirdeğin yürütme birimlerini ve önbelleklerini paylaşır.

- **Hibrit çekirdekler**: aynı yongada başarım (P) ve verimlilik (E) çekirdekleri bulunur. ARM'ın big.LITTLE tasarımı ve Intel'in 2021'den beri ürettiği işlemciler bu türdendir. Çizelgeleyici, hangi işin hangi tür çekirdekte çalışacağını bilmelidir: arka plandaki bir güncelleme E çekirdeğinde pil harcamadan ilerleyebilir, oyunun ana iş parçacığı ise P çekirdeğine aittir.

Örnek 1.4

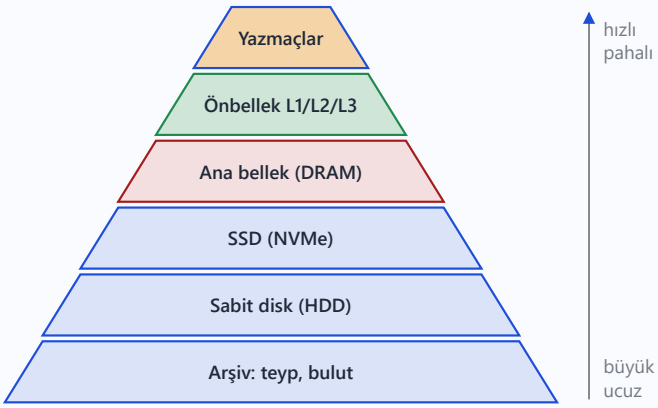
Bir dizüstü işlemcide SMT destekli 6 başarım çekirdeği ve SMT desteği olmayan 8 verimlilik çekirdeği vardır. İşletim sistemi kaç mantıksal CPU görür?

Çözüm. Her P çekirdeği iki donanım iş parçacığı çalıştırdığı için $6 \times 2 = 12$ mantıksal CPU verir. E çekirdekleri birer mantıksal CPU sayılır: 8. Toplam $12 + 8 = 20$ mantıksal CPU görünür. Bu 20 CPU'nun gücü eşit değildir. Aynı P çekirdeğini paylaşan iki mantıksal CPU, iki ayrı çekirdek kadar hızlı değildir; bu yüzden çizelgeleyici önce boştaki fiziksel çekirdekleri doldurmaya çalışır.

İşletim sistemi açısından çok çekirdeğin sonucu üç başlıkta toplanır. **Paralellik**: işler artık gerçekten aynı anda çalışır ve işletim sistemi onları çekirdeklere dengeli dağıtmalıdır. **Senkronizasyon**: iki işlemci çekirdeği, İS çekirdeğinin aynı veri yapısını aynı anda değiştirmeye kalkabilir; 1.3.9'daki sayaç deneyinin sorunu artık İS'nin kendi içinde de vardır. **Önbellek tutarlılığı**: her çekirdeğin kendi önbelleği olduğu için bir çekirdeğin yazdığı değerin öteki çekirdeğin önbelleğindeki eski kopyayı geçersiz kılması gerekir. Bu son iş donanımca yapılır, ama maliyeti işletim sistemi tasarımını etkiler.

1.5.4 Bellek hiyerarşisi

İdeal bellek hem çok hızlı, hem çok büyük, hem de çok ucuz olurdu. Böyle bir teknoloji yoktur. Hızlı bellek pahalıdır ve küçük üretilir; büyük ve ucuz bellek ise yavaştır. Bilgisayarlar bu yüzden farklı teknolojileri katmanlar hâlinde birlikte kullanır.



Şekil 1.26: Bellek hiyerarşisi piramidi. Üstten alta yazmaçlar, L1/L2/L3 önbellekleri, ana bellek (DRAM), SSD (NVMe), sabit disk (HDD) ve arşiv (teyp, bulut). Yukarı çıktıkça bellek hızlanır ve pahalılaşır; aşağı indikçe

büyür ve ucuzlar.

Düzey	Erişim (yakl.)	Kapasite (tipik)
Yazmaçlar	< 1 ns	< 1 KB
L1 önbellek	~1 ns	32–64 KB
L2 / L3	3–20 ns	1–100+ MB
DRAM	60–100 ns	8–64 GB
NVMe SSD	20–100 µs	0,5–4 TB
HDD	5–10 ms	2–24 TB

Tablo 1.7: Bellek düzeylerinin yaklaşık erişim süreleri ve tipik kapasiteleri (2026 itibarıyla büyüklük sırası).

Tanım 1.16

Bellek hiyerarşisi (*memory hierarchy*), farklı hız, kapasite ve maliyetteki bellek teknolojilerinin, her düzey altındakinin önbelleği gibi çalışacak biçimde katmanlar hâlinde düzenlenmesidir.

Tablodaki iki sınır özel önem taşır. Birincisi **uçuculuk** sınırıdır: DRAM ve üstündeki düzeyler güç gidince içeriğini kaybeder, SSD ve HDD ise kalıcıdır. Kalıcılığın neden dosya sistemi gerektirdiği buradan anlaşılır. İkincisi **büyüklük farkıdır**: komşu düzeyler arasında hız bazen on kat, bazen bin kat değişir. DRAM'den SSD'ye geçerken erişim süresi nanosaniyelerden mikrosaniyelere, HDD'de milisaniyelere çıkar.

Hiyerarşinin işe yaramasının nedeni **yerelliktir**. Programlar kısa bir süre içinde genellikle aynı veriye ya da ona yakın adreslerdeki veriye tekrar tekrar erişir. Bir döngü aynı değişkenleri her turda kullanır; bir dizi baştan sona sırayla okunur. Bu yüzden üst düzeyde tutulan küçük bir kopya, erişimlerin büyük kısmını karşılar ve yavaş alt düzeye seyrek inilir. Sonuçta sistem, neredeyse üst düzeyin hızında ve alt düzeyin kapasitesinde bir bellek gibi davranır.

Örnek 1.5

2,5 GHz'de çalışan bir çekirdekte bir saat turu kaç ns sürer? 100 ns'lik bir DRAM erişimi ve 8 ms'lik bir HDD erişimi kaç saat turuna denk gelir?

Çözüm. 2,5 GHz, saniyede $2{,}5 \times 10^9$ tur demektir. Bir tur $1 / (2{,}5 \times 10^9)\ s = 0{,}4\ ns$ sürer; başka bir deyişle 1 ns'de 2,5 tur geçer. DRAM erişimi $100 \times 2{,}5 = 250$ tura denk gelir. HDD erişimi önce nanosaniyeye çevrilir: $8\ ms = 8 \times 10^6\ ns$; buradan $8 \times 10^6 \times 2{,}5 = 2 \times 10^7$, yani 20 milyon tur bulunur.

Bu sayılar işletim sisteminin davranışını açıklar. 250 turluk DRAM beklemesini donanım önbellekleri gizler; işletim sisteminin araya girip süreç değiştirmesine değmez, çünkü bağlam değiştirmenin kendisi bundan uzun sürer. 20 milyon turluk disk beklemesi ise çok büyüktür. İşletim sistemi bu sürede süreci bekletir ve işlemciyi başka bir sürece verir. Çoklu programlamanın temel gerekçesi de budur.

Sık yapılan hata

SSD'yi "belleğin hızlı bir türü" sanmak. NVMe SSD, HDD'den yaklaşık yüz kat hızlıdır, ama DRAM'den hâlâ yaklaşık bin kat yavaştır. İşletim sistemi için SSD de bir G/Ç aygıtıdır: okuma beklenirken süreç bekletilir ve işlemci başka işe verilir.

1.6 G/Ç yöntemleri

İşlemcinin aygıtlarla konuşması, işletim sisteminin en sık yaptığı işlerden biridir. Bir tuşa basılması, bir ağ paketinin gelmesi, bir disk bloğunun okunması: hepsi bir G/Ç olayıdır. Bu kesim, işlemcinin bir G/Ç işinin bittiğini nasıl öğrendiğini ele alır.

1.6.1 Denetleyici ve sürücü

Her G/Ç aygıtının bir **aygıt denetleyicisi** (*device controller*) vardır. Denetleyici, aygıtın üzerindeki küçük bir işlemci ya da devredir; aygıtı fiziksel olarak sürer ve işlemciye birkaç yazmaçtan oluşan bir arayüz sunar. Tipik olarak bir *komut* yazmacı (ne yapılacak), bir *durum* yazmacı (iş bitti mi, hata var mı) ve bir *veri* yazmacı ya da tamponu bulunur.

Çekirdekteki **aygıt sürücüsü** (*device driver*), belirli bir denetleyiciyle konuşmayı bilen yazılımdır. Sürücü denetleyicinin yazmaçlarına komut yazar ve durumunu okur. Her denetleyici farklı olduğu için her aygıt türünün kendi sürücüsü vardır; işletim sisteminin geri kalanı ise sürücülerle ortak bir arayüzden konuşur. 1.1.6'daki Aygıt Yöneticisi'nde listelenen sürücüler bunlardır.

1.6.2 Yoklama, kesme ve DMA

Sürücü denetleyiciye bir komut verdikten sonra işin ne zaman bittiğini öğrenmesi gerekir. Bunun üç yolu vardır.

Yöntem	Temel fark
Yoklama (*polling*)	CPU, aygıt hazır olana kadar durum yazmacını döngüde okur; beklerken başka iş yapamaz (meşgul bekleme).
Kesme (*interrupt*)	Aygıt her iş biriminin sonunda CPU'ya sinyal gönderir; CPU arada başka süreci çalıştırır.

Yöntem	Temel fark
DMA (doğrudan bellek erişimi)	Denetleyici bütün bloğu belleğe kendisi taşır; CPU yalnız başta (programlama) ve sonda (tek kesme) devreye girer.

Tablo 1.8: Üç G/Ç yöntemi.

Yoklamanın mantığı basittir: işlemci bir döngüde durum yazmacını okur, "hazır" bitini görene kadar tekrar okur. Bu yönteme **meşgul bekleme** (*busy waiting*) de denir, çünkü işlemci beklerken tam güçle çalışır ama hiçbir yararlı iş yapmaz. Aygıt çok hızlı yanıt veriyorsa, yani bekleme birkaç turdan ibaretse, yoklama en ucuz yoldur. Bekleme uzunsa işlemci zamanı boşa gider.

Kesmede işlemci denetleyiciye komutu verdikten sonra başka bir süreci çalıştırmaya döner. Aygıt bir iş birimini, örneğin bir baytı ya da bir sözcüğü bitirince işlemciye bir **kesme** sinyali gönderir. İşlemci o anki komutu tamamlar, çekirdek moduna geçer ve kesme vektörü denen tablodan ilgili **kesme işleyicisini** bulup çalıştırır. İşleyici veriyi alır, aygıta sıradaki işi verir ve kesilen süreç kaldığı yerden devam eder. Böylece işlemci bekleme sürelerinde yararlı iş yapar. Ama her iş biriminin sonunda bir kesme geldiği için büyük aktarımlarda kesmelerin sayısı ve toplam maliyeti çok yükselir.

DMA (*direct memory access*: doğrudan bellek erişimi) bu son sorunu çözer. İşlemci denetleyiciye yalnız "şu adresten başlayarak şu kadar baytı belleğe yaz" bilgisini verir. Denetleyici bütün bloğu belleğe kendisi taşır ve aktarım bitince *tek bir* kesme üretir. İşlemci yalnız başta, DMA'yı programlarken, ve sonda, bu tek kesmeyi işlerken devreye girer.

1.6.3 CPU yükünü hesaplamak

Örnek 1.6

Bir aygıttan 4 KiB'lık bir blok alınacaktır. Kesme güdümlü G/Ç'de her 4 bayt için bir kesme oluşur ve her kesme CPU'ya 2 µs'ye mal olur. DMA ile CPU denetleyiciyi 3 µs'de programlar ve aktarımın sonunda tek bir kesme (2 µs) gelir. İki yöntemde CPU'nun bu aktarıma harcadığı süre nedir?

1. Önce blok bayta çevrilir. KiB ikili önektir: $4\ KiB = 4 \times 1024 = 4096$ bayt.
2. Kesme güdümlü G/Ç: kesme sayısı $4096 / 4 = 1024$. CPU'nun harcadığı süre $1024 \times 2\ \mu s = 2048\ \mu s \approx 2{,}05\ ms$.
3. DMA: $3\ \mu s + 2\ \mu s = 5\ \mu s$.
4. Oran: $2048 / 5 \approx 410$. DMA, CPU yükünü yaklaşık 410 kat azaltır.

Aktarımın kendisi iki yöntemde de aygıtın hızına bağlıdır ve yaklaşık aynı sürer. Fark, işlemcinin bu süreyi nasıl geçirdiğindedir. DMA'da işlemci 5 µs dışında başka süreçleri çalıştırır. Klavye gibi saniyede birkaç olay üreten yavaş aygıtlarda bu fark önemsizdir; SSD ve ağ kartı gibi hızlı blok aygıtlarında ise DMA olmadan çalışmak düşünülemez.

Sık yapılan hata

DMA'nın güvenlik de sağladığını sanmak. DMA bir başarım mekanizmasıdır; işlemcinin kopyalama yükünü alır. Tersine, belleğe doğrudan yazabilen bir aygıt yanlış programlanırsa ya da kötü niyetliyse belleğin her yerini bozabilir. Aygıtların erişebileceği bellek bölgelerini sınırlamak için IOMMU denen ayrı bir donanım birimi gerekir.

1.6.4 Kesme, tuzak ve istisna

Kesme sözcüğü iki farklı olay için kullanıldığından burada dikkatli olmak gerekir. **Donanım kesmesi**, işlemcinin dışındaki bir aygıttan gelir: zamanlayıcı süresini doldurmuştur, ağ kartına paket gelmiştir, kullanıcı bir tuşa basmıştır. Bu olaylar yürütülen komut akışından bağımsızdır ve her an gelebilir; bu yüzden *eş zamansızdır*.

Tuzak ya da **istisna** ise yürütülen komutun kendisinden doğar: sıfıra bölme, geçersiz bellek erişimi, sayfa hatası ya da bilerek yapılan bir sistem çağrısı. Belirli bir komutun sonucu olduğu için *eş zamanlıdır*: aynı program aynı girdiyle çalıştırılırsa tuzak her seferinde aynı noktada oluşur.

Kaynakları farklı olsa da iki olay çekirdeğe aynı yolla geçer. İşlemci çekirdek moduna girer, o anki durumu kaydeder ve olayın türüne göre bir tablodan (kesme vektörü ya da istisna tablosu) ilgili işleyiciye atlar. İşleyici işini bitirince ya kesilen komuttan sonrasına döner ya da, ölümcül bir hata varsa, süreci sonlandırır. Bu ortak yol, 1.2.2'deki “üç kapı” ile aynı şeydir: sistem çağrısı ve istisna komut akışının içinden, kesme dışarıdan gelir.

Örnek 1.7

Aşağıdaki olaylardan hangileri kesme, hangileri tuzaktır? (a) diskin bir okumayı tamamlaması, (b) bir programın `NULL` işaretçisinden okuma yapması, (c) `fork()` çağrısı, (d) fare hareketi, (e) sayfa hatası.

Çözüm. (a) ve (d) dış aygıtlardan gelir ve komut akışından bağımsızdır: kesme. (b) geçersiz bellek erişimidir ve belirli bir komutun sonucudur: tuzak (istisna). (c) programın bilerek yaptığı sistem çağrısıdır: tuzak. (e) erişilen sayfanın bellekte olmamasından doğar ve erişimi yapan komuta bağlıdır: tuzak (istisna). Program aynı girdiyle yeniden çalıştırıldığında (b) ve (c) aynı komutlarda tekrar oluşur; (e) de aynı türden bir komutta oluşur, ama yalnız sayfa o anda bellekte değilse. (a) ve (d)'nin zamanlaması ise her seferinde farklıdır.

1.7 İşletim sistemi çeşitleri

“İşletim sistemi” adı, bir bankanın veri merkezindeki dev makineyi yöneten yazılım için de bir banka kartının yongasındaki küçük program için de kullanılır. Bu sistemlerin ortak yanı aynı iki rolü üstlenmeleridir; farkları ise hangi kaynağın kıt olduğunda ve neyin en önemli sayıldığındadır.

1.7.1 Büyükten küçüğe sistemler



Şekil 1.27: İşletim sistemi çeşitleri, tipik RAM miktarına göre logaritmik bir eksende. Soldan sağa: akıllı kart ve algılayıcı düğümü (kilobaytlar), gömülü sistem (megabaytlar), mobil ve kişisel sistemler (gigabaytlar), sunucu ve anabilgisayar (yüzlerce gigabayttan terabaytlara).

Şekil 1.27 eksenini logaritmik tutar, çünkü bellek miktarları arasında milyar katlık fark vardır. Aşağıdaki liste en büyükten en küçüğe doğru ilerler.

- **Anabilgisayar** (*mainframe*): çok yüksek G/Ç kapasitesi, güvenilirlik ve kesintisiz erişilebilirlik için tasarlanır. Toplu iş, saniyede binlerce küçük işlem (*transaction*, örneğin bir para transferi ya da uçak rezervasyonu) ve zaman paylaşımı hizmetleri verir. Bankalar ve havayolları bu sistemlere dayanır. Örnekler: IBM z/OS, z/VM, Linux on Z, Unisys MCP, Fujitsu BS2000.
- **Sunucu** (*server*): ağ üzerinden çok sayıda kullanıcıya web, veritabanı, dosya ve e-posta hizmeti verir. Veri merkezleri, bulut, sanallaştırma (KVM, Hyper-V) ve konteynerler bu dünyaya aittir. Örnekler: Linux, Windows Server, FreeBSD.
- **Çok işlemcili** (*multiprocessor*): birden çok çekirdek ya da yuvaya takılı birden çok işlemci kullanır, işleri paralel yürütür. Senkronizasyonla ve belleğin bazı bölgelerinin bazı işlemcilere daha yakın olduğu NUMA düzeniyle uğraşır. Bugün bütün genel amaçlı işletim sistemleri bu sınıftadır: Linux, Windows, macOS.
- **Kişisel** (*personal*): tek kullanıcıya iyi bir GUI ve çoklu görev sunar; tarayıcı, ofis, oyun ve dosya yönetimi gibi işler için kullanılır. Örnekler: Windows 11, macOS, Linux dağıtımları, ChromeOS.
- **Mobil** (*handheld*): dokunmatik arayüz, pil ve ısı yönetimi, algılayıcılar, uygulama mağazası ve izin modeli öne çıkar. Örnekler: Android, iOS/iPadOS, HarmonyOS. Windows Phone 2017'de sona erdi.
- **Gömülü** (*embedded*): araçların, ev aletlerinin, ağ aygıtlarının ve uçakların içinde çalışır. Kullanıcı genellikle yazılım yüklemez; sistem flash bellekte sabittir. Örnekler: gömülü Linux (Yocto), Zephyr, FreeRTOS, QNX, VxWorks.
- **Algılayıcı düğümü** (*sensor node*): birkaç on KB RAM ile, pil üzerinde yıllarca çalışacak biçimde tasarlanır. Olay güdümlüdür: çoğu zaman uyur, bir ölçüm ya da ileti geldiğinde uyanır. Düşük güçlü radyolarla (IEEE 802.15.4, BLE) haberleşir; IoT ve makineden makineye (M2M) ağlarda kullanılır. Örnekler: Contiki-NG, RIOT, Zephyr. TinyOS artık tarihsel bir örnektir.
- **Akıllı kart** (*smart card*): işlemcisi ve belleği son derece kısıtlıdır. Kimlik doğrulama, ödeme, kriptografi ve güvenli saklama işleri yapar. Banka kartları, SIM ve eSIM, kimlik kartları bu sınıftadır. Örnekler: Java Card, MULTOS.

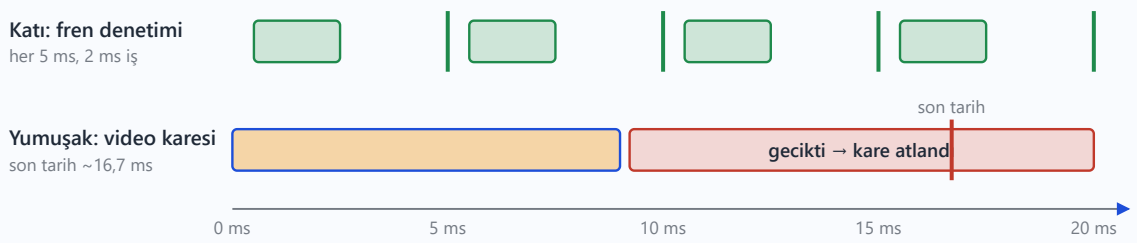
Listenin iki ucu arasında öncelikler tersine döner. Anabilgisayarda ve sunucuda kıt kaynak, binlerce kullanıcının isteğini aynı anda karşılayacak işlem gücü ve G/Ç kapasitesidir. Algılayıcı düğümünde ise kıt kaynak enerjidir; işletim sisteminin en önemli işi, işlemciyi ve radyoyu olabildiğince uzun süre kapalı tutmaktır.

1.7.2 Gerçek zamanlı işletim sistemleri

Bazı sistemlerde bir hesabın doğru olması yetmez; zamanında bitmesi de gerekir. Bir aracın fren denetim yazılımı, tekerleğin kilitlendiğini birkaç milisaniye içinde fark edip frenin basıncını ayarlamalıdır. Doğru ama geç verilen yanıt, yanlış yanıt kadar tehlikelidir.

Tanım 1.17

Gerçek zamanlı işletim sistemi (*real-time operating system*, RTOS), işlerin belirli bir **son tarihten** (*deadline*) önce bitmesini güvence altına almaya çalışan işletim sistemidir. Önemli olan hız değil, *öngörülebilirliktir*: en kötü durumda bile yanıt süresinin bilinen bir sınırı aşmaması.



Şekil 1.28: Katı ve yumuşak gerçek zamanlı görevler, 0–20 ms arasında. Üstte fren denetimi: her 5 ms'de bir 2 ms'lik iş, her iş kendi son tarihinden (dikey çizgiler) önce biter. Altta video karesi: son tarih yaklaşık 16,7 ms'dir; ikinci kare gecikmiş ve atlanmıştır.

Gerçek zamanlı sistemler son tarihin kaçırılmasının sonucuna göre ikiye ayrılır:

- **Katı** (*hard*) gerçek zamanlı: son tarihi kaçırmak felakettir. Fren, hava yastığı, uçuş denetimi ve endüstriyel robotlar bu sınıftadır. Şekil 1.28'deki fren görevi her 5 ms'de bir gelir ve 2 ms'lik işini her seferinde son tarihinden önce bitirir.
- **Yumuşak** (*soft*) gerçek zamanlı: son tarihi ara sıra kaçırmak kabul edilir; yalnız kalite düşer. Ses ve video akışı ile oyunlar bu sınıftadır. Saniyede 60 kare gösteren bir video için her kare $1000 / 60 \approx 16{,}7$ ms içinde hazır olmalıdır. Şekildeki ikinci kare bu süreyi aşmış ve atlanmıştır; izleyici kısa bir takılma görür, ama kimse zarar görmez.

Bilinen RTOS örnekleri arasında Mars gezginlerinde kullanılan VxWorks, araç sistemlerinde yaygın QNX, küçük mikrodenetleyiciler için FreeRTOS ve Zephyr sayılabilir. Linux da uzun süre ayrı tutulan PREEMPT_RT yamalarıyla gerçek zamanlı çalışabiliyordu. Bu yamalar 2024'te Linux 6.12 ile ana çe-

kirdeğe katıldı; eski RTLinux projesi ise sona ermiştir. Gerçek zamanlı sistemler ses, video, hareket ve algılayıcı verisi işleyen uygulamalarda, tren denetiminde ve savunma sistemlerinde kullanılır.

Sık yapılan hata

"Gerçek zamanlı" sözünü "çok hızlı" diye anlamak. A sisteminin ortalama yanıt süresi 1 ms'dir, ama arada bir 500 ms'ye çıkar. B sisteminin ortalaması 5 ms'dir ve hiçbir zaman 8 ms'yi aşmaz. Ortalamada A daha hızlıdır. Yine de katı gerçek zamanlı bir iş için B uygundur, çünkü en kötü durumu bilinir ve sınırlıdır.

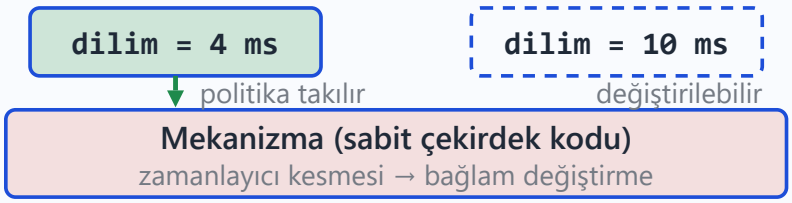
1.8 Temel kavramlar

1.8.1 Politika ve mekanizma

İşletim sistemi tasarımının en kullanışlı ilkelerinden biri, iki farklı soruyu birbirinden ayırmaktır: *ne yapılacak* ve *nasıl yapılacak*.

Tanım 1.18

Politika (*policy*), ne yapılacağına dair karardır. **Mekanizma** (*mechanism*) ise bu kararın nasıl uygulanacağını sağlayan altyapıdır.



Şekil 1.29: Politika, mekanizmaya takılan değiştirilebilir bir parametredir. Mekanizma (sabit çekirdek kodu): zamanlayıcı kesmesi ve bağlam değiştirme. Takılı politika "dilim = 4 ms"dir; yerine "dilim = 10 ms" takılabilir.

İkisini ayırmak en yüksek esnekliği sağlar: politika kararları sonradan değiştiğinde mekanizmayı yeniden yazmak gerekmez. Zamanlayıcı kesmesi ve bağlam değiştirme, işlemcinin süreçler arasında nasıl döndürüleceğini sağlayan mekanizmadır. Zaman diliminin 4 ms mi, 10 ms mi olacağı ise bir politikadır. Şekil 1.29'daki gibi dilim değeri mekanizmaya takılan bir parametre olarak tasarlanırsa, aynı çekirdek kodu masaüstünde kısa dilimle, sunucuda uzun dilimle çalışabilir. Linux'ta bir sürecin çizelgeleme sınıfı `sched_setscheduler()` çağrısıyla, bir diskin G/Ç çizelgeleyicisi ise `/sys` altındaki bir dosyaya yazılarak sistem çalışırken değiştirilebilir; çekirdek kodu değişmez.

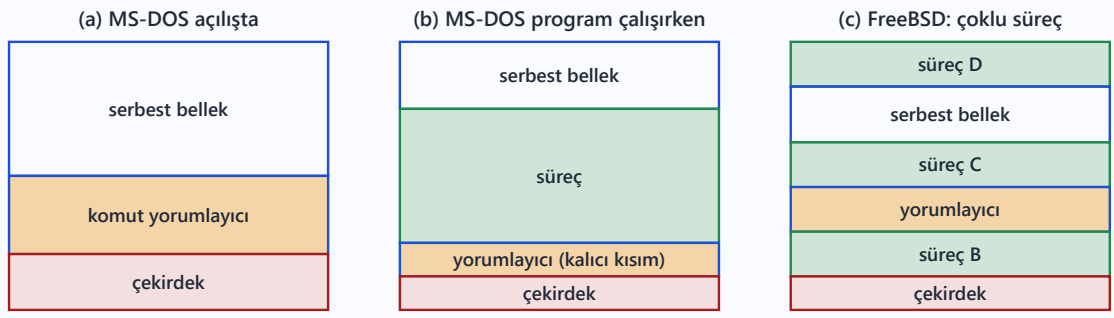
Mekanizma	Politika
Zamanlayıcı kesmesi	Zaman dilimi kaç ms olmalı?
Öncelikli çizelgeleyici	Hangi süreç yüksek öncelikli?
Dosya izin bitleri (rwx)	Kime hangi izin verilecek?
Sayfa yer değiştirme altyapısı	Hangi sayfa dışarı atılacak?
cgroups	Hangi konteynere ne kadar CPU ve bellek?

Tablo 1.9: Mekanizma ve politika örnekleri.

Tablodaki her satırda soldaki sütun bir "araç", sağdaki sütun bu aracın nasıl kullanılacağına dair bir "karar"dır. Dosya izin bitleri, her erişimde izni denetleyen mekanizmayı oluşturur; hangi kullanıcıya okuma, hangisine yazma izni verileceği ise yöneticinin politikasıdır. Bir ayrımın politika mı, mekanizma mı olduğunu anlamanın kolay yolu şu soruyu sormaktır: "Bu karar değişirse kodun değişmesi gerekir mi?" Gerekmiyorsa o karar bir politikadır. İşletim sistemi tasarımı bu dengeleri kurmak için mühendislik kadar yaratıcılık da gerektirir.

1.8.2 MS-DOS ve FreeBSD

Bir programın nasıl başlatıldığı, işletim sisteminin yapısı hakkında çok şey söyler. Tek görevli MS-DOS ile çok görevli bir UNIX türevi olan FreeBSD bu açıdan iyi bir karşıtlık oluşturur.



Şekil 1.30: Bellek düzenleri. (a) MS-DOS açılışta: altta çekirdek, üstünde komut yorumlayıcı, en üstte serbest bellek. (b) MS-DOS program çalışırken: süreç, yorumlayıcının geçici kısmının üzerine yüklenmiş, yalnız kalıcı kısım kalmıştır. (c) FreeBSD: çekirdeğin üstünde B, C, D süreçleri ve yorumlayıcı aynı anda bellektedir.

MS-DOS tek görevli bir sistemdir ve tek bir bellek alanı kullanır. Açılışta kabuk, yani COMMAND.COM, belleğe yüklenir. Kullanıcı bir program çalıştırınca program, kabuğun geçici kısmının üzerine yazılarak yüklenir; kabuğun yalnız küçük kalıcı kısmı bellekte kalır. Program bitince kabuk kendini diskten yeniden yükler. Aynı anda tek program çalışır; çoklu görev ve bellek koruması yoktur.

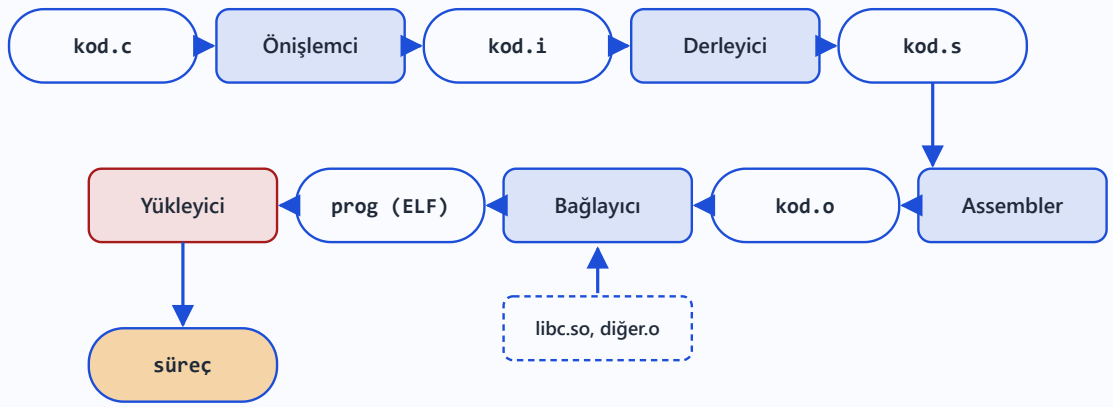
FreeBSD çok görevlidir. Kullanıcı giriş yapınca bir kabuk başlar. Kullanıcı bir komut yazınca kabuk önce `fork()` sistem çağrısıyla kendisinin bir kopyası olan yeni bir süreç yaratır. Bu yeni süreç `exec()` ile istenen programı kendi belleğine yükler. Kabuk ya programın bitmesini bekler ya da ko-

mutun sonuna `&` konmuşsa programı arka planda çalıştırıp yeni komut almaya devam eder. Şekil 1.30'daki (c) düzeninde olduğu gibi pek çok süreç aynı anda bellekte bulunur ve her biri kendi bölgesinde korunur.

1.8.3 Çalıştırılabilir dosya oluşturma

Bir C kaynak dosyasının çalışan bir sürece dönüşmesi birkaç ayrı programın sırayla çalışmasıyla olur. GCC derleyici sürücüsü bu adımların her birini ayrı ayrı yaptırmaya izin verir:

```
$ gcc -E kod.c -o kod.i
$ gcc -S kod.i -o kod.s
$ gcc -c kod.s -o kod.o
$ gcc kod.o diger.o -o prog
$ ./prog
```



Şekil 1.31: C kaynağından çalışan sürece. `kod.c` önişlemciden geçip `kod.i`, derleyiciden geçip `kod.s`, assembler'dan geçip `kod.o` olur. Bağlayıcı `kod.o`'yu `libc.so` ve `diger.o` ile birleştirip `prog` (ELF) dosyasını üretir. Yükleyici bu dosyadan bir süreç yaratır.

1. **Kaynak dosya**: `kod.c`, insanın okuyup yazdığı C kodudur.
2. **Önişlemci** (`gcc -E`): `#include` satırlarındaki başlık dosyalarını metne ekler, makroları genişletir, `#ifdef` ile koşullu derleme bloklarını çözer. Sonuç yine C kodudur: `kod.i`.
3. **Derleyici** (`gcc -S`): C kodunu hedef işlemcinin assembly diline çevirir ve en iyileme yapar: `kod.s`.
4. **Assembler** (`gcc -c`): assembly'yi makine koduna çevirir ve bir *nesne dosyası* üretir: `kod.o`. Bu dosyada `printf` gibi dışarıdaki fonksiyonların adresleri henüz belli değildir; yalnız "burada `printf` çağrılıyor" diye not düşülmüştür.
5. **Bağlayıcı** (`gcc kod.o diger.o -o prog`): nesne dosyalarını ve kütüphaneleri birleştirir, açık kalan adresleri çözer ve **çalıştırılabilir dosyayı** üretir. Linux'ta bu dosyanın biçimi ELF, Windows'ta PE'dir (.exe).

6. **Yükleme** (`./prog`): kabuk `fork` ve `exec` yapar. Çekirdeğin yükleyicisi ELF dosyasını belleğe eşler, dinamik bağlayıcı `libc.so` gibi paylaşılan kütüphaneleri bağlar ve program bir süreç olarak çalışmaya başlar.

Bu zincirde işletim sisteminin payı yalnız son adımdadır. İlk dört adımı kullanıcı modunda çalışan sıradan programlar yapar: önişlemci, derleyici, assembler ve bağlayıcı. 1.2.4'te belirtildiği gibi programı makine diline derleyici çevirir. İşletim sistemi, ortaya çıkan dosyayı bir sürece dönüştürür: bellek ayırır, kodu ve veriyi bu belleğe eşler, yığını kurar ve işlemciyi programın ilk komutuna yönlendirir.

> **Sık yapılan hata**
>
> Program ile süreci aynı şey sanmak. Program, diskte duran edilgen bir dosyadır (`prog`). Süreç ise o programın çalışan bir örneğidir: kendi belleği, yazmaçları ve açık dosyaları vardır. Aynı `prog` dosyasından aynı anda üç süreç başlatılabilir; 1.3.9'daki `yaz` deneyi tam olarak bunu yapar.

1.8.4 Metrik birimler

İşletim sistemi konularında hem çok küçük süreler (nanosaniyeler) hem de çok büyük boyutlar (terabaytlar) geçer. Tablo 1.10 ondalık önekleri, Tablo 1.11 ikili önekleri verir.

Üs	Ad	Sembol	Üs	Ad	Sembol
10^{-3}	mili	m	10^{3}	kilo	k
10^{-6}	mikro	µ	10^{6}	mega	M
10^{-9}	nano	n	10^{9}	giga	G
10^{-12}	piko	p	10^{12}	tera	T
10^{-15}	femto	f	10^{15}	peta	P
10^{-18}	atto	a	10^{18}	eksa	E

Tablo 1.10: Ondalık (SI) önekler.

Birim	Değer
1 KiB (kibibayt)	2^{10} = 1024 B
1 MiB (mebibayt)	2^{20} B
1 GiB (gibibayt)	2^{30} B
1 TiB (tebibayt)	2^{40} B

Tablo 1.11: İkili (IEC) önekler.

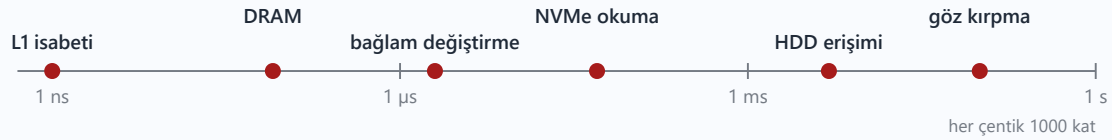
İki önek ailesinin bir arada kullanılması sık karışıklığa yol açar. Bellek genellikle ikili ölçülür: “16 GB RAM” denen bellek gerçekte 16 GiB, yani $16 \times 2^{30} = 17.179.869.184$ bayttır. Disk ve ağ ise ondalık ölçülür: $1\ TB = 10^{12}$ B ve $1\ Gb/s = 10^{9}$ bit/s. Bayt için büyük B, bit için küçük b kullanılır; 1 Gb/s'lik bir bağ-

lantı saniyede en çok $10^9 / 8 = 125$ milyon bayt taşır.

Örnek 1.8

"512 GB" etiketli bir SSD, boyutları ikili önekle gösteren bir işletim sisteminde kaç GiB görünür? 1 TB kaç TiB'dir?

Çözüm. Etiket ondalıktır: $512\ GB = 512 \times 10^9$ B. Bir GiB $2^{30} = 1.073.741.824$ bayttır. Bölüm $512 \times 10^9 / 2^{30} \approx 476{,}8$ GiB'dir. Aynı yoldan $1\ TB = 10^{12} / 2^{40} \approx 0{,}91$ TiB bulunur. Diskte eksik bir şey yoktur; aynı bayt sayısı iki farklı birimle gösterilmiştir. Fark birim büyüdükçe artar: kilo düzeyinde %2,4 olan fark, tera düzeyinde yaklaşık %9'a çıkar.



Şekil 1.32: Logaritmik zaman ekseninde tipik gecikmeler; her çentik 1000 kattır. Soldan sağa: L1 isabeti (yaklaşık 1 ns), DRAM erişimi, bağlam değiştirme (mikrosaniyeler), NVMe okuma, HDD erişimi (milisaniyeler) ve göz kırpma (onlarca, yüzlerce milisaniye).

Şekil 1.32, bu notta geçen sürelerin ölçeğini tek bakışta gösterir. Bir L1 önbellek isabeti ile bir sabit disk erişimi arasında yaklaşık on milyon kat vardır. İnsan algısı ise bunların hepsinin ötesindedir: bir göz kırpma süresinde işlemci yüz milyonlarca komut yürütür. İşletim sisteminin bir kullanıcıya "aynı anda" çok sayıda program çalıştırıyormuş izlenimi verebilmesi, bu ölçek farkı sayesindedir.

1.9 Özet

- İşletim sistemi, donanımı yöneten ve uygulamalara basit, güvenli ve ortak bir arayüz sunan yazılım katmanıdır. Dar anlamda işletim sistemi, ayrıcalıklı modda çalışan çekirdektir; kabuk ve GUI sıradan kullanıcı programlarıdır.
- Koruma, işlemcideki mod bitine dayanır. Kullanıcı modunda ayrıcalıklı komutlar yasaktır; çekirdeğe yalnız sistem çağrısı, kesme ve istisna kapılarından, önceden belirlenmiş giriş noktalarına geçilir.
- Bir `read()` çağrısında libc sarmalayıcısı numarayı ve argümanları yazmaçlara koyar, `syscall` tuzak üretir, çekirdek tablodan `sys_read()`'i bulup argümanları denetler, `sysret` kullanıcı moduna döner.
- İşletim sistemi yukarıdan bakınca genişletilmiş makinedir: çirkin donanımı süreç, adres alanı, dosya ve soket gibi soyutlamalara dönüştürür. Aşağıdan bakınca kaynak yöneticisidir: kaynakları zamanda ya da alanda paylaştırır, verimi ve adaleti dengeler.

- Biriktirme, paylaşılamayan yavaş bir aygıtı tek bir hizmet sürecinin tekeline verir ve G/Ç ile hesaplamayı örtüştürür. Dairesel kuyrukta `giris` ile `cikis`'in eşitliği boş ile doluyu ayırmaz; bu yüzden `adet` sayacı tutulur.
- Toplu iş, işler arasındaki el ile geçişi kaldırdı. Çoklu programlama, bir iş G/Ç beklerken CPU'yu başka bir işe vererek CPU kullanımını artırdı; örnekte iki iş 15 yerine 9 birimde bitti ve kullanım %53'ten %89'a çıktı. Zaman paylaşımı, zamanlayıcı kesmesiyle her kullanıcıya kısa tepki süresi sağladı.
- Bağlam değiştirmede bütün yazmaçlar kaydedilir, çünkü işlemcide tek yazmaç takımı vardır. Çok çekirdekli yongalar paralellik, senkronizasyon ve önbellek tutarlılığı sorunları getirir; SMT her çekirdeği iki mantıksal CPU gibi gösterir.
- Bellek hiyerarşisi, yerellik sayesinde neredeyse üst düzeyin hızında ve alt düzeyin kapasitesinde çalışır. Disk beklemesi milyonlarca saat turu sürdüğü için işletim sistemi bu sürede başka bir süreci çalıştırır.
- Yoklamada CPU meşgul bekler; kesmede her iş birimi için bir kesme gelir; DMA'da denetleyici bütün bloğu taşır ve tek kesme üretir. Kesme dışarıdan ve eş zamansız, tuzak komutun kendisinden ve eş zamanlı doğar; ikisi de çekirdeğe aynı yolla geçer.
- İşletim sistemleri anabilgisayardan akıllı karta kadar çok çeşitlidir. Gerçek zamanlı sistemlerde önemli olan hız değil, öngörülebilirliktir; katı sistemde son tarihi kaçırmak felakettir, yumuşak sistemde kaliteyi düşürür.
- Politika ne yapılacağını, mekanizma nasıl yapılacağını belirler; ikisi ayrılırsa politika çekirdek kodu değişmeden değiştirilebilir.
- C kaynağını önişlemci, derleyici, assembler ve bağlayıcı çalıştırılabilir dosyaya çevirir; işletim sistemi bu dosyayı yükleyip bir sürece dönüştürür. Bellek ikili (GiB), disk ve ağ ondalık (GB, Gb/s) ölçülür.

1.10 Kavram listesi

işletim sistemi (İS) *operating system (OS)* · çekirdek *kernel* · kabuk *shell* · GUI (grafik kullanıcı arayüzü) *graphical user interface* · sistem çağrısı *system call* · çekirdek modu *kernel mode* · kullanıcı modu *user mode* · çift modlu çalışma *dual-mode operation* · tuzak *trap* · istisna *exception* · kesme *interrupt* · kesme işleyicisi *interrupt handler* · soyutlama *abstraction* · adres alanı *address space* · sanallaştırma *virtualization* · sanal adres *virtual address* · eş zamanlılık *concurrency* · paralellik *parallelism* · yarış durumu *race condition* · biriktirme *spooling* · toplu iş sistemi *batch system* · çoklu programlama *multiprogramming* · zaman paylaşımı *time sharing* · çizelgeleyici *scheduler* · verim *throughput* · tepki süresi *response time* · yazmaç *register* · program sayacı *program counter (PC)* · getir–çöz–yürüt döngüsü *fetch–decode–execute cycle* · bağlam değiştirme *context switch*

çok çekirdekli *multicore* · eş zamanlı çoklu iş parçacığı (SMT) *simultaneous multithreading* · önbellek tutarlılığı *cache coherence* · bellek hiyerarşisi *memory hierarchy* · yerellik *locality* · bellek yönetim birimi (MMU) *memory management unit* · veri yolu *bus* · aygıt denetleyicisi *device controller* · aygıt sürücüsü *device driver* · yoklama *polling* · meşgul bekleme *busy waiting* · doğrudan bellek erişimi (DMA) *direct memory access* · gerçek zamanlı işletim sistemi *real-time operating system (RTOS)* · politika / mekanizma *policy / mechanism* · çalıştırılabilir dosya *executable file* · hipervizör *hypervisor* · konteyner *container*

1.11 Kendini Dene

1. Aşağıdaki olaylardan hangileri işlemcinin kullanıcı modundan çekirdek moduna geçmesine yol açar? (a) `getpid()` çağrısı, (b) `i++`, (c) sayfa hatası, (d) ağ kartına paket gelmesi, (e) kullanıcı belleğinde `memcpy`.

   Cevap

   (a), (c) ve (d) geçer. `getpid()` bir sistem çağrısıdır; libc sarmalayıcısı `syscall` ile tuzak üretir. Sayfa hatası, bellek erişimi yapan komuttan doğan bir istisnadır. Paket gelmesi, ağ kartından gelen bir donanım kesmesidir. `i++` yazmaç ve bellek üzerinde sıradan bir işlem, `memcpy` ise kullanıcı belleğinde kopyalama yapan bir kütüphane fonksiyonudur; ikisi de kullanıcı modunda kalır.

2. Tek CPU'lu bir sistemde A işi 1 birim CPU, 3 birim G/Ç, 1 birim CPU; B işi 2 birim CPU, 1 birim G/Ç, 2 birim CPU kullanır. Tek programlı ve çoklu programlı sistemde iki iş ne zaman biter, CPU kullanımı nedir? (Çoklu programlamada CPU boşalınca hazır işlerden önce A seçilir.)

   Cevap

   Tek programlı: A 0–1, boşta 1–4, A 4–5; B 5–7, boşta 7–8, B 8–10. Bitiş $t = 10$, CPU kullanımı $6 / 10 = \%60$. Çoklu programlı: A 0–1; A G/Ç'ye gidince B 1–3; B G/Ç'ye gider (3–4), A'nın G/Ç'si de 4'te bittiği için CPU 3–4 arasında boştadır; $t = 4$'te iki iş de hazırdır, A 4–5, B 5–7. Bitiş $t = 7$, CPU kullanımı $6 / 7 \approx \%85{,}7$.

3. Biriktirme kodunda kuyruk doluyken (a.pdf, b.txt, c.doc, d.png) bir program `ekle("e.jpg")` çağı-

rırsa ne olur? Ardından `biriktirme_hizmeti()` çalışırsa hangi dosyalar hangi sırayla basılır?

Cevap

`ekle()` doluluk denetimi yapmaz. `giris = 0` olduğu için e.jpg 0. yuvaya, a.pdf'in üzerine yazılır; `giris = 1`, `adet = 5` olur. Hizmet döngüyü 5 kez döner: `cikis` 0, 1, 2, 3, 0 değerlerini alır. Sırayla e.jpg, b.txt, c.doc ve d.png basılır; her biri gönderildikten sonra yuvası `NULL` yapılır. Beşinci turda `cikis` yeniden 0'dır ve bu yuva artık boştur: yazıcıya `NULL` gönderilir. a.pdf kaybolmuş, sayaç ise var olmayan bir işi saymıştır. Doğru bir `ekle()`, `adet == 4` iken işi reddetmeli ya da beklemelidir.

4. 4 GHz'de çalışan bir çekirdekte 60 ns'lik bir DRAM erişimi ve 50 µs'lik bir NVMe okuması kaç saat turu sürer? İşletim sistemi bu iki bekleme için neden farklı davranır?

Cevap

Bir tur *0,25 ns*'dir; 1 ns'de 4 tur geçer. DRAM: *60 × 4 = 240* tur. NVMe: *50 µs = 50.000 ns*, *50.000 × 4 = 200.000* tur. DRAM beklemesi bağlam değiştirmeden kısadır ve önbelleklerle gizlenir; işletim sistemi araya girmez. 200.000 turluk SSD beklemesinde ise süreç bekletilir ve CPU başka bir sürece verilir.

5. Bir aygıttan 16 KiB veri alınıyor. Kesme güdümlü G/Ç'de her 16 bayt için bir kesme oluşuyor ve her kesme CPU'ya 2 µs'ye mal oluyor. DMA ile programlama 5 µs, sondaki tek kesme 2 µs sürüyor. İki yöntemde CPU'nun harcadığı süre nedir?

Cevap

16 KiB = 16 × 1024 = 16.384 bayt. Kesme sayısı *16.384 / 16 = 1024*; CPU süresi *1024 × 2 = 2048 µs*. DMA: *5 + 2 = 7 µs*. Oran yaklaşık *2048 / 7 ≈ 293*'tür.

6. Bir işlemcide SMT destekli 4 çekirdek ve SMT desteği olmayan 4 çekirdek vardır. İşletim sistemi kaç mantıksal CPU görür? Bu CPU'ların hepsi eşit güçte midir?

Cevap

4 × 2 + 4 = 12 mantıksal CPU görünür. Eşit güçte değildir: aynı çekirdeği paylaşan iki mantıksal CPU o çekirdeğin yürütme birimlerini ve önbelleğini paylaşır, bu yüzden iki ayrı çekirdek kadar hızlı çalışmaz.

7. “256 GB” etiketli bir USB bellek, ikili önek kullanan bir işletim sisteminde yaklaşık kaç GiB görü-

nür?

Cevap

$256 \times 10^9 / 2^{30} \approx 238{,}4$ GiB. Etiket ondalık, işletim sisteminin gösterimi ikilidir; bayt sayısı aynıdır.

8. Aşağıdakilerden hangileri mekanizma, hangileri politikadır? (a) Bağlam değiştirme kodu, (b) etkileşimli süreçlere daha kısa zaman dilimi verilmesi, (c) sayfa tablosu girdisindeki geçerli bitin donanımca denetlenmesi, (d) bellek dolunca en uzun süredir kullanılmayan sayfanın dışarı atılması.

Cevap

(a) ve (c) mekanizmadır: bir işin nasıl yapılacağını sağlarlar. (b) ve (d) politikadır: "kime ne kadar dilim?" ve "hangi sayfa atılacak?" sorularına verilen kararlardır. Aynı mekanizmalar başka politikalarla da kullanılabilir.

9. Bir kalp pili, bir müzik akışı uygulaması ve bir web sunucusu: hangisi katı, hangisi yumuşak gerçek zamanlıdır, hangisi genel amaçlıdır?

Cevap

Kalp pili katı gerçek zamanlıdır: uyarının zamanında verilmemesi hayati sonuç doğurur. Müzik akışı yumuşak gerçek zamanlıdır: arada bir gecikme kısa bir kesintiye yol açar, ama tehlikeli değildir. Web sunucusu genel amaçlıdır; ortalama başarımı ve adaleti öne çıkarır, katı bir son tarihi yoktur.

10. Zaman paylaşımı, çoklu programlamadan hangi donanım özelliği sayesinde ayrılır? Hiç G/Ç yapmayan bir iş iki sistemde nasıl davranır?

Cevap

Zamanlayıcı kesmesi sayesinde. Çoklu programlamada CPU bir işten ancak iş beklemeye girince alınır; hiç G/Ç yapmayan bir iş CPU'yu bitene kadar tutar. Zaman paylaşımında zamanlayıcı her dilimin sonunda kesme üretir ve çekirdek CPU'yu bu işten alıp sıradakine verir; böylece öteki kullanıcıların tepki süresi kısa kalır.

1.12 İleri okuma

- A. S. Tanenbaum ve H. Bos, *Modern Operating Systems* (5. baskı, 2023), Bölüm 1: işletim sisteminin iki rolünü, nesilleri, IBM 1401/7094 toplu iş düzenini ve İS çeşitlerini bu notla aynı sırayla ve bol tarihî ayrıntıyla anlatır.

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Bölüm 2: sanallaştırma, eş zamanlılık ve kalıcılık deneylerinin C kodlarını verir; kitap çevrim içi olarak ücretsiz okunabilir.
- A. Silberschatz, P. B. Galvin ve G. Gagne, *Operating System Concepts* (10. baskı, 2018), Bölüm 1: kesmeler, bellek hiyerarşisi, DMA ve çift modlu çalışmayı donanım açısından ayrıntılı işler.
- F. P. Brooks, *The Mythical Man-Month* (1975; 20. yıl baskısı 1995): OS/360 projesinden çıkarılan yazılım mühendisliği derslerini anlatan klasik deneme kitabı.

1.13 Kaynaklar

1. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, sürüm 1.10, pages.cs.wisc.edu/~remzi/OSTEP.
2. F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, 20. yıl baskısı, Addison-Wesley, 1995.
3. Linux kılavuz sayfaları: syscall(2), fork(2), execve(2), proc(5).
4. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018.
5. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı, Pearson, 2014; 5. baskı, 2023.

KISIM I · BÖLÜM 2

İşletim Sistemi Yapıları

Bir işletim sistemi tek parça bir program gibi görünür. Oysa içinde süreç yönetimi, bellek yönetimi, dosya sistemi, aygıt sürücüleri ve ağ yığını gibi onlarca bileşen vardır. Bu bileşenlerin hangi modda çalışacağı ve birbirleriyle nasıl konuşacağı, sistemin hızını, güvenilirliğini ve bakımını belirler. Not, işletim sisteminin kullanıcılara ve programlara sunduğu hizmetlerle başlar. Ardından süreç, adres alanı, dosya, bağlama, boru ve kilitlenme gibi temel kavramları tanıtır. Donanımın işletim sistemini nasıl koruduğu, programların sistem çağrılarıyla çekirdekten nasıl hizmet istediği ve temel POSIX çağrıları kod izlemeleriyle adım adım anlatılır. Son kesimler monolitik, katmanlı, mikro çekirdek, modüler, hibrit ve exokernel yapıları karşılaştırır; sanal makinelere, bağlayıcı ile yükleyiciye ve bilgisayarın açılış sürecine değinir.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum02` · **Laboratuvar:** `github.com/sercankulcu/operating-systems-labs/lab02`

ÖĞRENME ÇIKTILARI

Hizmetler ve kavramlar
İşletim sisteminin sunduğu hizmetleri; süreç, adres alanı, dosya, bağlama ve boru hattı kavramlarını açıklamak.

Sistem çağrıları
Kullanıcı modundan çekirdek moduna geçişi, parametre aktarımını ve temel POSIX çağrılarını adım adım izlemek.

Çekirdek yapıları
Monolitik, katmanlı, mikro çekirdek, modüler, hibrit ve exokernel yapılarını güçlü ve zayıf yönleriyle karşılaştırmak.

Sanallaştırma ve açılış
Sanal makine ve konteyner kavramlarını, bağlayıcı ile yükleyicinin işini ve açılış adımlarını tanımak.

2.1 Yapı ve hizmetler

Bir işletim sistemini iki açıdan incelemek mümkündür. Dışarıdan bakan biri, sistemin ne iş gördüğünü sorar: hangi hizmetleri sunar, programlar bu hizmetlere nasıl ulaşır? İçeriden bakan biri ise sistemin nasıl kurulduğunu sorar: bileşenler nerede çalışır, birbirlerini nasıl çağırır? Bu kesim iki soruyu da kısaca ele alır. Önce "yapı" sözcüğüyle ne kastedildiğini, sonra işletim sisteminin dışarıya sunduğu hizmetleri tanıtır.

2.1.1 Yapı neden önemlidir?

Günümüzün bir işletim sistemi tek başına küçük bir program değildir. Süreçleri oluşturan ve sonlandıran süreç yöneticisi, belleği paylaştıran bellek yöneticisi ve diskteki verileri dosya olarak sunan dosya sistemi bunlardan yalnız üçüdür. Her aygıtın ayrı bir sürücüsü, ağ paketlerinin de kendi yığını vardır. Linux çekirdeğinin kaynak kodu 2025 yılında 40 milyon satırı geçmiştir. Bu büyüklükte bir yazılımı tek bir düzensiz yığın olarak yazmak, anlamak ve bakımını yapmak mümkün değildir.

Tasarımcı her bileşen için iki karar verir. Birincisi, bileşenin **nerede** çalışacağıdır: donanımın bütün yetkilerini kullanan çekirdek modunda mı, yoksa yetkileri kısıtlı kullanıcı modunda mı? İkincisi, bileşenlerin **nasıl haberleşeceğidir**: biri ötekini sıradan bir fonksiyon gibi mi çağıracak, yoksa iki bileşen birbirine ileti mi gönderecek?

> **Tanım 2.1**
>
> İşletim sistemi bileşenlerinin nerede çalışacağına (çekirdek modu ya da kullanıcı modu) ve nasıl haberleşeceğine (fonksiyon çağrısı ya da ileti) dair genel stratejiye **işletim sistemi yapısı** denir.

Yapı seçimi, sistemin günlük hayatını doğrudan etkiler. Karmaşıklığın ne kadar iyi yönetildiği ve bir değişikliğin kaç dosyaya dokunduğu bu seçime bağlıdır. Bir hatanın kaç satır kod içinde aranacağı ve sistemin başka bir işlemciye ne kadar kolay taşınacağı da öyle. Ne var ki hiçbir yapı her ölçütte en iyisi değildir. Şekil 2.1 üç ana hedefi bir üçgenin köşelerine yerleştirir: performans, güvenilirlik ve yalıtım, esneklik ve bakım. Bir köşeye yaklaşan tasarım, genellikle öteki köşelerden uzaklaşır.



Şekil 2.1: Yapı seçimindeki ödünleşim. Bileşenleri aynı modda toplayıp doğrudan çağırmak performansı artırır; onları ayrı süreçlere bölmek güvenilirliği ve yalıtımı artırır; açık arayüzler ve değiştirilebilir parçalar esnekliği ve bakımı kolaylaştırır.

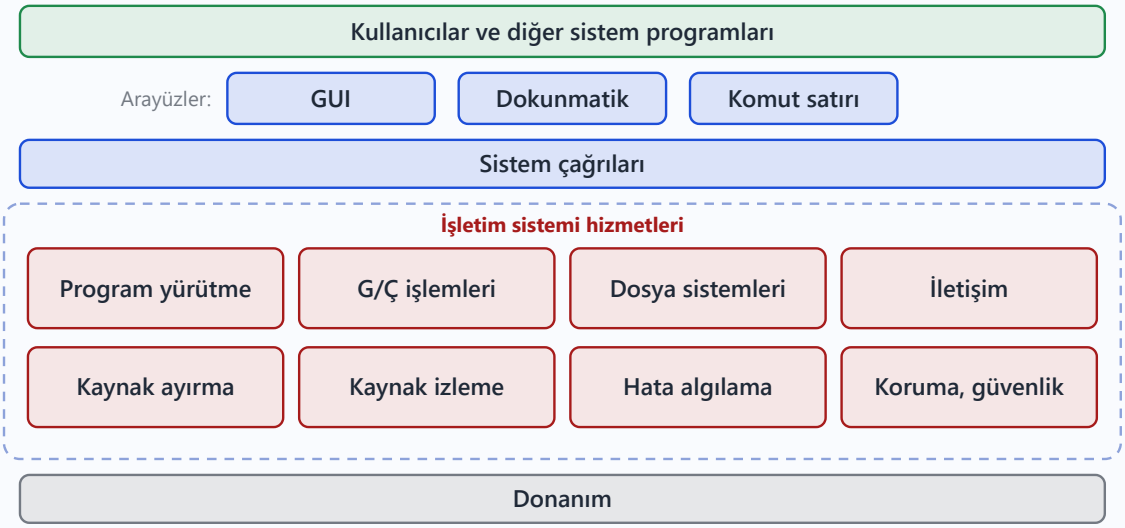
Bir örnek, bu ödünleşimi somutlaştırır. Bir dosya sistemi kodu, çekirdeğin içinde öteki bileşenlerle birlikte çalışıyorsa bir dosyayı okumak birkaç fonksiyon çağrısı kadar ucuzdur. Aynı kod ayrı bir süreç olarak kullanıcı modunda çalışıyorsa her istek bir ileti ve birkaç mod geçişi gerektirir; okuma pahalılaşır. Buna karşılık dosya sistemi kodundaki bir hata, ikinci durumda yalnız o süreci çökertir, birinci durumda ise bütün çekirdeği düşürebilir. Masaüstü bir bilgisayar, bir veri merkezi sunucusu, bir gömülü cihaz ve bir uçağın uçuş denetim bilgisayarı bu dengeyi farklı noktalarda kurar. Tek doğru yapının olmaması bundandır.

Sık yapılan hata

Yapıyı, kullanıcının gördüğü arayüzle karıştırmak. İki işletim sistemi aynı sistem çağrılarını, aynı komutları ve aynı disk biçimini sunup yine de çok farklı yapıda olabilir. Yapı, sistemin içindeki iş bölümüyle ilgilidir; dışarıdan bakan kullanıcı çoğu zaman bu farkı görmez.

2.1.2 İşletim sistemi hizmetleri

İşletim sistemi, üzerinde çalışan programlara bir çalışma ortamı hazırlar ve bu ortamın parçası olan bir dizi **hizmet** (*operating system services*) sunar. Hizmetlerin ayrıntısı sistemden sisteme değişir, ama ana gruplar hemen her sistemde aynıdır. Şekil 2.2 bu grupları katmanlar hâlinde gösterir.



Şekil 2.2: İşletim sistemi hizmetlerinin katmanlı görünümü (Silberschatz, Şekil 2.1'den uyarlanmıştır). En üstte kullanıcılar ve sistem programları, altında grafik, dokunmatik ve komut satırı arayüzleri yer alır. Arayüzler ve programlar hizmetlere sistem çağrıları üzerinden ulaşır. Hizmetlerin altında donanım bulunur.

Şekildeki en önemli ayrıntı, katmanlar arasındaki geçiş noktasıdır. Kullanıcı ister bir pencereye tıklasın ister komut satırına komut yazsın, yaptığı iş sonunda bir ya da birkaç **sistem çağrısına** dönüşür. Arayüz, kullanıcının isteğini sistem çağrılarına çeviren bir aracıdır. Hizmetlerin kendisi sistem çağrısı katmanının altında, işletim sisteminin içindedir ve donanımı doğrudan yönetir.

2.1.3 Hizmetler kime hizmet eder?

Hizmetleri, onlardan yararlanan tarafa göre iki gruba ayırmak alışılmış bir yoldur. Birinci grup, kullanıcıya ve programlara doğrudan yardım eder:

- **Arayüz.** Kullanıcı sistemle komut satırı, grafik kullanıcı arayüzü (*graphical user interface*, GUI), dokunmatik ekran ya da sesli komutla konuşur.
- **Program yürütme.** Sistem bir programı belleğe yükler, çalıştırır ve sonlandırır. Programın normal biçimde mi yoksa bir hatayla mı bittiğini de kaydeder.
- **Giriş/çıkış (G/Ç) işlemleri.** Program bir aygıta doğrudan erişmez; isteğini işletim sistemine iletir. Böylece hem verimlilik hem koruma sağlanır.
- **Dosya sistemi.** Dosya oluşturma, silme, okuma, yazma, arama ve izinleri yönetme işleri bu hizmetin kapsamındadır.
- **İletişim.** Süreçler aynı makinede ya da ağ üzerinden bilgi alışverişi yapar. Bunun iki temel yolu paylaşımlı bellek ve ileti geçirmedir.
- **Hata algılama.** Bellek hatası, aygıt arızası, sıfıra bölme ya da yetkisiz bellek erişimi gibi durumlar algılanır ve uygun biçimde ele alınır.

İkinci grup, kullanıcıya doğrudan görünmez; sistemin bir bütün olarak verimli ve güvenli çalışmasını sağlar:

- **Kaynak ayırma.** İşlemci (CPU), bellek ve depolama birçok süreç arasında paylaştırılır.
- **Kaynak izleme** (*accounting*). Hangi kullanıcının ya da sürecin ne kadar kaynak kullandığı kaydedilir; günlük kayıtları tutulur.
- **Koruma.** Süreçler birbirine ve çekirdeğe zarar veremez.
- **Güvenlik.** Kimlik doğrulama, yetkilendirme ve dış saldırılara karşı savunma bu hizmetin işidir.

Kullanıcıların çoğu işletim sistemini sistem çağrıları üzerinden değil, **sistem programları** üzerinden tanır. Dizin içeriğini listeleyen `ls`, dosya kopyalayan `cp`, çalışan süreçleri gösteren `ps`, paket yöneticisi, derleyici ve dosya yöneticisi bu tür programlardır. Bunlar sıradan kullanıcı programlarıdır, ama işlerini yapmak için yoğun biçimde sistem çağrısı kullanırlar. Kullanıcının gözünde "işletim sistemi", çoğu zaman bu programların toplamıdır.

Örnek 2.1

Bir elektronik tablo programında bir formülün yeniden hesaplanması bir işletim sistemi hizmeti midir?

Çözüm. Değildir. Formülü hesaplamak programın kendi mantığıdır; işlemci bu hesabı programın kendi kodunu yürüterek yapar. İşletim sistemi bu sırada programa yalnız işlemci zamanı ve bellek sağlar. Tablo diske kaydedilirken ise durum değişir: programın diske doğrudan erişme yetkisi yoktur, dosya sistemi hizmetini sistem çağrılarıyla kullanır. Aynı programın bir işi kendi kodudur, öteki işi işletim sistemi hizmetidir.

2.2 Temel kavramlar

İşletim sistemleri, donanımın karmaşık ayrıntılarını birkaç temel soyutlamanın arkasına gizler. Bu soyutlamalar o kadar yaygındır ki işletim sistemini anlatan her metin aynı kavramlara dayanır: süreç, adres alanı, dosya, dizin, bağlama, boru ve kilitlenme. Kesimin amacı bu kavramlara ilk bir tanım vermektir. Her birinin ayrıntısı, ilgili kavramın kendi bölümünde derinleştirilir; burada önemli olan, kavramların birbiriyle nasıl ilişkilendiğini görmektir.

2.2.1 Süreç

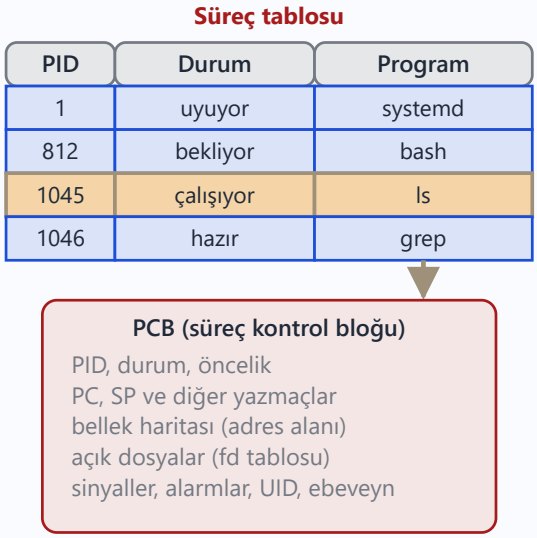
Diskteki bir program dosyası, kendi başına hiçbir şey yapmaz. Makine komutlarından ve başlangıç verilerinden oluşan pasif bir dosyadır. Program çalıştırıldığında işletim sistemi onu belleğe yükler, ona bir yığın ayırır, işlemciye ilk komutunu gösterir ve programı izlemeye başlar. Çalışmakta olan bu canlı örneğe süreç adı verilir.

Tanım 2.2

Süreç (*process*), yürütülmekte olan bir programdır. Program diskteki pasif dosyadır; süreç onun etkin örneğidir. Her süreçle birlikte bir adres alanı (kod, veri, yığın), yazmaçların değerleri (program sayacı ve yığın işaretçisi dahil), açık dosyalar, alarmlar, ilişkili süreçlerin listesi ve öteki kaynaklar tutulur.

Aynı programdan aynı anda birden çok süreç oluşturulabilir. İki ayrı terminalde `bash` çalıştıran bir kullanıcının makinesinde, diskteki tek `bash` dosyasından iki süreç vardır. İkisi aynı kodu yürütür, ama her birinin değişkenleri, açık dosyaları ve o anda bulunduğu komut farklıdır.

İşletim sistemi zaman zaman bir süreci askıya alıp başka bir süreci çalıştırmak zorunda kalır. Askıya alınan sürecin daha sonra kaldığı yerden sürmesi için o anki bütün durumunun bir yerde saklanması gerekir. Bu bilgiler **süreç tablosunda** tutulur. Tablonun her girdisi bir sürece aittir ve **süreç kontrol bloğu** (*process control block*, PCB) adını alır. Şekil 2.3 küçük bir süreç tablosunu ve çalışan sürecin PCB içeriğini gösterir.



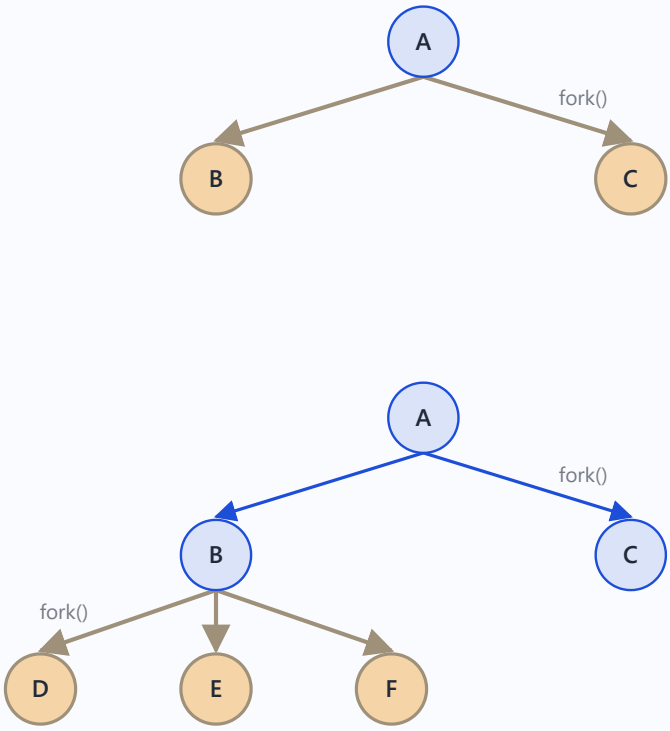
Şekil 2.3: Süreç tablosu ve bir süreç kontrol bloğu. Tabloda dört süreç vardır: PID 1 (`systemd`, uyuyor), 812 (`bash`, bekliyor), 1045 (`ls`, çalışıyor) ve 1046 (`grep`, hazır). Çalışan sürecin PCB'sinde PID, durum ve öncelik; program sayacı, yığın işaretçisi ve öteki yazmaçlar; bellek haritası; açık dosyalar tablosu; sinyaller, alarmlar, kullanıcı kimliği ve ebeveyn bilgisi bulunur.

Şekildeki her süreç bir **süreç kimliğiyle** (*process identifier*, PID) tanınır. Durum sütunu, sürecin o anda ne yaptığını gösterir. Tek işlemcili bir makinede aynı anda yalnız bir süreç çalışır. Öteki süreçler ya işlemci sırası bekler (hazır) ya da klavyeden bir tuşa basılması gibi bir olayı bekler. PCB'deki yazmaç değerleri, süreç işlemciden alınırken kaydedilir ve süreç yeniden çalışacağı zaman işlemciye geri yüklenir. Süreç, aradaki kesintiyi fark etmez.

Süreçler çoğu zaman tek başına çalışmaz; birbirleriyle veri alışverişi yapar. Bunun için işletim sisteminin sunduğu mekanizmaların ortak adı **süreçler arası iletişimdir** (*interprocess communication*, IPC). Boru, paylaşımlı bellek, ileti kuyruğu ve soket bu mekanizmalardan bazılarıdır.

2.2.2 Süreç ağacı

UNIX benzeri sistemlerde yeni bir süreç, var olan bir sürecin `fork()` sistem çağrısıyla oluşur. Çağrıyı yapan sürece **ebeveyn**, oluşan sürece **çocuk** denir (*parent / child process*). Çocuk da kendi çocuklarını oluşturabildiği için süreçler kendiliğinden bir ağaç biçiminde dizilir. Şekil 2.4 bu ağacın iki aşamasını gösterir.



Şekil 2.4: Süreç ağacının oluşması. Üstte A süreci iki kez `fork()` çağırmıştır: B ve C, A'nın çocuklarıdır. Altta B süreci de D, E ve F olmak üzere üç çocuk oluşturmuştur.

Başlangıçta yalnız A süreci vardır. A iki kez `fork()` çağırınca B ve C doğar; A ikisinin de ebeveynidir. Ardından B üç kez `fork()` çağırır ve D, E, F süreçleri oluşur. Bu ağaçta D'nin ebeveyni B, B'nin ebeveyni A'dır. Linux'ta bütün ağacın kökü, açılışta çekirdeğin başlattığı ve PID'si 1 olan `systemd` (eski sistemlerde `init`) sürecidir. Çalışan bir sistemde `pstree -p` komutu bu ağacı süreç kimlikleriyle birlikte yazdırır.

Sık yapılan hata

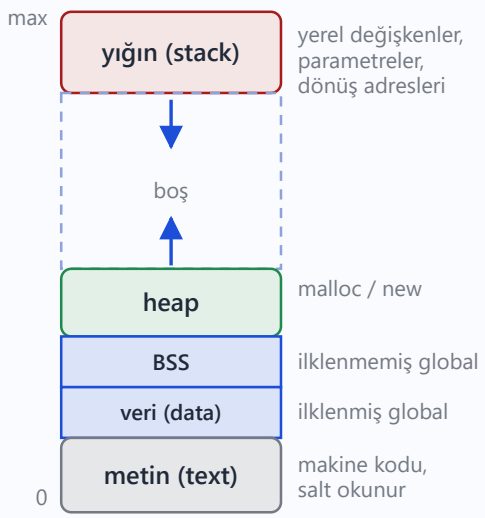
`fork()`'un yeni ve boş bir program başlattığını sanmak. `fork()` ebeveynin bir kopyasını oluşturur; çocuk, ebeveynle aynı kodu aynı noktadan yürütmeye devam eder. Çocuğun başka bir program çalıştırması için ayrıca `execve` çağrısı gerekir. Bu ikili, kabuk örneğiyle 2.5.2'de ayrıntılı olarak izlenir.

2.2.3 Adres alanı ve bellek kesimleri

Her süreç belleği kendi başına kullanıyormuş gibi görür. Sürecin kullanabildiği adreslerin kümesine onun **adres alanı** (*address space*) denir. Adres alanı 0'dan başlayıp bir üst sınıra kadar uzanır. x86-64 mimarisindeki Linux'ta bir sürecin kullanıcı alanı 128 TiB'tır; bu, çoğu makinenin fiziksel belleğinden çok daha büyüktür. Fiziksel bellekten büyük adres alanlarını mümkün kılan teknik **sanal bellektir**.

Bellekte aynı anda çok sayıda süreç bulunur. İşletim sistemi bu süreçleri hem birbirinden hem de kendisinden korumak zorundadır: bir sürecin hatalı bir işaretçisi başka bir sürecin verisini ya da çekirdeğin kodunu bozmamalıdır. Bu koruma, donanımdaki bellek yönetim birimi ve işletim sisteminin birlikte çalışmasıyla sağlanır.

Bir sürecin adres alanı birkaç kesime ayrılır. Şekil 2.5 tipik düzeni aşağıdan yukarıya doğru gösterir.



Şekil 2.5: Bir sürecin adres alanı. En altta (adres 0'a yakın) metin kesimi, onun üstünde ilklenmiş veri ve ilklenmemiş veri (BSS) kesimleri, sonra heap bulunur. En üstte yığın yer alır. Heap yukarı, yığın aşağı doğru büyür; aradaki boşluk ikisinin de büyümesine yer bırakır.

- **Metin.** Programın makine kodudur. Salt okunur tutulur; aynı programı çalıştıran süreçler ve bir sürecin bütün iş parçacıkları bu kesimi paylaşır.
- **Veri ve BSS.** Global ve `static` değişkenler burada durur. İlk değeri kaynak kodda verilmiş olanlar veri kesimine, verilmemiş olanlar BSS kesimine yerleşir.

- **Heap** (*heap*). Çalışma anında `malloc` ya da `new` ile ayrılan dinamik bellektir. Veri kesimi gibi heap de sürecin bütün iş parçacıklarınca paylaşılır.
- **Yığın** (*stack*). Fonksiyon parametreleri, yerel değişkenler ve dönüş adresleri burada tutulur. Her iş parçacığının kendi yığını vardır.

Örnek 2.2

Bir C programında `int sayac = 5;` ve `int tablo[1000];` global olarak, `main` içinde ise `int i;` ve `char *p = malloc(64);` tanımlanmıştır. Her nesne hangi kesimdedir?

Çözüm. `sayac` ilk değeri olan bir global değişkendir: veri kesimindedir. `tablo` ilk değer almamış bir global dizidir: BSS kesimindedir ve yükleme sırasında sıfırlarla doldurulur. `i` ve `p` işaretçisinin kendisi `main`'in yerel değişkenleridir: yığındadır. `p`'nin gösterdiği 64 baytlık alan ise `malloc` ile ayrıldığı için heap'tedir. Fonksiyonların makine kodu da metin kesimindedir.

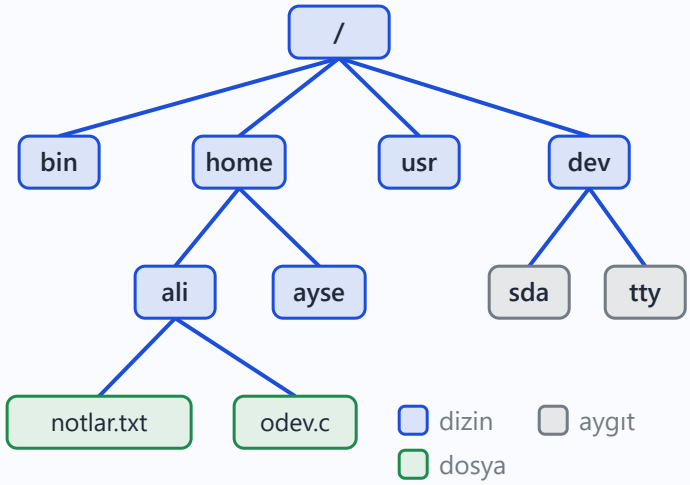
2.2.4 Dosyalar ve dizin hiyerarşisi

Disk, kendi başına yalnız numaralı bloklardan oluşan bir depolama alanıdır. Bir program "47.291 numaralı bloğu oku" gibi isteklerle çalışmak zorunda kalsaydı her programcı diskin iç yapısını öğrenmek zorunda kalırdı. İşletim sistemi bu ayrıntıyı gizler ve programlara daha temiz bir soyutlama sunar: dosya.

Tanım 2.3

Dosya, verileri kalıcı olarak saklayan adlandırılmış bir birimdir. Metin, resim, video ya da çalıştırılabilir bir program bir dosyada saklanabilir. Dosyalar **dizinlerde** (*directory*) toplanır; dizinler de başka dizinleri içerebildiği için bir ağaç oluşur.

Ağaçtaki bir dosyaya **yol adıyla** (*path name*) ulaşılır. Mutlak yol, kök dizinden (`/`) başlar: `/home/ali/notlar.txt`. Göreli yol ise sürecin o anki çalışma dizininden başlar. Çalışma dizini `/home` olan bir süreç aynı dosyaya `ali/notlar.txt` diye ulaşır. Şekil 2.6 örnek bir dizin ağacını gösterir.



Şekil 2.6: Örnek bir dosya sistemi ağacı. Kök dizinin altında `bin`, `home`, `usr` ve `dev` dizinleri vardır. `/home/ali` dizininde `notlar.txt` ve `odev.c` dosyaları, `/dev` dizininde ise `sda` ve `tty` aygıt dosyaları bulunur.

Bir süreç bir dosyayı açtığında işletim sistemi ona küçük bir tam sayı verir. Bu sayıya **dosya tanımlayıcısı** (*file descriptor*, fd) denir. Süreç daha sonra dosyayı adıyla değil bu sayıyla anar: "3 numaralı dosyadan 100 bayt oku" gibi. Her süreç üç açık tanımlayıcıyla başlar: 0 standart girdi (`stdin`), 1 standart çıktı (`stdout`), 2 standart hata (`stderr`). Süreç yeni bir dosya açtığında ona en küçük boş numara verilir; ilk açılan dosya bu yüzden genellikle 3 numarayı alır.

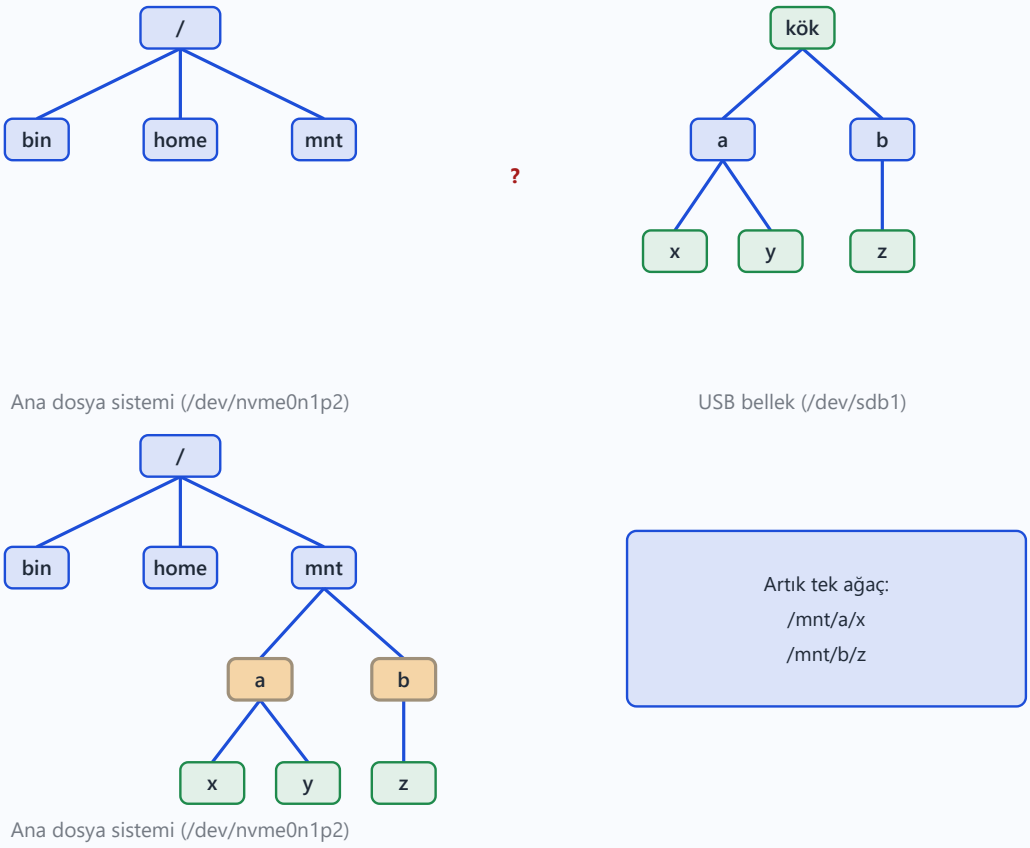
UNIX'in dikkat çekici bir fikri, aygıtların da dosya gibi görünmesidir. Bunlara **özel dosya** denir ve iki türü vardır. **Blok aygıtları** veriyi sabit büyüklükteki bloklar hâlinde saklar ve bloklara istenen sırayla erişime izin verir: `/dev/sda` ve `/dev/nvme0n1` diskleri gibi. **Karakter aygıtları** ise veriyi bir karakter akışı olarak üretir ya da tüketir: terminal (`/dev/tty`) ve seri port gibi. Bir program terminale yazarken de bir dosyaya yazarken de aynı `write` çağrısını kullanır.

2.2.5 Dosya sistemini bağlama

Bir bilgisayara USB bellek takıldığında, USB bellekteki dosya sistemi başlangıçta ayrı bir ağaçtır. Sistemin ana ağacıyla hiçbir bağı yoktur ve hiçbir yol adıyla ona ulaşılamaz. UNIX'te tek bir dizin ağacı vardır; yeni bir dosya sisteminin bu ağaca eklenmesine **bağlama** (*mount*) denir.

Tanım 2.4

Bağlama, bir dosya sisteminin kök dizinini mevcut ağaçtaki bir dizine iliştirme işlemidir. İliştirmenin yapıldığı dizine **bağlama noktası** denir. Bağlamadan sonra yeni dosya sistemindeki dosyalara, bağlama noktasıyla başlayan yol adlarıyla ulaşılır.



Şekil 2.7: Bağlamadan önce ve sonra. Üstte ana dosya sistemi (`/dev/nvme0n1p2`) ve USB bellekteki ayrı ağaç görülür; USB'deki `a` ve `b` dizinlerine hiçbir yoldan ulaşılamaz. Altta USB'nin kökü `/mnt` dizinine bağlanmıştır; artık `/mnt/a/x` ve `/mnt/b/z` yolları geçerlidir.

Şekil 2.7 işlemi iki anda gösterir. Bağlama noktası olarak genellikle boş bir dizin kullanılır. Dizin boş değilse eski içeriği silinmez; yalnız bağlantı kaldırılana kadar görünmez olur. Windows bu soruyu farklı çözer: her disk ya da bölüm genellikle ayrı bir sürücü harfi (C:, D:) alır ve ayrı bir ağaç olarak kalır.

Linux'ta bağlama işi `mount` komutuyla yapılır. Aşağıdaki komut dizisi sık kullanılan biçimleri gösterir:

```
$ lsblk                            # diskleri ve bölümleri listele
$ sudo mount /dev/sdb1 /mnt        # sdb1'i /mnt dizinine bağla
$ sudo mount -o ro /dev/sdb1 /mnt  # salt okunur (read-only) bağla
$ sudo mount -o remount,rw /mnt    # okuma-yazma olarak yeniden bağla
$ findmnt /mnt                     # bağlama bilgisini göster
$ sudo umount /mnt                 # bağlantıyı kaldır
```

`lsblk` sistemdeki diskleri ve bölümleri listeler; buradan USB belleğin `/dev/sdb1` adını aldığı görülür. İkinci satır onu `/mnt`'ye okuma-yazma kipinde bağlar. `-o ro` seçeneği salt okunur bağlama yapar; kurtarma ve adli inceleme işlerinde diskteki veriyi yanlışlıkla değiştirmemek için bu kip tercih

edilir. `remount,rw` zaten bağlı bir dosya sistemini okuma-yazma kipine geçirir. `findmnt` bağlama bilgisini, `umount` ise bağlantının kaldırılmasını sağlar. Bu komutların çoğu yönetici yetkisi gerektirdiği için `sudo` ile çalıştırılır.

Açılışta kendiliğinden bağlanacak dosya sistemleri `/etc/fstab` dosyasında ya da `systemd` bağlama birimlerinde tanımlanır. Masaüstü sistemlerde takılan bir USB bellek ise `udisks2` hizmeti aracılığıyla `/run/media/…` altına kendiliğinden bağlanır. Şekil 2.8 bağlamadan sonra USB'deki `belge.pdf` dosyasının nasıl `/mnt/belge.pdf` yoluyla erişilebilir hâle geldiğini özetler.



Şekil 2.8: `mount /dev/sdb1 /mnt` komutundan sonra ağaç. `/mnt` dizini USB'deki dosya sistemine bağlanmıştır; USB'nin kökündeki `belge.pdf` artık `/mnt/belge.pdf` yoluyla açılır.

2.2.6 Boru

Kabukta `ls -l | grep "notlar"` gibi bir komut yazıldığında iki program aynı anda çalışır: `ls` dizin listesini üretir, `grep` bu listeden içinde "notlar" geçen satırları seçer. İki program arasında geçici bir dosya yoktur. Veriyi birinden ötekine taşıyan yapı bir borudur.

Tanım 2.5

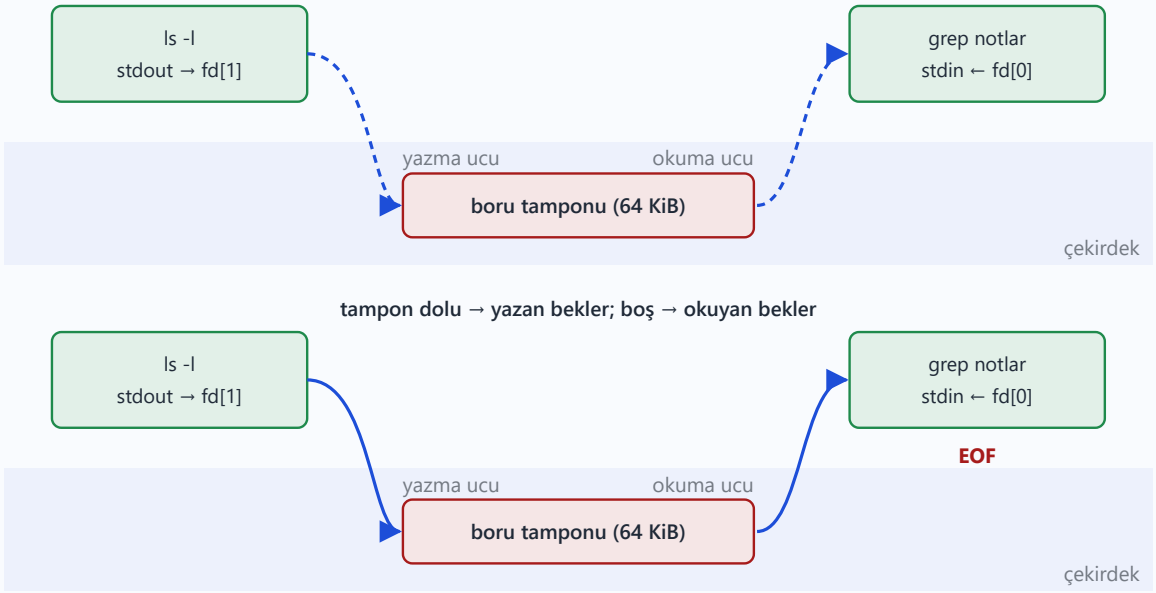
Boru (*pipe*), iki süreci bağlayan, çekirdekte tamponlanan tek yönlü bir bayt kanalıdır. Bir süreç borunun yazma ucuna yazar, öteki süreç okuma ucundan okur. Süreçler boruyu sıradan bir dosya gibi `read` ve `write` çağrılarıyla kullanır.

İki tür boru vardır. **Anonim boru** `pipe()` çağrısıyla oluşur ve adı yoktur; bu yüzden yalnız ortak ataya sahip süreçler, tipik olarak ebeveyn ve çocuk süreçler arasında kullanılır. **Adlandırılmış boru** (FIFO) ise `mkfifo` ile oluşturulur ve dosya sisteminde bir adı vardır. Birbiriyle akrabalığı olmayan süreçler bile bu adı açarak aynı boruyu kullanabilir.

Kabuğun yukarıdaki komutu nasıl kurduğu dört adımda izlenebilir:

1. **Boru oluşturulur.** Kabuk önce `pipe(fd)` çağırır. Çekirdek bir tampon ayırır ve iki tanımlayıcı döndürür: `fd[0]` okuma ucu, `fd[1]` yazma ucudur.
2. **Süreçler oluşturulur ve uçlar yönlendirilir.** Kabuk iki kez `fork` eder. Birinci çocukta `dup2` çağrısıyla standart çıktı (1) borunun yazma ucuna, ikinci çocukta standart girdi (0) okuma ucuna yönlendirilir. Ardından çocuklar kendilerini `ls` ve `grep` programlarıyla değiştirir.
3. **Veri akar.** `ls` listeyi standart çıktısına, yani boruya yazar. `grep` standart girdisinden, yani borudan okur ve yalnız "notlar" içeren satırları terminale yazar. İki program da bir boruya bağlı olduğunu bilmez.

4. **Akış denetimi ve bitiş.** Tampon dolunca yazan süreç, boşalınca okuyan süreç bloke olur. `ls` bitip borunun bütün yazma uçları kapanınca `grep`'in `read` çağrısı 0 döndürür. Bu değer dosya sonu (EOF) demektir ve `grep` de biter.



Şekil 2.9: `ls -l | grep "notlar"` boru hattı. Üstte uçların yönlendirilmesi: `ls`'in standart çıktısı `fd[1]`'e, `grep`'in standart girdisi `fd[0]`'a bağlanmıştır. Altta akış denetimi: boru tamponu (64 KiB) çekirdektedir; tampon dolunca yazan, boşken okuyan bekler, yazma ucu kapanınca okuyan dosya sonunu görür.

Sık yapılan hata

Kabuğun kendi elindeki boru uçlarını kapatmayı unutması. Okuyan süreç, dosya sonunu ancak borunun *bütün* yazma uçları kapandığında görür. Ebeveyn kabuk, `fork` sonrasında kendi kopyasındaki yazma ucunu açık tutarsa `ls` bitse bile `grep` sonsuza dek yeni veri bekler.

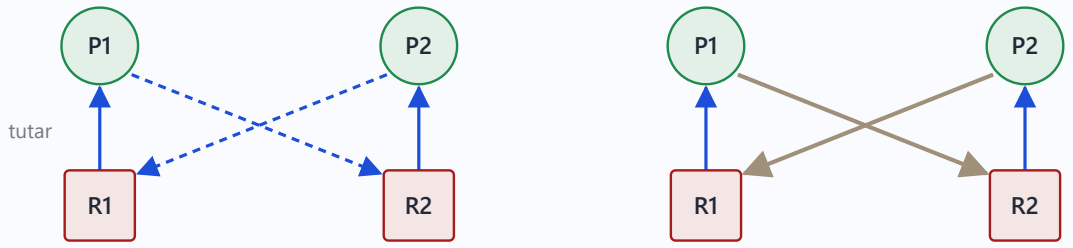
2.2.7 Kilitlenme

Süreçler yalnız işlemciyi değil, yazıcı, dosya, bellek tamponu ya da kilit gibi başka kaynakları da kullanır. Bazı kaynakları aynı anda yalnız bir süreç tutabilir. İki ya da daha fazla süreç birbirinin elindeki kaynağı beklemeye başlarsa hiçbiri ilerleyemez.

Tanım 2.6

Bir süreç kümesindeki her süreç, kümedeki başka bir sürecin tuttuğu kaynağı bekliyorsa ve bu yüzden hiçbiri ilerleyemiyorsa kümede **kilitlenme** (*deadlock*) vardır.

Günlük hayattaki karşılığı, dört yönden aynı anda bir kavşağa giren ve her biri sağındaki aracın geçmesini bekleyen dört araçtır. Kurala uyan sürücülerin hiçbiri yerinden kıpırdamaz. Durumu çizmenin standart yolu, **kaynak tahsis çizgesidir** (*resource allocation graph*). Şekil 2.10 iki süreç ve iki kaynaklık bir örneği iki aşamada gösterir.



(a) Potansiyel: istekler henüz yapılmadı

(b) Gerçekleşmiş: döngü P1→R2→P2→R1→P1

Şekil 2.10: Kaynak tahsis çizgesi. Kareden sürece ok, kaynağın o süreçte olduğunu; süreçten kareye ok, sürecin o kaynağı istediğini gösterir. (a) Potansiyel kilitlenme: P1 R1'i, P2 R2'yi tutar; kesikli oklar henüz yapılmamış istekleri gösterir. (b) Gerçekleşmiş kilitlenme: istekler yapılmış ve P1 → R2 → P2 → R1 → P1 döngüsü kapanmıştır.

(a) durumunda henüz sorun yoktur: P1 R1'i, P2 R2'yi tutmaktadır ve ileride ötekinin kaynağını isteyeceklerdir. Bu isteklerin ikisi de yapıldığında (b) durumuna gelinir. P1, P2'nin tuttuğu R2'yi; P2 de P1'in tuttuğu R1'i bekler. Çizgede kapalı bir döngü oluşmuştur ve iki süreç de sonsuza dek bekler. Kaynakların her birinin tek örneği varsa çizgedeki bir döngü kilitlenmenin kesin işaretidir. Kaynaklar birden çok örnekliyse döngü, kilitlenme olasılığını gösterir ama tek başına kanıt değildir.

2.3 Koruma

Bellekte aynı anda çekirdek ve onlarca kullanıcı süreci bulunur. Süreçlerden biri hatalı ya da kötü niyetli olabilir. Böyle bir süreç diske doğrudan yazarsa dosya sistemini bozabilir; kesmeleri kapatırsa işletim sistemini devre dışı bırakabilir; sonsuz bir döngüye girerse işlemciyi hiç bırakmayabilir. İşletim sistemi bu tehlikelere karşı yalnız yazılımla korunamaz, çünkü sorunlu kod da aynı işlemcide çalışmaktadır. Korumanın temeli donanımdadır.

2.3.1 Çift modlu çalışma

Günümüz işlemcileri en az iki çalışma modu tanır. Hangi modda olunduğu, işlemcideki bir **mod bitinde** tutulur. Silberschatz'ın kullandığı gösterimde bu bit çekirdek modunda 0, kullanıcı modunda 1'dir.

Tanım 2.7

Çift modlu çalışma (*dual-mode operation*), işlemcinin **çekirdek modu** (*kernel mode*) ile **kullanıcı modu** (*user mode*) arasında ayrım yapmasıdır. **Ayrıcalıklı komutlar** yalnız çekirdek modunda yürütülebilir. Kullanıcı modunda böyle bir komut denenirse donanım bir tuzak üretir ve kontrol işletim sistemine geçer.

Ayrıcalıklı komutların ortak yanı, yanlış kullanıldıklarında başka süreçlere ya da sistemin bütününe zarar verebilmeleridir. G/Ç aygıtlarına doğrudan erişmek, zamanlayıcıyı ayarlamak, kesmeleri kapatmak ve sayfa tablolarını değiştirmek bu gruptadır. Sistem saatini okumak gibi kimseye zarar vermeyen işler ise kullanıcı modunda serbestçe yapılabilir.

Kullanıcı modunda ayrıcalıklı bir komut denendiğinde donanım komutu yürütmez; bunun yerine bir **tuzak** (*trap*) üretir. İşlemci çekirdek moduna geçer ve işletim sisteminin önceden belirlediği bir işleyiciye atlar. İşletim sistemi de çoğu zaman hatalı süreci sonlandırır. Çekirdeğe girmenin yalnız üç meşru yolu vardır: sistem çağrısı, donanım kesmesi ve istisna (sıfıra bölme, geçersiz bellek erişimi gibi). Üçünde de işlemci, sürecin seçtiği bir adrese değil, çekirdeğin belirlediği sabit bir giriş noktasına atlar.

Gerçek işlemcilerde mod sayısı ikiden fazla olabilir. x86 mimarisi 0'dan 3'e kadar dört koruma halkası tanımlar, ama Linux ve Windows yalnız halka 0'ı (çekirdek) ve halka 3'ü (kullanıcı) kullanır. ARM mimarisinde EL0'dan EL3'e kadar dört ayrıcalık düzeyi vardır; bunlardan EL2 sanallaştırmaya ayrılmıştır.

Sık yapılan hata

Çift modun tek başına yeterli olduğunu düşünmek. Mod ayrımı, sürecin ayrıcalıklı komut çalıştırmasını engeller; ama iki sorunu çözmez. Süreç ayrıcalıklı hiçbir komut kullanmadan sonsuz döngüye girip işlemciyi tutabilir; bunun çaresi zamanlayıcıdır. Süreç başka bir sürecin belleğine sıradan bir yükleme komutuyla erişmeye çalışabilir; bunun çaresi de bellek yönetim birimidir (MMU).

2.3.2 Zamanlayıcı

Bir süreç `while (1) ;` gibi bir döngüye girerse hiçbir ayrıcalıklı komut çalıştırmadan işlemciyi sonsuza dek elinde tutabilir. İşletim sistemi de çalışmak için işlemciye ihtiyaç duyduğundan bu durumu kendi başına fark edemez. Çözüm, belirli bir süre sonunda işlemciyi kesen bir donanım parçasıdır.

Tanım 2.8

Zamanlayıcı (*timer*), belirli bir süre sonunda kesme üreten donanım sayacıdır. İşletim sistemi sayaca bir değer yükler; fiziksel saatin her darbesinde sayaç bir azalır; sayaç sıfıra indiğinde bir kesme üretilir ve kontrol işletim sistemine geçer.

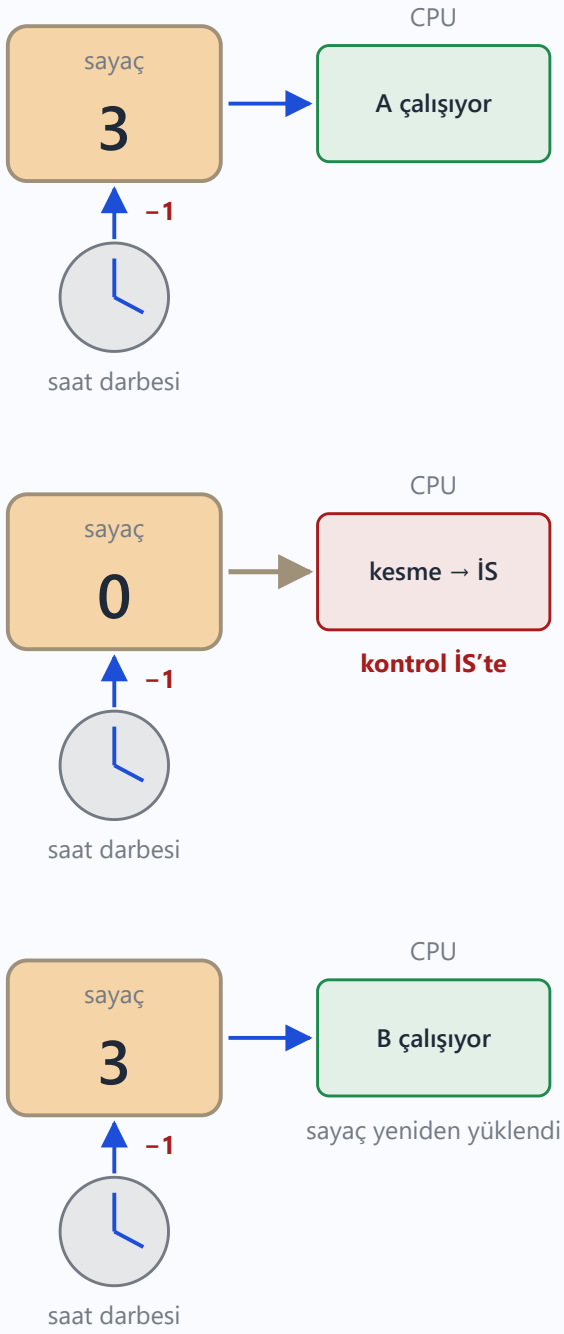
Sayaca değer yüklemek ayrıcalıklı bir komuttur. Böylece kullanıcı programı zamanlayıcıyı durduramaz ya da süresini uzatamaz. Zamanlayıcı kesmesi geldiğinde işletim sistemi şunu sorar: o anda çalışan süreç kendisine ayrılan **zaman dilimini** doldurdu mu? Doldurduysa süreç işlemciden alınır ve işlemci başka bir sürece verilir. Bu düzene **kesmeli çizelgeleme** (*preemptive scheduling*) denir. Sürecin kendisi sonlandırılmaz; yalnız sırası başkasına geçer. Süreç ancak kendisine tanınan toplam işlemci süresi sınırını (Linux'ta `RLIMIT_CPU`) aşarsa sonlandırılır.

Örnek 2.3

A ve B adlı iki süreç vardır. İşletim sistemi A'yı çalıştırmadan önce sayaca 3 yüklüyor. Saatin her darbesinde ne olur?

1. **Yükleme.** İşletim sistemi çekirdek modunda sayaca 3 yazar, ardından kullanıcı moduna geçip A'yı başlatır.
2. **1. darbe.** Sayaç 2 olur. A çalışmayı sürdürür; olup bitenden haberi yoktur.
3. **2. darbe.** Sayaç 1 olur. A hâlâ işlemcidedir.
4. **3. darbe.** Sayaç 0'a iner ve zamanlayıcı kesmesi üretilir. İşlemci çekirdek moduna geçer ve kesme işleyicisi çalışır.
5. **Çizelgeleme.** Çizelgeleyici A'nın diliminin dolduğunu görür, A'nın durumunu PCB'sine kaydeder ve B'yi seçer. Sayaca yeniden değer yükler ve B'yi kullanıcı modunda başlatır.

A, üç darbe süresince işlemciyi kullanmıştır. A sonsuz bir döngüde olsa bile bu süre sonunda işlemciyi bırakmak zorunda kalır.



Şekil 2.11: Örnek 2.3'ün üç anı. İlk karede sayaç 3'tür ve A çalışır. İkinci karede sayaç 0'a inmiş, kesme üretilmiş ve kontrol işletim sistemine geçmiştir. Üçüncü karede çizelgeleyici B'yi çalıştırmış ve sayacı yeniden yüklemiştir.

Linux'ta zamanlayıcı kesmesi düzenli aralıklarla, saniyede 100 ile 1000 arasında üretilir; her kesmeye bir **tik** denir. Sık tik, daha ince zaman dilimleri ve daha duyarlı çizelgeleme sağlar, ama her tikin işlenmesi işlemciden zaman alır. Üstelik sistem boştayken bile işlemci her tikte uyanmak zorunda kalır. Bu yüzden güncel çekirdekler, boşta tik üretmeyen bir kip (*tickless*, NO_HZ) de sunar; özellikle dizüstü bilgisayarlarda bu kip enerji kazandırır.

2.4 Sistem çağrıları

Çift modlu çalışma, kullanıcı programlarının donanıma doğrudan erişmesini yasaklar. Oysa programların dosya okuması, bellek istemesi, ekrana yazması ve yeni süreç oluşturması gerekir. Bu işleri yapabilen tek taraf çekirdektir. Programın çekirdekten hizmet istemesinin denetimli yolu sistem çağrısıdır.

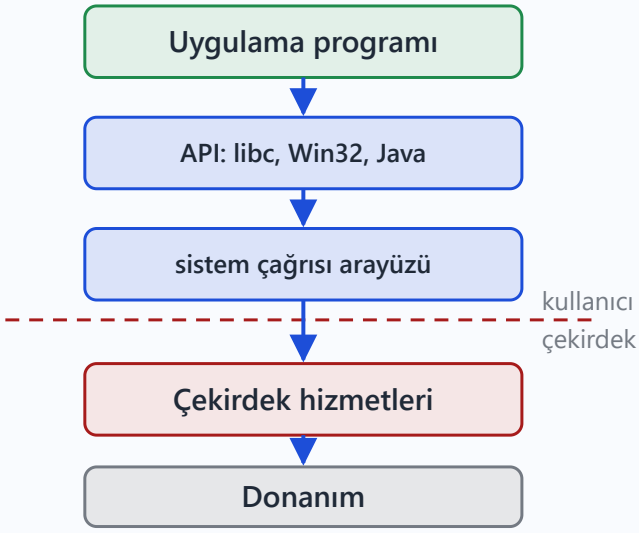
2.4.1 Sistem çağrısı nedir?

Tanım 2.9

Sistem çağrısı (*system call*), programların dosya, bellek, süreç, zaman ve ağ gibi işletim sistemi hizmetlerine erişmek için kullandığı tanımlı arayüzdür. Çağrıyı uygulama başlatır; istenen işi çekirdek, çekirdek modunda yürütür.

Programcı sistem çağrılarını çoğu zaman doğrudan kullanmaz. Arada bir **uygulama programlama arayüzü** (*application programming interface*, API) bulunur. UNIX benzeri sistemlerde bu arayüz POSIX standardıdır ve C kütüphanesi (glibc ya da musl) içinde gerçeklenir. Windows'ta Windows API, Java'da ise Java API aynı görevi görür. API iki yarar sağlar. Birincisi taşınabilirliktir: POSIX'e uygun yazılmış bir program, sistem çağrılarının numaraları ve ayrıntıları farklı olan başka bir UNIX'te de derlenip çalışır. İkincisi kolaylıktır: API fonksiyonları, ham sistem çağrılarının zahmetli ayrıntılarını gizler.

Sistem çağrılarının kümesi sistemden sisteme değişir. x86-64 üzerindeki Linux'ta yaklaşık 370 sistem çağrısı vardır. Temel kavramlar ise ortaktır: her sistemde süreç oluşturan, dosya açan, okuyan ve yazan çağrılar bulunur. Bir programın hangi sistem çağrılarını yaptığını görmek için Linux'ta `strace`, macOS'ta `dtruss`, Windows'ta Process Monitor aracı kullanılır.

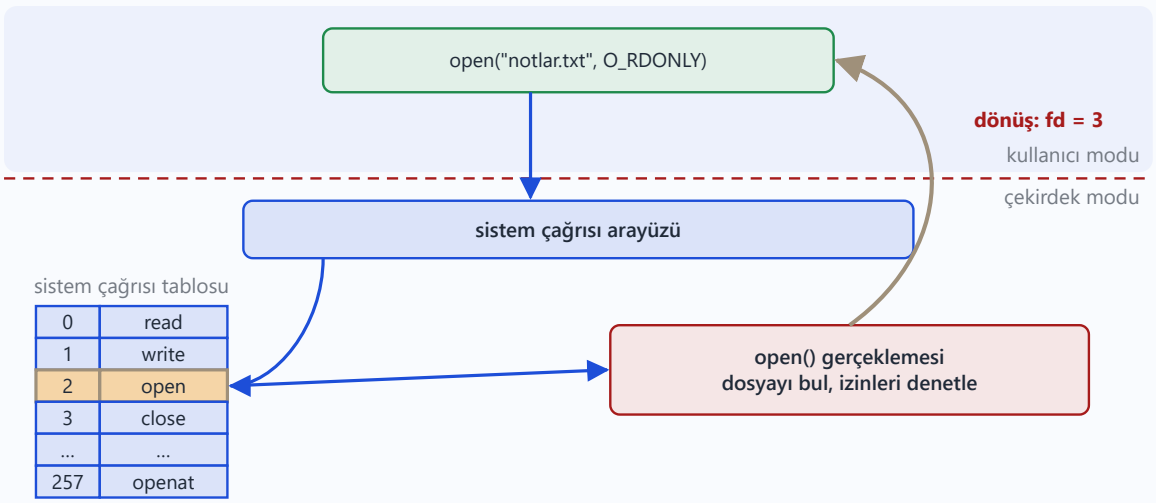


Şekil 2.12: Sistem çağrısının geçtiği katmanlar: uygulama programı, API (libc, Win32, Java), sistem çağrısı arayüzü, çekirdek hizmetleri ve donanım. Kesikli çizgi kullanıcı modunu çekirdek modundan ayırır.

2.4.2 API'den çekirdeğe

Bir C programının `open("notlar.txt", O_RDONLY)` çağrısını izleyelim. Şekil 2.13 çağrının yolculuğunu baştan sona gösterir.

1. **Sarmalayıcı.** Program, C kütüphanesindeki `open()` fonksiyonunu sıradan bir fonksiyon gibi çağırır. Bu fonksiyon bir **sarmalayıcıdır** (*wrapper*): çağrı numarasını ve parametreleri yazmaçlara koyar, ardından tuzak komutunu çalıştırır. İşlemci çekirdek moduna geçer.
2. **Sistem çağrısı arayüzü.** Çekirdekteki giriş kodu, çağrı numarasını **sistem çağrısı tablosunda** indis olarak kullanır. x86-64 Linux'ta `read` 0, `write` 1, `open` 2, `close` 3 numaralıdır.
3. **Gerçekleme.** Tablodaki işaretçi, çekirdekteki `open` kodunu gösterir. Bu kod yol adını çözer, dosyayı bulur, izinleri denetler ve bir açık dosya kaydı oluşturur.
4. **Dönüş.** Sonuç kullanıcıya döner. 0, 1 ve 2 dolu olduğundan ilk boş tanımlayıcı 3'tür; `open` 3 döndürür ve işlemci yeniden kullanıcı moduna geçer.



Şekil 2.13: `open` çağrısının kullanıcı modundan çekirdek moduna yolu. Sistem çağrısı arayüzü 2 numaralı girdiyi bulur, tablodaki işaretçi çekirdekteki `open()` gerçeklemesine götürür ve sonuç (`fd = 3`) kullanıcı moduna döner.

Bir ayrıntı, kütüphane ile çekirdek arasındaki ilişkiyi iyi gösterir. Güncel glibc, programın çağırdığı `open()` 'u aslında `openat` sistem çağrısıyla (x86-64'te 257 numara) gerçekler. Program yine `open` yazar ve aynı sonucu alır; aradaki değişikliği API gizler. `strace` ile bakıldığında ise çekirdeğe giden çağrının adı `openat` olarak görünür.

2.4.3 Sistem çağrısı tablosu

Sistem çağrısı tablosunun nasıl çalıştığını küçük bir C modeliyle izlemek mümkündür. Aşağıdaki kodda tablo yalnız üç girdiden oluşur. Gerçek Linux çekirdeğinde tablo birkaç yüz girdi içerir, ama mantık aynıdır.

```
typedef long (*cagri_t)(long, long, long);
cagri_t tablo[3] = {sys_read, sys_write,
                    sys_open};
long syscall_giris(long no, long a,
                   long b, long c) {
  if (no < 0 || no >= 3)
    return -ENOSYS;     /* geçersiz numara */
  return tablo[no](a, b, c);
}
```

İlk satır, üç `long` parametre alıp `long` döndüren bir fonksiyonu gösteren bir işaretçi türü tanımlar ve buna `cagri_t` adını verir. İkinci satırdaki `tablo`, bu türden üç işaretçilik bir dizidir: 0. girdi `sys_read` 'i, 1. girdi `sys_write` 'ı, 2. girdi `sys_open` 'ı gösterir. `syscall_giris` fonksiyonu çekirde-

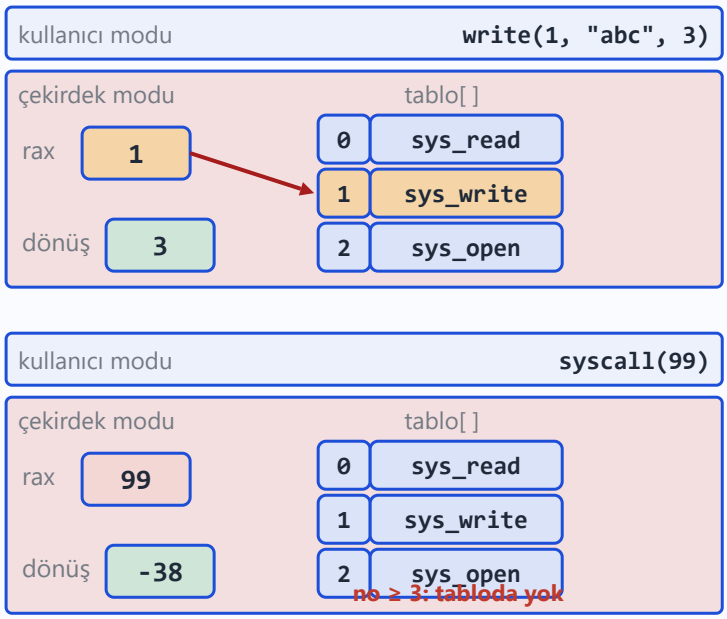
ğin tek giriş noktasını temsil eder. Önce çağrı numarasının tablonun sınırları içinde olup olmadığını denetler. Numara geçersizse `-ENOSYS` döndürür; geçerliyse tablodaki işaretçiyi kullanarak doğru fonksiyonu çağırır ve onun sonucunu döndürür.

Örnek 2.4

Programın art arda üç çağrı yaptığını varsayalım: üç bayt yazan bir `write`, `open("notlar.txt", 0)` ve tabloda karşılığı olmayan `syscall(99)`. Her çağrıda `syscall_giris` nasıl çalışır?

Çağrı	`no`	Sınır denetimi	Çalışan kod	Sonuç
`write`	1	0 ≤ 1 < 3: geçerli	`tablo[1]` → `sys_write`	3 (yazılan bayt sayısı)
`open`	2	0 ≤ 2 < 3: geçerli	`tablo[2]` → `sys_open`	3 (yeni tanımlayıcı)
`syscall(99)`	99	99 ≥ 3: geçersiz	yok	−ENOSYS (−38); programa −1, `errno = ENOSYS`

İlk iki çağrıda libc sarmalayıcısı numarayı `rax` yazmacına koyar ve `syscall` komutunu çalıştırır. Donanım çekirdek moduna geçip tek giriş noktasına atlar. Sınır denetimi geçer ve tablodaki fonksiyon işaretçisi doğru işleyiciyi çağırır. Üçüncü çağrıda numara tablonun dışındadır. Denetim olmasaydı `tablo[99]`, dizinin sonundan çok ötedeki bir bellek hücresini fonksiyon işaretçisi gibi kullanırdı; çekirdek rastgele bir adrese atlardı. Denetim sayesinde çağrı −38 değeriyle reddedilir. libc bu negatif değeri `errno = ENOSYS` biçimine çevirir ve programa −1 döndürür.



Şekil 2.14: Örnek 2.4'ün iki anı. İlk karede `no = 1` : `tablo[1]` seçilmiş ve `sys_write` 3 döndürmüştür. İkinci karede `no = 99` : numara tablonun dışında kaldığı için çağrı –ENOSYS ile reddedilmiştir.

Bu modelden çıkan ders geneldir: çekirdek, kullanıcıdan gelen hiçbir değere doğrulamadan güvenmez. Çağrı numarası, dosya tanımlayıcısı, tampon adresi ve uzunluğu, kullanıcı programının yazdığı değerlerdir ve hepsi denetlenir. `read` çağrısına verilen tampon adresinin gerçekten o sürecin belleğinde olup olmadığı da bu denetimlerden biridir.

2.4.4 Örnek: bir dosyayı kopyalamak

Sıradan görünen bir program bile çok sayıda sistem çağrısı yapar. Bir dosyayı başka bir dosyaya kopyalayan ve dosya adlarını kullanıcıdan isteyen bir program düşünelim. Programın adımları ve her adımın kullandığı sistem çağrıları şöyledir:

Adım	Sistem çağrıları
Girdi dosyasının adını iste ve oku	`write` , `read`
Çıktı dosyasının adını iste ve oku	`write` , `read`
Girdi dosyasını aç (yoksa hata ver ve çık)	`open`
Çıktı dosyasını oluştur (varsa hata ver ve çık)	`open` + `O_CREAT`
Döngü: girdiden oku, çıktıya yaz	`read` , `write`
Dosyaları kapat	`close`
Ekrana "tamamlandı" yaz	`write`
Normal biçimde sonlan	`exit`

Tablo 2.1: Dosya kopyalama programının adımları ve sistem çağrıları.

Kullanıcıdan bir dosya adı almak bile iki çağrı ister: ekrana bir istem yazmak için `write`, klavyeden okumak için `read`. Dosya açma başarısız olabilir: dosya yoktur ya da okuma izni yoktur. Bu durumda `open` −1 döndürür ve program bir hata iletisi yazıp çıkar. Çıktı dosyası `O_WRONLY | O_CREAT | O_EXCL` bayraklarıyla açılırsa, aynı adda bir dosya zaten varsa çağrı hata verir; böylece var olan bir dosyanın üzerine yanlışlıkla yazılmaz. Asıl iş döngüdedir: `read` 0 döndürene, yani dosya sonuna gelinene kadar okunan baytlar `write` ile yazılır. Bu kadar basit bir program bile onlarca sistem çağrısı yapar; Linux'ta `strace -c cp a b` komutu, `cp` programının yaptığı çağrıları türlerine göre sayar.

Döngünün kendisini ayrıntılı izlemek için aşağıdaki kod parçasını ele alalım:

```
char buf[4096];
int in = open("kaynak", O_RDONLY);
int out = creat("hedef", 0644);
ssize_t n;  long toplam = 0;
while ((n = read(in, buf, 4096)) > 0) {
  write(out, buf, n);
  toplam += n;
}
close(in);  close(out);
printf("%ld bayt\n", toplam);
```

Program 4096 baytlık bir tampon ayırır. `kaynak` dosyasını salt okunur açar ve tanımlayıcısını `in`'e koyar. `creat` çağrısı `hedef` adlı boş bir dosya oluşturur ve izinlerini 0644 (sahibi okur ve yazar, diğerleri yalnız okur) olarak belirler. Döngü koşulu hem okumayı yapar hem de sonucu sınar: `read` en çok 4096 bayt okur ve okuduğu bayt sayısını `n`'ye yazar; `n` pozitif olduğu sürece döngü sürer. Döngünün gövdesi tampondaki *n* baytı hedefe yazar ve `toplam`'ı günceller. Sonda iki dosya kapatılır ve kopyalanan bayt sayısı yazdırılır.

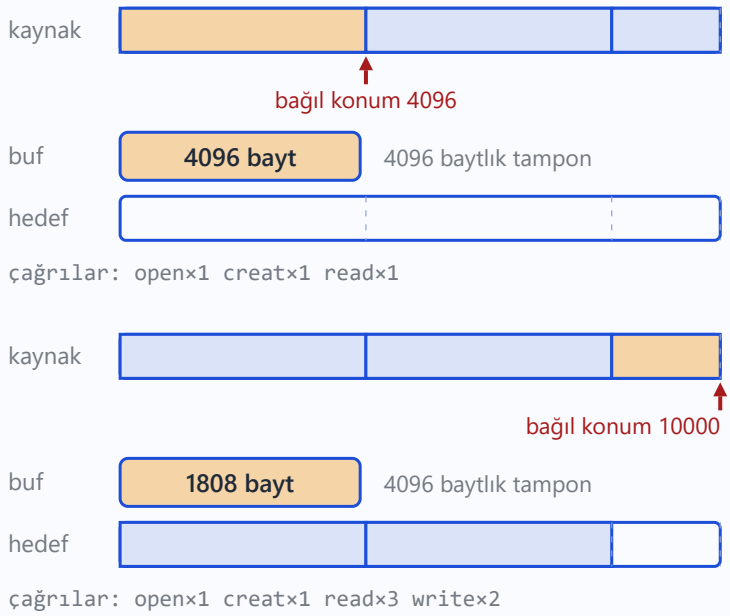
Örnek 2.5

`kaynak` 10.000 baytlık bir dosyadır ve sürecin yalnız 0, 1, 2 tanımlayıcıları açıktır. Kod parçası adım adım şöyle ilerler.

Adım	Çağrı	`n`	Dosya konumu	`toplam`
1	`open("kaynak") → in = 3`	—	0	0
2	`creat("hedef") → out = 4`	—	0	0
3	`read`, `write`	4096	4096	4096
4	`read`, `write`	4096	8192	8192
5	`read`, `write`	1808	10000	10000
6	`read`	0	10000	10000
7	`close(3)`, `close(4)`	0	—	10000
8	`printf → write(1, …)`	0	—	10000

İlk `open` en küçük boş tanımlayıcıyı döndürür; 0, 1, 2 dolu olduğu için `in = 3`, ardından `out = 4` olur. İlk `read` 4096 bayt okur ve çekirdek dosya konumunu 4096 ilerletir. İkinci tur da 4096 bayt okur; konum 8192'ye gelir. Dosyada artık yalnız 10.000 − 8192 = 1808 bayt kalmıştır. Üçüncü `read` istenenden azını, 1808 baytı döndürür; bu bir hata değildir. Konum dosya sonundadır ve dördüncü `read` 0 döndürür. Döngü koşulu yanlış olur ve döngü biter. Program `10000 bayt` yazdırır.

Toplam sistem çağrısı sayısı 12'dir: 1 `open`, 1 `creat`, 4 `read`, 3 `write`, 2 `close` ve `printf`'in yaptığı 1 `write`.



Şekil 2.15: Kopyalama döngüsünün iki anı. İlk karede birinci `read`: kaynağın ilk 4096 baytı tampona alınmıştır. İkinci karede üçüncü `read`: dosyada kalan 1808 bayt okunmuştur.

Sık yapılan hata

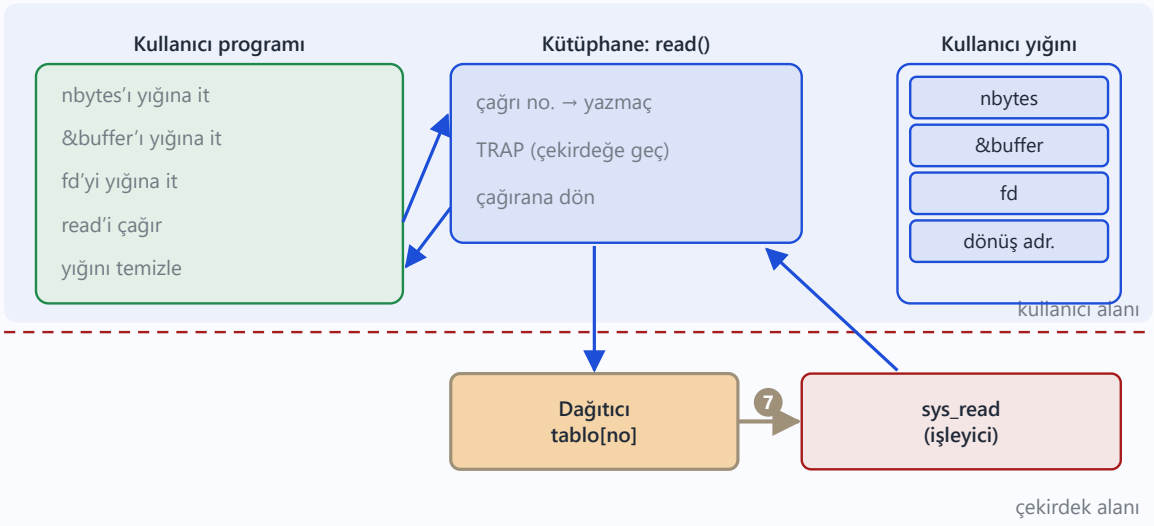
Döngü gövdesinde `write(out, buf, 4096)` yazmak. Son turda tamponda yalnız 1808 geçerli bayt vardır; geri kalan 2288 bayt önceki turdan kalan eski veridir. Sabit 4096 yazılırsa hedef dosya 12.288 bayt olur ve sonuna anlamsız veri eklenir. Doğrusu, `read`'in döndürdüğü `n` kadar bayt yazmaktır.

Tampon boyutu, yapılan sistem çağrısı sayısını doğrudan belirler. Her sistem çağrısı iki mod geçişi ve çekirdekteki giriş-çıkış işleri yüzünden sıradan bir fonksiyon çağrısından çok daha pahalıdır. Aynı 10.000 bayt 1 baytlık bir tamponla kopyalansaydı yalnız okuma ve yazma için yaklaşık 20.000 çağrı gerekirdi: 10.001 `read` ve 10.000 `write`. 4096 baytlık tamponla bu sayı 7'dir.

2.4.5 read çağrısının on bir adımı

Tanenbaum'un ders kitabı, `read(fd, buffer, nbytes)` çağrısını on bir adıma ayırarak anlatır. Örnek, tarihsel 32 bitlik x86 kuralına dayanır: parametreler yığın üzerinden aktarılır. Günümüzün x86-64 sistemleri ilk altı parametreyi yazmaçlarla aktarır, ama adımların sırası ve anlamı değişmez.

1. **Adım 1–3.** Kullanıcı programı parametreleri yığına iter: önce `nbytes`, sonra `buffer`'ın adresi, en son `fd`. C derleyicileri parametreleri sondan başa doğru iter.
2. **Adım 4.** Program kütüphanedeki `read()` fonksiyonunu sıradan bir fonksiyon gibi çağırır; dönüş adresi yığına girer.
3. **Adım 5.** Kütüphane, çağrı numarasını çekirdeğin beklediği yazmaca koyar (Linux x86-64'te `rax = 0`).
4. **Adım 6.** Tuzak komutu (x86-64'te `syscall`) çalışır. İşlemci çekirdek moduna geçer ve çekirdekteki sabit bir adrese atlar.
5. **Adım 7.** Çekirdek çağrı numarasına bakar ve sistem çağrısı tablosu üzerinden doğru işleyiciye dallanır.
6. **Adım 8.** İşleyici, `fd`'nin gösterdiği dosyadan `buffer`'a en çok `nbytes` bayt kopyalar. Veri hazır değilse, örneğin klavyeden henüz bir satır gelmemişse, süreç bloke olur ve işlemci başka bir sürece verilir.
7. **Adım 9.** İş bitince kontrol kullanıcı moduna, tuzak komutundan sonraki komuta döner. Sonuç, yani okunan bayt sayısı, bir yazmaçtadır.
8. **Adım 10.** Kütüphanedeki `read()` fonksiyonu, kendisini çağıran kullanıcı programına döner.
9. **Adım 11.** Program, yığına ittiği parametreleri temizler (yığın işaretçisini artırır) ve sonraki komutla devam eder.



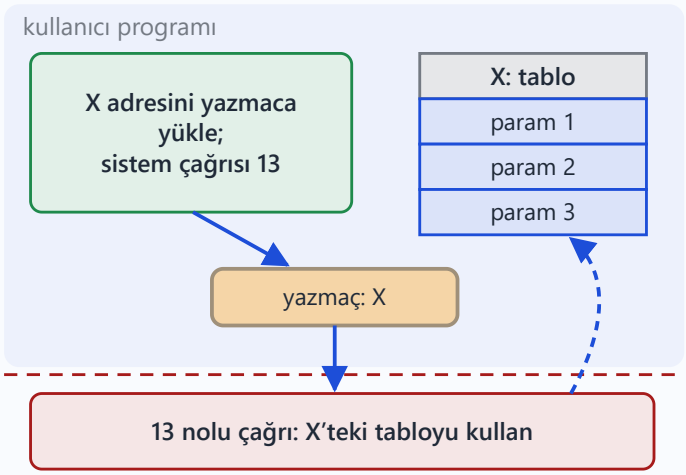
Şekil 2.16: `read` çağrısının 7. adımı. Parametreler (`nbytes`, `&buffer`, `fd`) ve dönüş adresi kullanıcı yığınındadır. Kütüphane tuzak komutuyla çekirdeğe geçmiş, çekirdekteki dağıtıcı `tablo[no]` üzerinden `sys_read` işleyicisine dallanmıştır. Kesikli çizgi kullanıcı alanını çekirdek alanından ayırır.

Adımların ilk dördü ve son ikisi tamamen kullanıcı modunda geçer; bunlar sıradan fonksiyon çağrısı adımlarıdır. Çekirdeğe giriş 6. adımda, çıkış 9. adımda olur. 8. adım, sistem çağrısının sıradan bir fonksiyondan en önemli farkını gösterir: çağrı bir olayı bekleyebilir. Beklenen olay gerçekleşene kadar süreç işlemciyi kullanmaz; işletim sistemi bu süreyi başka süreçler için değerlendirir.

2.4.6 Parametre aktarımı

Sistem çağrısına parametre aktarmanın üç genel yolu vardır:

1. **Yazmaçlar.** En hızlı yoldur. Linux, bir sistem çağrısına en çok altı parametreyi yazmaçlarla aktarır.
2. **Bellekte blok ya da tablo.** Parametreler bellekte bir yapıya yazılır; yapının adresi bir yazmaçla aktarılır. Parametre sayısı çok olduğunda bu yol kullanılır. Linux'taki `clone3(struct clone_args *)` çağrısı buna bir örnektir.
3. **Yığın.** Program parametreleri yığına iter, işletim sistemi oradan çeker. Tarihsel 32 bitlik sistemlerde yaygındı.



Şekil 2.17: Parametrelerin bellekteki bir tabloyla aktarılması. Kullanıcı programı X adresindeki tabloya üç parametreyi yazar, X'i bir yazmaca yükler ve 13 numaralı sistem çağrısını yapar. Çekirdek, X'teki tabloyu okuyarak parametrelere ulaşır.

x86-64 Linux'ta yazmaçların görevleri sabittir:

Yazmaç	Görev
`rax`	çağrı numarası; dönüşte sonuç
`rdi`, `rsi`, `rdx`	1., 2. ve 3. parametre
`r10`, `r8`, `r9`	4., 5. ve 6. parametre (C fonksiyon çağrısı kuralındaki `rcx` yerine `r10`)
`syscall`	çekirdeğe geçiş komutu; `rcx` ve `r11` yazmaçlarının değeri bozulur
ARM64	`x8` çağrı numarası, `x0` – `x5` parametreler; geçiş komutu `svc #0`

Tablo 2.2: x86-64 Linux'ta sistem çağrısı yazmaçları.

Dördüncü parametrenin neden `rcx` yerine `r10`'da olduğu sorusunun cevabı tablonun son satırındadır. `syscall` komutu, dönüş adresini `rcx`'e, bayrak yazmacını da `r11`'e kaydeder. `rcx`'e konan bir parametre bu yüzden çekirdeğe ulaşmadan silinirdi.

Hata bildirimi de ortak bir kurala bağlıdır. Çekirdek bir hata olduğunda −1 ile −4095 arasında bir değer döndürür; bu değer, hata kodunun eksi işaretlisidir (−errno). libc sarmalayıcısı bu değeri görünce hata kodunu `errno` değişkenine yazar ve çağırana −1 verir. Program, bir sistem çağrısının −1 döndürüp döndürmediğine bakarak hatayı fark eder ve nedenini `errno`'dan öğrenir.

2.4.7 write ve read ile kod izleme

Ekrana yazı yazan en küçük programlardan biri, `printf` yerine doğrudan `write` kullanan şu programdır:

```c
#include <unistd.h>      // write()
#include <string.h>      // strlen()

int main(void) {
    const char *msg = "Merhaba, Dünya!\n";
    size_t len = strlen(msg);        // 17 bayt
    ssize_t n = write(1, msg, len);  // 1: stdout
    if (n == -1)
        return 1;                    // hata
    return 0;
}
```

`msg` işaretçisi, salt okunur veri kesimindeki dizgiyi gösterir. `strlen(msg)` dizginin bayt uzunluğunu verir. `write(1, msg, len)`, 1 numaralı tanımlayıcıya, yani standart çıktıya `len` bayt yazar. Dönen değer −1 ise bir hata olmuştur ve program 1 ile çıkar; değilse 0 ile çıkar.

Örnek 2.6

Program satır satır şöyle çalışır. Dizgideki Türkçe karakter, bayt sayısını etkiler.

1. `strlen`. UTF-8 kodlamasında “Merhaba, ” 9 bayttır (virgülden sonraki boşluk dahil). “Dünya” 6 bayttır, çünkü ü harfi 2 bayt tutar. “!” ve satır sonu karakteri “\n” 2 bayttır. Toplam `len = 17`'dir; karakter sayısı ise 16'dır.
2. **Sarmalayıcı.** libc yazmaçları doldurur: `rax = 1` (`write`), `rdi = 1`, `rsi = msg`, `rdx = 17`. Ardından `syscall` komutunu çalıştırır.
3. **Çekirdek.** Çekirdek modunda `sys_write`, 17 baytı 1 numaralı tanımlayıcının bağlı olduğu terminale aktarır. Ekranda `Merhaba, Dünya!` görünür.
4. **Dönüş.** Kullanıcı moduna dönülür ve `n = 17` olur. Bu değer yazılan bayt sayısıdır; bazı durumlarda istenenden küçük olabilir (kısmi yazma).

5. **Hata denetimi.** `n` −1 değildir, hata yoktur. Hata olsaydı `write` −1 döndürür ve `errno` ayarlanırdı.
6. **Bitiş.** `main` 0 döndürür; libc `exit_group(0)` çağrısıyla süreci sonlandırır.

Programcı uzunluğu elle hesaplayıp sabit 15 yazsaydı, son iki bayt ("!" ve "\n") gönderilmez ve çıktı "Merhaba, Dünya" olarak kesilirdi. Uzunluğu `strlen` ile hesaplamak bu hatayı önler.

Okuma tarafındaki eş program, klavyeden bir satır okuyup aynen ekrana yazar:

```c
#include <unistd.h>
#include <stdio.h>

int main(void) {
    char buf[100];
    ssize_t n = read(0, buf, sizeof(buf));
    if (n == -1) {
        perror("okuma hatası");
        return 1;
    }
    write(1, buf, n);    // tam n bayt
    return 0;
}
```

Örnek 2.7

Kullanıcı program çalışırken `selam` yazıp Enter tuşuna basıyor. Program nasıl ilerler?

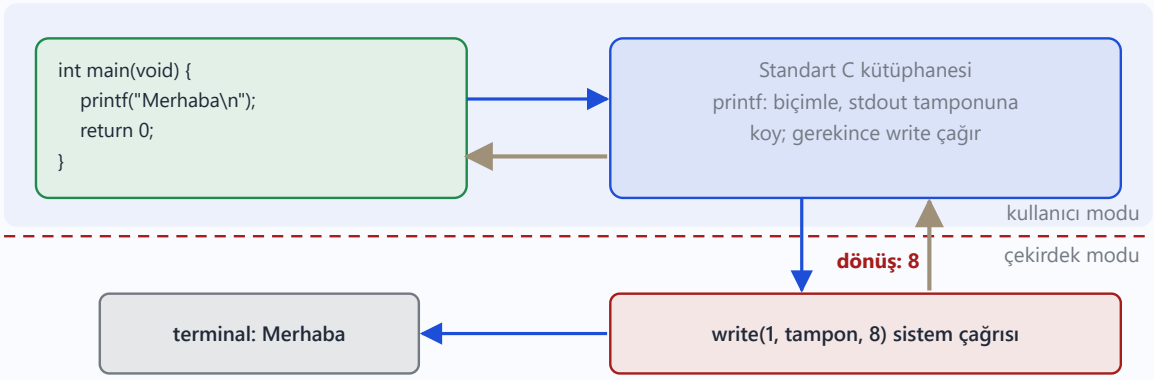
1. **Tampon.** `buf` yığında ayrılan 100 baytlık bir dizidir; içeriği ilklenmemiştir, yani rastgele değerler taşır.
2. **Bloke olma.** `read(0, buf, 100)` standart girdiden, yani klavyeden okumak ister. Henüz bir satır yazılmamıştır. Süreç bloke olur ve çizelgeleyici başka bir süreci çalıştırır.
3. **Kullanıcı yazar.** Kullanıcı `selam` yazıp Enter'a basar. Terminal sürücüsü karakterleri ekrana yansıtır ve satırı kendi tamponunda biriktirir.
4. **Uyanma.** Satır tamamlanınca 6 bayt (`selam` ve "\n") `buf` 'a kopyalanır, süreç uyanır ve `read` 6 döndürür. Tamponun sonuna "\0" eklenmez.
5. **Hata denetimi.** `n = 6` 'dır, hata yoktur; `perror` atlanır.
6. **Yazma.** `write(1, buf, 6)` tam 6 baytı ekrana yazar. Ekranda `selam` iki kez görünür: birincisi terminal sürücüsünün yansıtması, ikincisi programın çıktısıdır.
7. **Bitiş.** Program 0 ile sonlanır.

Sık yapılan hata

Okunan tamponu `printf("%s", buf)` ile yazdırmak. `read`, tamponun sonuna dizgi sonu karakteri (“\0”) koymaz; `buf` bir C dizgisi değildir. `printf` ilk sıfır baytını bulana kadar okumayı sürdürür ve ilklenmemiş bellekteki rastgele baytları da ekrana döker. Doğrusu, `read`'in döndürdüğü `n` değeriyle tam `n` bayt yazmaktır.

2.4.8 Standart C kütüphanesi

C programcısının her gün kullandığı `printf`, bir sistem çağrısı değildir. Kullanıcı modunda çalışan bir kütüphane fonksiyonudur ve gerektiğinde bir sistem çağrısı yapar. Şekil 2.18 `printf("Merhaba\n")` çağrısının yolunu gösterir.



Şekil 2.18: `printf` çağrısının `write` sistem çağrısına dönüşmesi. Program `printf`'i çağırır; standart C kütüphanesi metni biçimler ve `stdout` tamponuna koyar. Satır sonu gelince kütüphane `write(1, tampon, 8)` sistem çağrısını yapar; çekirdek baytları terminale aktarır ve 8 döndürür.

1. Program `printf`'i çağırır. Bu, kullanıcı modunda çalışan sıradan bir fonksiyon çağrısıdır.
2. libc biçim dizgisini işler ve sonucu `stdout`'un tamponuna koyar. Çıktı bir terminale gidiyorsa tampon **satır tamponludur**: satır sonu gelince boşaltılır. Çıktı bir dosyaya ya da boruya yönlendirildiyse tampon **tam tamponludur**: tampon dolunca ya da program bitince boşaltılır.
3. Metin “\n” ile bittiği için libc `write(1, tampon, 8)` çağrısını yapar. “Merhaba” 7 bayt, satır sonu 1 bayttır.
4. `write` 8 döndürür, `printf` da çağırana döner. `strace ./merhaba` komutu yalnız `write`'ı gösterir; `printf` bir sistem çağrısı olmadığı için listede yer almaz.

Standart C kütüphanesinin işlevleri başlık dosyalarına göre gruplanır: `stdio.h` (`printf`, `fopen`), `stdlib.h` (`malloc`, `exit`), `string.h` (`strlen`, `memcpy`), `math.h` ve `time.h`. POSIX ayrıca `unistd.h` başlığında `fork`, `read` ve `write` gibi sistem çağrısı sarmalayıcılarını tanımlar.

Kütüphanenin yaygın gerçeklemeleri Linux'ta glibc ve musl, Windows'ta UCRT'dir. Kütüphane işlevleri iki gruba ayrılır. `strlen` gibi işlevler tamamen kütüphanenin içinde çalışır ve çekirdeğe hiç girmez. `write` gibileri ise bir sistem çağrısını sarmalar.

2.4.9 Sistem çağrısı türleri

Sistem çağrıları yaptıkları işe göre altı grupta toplanır. Tablo 2.3 her grubun görevlerini ve POSIX'teki örneklerini listeler.

Tür	Görevler	POSIX örnekleri
Süreç denetimi	oluştur, sonlandır, yükle, bekle, sinyal gönder	`fork`, `execve`, `exit`, `waitpid`, `kill`
Dosya yönetimi	oluştur, sil, aç, kapat, oku, yaz, öznitelik al	`open`, `read`, `write`, `close`, `stat`
Aygıt yönetimi	aygıt iste ve bırak, oku, yaz, denetle	`ioctl`, `read`, `write`, `mmap`
Bilgi bakımı	saat, tarih, sistem ve süreç bilgisi	`getpid`, `uname`, `clock_gettime`
İletişim	bağlantı kur, ileti gönder ve al, paylaşımlı bellek	`pipe`, `socket`, `sendmsg`, `shm_open`
Koruma	izinleri al ve ayarla, kullanıcı kimliği	`chmod`, `chown`, `umask`, `setuid`

Tablo 2.3: Sistem çağrısı türleri (Silberschatz, Şekil 2.8'e göre).

Aynı gruplar Windows'ta da vardır; yalnız adlar değişir. Süreç oluşturmak için `CreateProcess`, dosya okumak için `ReadFile`, aygıt denetimi için `DeviceIoControl`, saati okumak için `GetSystemTime`, boru oluşturmak için `CreatePipe` ve dosya izinleri için `SetFileSecurity` kullanılır. Bazı çağrıların birden çok grupta görünmesi doğaldır: `read` ve `write` hem dosyalar hem de aygıtlar için kullanılır, çünkü UNIX'te aygıtlar da dosya gibi görünür.

2.5 POSIX ve Win32 çağrıları

POSIX standardı yüzlerce çağrı tanımlar, ama bir UNIX sisteminin temel davranışını anlamak için birkaç düzine çağrı yeterlidir. Bu kesim, Tanenbaum'un seçtiği temel çağrıları üç grupta ele alır: süreç yönetimi, dosya yönetimi ve dizin yönetimi. Kesimin sonunda aynı çağrıların Windows'taki karşılıklarını verilir. Tablolarda ortak bir gösterim kullanılır: `pid` süreç kimliğini, `fd` dosya tanımlayıcısını, `n` bayt sayısını, `s` ise çağrının dönüş kodunu gösterir. Hata durumunda çağrıların hepsi −1 döndürür ve `errno` değişkenini ayarlar.

2.5.1 Süreç yönetimi

Çağrı	Açıklama
`pid = fork()`	Ebeveynin kopyası olan bir çocuk süreç oluşturur.
`pid = waitpid(pid, &statloc, options)`	Bir çocuğun sonlanmasını bekler; çıkış durumunu `statloc`'a yazar.

Çağrı	Açıklama
`s = execve(name, argv, envp)`	Sürecin bellek görüntüsünü yeni bir programla değiştirir; başarılıysa geri dönmez.
`exit(status)`	Süreci sonlandırır ve çıkış durumunu ebeveyne iletir.

Tablo 2.4: Süreç yönetimi için POSIX çağrıları.

`fork`, UNIX'te yeni süreç oluşturmanın tek yoludur. Çağrı, ebeveynin bellek görüntüsünün, açık dosyalarının ve yazmaçlarının bir kopyasıyla yeni bir süreç oluşturur. Çağrıdan sonra iki süreç vardır ve ikisi de `fork`'tan dönen aynı noktadan devam eder. İki sürecin hangisinin hangisi olduğunu ayırt etmenin tek yolu dönüş değeridir: ebeveynde `fork` çocuğun PID'sini, çocukta 0 döndürür. Çağrı başarısız olursa ebeveyne −1 döner ve çocuk oluşmaz. `fork`'un "bir kez çağrılıp iki kez döndüğü" bu yüzden söylenir.

Çocuğun çoğu zaman ebeveynden farklı bir iş yapması istenir. Bunun için çocuk `execve` çağırır. `execve`, sürecin kodunu, verisini ve yığınını diskteki yeni programın görüntüsüyle değiştirir. Süreç aynı süreçtir, PID'si değişmez; yalnız artık başka bir program çalıştırmaktadır. Başarılı bir `execve` geri dönmez, çünkü onu çağıran kod artık bellekte yoktur. `execve`'den sonraki satır ancak çağrı başarısız olursa çalışır.



Şekil 2.19: Süreç yaşam döngüsü. Ebeveyn `fork()` ile bir kopya oluşturur ve `waitpid` ile bloke olur. Çocuk `execve()` ile yeni programa geçer (PID aynı kalır), işini bitirince `exit(status)` çağırır. Bu anda ebeveynin `waitpid` çağrısı döner ve çıkış durumu alınır.

Windows bu işi iki çağrıya bölmez: tek bir `CreateProcess` çağrısı hem yeni süreci oluşturur hem de ona yeni programı yükler. UNIX'in iki adımlı yaklaşımının yararı, `fork` ile `execve` arasındaki kısa aralıktır. Çocuk bu aralıkta kendi açık dosyalarını değiştirebilir; örneğin standart çıktısını bir dosyaya ya da boruya yönlendirebilir. Kabuğun `>` yönlendirmesi ve boru hatları tam bu aralıkta kurulur.

2.5.2 Basit bir kabuk

Süreç yönetimi çağrılarının birlikte nasıl çalıştığını görmenin en iyi yolu, bir **kabuk** (*shell*) yazmaktır. Kabuk, kullanıcıdan komut alıp çalıştıran sıradan bir kullanıcı programıdır. Tanenbaum'un verdiği iskelet, ayrıntıları gizleyerek yalnız asıl yapıyı gösterir:

```
#define TRUE 1

while (TRUE) {              /* sonsuza dek */
    type_prompt();          /* istemi yaz */
    read_command(command, parameters);

    if (fork() != 0) {      /* ebeveyn */
        waitpid(-1, &status, 0);
    } else {                /* çocuk süreç */
        execve(command, parameters, 0);
        perror("execve");   /* başarısızsa */
        _exit(127);
    }
}
```

Kabuk sonsuz bir döngüde çalışır. Her turda istemi yazar (`type_prompt`), kullanıcının yazdığı satırı okuyup komut adına ve parametrelere ayırır (`read_command`). Sonra `fork` çağırır. Dönüş değeri 0'dan farklıysa bu kod ebeveynde, yani kabuğun kendisinde çalışıyordur; kabuk `waitpid` ile çocuğun bitmesini bekler. Dönüş değeri 0 ise kod çocukta çalışıyordur; çocuk `execve` ile kendini istenen programla değiştirir. Başarılı `execve` geri dönmediği için sonraki iki satır yalnız çağrı başarısız olursa çalışır: çocuk hatayı yazar ve `_exit(127)` ile sonlanır (127, kabukların “komut bulunamadı” için kullandığı çıkış durumudur).

Örnek 2.8

PID'si 812 olan kabukta kullanıcı `ls` yazıyor. Çalışma dizininde `notlar.txt` ve `odev.c` dosyaları var. Kabuğun bir turu aşağıdaki tabloda görülür.

Adım	Satır	Olay	Süreçler	Ekran
1	4	İstem yazılır, kullanıcı beklenir.	kabuk (812)	`$`
2	5	Satır ayrıştırılır: `command = "/bin/ls"` , `parameters = ["ls", NULL]` .	kabuk (812)	`$ ls`
3	7	`fork` : ebeveynde 1045, çocukta 0 döner.	kabuk (812), çocuk (1045)	`$ ls`
4	8	Ebeveyn `waitpid(-1, …)` ile bloke olur.	kabuk (bekliyor), çocuk (1045)	`$ ls`
5	10	Çocuk `execve` ile `ls` olur; dizini listeler.	kabuk (bekliyor), ls (1045)	`notlar.txt odev.c`
6	8	`ls` `exit(0)` ile biter; `waitpid` döner, `status = 0` .	kabuk (812)	
7	4	Döngü başa döner, yeni istem yazılır.	kabuk (812)	`$`

3. adımda `fork` 'tan sonra aynı kod iki süreçte birden çalışır. Kabukta `fork() != 0` koşulu doğrudur (1045 ≠ 0) ve kabuk 8. satıra gider. Çocukta koşul yanlıştır (0) ve çocuk `else` dalındaki 10. satıra gider. 5. adımda çocuğun PID'si 1045 olarak kalır; değişen yalnız çalıştırdığı programdır. `waitpid` 'in ilk parametresi −1 olduğu için kabuk, hangi çocuğu olursa olsun ilk biten çocuğu bekler.

Gerçek kabuklar bu iskeletin çok ötesindedir. Komut adını `PATH` ortam değişkenindeki dizinlerde arar, `&` ile arka planda çalışan işleri yönetir, yönlendirmeleri ve boru hatlarını kurar. Ama hepsinin kalbinde aynı `fork` – `execve` – `waitpid` üçlüsü vardır.

Sık yapılan hata

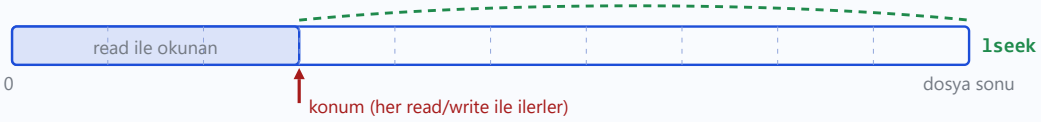
`execve` başarısız olduğunda çocuğun sonlandırılmasını unutmak. Kullanıcı var olmayan bir komut yazarsa `execve` −1 döndürür ve çocuk, kabuk koduyla çalışmaya devam eder. Çocukta `execve` 'nin ardından bir `exit` yoksa çocuk döngünün başına döner ve kendisi de bir kabuk gibi davranmaya başlar. Kullanıcı artık iki kabukla karşı karşıyadır. İskeletteki `_exit(127)` satırı bu yüzden vardır.

2.5.3 Dosya yönetimi

Çağrı	Açıklama
`fd = open(file, how, …)`	Dosyayı okuma, yazma ya da ikisi için açar; gerekirse oluşturur.
`s = close(fd)`	Açık dosyayı kapatır.
`n = read(fd, buffer, nbytes)`	Dosyadan tampona en çok `nbytes` bayt okur.
`n = write(fd, buffer, nbytes)`	Tampondan dosyaya veri yazar.
`position = lseek(fd, offset, whence)`	Dosya konum işaretçisini taşır (`SEEK_SET` , `SEEK_CUR` , `SEEK_END`).
`s = stat(name, &buf)`	Dosyanın durum bilgisini (boyut, izinler, zamanlar) alır.

Tablo 2.5: Dosya yönetimi için POSIX çağrıları.

Çekirdek her açık dosya için bir **konum işaretçisi** tutar. Bu işaretçi, bir sonraki `read` ya da `write` çağrısının dosyanın hangi baytından başlayacağını belirler. Her okuma ve yazma, işaretçiyi işlenen bayt sayısı kadar ilerletir. Kopyalama örneğinde dosyanın baştan sona sırayla okunabilmesi bu sayedir: program hiçbir yerde konum belirtmez, çekirdek konumu kendisi izler.



Şekil 2.20: Dosya konum işaretçisi. Okunan kısım soldadır ve işaretçi her `read` ya da `write` ile sağa ilerler. `lseek` işaretçiyi dosyanın istenen bir noktasına taşır.

Dosyanın belirli bir yerine gitmek için `lseek` kullanılır. `whence` parametresi, `offset`'in neye göre ölçüleceğini belirler: `SEEK_SET` dosyanın başına, `SEEK_CUR` işaretçinin o anki yerine, `SEEK_END` dosyanın sonuna göre. Çağrı, işaretçinin yeni konumunu dosyanın başından itibaren bayt olarak döndürür. Örneğin `lseek(fd, 0, SEEK_END)` işaretçiyi dosya sonuna taşır ve dönüş değeri dosyanın boyutuna eşit olur.

Örnek 2.9

1000 baytlık bir dosya `fd = open(…, O_RDONLY)` ile açılıyor. Ardından `read(fd, buf, 300)`, `lseek(fd, 100, SEEK_CUR)`, `read(fd, buf, 300)`, `lseek(fd, -50, SEEK_END)` ve `read(fd, buf, 300)` çağrıları yapılıyor. Her çağrı ne döndürür?

Çözüm. İşaretçi açılışta 0'dır. İlk `read` 300 bayt okur ve 300 döndürür; işaretçi 300'e gelir. `lseek(fd, 100, SEEK_CUR)` işaretçiyi 100 ileri taşır ve 400 döndürür. İkinci `read` 400–699. baytları okur, 300 döndürür; işaretçi 700 olur. `lseek(fd, -50, SEEK_END)` işaretçiyi sondan 50 bayt geriye, 950'ye taşır ve 950 döndürür. Son `read` 300 bayt ister, ama dosyada yalnız 50 bayt kalmıştır; 50 döndürür.

`stat`, dosyanın içeriğine dokunmadan onun hakkındaki bilgileri bir yapıya doldurur: boyutu, sahibi, izin bitleri, son erişim ve değişiklik zamanları gibi. Güncel Linux bu temel çağrıların yanında `openat`, `pread` / `pwrite` (konum işaretçisini değiştirmeden belirli bir konumdan okuyup yazan çağrılar) ve `statx` gibi genişletilmiş sürümler de sunar.

2.5.4 Dizin yönetimi ve diğer çağrılar

Çağrı	Açıklama
`s = mkdir(name, mode)`	Yeni bir dizin oluşturur.
`s = rmdir(name)`	Boş bir dizini siler.
`s = link(name1, name2)`	`name1`'in gösterdiği dosyaya `name2` adlı yeni bir dizin girdisi oluşturur.
`s = unlink(name)`	Bir dizin girdisini siler.
`s = mount(special, name, flag)`	Bir dosya sistemini bağlar (Linux'taki biçimi: `mount(kaynak, hedef, tür, bayrak, veri)`).
`s = umount(special)`	Bir dosya sisteminin bağlantısını kaldırır.
`s = chdir(dirname)`	Çalışma dizinini değiştirir.
`s = chmod(name, mode)`	Dosyanın koruma (izin) bitlerini değiştirir.
`s = kill(pid, signal)`	Bir sürece sinyal gönderir; adına rağmen yalnız süreç öldürmek için kullanılmaz.
`seconds = time(&seconds)`	1 Ocak 1970 00.00 UTC'den (Unix epoch) bu yana geçen saniye sayısını verir.

Tablo 2.6: Dizin yönetimi ve diğer POSIX çağrıları.

Tablodaki son çağrı, yazılım dünyasının bilinen bir sorununa kaynaklık eder. Eski sistemlerde zaman değeri 32 bitlik işaretli bir tam sayıda (`time_t`) tutuluyordu. Böyle bir tam sayının alabileceği en büyük değer $2^{31} - 1 = 2.147.483.647$'dir. Bu kadar saniye, 1970'in başından itibaren sayıldığında 19 Ocak 2038 tarihine denk gelir. O andan sonra sayaç taşar ve tarih 1901'e geri döner. Bu sorun "2038 sorunu" adıyla anılır. Güncel sistemler zamanı 64 bitlik bir `time_t` ile tutarak sorunu ortadan kaldırır.

`chdir` çağrısının bir süreç özelliğini değiştirdiğine dikkat etmek gerekir. Çalışma dizini sürece aittir; bir çocuk süreç kendi çalışma dizinini değiştirse bile ebeveynin çalışma dizini aynı kalır. Kabuktaki `cd` komutunun ayrı bir program olarak değil, kabuğun kendi içinde yerleşik bir komut olarak gerçeklenmesi bu yüzdendir.

2.5.5 link ve unlink

UNIX'te bir dosyanın adı ile dosyanın kendisi ayrı şeylerdir. Dosyanın asıl bilgileri (sahibi, izinleri, boyutu, verinin diskteki blokları) **i-node** adı verilen bir yapıda tutulur. Dizin ise yalnız (i-node numarası, ad) çiftlerinden oluşan bir tablodur. Bir ad, bir i-node'u gösteren bir dizin girdisinden ibarettir.

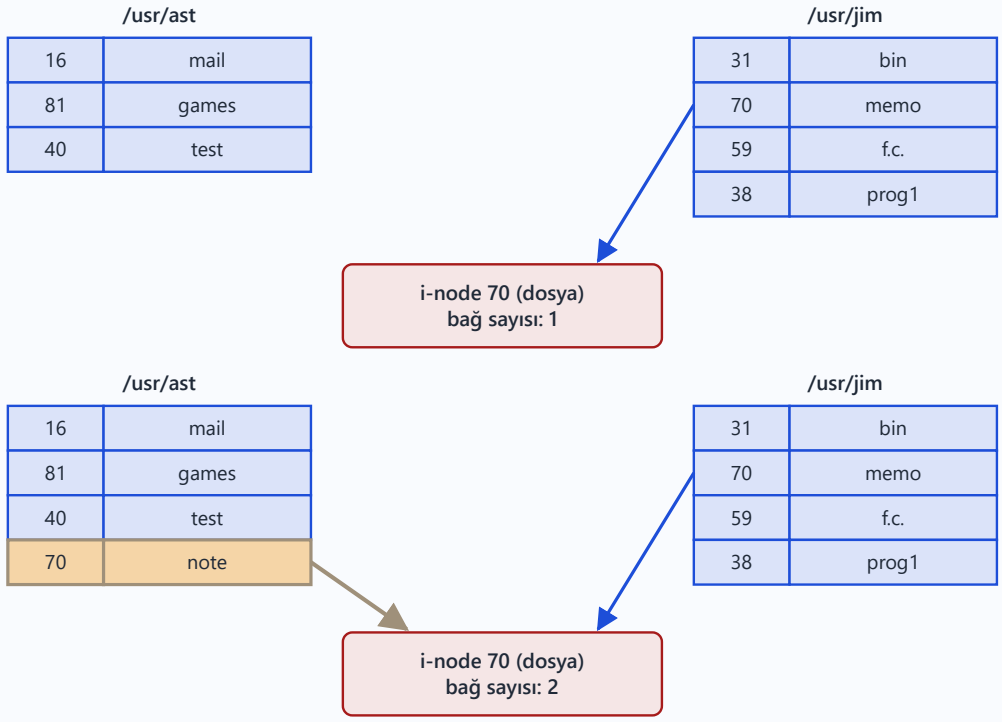
Bu ayrımın doğal bir sonucu vardır: aynı i-node'u gösteren birden çok dizin girdisi olabilir. Böylece bir dosyanın birden çok adı olur. Bu adların her birine **katı bağ** (*hard link*) denir. i-node, kendisini gösteren ad sayısını **bağ sayısı** adlı bir sayaçta tutar.

Örnek 2.10

`/usr/jim` dizininde `memo` adlı bir dosya vardır ve 70 numaralı i-node'u gösterir. Önce `link("/usr/jim/memo", "/usr/ast/note")`, sonra `unlink("/usr/jim/memo")` çağrılıyor.

1. **Başlangıç.** `/usr/jim` dizininde (31, bin), (70, memo), (59, f.c.), (38, prog1) girdileri vardır. i-node 70'in bağ sayısı 1'dir.
2. `link`. `/usr/ast` dizinine (70, note) girdisi eklenir. Veri kopyalanmaz; yeni girdi aynı i-node'u gösterir. Bağ sayısı 2 olur. Artık `/usr/jim/memo` ile `/usr/ast/note` aynı dosyadır: biri üzerinden yapılan değişiklik öteki üzerinden de görülür.
3. `unlink`. `/usr/jim`'deki (70, memo) girdisi silinir. Dosyanın kendisi silinmez, `note` adıyla yaşamaya devam eder. Bağ sayısı 1'e iner.

Bağ sayısı 0'a indiğinde ve dosyayı açık tutan bir süreç kalmadığında çekirdek i-node'u ve dosyanın veri bloklarını serbest bırakır. UNIX'te "dosya silmek" diye ayrı bir çağrının olmaması bundandır; `rm` komutu aslında `unlink` çağırır.



Şekil 2.21: `link` çağrısı. Üstte başlangıç: `/usr/jim/memo` 70 numaralı i-node'u gösterir, bağ sayısı 1'dir. Altta `link`'ten sonra `/usr/ast`'ta da 70'i gösteren `note` girdisi vardır; bağ sayısı 2'dir.

Dosyanın açık olması, bağ sayısından ayrı bir sayaçla izlenir. Bir süreç dosyayı `open` ile açtığında çekirdek yol adını çözer ve i-node'a ulaşır. Bundan sonra süreç dosyaya adı üzerinden değil, tanımlayıcı üzerinden doğrudan i-node'a giderek erişir. Ad silinse bile açık tanımlayıcı çalışmaya devam eder. Aşağıdaki kod bu durumu izler:

```
int fd = open("/usr/jim/memo", O_RDONLY);
link("/usr/jim/memo", "/usr/ast/note");
unlink("/usr/jim/memo");
unlink("/usr/ast/note");
n = read(fd, buf, 100);    /* hâlâ okunur */
close(fd);
```

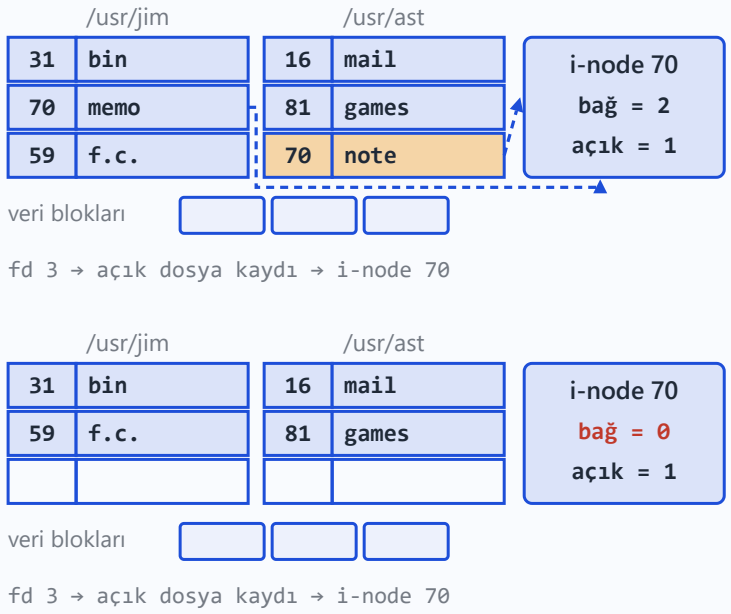
Örnek 2.11

`memo` 85 baytlık bir dosyadır ve sürecin yalnız 0, 1, 2 tanımlayıcıları açıktır. Tablo, her satırdan sonra i-node 70'in bağ sayısını ve açık sayacını verir.

Satır	Çağrı	Bağ sayısı	Açık	Not
—	başlangıç	1	0	yalnız `memo` adı var
1	`open`	1	1	`fd = 3`
2	`link`	2	1	`note` eklendi

Satır	Çağrı	Bağ sayısı	Açık	Not
3	`unlink(memo)`	1	1	yalnız `note` kaldı
4	`unlink(note)`	0	1	hiçbir yol yok, ama dosya açık
5	`read`	0	1	`n = 85`
6	`close`	0	0	i-node ve bloklar serbest

4. satırdan sonra dosyanın dizin ağacında hiçbir adı yoktur. Yine de 3 numaralı tanımlayıcı onu açık tuttuğu için çekirdek i-node'u ve blokları silmez. 5. satırdaki `read`, tanımlayıcı üzerinden doğrudan i-node'a gider; ada ihtiyacı yoktur. 100 bayt ister ve dosyada 85 bayt olduğu için 85 döndürür. Son tanımlayıcı 6. satırda kapanınca hem bağ sayısı hem açık sayacı 0 olur ve çekirdek ancak bu anda i-node'u ve veri bloklarını serbest bırakır.



Şekil 2.22: Örnek 2.11'in iki anı. İlk karede `link`'ten sonra: `/usr/jim/memo` ve `/usr/ast/note` aynı i-node'u gösterir, bağ sayısı 2'dir. İkinci karede iki `unlink`'ten sonra: dizinlerde dosyanın adı kalmamıştır, ama tanımlayıcı açık olduğu için i-node yaşar.

Programlar bu davranıştan bilerek yararlanır. Geçici bir dosya oluşturan program, dosyayı açtıktan hemen sonra `unlink` ile adını siler. Dosya, program çalıştığı sürece tanımlayıcı üzerinden kullanılabilir ama dizin ağacında görünmez. Program ister normal biçimde bitsin ister çöksün, son tanımlayıcı kapanınca çekirdek dosyayı kendiliğinden temizler.

2.5.6 Win32 API karşılıkları

Windows'un programlama arayüzü Win32 API'dir. Bu arayüz binlerce fonksiyon içerir ve POSIX'teki temel çağrıların çoğunun bir karşılığı vardır. Tablo 2.7 Tanenbaum'un listesini güncel karşılıklarıyla verir.

UNIX	Win32	UNIX	Win32
`fork`	`CreateProcess`	`mkdir`	`CreateDirectory`
`waitpid`	`WaitForSingleObject`	`rmdir`	`RemoveDirectory`
`execve`	(yok: `CreateProcess`)	`link`	`CreateHardLink`
`exit`	`ExitProcess`	`unlink`	`DeleteFile`
`open`	`CreateFile`	`mount`	`SetVolumeMountPoint`
`close`	`CloseHandle`	`umount`	`DeleteVolumeMountPoint`
`read`	`ReadFile`	`chdir`	`SetCurrentDirectory`
`write`	`WriteFile`	`chmod`	(yok: ACL, `SetFileSecurity`)
`lseek`	`SetFilePointerEx`	`kill`	(yok: `TerminateProcess`)
`stat`	`GetFileAttributesEx`	`time`	`GetSystemTime`

Tablo 2.7: UNIX çağrılarının Win32 API karşılıkları.

Tablodaki "yok" işaretleri tasarım farklarından kaynaklanır. `execve`'nin karşılığı yoktur, çünkü Windows süreç oluşturma ile program yüklemeyi tek `CreateProcess` çağrısında birleştirir. `chmod`'un karşılığı yoktur, çünkü Windows dosya izinlerini UNIX'in dokuz izin biti yerine erişim denetim listeleriyle (*access control list*, ACL) yönetir. `kill`'in karşılığı yoktur, çünkü Windows'ta UNIX'teki gibi bir sinyal mekanizması bulunmaz; bir süreci sonlandırmak için `TerminateProcess` kullanılır. Tanenbaum'un kitabının eski baskılarında karşılığı olmayan `link` ve `mount` ise bugün `CreateHardLink` ve `SetVolumeMountPoint` ile mümkündür.

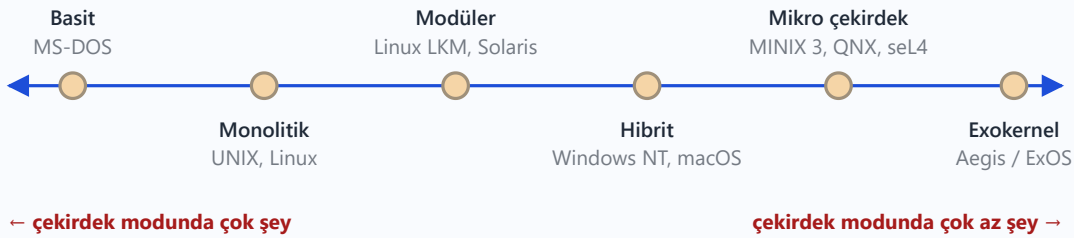
Win32 fonksiyonları çekirdeğe doğrudan geçmez. Asıl çekirdek geçişini, `ntdll.dll` kütüphanesindeki `Nt…` ile başlayan fonksiyonlar yapar; örneğin `CreateFile` sonunda `NtCreateFile`'ı çağırır. Bu fonksiyonların çekirdekteki çağrı numaraları Windows sürümünden sürümüne değişir ve belgelenmez. Programlar bu yüzden her zaman API'yi kullanır; numaralara doğrudan dayanan bir program bir sonraki Windows güncellemesinde çalışmaz hâle gelebilir.

2.6 Çekirdek yapıları

Sistem çağrıları, işletim sisteminin dışarıya bakan yüzüdür. Kesimin konusu ise içeridir: çağrıyı karşılayan bileşenler nasıl düzenlenir? Bu soruya verilen cevaplar onlarca yıl içinde birkaç ana yapı etrafında toplanmıştır. Her yapı, 2.1.1'deki ödünleşim üçgeninde farklı bir noktayı seçer.

2.6.1 Genel bakış

Yapıları sıralamanın kullanışlı bir yolu, çekirdek modunda ne kadar iş yapıldığına bakmaktır. Şekil 2.23 yapıları bu ölçüte göre bir doğru üzerine dizer.

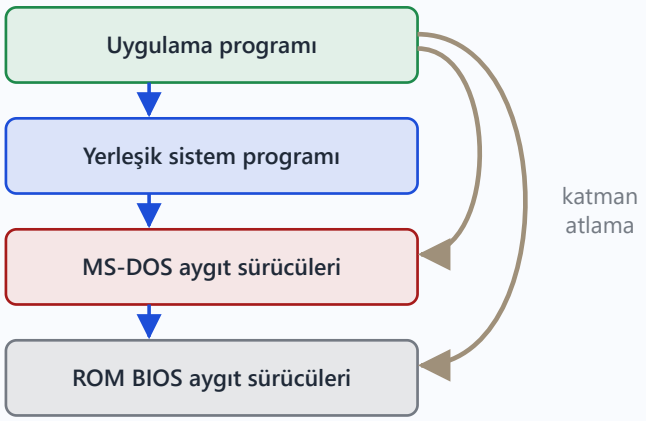


Şekil 2.23: Çekirdekte bulunan işlev miktarına göre yapılar. Soldan sağa: basit yapı (MS-DOS), monolitik (UNIX, Linux), modüler (Linux LKM, Solaris), hibrit (Windows NT, macOS), mikro çekirdek (MINIX 3, QNX, seL4) ve exokernel (Aegis/ExOS). Sol uçta çekirdek modunda çok şey, sağ uçta çok az şey vardır.

İki kavram bu doğruya tam oturmaz. **Katmanlı yapı** bir yerleşim değil, bir düzenleme ilkesidir: bileşenler hiyerarşik katmanlara ayrılır. THE sistemi ve MULTICS'in koruma halkaları bu ilkenin örnekleridir ve ilke, doğrunun herhangi bir noktasındaki bir sisteme uygulanabilir. **Sanal makineler** ise ayrı bir eksendir: donanımın kendisi sanallaştırılır ve tek bir makinede birden çok işletim sistemi çalışır.

2.6.2 Basit yapı: MS-DOS

Bazı işletim sistemlerinin belirgin bir yapısı yoktur. Bu sistemler küçük, basit ve sınırlı donanım için yazılmış, zamanla beklenenin ötesinde büyümüştür. MS-DOS bunun en bilinen örneğidir.



Şekil 2.24: MS-DOS'un katmanları: uygulama programı, yerleşik sistem programı, MS-DOS aygıt sürücüleri ve ROM BIOS aygıt sürücüleri. Turuncu oklar, uygulamaların aradaki katmanları atlayarak alt katmanlara doğrudan erişebildiğini gösterir.

MS-DOS'ta arayüzler ve işlev düzeyleri iyi ayrılmamıştır. Bir uygulama, MS-DOS'un sunduğu hizmetleri kullanmak yerine BIOS rutinlerini doğrudan çağırabilir, hatta ekran belleğine ya da aygıt portlarına doğrudan yazabilir. Bunun temel nedeni donanımdır: MS-DOS'un yazıldığı Intel 8088 işlemcisinde çift mod yoktu. Donanım koruması olmadığı için tek bir hatalı program bütün sistemi çökertebiliyordu. Modüller arasında net sınırlar ve **veri gizleme** (*data hiding*) da yoktu: her bileşen ötekinin verisine dokunabiliyordu. Silberschatz bu tür sistemleri "basit yapı" başlığı altında, monolitik çekirdeklerden ayrı ele alır.

2.6.3 Monolitik sistem

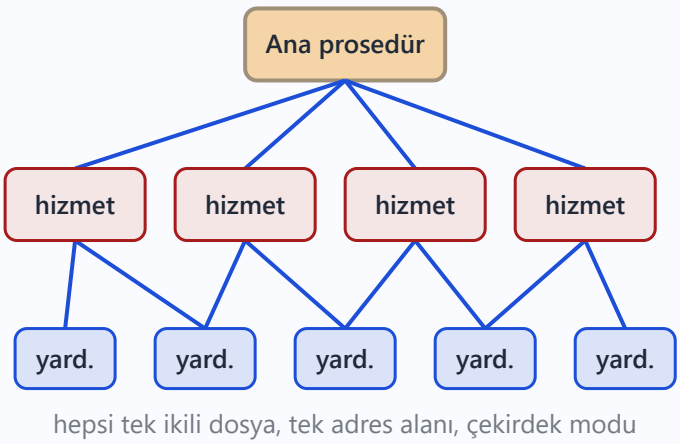
En yaygın yapı, bütün işletim sisteminin çekirdek modunda çalışan tek bir program olarak bağlandığı yapıdır.

Tanım 2.10

Monolitik çekirdekte (*monolithic kernel*) bütün işletim sistemi hizmetleri tek bir adres alanında, çekirdek modunda çalışır. Çekirdekteki her prosedür ötekini doğrudan çağırabilir.

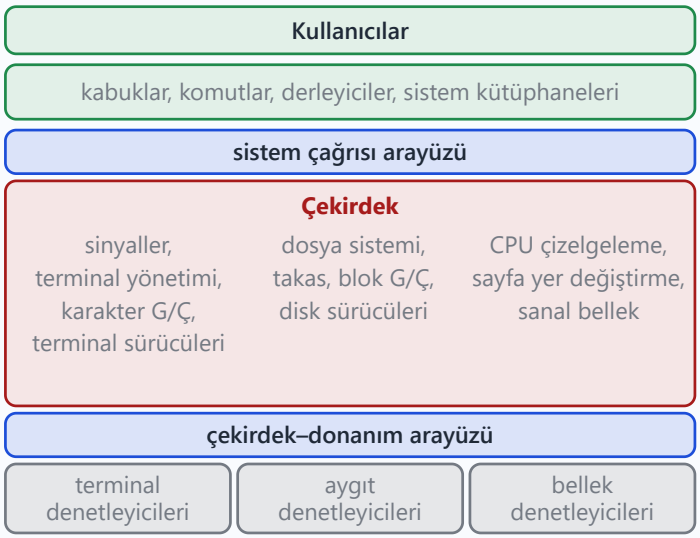
"Monolitik" sözcüğü düzensizlik anlamına gelmez. Tanenbaum, monolitik bir sistemin bile temel bir düzeni olduğunu üç katmanla anlatır:

1. **Ana program.** Sistem çağrısının numarasına bakar ve istenen hizmet prosedürünü başlatır.
2. **Hizmet prosedürleri.** Her biri bir sistem çağrısının işini yapar.
3. **Yardımcı prosedürler.** Birden çok hizmetin ortak ihtiyacı olan işleri görür; örneğin kullanıcı programının belleğinden veri almak.



Şekil 2.25: Monolitik sistemin iç düzeni. Ana prosedür hizmet prosedürlerini çağırır; hizmet prosedürleri yardımcı prosedürleri ortaklaşa kullanır. Hepsi tek bir ikili dosyada, tek bir adres alanında ve çekirdek modundadır.

Geleneksel UNIX ve Linux monolitik çekirdeklerdir. Şekil 2.26 geleneksel UNIX'in katmanlarını gösterir. Kullanıcılar ve sistem programları (kabuklar, komutlar, derleyiciler, kütüphaneler) sistem çağrısı arayüzünün üstündedir. Bu arayüzün altındaki çekirdek; sinyalleri, terminal yönetimini, dosya sistemini, takas alanını, disk sürücülerini, işlemci çizelgelemesini ve sanal belleği tek bir bütün olarak barındırır. Çekirdeğin altında, çekirdek ile donanım arasındaki arayüz ve aygıt denetleyicileri bulunur.



Şekil 2.26: Geleneksel UNIX sistem yapısı. Sistem çağrısı arayüzü ile çekirdek–donanım arayüzü arasındaki her şey tek bir çekirdektir.

Monolitik yapının güçlü yanları açıktır. Bileşenler birbirini doğrudan fonksiyon çağrısıyla kullanır; arada ileti hazırlama, kopyalama ya da mod geçişi yoktur. Bu yüzden monolitik çekirdekler hızlıdır. Tasarımları ve gerçeklemeleri de görece basittir. Zayıf yanları ise aynı yakınlıktan doğar. Bütün bileşenler aynı adres alanında olduğu için tek bir sürücüdeki hatalı bir işaretçi, çekirdeğin herhangi bir yapısını bozabilir ve bütün sistemi çökertebilir. Kod tabanı da devasadır: 40 milyon satırı aşan Linux'ta bir hatayı bulmak ve bakım yapmak kolay değildir. Çoğu monolitik çekirdek bugün sürücüleri ve dosya sistemlerini çalışma anında yüklenebilen modüller olarak da alır; bu ek esnekliğe 2.6.9'da değinilir.

2.6.4 Katmanlı yapı

Büyük bir sistemi yönetilebilir kılmanın klasik yolu onu katmanlara bölmektir.

Tanım 2.11

Katmanlı yapıda (*layered system*) işletim sistemi katmanlara bölünür. En alttaki katman donanıma, en üstteki katman kullanıcıya en yakındır. Her katman yalnız kendi altındaki katmanların hizmetlerini kullanır.

Bu yapının ilk örneği, Edsger Dijkstra'nın öğrencileriyle 1968'de geliştirdiği THE sistemidir. Şekil 2.27 THE'nin altı katmanını gösterir.



Şekil 2.27: THE sisteminin katmanları. 0: işlemci tahsisi ve çoklu programlama; 1: bellek ve tambur yönetimi; 2: operatör–süreç iletişimi; 3: G/Ç yönetimi; 4: kullanıcı programları; 5: operatör.

- **Katman 0**, kesmeler geldiğinde ve zamanlayıcı dolduğunda süreçler arasında geçiş yapar. Bu katmanın üstündekiler, birden çok sürecin tek işlemciyi paylaştığını düşünmek zorunda kalmaz; her süreç kendi işlemcisi varmış gibi yazılır.
- **Katman 1**, süreçlere bellek ayırır ve sayfaları o dönemin ikincil belleği olan tambur ile ana bellek arasında taşır. Üst katmanlar, sayfanın bellekte mi tamburda mı olduğunu bilmez.
- **Katman 2**, her süreç ile operatör konsolu arasındaki iletişimi yönetir. Üstteki her süreç kendi konsolu varmış gibi davranır.
- **Katman 3**, G/Ç aygıtlarını yönetir. Üst katmanlar gerçek aygıtların ayrıntıları yerine soyut aygıtlarla çalışır.
- **Katman 4**, kullanıcı programlarıdır. Bu programlar bellek, konsol ve G/Ç ayrıntılarıyla uğraşmaz.
- **Katman 5**, sistem operatörüdür.

Katmanlı yapının en büyük yararı, hata ayıklamanın ve doğrulamanın katman katman yapılabilmesidir. Katman 0 tek başına sınanır ve doğruluğundan emin olunur. Sonra katman 1 eklenir; bir hata çıkarsa, alttaki katman doğru olduğuna göre hata katman 1'dedir. Soyutlama ve veri gizleme de kolaydır: her katman kendi veri yapılarını üst katmanlardan saklar.

Yapının iki önemli zorluğu vardır. Birincisi, katmanları doğru tanımlamaktır. Bileşenler arasındaki bağımlılıklar çoğu zaman iki yönlüdür. Örneğin bellek yöneticisi diske sayfa yazmak için G/Ç sürücüsüne ihtiyaç duyar; G/Ç sürücüsü de tampon ayırmak için bellek yöneticisine. Hangisinin altta olacağına karar vermek güçtür. İkincisi, performanstır: bir istek her katman geçişinde parametre hazırlama ve

denetim gibi ek işler doğurur. Bu yüzden saf katmanlı işletim sistemleri yaygınlaşmamıştır. Fikir ise bugün ağ protokol yığınlarında yaşar: bir ağ paketi uygulama, taşıma, ağ ve bağlantı katmanlarından sırayla geçer.

2.6.5 Mikro çekirdek

Monolitik yapının en büyük sorunu, bir bileşendeki hatanın bütün sistemi düşürebilmesidir. Mikro çekirdek yaklaşımı bu soruna kökten bir cevap verir: çekirdekte yalnız gerçekten orada bulunması gereken şeyler kalır.

> **Tanım 2.12**
>
> **Mikro çekirdekte** (*microkernel*) çekirdek modunda yalnız süreçler arası iletişim (ileti geçirme), temel çizelgeleme, adres alanı yönetimi ve kesme işleme kalır. Sürücüler, dosya sistemleri ve ağ yığını kullanıcı modunda **sunucu süreçler** olarak çalışır ve birbirleriyle iletilerle haberleşir.

Yeni bir hizmet eklemek, mikro çekirdekte kullanıcı alanına yeni bir sunucu eklemek demektir; çekirdeğin kendisi değişmez. Mach, MINIX 3, otomotiv sektöründe yaygın QNX Neutrino, L4 ailesi, biçimsel olarak doğruluğu kanıtlanmış seL4 ve Google Fuchsia'nın Zircon çekirdeği bu yaklaşımın örnekleridir. macOS, Mach'tan türemiş olsa da saf bir mikro çekirdek değil, hibrit bir çekirdektir.

Mikro çekirdek tasarımında sık anılan bir ilke, **mekanizma ile politikanın ayrılmasıdır** (*policy / mechanism*). Mekanizma bir işin *nasıl* yapılacağını, politika *ne* yapılacağını belirler. Örneğin çekirdek, en yüksek öncelikli hazır süreci çalıştırma mekanizmasını sağlar. Süreçlere hangi önceliklerin verileceğine ise kullanıcı modundaki bir bileşen karar verir. Böylece politika değiştirilmek istendiğinde çekirdeğe dokunmak gerekmez.



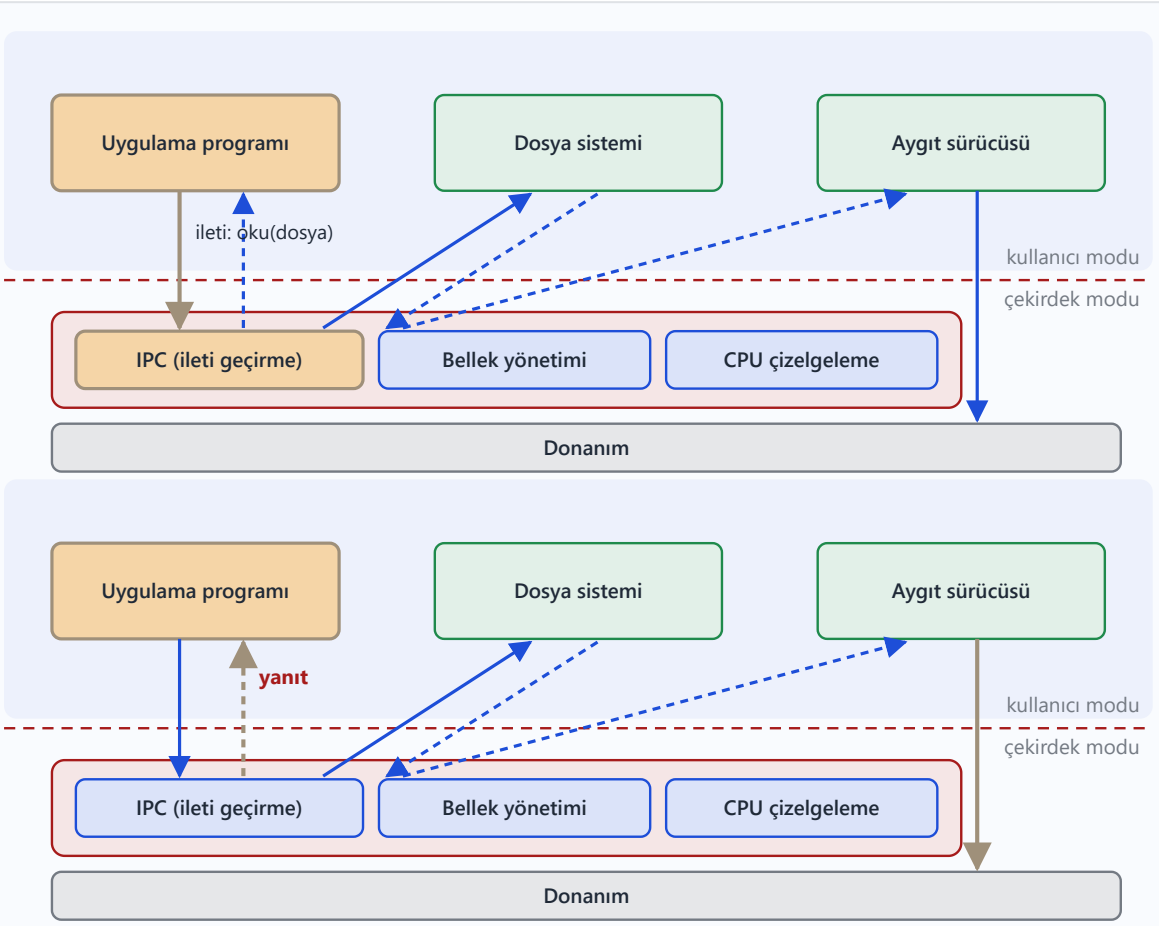
Şekil 2.28: Monolitik çekirdek ile mikro çekirdeğin karşılaştırılması. Solda dosya sistemi, sürücüler ve ağ yığını çekirdek modundadır. Sağda dosya sistemi ve sürücü kullanıcı modunda ayrı süreçlerdir; çekirdekte yalnız IPC ve çizelgeleme kalmıştır.

Mikro çekirdeğin artıları şunlardır. Çöken bir sürücü sistemi düşürmez; yalnız o sürücünün süreci ölür ve yeniden başlatılabilir. Çekirdek küçük olduğu için sınanması, doğrulanması ve başka bir donanıma taşınması kolaydır; seL4'ün birkaç bin satırlık çekirdeğinin doğruluğu matematiksel olarak ka-

nıtlanabilmiştir. Ayrıcalıklı kod az olduğu için saldırganın hedef alabileceği yüzey de küçüktür. Eksisi ise performanstır: bileşenler arası her istek bir ileti ve birkaç mod geçişi gerektirir. L4 ailesinin tasarımları, ileti geçirmeyi son derece hızlandırarak bu ek yükü büyük ölçüde azaltmıştır.

2.6.6 Mikro çekirdekte ileti alışverişi

Mikro çekirdekte bir uygulamanın bir dosyadan okuma yapması, monolitik çekirdektekinden çok daha uzun bir yol izler. Şekil 2.29 bu yolun iki anını gösterir.



Şekil 2.29: Mikro çekirdekte bir dosya okuma. Üstte uygulama, dosya sistemi sunucusuna bir ileti gönderir; ileti çekirdeğin IPC mekanizmasından geçer. Altta aygıt sürücüsü, çekirdeğin izin verdiği G/Ç ile donanıma erişmiş; yanıt aynı yoldan uygulamaya dönmektedir.

Uygulama isteğini bir ileti olarak çekirdeğe verir; çekirdek iletiyi kullanıcı modundaki dosya sistemi sunucusuna teslim eder. Dosya sistemi hangi disk bloğunun okunacağını bulur, ama diske kendisi erişemez; bu yüzden yine çekirdek üzerinden disk sürücüsüne bir ileti yollar. Sürücü, çekirdeğin izin verdiği G/Ç işlemleriyle bloğu okur. Yanıtlar da aynı yolu tersinden izler. Monolitik çekirdekte aynı iş, tek bir sistem çağrısının içinde birbirini çağıran fonksiyonların zinciridir.

Yolu daha ayrıntılı görmek için istemci ile dosya sistemi sunucusunun (FS) kodunu izleyelim. Aşağıdaki kod, `send` ve `receive` adlı iki ilkel çağrıyla yazılmıştır. `send(hedef, &m)` iletiyi hedefe gönderir; `receive(kaynak, &m)` belirtilen kaynaktan (ya da `ANY` ise herhangi birinden) bir ileti gelene kadar bekler.

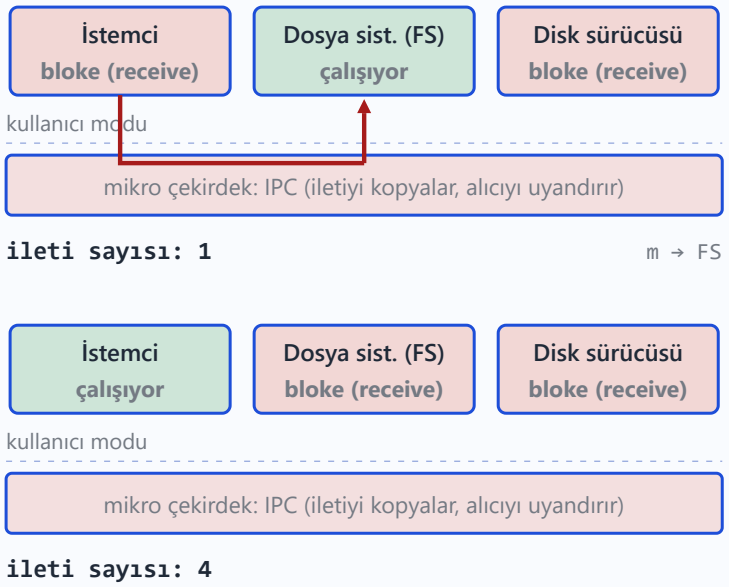
```
/* istemci */
m.tur = OKU; m.dosya = 7; m.ofset = 0;
send(FS, &m);
receive(FS, &m);       /* yanıtı bekle */
/* dosya sistemi sunucusu (FS) */
while (1) {
  receive(ANY, &m);
  blok = blok_bul(m.dosya, m.ofset);
  send(DISK, &blok);
  receive(DISK, &veri);
  m.sonuc = veri;
  send(m.kaynak, &m);
}
```

Örnek 2.12

İstemci, 7 numaralı dosyanın 0. baytından okumak istiyor. Dosya sistemi bu konumun 120 numaralı blokta olduğunu biliyor. Başlangıçta FS ve disk sürücüsü `receive`'de iş bekliyor.

1. **İstek hazırlanır.** İstemci isteğini ileti yapısına yazar: `m.tur = OKU`, `m.dosya = 7`, `m.ofset = 0`.
2. **1. ileti.** `send(FS, &m)` çekirdeğe bir tuzaktır. Çekirdek iletiyi FS'nin tamponuna kopyalar ve FS'yi uyandırır. İstemci, yanıt için `receive(FS, &m)`'de bloke olur.
3. **FS işi alır.** FS'nin `receive(ANY, &m)` çağrısı döner. İletinin kimden geldiği `m.kaynak` alanındadır. FS, `blok_bul` ile istenen verinin 120 numaralı blokta olduğunu bulur.
4. **2. ileti.** FS de diske doğrudan erişemez. `send(DISK, &blok)` ile sürücüye ikinci iletiyi yollar ve `receive(DISK, &veri)` ile yanıtı bekler.
5. **3. ileti.** Sürücü, çekirdeğin izin verdiği G/Ç ile bloğu okur ve üçüncü iletiyle veriyi FS'ye gönderir. FS, `m.sonuc = veri` ile yanıtı hazırlar.
6. **4. ileti.** `send(m.kaynak, &m)` yanıtı istemciye taşır; istemci hazır duruma geçer.
7. **FS döngüye döner.** FS, `while` döngüsünün başındaki `receive(ANY, &m)`'de yeni bir iş beklemeye başlar.
8. **İstemci devam eder.** İstemcinin `receive` çağrısı döner; veri `m.sonuc` alanındadır.

Tek bir okuma için dört ileti gönderilmiştir. Her iletide en az bir kullanıcı → çekirdek ve bir çekirdek → kullanıcı geçişi olur. Monolitik çekirdekte aynı iş tek bir sistem çağrısı içinde, iki mod geçişi ve birkaç fonksiyon çağrısıyla biter.



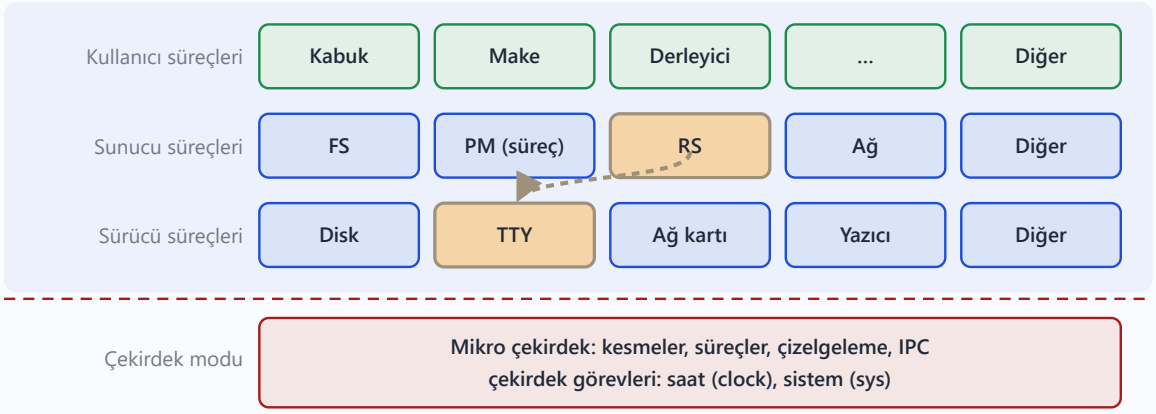
Şekil 2.30: Örnek 2.12'nin iki anı. İlk karede birinci ileti: istemci isteğini göndermiş ve yanıtı beklemektedir, FS uyanmıştır. İkinci karede sonuç: dört ileti tamamlanmış, istemci veriyi almıştır.

Sık yapılan hata

Mikro çekirdeğin ek yükünü yalnız ileti sayısıyla ölçmek. Her ileti, iki mod geçişinin yanında genellikle bir bağlam değiştirme de gerektirir. İşlemci istemci sürecinden sunucu sürecine geçer ve sunucunun adres alanı etkin hâle gelir. Önbellekteki veriler de büyük ölçüde işe yaramaz olur. Ek yükün asıl kaynağı çoğu zaman bu bağlam değiştirmelerdir.

2.6.7 MINIX 3 ve istemci–sunucu modeli

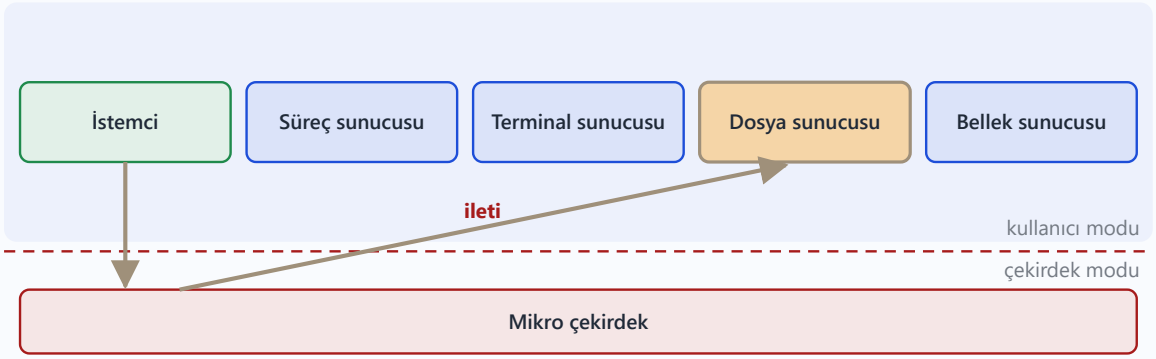
MINIX 3, Andrew Tanenbaum ve ekibinin geliştirdiği, güvenilirliği ön plana çıkaran bir mikro çekirdek işletim sistemidir. Şekil 2.31 yapısını gösterir.

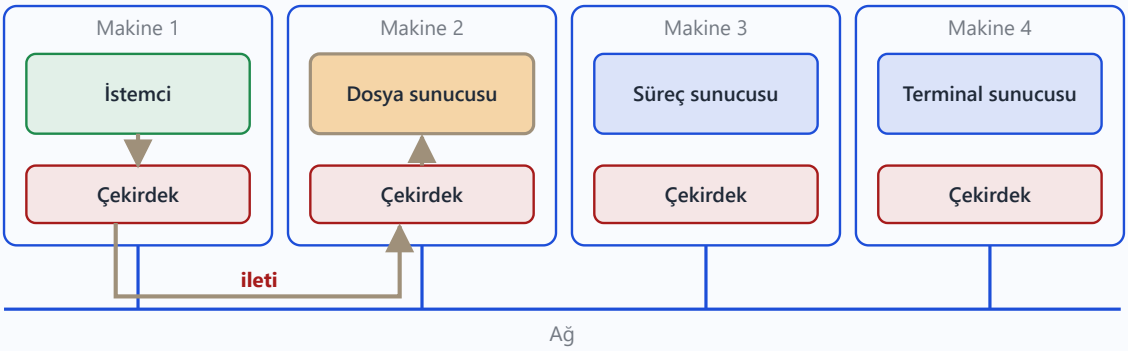


Şekil 2.31: MINIX 3'ün yapısı. Kullanıcı süreçleri (kabuk, make, derleyici), sunucu süreçleri (FS, süreç yöneticisi PM, reenkarnasyon sunucusu RS, ağ) ve sürücü süreçleri (disk, TTY, ağ kartı, yazıcı) kullanıcı modundadır. Çekirdek modunda yalnız kesmeleri, süreçleri, çizelgelemeyi ve IPC'yi yöneten mikro çekirdek ile saat ve sistem görevleri bulunur. Kesikli ok, RS'nin TTY sürücüsünü yeniden başlatmasını gösterir.

MINIX 3'ün iki özelliği dikkat çekicidir. Birincisi **reenkarnasyon sunucusudur** (RS). RS, sürücülerin sağlığını sürekli denetler. Bir sürücü, örneğin terminal sürücüsü (TTY), çökerse ya da yanıt vermeyi keserse RS onu kullanıcı fark etmeden yeniden başlatır. Sistem kendi kendini onarır. İkincisi **bölünmüş ayrıcalıktır**. Sürücüler kullanıcı modunda çalışır ve G/Ç portlarına yalnız çekirdek çağrılarıyla, kendilerine izin verilen ölçüde erişebilir. Hatalı bir sürücü kendi belleğini bozabilir, ama çekirdeğin ya da başka bir sürecin belleğine dokunamaz.

Mikro çekirdek üzerinde çalışan süreçleri iki sınıfa ayırmak doğaldır: hizmet isteyen **istemciler** ve hizmet veren **sunucular**. Haberleşme yalnız iletilerle olur. Bu düzene **istemci–sunucu modeli** (*client–server model*) denir.





Şekil 2.32: İstemci–sunucu modeli. Üstte tek bir makinede istemci, dosya, süreç ve terminal sunucuları mikro çekirdek üzerinde çalışır ve iletiler çekirdek üzerinden geçer. Altta her süreç ayrı bir makinededir; istemcinin iletisi ağ üzerinden dosya sunucusunun makinesine gider.

Modelin önemli bir sonucu vardır. İstemci, iletinin kendi makinesinde mi yoksa ağdaki başka bir makinede mi işlendiğini bilmek zorunda değildir: ileti gönderir ve yanıtı bekler. Bu yüzden model, doğal olarak **dağıtık sistemlere** genişler. Bugünün web uygulamaları ve mikroservis mimarileri aynı fikrin geniş ölçekli uygulamalarıdır. Yine de ileti alışverişi kendiliğinden güvenli değildir. Güvenlik, çekirdeğin hangi sürecin hangi sunucuya ileti gönderebileceğini denetlemesine bağlıdır.

2.6.8 Tanenbaum–Torvalds tartışması

1991'de Linus Torvalds, Linux çekirdeğinin ilk sürümünü yayımladı. Birkaç ay sonra, 1992'nin başında, MINIX'in yazarı Andrew Tanenbaum bir haber grubunda “LINUX is obsolete” (Linux eskimiştir) başlıklı bir ileti yazdı. Bu ileti, işletim sistemi tarihinin en çok anılan tartışmalarından birini başlattı.

Tanenbaum'un savı şuydu: monolitik tasarım 1970'lerin geride kalmış bir yaklaşımıdır. Mikro çekirdek daha modüler, daha esnek ve daha kolay taşınabilirdir; bir bileşendeki hata sınırlı zarar verir. Donanım hızlandıkça ileti alışverişinin ek yükü önemsizleşecektir. Torvalds ise şöyle karşılık verdi: monolitik çekirdeğin tasarımı ve gerçeklemesi daha basittir ve doğrudan fonksiyon çağrıları daha hızlıdır. Süreçler arası ileti alışverişine gerek yoktur; asıl önemli olan pratikte çalışan koddur. Ayrıca açık kaynak geliştirme modeli, Linux'un hızla gelişmesini sağlayacaktır.

Aradan geçen otuz yılda iki taraf da kısmen haklı çıktı. Linux monolitik kaldı ve dünyanın en yaygın çekirdeklerinden biri oldu. Ama esneklik için mikro çekirdek fikirlerinden de yararlandı. Yüklenebilir modüller ve kullanıcı alanında çalışan dosya sistemleri (FUSE) ile sürücüler (VFIO) bunun örnekleridir. Çekirdeğe güvenli biçimde küçük programlar yüklemeyi sağlayan eBPF de aynı yöndedir. Mikro çekirdekler ise güvenilirliğin performanstan önemli olduğu güvenlik-kritik ve gömülü sistemlerde (QNX, seL4) yaygınlaştı.

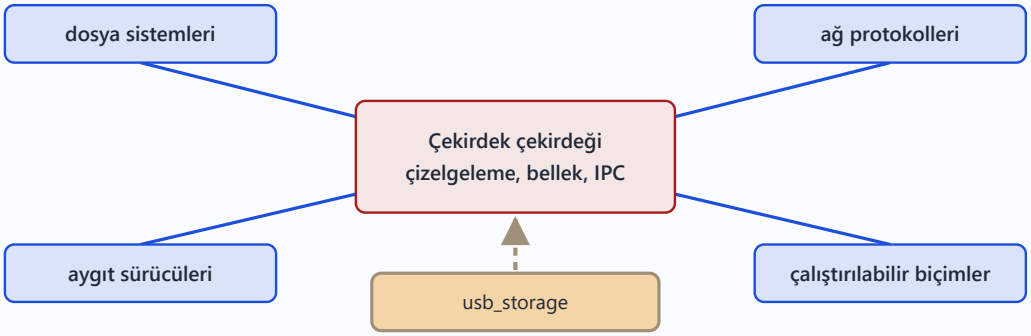
2.6.9 Modüler yapı

Günümüz işletim sistemlerinin çoğu, monolitik çekirdeğin hızını korurken esneklik kazanmak için modüler bir yaklaşım izler.

Tanım 2.13

Modüler yapıda çekirdekte yalnız temel bileşenler bulunur; öteki hizmetler açılışta ya da çalışma anında **yüklenebilir çekirdek modülü** (*loadable kernel module*, LKM) olarak çekirdeğe eklenir. Modüller çekirdek modunda ve çekirdeğin adres alanında çalışır.

Linux, FreeBSD ve Solaris bu yapıyı kullanır. Modüler yapı iki ayrı yapıya benzer, ama ikisinden de farklıdır. Katmanlı yapıya benzer, çünkü her modülün tanımlı bir arayüzü vardır; ondan daha esnektir, çünkü her modül herhangi bir modülü çağırabilir. Mikro çekirdeğe benzer, çünkü çekirdeğin özü küçüktür ve hizmetler ayrı parçalar hâlindedir; ondan farkı, modüllerin ileti kullanmadan doğrudan fonksiyon çağrılarıyla haberleşmesi ve çekirdek modunda çalışmasıdır.



Şekil 2.33: Yüklenebilir çekirdek modülleri. Ortadaki çekirdek özü çizelgeleme, bellek ve IPC'yi barındırır; dosya sistemleri, ağ protokolleri, aygıt sürücüleri ve çalıştırılabilir dosya biçimleri modül olarak eklenir. USB bellek takıldığında `usb_storage` modülü yüklenir; `vfat` modülü bellek bağlanırken eklenir.

Bir modülün yaşam döngüsü Linux'ta şöyle izlenebilir. Açılışta çekirdek ve başlangıç RAM diskindeki (initramfs) gerekli modüller yüklüdür; `lsmod` komutu yüklü modülleri listeler. Kullanıcı bir USB bellek takar. Çekirdek bir olay üretir. `udev` hizmeti bu olayı görür ve `modprobe` ile `usb-storage` modülünü bağımlılıklarıyla birlikte yükletir. Bellekteki FAT dosya sistemini okumak için gereken `vfat` modülü ise bu sırada değil, bellek bağlanırken (`mount` çağrısı dosya sistemi türünü istediğinde) yüklenir. Modül yüklenince kendini çekirdeğe kaydeder ve bundan sonra doğrudan fonksiyon çağrılarıyla çalışır. Modülü kullanan kalmayınca, yani kullanım sayısı 0'a inince `modprobe -r` ya da `rmmod` komutu onu bellekten çıkarır.

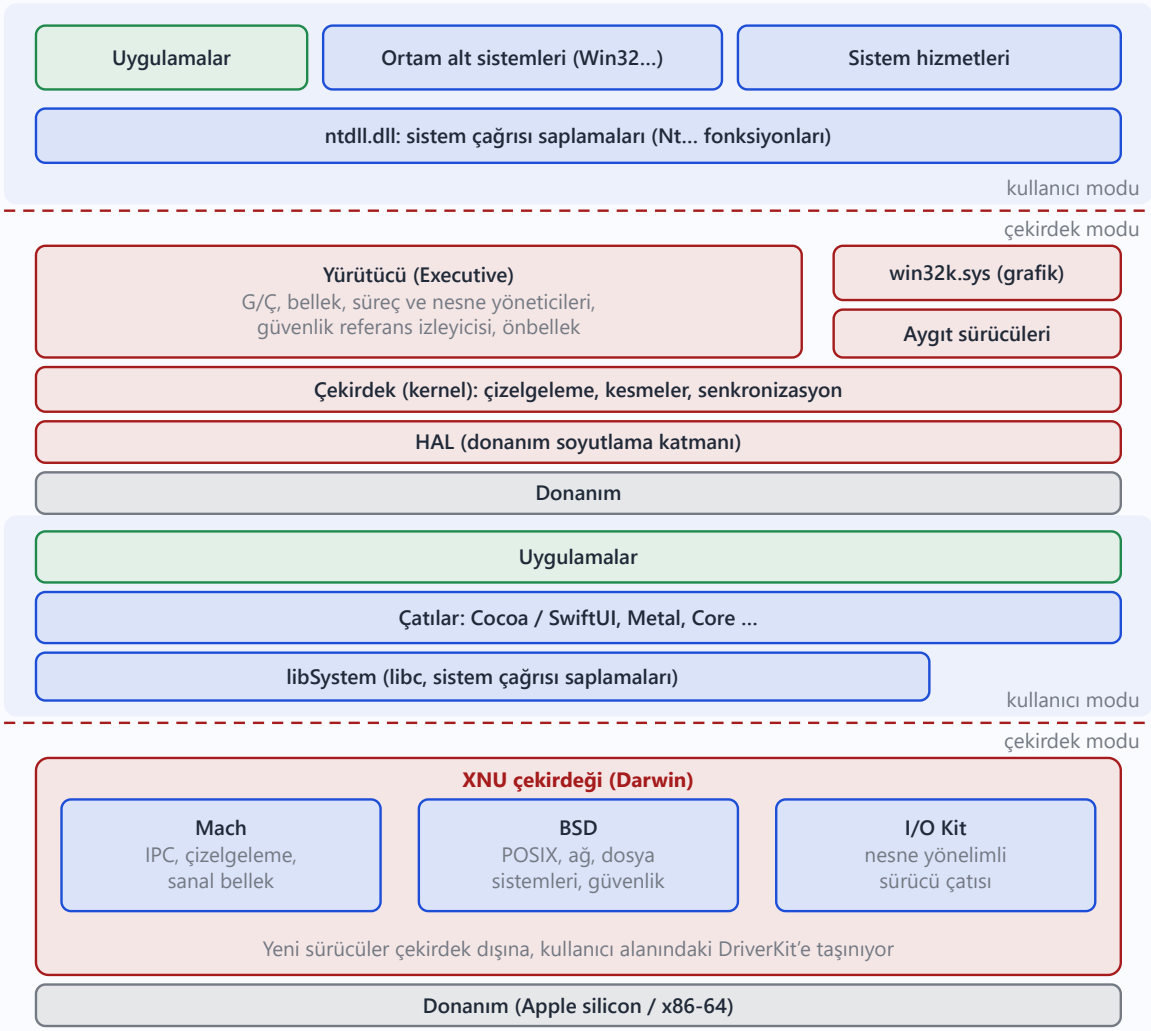
Modüler yapının bedeli yalıtımın olmamasıdır. Modül çekirdek modunda çalıştığı için hatalı bir modül çekirdeği çökertebilir; kötü niyetli bir modül sistemin bütün denetimini ele geçirebilir. Ayrıca modüllerin, yüklendikleri çekirdeğin sürümüyle uyumlu derlenmiş olması gerekir.

Sık yapılan hata

Yüklenebilir modülleri olduğu için Linux'u bir mikro çekirdek saymak. Mikro çekirdeği tanımlayan, bileşenlerin ayrı parçalar olması değil, kullanıcı modunda ayrı süreçler olarak çalışıp iletilerle haberleşmesidir. Linux modülleri çekirdek modunda, çekirdeğin adres alanında çalışır ve fonksiyon çağrısıyla haberleşir. Linux bu yüzden modüler bir monolitik çekirdektir.

2.6.10 Hibrit çekirdek

Bazı sistemler, mikro çekirdek fikirleriyle yola çıkıp performans kaygısıyla birçok hizmeti yeniden çekirdek moduna almıştır. Bu sistemlere **hibrit çekirdek** (*hybrid kernel*) denir. Windows ve macOS en bilinen iki örnektir.



Şekil 2.34: İki hibrit çekirdek. Üstte Windows NT: kullanıcı modunda uygulamalar, ortam alt sistemleri, sistem hizmetleri ve `ntdll.dll`; çekirdek modunda yürütücü, `win32k.sys`, aygıt sürücüleri, çekirdek ve HAL. Altta macOS ve iOS'un XNU çekirdeği: Mach, BSD ve I/O Kit aynı çekirdek adres alanında birleşmiştir.

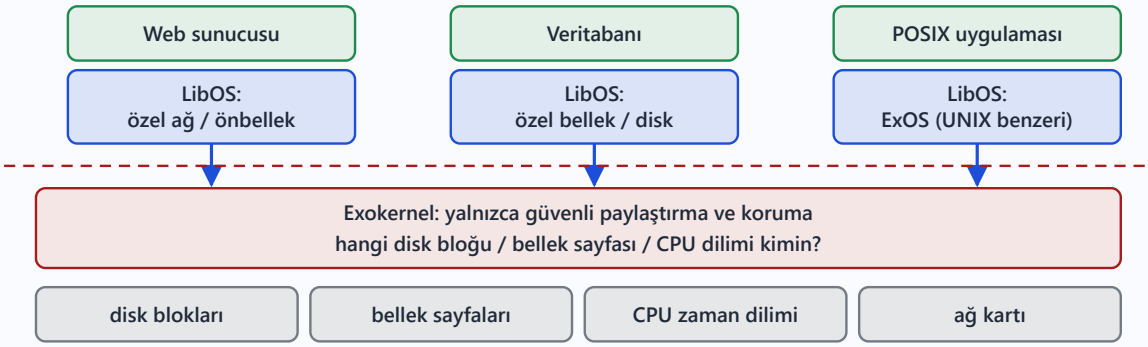
Windows NT'nin tasarımı, mikro çekirdek fikirlerinden izler taşır: alt sistemler, nesne ve ileti temelli bir yapı, donanımı soyutlayan bir katman (*hardware abstraction layer*, HAL). Ama performans için G/Ç, bellek, süreç ve nesne yöneticilerini içeren yürütücü (*executive*), grafik alt sistemi ve aygıt sürücüleri çekirdek moduna alınmıştır. macOS ve iOS'un çekirdeği XNU ise Mach mikro çekirdeği ile BSD UNIX katmanını aynı çekirdek adres alanında birleştirir. Mach IPC'yi, çizelgelemeyi ve sanal belleği, BSD POSIX arayüzünü, ağı ve dosya sistemlerini, I/O Kit de nesne yönelimli sürücü çatısını sağlar. Amaç, mikro çekirdeğin modülerliğini monolitik çekirdeğin hızıyla buluşturmaktır. Apple son yıllarda yeni sürücüleri çekirdeğin dışına, kullanıcı alanındaki DriverKit çatısına taşımaktadır.

2.6.11 Exokernel

Yelpazenin en uç noktasında exokernel bulunur. Bu yaklaşım, MIT'de Dawson Engler ve Frans Kaashoek'un 1995'te önerdiği bir araştırma mimarisidir. Çekirdeğin adı Aegis, onunla birlikte çalışan kütüphane işletim sisteminin adı ExOS'tur. Exokernel ticari bir işletim sistemi değil, bir fikirdir.

Tanım 2.14

Exokernel, yalnız donanım kaynaklarını korumalı biçimde dağıtan ve hangi kaynağın kime ait olduğunu izleyen çekirdektir. Dosya ve sanal bellek gibi soyutlamaları çekirdek değil, uygulamalarla birlikte kullanıcı modunda çalışan **kütüphane işletim sistemleri** (LibOS) sağlar.



Şekil 2.35: Exokernel yapısı. Web sunucusu, veritabanı ve POSIX uygulaması kendi LibOS'larıyla çalışır (özel ağ ve önbellek, özel bellek ve disk, UNIX benzeri ExOS). Exokernel yalnız disk bloklarının, bellek sayfalarının, işlemci zaman dilimlerinin ve ağ kartının kime ait olduğunu izler.

Exokernel'i sanal makine yaklaşımıyla karşılaştırmak, fikri anlamayı kolaylaştırır. Sanal makine yöneticisi her sanal makineye bütün makinenin bir kopyasını verir. Sanal makine kendi diskinin 0. bloğundan başladığını sanır; yönetici de bu bloğu gerçek diskteki bir bloğa eşlemek zorunda kalır. Exokernel ise her uygulamaya gerçek kaynağın bir bölümünü verir ve yalnız sahipliği denetler. Uygulama hangi gerçek disk bloklarını aldığını bilir; arada bir yeniden eşleme katmanı gerekmez.

Yaklaşımın sloganı, **yönetimi korumadan ayırmaktır**. Çekirdek yalnız korur: bir uygulama başkasının bloğuna ya da sayfasına dokunamaz. Kaynakların nasıl yönetileceği ise uygulamaya kalır. Bir veritabanı kendi önbellekleme ve disk yerleşim stratejisini, bir web sunucusu kendi ağ tamponlarını kurabilir. Uygulamaya özgü bu yönetim yüksek başarım sağlar; bedeli, uygulama geliştirmenin karmaşıklaşmasıdır. Exokernel fikri bugün iki biçimde yaşar. Birincisi, tek bir uygulamayı kendi küçük işletim sistemiyle birlikte derleyen **unikernel**'lerdir. İkincisi, ağ ve depolama G/Ç'sinde çekirdeği atlayıp aygıta kullanıcı alanından erişen DPDK ve SPDK gibi kütüphanelerdir.

2.6.12 Yapıların karşılaştırılması

Yapı	Çekirdekte ne var?	Güçlü yön	Zayıf yön	Örnek
Basit	her şey, sınırsız	küçük, basit	koruma yok	MS-DOS
Monolitik	bütün hizmetler	hız	hata yalıtımı yok	UNIX, Linux
Katmanlı	hiyerarşik katmanlar	doğrulama kolay	ek yük, katman tanımı zor	THE, MULTICS
Mikro çekirdek	IPC, çizelgeleme, bellek	güvenilirlik	ileti ek yükü	MINIX 3, QNX, seL4
Modüler	çekirdek özü + modüller	esneklik, hız	modül = tam yetki	Linux, FreeBSD
Hibrit	mikro çekirdek + seçili hizmetler	denge	karmaşıklık	Windows, macOS
Exokernel	yalnız koruma	uygulamaya özgü hız	geliştirme zor	Aegis/ExOS

Tablo 2.8: İşletim sistemi yapılarının karşılaştırılması.

Tablo yapıları saf biçimleriyle verir. Gerçek sistemler ise nadiren saftır. Linux monolitik ve modülerdir; Windows ve macOS hibrittir; hemen hepsi bir ölçüde katmanlıdır. Bir sistemi tek bir kutuya yerleştirmek yerine hangi kararları neden verdiğini sormak daha öğreticidir.

2.7 Sanal makineler ve çok işlemcili sistemler

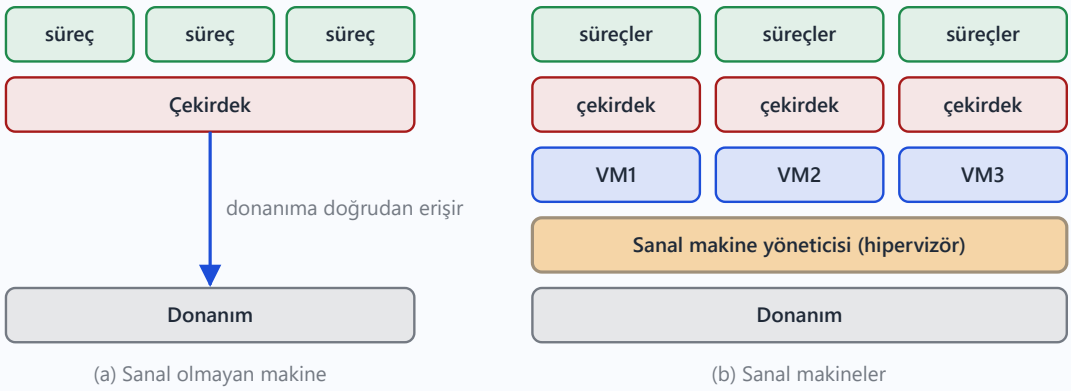
Çekirdek yapıları, bir işletim sisteminin kendi içini nasıl düzenlediğini anlatır. Bu kesimin iki konusu ise işletim sisteminin altındaki donanımla ilgilidir: donanımın yazılımla çoğaltılması ve birden çok işlemcinin aynı belleği paylaşması.

2.7.1 Sanal makineler

Tanım 2.15

Hipervizör (*hypervisor*) ya da sanal makine yöneticisi (*virtual machine monitor*, VMM), tek bir fiziksel donanım üzerinde her biri bağımsız bir bilgisayar gibi görünen **sanal makineler** (*virtual machine*, VM) oluşturan yazılım katmanıdır. Sanal makinenin içinde çalışan işletim sistemine **misafir işletim sistemi** denir.

Her misafir işletim sistemi kendi işlemcisi, belleği, diski ve G/Ç aygıtları olduğuna inanır. Bu yüzden farklı işletim sistemleri aynı donanımda aynı anda çalışabilir: aynı sunucuda bir sanal makinede Linux, ötekinde Windows çalışabilir. Şekil 2.36 sanal olmayan bir makineyi sanal makinelerle karşılaştırır.



Şekil 2.36: (a) Sanal olmayan makine: süreçler tek bir çekirdeğin üzerinde çalışır ve çekirdek donanıma doğrudan erişir. (b) Sanal makineler: hipervizör donanımın üzerinde üç sanal makine (VM1, VM2, VM3) oluşturur; her birinin kendi çekirdeği ve kendi süreçleri vardır.

Fikir yeni değildir; IBM'in 1972'de yayımladığı VM/370 sistemine dayanır. Bugün ise bulut bilişimin temelidir. Sanallaştırmanın başlıca yararları şunlardır:

- **Sunucu birleştirme.** Az kullanılan çok sayıda sunucu tek bir güçlü makinede toplanır.
- **Yalıtım.** Bir sanal makinedeki sorun ötekilere sıçramaz.
- **Anlık görüntü.** Sanal makinenin o anki durumu kaydedilir ve gerekirse bu duruma geri dönülür.
- **Canlı taşıma.** Çalışan bir sanal makine kapatılmadan başka bir fiziksel makineye taşınır.

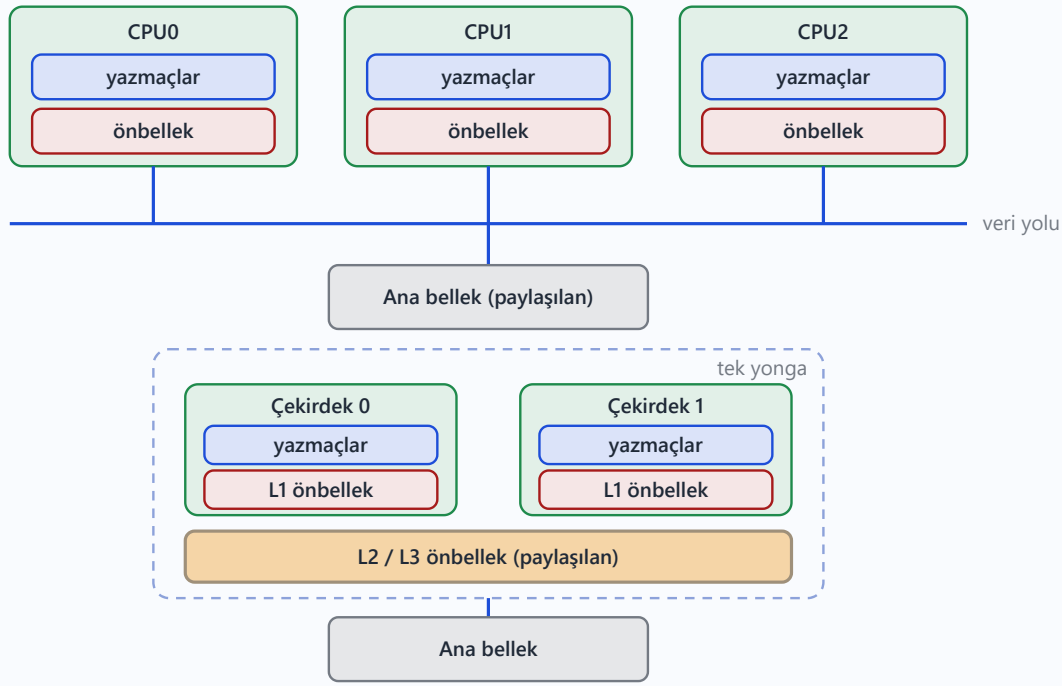
Hipervizörler çalıştıkları yere göre iki türe ayrılır. **Tip 1** hipervizör doğrudan donanımın üzerinde çalışır; altında başka bir işletim sistemi yoktur. Veri merkezlerinde kullanılan hipervizörler bu türdendir. **Tip 2** hipervizör ise sıradan bir işletim sisteminin üzerinde bir uygulama gibi çalışır; bir masaüstü bilgisayarda sanal makine çalıştıran programlar bu türdendir.

Sanal makineyle sık karıştırılan bir kavram **konteynerdir** (*container*). Konteyner, kendi çekirdeğini taşımaz. Aynı çekirdeği paylaşan bir grup süreci, çekirdeğin ad alanları ve kaynak sınırlama mekanizmalarıyla öteki süreçlerden yalıtır. Konteyner bu yüzden sanal makineden çok daha hafiftir ve saniyenin altında başlatılabilir. Ama aynı çekirdeği paylaştığı için içinde yalnız o çekirdekle uyumlu programlar çalışır ve yalıtımı sanal makineninki kadar güçlü değildir.

> **Sık yapılan hata**
>
> Exokernel ile hipervizörü aynı şey sanmak. İkisi de kaynakları birden çok tarafa paylaştırır, ama farklı biçimde. Hipervizör her sanal makineye makinenin tam bir kopyasını verir ve sanal kaynakları gerçek kaynaklara kendisi eşler. Exokernel ise gerçek kaynakların parçalarını dağıtır ve yalnız sahipliği denetler; soyutlamaları uygulamanın LibOS'u kurar.

2.7.2 Simetrik çok işlemcili ve çok çekirdekli mimari

Günümüzün hemen her bilgisayarında birden çok işlemci ya da işlemci çekirdeği vardır. Bu durum, işletim sisteminin yapısını da etkiler.



Şekil 2.37: Üstte simetrik çok işlemcili (SMP) mimari: her işlemcinin kendi yazmaçları ve önbelleği vardır, ana bellek bir veri yolu üzerinden paylaşılır. Altta çok çekirdekli yonga: iki çekirdek aynı yongadadır; her birinin kendi L1 önbelleği vardır, L2/L3 önbellek paylaşılır.

Simetrik çok işlemcili (*symmetric multiprocessor*, SMP) bir sistemde her işlemcinin kendi yazmaçları ve önbelleği vardır, ama ana bellek ortaktır. "Simetrik" sözcüğü, bütün işlemcilerin eşit haklı olduğunu anlatır: hepsi, işletim sistemi kodu da dahil her işi yapabilir. Bunun önemli bir sonucu vardır. İki işlemci aynı anda çekirdek kodunu çalıştırabildiği için çekirdeğin ortak veri yapıları kilitlerle korunmak zorundadır.

Örnek 2.13

Çekirdekte hazır süreçlerin bağlı bir listesi vardır ve listenin başını `bas` işaretçisi gösterir. Başlangıçta liste yalnız A görevinden oluşur. İki işlemci aynı anda listeye görev ekliyor: CPU0 X görevini, CPU1 Y görevini. Ekleme iki adımdır: (1) `g->sonraki = bas;` (2) `bas = g;`. Adımlar CPU0(1), CPU1(1), CPU0(2), CPU1(2) sırasıyla işlenirse ne olur?

Çözüm. CPU0(1): `X.sonraki = A`. CPU1(1): `Y.sonraki = A`. CPU0(2): `bas = X`; liste X → A olur. CPU1(2): `bas = Y`; liste Y → A olur. X görevi listeden kaybolmuştur: hiçbir işaretçiden ulaşılamaz ve hiçbir zaman çizelgelenmez. Sonuç, adımların hangi sırayla işlendiğine bağlıdır; buna **yarış durumu** (*race condition*) denir. Çözüm, ekleme işlemini bir kilitle korumaktır: kilidi al, (1) ve (2)'yi yap, kilidi bırak.

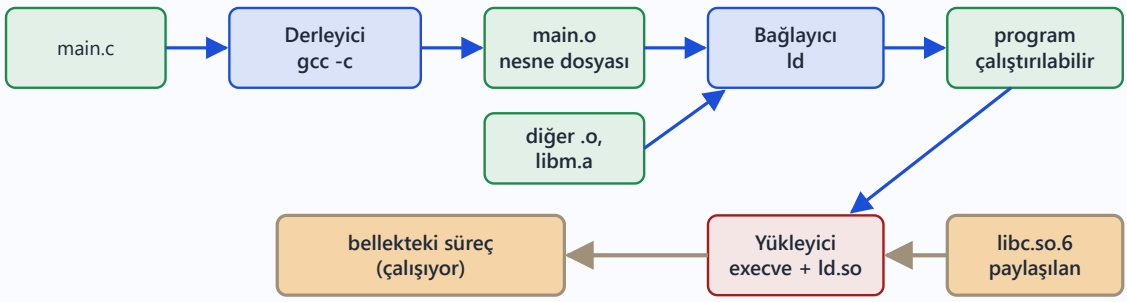
İlk çok işlemcili Linux sürümleri bu sorunu kaba bir yolla çözmüştü: bütün çekirdeği tek bir kilitle, "büyük çekirdek kilidiyle" (*big kernel lock*) koruyordu. Aynı anda yalnız bir işlemci çekirdek kodu çalıştırabiliyordu. Bu kilit yıllar içinde daha ince taneli kilitlerle değiştirildi ve 2011'de tamamen kaldırıldı.

Çok çekirdekli yongada (*multicore*) birden çok çekirdek aynı yonganın üzerindedir. Çekirdekler yakın önbellekleri paylaştığı için ayrı yongalardaki işlemcilere göre daha hızlı ve daha az enerjiyle haberleşir. 2026'da yaygın olan bir tasarım, aynı yongada iki tür çekirdek bulundurmaktır: hızlı ama enerji harcayan performans (P) çekirdekleri ve yavaş ama tutumlu verimlilik (E) çekirdekleri. Bu tasarım çizelgeleyiciye yeni bir görev yükler: hangi işin hangi tür çekirdekte çalışacağına karar vermek.

2.8 Bağlama, yükleme ve açılış

Son iki konu, bir programın ve bir işletim sisteminin nasıl çalışmaya başladığıdır. Programcının yazdığı kaynak kodun bir süreç hâline gelmesi birkaç aşamadan geçer. Aynı biçimde, güç düğmesine basıldığı andan giriş ekranının görünmesine kadar da bir dizi adım vardır.

2.8.1 Derleme, bağlama ve yükleme



Şekil 2.38: Derleme, bağlama ve yükleme hattı. `main.c` derleyiciyle (`gcc -c`) `main.o` nesne dosyasına çevrilir. Bağlayıcı (`ld`) bunu öteki nesne dosyaları ve statik kütüphanelerle (`libm.a`) birleştirip çalıştırılabilir program üretir. Yükleyici (`execve` ve `ld.so`) programı paylaşımlı `libc.so.6` ile birlikte belleğe alır ve süreç çalışmaya başlar.

1. **Derleme.** Derleyici kaynak kodu **yeniden konumlandırılabilir nesne dosyasına** çevirir. Bu dosyadaki adresler henüz kesin değildir, çünkü kodun bellekte nereye yerleşeceği bilinmez. Programın kullandığı ama başka bir dosyada tanımlı semboller, örneğin `printf`, de çözülmemiştir.
2. **Bağlama. Bağlayıcı** (*linker*) nesne dosyalarını ve statik kütüphaneleri birleştirir. Her sembolün hangi dosyada tanımlandığını bulur, adresleri düzeltir ve bir **çalıştırılabilir dosya** (*executable file*) üretir.
3. **Yükleme.** Kabuk `execve` çağırır. Çekirdekteki **yükleyici** (*loader*) dosyanın başlığını okur, kod ve veri kesimlerini sürecin adres alanına eşler. Program paylaşımlı kütüphane kullanıyorsa dinamik bağlayıcıyı (`ld-linux.so`) başlatır.
4. **Dinamik bağlama. Paylaşımlı kütüphaneler** (*shared library*), örneğin `libc.so.6`, çalışma anında bağlanır. Kütüphanenin kodu bellekte tek kopya olarak durur ve onu kullanan bütün süreçler bu kopyayı paylaşır.

Örnek 2.14

Bir sunucuda C kütüphanesini kullanan 50 farklı program aynı anda çalışıyor ve kütüphanenin kodu 3 MiB tutuyor. Kütüphane statik bağlanırsa ve dinamik bağlanırsa bellekte kütüphane kodu için ne kadar yer gerekir?

Çözüm. Statik bağlamada kütüphanenin kullanılan kodu her çalıştırılabilir dosyanın içine kopyalanır. Her süreç kendi kopyasını belleğe yükler: 50 × 3 MiB = 150 MiB gerekir. Dinamik bağlamada kütüphanenin kod sayfaları salt okunur olduğundan tek bir kopya bütün süreçlerin adres alanına eşlenir: yaklaşık 3 MiB yeterlidir. Her süreç ayrıca kütüphanenin yazılabilir veri-

si için küçük, özel bir alan kullanır. Dinamik bağlamanın bir yararı daha vardır: kütüphanedeki bir güvenlik açığı kapatıldığında yalnız `libc.so` güncellenir. Statik bağlamada ise 50 programın hepsinin yeniden bağlanması gerekir.

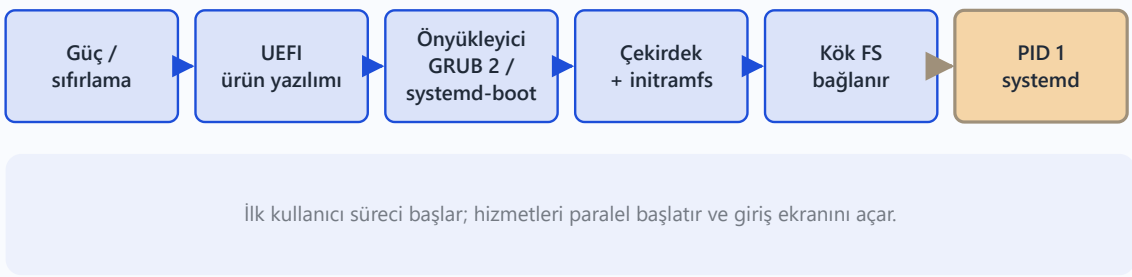
Çalıştırılabilir dosyanın biçimi işletim sistemine özgüdür: Linux ELF (*Executable and Linkable Format*), Windows PE (*Portable Executable*), macOS ise Mach-O biçimini kullanır. Bir programın bir işletim sisteminden ötekine doğrudan taşınamamasının bir nedeni budur. Aynı işlemci üzerinde bile bir sistemin çalıştırılabilir dosyası, öteki sistemin yükleyicisinin anlamadığı bir biçimdedir ve iki sistemin sistem çağrıları da farklıdır.

2.8.2 Açılış süreci

Tanım 2.16

Açılış ya da **önyükleme** (*booting*), bilgisayara güç verildiği andan çekirdeğin belleğe yüklenip ilk kullanıcı sürecini başlatmasına kadar geçen süreçtir.

Eski PC'lerde açılışı 1981'den beri kullanılan BIOS (16 bitlik gerçek kipte çalışan ürün yazılımı) yönetirdi. BIOS diskin ilk sektöründeki ana önyükleme kaydını (MBR) belleğe alıp çalıştırırdı. Günümüzün bilgisayarları ise UEFI kullanır. UEFI bir sektörü değil, bir dosyayı (`.efi` uzantılı bir program) yükler. Şekil 2.39 UEFI kullanan bir Linux bilgisayarın açılış adımlarını gösterir.



Şekil 2.39: İşletim sisteminin açılışı: güç verilmesi ve sıfırlama, UEFI ürün yazılımı, önyükleyici (GRUB 2 ya da systemd-boot), çekirdek ve initramfs, kök dosya sisteminin bağlanması ve PID 1 (`systemd`).

1. **Güç ve sıfırlama.** Güç verilince işlemci sıfırlanır ve donanımın belirlediği sabit bir adresten, anakart üzerindeki flash bellekte duran ürün yazılımını yürütmeye başlar. Bu anda bellekte henüz işletim sistemi yoktur.
2. **UEFI.** Ürün yazılımı önce açılış öz sınamasını (*power-on self-test*, POST) yapar: belleği dener, aygıtları bulur. Ardından kalıcı ayarlarında (NVRAM) kayıtlı açılış sırasına bakar ve sırasıyla NVMe disk, USB ya da ağ üzerinden açılış (PXE) seçeneklerini dener. GPT biçimli diskteki EFI sistem bölümün-

den (FAT32 dosya sistemi) önyükleyiciyi yükler. Güvenli Önyükleme (*Secure Boot*) açıksa önyükleyicinin dijital imzası doğrulanır; imzasız ya da değiştirilmiş bir önyükleyici çalıştırılmaz.

3. **Önyükleyici. Önyükleyici** (*bootloader*), GRUB 2 ya da systemd-boot, kullanıcıya bir menü sunar. Seçilen çekirdeği ve başlangıç RAM diskini (initramfs) belleğe koyar ve kontrolü çekirdeğe verir.
4. **Çekirdek.** Çekirdek bellek yönetimini, çizelgeleyiciyi ve sürücüleri başlatır. initramfs içindeki betikler, gerçek kök dosya sistemine ulaşmak için gereken modülleri (örneğin disk denetleyicisi ya da şifreli disk desteği) yükler.
5. **Kök dosya sistemi.** Gerçek kök dosya sistemi bağlanır ve artık gerekmeyen initramfs bellekten bırakılır.
6. **PID 1.** Çekirdek ilk kullanıcı sürecini, PID'si 1 olan `systemd`'yi başlatır. `systemd` hizmetleri, ağı ve giriş ekranını mümkün olduğunca paralel biçimde başlatır.

PID 1'in özel bir yeri vardır. Bundan sonra sistemde oluşan her süreç onun soyundandır; süreç ağacının kökü odur. Ebeveyni ölen süreçleri de PID 1 evlat edinir ve onların çıkış durumlarını toplar. Linux'ta `systemd-analyze` komutu, açılışın her aşamasının ne kadar sürdüğünü gösterir. Windows'ta aynı zincirin karşılığı `winload.efi` → `ntoskrnl.exe` → `smss.exe` → `winlogon.exe` biçimindedir.

2.9 Özet

- **İşletim sistemi yapısı**, bileşenlerin nerede çalışacağına (çekirdek ya da kullanıcı modu) ve nasıl haberleşeceğine (fonksiyon çağrısı ya da ileti) dair stratejidir. Her yapı performans, güvenilirlik ve esneklik arasında bir ödünleşim seçer.
- İşletim sistemi kullanıcıya ve programlara arayüz, program yürütme, G/Ç, dosya sistemi, iletişim ve hata algılama; sistemin bütününe kaynak ayırma, kaynak izleme, koruma ve güvenlik hizmetleri sunar. Hizmetlere sistem çağrılarıyla ulaşılır.
- **Süreç** çalışmakta olan programdır; bilgileri süreç tablosundaki PCB'de tutulur. Süreçler `fork()` ile bir ağaç oluşturur. **Adres alanı** metin, veri, BSS, heap ve yığın kesimlerinden oluşur.
- **Dosyalar** bir dizin ağacında durur; açık dosyalar küçük tam sayılarla (tanımlayıcılar) anılır ve yeni dosya en küçük boş numarayı alır. **Bağlama**, bir dosya sistemini ağaçtaki bir dizine iliştirir. **Boru**, iki süreci bağlayan tek yönlü bir bayt kanalıdır.
- **Çift modlu çalışma** ayrıcalıklı komutları çekirdek moduna ayırır; **zamanlayıcı** işlemcinin her zaman işletim sistemine dönmesini sağlar. Çekirdeğe yalnız sistem çağrısı, kesme ve istisnayla girilir.
- Bir **sistem çağrısında** libc sarmalayıcısı çağrı numarasını ve parametreleri yazmaçlara koyar, tuzak komutunu çalıştırır; çekirdek numarayı denetleyip sistem çağrısı tablosundan işleyiciyi bulur. Hata durumunda çağrı −1 döndürür ve `errno` ayarlanır.
- `printf` bir kütüphane fonksiyonudur ve tamponu boşaltırken `write` sistem çağrısını yapar. Tampon boyutu, yapılan sistem çağrısı sayısını belirler.
- Kabuk, `fork`, `execve` ve `waitpid` üçlüsüyle çalışır. `link` bir i-node'a yeni ad ekler, `unlink` bir adı siler; dosya ancak bağ sayısı ve açık sayacı 0 olunca yok olur.

- **Monolitik** çekirdek hızlıdır ama hata yalıtımı yoktur. **Katmanlı** yapı doğrulamayı kolaylaştırır ama ek yük getirir. **Mikro çekirdek** güvenilirdir ama ileti ek yükü taşır. **Modüler** yapı esnekliği hızla birleştirir, **hibrit** çekirdekler bu fikirleri harmanlar. **Exokernel** yalnız korur, yönetimi uygulamaya bırakır.
- **Hipervizör** donanımı sanallaştırıp birden çok işletim sistemini aynı makinede çalıştırır; **konteyner** ise aynı çekirdeği paylaşan süreçleri yalıtır. SMP sistemlerde çekirdek veri yapıları kilitlerle korunmalıdır.
- Kaynak kod derleyici, bağlayıcı ve yükleyiciden geçerek sürece dönüşür. Açılışta ürün yazılımı, önyükleyici ve çekirdek sırasıyla çalışır ve çekirdek ilk kullanıcı süreci olan PID 1'i başlatır.

2.10 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

işletim sistemi hizmetleri *operating system services* · süreç *process* · süreç kontrol bloğu *process control block (PCB)* · ebeveyn / çocuk süreç *parent / child process* · adres alanı *address space* · süreçler arası iletişim *interprocess communication (IPC)* · dosya tanımlayıcısı *file descriptor* · yol adı *path name* · blok / karakter aygıtı *block / character device* · bağlama *mount* · boru *pipe* · kilitlenme *deadlock* · kaynak tahsis çizgesi *resource allocation graph* · çift modlu çalışma *dual-mode operation* · çekirdek modu / kullanıcı modu *kernel mode / user mode* · tuzak *trap* · zamanlayıcı *timer* · sistem çağrısı *system call* · API *application programming interface* · kabuk *shell* · i-node *i-node* · katı bağ *hard link* · monolitik çekirdek *monolithic kernel* · katmanlı yapı *layered system* · mikro çekirdek *microkernel* · ileti geçirme *message passing* · politika / mekanizma *policy / mechanism* · istemci–sunucu modeli *client–server model* · yüklenebilir çekirdek modülü *loadable kernel module (LKM)* · hibrit çekirdek *hybrid kernel* · exokernel *exokernel* · sanal makine *virtual machine (VM)* · hipervizör *hypervisor* · misafir işletim sistemi *guest operating system* · konteyner *container* · simetrik çok işlemci *symmetric multiprocessor (SMP)* · yarış durumu *race condition* · çalıştırılabilir dosya *executable file* · paylaşımlı kütüphane *shared library* · önyükleme *booting* · önyükleyici *bootloader*

2.11 Kendini Dene

1. Yeni başlamış bir süreçte yalnız 0, 1, 2 açıktır. Sırasıyla `a = open("x", O_RDONLY)`, `b = open("y", O_RDONLY)`, `close(a)`, `c = open("z", O_RDONLY)`, `close(2)` ve `d = dup(b)` çalışı-

yor. `a`, `b`, `c` ve `d` kaçtır?

Cevap

Her çağrı en küçük boş numarayı verir. `a = 3`, `b = 4`. `close(a)` 3'ü boşaltır; `c = 3` olur. `close(2)` standart hatayı kapatır ve en küçük boş numara 2 olur; `d = dup(4) = 2`. Sonuç: 3, 4, 3, 2. Bu süreçteki hata iletileri artık `y` dosyasına yönelmiştir; ama dosya salt okunur açıldığı için oraya yazma girişimi hata verir.

2. Notta izlenen kopyalama kodu 4096 baytlık tamponla 9000 baytlık bir dosyayı kopyalıyor. Kaç `read` ve kaç `write` yapılır, her `read` ne döndürür? `open`, `creat`, iki `close` ve `printf` ile birlikte toplam kaç sistem çağrısı yapılır?

Cevap

`read` sırasıyla 4096, 4096, 808 ve 0 döndürür: 4 `read`. Son çağrı dosya sonunu bildirdiği için ardından `write` yapılmaz: 3 `write`. Toplam 1 + 1 + 4 + 3 + 2 + 1 = 12 sistem çağrısıdır. Program `9000 bayt` yazdırır.

3. `write(1, "Günaydın\n", n)` çağrısında metnin tamamının yazılması için `n` kaç olmalıdır? `n` karakter sayısı olarak verilirse ekranda ne görünür?

Cevap

Metinde 9 karakter vardır (G, ü, n, a, y, d, ı, n, \n). UTF-8'de ü ve ı ikişer bayt tuttuğu için bayt sayısı 11'dir; `n = 11` olmalıdır. `n = 9` verilirse son iki bayt, yani son “n” harfi ile satır sonu gönderilmez ve ekranda “Günaydı” görünür; imleç aynı satırda kalır.

4. x86-64 Linux'ta `mmap(adres, uzunluk, koruma, bayraklar, fd, ofset)` çağrısının altı parametresi ve çağrı numarası hangi yazmaçlara konur? Dördüncü parametre neden `rcx`'e konmaz?

Cevap

Çağrı numarası `rax`'a; `adres` `rdi`'ye, `uzunluk` `rsi`'ye, `koruma` `rdx`'e, `bayraklar` `r10`'a, `fd` `r8`'e, `ofset` `r9`'a konur. `syscall` komutu dönüş adresini `rcx`'e, bayrak yazmacını `r11`'e kaydettiği için `rcx`'teki bir parametre çekirdeğe ulaşmadan silinirdi; bu yüzden C kuralındaki `rcx` yerine `r10` kullanılır.

5. Bir zamanlayıcının sayacı 1 MHz'lik bir saatle her darbede 1 azalıyor. Çekirdek saniyede 100 tik istiyor. Sayaca hangi değer yüklenmelidir? Her tik kesmesinin işlenmesi 5 µs sürüyorsa işlemci za-

manının yüzde kaçı tiklere gider?

Cevap

1.000.000 / 100 = 10.000; sayaca 10.000 yüklenir ve iki tik arası 10 ms olur. Ek yük saniyede 100 × 5 µs = 500 µs'dir; bu, işlemci zamanının %0,05'idir.

6. Bir süreç `f` adlı dosyayı oluşturuyor (bağ sayısı 1). Sonra `link("f", "g")`, `link("g", "h")`, `unlink("f")`, `fd = open("h", O_RDONLY)`, `unlink("g")` ve `unlink("h")` çağrılıyor. Her adımdan sonra bağ sayısı kaçtır? Dosyanın blokları ne zaman serbest kalır?

Cevap

Bağ sayısı sırasıyla 2, 3, 2, 2, 1 ve 0 olur. `link("g", "h")`, `g`'nin gösterdiği i-node'a, yani aynı dosyaya üçüncü bir ad ekler. `open` bağ sayısını değiştirmez, açık sayacını 1 yapar. Son `unlink`'ten sonra dosyanın adı kalmaz, ama tanımlayıcı açık olduğu için bloklar serbest bırakılmaz. Bloklar ancak `close(fd)` çağrıldığında (ya da süreç sonlandığında) serbest kalır.

7. Basit kabuk iskeletinde çocuktaki `perror` ve `_exit(127)` satırları silinmiş olsun: çocuk `execve`'den sonra `exit` çağırmıyor. Kullanıcı `lss` diye var olmayan bir komut yazarsa ne olur?

Cevap

`execve` programı bulamaz ve −1 döndürür. Çocuk, kabuğun koduyla `else` bloğundan çıkar ve `while` döngüsünün başına döner: istem yazar ve komut beklemeye başlar. Ebeveyn kabuk ise hâlâ `waitpid`'de bu çocuğun bitmesini beklemektedir. Kullanıcı farkında olmadan ikinci bir kabukla konuşur; ancak o "kabuktan" çıkıldığında ebeveyn uyanır.

8. Mikro çekirdekli bir sistemde istemcinin istediği dosya bloğu dosya sistemi sunucusunun önbelleğinde bulunuyor. Okuma kaç iletiyle ve en az kaç mod geçişiyle tamamlanır? Monolitik çekirdekte kaç mod geçişi olur?

Cevap

Disk sürücüsüne gitmeye gerek yoktur. İstemcinin isteği ve sunucunun yanıtı olmak üzere 2 ileti yeterlidir. Her ileti bir kullanıcı → çekirdek ve bir çekirdek → kullanıcı geçişi gerektirdiği için en az 4 mod geçişi olur. Monolitik çekirdekte okuma tek bir sistem çağrısıdır: girişte ve dönüşte birer geçiş, toplam 2.

9. Aşağıdaki işlemlerin hangileri yalnız kütüphane koduyla tamamlanır, hangileri bir sistem çağrısı

gerektirir: `strlen`, `getpid`, `sqrt`, `fork`, `printf("a")` (satır sonu yok, çıktı terminalde)?

Cevap

`strlen` ve `sqrt` yalnız sürecin kendi verisiyle çalışır; kütüphane kodudur. `getpid` ve `fork` çekirdekteki süreç bilgisine dokunur; sistem çağrısıdır. `printf("a")` bir kütüphane fonksiyonudur ve terminalde satır tamponlu çalıştığı için satır sonu gelmeden çekirdeğe girmez; "a" tamponda bekler. Tampon daha sonra boşaltılırken (satır sonu, `fflush` ya da program sonu) `write` çağrılır.

10. Bir sürücünün hatalı bir işaretçiyle bellek bozduğunu varsayalım. Bu hata MINIX 3'te, Linux'ta (sürücü yüklenebilir modül olarak) ve exokernel tabanlı bir sistemde (sürücü uygulamanın LibOS'unda) nelere zarar verebilir?

Cevap

MINIX 3'te sürücü kullanıcı modunda ayrı bir süreçtir; yalnız kendi belleğini bozabilir, çöker ve reenkarnasyon sunucusu onu yeniden başlatır. Linux'ta modül çekirdek modunda ve çekirdeğin adres alanında çalışır; çekirdeğin herhangi bir yapısını bozabilir ve bütün sistemi çökertebilir. Exokernel'de LibOS uygulamayla aynı adres alanındadır; hata o uygulamayı bozar, ama exokernel sahipliği denetlediği için öteki uygulamaların kaynaklarına dokunamaz.

2.12 İleri okuma

- A. Silberschatz, P. B. Galvin ve G. Gagne, *Operating System Concepts* (10. baskı, 2018), Bölüm 2: işletim sistemi hizmetlerini, sistem çağrılarını, bağlayıcı ve yükleyiciyi, yapıları ve açılışı bu notun izlediği sırayla, bol şekille anlatır.
- A. S. Tanenbaum ve H. Bos, *Modern Operating Systems* (4. baskı, 2014), Kesim 1.5–1.7: süreç, adres alanı ve dosya kavramlarını, POSIX çağrılarını, `read`'in on bir adımını ve monolitik, katmanlı, mikro çekirdek, istemci–sunucu ve exokernel yapılarını ayrıntılı işler.
- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (sürüm 1.10), "Process API" ve "Limited Direct Execution" bölümleri: `fork` / `exec` / `wait` üçlüsünü ve tuzak ile zamanlayıcıya dayanan korumayı kısa ve açık örneklerle anlatır; çevrim içi ve ücretsizdir.
- D. R. Engler, M. F. Kaashoek ve J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management" (SOSP 1995): exokernel fikrini ortaya koyan özgün makaledir; yönetimi korumadan ayırma ilkesini Aegis ve ExOS üzerinde ölçümlerle savunur.

2.13 Kaynaklar

1. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, sürüm 1.10, pages.cs.wisc.edu/~remzi/OSTEP.

2. D. R. Engler, M. F. Kaashoek, J. O'Toole, “Exokernel: An Operating System Architecture for Application-Level Resource Management”, *SOSP*, 1995.
3. Linux man sayfaları: `syscalls(2)`, `syscall(2)`, `fork(2)`, `pipe(7)`, `mount(8)`.
4. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 (Bölüm 2 ve 18).
5. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı, Pearson, 2014 (Bölüm 1.5–1.7 ve 7).

KISIM II

Süreç Yönetimi

Bölüm 3–7

KISIM II · BÖLÜM 3

Süreçler

Bir bilgisayarda aynı anda bir tarayıcı, bir müzik çalar, bir derleyici ve onlarca arka plan hizmeti çalışır. İşletim sistemi bunların her birini ayrı bir **süreç** olarak görür. Her sürecin kendi belleği, kendi yazmaç değerleri ve kendi açık dosyaları vardır. Süreç, işletim sisteminin iş bölümünün temel birimidir: işlemci süreçlere dağıtılır, bellek süreçlere ayrılır, dosyalar süreçler adına açılır. Not, sürecin ne olduğu ve bellekte nasıl yerleştiği sorusuyla başlar. Ardından UNIX'in `fork`, `exec`, `wait` ve `exit` çağrılarıyla süreçlerin nasıl doğup öldüğünü kod izlemeleriyle anlatır. Süreç durumları, süreç kontrol bloğu, kuyruklar ve bağlam değiştirme, çekirdeğin süreçleri nasıl izlediğini gösterir. Son kesimler süreçlerin birbiriyle nasıl haberleştiğini ve mobil sistemlerin arka plandaki süreçleri neden sınırladığını ele alır.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum03` · **Laboratuvar:** `github.com/sercankulcu/operating-systems-labs/lab03`

ÖĞRENME ÇIKTILARI

Süreç modeli
Program ile süreç arasındaki farkı ve bir sürecin bellek yerleşimini açıklamak; bir C programındaki her değişkenin hangi bölgede durduğunu söylemek.

Yaşam döngüsü
Süreç durumlarını ve aralarındaki geçişleri tanımlamak; süreç kontrol bloğunun (PCB) içeriğini ve görevini açıklamak.

UNIX API
`fork()`, `exec()`, `wait()`, `exit()` ve `pipe()` kullanan küçük programları satır satır izlemek; çıktıyı ve oluşan süreç sayısını bulmak.

Çizelgeleme altyapısı
Hazır ve bekleme kuyruklarını, çoklu programlamanın CPU kullanımına etkisini ve bağlam değiştirmenin maliyetini açıklamak ve hesaplamak.

IPC
Paylaşımlı bellek, ileti geçirme, boru ve soketleri hız, eşgüdüm ve kullanım alanı bakımından karşılaştırmak.

Mobil çoklu görev
iOS ve Android'in arka plandaki süreçleri neden ve nasıl sınırladığını açıklamak.

3.1 Süreç kavramı

İşletim sistemi, donanımı doğrudan kullanmanın güçlüğünü birkaç soyutlamanın arkasına saklar. Dosya, diskteki blokların soyutlamasıdır; adres alanı, fiziksel belleğin soyutlamasıdır. **Süreç** ise işlemcinin soyutlamasıdır. Tek bir işlemci bile, süreç kavramı sayesinde, sanki her programa ayrı bir işlemci verilmiş gibi kullanılabilir. Bu kesim sürecin tanımını, bellekteki görünümünü ve aynı anda pek çok sürecin nasıl ilerlediğini anlatır.

3.1.1 Program ve süreç

Diskte duran `a.out` ya da `vim` dosyası bir **programdır**. Program, makine komutlarından ve başlangıç verilerinden oluşan bir **çalıştırılabilir dosyadır** (*executable file*). Kendi başına hiçbir şey yapmaz; disk üzerinde bekler. Bu yüzden program *pasif* bir varlıktır.

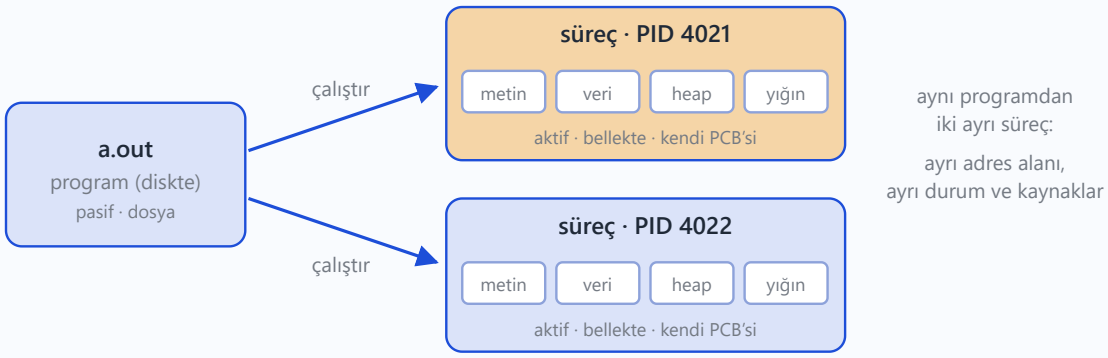
Program çalıştırıldığında işletim sistemi onun için bellek ayırır, komutlarını belleğe yükler ve işlemciye "buradan başla" der. O andan sonra ortada yürümekte olan bir iş vardır. Bu işin bir **program sayacı** (sıradaki komutun adresi), yazmaç değerleri, bir yığını, açık dosyaları ve bir kimlik numarası olur. Bu canlı varlığa **süreç** denir.

Tanım 3.1

Süreç (*process*), yürütülmekte olan bir program örneğidir. Bir sürecin üç bileşeni vardır: programın kodunu ve verisini taşıyan adres alanı, işlemcinin o anki durumu (program sayacı ve yazmaçlar) ve kaynaklar. Kaynaklar, işletim sisteminin sürece ayırdığı açık dosyalar, bellek bölgeleri ve kimlik bilgileridir. Her süreç sistem içinde tekil bir **süreç kimliği** (*process identifier*, PID) taşır.

Program ile süreç arasındaki ilişki, yemek tarifi ile mutfakta pişen yemek arasındaki ilişkiye benzer. Tarif kâğıt üzerinde durur ve değişmez. Pişirme ise bir süreçtir: hangi adımda olunduğu, tencerede ne olduğu, ocağın açık olup olmadığı her an değişir. Aynı tariften iki ayrı mutfakta aynı anda iki yemek pişebilir; ikisi birbirini etkilemez.

Aynı durum programlarda da geçerlidir. İki terminal penceresinde `vim` açıldığında diskteki tek bir dosyadan iki süreç oluşur. Her birinin ayrı adres alanı, ayrı açık dosyaları ve ayrı imleç konumu vardır. Birinde yazılan metin ötekinde görünmez. Programın salt okunur kodu ise iki süreç arasında paylaşılabilir; aynı makine komutlarının bellekte iki kopyasını tutmak gereksizdir. Şekil 3.1 bu durumu gösterir.



Şekil 3.1: Diskteki tek bir `a.out` programı iki kez çalıştırılınca iki ayrı süreç oluşur (PID 4021 ve 4022). Her sürecin kendi metin, veri, heap ve yığın bölgeleri ve kendi PCB'si vardır.

Bir sürecin içinde komutlar tek bir sıra hâlinde de yürüyebilir, birden çok sıra hâlinde de. Tek **iş parçacıklı** (*thread*) bir süreçte tek bir program sayacı vardır ve komutlar birbiri ardına yürütülür. Çok iş parçacıklı bir süreçte birden çok program sayacı bulunur. Bu iş parçacıkları aynı adres alanını paylaşır ve çok çekirdekli bir makinede gerçekten aynı anda ilerleyebilir. Bu notta aksi söylenmedikçe süreçler tek iş parçacıklıdır.

> **Sık yapılan hata**
>
> "Program" ile "süreç" sözcüklerini aynı anlamda kullanmak. Program bir dosyadır ve tek başına kaynak tüketmez. Süreç ise bellek, işlemci zamanı ve açık dosyalar tüketir. Bir programın hiç çalışmayan, bir kez çalışan ya da aynı anda yüz kez çalışan hâlleri olabilir; her çalışma ayrı bir süreçtir.

3.1.2 Bellekte süreç yerleşimi

Her süreç, belleği kendisine ait sanki tek parça bir dizi gibi görür. Bu dizinin adreslerine **adres alanı** (*address space*) denir. 0'dan başlayıp en yüksek adrese kadar uzanan bu alan birkaç bölgeye ayrılır. Her bölgenin ayrı bir görevi ve ayrı erişim hakları vardır. Şekil 3.2, alttan üste doğru bu bölgeleri gösterir.



Şekil 3.2: Bir sürecin bellek yerleşimi. Düşük adreslerde metin, veri ve BSS bölgeleri sabit boyutludur. Heap yukarı, yığın aşağı doğru büyür; aradaki boş alan ikisinin de büyümesine yer bırakır. En üstteki çekirdek alanı her sürece eşlenir ama kullanıcı modunda erişilemez.

Bölgeleri en düşük adresten başlayarak tek tek ele alalım:

- **Metin** (*text*) bölgesi programın derlenmiş makine kodunu taşır. Salt okunurdur: süreç kendi kodunu yanlışlıkla değiştiremez. Aynı programdan oluşan süreçler bu bölgeyi paylaşabilir.
- **Veri** (*data*) bölgesi ilk değeri verilmiş global ve `static` değişkenleri tutar. Bu değerler çalıştırılabilir dosyanın içinde saklanır ve yükleme sırasında belleğe kopyalanır.
- **BSS** bölgesi ilk değeri verilmemiş global ve `static` değişkenleri tutar. Adı eski bir assembly yönergesinden (*block started by symbol*) gelir. Bu değişkenlerin hepsi sıfırla başlar. Dosyada yalnız bölgenin boyutu yazılır; yükleyici o kadar sıfırlı bellek ayırır. Böylece 4 MB'lık sıfır dolu bir dizi çalıştırılabilir dosyayı 4 MB büyütmez.
- **Heap** (*heap*) çalışma sırasında `malloc()` (C++'ta ve Java'da `new`) ile istenen belleği karşılar. Program bellek istedikçe heap yukarı doğru büyür; `free()` ile bırakılan alan yeniden kullanılır.
- **Yığın** (*stack*) fonksiyon çağrılarını izler. Her çağrıda yığına bir **çerçeve** eklenir: parametreler, yerel değişkenler ve dönüş adresi bu çerçevede durur. Fonksiyon dönünce çerçeve kaldırılır. Yığın yüksek adreslerden aşağı doğru büyür.
- **Çekirdek alanı** adres alanının en üst kısmıdır. Çekirdeğin kodu ve verisi buraya eşlenir; böylece bir sistem çağrısı sırasında adres alanını değiştirmeye gerek kalmaz. Kullanıcı modunda çalışan kod bu bölgeye dokunamaz; dokunmaya çalışırsa donanım bir hata üretir.

Heap ile yığının karşılıklı büyümesi bilinçli bir tasarımdır. İki bölgenin ne kadar büyüyeceği önceden bilinmez: özyinelemeli bir program çok yığın, büyük veri işleyen bir program çok heap kullanır. Aralarındaki boş alanı ortak kullanmaları, belleğin ikisi arasında esnekçe paylaşılmasını sağlar.

Program sayacı bu resimde yer almaz, çünkü bellekte değil işlemcinin içinde bir yazmaçtır. Program sayacı metin bölgesindeki bir adresi gösterir: sürecin "şu anki konumu" odur. Yığının tepesini gösteren **yığın işaretçisi** de bir yazmaçtır. Süreç işlemciden ayrıldığında bu yazmaçların değeri kaybolmasın diye çekirdek onları bir yere kaydeder. Bu kaydın yeri 3.4.4'te ele alınan süreç kontrol bloğudur.

Gerçek sistemlerde bölgelerin adresleri her çalıştırmada aynı değildir. **Adres alanı düzeninin rastgeleleştirilmesi** (*address space layout randomization*, ASLR) yığının, heap'in ve paylaşılan kütüphanelerin başlangıç adreslerini her çalıştırmada rastgele kaydırır. Saldırgan, bellekte neyin nerede olduğunu kestiremediği için pek çok saldırı zorlaşır. Bölgelerin sırası ve büyüme yönleri ise değişmez.

3.1.3 Değişkenler nereye yerleşir?

Bölgeleri somutlaştırmanın en iyi yolu küçük bir C programındaki her adın nereye düştüğünü bulmaktır. Aşağıdaki program sunumdakiyle aynıdır:

```c
#include <stdlib.h>
int sayac = 5;    /* data */
int tablo[1000];  /* bss */

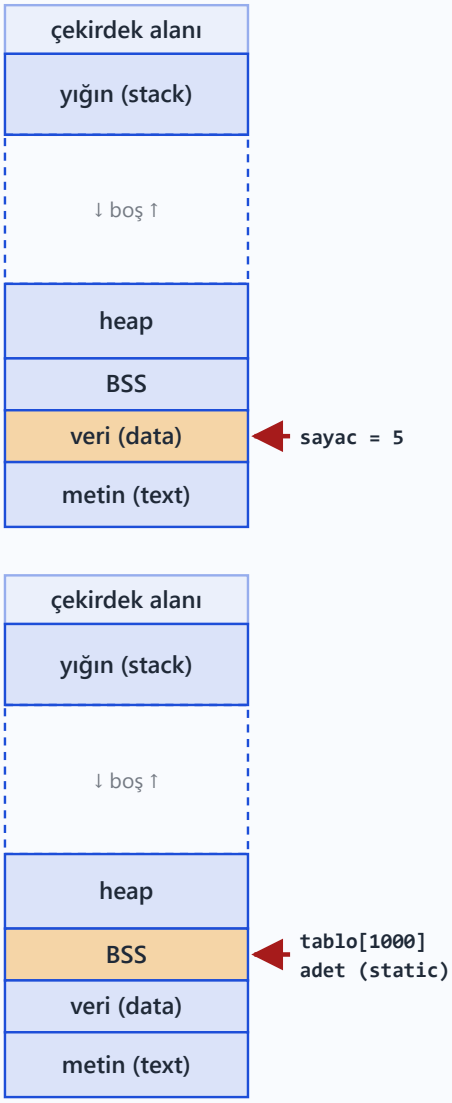
int main(int argc, char *argv[]) {
    int yerel = 3;
    static int adet;
    int *p = malloc(4 * sizeof *p);
    p[0] = yerel + sayac;
    free(p);
    return 0;
}
```

Programdaki her öğenin yerini sırayla bulalım. Şekil 3.3 metin dışındaki dört bölgeyi yan yana gösterir.

Örnek 3.1

1. **Metin.** `main` fonksiyonunun derlenmiş komutları metin bölgesindedir. `malloc` ve `free` 'nin kodu da C kütüphanesinin metin bölgesinde durur. Bu bölge salt okunurdur ve aynı programın süreçleri arasında paylaşılabilir.
2. **Veri.** `int sayac = 5;` global ve ilk değerlidir. 5 değeri çalıştırılabilir dosyada saklanır; yükleme sırasında veri bölgesine kopyalanır.
3. **BSS.** `int tablo[1000];` global ama ilk değersizdir; 4000 bayt sıfırla başlar ve dosyada yer kaplamaz. `main` içindeki `static int adet;` de BSS'dedir. Fonksiyonun içinde tanımlanmış olması onu yığına koymaz; `static` sözcüğü değişkenin bütün program boyunca yaşamasını sağlar.

4. **Yığın.** `main` çağrıldığında yığında bir çerçeve açılır. `argc`, `argv`, yerel değişken `yerel = 3`, işaretçi `p`'nin kendisi ve dönüş adresi bu çerçevededir. `main` dönünce çerçeve ortadan kalkar.
5. **Heap.** `malloc(4 * sizeof *p)` heap'ten 4 · 4 = 16 bayt ayırır (`int` 4 bayt varsayımıyla). `p` yığındadır ama gösterdiği 16 bayt heap'tedir. `p[0] = yerel + sayac` satırı heap'teki ilk hücreye 3 + 5 = 8 yazar. `free(p)` bu alanı geri verir; `p` değişkeni ise yığında kalmaya devam eder.





Şekil 3.3: Örnek programdaki adların yerleri. (a) Veri: `sayac = 5`. (b) BSS: `tablo[1000]` ve `static adet`. (c) Yığın: `main`'in çerçevesi; `argc`, `argv`, `yerel`, `p` ve dönüş adresi. (d) Heap: `malloc` ile ayrılan 16 bayt.

Linux'ta bu yerleşim komut satırından da görülebilir. `size a.out` komutu metin, veri ve BSS bölgelerinin boyutunu yazar. Çalışan bir sürecin bütün bölgeleri ise `cat /proc/PID/maps` ile listelenir. Bu listede `[heap]` ve `[stack]` adlı satırlar ile programın ve kütüphanelerin eşlendiği aralıklar görünür.

Sık yapılan hata

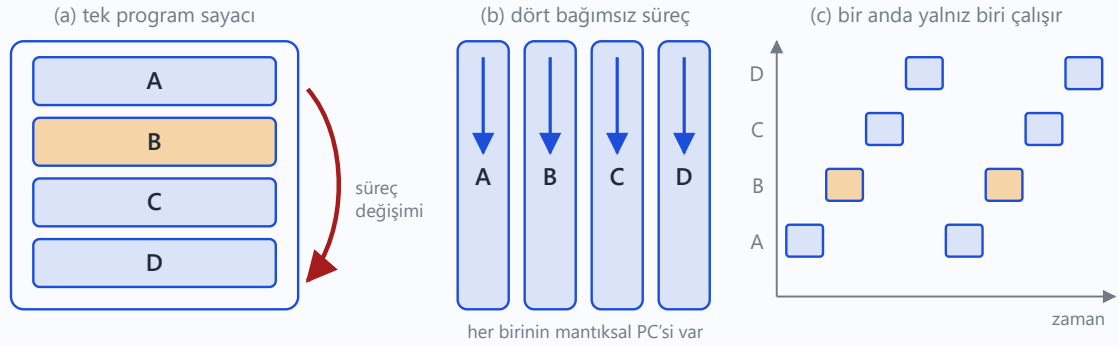
"İşaretçi heap'te durur" demek. `int *p = malloc(16);` satırında iki ayrı bellek parçası vardır. `p` değişkeni yerel olduğu için yığındadır ve 8 baytlık bir adres taşır. Bu adresin gösterdiği 16 bayt heap'tedir. Fonksiyon dönünce `p` kaybolur ama `free` çağrılmadıysa heap'teki alan ayrılmış kalır. Bellek sızıntısı tam olarak budur.

3.1.4 Sözde paralellik ve çoklu programlama

Tek çekirdekli bir işlemci herhangi bir anda yalnız bir komut dizisini yürütebilir. Buna karşın kullanıcı aynı anda müzik dinleyip belge yazabilir. Bu, işlemcinin süreçler arasında çok hızlı geçiş yapmasıyla mümkün olur. Her süreç birkaç milisaniye çalışır, sonra işlemci bir sonrakine geçer. Saniyede yüzlerce geçiş olduğu için kullanıcı bütün süreçlerin aynı anda çalıştığını sanır. Bu yanılsamaya **sözde paralellik** (*pseudoparallelism*) denir.

> **Tanım 3.2**
>
> **Çoklu programlama** (*multiprogramming*), bellekte aynı anda birden çok süreç tutup işlemciyi aralarında paylaştırmaktır. Amaç, bir süreç beklerken (örneğin diskten veri gelmesini beklerken) işlemciyi boş bırakmamak ve başka bir süreçle meşgul etmektir. İşlemciyi hangi sürecin alacağına **çizelgeleyici** (*scheduler*) karar verir.

Şekil 3.4 bu durumu üç açıdan gösterir. (a) bölümünde fiziksel gerçek vardır: tek bir program sayacı A, B, C ve D süreçleri arasında sırayla dolaşır. (b) bölümü kavramsal modeldir: her süreç kendi mantıksal program sayacı olan bağımsız bir iş gibi düşünülür. Süreç işlemciden ayrılırken fiziksel program sayacının değeri o sürecin mantıksal sayacına kaydedilir. (c) bölümü zaman eksenidir: uzun bir aralığa bakıldığında dört süreç de ilerlemiştir, ama herhangi bir anda yalnız biri çalışır.



Şekil 3.4: Çoklu programlama. (a) Tek program sayacı dört süreç arasında geçiş yapar. (b) Kavramsal olarak dört bağımsız süreç vardır; her birinin mantıksal program sayacı bulunur. (c) Zaman ekseninde her süreç ilerler, ama bir anda yalnız biri çalışır.

Geçişlerin sıklığını belirleyen büyüklüğe **zaman dilimi** (*time slice*, *quantum*) denir. Tipik değerler birkaç milisaniyedir. Güncel Linux çekirdeklerinin çizelgeleyicisi EEVDF'de (*Earliest Eligible Virtual Deadline First*) taban dilim yaklaşık 3 ms'dir. Eski CFS (*Completely Fair Scheduler*) çizelgeleyicisinde ise hazır süreçlerin hepsinin bir kez çalışması için hedeflenen süre, çekirdek sayısına göre 6–24 ms arasındaydı.

Sözde paralellik kullanıcı için rahattır ama programcı için bir uyarı taşır: bir sürecin ne zaman kesileceği önceden bilinemez. Kesilme, komutların ortasında, iki satır arasında, hatta bir değişkenin okunmasıyla yazılması arasında olabilir. Bir sürecin hızı da sabit değildir; aynı döngü bir çalıştırmada 2 ms, ötekinde 20 ms sürebilir. Bu yüzden doğru çalışan bir program, zamanlamayla ilgili hiçbir varsayım yapmaz. "Çocuk süreç bu sırada çoktan bitmiştir" ya da "bu iki satır kesintisiz çalışır" gibi düşünceler, ancak açık bir eşgüdüm mekanizmasıyla güvence altına alınabilir.

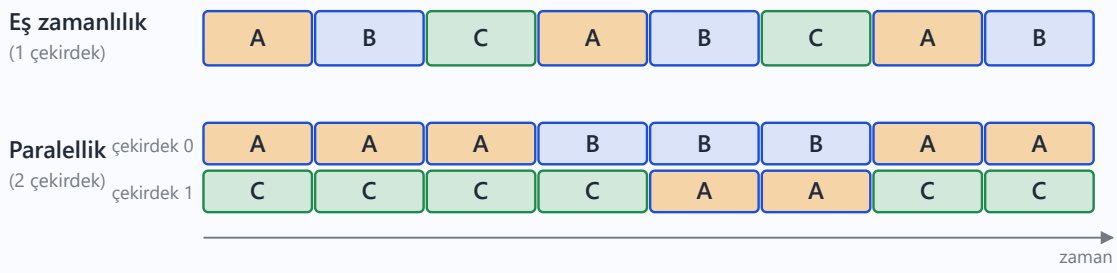
3.1.5 Eş zamanlılık ve paralellik

Gündelik dilde aynı anlamda kullanılan iki sözcük, işletim sistemlerinde farklı şeyleri anlatır.

Tanım 3.3

Eş zamanlılık (*concurrency*), birden çok işin aynı zaman aralığında ilerlemesidir. İşler tek bir çekirdekte iç içe geçerek (*interleaving*) yürüyebilir; herhangi bir anda yalnız biri çalışıyor olabilir. **Paralellik** (*parallelism*) ise birden çok işin aynı anda, fiziksel olarak birlikte yürütülmesidir ve birden çok çekirdek ya da işlemci gerektirir.

Şekil 3.5 farkı açıkça gösterir. Üst satırda tek çekirdek A, B ve C işlerini sırayla parça parça çalıştırır: üç iş de aynı aralıkta ilerler, ama hiçbir an ikisi birlikte çalışmaz. Alt kısımda iki çekirdek vardır; A ile C aynı anda yürür. Paralel bir sistem her zaman eş zamanlıdır, ama eş zamanlı bir sistem paralel olmak zorunda değildir.

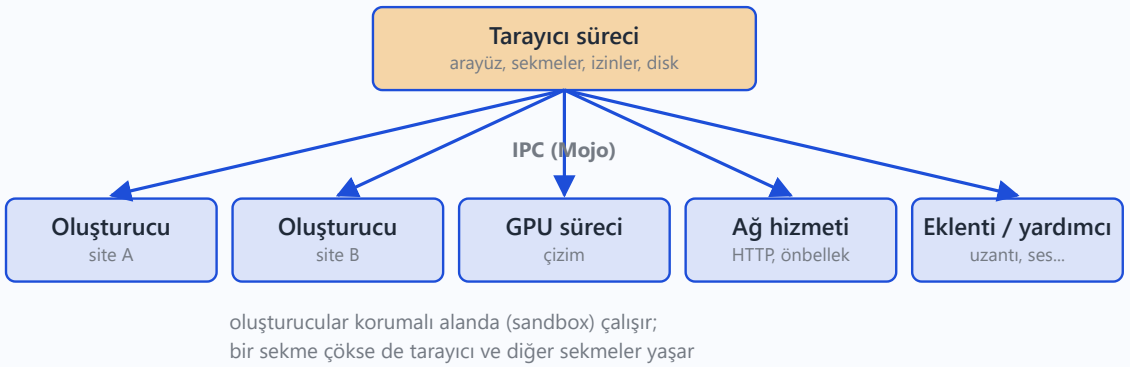


Şekil 3.5: Tek çekirdekte eş zamanlılık ve iki çekirdekte paralellik. Üstte işler iç içe geçer; altta iki çekirdek aynı anda farklı işleri yürütür.

Çok çekirdekli işlemciler artık her yerdedir; 2026 itibarıyla telefonlarda bile 6–10 çekirdek bulunur. Bu, geçişlere gerek kalmadığı anlamına gelmez. Sıradan bir dizüstü bilgisayarda bile yüzlerce süreç vardır ve çekirdek sayısı bunun çok altındadır. Bu yüzden çoklu programlama ve süreçler arası geçiş her sistemde, çekirdek sayısı ne olursa olsun yaşar.

3.1.6 Örnek: Chrome'un çok süreçli mimarisi

Süreç kavramının pratikteki değerini en iyi gösteren örneklerden biri Chrome tarayıcısıdır. Eski tarayıcılar tek bir süreç olarak çalışırdı. Bir sekmedeki hatalı bir sayfa ya da eklenti bütün tarayıcıyı çökertebiliyordu. Chrome bunun yerine işi birçok sürece böler (Şekil 3.6).



Şekil 3.6: Chrome'un çok süreçli mimarisi. Ayrıcalıklı tarayıcı süreci, korumalı alanda çalışan oluşturucu süreçleri ve GPU, ağ, yardımcı süreçleri Mojo adlı IPC sistemiyle haberleşir.

- **Tarayıcı süreci** tektir ve ayrıcalıklıdır. Arayüzü, sekmeleri, disk erişimini ve izinleri yönetir.
- **Oluşturucu** (*renderer*) süreçleri HTML, CSS ve JavaScript'i işler. **Site yalıtımı** (*site isolation*) sayesinde her site ayrı bir oluşturucu sürecinde çalışır. Bu süreçler **korumalı alanda** (*sandbox*) çalışır: diske ve ağa doğrudan erişemez, her isteği tarayıcı sürecine iletir.
- **GPU, ağ ve yardımcı süreçler** çizimi, ağ trafiğini ve uzantıları ayrı süreçlerde yürütür. Eski eklenti desteği ise kaldırılmıştır: NPAPI eklentileri 2015'te, Flash 2020 sonunda.

Bu tasarımın üç kazancı vardır. Birincisi *kararlılıktır*: bir sekme çökerse yalnız onun oluşturucu süreci ölür, tarayıcı ve öteki sekmeler çalışmaya devam eder. İkincisi *güvenliktir*: kötü niyetli bir sayfa kendi oluşturucu sürecini ele geçirse bile korumalı alandan çıkamaz ve öteki sitelerin verisini okuyamaz. Üçüncüsü *paralelliktir*: farklı sekmeler farklı çekirdeklerde paralel çalışabilir. Bedeli ise bellektir; her süreç kendi adres alanını ve veri yapılarını taşır. Bu yüzden çok sekmeli Chrome, tek süreçli bir tarayıcıdan daha çok bellek kullanır.

3.2 Süreç oluşturma

Sistemdeki her süreç bir başka süreçten doğar. Bu kesim süreçlerin hangi olaylarla ve hangi sistem çağrılarıyla oluştuğunu anlatır. Ağırlık UNIX'in iki temel çağrısındadır: sürecin kopyasını çıkaran `fork()` ve süreçteki programı değiştiren `exec()`. İkisinin birlikte kullanımı, kabuğun bir komutu nasıl çalıştırdığını ve çıktıyı nasıl dosyaya yönlendirdiğini de açıklar.

3.2.1 Süreç oluşturan olaylar ve sistem çağrıları

Yeni bir süreç dört tür olayla ortaya çıkar:

1. **Sistem açılışı.** Çekirdek yüklendikten sonra ilk kullanıcı sürecini başlatır. Linux'ta bu genellikle `systemd`'dir. O da arka planda sürekli çalışan hizmetleri başlatır. Kullanıcıyla doğrudan etkileşmeyen bu hizmetlere **daemon** denir; web sunucusu, yazdırma hizmeti ve günlük tutucu bu türdendir.
2. **Çalışan bir sürecin isteği.** Bir süreç, işini bölmek için yeni süreçler oluşturabilir. Örneğin bir derleme aracı her kaynak dosya için ayrı bir derleyici süreci başlatır.
3. **Kullanıcının bir komut ya da uygulama başlatması.** Kabukta `ls` yazmak ya da masaüstünde bir simgeye tıklamak yeni bir süreç oluşturur.
4. **Toplu bir işin başlatılması.** Büyük sunucularda ve hesaplama kümelerinde kuyruğa alınmış işler, kaynak boşaldıkça süreç olarak başlatılır.

Bu dört durum dışarıdan farklı görünür, ama teknik olarak aynı şey olur: var olan bir süreç, yeni süreç oluşturan bir sistem çağrısı yapar. Sistem açılışında bile ilk süreci çekirdek kendisi hazırlar; sonraki her süreç bir öncekinin çağrısıyla doğar.

Sistem	Çağrı
UNIX / Linux	`fork()` + `exec()`; ayrıca `posix_spawn()`, Linux'ta `clone()`
Windows	`CreateProcess()`: oluşturma ve program yüklemeyi tek çağrıda yapar

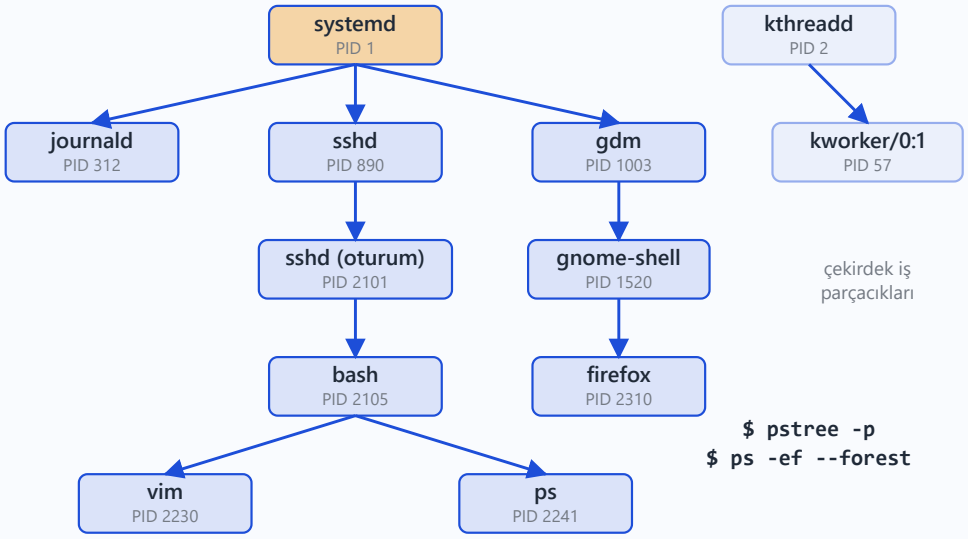
Tablo 3.1: Süreç oluşturan sistem çağrıları.

UNIX işi iki adıma böler. `fork()` çağıran sürecin bir kopyasını oluşturur; `exec()` ise bir sürecin içindeki programı başka bir programla değiştirir. Windows'un `CreateProcess()` çağrısı ikisini birden yapar: yeni süreci oluşturur ve içine verilen programı yükler. `posix_spawn()` UNIX'te de tek adımlı bir yol sunar. Linux'un `clone()` çağrısı ise yeni görevin ebeveyniyle neyi paylaşacağını (bellek, dosyalar, sinyal işleyicileri) tek tek seçtirir; iş parçacıkları da bu çağrıyla oluşturulur.

Süreci oluşturan sürece **ebeveyn** (*parent*), oluşan sürece **çocuk** (*child*) denir. Çocuk da kendi çocuklarını oluşturabilir. Her süreç tekil bir PID taşır ve ebeveyninin PID'ini bilir.

3.2.2 Süreç ağacı

Ebeveyn–çocuk ilişkileri bir ağaç oluşturur. Linux'ta ağacın kökü açılışta PID 1 ile başlayan `systemd`'dir (ya da başka bir `init` programı). PID 2 olan `kthreadd` ise çekirdek iş parçacıklarının ebeveynidir; `kworker` gibi çekirdek içi işçiler onun altında durur. Şekil 3.7 tipik bir masaüstü sistemin ağacından bir kesit gösterir. Şekildeki PID değerleri örnektir; gerçek değerler her açılışta değişir.



Şekil 3.7: Linux süreç ağacından bir kesit. `systemd` (PID 1) kullanıcı süreçlerinin, `kthreadd` (PID 2) çekirdek iş parçacıklarının köküdür. Ağaç `pstree -p` ya da `ps -ef --forest` ile görülebilir.

Ağaçta bir kabuk oturumunun izi kolayca okunur. Kullanıcı SSH ile bağlandığında `sshd` (PID 890) oturum için yeni bir `sshd` (PID 2101) oluşturur. O, kullanıcının kabuğunu (`bash` , PID 2105) başlatır. Kabukta yazılan her komut, örneğin `vim` ya da `ps` , kabuğun çocuğu olur. `ps` çıktısında her sürecin yanında ebeveyninin PID'i (PPID) yazar; bu sütun ağacın dallarını verir.

3.2.3 Ebeveyn ve çocuk ilişkisi

Yeni bir süreç oluşurken işletim sistemi dört soruya cevap verir:

- **Kaynak paylaşımı:** Çocuk, ebeveynin kaynaklarının tümünü ya da bir kısmını paylaşabilir ya da hiçbirini paylaşmayabilir. UNIX'te çocuk, ebeveynin açık dosya tanımlayıcılarını miras alır; aynı dosyalara aynı numaralarla erişir.
- **Yürütme:** Ebeveyn ve çocuk eş zamanlı çalışabilir. Ya da ebeveyn `wait()` ile çocuğun bitmesini bekleyebilir.
- **Adres alanı:** UNIX'te `fork()` sonrasında çocuk, ebeveynin adres alanının bir *kopyasını* alır. Aynı belleği paylaşmaz. Çocuk ardından `exec()` ile içine yeni bir program yükleyebilir.
- **Kimlikler:** `getpid()` sürecin kendi PID'ini, `getppid()` ebeveyninin PID'ini döndürür.

Adres alanının kopyalanması pahalı görünür: ebeveynin yüzlerce megabaytlık belleği varsa hepsi mi kopyalanır? Hayır. Çocuğun adres alanı mantıksal olarak ebeveyninkinden ayrıdır; fiziksel sayfalar ise ancak biri yazana kadar ortak tutulur. Bu tekniğin adı **yazınca kopyalamadır** ve bir sonraki alt kesimin konusudur.

3.2.4 fork(): bir çağrı, iki dönüş

`fork()` sıradan fonksiyonlara benzemez: bir kez çağrılır ama iki kez döner. Biri ebeveynde, öteki yeni oluşan çocukta. İki süreç de çağrının hemen arkasındaki satırdan devam eder. Hangisinin hangisi olduğunu ayırt etmenin tek yolu dönüş değeridir.

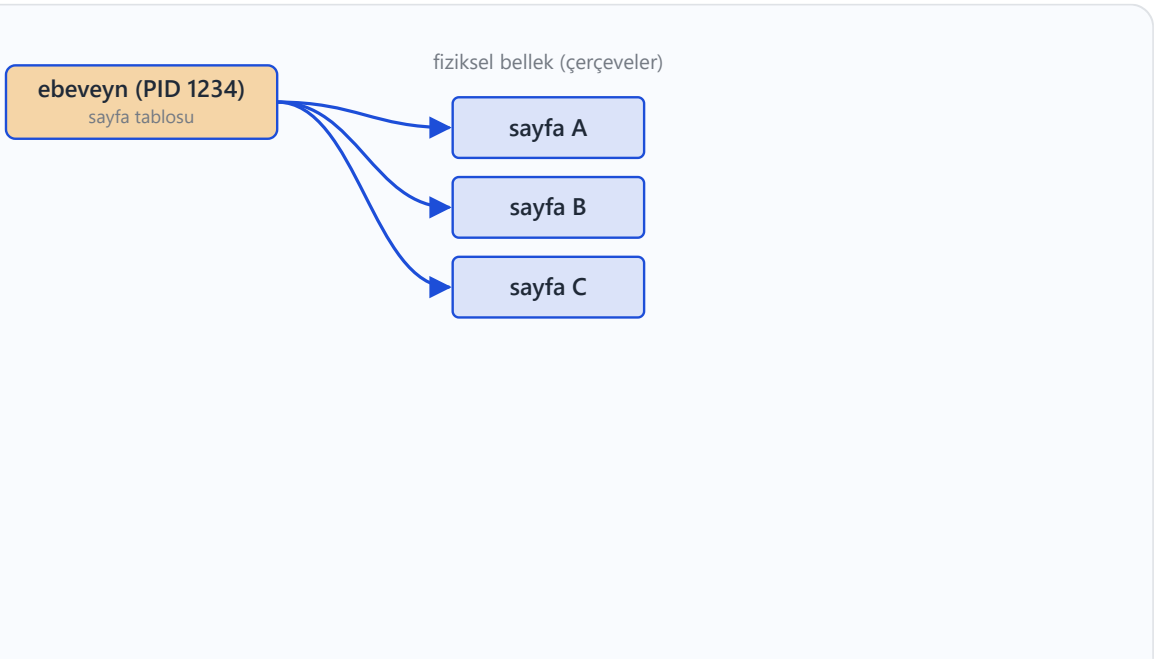
> **Tanım 3.4**
>
> `pid_t fork(void)` çağıran sürecin bir kopyası olan yeni bir süreç oluşturur. Başarı durumunda ebeveyne çocuğun PID'ini (pozitif bir sayı), çocuğa 0 döndürür. Başarısızlıkta çocuk oluşmaz ve yalnız ebeveyne −1 döner; neden `errno` değişkenine yazılır.

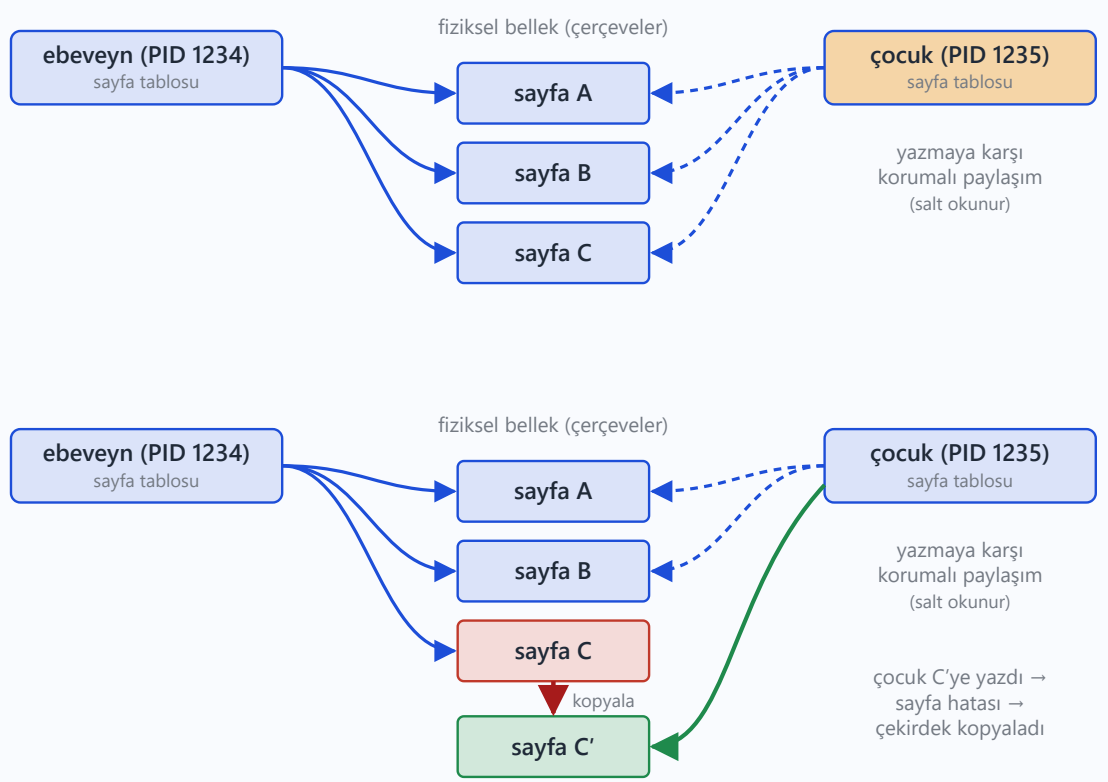
Bu üç durum, her `fork` sonrasında yazılan kalıbı belirler:

```
pid_t pid = fork();
if (pid < 0)        { perror("fork"); exit(1); }
else if (pid == 0)  { /* çocuk: fork 0 döndürdü */ }
else                { /* ebeveyn: pid = çocuğun PID'i */ }
```

Çocuk neden 0 alır da kendi PID'ini almaz? Çünkü çocuk kendi PID'ini her zaman `getpid()` ile öğrenebilir; ebeveynini de `getppid()` ile. Ebeveynin ise çocuğun PID'ini öğrenecek başka bir yolu yoktur. Bir ebeveynin birçok çocuğu olabilir ve onları beklemek ya da onlara sinyal göndermek için PID'lerini bilmesi gerekir. 0 değeri de hiçbir kullanıcı sürecinin PID'i olamayacağı için güvenli bir işarettir.

`fork` çekirdekte şunları yapar: yeni bir PCB ayırır ve ebeveyninkini içine kopyalar; yeni PID'i atar; açık dosya tablosunu kopyalar; sayfa tablosunu kopyalar. Sayfaların kendisini *kopyalamaz*. Şekil 3.8 bunun nasıl çalıştığını üç aşamada gösterir.





Şekil 3.8: `fork()` ve yazınca kopyalama. (a) Ebeveynin sayfa tablosu fiziksel A, B, C sayfalarını gösterir. (b) `fork` sonrası çocuğun sayfa tablosu aynı sayfaları gösterir; sayfalar iki tarafa da salt okunur eşlenir. (c) Çocuk C'ye yazınca sayfa hatası oluşur; çekirdek yalnız o sayfayı kopyalar (C′) ve çocuğun tablosunu yeni kopyaya yönlendirir.

Tanım 3.5

Yazınca kopyalama (*copy-on-write*, COW), iki sürecin aynı fiziksel sayfayı salt okunur olarak paylaşması ve sayfanın ancak süreçlerden biri ona yazmaya kalktığında kopyalanmasıdır. Yazma girişimi bir **sayfa hatası** (*page fault*) üretir; çekirdek sayfayı kopyalar, yazan sürecin sayfa tablosunu yeni kopyaya yönlendirir ve yazma iznini açar.

Yazınca kopyalama sayesinde `fork`'un maliyeti adres alanının boyutuyla değil, sayfa tablosunun boyutuyla orantılıdır. Daha önemlisi, çocuk hemen `exec` çağıracaksa (kabukta neredeyse her zaman böyledir) neredeyse hiçbir sayfa kopyalanmaz. Kopyalanacak sayfalar zaten `exec` sırasında atılacaktır. COW'un ayrıntıları sanal bellek konusunun parçasıdır; burada yalnız sonucu önemlidir: mantıksal olarak çocuk ebeveynin tam bir kopyasıdır ve iki taraf birbirinin değişkenlerini değiştiremez.

Sık yapılan hata

`fork` sonrasında ebeveyn ile çocuğun değişkenleri paylaştığını sanmak. Global bir değişken, heap'teki bir dizi ya da yığındaki bir yerel değişken, `fork` anında iki ayrı kopya hâline gelir. Çocuğun yaptığı atama ebeveynin kopyasını etkilemez. Süreçlerin veri paylaşması için açıkça

paylaşımlı bellek, boru ya da başka bir IPC aracı kurmak gerekir.

3.2.5 Kod izleme: fork() ve wait()

Aşağıdaki program, `fork`'tan sonra iki sürecin ayrı değişkenlerle çalıştığını ve `wait`'in çıktı sırasını nasıl sabitlediğini gösterir:

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    int x = 10;
    pid_t pid = fork();
    if (pid == 0) {
        x = x + 5;
        printf("çocuk: x=%d\n", x);
        return 0;
    }
    wait(NULL);
    printf("ebeveyn: x=%d\n", x);
    return 0;
}
```

Program 6. satırda `x = 10` ile başlar ve 7. satırda `fork` çağırır. Bundan sonra iki süreç vardır ve her birinde ayrı bir `x` bulunur. Çocukta `pid == 0` olduğu için `if` bloğuna girilir; ebeveyn bloğu atlayıp `wait(NULL)`'a gelir. `wait`, çağıran sürecin çocuklarından biri bitene kadar onu bekletir ve biten çocuğun PID'ini döndürür. `NULL` argümanı, çocuğun çıkış durumuyla ilgilenilmediğini söyler. Ebeveynin PID'i 1234, çocuğunki 1235 kabul edilsin. Tablo 3.2 bir çalıştırmanın adımlarını verir.

Adım	Satır	Ebeveyn (1234)	Çocuk (1235)	Çıktı
1	6	çalışıyor; `x = 10`	—	
2	7	`pid = 1235`, `x = 10`	oluştu; `pid = 0`, `x = 10`	
3	13	`wait(NULL)` → bekliyor	hazır	
4	8	bekliyor	çalışıyor; `pid == 0` doğru	
5	9	bekliyor; `x = 10`	`x = 15`	
6	10	bekliyor	`printf`	`çocuk: x=15`
7	11	uyandı (SIGCHLD)	`return 0` → zombi	
8	13	`wait` 1235 döndürür	biçildi (girdisi silindi)	
9	14	`printf`; `x` hâlâ 10	—	`ebeveyn: x=10`
10	15	sonlandı	—	

Tablo 3.2: `fork()` + `wait()` programının izlemesi (ebeveyn 1234, çocuk 1235).

İzlemenin üç noktası önemlidir. Birincisi, 5. adımda çocuk `x`'i 15 yapar ama ebeveynin `x`'i 10'da kalır. Aynı adı taşıyan iki ayrı değişken vardır; 9. adımda ebeveyn 10 yazar. İkincisi, tablodaki 3. ve 4. adımların sırası değişebilir. `fork`'tan sonra önce çocuk da çalışabilir; o zaman çocuk yazıp çıkar, ebeveyn `wait`'e geldiğinde çocuk zaten bitmiştir ve `wait` hemen döner. Her iki sırada da çıktı aynıdır:

```
çocuk: x=15
ebeveyn: x=10
```

Üçüncüsü, 7. adımda çocuk `main`'den döndüğünde hemen yok olmaz. C çalışma zamanı `main`'in dönüş değeriyle `exit(0)`'ı çağırır. Çekirdek çocuğun belleğini geri alır, ama çıkış durumunu ebeveyn okuyana kadar saklar. Bu kısa aralıkta çocuk **zombidir**. Çekirdek ebeveyne `SIGCHLD` sinyali gönderir ve onu uyandırır. 8. adımda `wait` çocuğun durumunu alır ve girdisini süreç tablosundan siler. Bu işleme **biçme** (*reaping*) denir. Zombi ve biçme 3.3.4'te ayrıntılı ele alınır.

3.2.6 wait() olmadan çıktı sırası belirsizdir

Önceki programdan `wait` çıkarılırsa ne olur? Aşağıdaki kısa parçaya bakalım:

```c
pid_t pid = fork();
if (pid == 0) printf("çocuk\n");
else          printf("ebeveyn\n");
```

`fork`'tan sonra iki süreç de hazırdır ve hangisinin önce çalışacağına çizelgeleyici karar verir. Programın bu kararı etkileyecek hiçbir yolu yoktur. Şekil 3.9 tek çekirdekte iki olası çizelgeyi gösterir: birinde önce ebeveyn, ötekinde önce çocuk yazar. Çok çekirdekli bir makinede iki süreç aynı anda bile çalışabilir. Linux'ta hangisinin önce çalıştığı sürüme ve ayara göre değişmiştir; programcı buna güvenemez.



Şekil 3.9: `fork` sonrası iki farklı çizelge ve iki farklı çıktı sırası. Üstte önce ebeveyn, altta önce çocuk çalışır; terminal kutuları ekranda görülen çıktıyı gösterir.

Programın doğruluğu bu sıraya bağlıysa (örneğin ebeveyn, çocuğun yazdığı bir dosyayı okuyacaksa) sıra açıkça kurulmalıdır. En basit yol `wait()` ya da `waitpid()`'dir. Daha karmaşık durumlar için borular, semaforlar ve başka eşgüdüm araçları vardır. Çıktının sırasının çalıştırmadan çalıştırmaya değişmesi, eş zamanlı programlardaki en temel sorunun ilk belirtisidir: sonuç, denetlenemeyen bir zamanlamaya bağlı olmamalıdır.

3.2.7 Kaç süreç oluşur? Tamponlu çıktı ve fork()

`fork`'un "her çağrıda süreç sayısını ikiye katlama" davranışı sınav sorularının vazgeçilmezidir. Kuralı bir kez doğru kurmak, en karışık görünen soruyu bile kolaylaştırır. Kural şudur: bir `fork` satırına o anda k süreç ulaşıyorsa, satırdan sonra $2k$ süreç vardır. Çünkü her biri bir çocuk doğurur ve çocuk da aynı satırdan devam eder.

```
fork();
fork();
printf("X\n");          /* kaç kez yazılır? */

for (int i = 0; i < 3; i++)
    fork();
printf("Y\n");          /* ya bu? */
```

Örnek 3.2

1. **İlk parça.** Başta tek süreç vardır. Birinci `fork`'tan sonra 2 süreç olur. İki süreç de ikinci `fork`'u çalıştırır; sonra 4 süreç olur. Dördü de `printf`'e ulaşır: 4 satır `X` yazılır. Yeni oluşan süreç sayısı 3'tür.
2. **Döngülü parça.** Döngü `i = 0`, 1, 2 için üç kez döner. Her turda süreç sayısı ikiye katlanır: $1 \to 2 \to 4 \to 8$. Çocuklar da döngü değişkeninin kopyasını taşır ve döngüye kaldıkları yerden devam eder. Sonuçta $2^3 = 8$ süreç `Y` yazar; 7'si yenidir.
3. **Genelleme.** Koşulsuz n tane `fork` art arda çalışırsa 2^n süreç oluşur ve $2^n - 1$'i yenidir. Koşullu `fork`'larda (örneğin `if (fork() && fork())`) bu kural doğrudan uygulanamaz; her süreç için dönüş değerini ayrı izlemek gerekir.

Süreç sayısını bulmak yetmez; bazen çıktının içeriği de şaşırtıcıdır. Şu üç satıra bakalım:

```
printf("A");
fork();
printf("B\n");
```

Akla gelen cevap "bir `A`, iki `B`"dir. Oysa program iki satır `AB` yazar. Bunun nedeni C kütüphanesinin **çıktı tamponudur**. `printf` karakterleri hemen çekirdeğe göndermez; önce sürecin kendi belleğindeki bir tampona yazar. Tampon ancak belirli durumlarda boşaltılır ve içeriği `write` sistem çağrısıyla çekirdeğe gider. Terminale yazarken tampon *satır tamponludur*: satır sonu (`\n`) gelince boşaltılır. Bir dosyaya ya da boruya yazarken ise *tam tamponludur*: tampon dolunca ya da süreç çıkarken boşaltılır.

İlk satırdaki `"A"` satır sonu içermez, bu yüzden `fork` anında hâlâ tampondadır. Tampon sürecin belleğinde durduğu için `fork` onu da kopyalar. Artık iki sürecin tamponunda da `A` vardır. Her süreç `"B\n"`'yi ekler ve satır sonuyla birlikte `AB` yazar. Çıktı terminale de gitse, dosyaya da yönlendirilse iki kez `AB` görülür.

İlk satır `printf("A\n")` olsaydı sonuç çıktının nereye gittiğine bağlı olurdu. Terminalde satır tamponu `A`'yı `fork`'tan önce boşaltır ve çıktı `A`, `B`, `B` olur. Çıktı `./prog > cikti.txt` ile dosyaya yönlendirilirse tam tampon `A\n`'yi tutmaya devam eder. Bu durumda iki süreç de çıkışta `A`, `B` yazar; dosyada dört satır bulunur: `A`, `B`, `A`, `B`.

Sık yapılan hata

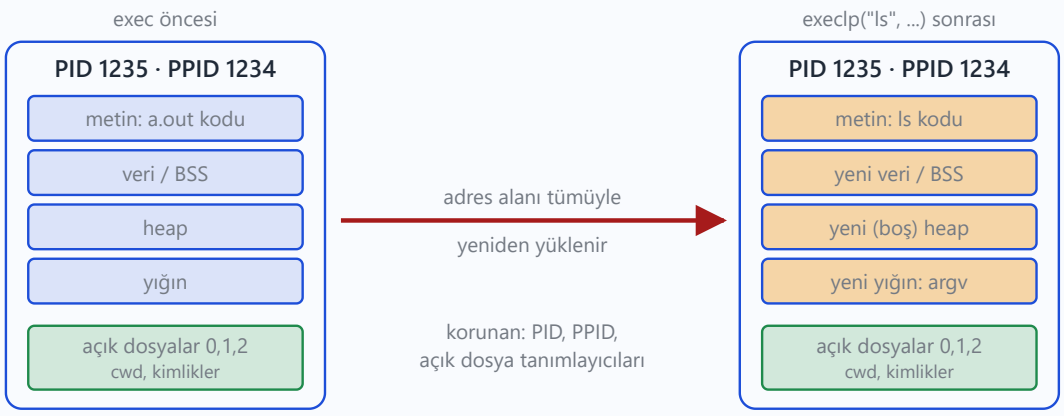
`printf` çağrısının döndüğü anda metnin ekrana ya da dosyaya yazıldığını varsaymak. `fork` ya da `exec` öncesinde tampon boşaltılmazsa yazılmamış veri ya çoğalır ya kaybolur. Güvenli yol, `fork`'tan önce `fflush(stdout)` çağırmaktır. Hata iletileri için `stderr` tercih edilir, çünkü `stderr` tamponsuzdur.

3.2.8 exec() ailesi

`fork` yalnız bir kopya üretir; kabuk her komutta kendini kopyalamakla kalsaydı hiçbir zaman `ls` çalıştıramazdı. Başka bir programı çalıştırmak için ikinci bir çağrı gerekir. `exec` ailesindeki fonksiyonlar, çağıran sürecin içindeki programı diskteki başka bir programla değiştirir.

Tanım 3.6

`exec` ailesi, çağıran sürecin bellek görüntüsünü (kod, veri, heap ve yığın) verilen çalıştırılabilir dosyadan yüklenen yeni bir görüntüyle değiştirir. Yeni süreç oluşmaz: PID, ebeveyn ilişkisi ve kapatılmamış açık dosya tanımlayıcıları korunur. Başarılı bir `exec` çağıran koda geri dönmez; dönüyorsa bir hata olmuştur ve dönüş değeri −1'dir.



Şekil 3.10: `exec` süreç görüntüsünü değiştirir. Soldaki `a.out` kodu ve verisi gider; yerine `ls`'nin kodu, yeni veri, boş heap ve `argv` taşıyan yeni bir yığın gelir. PID, PPID, açık dosyalar, çalışma dizini ve kimlikler korunur.

Ailenin altı üyesi aynı işi yapar; yalnız argümanları verme biçimleri farklıdır. Adlarındaki harfler bu farkı kodlar. **l** (*list*): argümanlar tek tek, `NULL` ile biten bir liste olarak verilir. **v** (*vector*): argümanlar bir dizi olarak verilir. **p** (*path*): program `PATH` ortam değişkenindeki dizinlerde aranır. **e** (*environment*): ortam değişkenleri açıkça verilir.

Çağrı	Argümanlar	PATH araması	Ortam
`execl` / `execv`	liste / dizi	hayır	devralınır
`execlp` / `execvp`	liste / dizi	evet	devralınır
`execle` / `execve`	liste / dizi	hayır	açıkça verilir

Tablo 3.3: `exec` ailesi.

Bu fonksiyonların hepsi C kütüphanesinde tanımlıdır ve sonunda tek bir sistem çağrısına varır: `execve()`. Örneğin `execlp("ls", "ls", "-l", NULL)` önce `PATH` içinde `ls`'yi arar ve `/usr/bin/ls`'yi bulur. Sonra argümanları `{"ls", "-l", NULL}` dizisine çevirip `execve`'yi çağırır. İlk argümanın programın adını tekrar etmesi bir alışkanlıktır: bu değer yeni programda `argv[0]` olur.

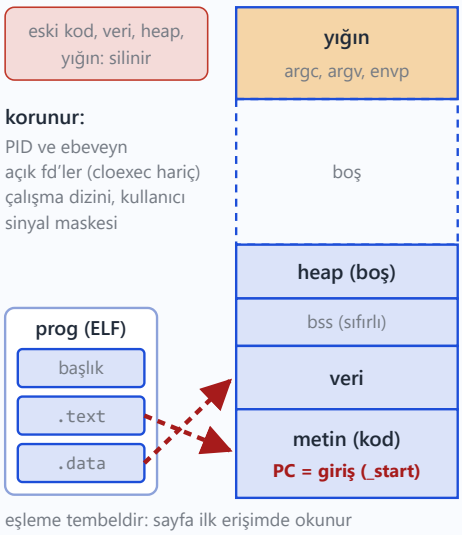
Sık yapılan hata

Argüman listesini `NULL` ile bitirmeyi unutmak ya da `argv[0]`'ı atlamak. `execlp("ls", "-l", NULL)` çağrısında `-l` programın adı (`argv[0]`) sayılır ve `ls` hiçbir seçenek almadan çalışır. `NULL` unutulursa kütüphane listenin sonunu bulamaz ve bellekteki rastgele değerleri argüman sanır.

3.2.9 exec çekirdekte ne yapar?

Dışarıdan bakınca `exec` tek bir adımdır: süreç bir programken öteki program olur. Çekirdeğin içinde ise sıralı altı iş yapılır.

1. **Dosyayı aç; izni ve biçimi denetle.** Dosya yoksa, çalıştırma izni yoksa ya da biçimi tanınmıyorsa `exec` −1 döndürür ve eski program kaldığı yerden sürer. Linux iki biçimi tanır: ELF başlığıyla başlayan ikili dosyalar ve `#!` satırıyla başlayan betikler. Betikte çekirdek, `#!` 'den sonra yazan yorumlayıcıyı (örneğin `/bin/sh`) çalıştırır ve betiğin adını ona argüman olarak verir.
2. **Eski adres alanını bırak.** Kod, veri, heap ve yığın eşlemeleri silinir; süreçteki öteki iş parçacıkları sonlandırılır. Bu noktadan sonra geri dönülecek bir program yoktur. Bu yüzden bütün denetimler birinci adımda yapılır.
3. **close-on-exec işaretli tanımlayıcıları kapat; sinyal işleyicilerini sıfırla.** `O_CLOEXEC` ile açılmış dosyalar kapatılır, ötekiler açık kalır. Yakalanan sinyaller varsayılan eyleme döner, çünkü işleyici fonksiyonun kodu artık bellekte değildir. Yok sayılan sinyaller yok sayılmaya devam eder.
4. **Kod ve veri bölümlerini dosyadan eşle.** Kopyalama yapılmaz. Bir sayfa ilk kez kullanıldığında sayfa hatası oluşur ve o sayfa diskten getirilir. Bu **tembel yükleme** sayesinde büyük bir programın hiç kullanılmayan kısımları belleğe hiç gelmez.
5. **BSS, heap ve yığın için sıfırlı bellek ayır.** Yığının tepesine `argv` ve `envp` dizgileri, onları gösteren işaretçi dizileri ve `argc` yerleştirilir.
6. **Program sayacını giriş noktasına ayarla; kullanıcı moduna dön.** Giriş noktası `main` değil, C çalışma zamanının `_start` yordamıdır. `_start` çalışma zamanını kurar ve `main(argc, argv, envp)` 'yi çağırır. `main` dönünce dönüş değeriyle `exit` 'i çağırır. Bütün bu işlem boyunca PID hiç değişmemiştir.



Şekil 3.11: `exec` sonrası yeni adres alanı. Metin ve veri dosyadan tembel olarak eşlenir; BSS, heap ve yığın sıfırlı bellekten gelir; yığının tepesinde `argc`, `argv`, `envp` durur. Eski kod, veri, heap ve yığın silinmiştir; PID, ebeveyn, açık tanımlayıcılar (close-on-exec olanlar hariç), çalışma dizini ve kullanıcı kimliği korunur.

Bu sıralamadan önemli bir sonuç çıkar: `exec`'ten sonra gelen satır, yalnız `exec` başarısız olursa çalışır. Bu yüzden `exec`'in arkasına `if (… == -1)` denetimi yazmaya gerek yoktur; arkasındaki satıra ulaşılmışsa zaten hata vardır. Doğru kalıp, `exec`'in hemen ardından hatayı yazdırıp süreci sonlandırmaktır.

3.2.10 Kod izleme: fork, exec ve waitpid

Aşağıdaki program, UNIX'te başka bir programı çalıştırmanın tam kalıbıdır: kopyala, çocukta programı değiştir, ebeveynde bekle.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
  pid_t pid = fork();
  if (pid == 0) {
    execlp("ls", "ls", "-l", NULL);
    perror("execlp");  /* hata */
    exit(1);
  }
  int st;
  waitpid(pid, &st, 0);
  printf("bitti: %d\n", WEXITSTATUS(st));
  return 0;
}
```

Burada `wait` yerine `waitpid` kullanılmıştır. `waitpid(pid, &st, 0)` belirli bir çocuğu bekler ve onun çıkış durumunu `st`'ye yazar. Üçüncü argüman seçenekleri taşır; 0 “bitene kadar bekle” demektir. `st` yalnız çıkış kodunu değil, sürecin nasıl bittiğini de kodlar. `WIFEXITED(st)` normal çıkış olup olmadığını, `WEXITSTATUS(st)` çıkış kodunu, `WIFSIGNALED(st)` ise bir sinyalle öldürülüp öldürülmediğini söyler.

Adım	Satır	Ebeveyn (1234)	Çocuk (1235)	Çıktı
1	7	`pid = 1235`	`pid = 0`; aynı `a.out` görüntüsü	
2	14	`waitpid(1235, …)` → bekliyor	hazır	
3	9	bekliyor	`execlp`: `PATH`'te `ls` bulunur; görüntü `/usr/bin/ls` olur, PID yine 1235	
4	—	bekliyor	`ls -l` çalışır; miras aldığı `stdout`'a yazar	(`ls -l` çıktısı)
5	14	uyandı; `waitpid` döner, `st` = kodlanmış 0	`exit(0)` → biçildi	

Adım	Satır	Ebeveyn (1234)	Çocuk (1235)	Çıktı
6	15	`WEXITSTATUS(st) = 0` yazılır	—	`bitti: 0`

Tablo 3.4: fork + exec + waitpid izlemesi (ebeveyn 1234, çocuk 1235).

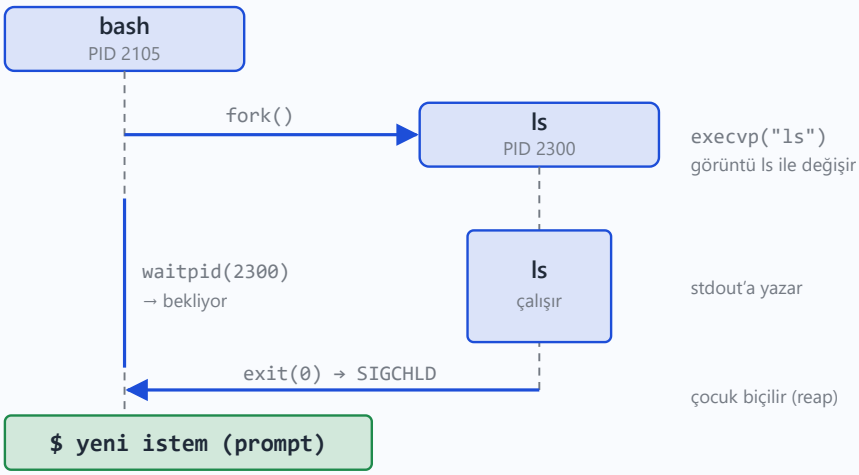
3. adım programın kalbidir. `execlp` başarılı olduğu için çocuktaki 10. ve 11. satırlar hiç çalışmaz: o satırları içeren kod artık bellekte değildir. Çocuk süreç bundan sonra `ls` programıdır. `ls`, kendisini kimin başlattığını bilmez; yalnız 1 numaralı dosya tanımlayıcısına yazar. O tanımlayıcı ebeveynden miras kalmıştır ve terminali gösterir. Bu yüzden liste ekrana çıkar.

Dosya adı yanlış yazılsaydı, örneğin `execlp("lss", …)`, `execlp` −1 döndürürdü. Çocuk 10. satırda `perror` ile `execlp: No such file or directory` iletisini yazar ve `exit(1)` ile çıkardı. Ebeveyn bu durumda `bitti: 1` yazardı. `exit(1)` satırı unutulsaydı ne olurdu? Çocuk `if` bloğundan çıkıp 13. satırdan devam eder ve ebeveynin kodunu çalıştırırdı. Sonuçta iki süreç `waitpid` çağırır ve iki kez "bitti" yazılırdı. Bu, `fork` kullanan programlarda en sık görülen hatalardan biridir.

3.2.11 Kabuk döngüsü: fork, exec, wait

Komut satırı kabuğu, önceki alt kesimdeki kalıbı sonsuz bir döngüde çalıştıran bir programdır. Döngünün her turu altı adımdan oluşur. Şekil 3.12 bir turu gösterir.

1. Kabuk (`bash`, PID 2105) istemi gösterir ve kullanıcının yazdığı `ls` komutunu okur.
2. `fork()` ile kendinin bir kopyasını (PID 2300) oluşturur. Ebeveyn `waitpid(2300, …)` ile beklemeye geçer.
3. Çocuk `execvp("ls", argv)` çağırır; görüntüsü `ls` ile değişir.
4. `ls` çalışır ve miras aldığı terminal `stdout`'una yazar.
5. `ls` çıkar. Çekirdek ebeveyni `SIGCHLD` ile bilgilendirir ve `waitpid` çocuğu biçer.
6. Kabuk yeni istemi gösterir ve döngü başa döner.



Şekil 3.12: Kabuk döngüsünün bir turu. `bash` çatallanır, çocuk `ls` 'ye dönüşüp terminale yazar, çıkınca `SIGCHLD` gelir ve kabuk çocuğu biçip yeni istemi gösterir.

Komutun sonuna `&` eklenirse (`sleep 60 &`) kabuk 2. adımda `waitpid` çağırmaz; hemen yeni istemi gösterir. Bu bir **arka plan işidir**. Arka plandaki çocuk bittiğinde de biçilmesi gerekir. Kabuk bunu `SIGCHLD` geldiğinde ya da bir sonraki istemden önce bekleyen çocukları toplayarak yapar.

3.2.12 Neden fork ve exec ayrı? Çıktı yönlendirme

Windows tek bir `CreateProcess` çağrısıyla yetinirken UNIX neden iki çağrı kullanır? Cevap, iki çağrı arasındaki kısa aralıktadır. `fork` 'tan sonra ve `exec` 'ten önce çocuk hâlâ kabuğun kodunu çalıştırır. Bu sırada kendi ortamını istediği gibi değiştirebilir: dosya tanımlayıcılarını yönlendirebilir, çalışma dizinini ya da önceliğini değiştirebilir, bazı sinyalleri yok sayabilir. Sonra `exec` ile yeni programa geçer. Yeni program bu değişiklikleri miras alır ama onlardan haberi yoktur.

Kabukta `wc -l notlar.txt > sayim.txt` yazıldığında olan budur. Aşağıdaki program aynı işi yapar:

```
int main(void) {
  pid_t pid = fork();
  if (pid == 0) {    /* çocuk */
    close(1);        /* stdout boşalır */
    int fd = open("sayim.txt",
             O_WRONLY | O_CREAT | O_TRUNC, 0644);
    /* en küçük boş numara: fd == 1 */
    execlp("wc", "wc", "-l", "notlar.txt", NULL);
    perror("execlp");
    exit(1);
  }
  waitpid(pid, NULL, 0);
  printf("bitti\n");
  return 0;
}
```

Programın anahtarı, `open`'ın her zaman **en küçük boş tanımlayıcı numarasını** vermesidir. Çocuk önce `close(1)` ile `stdout`'u kapatır; 1 numara boşalır. Hemen ardından açılan `sayim.txt` bu yüzden 1 numarayı alır. Şekil 3.13 çocuğun dosya tanımlayıcı tablosunun nasıl değiştiğini gösterir.



Şekil 3.13: Çocuğun dosya tanımlayıcı tablosu. (a) `fork` sonrası: 0, 1, 2 terminali gösterir. (b) `close(1)` : 1 numara boşalır. (c) `open` en küçük boş numarayı, 1'i verir; `stdout` artık dosyadır. (d) `exec` kodu değiştirir, tabloyu korur: `wc` farkında olmadan dosyaya yazar. Ebeveynin tablosu baştan sona değişmez.

Adım	Satır	Çocuğun fd 1'i	Olay	Ekran
1	2	terminal	`fork` : çocuk ebeveynin fd tablosunu kopyalar	
2	12	terminal	ebeveyn `waitpid` ile bekler	
3	4	(boş)	çocuk `close(1)` yapar	
4	5–6	`sayim.txt`	`open` 1 döndürür: `fd == 1`	
5	8	`sayim.txt`	`exec` : çocuk artık `wc -l` ; sonucu fd 1'e yazar	
6	13	—	`wc` biter; ebeveyn kendi fd 1'ine, terminale yazar	`bitti`

Tablo 3.5: Çıktı yönlendirme izlemesi (ebeveyn 1500, çocuk 1501).

Ekranda yalnız `bitti` görünür. `wc`'nin çıktısı (örneğin `42 notlar.txt`) `sayim.txt`'ye gitmiştir; `cat sayim.txt` ile görülebilir. `wc`'nin kaynak kodunda yönlendirmeyle ilgili tek satır yoktur. Program yalnız 1 numaraya yazar; o numaranın neyi gösterdiğini ise kabuk `exec`'ten önce ayarlamıştır.

Bu programda `close` ile `open` arasında başka bir dosya açılmadığı varsayılır. Gerçek kabuklar aynı işi daha güvenli biçimde `dup2(fd, 1)` ile yapar: dosyayı herhangi bir numarayla açar, sonra 1 numarayı ona yönlendirir. Windows'un `CreateProcess`'i aynı ayarları (standart giriş ve çıkış, çalışma dizi-

ni, öncelik) parametre olarak ister. Bu yüzden çağrının on parametresi vardır. UNIX'te ise her ayar ayrı, küçük bir sistem çağrısıdır ve `fork` ile `exec` arasında istenen sırayla yapılabilir.

3.3 Süreç sonlandırma

Her süreç bir gün biter. Bitiş, sürecin kendi isteğiyle olabileceği gibi bir hata ya da başka bir sürecin müdahalesiyle de olabilir. Hangi yolla biterse bitsin, çekirdek sürecin kaynaklarını geri alır. Yalnız küçük bir parça, çıkış durumu, ebeveyn onu okuyana kadar saklanır. Bu kesim bitişin türlerini, sinyalleri ve bu küçük parçadan doğan zombi ve yetim süreçleri anlatır.

3.3.1 Sonlandırma nedenleri, exit() ve abort()

Bir süreç dört nedenle sonlanır. İlk ikisi *gönüllüdür*: süreç bitmeye kendisi karar verir. Son ikisi *istemsizdir*: süreç dışarıdan durdurulur.

Neden	Tür	Örnek
Normal çıkış	gönüllü	`return 0` , `exit(0)` ; derleyicinin işi bitti
Hata sonrası çıkış	gönüllü	dosya bulunamadı → `exit(1)`
Ölümcül hata	istemsiz	geçersiz bellek erişimi (`SIGSEGV`), sıfıra bölme (`SIGFPE`)
Başka bir sürecin müdahalesi	istemsiz	`kill(pid, SIGTERM)` ; Windows'ta `TerminateProcess()`

Tablo 3.6: Süreç sonlandırma nedenleri.

Gönüllü çıkışın olağan yolu `exit(durum)` 'dur. `main` 'den `return` ile dönmek de aynı sonucu verir, çünkü C çalışma zamanı `main` 'in dönüş değeriyle `exit` 'i çağırır. `exit` bir C kütüphanesi fonksiyonudur ve üç iş yapar: `stdio` tamponlarını boşaltır, `atexit` ile kaydedilmiş işleyicileri çalıştırır, sonra `_exit` sistem çağrısıyla çekirdeğe geçer. `_exit` doğrudan çağrılırsa tamponlar boşaltılmaz. Bu fark, `fork` sonrasında çocuğun tamponları ikinci kez yazmasını önlemek için kullanılır.

Çıkış durumu bir tam sayıdır, ama ebeveyne yalnız alttaki 8 biti ulaşır. Bu yüzden anlamlı değerler 0–255 aralığındadır. Kural olarak 0 başarı, sıfırdan farklı her değer bir hata türü demektir. Kabukta son komutun çıkış durumu `$?` ile görülür ve `&&` , `||` gibi işleçler bu değere göre çalışır.

`abort()` ise anormal bir çıkıştır. Çağıran süreç kendine `SIGABRT` sinyali gönderir ve bu sinyalle ölür. Sistem ayarına göre bir **çekirdek dökümü** (*core dump*) dosyası oluşabilir. `assert` makrosu başarısız olunca `abort` 'u çağırır. Başka bir süreci sonlandırmak için ise `kill()` sistem çağrısı kullanılır.

Süreç hangi yolla biterse bitsin çekirdek aynı temizliği yapar: sürecin belleğini serbest bırakır, açık dosyalarını kapatır, öteki kaynaklarını geri alır. Geriye yalnız süreç tablosundaki girdi kalır: PID, çıkış durumu ve kullanılan CPU zamanı gibi birkaç bilgi. Bu bilgiler ebeveyn `wait` çağırana kadar saklanır.

Ebeveyn önce biterse çocuklara ne olacağı sistemden sisteme değişir. Bazı sistemler ebeveynle birlikte bütün çocukları da sonlandırır; buna **basamaklı sonlandırma** (*cascading termination*) denir. UNIX'te ise çocuklar yaşamaya devam eder ve yeni bir ebeveyne bağlanır.

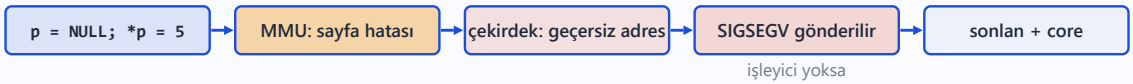
3.3.2 Ölümcül hatalar

Bir süreç geçersiz bir işlem yaptığında işlemci bunu bir **istisna** olarak algılar ve denetimi çekirdeğe verir. Çekirdek istisnanın türüne göre sürece bir **sinyal** gönderir. Sinyal, bir sürece bir olayın olduğunu bildiren küçük bir eş zamansız iletidir. Süreç o sinyal için bir işleyici kurmamışsa varsayılan eylem uygulanır; ölümcül hatalarda bu eylem sürecin sonlandırılmasıdır.

Hata	Sinyal
Geçersiz ya da izinsiz bellek erişimi	`SIGSEGV`
Tam sayı sıfıra bölme	`SIGFPE`
Geçersiz komut	`SIGILL`
Hizalama ya da veri yolu hatası	`SIGBUS`
Kapalı boruya yazma	`SIGPIPE`

Tablo 3.7: Ölümcül hatalar ve sinyaller.

Şekil 3.14 en bilinen örneği adım adım gösterir. Program `p = NULL` iken `*p = 5` yazmaya çalışır. 0 adresi hiçbir sayfaya eşlenmemiştir; bellek yönetim birimi (MMU) bir sayfa hatası üretir. Çekirdek adresin geçersiz olduğunu görür ve sürece `SIGSEGV` gönderir. Süreç bu sinyal için işleyici kurmadıysa sonlanır ve isteğe bağlı olarak bir çekirdek dökümü yazılır.



Şekil 3.14: Ölümcül bir hatanın akışı: `NULL` işaretçiye yazma → MMU sayfa hatası → çekirdek adresin geçersiz olduğunu görür → `SIGSEGV` → işleyici yoksa süreç sonlanır ve çekirdek dökümü alınır.

Çekirdek dökümü, sürecin öldüğü andaki belleğinin ve yazmaçlarının bir fotoğrafıdır. Hata ayıklayıcı bu dosyayla açılınca (`gdb a.out core`; systemd kullanan dağıtımlarda `coredumpctl`) programın hangi satırda, hangi çağrı zinciriyle öldüğü görülür. Bu inceleme türüne *çökme dökümü analizi* denir.

Sık yapılan hata

Bellek sızıntısını ölümcül hata saymak. `free` edilmeyen bellek ya da kapatılmayan dosyalar süreci kendiliğinden öldürmez; süreç çalışmayı sürdürür ve kaynakları süreç bitince zaten geri alınır. Uzun ömürlü bir sunucuda ise sızıntı belleği giderek tüketir. Linux'ta bellek tükenirse **OOM killer** (*out-of-memory killer*) en çok bellek tutan süreçlerden birini seçip `SIGKILL` ile sonlandırabilir. Süreç o zaman da sızıntı yüzünden değil, bu sinyal yüzünden ölür.

3.3.3 Sinyallerle sonlandırma

Bir süreç başka bir süreci `kill(pid, sinyal)` sistem çağrısıyla sonlandırabilir. Adı yanıltıcıdır: `kill` yalnız bir sinyal gönderir; sinyalin sonucu sürecin ölmesi olmak zorunda değildir. Komut satırındaki `kill`, `pkill` ve `killall` komutları bu çağrının kullanıcı arayüzleridir.

Sinyal	No	Anlamı
`SIGTERM`	15	kibarca bitir; süreç yakalayıp temizlik yapabilir (`kill`'in varsayılanı)
`SIGKILL`	9	zorla bitir; yakalanamaz, yok sayılamaz, temizlik yapılamaz
`SIGINT`	2	terminalde Ctrl+C
`SIGTSTP`	20	terminalde Ctrl+Z (duraklat); `fg` ya da `bg` sürdürür
`SIGSTOP` / `SIGCONT`	19 / 18	durdur (yakalanamaz) / sürdür

Tablo 3.8: Süreç denetiminde kullanılan sinyaller (numaralar x86 ve ARM Linux içindir).

Sinyal numaraları mimariden mimariye değişebilir; tablodaki değerler x86 ve ARM Linux içindir. Bu yüzden taşınabilir kod numaraları değil adları kullanır: `kill(pid, 15)` yerine `kill(pid, SIGTERM)`. Komut satırında tipik bir kullanım şöyledir:

```
$ ps -e | grep yaz
 1234 ?  00:00:03 yaz
$ kill 1234       # SIGTERM
$ kill -9 1234    # SIGKILL
$ pkill yaz       # ada göre
$ killall yaz
```

İyi bir alışkanlık, önce `SIGTERM` göndermek ve süreç yanıt vermezse `SIGKILL`'e geçmektir. `SIGTERM`'i yakalayan bir program geçici dosyalarını siler, yarım kalan yazmaları bitirir, bağlantılarını düzgünce kapatır. `SIGKILL` bu fırsatı tanımaz: süreç olduğu yerde durdurulur. Bir veritabanını `kill -9` ile öldürmek, bir sonraki açılışta kurtarma işlemi gerektirebilir.

Her süreç her sürece sinyal gönderemez. Gönderen sürecin hedefle aynı kullanıcıya ait olması ya da yetkili kullanıcı (root) olması gerekir. Aksi hâlde `kill` "işlem izni yok" hatasıyla döner. Bu kural, bir kullanıcının başka bir kullanıcının süreçlerini sonlandırmasını engeller.

3.3.4 Zombi ve yetim süreçler

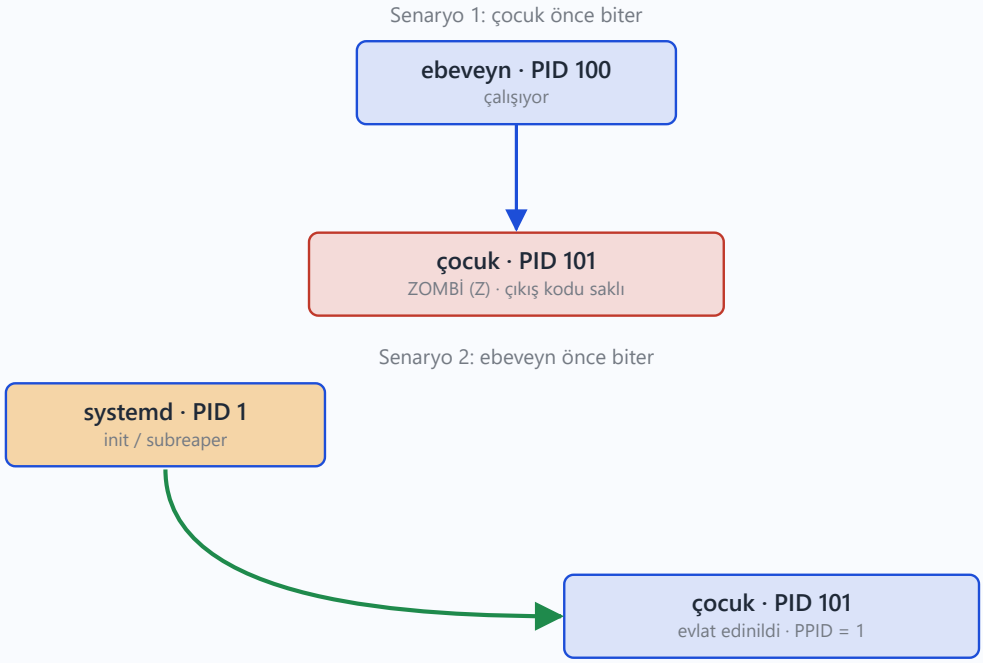
Ebeveyn ile çocuğun bitiş sırası iki özel durum doğurur. İkisi de `wait` çağrısıyla ilgilidir.

Tanım 3.7

Zombi süreç (*zombie process*), bitmiş ama ebeveyni henüz `wait` ile çıkış durumunu almamış süreçtir. Belleği ve kaynakları serbest bırakılmıştır; süreç tablosunda yalnız PID'i ve çıkış durumu kalır. **Yetim süreç** (*orphan process*) ise ebeveyni kendisinden önce sonlanmış ve hâlâ çalışan süreçtir.

Şekil 3.15 iki senaryoyu yan yana koyar. Birinci senaryoda ebeveyn (PID 100) ve çocuk (PID 101) çalışıyor. Çocuk `exit` ile biter, ama ebeveyn henüz `wait` çağırmamıştır. Çekirdek çocuğun çıkış durumunu atamaz, çünkü ebeveyn onu ileride isteyebilir. Çocuk zombi olur: `ps` çıktısında durumu `Z`, adının yanında `<defunct>` yazar. Ebeveyn `wait` çağırdığında durumu alır ve çekirdek girdiyi siler. Bu adıma **biçme** denir; ardından zombi kalmaz.

İkinci senaryoda ebeveyn çocuktan önce biter. Çocuk (PID 101) hâlâ çalışıyor ve artık yetimdir. Çekirdek yetimi sahipsiz bırakmaz: onu `init` ya da `systemd`'ye (PID 1) bağlar. Sistemde bir **alt-biçici** (*subreaper*) süreç tanımlanmışsa yetim ona bağlanır; konteyner yöneticileri ve bazı oturum yöneticileri bu özelliği kullanır. Çocuk artık `getppid()` çağırırsa 1 alır (ya da alt-biçicinin PID'ini). Yetim bitince yeni ebeveyni onu `wait` ile biçer. `init` bu işi düzenli yaptığı için yetimlerden zombi birikmez.



Şekil 3.15: Zombi ve yetim. (a) Senaryo 1: çocuk (101) bitti ama ebeveyn (100) henüz `wait` çağırmadı; çocuk zombidir ve çıkış kodu saklıdır. (b) Senaryo 2: ebeveyn önce bitti; yetim kalan çocuk `systemd`'ye (PID 1) bağlandı, PPID'i artık 1'dir.

Zombiler bellek ya da işlemci harcamaz, ama bir sınırlı kaynağı tüketir: PID'i. Linux'ta PID sayısı sınırlıdır (varsayılan üst sınır sisteme göre 32.768 ya da daha büyük bir değerdir). `wait` çağırmayan uzun ömürlü bir sunucu her istekte bir çocuk oluşturursa binlerce zombi biriktirir. Sonunda yeni süreç oluşturulamaz ve `fork` başarısız olur. Zombiyi `kill` ile öldürmek de işe yaramaz, çünkü zaten ölüdür. Çözüm ya ebeveynin `wait` çağırmasıdır ya da ebeveynin sonlandırılmasıdır: o zaman zombiler `init`'e geçer ve biçilir.

Çocuk oluşturan bir sunucu, çocukları bitirdikçe biçmek için genellikle `SIGCHLD` sinyalini yakalar. Aşağıdaki iskelet bu kalıbı gösterir:

```
#include <signal.h>
#include <sys/wait.h>

void bic(int sig) {
    (void)sig;
    while (waitpid(-1, NULL, WNOHANG) > 0)
        ;                       /* biten bütün çocukları topla */
}

int main(void) {
    signal(SIGCHLD, bic);       /* çocuk bitince bic() çalışsın */
    /* ... sunucu döngüsü: her istek için fork() ... */
    return 0;
}
```

`waitpid`'in ilk argümanı −1 “herhangi bir çocuk” demektir. `WNOHANG` seçeneği verilince ise `waitpid` bitmiş çocuk yoksa beklemeden 0 döndürür. Döngü, aynı anda biten birden çok çocuğun hepsini toplar. Bu önemlidir, çünkü aynı türden birden çok sinyal art arda gelirse tek bir sinyal olarak teslim edilebilir.

> **Sık yapılan hata**
>
> Zombi ile yetimi karıştırmak ya da ikisinin aynı ölçüde zararlı olduğunu sanmak. Zombi *bitmiş* ama biçilmemiş bir süreçtir; yetim ise *çalışan* ama ebeveyni ölmüş bir süreçtir. Yetimler zararsızdır, çünkü `init` onları evlat edinir ve bitince biçer. Biriken zombiler ise PID tükenmesine yol açabilir; sorunun kaynağı `wait` çağırmayan ebeveyndir.

3.4 Süreç durumları ve PCB

Bir süreç ömrü boyunca sürekli çalışmaz. Çoğu zaman işlemciyi bekler ya da bir olayın olmasını bekler. Çekirdek her sürecin o an hangi durumda olduğunu bilmek ve ona dair her şeyi bir yerde tutmak zorundadır. Bu kesim süreç durumlarını ve bu bilgilerin tutulduğu veri yapısını, süreç kontrol bloğunu anlatır.

3.4.1 Beş durumlu model

Tanım 3.8

Süreç durumu (*process state*), sürecin o anki etkinliğini anlatır. Beş durumlu modelde durumlar şunlardır:

- **Yeni** (*new*): süreç oluşturuluyor.
- **Hazır** (*ready*): süreç çalışmaya hazır, işlemciyi bekliyor.
- **Çalışıyor** (*running*): komutları işlemcide yürütülüyor.
- **Bekliyor** (*waiting*, *blocked*): bir olayı (G/Ç'nin bitmesi, bir sinyal, bir çocuğun sonlanması) bekliyor.
- **Sonlandı** (*terminated*): yürütmesi bitti.



Şekil 3.16: Beş durumlu süreç modeli ve altı geçiş: kabul, çizelgeleyici seçti, kesme ya da zaman dilimi doldu, G/Ç ya da olay bekle, G/Ç ya da olay tamamlandı, çıkış.

Şekil 3.16'daki oklar, durumlar arasındaki geçişlerdir. Her geçişin bir nedeni vardır:

- **Yeni → Hazır (kabul):** Çekirdek süreç için gereken yapıları kurmuş ve süreci hazır kuyruğuna koymuştur.
- **Hazır → Çalışıyor (çizelgeleyici seçti):** Çizelgeleyici hazır süreçlerden bunu seçer ve bağlamını işlemciye yükler. Bu adıma **görevlendirme** (*dispatch*) denir.
- **Çalışıyor → Hazır (kesme):** Zaman dilimi dolmuş ya da daha öncelikli bir süreç hazır olmuştur. Süreç hiçbir şey beklemez; yalnız sırası geçmiştir.
- **Çalışıyor → Bekliyor (olay bekle):** Süreç diskten okuma, klavyeden giriş ya da bir çocuğun bitmesi gibi bir olayı beklemek zorundadır. İşlemciyi kendi bırakır.

- **Bekliyor → Hazır (olay tamamlandı):** Beklenen olay olmuştur. Süreç doğrudan çalışmaz, hazır kuyruğuna döner.
- **Çalışıyor → Sonlandı (çıkış):** Süreç `exit` çağırmış ya da öldürülmüştür.

Tek çekirdekte herhangi bir anda yalnız bir süreç *çalışıyor* durumunda olabilir. *n* çekirdekte en çok *n* süreç çalışabilir. Sistemdeki yüzlerce sürecin çoğu ise hazır ya da bekliyor durumundadır. Gerçekte süreçlerin büyük kısmı bekliyor durumundadır: bir editör tuş basılmasını, bir sunucu ağ isteğini bekler.

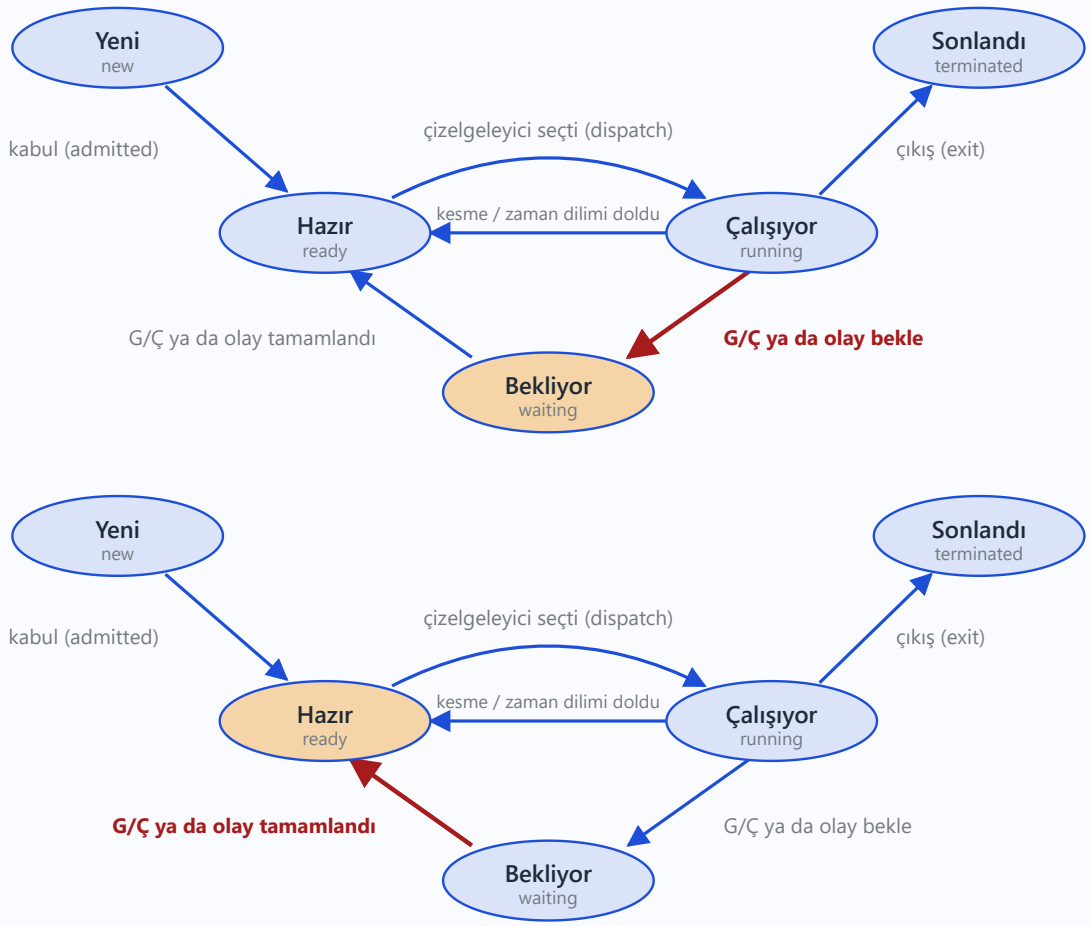
3.4.2 Durum geçişleri adım adım

Modeli somutlaştırmak için bir sürecin ömrünü baştan sona izleyelim. Süreç bir kez diskten veri okusun, bir kez de zaman dilimini doldursun.

Örnek 3.3

1. **Yeni.** Kullanıcı programı başlattı. Çekirdek PCB oluşturuyor ve bellek ayırıyor.
2. **Yeni → Hazır.** Çekirdek süreci kabul etti ve hazır kuyruğuna koydu.
3. **Hazır → Çalışıyor.** Çizelgeleyici bu süreci seçti; bağlamı yüklendi.
4. **Çalışıyor → Bekliyor.** Süreç diskten `read()` istedi. Veri gelene kadar yapacak işi yoktur ve işlemciyi bırakır.
5. **Bekliyor → Hazır.** Disk kesmesi geldi: veri hazır. Süreç doğrudan çalışmaz, hazır kuyruğuna döner. O anda işlemcide başka bir süreç çalışıyor olabilir.
6. **Hazır → Çalışıyor.** Yeniden seçildi.
7. **Çalışıyor → Hazır.** Zamanlayıcı kesmesi geldi: zaman dilimi doldu. Bu **kesmeli** (*preemptive*) çizelgelemedir; süreç bir şey beklemediği için bekliyor durumuna değil hazır durumuna geçer.
8. **Hazır → Çalışıyor.** Tekrar seçildi.
9. **Çalışıyor → Sonlandı.** Süreç `exit()` çağırdı. Ebeveyn `wait` edene kadar zombi olarak süreç tablosunda kalır.

Bu ömür boyunca süreç hazır kuyruğuna 3 kez girdi (2., 5. ve 7. adımlarda), 1 kez bekledi ve işlemciye 3 kez seçildi.



Şekil 3.17: İzlemenin iki anı. (a) 4. adım: süreç `read()` çağırdı ve bekliyor durumuna geçti. (b) 5. adım: disk kesmesiyle olay tamamlandı; süreç hazır durumuna döndü.

Sık yapılan hata

Beklediği olay gerçekleşen sürecin doğrudan çalışmaya geçtiğini düşünmek. Modelde *Bekliyor → Çalışıyor* oku yoktur. Olay gerçekleştiğinde işlemci büyük olasılıkla başka bir sürece aittir. Uyanan süreç önce hazır kuyruğuna girer; işlemciyi alıp almayacağına ve ne zaman alacağına çizelgeleyici karar verir. Bir başka yaygın hata, zaman dilimi dolan süreci bekliyor durumuna koymaktır. Zaman dilimi dolan süreç hiçbir şey beklemez; yalnız sırası geçmiştir ve hazır durumuna döner.

3.4.3 Linux'ta süreç durumları

Beş durumlu model kavramsaldır; gerçek çekirdekler bazı durumları birleştirir, bazılarını böler. Linux'ta durumlar `ps` komutunun `STAT` sütununda tek harfle gösterilir:

STAT	Anlam	Modeldeki karşılığı
R	çalışıyor ya da çalışmaya hazır	Çalışıyor + Hazır
S	kesilebilir uyku (olay bekliyor)	Bekliyor
D	kesilemez uyku (genelde disk G/Ç)	Bekliyor
T	durdurulmuş (SIGSTOP , hata ayıklayıcı)	askıya alınmış
Z	zombi	Sonlandı (biçilmedi)
I	boşta çekirdek iş parçacığı	—

Tablo 3.9: Linux süreç durumları ve beş durumlu modeldeki karşılıkları.

```
$ ps -eo pid,stat,comm
  PID STAT COMMAND
    1 Ss   systemd
  890 Ss   sshd
 2105 Ss   bash
 2241 R+   ps
 2307 Z    yaz <defunct>
```

Bu çıktıda `systemd`, `sshd` ve `bash` `S` durumundadır: hepsi bir olay bekler. Sonlarındaki `s` harfi sürecin bir **oturum lideri** olduğunu gösterir. `ps`'in kendisi `R+`'dır: o anda çalışan tek süreçtir ve `+` ön plan süreç grubunda olduğunu belirtir. `yaz` adlı süreç ise `Z` durumundadır; biçilmemiş bir zombidir.

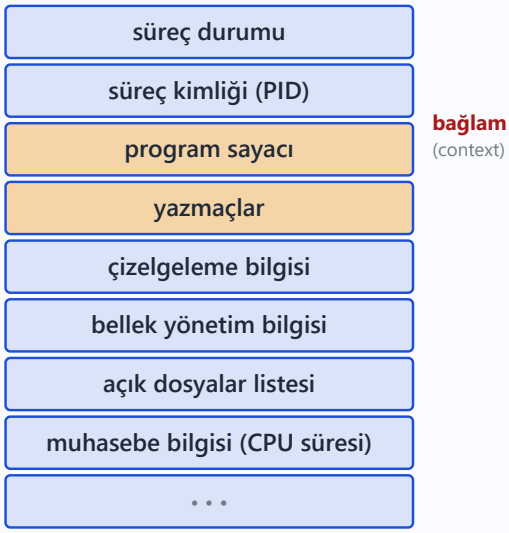
Linux hazır ile çalışıyor durumlarını ayırmaz; ikisi de `R` ile gösterilir. Bekliyor durumu ise ikiye bölünür. `S` durumundaki süreç bir sinyalle uyandırılabilir; örneğin Ctrl+C ile klavye bekleyen bir program hemen sonlanır. `D` durumundaki süreç ise sinyallere yanıt vermez. Bu durum, yarıda kesilmesi tutarsızlık doğuracak kısa disk işlemleri için kullanılır. `D`'de uzun süre takılı kalan süreçler genellikle yanıt vermeyen bir disk ya da ağ dosya sistemine işaret eder; bunlar `kill -9` ile bile sonlandırılamaz.

3.4.4 Süreç kontrol bloğu

Bir süreç işlemciden ayrıldığında yazmaçlarındaki değerler bir yere kaydedilmelidir; yoksa süreç kaldığı yerden devam edemez. Çekirdeğin sürece dair bilmesi gereken başka şeyler de vardır: durumu, önceliği, belleği, açık dosyaları. Bunların hepsi tek bir kayıtta toplanır.

Tanım 3.9

Süreç kontrol bloğu (*process control block*, PCB), çekirdeğin her süreç için tuttuğu ve o sürece dair bilinmesi gereken her şeyi içeren veri yapısıdır. Bütün PCB'lerin oluşturduğu yapıya **süreç tablosu** denir. PCB'nin, süreç işlemciden ayrılırken kaydedilen ve geri dönerken yüklenen kısmına (program sayacı ve yazmaçlar) **bağlam** (*context*) denir.



Şekil 3.18: Bir PCB'nin başlıca alanları: süreç durumu, PID, program sayacı, yazmaçlar, çizelgeleme bilgisi, bellek yönetim bilgisi, açık dosyalar listesi ve muhasebe bilgisi. Program sayacı ile yazmaçlar sürecin bağlamını oluşturur.

PCB'ler çekirdek belleğinde durur. Kullanıcı süreçleri onlara doğrudan erişemez; kendi PCB'lerindeki bir bilgiyi ancak sistem çağrılarıyla okuyabilir ya da değiştirebilir. `getpid()` PCB'den PID'i okur; `nice()` önceliği değiştirir; `open()` açık dosyalar listesine bir girdi ekler. Bu kısıtlama güvenliğin temelidir: bir süreç kendi önceliğini keyfince yükseltemez ya da başka bir sürecin dosyalarına erişemez.

3.4.5 PCB alanları

Tanenbaum, bir süreç tablosu girdisindeki alanları sorumlu oldukları yönetim işine göre üç gruba ayırır. Tablo 3.10 bu gruplamayı verir.

Süreç yönetimi	Bellek yönetimi	Dosya yönetimi
yazmaçlar · program sayacı · program durum sözcüğü (PSW) · yığın işaretçisi · süreç durumu · öncelik · çizelgeleme parametreleri · PID · ebeveyn süreç · süreç grubu · sinyaller · başlama zamanı · kullanılan CPU zamanı · çocukların CPU zamanı · sonraki alarm zamanı	metin, veri ve yığın bölümlerinin işaretçileri; günümüzde sayfa tablosu işaretçisi ve bellek bölgeleri listesi	kök dizin · çalışma dizini · dosya tanımlayıcıları · kullanıcı kimliği (UID) · grup kimliği (GID)

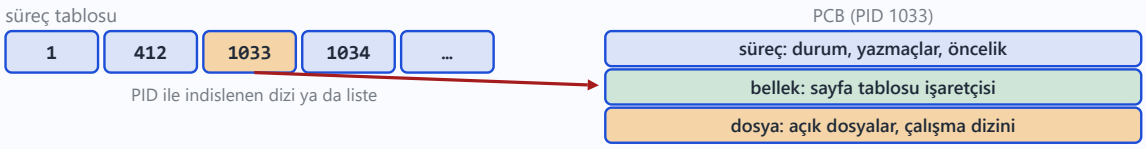
Tablo 3.10: Tanenbaum'a göre PCB alanları.

Tablodaki üç alan ayrıca açıklanmayı hak eder:

- **Program durum sözcüğü** (*program status word*, PSW): işlemcinin durum bayraklarını (sıfır, taşma, elde …), çalışma modunu (kullanıcı ya da çekirdek) ve kesme izinlerini tutan yazmaçtır. Bir karşılaştırmanın sonucu bu bayraklarda durduğu için PSW de bağlamın parçasıdır. Süreç, karşılaştırma ile koşullu atlama arasında kesilse bile doğru dala gitmelidir.
- **Süreç grubu**: birlikte yönetilen süreçlerdir. Kabukta `ls | grep x | wc -l` gibi bir boru hattı tek bir grup oluşturur; Ctrl+C'ye basıldığında `SIGINT` grubun hepsine gider.

- **Sonraki alarm zamanı**: süreç `alarm(saniye)` ile bir süre sonra uyarılmak isterse, `SIGALRM` sinyalinin teslim edileceği zaman burada tutulur.

Bellek yönetimi sütunundaki "metin, veri ve yığın bölümlerinin işaretçileri" ifadesi eski, bölümlemeli bellek düzenini yansıtır. Günümüzde PCB sürecin sayfa tablosunu ve bellek bölgelerinin listesini gösterir. Şekil 3.19, süreç tablosunun PID ile erişilen girdilerden oluştuğunu ve her girdinin bu üç gruptan bilgi taşıdığını gösterir.



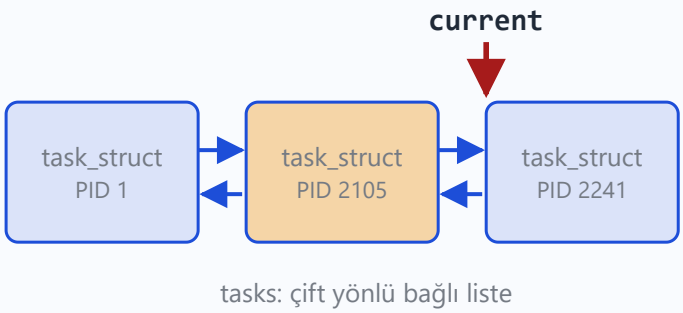
Şekil 3.19: Süreç tablosu: her girdi bir PCB'dir. PID 1033'ün girdisinde süreç (durum, yazmaçlar, öncelik), bellek (sayfa tablosu işaretçisi) ve dosya (açık dosyalar, çalışma dizini) bilgileri vardır.

3.4.6 Linux'ta PCB: task_struct

Linux'ta PCB'nin adı `task_struct`'tır ve `<linux/sched.h>` başlık dosyasında tanımlıdır. Linux süreç ile iş parçacığını aynı yapıyla temsil eder; ikisi de çekirdek gözünde bir "görev"dir (*task*). Yapının birkaç yüz alanı vardır; en önemlileri şunlardır:

```
struct task_struct {          /* <linux/sched.h> */
    pid_t pid;                    /* süreç kimliği */
    unsigned int __state;         /* süreç durumu */
    int prio;                     /* öncelik */
    struct sched_entity se;       /* CFS/EEVDF bilgisi */
    struct task_struct *parent;   /* ebeveyn süreç */
    struct list_head children;    /* çocuklar */
    struct files_struct *files;   /* açık dosyalar */
    struct mm_struct *mm;         /* adres uzayı */
    /* ... yüzlerce alan daha ... */
};
```

Alanların her biri, Tablo 3.10'daki bir grubun karşılığıdır. `pid` kimliği, `__state` durumu, `prio` önceliği taşır. `se` çizelgeleyicinin süreç başına tuttuğu bilgilerdir: sürecin ne kadar CPU zamanı kullandığı ve çizelgeleyicinin ağacındaki yeri. `parent` ve `children` süreç ağacını kurar. `files` açık dosya tablosunu, `mm` ise adres alanını (sayfa tablosu ve bellek bölgeleri) gösterir. `mm` ile `files`'ın işaretçi olması önemlidir: aynı süreçteki iş parçacıkları aynı `mm` ve `files` yapısını gösterir, böylece belleği ve dosyaları paylaşır.



Şekil 3.20: Bütün `task_struct`'lar çift yönlü bağlı bir listededir. `current`, o çekirdekte çalışan görevi gösterir.

Çekirdek bütün görevleri çift yönlü bağlı bir listede tutar (Şekil 3.20). Her çekirdekte `current` adlı bir işaretçi o anda çalışan görevin `task_struct`'ını gösterir. Bir sistem çağrısı geldiğinde çekirdek `current` üzerinden çağıranın bilgilerine ulaşır. Örneğin `getpid()` kabaca `current->pid` değerini döndürür (gerçekte iş parçacığı grubunun kimliğini döndürür).

Çekirdek kodu zamanla değişir; ders kitaplarındaki alan adları da eskiyebilir. Eski çekirdeklerde durum alanının adı `long state` idi ve `time_slice` diye ayrı bir alan vardı. Güncel çekirdeklerde durum `__state` adını almıştır. Zaman dilimi bilgisi de `sched_entity` yapısının içine taşınmıştır.

3.5 Çizelgeleme altyapısı

Çoklu programlamanın amacı işlemciyi sürekli meşgul tutmaktır. Bunun için çekirdeğin hangi süreçlerin işlemciyi beklediğini, hangilerinin bir olayı beklediğini bilmesi ve işlemciyi bir süreçten ötekine aktarabilmesi gerekir. Hangi sürecin seçileceğini belirleyen *politikalar* (ilk gelen önce, en kısa iş önce, döner sıralı ...) CPU çizelgeleme konusunun parçasıdır. Bu kesim politikaların üzerine kurulduğu *mekanizmayı* anlatır: kuyrukları, çoklu programlamanın yararını ölçen basit bir modeli ve işlemciyi süreçler arasında aktaran bağlam değiştirmeyi.

3.5.1 Hazır ve bekleme kuyrukları

Çizelgeleyici, hazır süreçler arasından işlemciye geçecek olanı seçer. İki hedefi vardır: işlemciyi boş bırakmamak ve kullanıcıya hızlı yanıt vermek. Seçimi yapabilmesi için süreçlerin düzenli bir biçimde tutulması gerekir. Çekirdek bunun için kuyruklar kullanır.

Tanım 3.10

Hazır kuyruğu (*ready queue*), bellekte bulunan ve işlemciyi bekleyen süreçlerin listesidir. **Bekleme kuyrukları** (*wait queues*) ise belirli bir olayı (bir diskin G/Ç'yi bitirmesi, bir çocuğun sonlanması, ağdan paket gelmesi) bekleyen süreçleri tutar. Her olay türü ya da her aygıt için ayrı bir bekleme kuyruğu vardır.



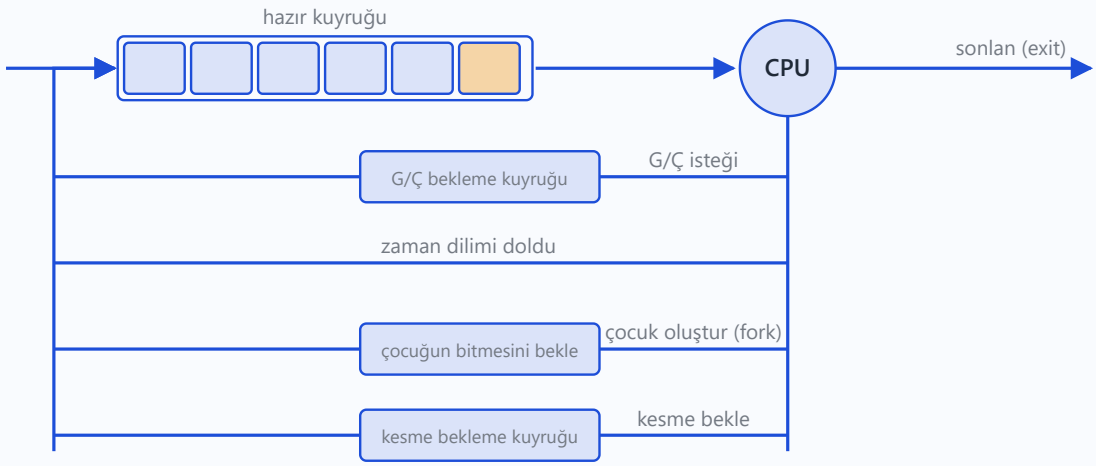
Şekil 3.21: Kuyruklar PCB'lere bağlı listeler olarak tutulur. Hazır kuyruğunda PCB 7, 2 ve 9; disk G/Ç bekleme kuyruğunda PCB 3 ve 14; ağ/klavye bekleme kuyruğunda PCB 6 vardır. PCB 5 işlemcide çalışıyor.

Kuyruklar ayrı bir veri yapısı değil, PCB'ler arasındaki bağlantılardır. Her kuyruğun başını ve sonunu gösteren iki işaretçisi vardır; her PCB'de de sıradaki PCB'yi gösteren bir alan bulunur. Bir sürecin bir kuyruktan ötekine geçmesi, PCB'sinin bir listeden çıkarılıp ötekine eklenmesidir; PCB'nin kendisi kopyalanmaz. Bir süreç aynı anda yalnız bir kuyrukta bulunur, çalışan süreç ise hiçbir kuyrukta değildir.

Gerçek sistemler basit bir FIFO listeden daha akıllı yapılar kullanır. Linux her işlemci çekirdeğine ayrı bir **çalışma kuyruğu** (*runqueue*) atar. Böylece çekirdekler hazır süreç seçerken birbirini beklemez. CFS ve onun yerini alan EEVDF çizelgeleyicileri bu kuyruğu kırmızı-siyah ağaç olarak tutar: sıradaki süreci bulmak hızlıdır ve süreç ekleyip çıkarmak *O(log n)* sürer.

3.5.2 Kuyruk diyagramı

Bir sürecin ömrü, kuyruklar arasında bir yolculuk olarak da çizilebilir. Şekil 3.22'deki **kuyruk diyagramında** dikdörtgenler kuyrukları, daire işlemciyi, oklar ise geçişleri gösterir.



Şekil 3.22: Süreç çizelgelemenin kuyruk diyagramı. Süreç hazır kuyruğundan işlemciye geçer; oradan dört yoldan biriyle ayrılır: G/Ç isteği, zaman diliminin dolması, çocuk oluşturup bekleme ya da bir kesmeyi bekleme.

Sonunda `exit` ile sistemden çıkar.

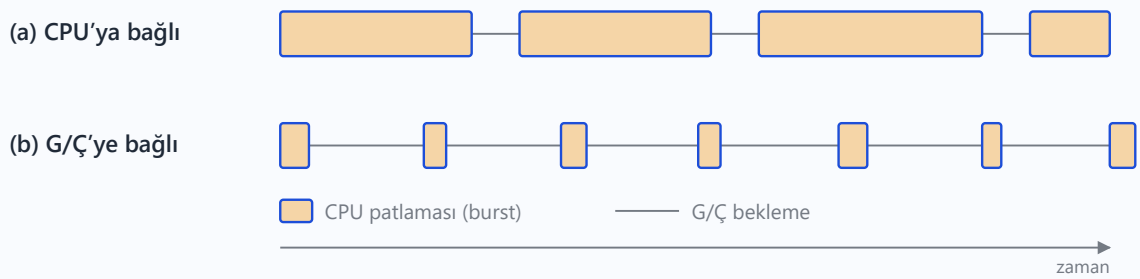
Yolculuk şöyle ilerler. Yeni süreç hazır kuyruğunun sonuna eklenir ve sırasını bekler. Çizelgeleyici onu seçtiğinde işlemcide çalışır. Çalışan süreç işlemciyi dört yoldan biriyle bırakabilir:

- **G/Ç isteği** yapar ve ilgili G/Ç bekleme kuyruğuna geçer. G/Ç bitince hazır kuyruğuna döner.
- **Zaman dilimi dolar**; zamanlayıcı kesmesiyle işlemciden alınır ve doğrudan hazır kuyruğuna döner. Bu yolda bekleme kuyruğu yoktur.
- **Çocuk oluşturur** ve `wait()` çağırır; çocuk bitene kadar bekler.
- **Bir kesmeyi ya da olayı bekler**; örneğin `sleep()` süresinin dolmasını ya da bir ağ paketinin gelmesini.

Süreç bu döngüyü işi bitene kadar tekrarlar. Sonunda `exit()` ile sistemden çıkar; PCB'si ebeveyn `wait` ettiğinde silinir. Diyagram, Şekil 3.16'daki durum modelinin kuyruklarla çizilmiş biçimidir: hazır kuyruğu “Hazır” durumuna, bekleme kuyrukları “Bekliyor” durumuna, işlemci “Çalışıyor” durumuna karşılık gelir.

3.5.3 CPU'ya bağlı ve G/Ç'ye bağlı süreçler

Hemen her süreç iki tür etkinlik arasında gidip gelir: işlemcide hesap yapar, sonra G/Ç bekler, sonra yine hesap yapar. Kesintisiz hesap yapılan aralıklara **CPU patlaması** (*CPU burst*) denir. Süreçler bu patlamaların uzunluğuna göre iki gruba ayrılır (Şekil 3.23).



Şekil 3.23: (a) CPU'ya bağlı süreç: uzun CPU patlamaları, seyrek G/Ç. (b) G/Ç'ye bağlı süreç: kısa CPU patlamaları, sık G/Ç bekleme.

Tanım 3.11

CPU'ya bağlı (*CPU-bound*) süreç zamanının çoğunu hesaplamayla geçirir; uzun CPU patlamaları ve seyrek G/Ç istekleri vardır. Derleme, video kodlama ve bilimsel hesap bu türdendir. **G/Ç'ye bağlı** (*I/O-bound*) süreç ise zamanının çoğunu G/Ç bekleyerek geçirir; CPU patlamaları kısadır. Metin editörü, web sunucusu ve veritabanı istemcisi bu türdendir.

Bir sürecin hangi grupta olduğunu patlamaların uzunluğu belirler; G/Ç'nin süresi belirlemez. G/Ç'ye bağlı bir süreç, verisi geldikten sonra az hesap yapıp hemen yeni bir G/Ç ister. İşlemciler disklerden çok daha hızlı geliştiği için zamanla daha çok süreç G/Ç'ye bağlı hâle gelmiştir. Aynı hesap her yıl daha kısa sürer, ama diski ya da ağı beklemek aynı oranda kısalmaz.

Bu ayrım çizelgeleyici için önemlidir. G/Ç'ye bağlı bir süreç hazır olduğunda işlemciyi hemen almalıdır; kısa işini bitirip yeni bir G/Ç başlatır ve diski meşgul eder. Bu süreç hazır kuyruğunda uzun bir CPU'ya bağlı sürecin arkasında beklerse disk boş kalır, kullanıcı da yavaş yanıt alır. İyi bir sistem iki türü dengeli karıştırır: CPU'ya bağlı süreçler işlemciyi, G/Ç'ye bağlı süreçler aygıtları meşgul eder.

3.5.4 Çoklu programlamanın modellenmesi

Bellekte kaç süreç tutmak gerekir? Basit bir olasılık modeli bu soruya kaba ama öğretici bir cevap verir.

Teorem 3.1 (Çoklu programlama modeli)

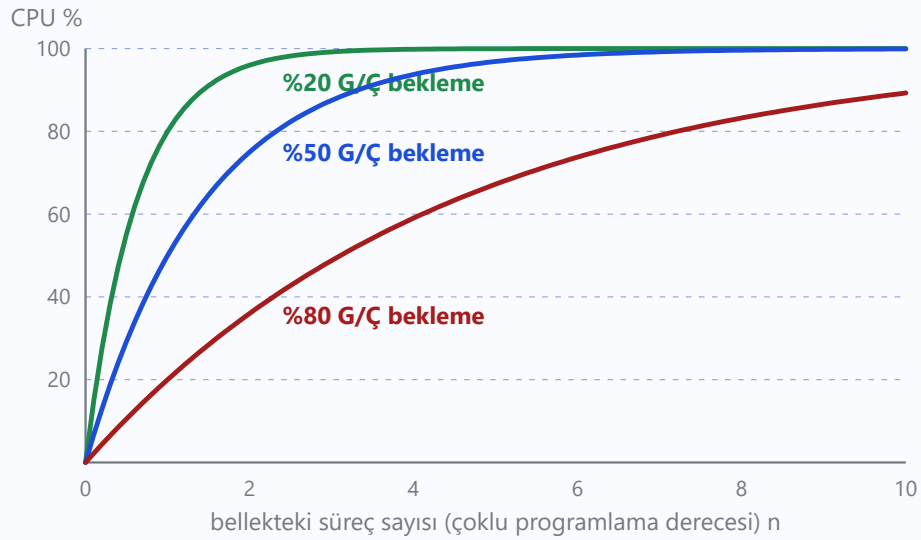
Her süreç zamanının p kesrini G/Ç bekleyerek geçirsin ve süreçler birbirinden bağımsız olsun. Bellekte n süreç varsa CPU kullanımı

$$\text{CPU kullanımı} = 1 - p^n$$

olur. Burada n'ye **çoklu programlama derecesi** denir.

İspat. İşlemci yalnız bellekteki süreçlerin hepsi aynı anda G/Ç beklerken boş kalır; en az biri hazırsa işlemci çalışır. Bir sürecin herhangi bir anda beklemekte olma olasılığı p'dir. Süreçler bağımsız olduğu için n sürecin hepsinin aynı anda beklemesi olasılığı olasılıkların çarpımıdır: $p \cdot p \cdot \ldots \cdot p = p^n$. İşlemcinin boş olma olasılığı p^n olduğuna göre meşgul olma olasılığı $1 - p^n$'dir.

■



Şekil 3.24: CPU kullanımının çoklu programlama derecesine göre değişimi; %20, %50 ve %80 G/Ç bekleme için üç eğri. Bekleme oranı yükseldikçe aynı kullanıma ulaşmak için daha çok süreç gerekir.

Şekil 3.24'teki eğriler modelin iki dersini gösterir. Birincisi, G/Ç'ye bağlı süreçlerde (p büyük) tek programlı sistem israftır. $p = 0{,}8$ ise tek süreçle kullanım yalnız %20'dir; 4 süreçle %59'a, 10 süreçle %89'a çıkar. İkincisi, eğri giderek yataylaşır. Her yeni süreç bir öncekinden daha az kazandırır, çünkü kullanım %100'e hiçbir zaman ulaşmaz.

Model kabadır. Süreçlerin bağımsız olduğu varsayılır; oysa tek bir diski bekleyen süreçler sırayla hizmet alır ve beklemeleri birbirine bağlıdır. Yine de model bellek planlaması için yararlıdır: bellek artırılınca kaç süreç daha sığacağı ve bunun kullanımı ne kadar artıracağı hızlıca kestirilebilir. Örneğin her süreç 2 GB kullanıyorsa ve $p = 0{,}8$ ise 8 GB'lık (4 süreç) bir makineye 8 GB daha eklemek kullanımı %59'dan %83'e çıkarır. Sonraki 8 GB ise yalnız %93'e çıkarır.

Örnek 3.4

Süreçler zamanlarının %60'ını G/Ç bekleyerek geçiriyor. (a) 3 süreçle CPU kullanımı nedir? (b) Kullanımın en az %95 olması için bellekte en az kaç süreç bulunmalıdır?

1. **(a)** $p = 0{,}6$ ve $n = 3$: $0{,}6^3 = 0{,}216$. Kullanım $1 - 0{,}216 = 0{,}784$, yani %78,4'tür.
2. **(b) Eşitsizlik.** $1 - 0{,}6^n \geq 0{,}95$ olmalıdır; bu da $0{,}6^n \leq 0{,}05$ demektir.
3. **Deneme.** $0{,}6^5 = 0{,}07776 \rightarrow$ kullanım %92,2; yetmez. $0{,}6^6 = 0{,}046656 \rightarrow$ kullanım %95,3; yeter.
4. **Logaritmayla denetim.** $n \geq \ln 0{,}05 \,/\, \ln 0{,}6 \approx 5{,}86$. n tam sayı olduğu için yukarı yuvarlanır: en az 6 süreç gerekir.

Sık yapılan hata

Logaritmayla bulunan değeri aşağı yuvarlamak. ln 0,05 / ln 0,6 ≈ 5,86 sonucunu 5'e yuvarlamak kullanımı %92,2'de bırakır ve koşul sağlanmaz. Benzer bir tuzak p'yi yanlış yorumlamaktır: p G/Ç bekleme oranıdır, CPU kullanım oranı değildir. “Süreçler zamanlarının %25'inde işlemci kullanıyor” denirse $p = 0{,}75$ alınır.

3.5.5 Kod izleme: CPU kullanımı $1 - p^n$

Modelin değerlerini bir döngüyle tablo hâlinde üretmek kolaydır. Sunumdaki program $p = 0{,}8$ için $n = 1…5$ değerlerini hesaplar:

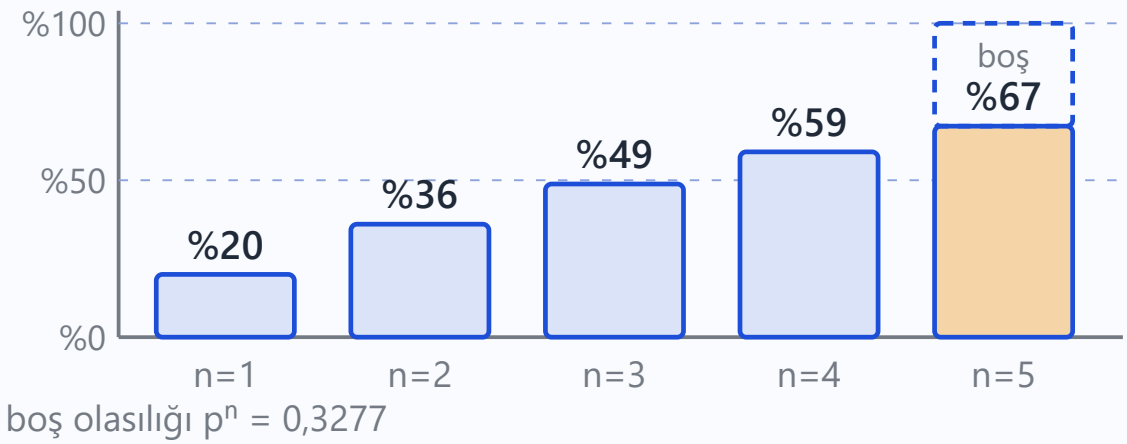
```
double p = 0.8;      /* G/Ç oranı */
double pn = 1.0;
for (int n = 1; n <= 5; n++) {
  pn = pn * p;       /* hepsi bekler */
  double kul = 1 - pn;
  printf("n=%d %%%.0f\n", n, kul * 100);
}
```

Programın püf noktası, p^n'yi her turda baştan hesaplamamasıdır. `pn` değişkeni bir önceki turun değerini tutar ve her turda bir kez `p` ile çarpılır. Böylece n. turda `pn` = p^n olur. `printf`'teki `%%` bir yüzde işareti basar; `%.0f` ise sayıyı ondalıksız, yuvarlayarak yazar.

n	pn	kul	Çıktı satırı	Artış (puan)
1	0,8	0,2	`n=1 %20`	—
2	0,64	0,36	`n=2 %36`	16
3	0,512	0,488	`n=3 %49`	12,8
4	0,4096	0,5904	`n=4 %59`	10,24
5	0,32768	0,67232	`n=5 %67`	8,19

Tablo 3.11: Döngünün izlemesi ($p = 0{,}8$).

Tablonun son sütunu modelin “azalan kazanç” özelliğini sayılarla gösterir. İkinci süreç kullanımı 16 puan artırır; beşinci süreç yalnız 8,19 puan. Aslında her artış bir öncekinin tam 0,8 katıdır: n'den $n + 1$'e geçişte artış $p^n - p^{n+1} = p^n(1 - p)$ olur. Çıktıda 48,8'in `%49` olarak, 67,232'nin `%67` olarak yazılması da `%.0f`'in yuvarlamasından gelir.



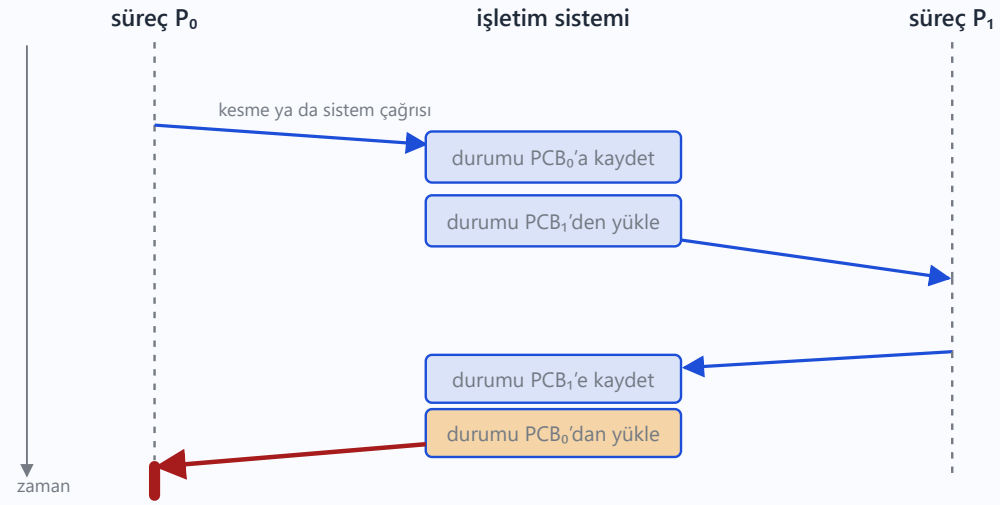
Şekil 3.25: Döngü bittiğinde n = 1…5 için CPU kullanımı: %20, %36, %49, %59, %67. Çubukların üst kısmı işlemcinin boş kaldığı kesirdir (p^n).

3.5.6 Bağlam değiştirme

Çizelgeleyici bir süreci seçtiğinde işlemcinin o sürece geçmesi gerekir. Bu geçiş, işlemcide o an çalışan sürecin durumunu bir yere saklamayı ve seçilen sürecin durumunu işlemciye yüklemeyi gerektirir.

Tanım 3.12

Bağlam değiştirme (*context switch*), işlemciyi bir süreçten ötekine aktarma işlemidir. Çekirdek çalışan sürecin bağlamını (program sayacı, yazmaçlar, PSW) onun PCB'sine kaydeder, seçilen sürecin bağlamını kendi PCB'sinden yükler ve adres alanını yeni sürecinkine çevirir.



Şekil 3.26: İki süreç arasında bağlam değiştirme. P_0 çalışırken bir kesme ya da sistem çağrısı gelir; çekirdek P_0'ın durumunu PCB_0'a kaydeder, P_1'inkini PCB_1'den yükler. Sonra ters yönde aynı iş yapılır. Gri bölümlerde

hiçbir kullanıcı işi yapılmaz.

Şekil 3.26 zaman ekseninde iki geçişi gösterir. P_0 çalışırken bir kesme ya da sistem çağrısı gelir. İşlemci çekirdek moduna geçer ve P_0'ın program sayacıyla yazmaçları PCB_0'a kaydedilir. Çizelgeleyici P_1'i seçer; P_1'in bağlamı PCB_1'den yüklenir ve adres alanı değişir. P_1 çalışırken P_0 boştadır: hazır ya da bekliyor durumundadır. Bir sonraki kesmede P_1'in durumu PCB_1'e kaydedilir, PCB_0'dakiler geri yüklenir ve P_0 kaldığı yerden devam eder.

Şekildeki gri bölümler bağlam değiştirmenin bedelidir. O sürede işlemci çalışır ama hiçbir kullanıcı programı ilerlemez. Bu yüzden bağlam değiştirme saf **ek yüktür** (*overhead*).

3.5.7 Kod izleme: bağlam değiştirme

Bağlam değiştirme gerçek çekirdeklerde assembly ile yazılır, çünkü yazmaçlara doğrudan erişmek gerekir. Mantığı ise birkaç satır C ile modellenebilir. Aşağıdaki programda `P` dizisi süreç tablosudur; her PCB'de üç yazmaç (`pc`, `sp`, `r1`) ve bir durum harfi vardır. Global `pc`, `sp` ve `r1` değişkenleri işlemcinin yazmaçlarını temsil eder.

```
struct pcb { int pc, sp, r1; char durum; };
struct pcb P[2];   /* süreç tablosu */
int pc, sp, r1;    /* CPU yazmaçları */
void baglam_degistir(int e, int y) {
  P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
  P[e].r1 = r1;  P[e].durum = 'H';
  pc = P[y].pc;  sp = P[y].sp;  /* yükle */
  r1 = P[y].r1;  P[y].durum = 'C';
}
/* kesme */  baglam_degistir(0, 1);
/* ... P1 çalışır ... */
/* kesme */  baglam_degistir(1, 0);
```

`baglam_degistir(e, y)` iki yarıdan oluşur. İlk yarı eski sürecin (`e`) yazmaçlarını onun PCB'sine kaydeder ve durumunu `'H'` (hazır) yapar. İkinci yarı yeni sürecin (`y`) yazmaçlarını işlemciye yükler ve durumunu `'C'` (çalışıyor) yapar. Başlangıçta P_0 çalışıyor: `pc = 104`, `sp = 896`, `r1 = 8`. P_1'in son durumu PCB_1'de saklıdır: `pc = 340`, `sp = 500`, `r1 = 2`.

Adım	Satır	CPU	PCB_0	PCB_1	Açıklama
1	10	104/896/8 (P_0)	—/—/— C	340/500/2 H	zamanlayıcı kesmesi; çekirdek P_1'e geçmeye karar verir
2	5	104/896/8	104/896/— C	340/500/2 H	P_0'ın `pc` ve `sp`'si kaydedildi
3	6	104/896/8	104/896/8 H	340/500/2 H	`r1` de kaydedildi; P_0 hazır

Adım	Satır	CPU	PCB_0	PCB_1	Açıklama
4	7	340/500/8	104/896/8 H	340/500/2 H	P_1'in `pc` ve `sp`'si yüklendi
5	8	340/500/2 (P_1)	104/896/8 H	340/500/2 C	`r1` yüklendi; P_1 çalışıyor
6	11	352/492/5	104/896/8 H	340/500/2 C	P_1 birkaç komut çalıştırdı; PCB_1 artık güncel değil
7	12	352/492/5	104/896/8 H	340/500/2 C	yeni kesme; P_0'a dönülecek
8	5	352/492/5	104/896/8 H	352/492/2 C	P_1'in `pc` ve `sp`'si kaydedildi
9	6	352/492/5	104/896/8 H	352/492/5 H	`r1` kaydedildi; P_1 hazır
10	7	104/896/5	104/896/8 H	352/492/5 H	PCB_0'dan `pc` ve `sp` yüklendi
11	8	104/896/8 (P_0)	104/896/8 C	352/492/5 H	`r1 = 8`; P_0 kesildiği yerden sürer

Tablo 3.12: Bağlam değiştirme izlemesi. Değerler `pc` / `sp` / `r1` sırasıyla yazılmıştır; "—" henüz kaydedilmemiş alanı gösterir.

İzlemede üç ara durum dikkat çeker. 4. adımda işlemcinin yazmaçları karışıktır: `pc` ve `sp` P_1'e, `r1` hâlâ P_0'a aittir. Bu yarım durum, bağlam değiştirmenin kesintisiz yapılması gerektiğini gösterir. Gerçek çekirdekler bu sırada kesmeleri kapatır. 6. adımda ise PCB_1'deki değerler (340/500/2) güncel değildir; P_1'in gerçek durumu yalnız işlemcidedir. PCB, süreç çalışırken değil, işlemciden ayrılırken güncellenir. 11. adımda P_0 tam olarak kesildiği durumdan, `pc = 104`'ten devam eder ve kesildiğini fark etmez.

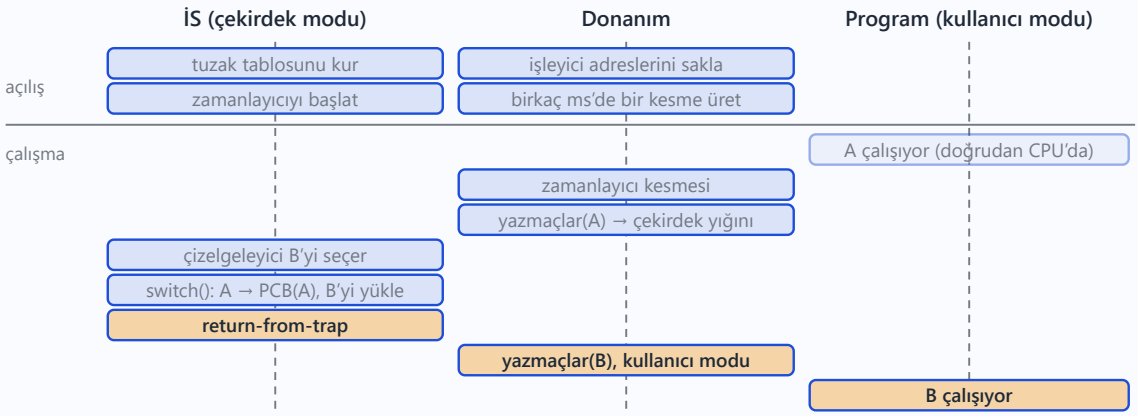
Model gerçeği sadeleştirir. Gerçek bir işlemcide onlarca genel yazmaç, kayan nokta ve vektör yazmaçları ile PSW vardır. Ayrıca adres alanı da değişir: işlemcinin sayfa tablosu yazmacı yeni sürecin tablosunu gösterecek biçimde yüklenir.

3.5.8 Sınırlı doğrudan yürütme

Bağlam değiştirme bir soruyu cevapsız bırakır: işlemci bir kullanıcı programını çalıştırırken çekirdek nasıl olur da denetimi geri alır? Çekirdek de bir programdır ve işlemci aynı anda yalnız bir şey çalıştırır. Kullanıcı programı sonsuz bir döngüye girerse çekirdeğin kodu hiç çalışmayabilir.

Bir çözüm, kullanıcı programını bir yorumlayıcıda adım adım çalıştırmak olabilirdi; ama bu çok yavaştır. İşletim sistemlerinin kullandığı yol **sınırlı doğrudan yürütmedir** (*limited direct execution*). Program işlemcide doğrudan, tam hızla çalışır. Ama iki sınırı vardır: ayrıcalıklı işleri kendisi yapamaz, çekirdekten ister; işlemciyi de istediği kadar tutamaz, çekirdek onu zorla geri alır. Şekil 3.27 bu düzeni beş adımda gösterir.

1. **Açılışta (çekirdek modunda).** İşletim sistemi donanıma hangi olayda hangi kodun çalışacağını bildirir: **tuzak** (*trap*) ve kesme tablosunu kurar. Ardından **zamanlayıcıyı** başlatır; zamanlayıcı birkaç milisaniyede bir kesme üretecektir. İki işlem de ayrıcalıklı komutlarla yapılır, kullanıcı programı bunları değiştiremez.
2. **A çalışıyor.** Süreç A kullanıcı modunda doğrudan işlemcide çalışır; arada yorumlayıcı yoktur. Dosya açmak gibi ayrıcalıklı bir iş için sistem çağrısıyla tuzağa düşmek zorundadır.
3. **Zamanlayıcı kesmesi.** A sonsuz döngüde olsa bile kesme gelir. Donanım A'nın yazmaçlarını çekirdek yığınına kaydeder, çekirdek moduna geçer ve işletim sisteminin kesme işleyicisine atlar. Denetim artık işletim sistemindedir.
4. **Çizelgeleyici B'yi seçer.** A'nın zaman dilimi dolduğu için işletim sistemi B'ye geçmeye karar verir. `switch()` yordamı A'nın bağlamını PCB(A)'ya yazar, B'ninkini yükler ve B'nin çekirdek yığınına geçer.
5. **Tuzaktan dönüş.** `return-from-trap` komutu, B sanki kesildiği yerden dönüyormuş gibi B'nin yazmaçlarını yükler ve kullanıcı moduna geçer. Artık B çalışır.



Şekil 3.27: Sınırlı doğrudan yürütme. Açılışta işletim sistemi tuzak tablosunu ve zamanlayıcıyı kurar. A doğrudan işlemcide çalışırken zamanlayıcı kesmesi denetimi işletim sistemine verir; `switch()` A'dan B'ye geçer ve tuzaktan dönüşle B çalışmaya başlar.

Zamanlayıcı kesmesi bu düzenin bel kemiğidir. Kesme olmasaydı işletim sistemi denetimi ancak süreç kendi isteğiyle bir sistem çağrısı yaptığında geri alabilirdi. Bu yaklaşıma **işbirlikçi çok görevlilik** (*cooperative multitasking*) denir ve Windows 3.x ile eski Mac OS sürümleri böyle çalışırdı. Bu sistemlerde sonsuz döngüye giren tek bir program bütün makineyi kilitlerdi; çözüm makineyi yeniden başlatmaktı.

3.5.9 Bağlam değiştirmenin maliyeti

Bağlam değiştirme bedavaya gelmez. İşlemci başka bir sürece geçerken eski sürecin durumunu kaydetmeli, yenisininkini yüklemelidir ve bu süre boyunca yararlı iş yapılmaz. PCB ne kadar büyük ve işletim sistemi ne kadar karmaşıksa geçiş o kadar uzun sürer. Maliyet iki türlüdür:

- **Doğrudan maliyet:** yazmaçların kaydedilip yüklenmesi ve çizelgeleyici kodunun çalışması. Güncel Linux'ta tipik olarak birkaç mikrosaniyedir.
- **Dolaylı maliyet:** önbelleğin ve TLB'nin (*translation lookaside buffer*, adres çevirilerini tutan küçük önbellek) "soğuması". Yeni süreç ilk anlarda verisini önbellekte bulamaz ve çok sayıda önbellek ıskası yaşar. Bu dolaylı maliyet çoğu zaman doğrudan maliyetten büyüktür.

Donanım bu maliyeti azaltmak için yardım eder. Bazı işlemcilerde birden çok yazmaç kümesi bulunur; geçiş, kümeyi gösteren bir işaretçinin değiştirilmesinden ibaret olur. TLB girdilerine eklenen adres alanı etiketi (ARM'da ASID, x86'da PCID) ise her geçişte TLB'yi boşaltma zorunluluğunu kaldırır: farklı süreçlerin girdileri etiketleriyle ayırt edilir.

Bağlam değiştirmenin maliyeti, zaman dilimi seçiminde doğrudan rol oynar. Dilim çok kısa seçilirse işlemcinin büyük kısmı geçişlere gider; çok uzun seçilirse süreçler uzun süre sıra bekler.

Örnek 3.5

Bir bağlam değiştirme $s = 0{,}05$ ms sürüyor. Hazır kuyruğunda sırayla çalışan 3 CPU'ya bağlı süreç var ve her biri $q = 4$ ms'lik dilimini tam kullanıyor. (a) İşlemci zamanının yüzde kaçı geçişlere gider? (b) Bir süreç iki dilimi arasında en çok ne kadar bekler?

1. **Bir döngüyü belirle.** Her dilimin ardından bir geçiş gelir. Yani her $q + s = 4{,}05$ ms'lik aralığın 0,05 ms'si ek yüktür.
2. **(a)** Ek yük oranı $s \,/\, (q + s) = 0{,}05 \,/\, 4{,}05 \approx 0{,}0123$, yani yaklaşık %1,23'tür.
3. **(b)** Süreç A dilimini bitirince bir geçiş olur, B çalışır; bir geçiş, C çalışır; bir geçiş daha ve A yeniden başlar. A'nın beklediği süre 3 geçiş ve 2 dilimdir: $3 \cdot 0{,}05 + 2 \cdot 4 = 8{,}15$ ms.
4. **Genelleme.** k süreç için bekleme $k \cdot s + (k - 1) \cdot q$'dur. Dilim küçüldükçe bekleme kısalır ama ek yük oranı büyür.

3.5.10 Kesmenin ele alınması

Bağlam değiştirmenin çoğu bir kesmeyle başlar. Tanenbaum, bir G/Ç kesmesi geldiğinde işletim sisteminin en alt katmanında olanları sekiz adımda özetler:

1. **Donanım program sayacını, PSW'yi ve birkaç yazmacı yığına kaydeder.** Kesme isteği bir komutun ortasında değil, komut sınırında kabul edilir. İşlemci çekirdek moduna geçer.
2. **Donanım kesme vektöründen yeni program sayacını yükler. Kesme vektörü**, her kesme türü için işleyicinin adresini tutan tablodur. İşletim sistemi bu tabloyu açılışta doldurur.
3. **Assembly yordamı yazmaçları kaydeder.** Donanımın kaydetmediği yazmaçlar o anki sürecin PCB'sine ya da çekirdek yığınına yazılır.
4. **Assembly yordamı yeni bir yığın hazırlar.** Kesme işleyicisi kullanıcının yığınında değil, çekirdek yığınında çalışır.

5. **C ile yazılmış kesme hizmet yordamı çalışır.** Genellikle aygıttan gelen veriyi okur ve tampona koyar. Bu veriyi bekleyen bir süreç varsa onu hazır durumuna geçirir.
6. **Çizelgeleyici sıradaki süreci seçer.** Uyanan süreç kesilen süreçten daha öncelikliyse o seçilebilir. Bu durumda kesme bir bağlam değiştirmeye dönüşür.
7. **C yordamı assembly koduna döner.**
8. **Assembly yordamı seçilen süreci başlatır.** Sürecin yazmaçları ve bellek eşlemesi yüklenir ve kullanıcı moduna dönülür.

Adımlardaki iş bölümü dikkat çekicidir. İlk iki adımı donanım yapar, çünkü o anda hiçbir yazılımın çalışabileceği güvenli bir ortam yoktur. Yazmaçlara doğrudan dokunan 3., 4. ve 8. adımlar assembly ile yazılır. Asıl iş ise taşınabilir C kodunda yapılır. Kesmeyle başlayan bu yol, kesilen sürecin kesildiğini fark etmeden devam edebilmesini sağlar; ya da çizelgeleyici karar verirse, işlemciyi bir başka sürece verir.

3.6 Süreçler arası iletişim

Süreçlerin ayrı adres alanları, birinin hatasının ötekine bulaşmasını önler. Ama aynı ayrılık, süreçlerin birlikte çalışmasını zorlaştırır: bir süreç ötekinin değişkenini okuyamaz. İşbirliği yapan süreçler veri alışverişi için işletim sisteminin sunduğu araçları kullanır. Bu kesim bu araçların iki temel modelini, paylaşımlı belleği ve ileti geçirmeyi, ardından UNIX'in boru ve soketlerini anlatır.

3.6.1 Bağımsız ve işbirlikçi süreçler

Bir süreç, başka süreçleri etkilemiyor ve onlardan etkilenmiyorsa **bağımsızdır**. Veri paylaşan ya da birbirinin işini etkileyen süreçler ise **işbirlikçidir**. Süreçlerin işbirliği yapmasının dört nedeni vardır:

- **Bilgi paylaşımı:** Birden çok uygulama aynı veriye erişmek isteyebilir; örneğin bir panodaki kopyalanmış metne.
- **Hızlanma:** Büyük bir iş parçalara bölünür ve parçalar farklı çekirdeklerde paralel işlenir.
- **Modülerlik:** Sistem, her biri tek bir işi yapan küçük süreçlerden kurulur.
- **Güvenlik ve yalıtım:** Riskli işler ayrı, yetkisi kısıtlı süreçlerde yapılır; Chrome'un oluşturucu süreçleri bunun örneğidir.

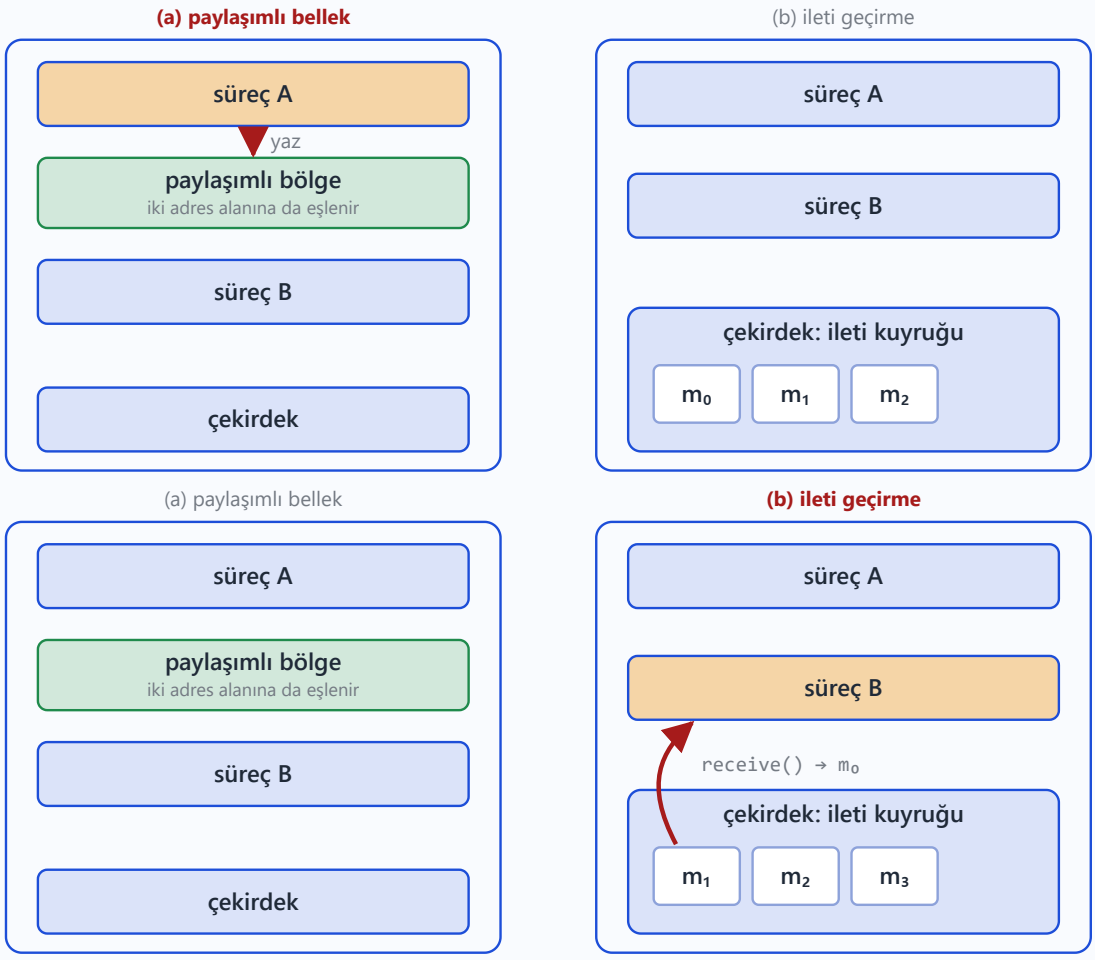
> **Tanım 3.13**
>
> **Süreçler arası iletişim** (*interprocess communication*, IPC), işbirlikçi süreçlerin veri alışverişi yapmasını ve eylemlerini eşgüdümlemesini sağlayan mekanizmaların genel adıdır. İki temel modeli vardır: **paylaşımlı bellek** ve **ileti geçirme**.

IPC üç soruya cevap vermelidir. Birincisi açıktır: bir süreç ötekine bilgiyi nasıl iletir? Öteki ikisi daha incedir. İkinci soru, süreçlerin birbirinin yoluna çıkmamasıdır. İki rezervasyon süreci aynı anda son boş koltuğu görüp ikisi de satarsa aynı koltuğa iki bilet kesilir. Üçüncü soru, bağımlılıklarda doğru sıranın sağlanmasıdır. Nişan almadan ateş edilmemeli; bir veri üretilmeden tüketilmemelidir.

Son iki soru **senkronizasyon** (*synchronization*) gerektirir. Paylaşılan veriye denetimsiz erişim, sonucun süreçlerin hangi sırayla çalıştığına bağlı olmasına yol açar. Bu duruma **yarış durumu** (*race condition*) denir. Aynı sorular, aynı adres alanını paylaşan iş parçacıkları için de geçerlidir.

3.6.2 Paylaşımlı bellek ve ileti geçirme

İki model, verinin süreçler arasında nereden geçtiği bakımından ayrılır (Şekil 3.28).



Şekil 3.28: İki IPC modeli. (a) Paylaşımlı bellek: çekirdek bir bölgeyi iki sürecin adres alanına eşler, A bölgeye doğrudan yazar. (b) İleti geçirme: çekirdek iletileri kendi kuyruğunda tutar; B `receive()` ile sıradaki iletiyi (m_0) alır.

Paylaşımlı bellekte çekirdek bir bellek bölgesi oluşturur ve onu iki sürecin adres alanına eşler. Sistem çağrısı yalnız kurulumda gerekir. Sonrasında A bölgeye sıradan bir bellek yazmasıyla yazar, B sıradan bir bellek okumasıyla okur. Çekirdek bu alışverişte hiç devreye girmez; bu yüzden paylaşımlı bellek çok hızlıdır. Bedeli şudur: kimin ne zaman yazacağını çekirdek denetlemez. Eşgüdümü süreçler kendileri, kilit ya da semafor gibi araçlarla sağlamalıdır.

İleti geçirmede A, `send(m)` çağrısıyla bir ileti gönderir; çekirdek iletiyi kendi kuyruğuna kopyalar. B, `receive()` çağrısıyla sıradaki iletiyi alır; çekirdek iletiyi bu kez B'nin belleğine kopyalar. Her iletide en az bir sistem çağrısı ve iki kopyalama vardır; bu yüzden ileti geçirme paylaşımlı bellekten yavaştır. Kazancı ise eşgüdümün çekirdekte olmasıdır: boş kuyruktan okumak isteyen süreç bekletilir, ileti yarım okunmaz. Ayrıca aynı arayüz ağ üzerinden de çalışabilir; paylaşımlı bellek ise yalnız aynı makinedeki süreçler için mümkündür.

Ölçüt	Paylaşımlı bellek	İleti geçirme
Çekirdeğin rolü	yalnız kurulumda	her iletide
Kopyalama	yok	iletinin çekirdeğe ve alıcıya kopyalanması
Eşgüdüm	süreçlerin sorumluluğu	çekirdek sağlar
Uygun olduğu durum	aynı makinede büyük veri	küçük iletiler; dağıtık sistemler

Tablo 3.13: Paylaşımlı bellek ile ileti geçirmenin karşılaştırması.

3.6.3 POSIX paylaşımlı bellek

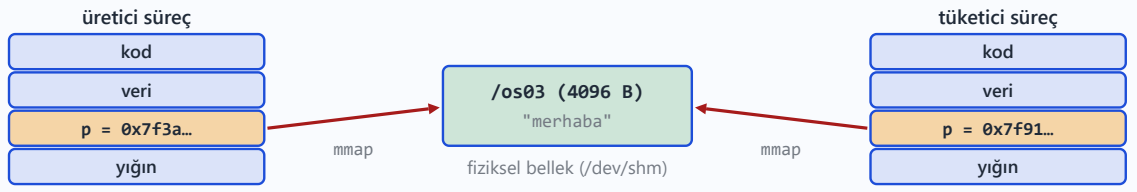
POSIX standardı paylaşımlı belleği bir dosya gibi adlandırılan bir nesneyle kurar. Aşağıdaki kodda iki ayrı sürecin parçaları vardır: üstteki üretici, alttaki tüketicidir.

```
/* üretici */
int fd = shm_open("/os03", O_CREAT|O_RDWR, 0600);
ftruncate(fd, 4096);
char *p = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
               MAP_SHARED, fd, 0);
strcpy(p, "merhaba");   /* doğrudan yaz */

/* tüketici (başka süreç) */
int fd = shm_open("/os03", O_RDONLY, 0);
char *p = mmap(NULL, 4096, PROT_READ,
               MAP_SHARED, fd, 0);
printf("%s\n", p);
shm_unlink("/os03");    /* sil */
```

Kurulum üç adımdan oluşur. `shm_open` `/os03` adlı bir paylaşımlı bellek nesnesi oluşturur ya da var olanı açar ve bir dosya tanımlayıcısı döndürür. Linux'ta bu nesne `/dev/shm` altında bir dosya olarak görünür. `ftruncate` nesnenin boyutunu 4096 bayta ayarlar. `mmap` ise nesneyi sürecin adres alanına eşler ve eşlemenin başlangıç adresini döndürür. `MAP_SHARED` bayrağı önemlidir: bu bölgeye yapılan yazmalar aynı nesneyi eşleyen bütün süreçlerde görünür.

Kurulumdan sonra paylaşımlı bellek sıradan bellektir. Üretici `strcpy` ile `"merhaba"`'yı doğrudan yazar; arada hiçbir sistem çağrısı yoktur. Tüketici aynı nesneyi salt okunur açar, eşler ve `printf` ile okur. İşi bitince `shm_unlink` nesnenin adını siler; nesne, onu eşlemiş son süreç de bıraktığında yok olur.



Şekil 3.29: İki süreç aynı fiziksel bölgeyi (`/os03` , 4096 bayt) kendi adres alanlarında farklı sanal adreslere eşler: üreticide `p = 0x7f3a…` , tüketicide `p = 0x7f91…` . İkisi de aynı `"merhaba"` verisini görür.

Şekil 3.29 ince bir noktayı da gösterir: iki süreçteki `p` değerleri farklıdır. Bölge her sürecin adres alanında farklı bir sanal adrese eşlenebilir. Bu yüzden paylaşımlı bölgeye işaretçi saklamak tehlikelidir; bir süreçte geçerli olan adres ötekinde anlamsız olabilir. Paylaşımlı veri yapıları işaretçi yerine bölgenin başından itibaren uzaklık (*offset*) kullanır.

> **Sık yapılan hata**
>
> Paylaşımlı belleğin eşgüdümü de kendiliğinden sağladığını sanmak. Örnek kodda tüketici, üretici yazmadan önce çalışırsa bölgeyi boş görür ve boş bir dizgi yazdırır. Daha kötüsü, üretici yazarken tüketici okursa yarım bir veri görebilir. Gerçek kodda bölgeye erişim bir semafor ya da kilitle düzenlenmelidir. Bunun eski bir arayüzü de vardır: System V'nin `shmget` ve `shmat` çağrıları. Mantık aynıdır, yalnız nesneler dosya adı yerine sayısal bir anahtarla adlandırılır.

3.6.4 İleti geçirme tasarımı

İleti geçirmenin iki temel işlemi vardır: `send(mesaj)` ve `receive(mesaj)` . İki süreç haberleşmeden önce aralarında bir **iletişim bağı** kurulmalıdır. Bu bağın tasarımında dört soru öne çıkar.

Boyut	Seçenekler
Fiziksel bağ	paylaşımlı bellek, donanım veri yolu, ağ
Adlandırma	doğrudan (sürecin adı) · dolaylı (posta kutusu ya da port)
Eş zamanlı / eş zamansız	bloklayan (senkron): gönderen ileti alınana kadar, alıcı ileti gelene kadar bekler · bloklamayan (eş zamansız): çağrı hemen döner
Tamponlama	sıfır kapasite (gönderen bekler: buluşma) · sınırlı kapasite (kuyruk doluysa gönderen bekler) · sınırsız kapasite (gönderen hiç beklemez)

Tablo 3.14: İleti geçirme tasarımının boyutları.

Tamponlama, bağın kaç iletiyi bekletebileceğidir. **Sıfır kapasiteli** bir bağda iletiyi saklayacak yer yoktur. Gönderen, alıcı `receive` çağırana kadar bekler ve ileti iki süreç buluştuğu anda doğrudan aktarılır. **Sınırlı kapasiteli** bir bağda kuyrukta *N* ileti bekleyebilir; kuyruk doluysa gönderen bekler. **Sınırsız kapasite** kavramsaldır; gerçekte bellek her zaman sınırlıdır.

Tanım 3.14

Buluşma (*rendezvous*), hem `send`'in hem `receive`'in bloklayan olduğu ileti geçirmedir. Gönderen, alıcı iletiyi alana kadar; alıcı, ileti gelene kadar bekler. Aktarım iki süreç aynı anda hazır olduğunda yapılır.

Buluşma, üretici–tüketici problemini kendiliğinden çözer. Üretici, tüketici bir iletiyi almadan yenisini gönderemez; tüketici de üretilmemiş bir iletiyi alamaz. Programcının ayrıca kilit ya da sayaç tutmasına gerek kalmaz. Gerçek sistemlerdeki örnekler arasında POSIX ileti kuyrukları (`mq_send`, `mq_receive`), macOS'un temelindeki Mach çekirdeğinin portları ve Windows'un ALPC (*Advanced Local Procedure Call*) mekanizması vardır.

Örnek 3.6

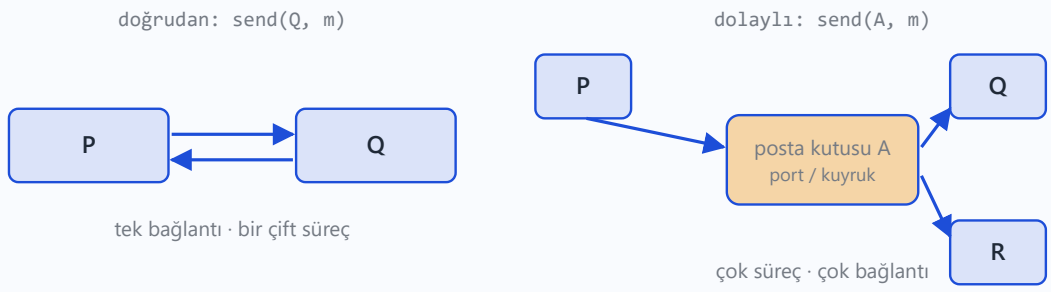
Bir ileti kuyruğunun kapasitesi 2'dir; `send` ve `receive` bloklayandır. Tüketici henüz hiç `receive` çağırmamışken üretici art arda m_1, m_2 ve m_3 iletilerini göndermek istiyor. Ne olur?

1. `send(`m_1`)` **:** Kuyruk boştur; m_1 kuyruğa kopyalanır ve çağrı hemen döner. Kuyrukta 1 ileti vardır.
2. `send(`m_2`)` **:** Yer vardır; m_2 eklenir. Kuyruk artık doludur (2/2).
3. `send(`m_3`)` **:** Kuyrukta yer yoktur; üretici bekliyor durumuna geçer.
4. **Tüketici** `receive()` **çağırır:** m_1'i alır (ileti kuyruğu FIFO'dur). Bir yer boşalır; çekirdek bekleyen üreticiyi uyandırır ve m_3 kuyruğa eklenir. Kuyrukta m_2 ve m_3 vardır.

Kapasite sıfır olsaydı üretici daha ilk `send`'de bekler, tüketici `receive` çağırınca m_1 doğrudan aktarılırdı.

3.6.5 Doğrudan ve dolaylı iletişim

Adlandırma sorusu, gönderenin iletiyi kime yolladığını nasıl söylediğidir (Şekil 3.30).



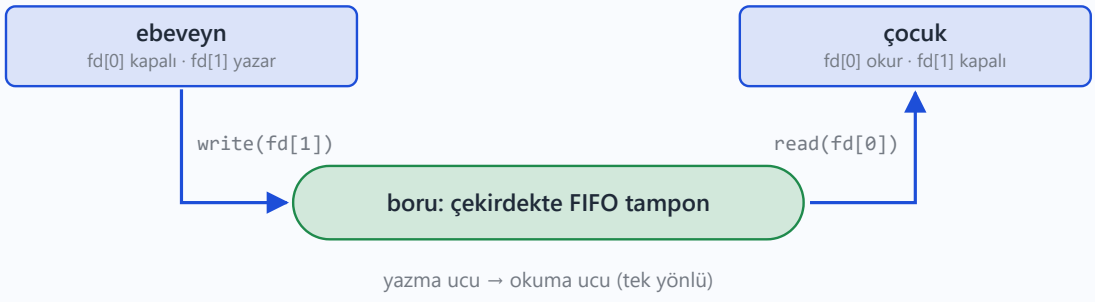
Şekil 3.30: Solda doğrudan iletişim: P, Q'ya adıyla gönderir; bağlantı tek bir süreç çiftine aittir. Sağda dolaylı iletişim: P, Q ve R, A posta kutusu üzerinden haberleşir; bir bağlantı çok sürece, bir çift çok bağlantıya ait olabilir.

Doğrudan iletişimde her süreç karşı tarafı adıyla anar: `send(P, m)` P'ye gönderir, `receive(Q, m)` Q'dan alır. Bağlantı otomatik kurulur ve tam olarak bir süreç çiftine aittir. Her çift arasında tek bir bağlantı vardır ve bağlantı genellikle çift yönlüdür. Bu yaklaşımın zayıflığı, süreç adlarının koda gömülmesidir: bir süreç yeniden başlayıp yeni bir kimlik alırsa onu anan bütün kodlar değişmelidir.

Dolaylı iletişimde iletiler bir **posta kutusuna** (*mailbox*) ya da **porta** gönderilir ve oradan alınır: `send(A, m)`, `receive(A, m)`. Her posta kutusunun tekil bir kimliği vardır; aynı kutuyu paylaşan süreçler haberleşebilir. Bir bağlantı ikiden çok sürece ait olabilir ve bir süreç çifti birden çok kutu üzerinden haberleşebilir. Bağlantı tek ya da çift yönlü olabilir. Bu esneklik bir soruyu da doğurur: P bir kutuya ileti koyduğunda Q ve R aynı anda `receive` çağırırsa iletiyi kim alır? Sistemler bunu ya aynı anda tek alıcıya izin vererek ya da alıcıyı kendisi seçerek çözer.

3.6.6 Borular

Boru (*pipe*), UNIX'in en eski ve en çok kullanılan IPC aracıdır. Bir boru, çekirdekte tutulan ve iki ucu olan bir FIFO tampondur. Bir uçtan yazılan baytlar, aynı sırayla öteki uçtan okunur (Şekil 3.31).



Şekil 3.31: Sıradan boru: ebeveyn `fd[1]` yazma ucuna yazar, çocuk `fd[0]` okuma ucundan okur. Veri çekirdekteki FIFO tampondan geçer ve tek yönlü akar. Her süreç kullanmadığı ucu kapatır.

Bir boru tasarlanırken dört soru sorulur. Akış tek yönlü mü (*simplex*), çift yönlü mü? Çift yönlüyse iki yön aynı anda mı kullanılabilir (tam çift yönlü, *full duplex*) yoksa sırayla mı (yarı çift yönlü, *half duplex*)? Haberleşen süreçler arasında ebeveyn–çocuk ilişkisi gerekli mi? Boru ağ üzerinden kullanılabilir mi? İki boru türü bu sorulara farklı cevaplar verir.

	Sıradan (adsız)	**Adlandırılmış (FIFO)**
Erişim	yalnız oluşturan süreç ve çocukları	dosya adını bilen her süreç
Yön	tek yönlü	UNIX: yarı çift yönlü; Windows: tam çift yönlü
Ömür	süreçler kapatınca biter	`mkfifo` ile oluşur; dosya sisteminde kalır
Ağ	hayır	UNIX: hayır; Windows: evet

Tablo 3.15: Sıradan ve adlandırılmış borular.

Sıradan borunun adı yoktur; ona yalnız dosya tanımlayıcılarıyla erişilir. Bu tanımlayıcılar da yalnız `fork` ile miras yoluyla başka süreçlere geçebilir. Bu yüzden sıradan boru akraba süreçler arasında kullanılır. Adlandırılmış boru ise dosya sisteminde bir ad taşır (`mkfifo /tmp/kanal`). Birbirini tanımayan iki süreç bu adı açarak haberleşebilir. UNIX'te FIFO'lar her iki yönde kullanılabilse de pratikte her yön için ayrı bir FIFO açılır, çünkü iki süreç aynı FIFO'ya yazarsa veriler karışır.

3.6.7 Kod izleme: pipe() ve fork()

Aşağıdaki program, ebeveynin bir boru üzerinden çocuğa `"merhaba"` göndermesini gösterir:

```
int main(void) {
  int fd[2];
  char buf[32];
  pipe(fd);
  pid_t pid = fork();
  if (pid == 0) {   /* çocuk: okur */
    close(fd[1]);
    ssize_t n = read(fd[0], buf, sizeof buf - 1);
    buf[n] = '\0';
    printf("çocuk aldı: %s\n", buf);
    close(fd[0]);
    return 0;
  }
  close(fd[0]);     /* ebeveyn: yazar */
  write(fd[1], "merhaba", 7);
  close(fd[1]);
  wait(NULL);
  return 0;
}
```

`pipe(fd)` çağrısı bir boru oluşturur ve iki tanımlayıcı döndürür: `fd[0]` okuma ucu, `fd[1]` yazma ucudur. 0, 1 ve 2 numaraları standart giriş, çıkış ve hata akışlarına ait olduğu için en küçük boş numaralar 3 ve 4'tür. Ardından `fork` gelir ve çocuk iki ucu da miras alır. Böylece borunun iki ucu da iki sü-

reçte açıktır. Her süreç kullanmayacağı ucu kapatır: çocuk yazma ucunu, ebeveyn okuma ucunu. Ebeveynin PID'i 1234, çocuğunki 1235 olsun.

Adım	Satır	Olay	Açık uçlar	Tampon	Çıktı
1	4	`pipe(fd)` : `fd[0] = 3` , `fd[1] = 4`	E: 3, 4	boş	
2	5	`fork` : çocuk uçları miras alır	E: 3, 4 · Ç: 3, 4	boş	
3	7	çocuk yazma ucunu kapatır	E: 3, 4 · Ç: 3	boş	
4	8	çocuk `read` : boru boş → bekliyor	E: 3, 4 · Ç: 3	boş	
5	14	ebeveyn okuma ucunu kapatır	E: 4 · Ç: 3	boş	
6	15	ebeveyn 7 bayt yazar; çocuk uyanır	E: 4 · Ç: 3	`merhaba`	
7	16–17	ebeveyn yazma ucunu kapatır, `wait` ile bekler	Ç: 3	`merhaba`	
8	8	`read` 7 döndürür; `n = 7`	Ç: 3	boş	
9	9–10	`buf[7] = '\0'` ; çocuk yazdırır	Ç: 3	boş	`çocuk aldı: merhaba`
10	11–12, 17–18	çocuk okuma ucunu kapatıp çıkar; ebeveyn `wait` 'ten döner ve biter	—	—	

Tablo 3.16: `pipe()` + `fork()` izlemesi. "Açık uçlar" sütununda E ebeveyni, Ç çocuğu gösterir.

İzlemedeki adımların sırası çizelgelemeye göre değişebilir. Örneğin ebeveyn, çocuk `read` çağırmadan önce yazabilir; o zaman çocuk hiç beklemeden veriyi alır. Çıktı her sırada aynıdır, çünkü yalnız çocuk yazdırır ve veri tek bir yoldan akar.

İzlemenin üç ayrıntısı boruların kurallarını gösterir. Birincisi, boş bir borudan okuyan süreç bekletilir (4. adım). İkincisi, `PIPE_BUF` 'tan (Linux'ta 4096 bayt) küçük ya da ona eşit yazmalar **bölünmez**: ya bütünüyle yazılır ya hiç. Böylece aynı boruya yazan iki sürecin küçük iletileri birbirine karışmaz. Üçüncüsü, `read` en çok istenen kadar bayt döndürür ama daha azını da döndürebilir. Program 31 bayt istemiştir (`sizeof buf - 1`), boruda 7 bayt vardır ve `read` 7 döndürür. Dönen değer, dizginin sonuna `'\0'` koymak için kullanılır.

Yazma ucunun kapatılması da önemlidir. `read` , boru boşken ve *hiçbir* süreçte açık yazma ucu kalmamışken 0 döndürür; bu dosya sonu (EOF) demektir. Yazma ucu bir yerde açık kaldıkça okuyucu "belki daha veri gelir" diye bekler.

Sık yapılan hata

Kullanılmayan boru uçlarını kapatmamak. Çocuk EOF'a kadar okuyan bir döngü kullansın ve ebeveyn `close(fd[1])` çağırmasın. Bu durumda boru boşaldığında bile bir yazma ucu açık kalır ve `read` hiçbir zaman 0 döndürmez. Çocuk sonsuza dek bekler, ebeveyn de `wait` 'te ço-

cuğu bekler: program kilitlenir. Çocuğun kendi yazma ucunu kapatmaması da aynı sonucu doğurur, çünkü o uç da "açık bir yazıcı" sayılır. Ters yönde de bir kural vardır: okuma ucu tamamen kapalı bir boruya yazan süreç `SIGPIPE` sinyali alır ve varsayılan olarak sonlanır.

3.6.8 Kabukta boru hattı

Kabukta `ls | wc -l` yazıldığında `ls`'nin çıktısı `wc`'nin girdisi olur ve ekranda dizindeki dosya sayısı görünür. Kabuk bunu, boruyu yönlendirmeyle birleştirerek yapar:

```
int fd[2]; pipe(fd);
if (fork() == 0) { dup2(fd[1], 1); close(fd[0]); close(fd[1]);
                   execlp("ls", "ls", NULL); }
if (fork() == 0) { dup2(fd[0], 0); close(fd[0]); close(fd[1]);
                   execlp("wc", "wc", "-l", NULL); }
close(fd[0]); close(fd[1]); wait(NULL); wait(NULL);
```



Şekil 3.32: `ls | wc -l` boru hattı. Kabuk bir boru ve iki çocuk oluşturur; `ls`'nin standart çıkışı borunun yazma ucuna, `wc`'nin standart girişi okuma ucuna yönlendirilir. Linux'ta boru tamponu varsayılan olarak 64 KiB'dir.

Kabuk önce bir boru oluşturur, sonra iki kez `fork` çağırır. Birinci çocuk `dup2(fd[1], 1)` ile standart çıkışını borunun yazma ucuna bağlar. `dup2(eski, yeni)`, `yeni` numaralı tanımlayıcıyı `eski`'nin kopyası yapar; `yeni` açıksa önce onu kapatır. Çocuk ardından iki boru tanımlayıcısını da kapatır, çünkü işi artık 1 numara üzerinden görecektir. Sonra `ls`'ye dönüşür. İkinci çocuk aynı işi okuma ucu ve standart giriş (0) için yapar ve `wc -l`'ye dönüşür. Kabuk kendi uçlarını kapatır ve iki çocuğu bekler.

Burada da kapatma kuralı belirleyicidir. Kabuk ya da ikinci çocuk yazma ucunu açık tutsaydı, `ls` bitip kendi ucunu kapattığında bile boruda açık bir yazıcı kalırdı. `wc` EOF göremez ve satır saymayı hiç bitiremezdi. Sunumdaki koddan hata denetimleri kısalık için çıkarılmıştır; gerçek bir kabuk her çağrının dönüş değerini denetler.

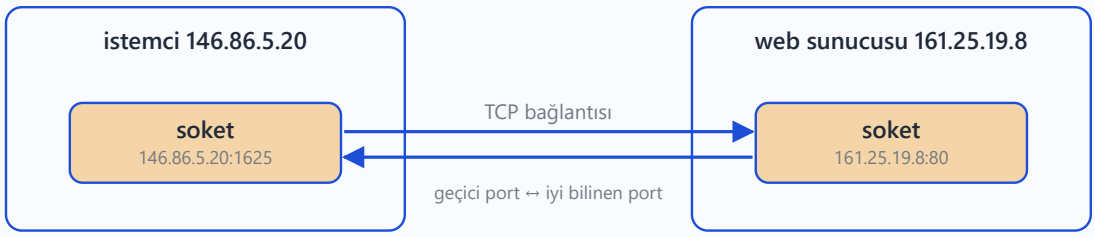
Bu tasarımın gücü, programların boruyu fark etmemesidir. `ls` standart çıkışa, `wc` standart girişe yazıp okur; onlar için terminal, dosya ya da boru arasında fark yoktur. Böylece birbirinden habersiz küçük programlar, kabukta `|` ile zincirlenerek büyük işler yapabilir. UNIX felsefesinin "her program tek bir işi iyi yapsın" ilkesi bu altyapıya dayanır.

3.6.9 Soketler

Borular aynı makinedeki süreçleri birleştirir. Farklı makinelerdeki süreçlerin haberleşmesi için soketler kullanılır.

Tanım 3.15

Soket (*socket*), iletişimin bir uç noktasıdır. Ağ soketi bir IP adresi ile bir port numarasının birleşimiyle tanımlanır; örneğin `161.25.19.8:80`. İletişim her zaman bir soket çifti arasında olur.



Şekil 3.33: Soketlerle iletişim. İstemci (146.86.5.20) geçici bir port (1625) üzerinden web sunucusunun (161.25.19.8) iyi bilinen 80 numaralı portuna TCP bağlantısı kurar.

IP adresi makineyi, port ise makinedeki hizmeti ayırt eder. Bir web sunucusu 80 numaralı portu dinler; istemci bağlanırken işletim sistemi istemciye geçici bir port (şekilde 1625) atar. Bağlantıyı `146.86.5.20:1625` ve `161.25.19.8:80` soketlerinden oluşan çift tanımlar. Aynı makineden aynı sunucuya açılan ikinci bir bağlantı başka bir geçici port alır ve iki bağlantı karışmaz.

1024'ün altındaki portlar **iyi bilinen** hizmetlere ayrılmıştır: SSH 22, HTTP 80, HTTPS 443. Bu portlarda sunucu olarak dinlemeye başlamak (`bind`) yetki gerektirir; sıradan bir kullanıcının sahte bir SSH sunucusu açması böylece önlenir. `127.0.0.1` adresi (IPv6'da `::1`) **geri döngü** (*loopback*) adresidir ve aynı makineyi gösterir; aynı makinedeki süreçler ağ soketleriyle de haberleşebilir.

Soketler iki temel iletişim türü sunar. **TCP** bağlantılıdır: veri sıralı, güvenilir bir bayt akışı olarak iletilir; kaybolan paket yeniden gönderilir. **UDP** bağlantısızdır: her datagram ayrı gönderilir ve ulaşacağı garanti edilmez; buna karşılık daha az ek yük taşır. Aynı makinede ise **UNIX alan soketleri** (`AF_UNIX`) daha hızlıdır, çünkü ağ yığınından geçmezler; dosya sisteminde bir yol adıyla adlandırılırlar. Docker, systemd ve X/Wayland görüntü sunucuları süreçlerle bu soketler üzerinden haberleşir.

3.7 Mobil sistemlerde çoklu görev

Masaüstünde açık bırakılan bir uygulama, kullanıcı başka pencereye geçse de çalışmayı sürdürür. Telefonlarda durum farklıdır. Bellek sınırlıdır, takas alanı ya yoktur ya küçüktür; en önemlisi pil sınırlıdır. Arka planda sürekli işlemci kullanan bir uygulama pili saatler içinde bitirebilir. Bu yüzden mobil işletim sistemleri arka plandaki süreçleri sıkı biçimde sınırlar.

iOS ve iPadOS. İlk iPhone yazılımı (2007) yalnız tek bir kullanıcı uygulamasının çalışmasına izin veriyordu; ön plandaki uygulama dışındakiler kapatılırdı. iOS 4 (2010) sınırlı bir çoklu görev getirdi. Bugün de arka planda yalnız belirli işler sürebilir: ses çalma, konum izleme, VoIP (internet üzerinden sesli görüşme) ve `BGTaskScheduler` ile planlanmış görevler. Bu izinlerden birini kullanmayan bir uygulama arka plana alınınca kısa süre sonra **askıya alınır**: süreç bellekte kalır ama işlemci verilmez. Bellek gerektiğinde sistem askıdaki uygulamaları uyarı vermeden sonlandırabilir; kullanıcı geri döndüğünde uygulama durumunu yeniden kurar.

Android. Android baştan beri arka plan süreçlerine izin verir. Uzun süren işler **hizmet** (*service*) bileşeniyle yürütülür; bir müzik çalar, arayüzü askıdayken bile hizmeti sayesinde çalmayı sürdürür. Ancak Android 8'den (2017) bu yana arka plan kısıtları vardır. Uzun bir iş ya kullanıcıya bildirim gösteren bir **ön plan hizmeti** olarak çalışmalı ya da sistemin uygun bir anda çalıştırdığı `WorkManager` görevlerine bölünmelidir.



Şekil 3.34: Android'de süreçlerin önem sırası: ön plan, görünür, hizmet ve önbellekteki süreçler. Bellek azalınca `lmkd` en alttaki gruptan başlayarak süreçleri sonlandırır.

Android bellek darlığında hangi süreci sonlandıracağına bir önem sıralamasıyla karar verir (Şekil 3.34). En üstte kullanıcının etkileşimde olduğu **ön plan** süreci vardır. Onu ekranda görünen ama odakta olmayan **görünür** süreçler ve arka planda iş yapan **hizmet** süreçleri izler. En altta **önbellekteki** (*cached*) süreçler durur. Eski belgelerde ayrı ayrı anılan "arka plan" ve "boş" süreçler bugün bu grupta toplanır. Bellek azalınca `lmkd` (*low memory killer daemon*) en alttan başlayarak süreçleri sonlandırır. Önbellekteki bir süreci öldürmek kullanıcıya görünmez; uygulama yeniden açıldığında baştan başlatılır.

İki sistemin yaklaşımı farklı olsa da gerekçe aynıdır: sınırlı bellek ve pil. Masaüstünde “süreç kendi isteğiyle biter” varsayımı mobilde geçerli değildir. Mobil uygulamalar her an askıya alınabileceklerini ya da sonlandırılabileceklerini varsayarak yazılır ve durumlarını düzenli olarak kaydeder.

3.8 Özet

- **Süreç**, yürütülmekte olan bir program örneğidir. Program diskte duran pasif bir dosyadır; süreç ise adres alanı, işlemci bağlamı (program sayacı, yazmaçlar) ve kaynakları (açık dosyalar, kimlikler) olan etkin bir varlıktır. Aynı programdan birçok süreç oluşabilir.
- Bir sürecin adres alanı alttan üste **metin**, **veri**, **BSS**, **heap** ve **yığın** bölgelerinden oluşur; en üstte kullanıcı modunda erişilemeyen çekirdek alanı vardır. Heap yukarı, yığın aşağı büyür. İlk değerli global ve `static` değişkenler veride, ilk değersizler BSS'de, yerel değişkenler yığında, `malloc` ile ayrılan alan heap'tedir.
- Tek çekirdek süreçler arasında hızla geçerek **sözde paralellik** yaratır; buna **çoklu programlama** denir. **Eş zamanlılık** işlerin aynı aralıkta ilerlemesi, **paralellik** aynı anda yürümesidir ve birden çok çekirdek gerektirir.
- `fork()` çağıran sürecin kopyasını oluşturur; ebeveyne çocuğun PID'ini, çocuğa 0, hatada ebeveyne −1 döndürür. Kopyalama **yazınca kopyalama** ile tembel yapılır. Art arda n koşulsuz `fork` 2^n süreç üretir. `stdio` tamponundaki yazılmamış veri `fork` ile çoğalır.
- `exec` ailesi sürecin görüntüsünü yeni bir programla değiştirir; PID, ebeveyn ve kapatılmamış dosya tanımlayıcıları korunur. Başarılı `exec` geri dönmez. Altı fonksiyonun hepsi `execve()`'ye varır.
- Kabuk her komutta **fork → exec → wait** döngüsünü çalıştırır. `fork` ile `exec`'in ayrı olması, çocuğun `exec`'ten önce kendi ortamını (yönlendirme, dizin, öncelik) hazırlamasını sağlar. `open` en küçük boş tanımlayıcı numarasını verir.
- Süreç gönüllü olarak (`exit`) ya da istemsiz olarak (ölümcül hata, sinyal) sonlanır. Ebeveyne çıkış durumunun yalnız alttaki 8 biti ulaşır. `SIGTERM` yakalanabilir, `SIGKILL` yakalanamaz.
- **Zombi**, bitmiş ama biçilmemiş süreçtir; PID tüketir. **Yetim**, ebeveyni ölmüş çalışan süreçtir; `init` / `systemd` (ya da bir alt-biçici) onu evlat edinir ve bitince biçer. Zombi birikmesinin çaresi `wait` / `waitpid` ya da `SIGCHLD` işleyicisidir.
- **Beş durumlu model**: Yeni → Hazır ⇄ Çalışıyor → Bekliyor → Hazır … → Sonlandı. Bekliyor → Çalışıyor geçişi yoktur; zaman dilimi dolan süreç Hazır'a döner. Linux hazır ile çalışıyor durumunu `R` olarak birleştirir, beklemeyi `S` ve `D` olarak böler.
- Çekirdek her süreci bir **PCB** ile izler; Linux'ta bu yapı `task_struct`'tır. PCB süreç, bellek ve dosya yönetimi bilgilerini taşır; program sayacı ve yazmaçlar sürecin **bağlamıdır**. Kuyruklar PCB'lerin bağlı listeleridir.
- Bağımsız G/Ç bekleme modelinde CPU kullanımı $\mathbf{1 - p^n}$'dir; her ek süreç bir öncekinden daha az kazandırır.

- **Bağlam değiştirme** eski sürecin bağlamını kaydedip yenisininkini yükler ve saf ek yüktür; dolaylı maliyeti (önbellek ve TLB'nin soğuması) çoğu zaman doğrudan maliyetinden büyüktür. **Sınırlı doğrudan yürütmede** program işlemcide tam hızla çalışır; zamanlayıcı kesmesi denetimi işletim sistemine geri verir.
- IPC'nin iki modeli vardır. **Paylaşımlı bellek** kurulumdan sonra çekirdeksiz ve çok hızlıdır ama eşgüdümü süreçlere bırakır. **İleti geçirme** her iletide çekirdeğe uğrar, eşgüdümü çekirdek sağlar ve ağ üzerinden de çalışır. Bloklayan `send` ve `receive` **buluşma** oluşturur.
- **Boru** çekirdekteki tek yönlü bir FIFO tampondur. Boş borudan okuma bekletir; açık yazma ucu kalmayınca `read` 0 (EOF) döndürür. Kullanılmayan uçlar kapatılmalıdır. Kabuk boru hatlarını `pipe`, `fork`, `dup2` ve `exec` ile kurar. **Soket** IP adresi ve porttan oluşan bir uç noktadır.
- Mobil sistemler sınırlı bellek ve pil yüzünden arka plandaki süreçleri askıya alır ve gerekirse sonlandırır; Android bunu bir önem sıralamasına göre `lmkd` ile yapar.

Çağrı	Ne yapar?	Dönüş değeri
`fork()`	çağıran sürecin kopyasını oluşturur	ebeveynde çocuğun PID'i, çocukta 0, hatada −1
`exec…()`	süreçteki programı değiştirir	başarıda dönmez; hatada −1
`wait()` / `waitpid()`	bir çocuğun bitmesini bekler ve onu biçer	biten çocuğun PID'i; `WNOHANG` ile bitmiş yoksa 0
`exit()`	tamponları boşaltıp süreci sonlandırır	dönmez
`kill()`	bir sürece sinyal gönderir	başarıda 0, hatada −1
`pipe()`	boru oluşturur: `fd[0]` okuma, `fd[1]` yazma ucu	başarıda 0, hatada −1
`dup2(e, y)`	`y` tanımlayıcısını `e`'nin kopyası yapar	`y`

Tablo 3.17: Bölümdeki temel UNIX çağrıları.

3.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

süreç *process* · çalıştırılabilir dosya *executable file* · adres alanı *address space* · program sayacı *program counter (PC)* · yazmaç *register* · iş parçacığı *thread* · çoklu programlama *multiprogramming* · eş zamanlılık *concurrency* · paralellik *parallelism* · ebeveyn / çocuk süreç *parent / child process* · yazınca kopyalama *copy-on-write (COW)* · sayfa hatası *page fault* · kabuk *shell* · dosya tanımlayıcısı *file descriptor* · sistem çağrısı *system call* · sinyal *signal* · zombi süreç *zombie process* · yetim süreç *orphan process* · süreç durumu *process state* · süreç kontrol bloğu *process control block (PCB)* · hazır kuyruğu *ready queue* · çizelgeleyici *scheduler* · görev dağıtıcı *dispatcher* · kesmeli / kesmesiz çizelgeleme *preemptive / nonpreemptive scheduling*

zaman dilimi *time slice (quantum)* · CPU'ya bağlı / G/Ç'ye bağlı *CPU-bound / I/O-bound* · bağlam değiştirme *context switch* · kesme *interrupt* · kesme vektörü *interrupt vector* · tuzak *trap* · zamanlayıcı *timer* · kullanıcı modu / çekirdek modu *user mode / kernel mode* · süreçler arası iletişim *interprocess communication (IPC)* · paylaşımlı bellek *shared memory* · ileti geçirme *message passing* · yarış durumu *race condition* · senkronizasyon *synchronization* · eş zamanlı / eş zamansız iletişim *synchronous / asynchronous communication* · tampon *buffer* · boru *pipe* · soket *socket*

3.10 Kendini Dene

1. Aşağıdaki parça çalıştırıldığında kaç satır `X` yazılır ve toplam kaç süreç oluşur? Süreç ağacı nasıldır?

```c
if (fork() == 0)
    fork();
printf("X\n");
```

Cevap

Başta P0 vardır. İlk `fork` P1'i oluşturur. P0'da dönüş değeri çocuğun PID'idir, yani 0 değildir; koşul yanlıştır ve P0 doğrudan `printf`'e geçer. P1'de dönüş 0'dır; koşul doğrudur ve P1 ikinci `fork` ile P2'yi oluşturur. P1 ve P2 de `printf`'e ulaşır. Toplam 3 süreç (2'si yeni) ve 3 satır `X` vardır. Ağaç bir zincirdir: P0 → P1 → P2.

2. Program ne yazdırır? Çıktının sırası kesin midir?

```c
int x = 1;
pid_t pid = fork();
if (pid == 0) x = x * 3;
else { wait(NULL); x = x + 4; }
printf("%d\n", x);
```

Cevap

Çocuk kendi kopyasında `x = 1 · 3 = 3` hesaplar ve `3` yazar. Ebeveyn önce çocuğu bekler; sonra kendi kopyasında `x = 1 + 4 = 5` hesaplar ve `5` yazar. Ebeveyn yazmadan önce `wait` ile çocuğun bitmesini beklediği için sıra kesindir: önce `3`, sonra `5`. Ebeveynin `x`'i 3 olmaz, çünkü `fork` sonrasında iki ayrı `x` vardır.

3. Aşağıdaki program önce terminalde, sonra `./prog > c.txt` ile çalıştırılıyor. İki durumda ne görülür? Neden?

```
printf("önce\n");
execlp("echo", "echo", "sonra", NULL);
printf("hata\n");
```

Cevap

Terminalde `stdout` satır tamponludur; `"önce\n"` satır sonuyla hemen yazılır. Ardından `exec` başarılı olur ve `echo sonra` yazar. Ekranda iki satır görülür: `önce`, `sonra`. Dosyaya yönlendirmede `stdout` tam tamponludur; `"önce\n"` sürecin belleğindeki tamponda bekler. `exec` adres alanını atınca tampon da kaybolur. `c.txt`'de yalnız `sonra` bulunur. `hata` iki durumda da yazılmaz, çünkü `exec` başarılıdır. Çözüm, `exec`'ten önce `fflush(stdout)` çağırmaktır.

4. Süreçler zamanlarının yarısını G/Ç bekleyerek geçiriyor. CPU kullanımının en az %99 olması için bellekte en az kaç süreç gerekir?

Cevap

$p = 0{,}5$ ve $1 - 0{,}5^n \geq 0{,}99$ olmalıdır; yani $0{,}5^n \leq 0{,}01$. $0{,}5^6 = 0{,}015625 \rightarrow$ kullanım %98,44; yetmez. $0{,}5^7 = 0{,}0078125 \rightarrow$ kullanım %99,22; yeter. Logaritmayla $n \geq \ln 0{,}01 / \ln 0{,}5 \approx 6{,}64$; yukarı yuvarlanınca 7 süreç gerekir.

5. Bir bağlam değiştirme 0,02 ms sürüyor. Hazır kuyruğunda sırayla çalışan 4 CPU'ya bağlı süreç var ve zaman dilimi 1 ms. İşlemci zamanının yüzde kaçı geçişlere gider? Bir süreç iki dilimi arasında en çok ne kadar bekler?

Cevap

Ek yük oranı $0{,}02 / (1 + 0{,}02) = 0{,}02 / 1{,}02 \approx$ %1,96'dır. Bir sürecin dilimi bitince 4 geçiş ve öteki 3 sürecin dilimi geçer: $4 \cdot 0{,}02 + 3 \cdot 1 = 3{,}08$ ms.

6. `ps` çıktısında `STAT` sütunu `S`, `D` ve `Z` olan üç süreç var. Hangisi Ctrl+C ile hemen sonlandırılabilir, hangisi `kill -9`'a bile yanıt vermez, hangisini öldürmek anlamsızdır? Sonuncusu nasıl te-

mizlenir?

Cevap

`S` kesilebilir uykudadır; bir sinyal onu uyandırır ve `SIGINT` ile sonlanır. `D` kesilemez uykudadır, genellikle bir disk işlemini bekler; işlem bitene kadar `SIGKILL` bile teslim edilmez. `Z` zombidir, yani zaten bitmiştir; öldürülecek bir şey kalmamıştır. Zombiyi ebeveyni `wait` ile biçer. Ebeveyn bunu yapmıyorsa ebeveyn sonlandırılır; zombi `init`'e geçer ve `init` onu biçer.

7. Çocuk süreç `exit(256)` ve başka bir çalıştırmada `exit(-1)` ile çıkıyor. Ebeveyn `WEXITSTATUS(st)` ile hangi değerleri okur? Kabuk birinci durumda komutu başarılı mı sayar?

Cevap

Ebeveyne yalnız alttaki 8 bit ulaşır. 256 mod 256 = 0, yani ebeveyn 0 okur. −1'in alttaki 8 biti 1111 1111'dir; ebeveyn 255 okur. Birinci durumda kabuk 0 gördüğü için komutu başarılı sayar, oysa program bir hata bildirmek istemiştir. Bu yüzden taşınabilir programlar yalnız 0–255 aralığını kullanır.

8. Çocuk süreç `exec`'ten önce şu iki satırı çalıştırıyor. `fd`'nin değeri nedir? Sonra `execlp("sort", "sort", NULL)` çağrılırsa `sort` veriyi nereden okur? Bu, kabuktaki hangi yazımın karşılığıdır?

```
close(0);
int fd = open("girdi.txt", O_RDONLY);
```

Cevap

`close(0)` standart girişi kapatır ve 0 numara boşalır. `open` en küçük boş numarayı verdiği için `fd = 0` olur. `exec` dosya tanımlayıcı tablosunu korur; `sort` standart girişten, yani 0 numaradan okur ve bu artık `girdi.txt`'dir. Bu, kabuktaki `sort < girdi.txt` yazımının karşılığıdır.

9. Bir sunucu her istek için `fork` ile bir çocuk oluşturuyor ama hiç `wait` çağırmıyor. Saniyede 50 istek geliyor ve her çocuk işini hemen bitiriyor. Sistemde aynı anda en çok 32.000 süreç bulunabiliyor ve sunucu başladığında 500 süreç var. Ne kadar süre sonra `fork` başarısız olmaya başlar?

Çare nedir?

Cevap

Her çocuk bitince zombi olur ve PID'ini tutmaya devam eder. Boş yer 32.000 − 500 = 31.500 süreçtir. Saniyede 50 zombi biriktiği için 31.500 / 50 = 630 saniye, yani 10,5 dakika sonra yeni süreç oluşturulamaz. Çare, `SIGCHLD` işleyicisinde `waitpid(-1, NULL, WNOHANG)` döngüsüyle biten çocukları biçmektir.

3.11 İleri okuma

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, "Processes", "Process API" ve "Limited Direct Execution" bölümleri: süreç soyutlamasını, `fork` / `exec` / `wait` kalıbını ve denetimin zamanlayıcı kesmesiyle geri alınmasını kısa bölümlerle anlatır. Kitap çevrim içi ve ücretsizdir.
- A. Silberschatz, P. B. Galvin ve G. Gagne, *Operating System Concepts* (10. baskı, 2018), Bölüm 3: süreç durumlarını, PCB'yi ve IPC modellerini Windows ve Android örnekleriyle daha geniş işler.
- A. S. Tanenbaum ve H. Bos, *Modern Operating Systems* (4. baskı), Kesim 2.1: süreç modelini, süreç tablosu alanlarını, kesmenin ele alınmasını ve çoklu programlama modelini ayrıntılı verir.
- M. Kerrisk, *The Linux Programming Interface* (2010), Bölüm 24–28 ve 44: `fork`, `exec`, `wait`, süreç sonlandırma ve borular için bir başvuru kitabıdır. Tamponlar, sinyaller ve zombiler gibi Linux'a özgü tuzakları ayrıntılı işler.

3.12 Kaynaklar

1. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books, pages.cs.wisc.edu/~remzi/OSTEP.
2. Chromium Projects, "Multi-process Architecture" ve "Site Isolation" belgeleri.
3. Android Developers, "Processes and app lifecycle".
4. M. Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010.
5. Linux kılavuz sayfaları: fork(2), execve(2), wait(2), pipe(7), signal(7), shm_overview(7).
6. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018.
7. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı, Pearson, 2014 (5. baskı 2023).

KISIM II · BÖLÜM 4

İş Parçacıkları

Bir süreç, çalışan programın bütün kaynaklarını bir arada tutar: adres alanını, açık dosyaları, sinyal işleyicilerini. Ama tek bir sürecin içinde birden çok işin aynı anda ilerlemesi çoğu zaman gerekir. Tarayıcı bir yandan sayfayı çizer, bir yandan ağdan veri bekler; web sunucusu yüzlerce isteğe aynı anda yanıt verir. İş parçacığı bu ihtiyaca verilen cevaptır. Aynı adres alanını paylaşan ama her biri kendi program sayacı, yazmaçları ve yığınıyla yürüyen akışlar, süreç oluşturmanın maliyetine katlanmadan eş zamanlılık sağlar. Not, iş parçacığının tanımıyla ve neden gerektiğiyle başlar. Ardından çok çekirdekli işlemcilerde paralel programlamanın sınırlarını Amdahl yasasıyla ele alır. Kullanıcı ve çekirdek düzeyi iş parçacıklarını, N:1, 1:1 ve M:N eşleme modellerini karşılaştırır. Pthreads ve Java ile iş parçacığı oluşturma, havuzlar, fork-join, OpenMP ve GCD gibi örtük yaklaşımlar kodlarıyla anlatılır. Son kesim, iş parçacıklarının beraberinde getirdiği sorunlara ayrılmıştır: `fork()`, sinyaller, iptal, iş parçacığına yerel depolama, yeniden girilir kod ve yarış durumu.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum04` · **Kod (Java):** `github.com/sercankulcu/operating-systems-java/Bolum04/src` · **Laboratuvar:** `github.com/sercankulcu/operating-systems-labs/lab04`

ÖĞRENME ÇIKTILARI

Kavram
İş parçacığını süreçten ayırmak; hangi durumun bütün iş parçacıklarınca paylaşıldığını, hangisinin her iş parçacığına özel olduğunu söylemek.

Performans
Eş zamanlılık ile paralelliği ayırt etmek ve Amdahl yasasıyla belirli bir çekirdek sayısında elde edilebilecek hızlanmayı ve hızlanmanın üst sınırını hesaplamak.

Modeller
Kullanıcı ve çekirdek düzeyi iş parçacıklarını; N:1, 1:1 ve M:N eşleme modellerini geçiş maliyeti, bloke olma ve çok çekirdek kullanımı açısından karşılaştırmak.

Programlama
Pthreads ve Java ile iş parçacığı oluşturmak, argüman geçirmek, sonuç almak ve beklemek; havuz, fork-join, OpenMP ve GCD gibi örtük yaklaşımları tanımak.

Sorunlar
Yarış durumunu, yeniden girilemeyen kodu, iş parçacığına yerel depolamayı, iptali, sinyal teslimini ve çok iş parçacıklı süreçte `fork()` davranışını açıklamak.

Gerçek sistemler
Linux `clone()` çağrısını, Java 21 sanal iş parçacıklarını ve Go çalışma zamanını eşleme modelleri açısından yorumlamak.

4.1 Temel kavramlar

Süreç kavramı, işletim sisteminin programları birbirinden yalıtmak için kullandığı temel araçtır. Her sürecin kendi adres alanı vardır; bir süreç ötekinin belleğini göremez, bozamaz. Bu yalıtım güvenlik için değerlidir, ama aynı veri üzerinde birlikte çalışması gereken işler için bir engele dönüşür. İki süreç aynı veriyi paylaşacaksa paylaşımlı bellek bölgesi kurmak ya da ileti göndermek gerekir; ikisi de zahmetli ve yavaştır. İş parçacığı, yalıtımı süreç düzeyinde bırakıp tek bir adres alanının içinde birden çok yürütme akışına izin verir.

4.1.1 İş parçacığı nedir?

Bir programın çalışmasını iki ayrı açıdan düşünmek mümkündür. Birinci açı kaynaklardır: program bellekte bir yer kaplar, dosya açar, ağ bağlantısı kurar, alt süreç başlatır. İkinci açı yürütmedir: işlemci programın komutlarını sırayla işler, bir program sayacı bir sonraki komutu gösterir, yazmaçlar ara değerleri tutar, yığın fonksiyon çağrılarının izini sürer. Klasik süreç kavramı bu iki açıyı birleştirir: her sürecin hem kaynakları hem de tek bir yürütme akışı vardır. İş parçacığı kavramı bu ikisini birbirinden ayırır.

Tanım 4.1

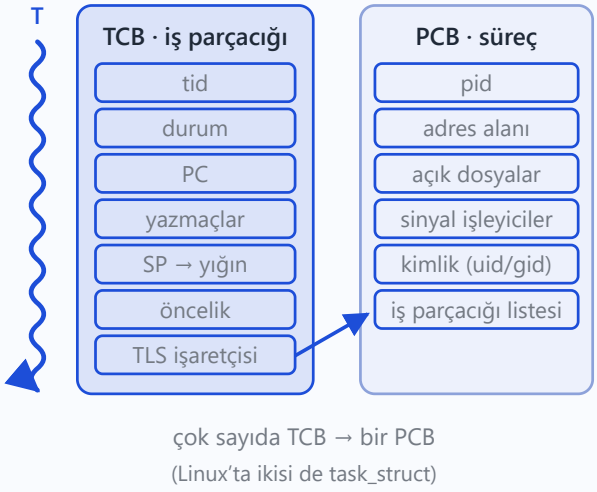
Süreç (*process*), yürütülmekte olan bir programdır. Adres alanı, açık dosyalar, çocuk süreçler, sinyal işleyicileri ve kimlik bilgisi gibi kaynakların sahibi süreçtir.

İş parçacığı (*thread*), süreç içindeki bir yürütme akışıdır. Kendi program sayacı (PC), yazmaç kümesi, yığını ve durumu vardır. İşlemci kullanımının ve çizelgelemenin temel birimi iş parçacığıdır.

Tanenbaum bu ayrımı tek cümleyle özetler: süreç kaynakları bir araya toplar, iş parçacığı ise işlemci üzerinde yürütülen varlıktır. Bir süreçte bir ya da daha çok iş parçacığı bulunur. Hepsi aynı adres alanını görür; birinin global bir değişkene yazdığı değeri öteki hemen okuyabilir. Buna karşılık her iş

parçacığı programın farklı bir yerinde olabilir, farklı fonksiyonlar çağırabilir, farklı yerel değişkenlerle çalışabilir.

İşletim sistemi her süreç için bir **süreç kontrol bloğu** (*process control block*, PCB) tutar. İş parçacıkları için de benzer bir yapı gerekir: **iş parçacığı kontrol bloğu** (*thread control block*, TCB). TCB'de iş parçacığının kimliği (tid), durumu, kaydedilmiş PC ve yazmaç değerleri, yığın işaretçisi (SP), önceliği ve iş parçacığına yerel depolama alanını gösteren bir işaretçi bulunur. Kaynak bilgileri TCB'de değil, iş parçacığının ait olduğu sürecin PCB'sindedir. Şekil 4.1 bu ilişkiyi gösterir.



Şekil 4.1: TCB ile PCB ilişkisi. Her iş parçacığının TCB'si tid, durum, PC, yazmaçlar, yığın işaretçisi, öncelik ve TLS işaretçisini tutar ve ait olduğu sürecin PCB'sini gösterir. PCB'de pid, adres alanı, açık dosyalar, sinyal işleyicileri, kimlik (uid/gid) ve iş parçacığı listesi bulunur. Birçok TCB tek bir PCB'yi gösterir. Linux'ta ikisi de aynı `task_struct` yapısıyla temsil edilir.

İş parçacıklarının eski bir adı **hafif süreçtir** (*lightweight process*, LWP). "Hafif" sözcüğü maliyete işaret eder. Yeni bir süreç oluşturmak, adres alanını kurmayı ya da kopyalamayı, sayfa tablolarını hazırlamayı, dosya tablosunu çoğaltmayı gerektirir. Yeni bir iş parçacığı için ise yalnız bir yığın ve bir TCB yeterlidir; adres alanı zaten vardır. Aynı nedenle iki iş parçacığı arasında geçiş de iki süreç arasındaki geçişten ucuzdur.

Sık yapılan hata

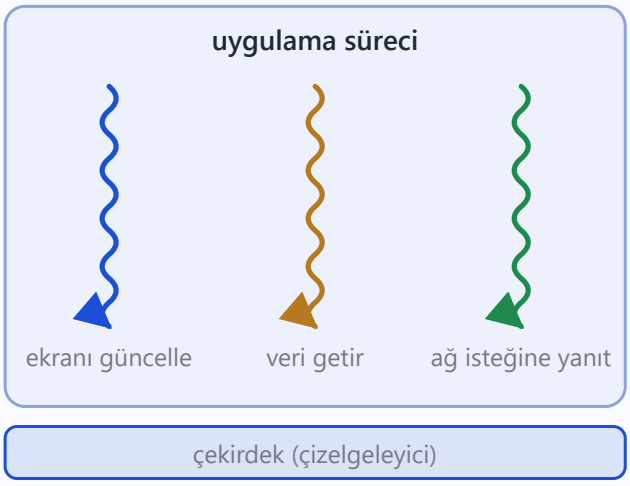
İş parçacığını "küçük bir süreç" sanıp her iş parçacığının kendi belleği olduğunu düşünmek. İş parçacıklarının ayrı bir adres alanı yoktur. Global değişkenler, heap'te ayrılan veriler ve kod bütün iş parçacıklarınca ortaktır. Ayrı olan yalnız yürütme durumudur: PC, yazmaçlar, yığın ve iş parçacığına yerel değişkenler.

4.1.2 Neden çok iş parçacığı?

Bugünkü uygulamaların çoğu çok iş parçacıklıdır. Bir web tarayıcısı, bir tümleşik geliştirme ortamı (*integrated development environment*, IDE), bir oyun motoru, bir veritabanı sunucusu ya da bir web sunucusu düzinelerce iş parçacığıyla çalışır. Bunun birkaç nedeni vardır.

- **Farklı görevler ayrı akışlarda yürür.** Bir uygulama aynı anda ekranı güncellemek, ağdan veri getirmek, ağ isteklerine yanıt vermek, yazım denetimi yapmak zorunda olabilir. Her görev kendi iş parçacığında, sanki programdaki tek iş oymuş gibi sıralı bir kodla yazılır.
- **Bekleyen görev ötekileri durdurmaz.** Disk ya da ağ beklenirken o iş parçacığı bloke olur, ama aynı süreçteki öteki iş parçacıkları ilerlemeye devam eder.
- **Ucuzdur.** Süreç oluşturmak pahalıdır; iş parçacığı oluşturmak çok daha ucuzdur.
- **Programı basitleştirir.** Her akış baştan sona okunabilen sıralı bir kod olarak kalır. Aynı işi tek akışla yapmak, programcıyı her görevin nerede kaldığını elle saklamaya zorlar.

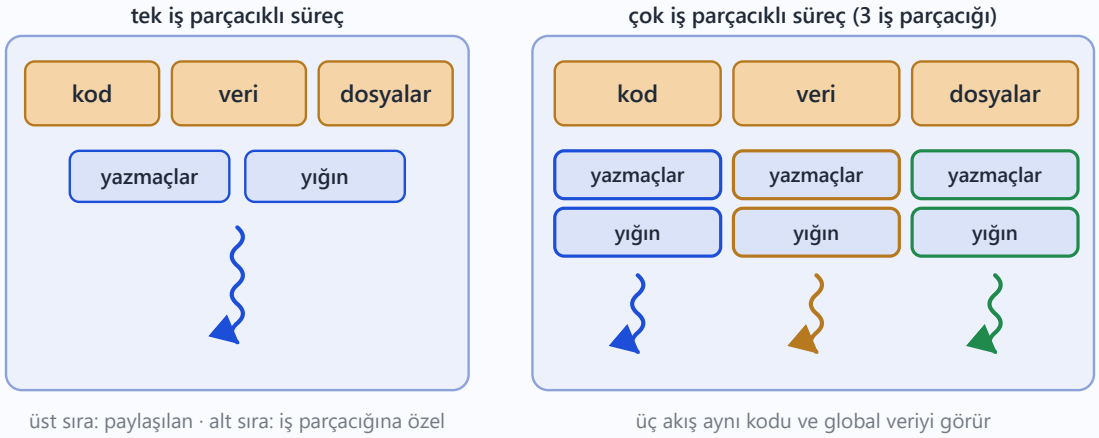
İşletim sistemi çekirdeği de çok iş parçacıklıdır. Linux'ta `ps` komutuyla bakıldığında köşeli parantez içinde adları görünen çekirdek iş parçacıkları buna örnektir: `kthreadd` öteki çekirdek iş parçacıklarını oluşturur, `kworker` iş parçacıkları ertelenmiş işleri yürütür, `ksoftirqd` yazılım kesmelerini işler. Şekil 4.2 bir uygulama sürecinin içindeki üç iş parçacığını ve bunları çizelgeleyen çekirdeği gösterir.



Şekil 4.2: Bir uygulama sürecinde üç iş parçacığı: biri ekranı günceller, biri veri getirir, biri ağ isteklerine yanıt verir. Üçünü de çekirdekteki çizelgeleyici işlemciye yerleştirir.

4.1.3 Tek ve çok iş parçacıklı süreç

Tek iş parçacıklı bir sürecin tek bir yürütme akışı vardır: bir PC, bir yazmaç kümesi ve bir yığın. Program bir anda yalnız bir yerde olabilir. Çok iş parçacıklı süreçte kod, global veri ve açık dosyalar yine birer tanedir; ama her akışın kendi yazmaçları ve kendi yığını vardır. Şekil 4.3 iki durumu yan yana gösterir.



Şekil 4.3: Solda tek iş parçacıklı, sağda üç iş parçacıklı süreç. Üst sıradaki turuncu kutular (kod, veri, dosyalar) paylaşılan kaynaklardır. Alt sıradaki yazmaçlar ve yığın her iş parçacığına özeldir. Dalgalı oklar yürütme akışlarını gösterir; üç akış aynı kodu ve aynı global veriyi görür.

Şekildeki ayrım, iş parçacıklarıyla ilgili hemen her sorunun kaynağıdır. Paylaşılan satırdaki her şey, bütün iş parçacıklarının aynı anda erişebildiği ortak veridir. Bir iş parçacığının global değişkene yazdığını öteki iş parçacıkları hemen görür. Bu, işbirliğini kolaylaştırır; ama iki iş parçacığı aynı değişkeni aynı anda değiştirmeye kalkarsa sonuç bozulabilir. Özel satırdaki öğeler ise her akışın kendi işini kaybetmeden sürdürebilmesini sağlar.

4.1.4 Paylaşılan ve özel öğeler

Hangi öğenin süreç başına, hangisinin iş parçacığı başına tutulduğunu bilmek, çok iş parçacıklı bir programı doğru yazmanın ilk adımıdır. Tablo 4.1 bu ayrımı listeler.

Süreç başına (bütün iş parçacıkları paylaşır)	**İş parçacığı başına (özel)**
Adres alanı: kod, global değişkenler, heap	Program sayacı (PC)
Açık dosyalar, soketler	Yazmaçlar
Çocuk süreçler	Yığın ve yığın işaretçisi
Bekleyen alarmlar (zamanlayıcılar)	Durum: çalışıyor, hazır, bekliyor, sonlandı
Sinyal işleyicileri (*signal handlers*)	İş parçacığı kimliği, sinyal maskesi
Kimlik ve hesap (*accounting*) bilgisi	İş parçacığına yerel depolama (TLS), `errno`

Tablo 4.1: Süreç başına ve iş parçacığı başına tutulan öğeler.

Tablonun sol sütunu, işletim sisteminin kaynakları süreç düzeyinde yönettiğini gösterir. Bir iş parçacığı bir dosya açarsa dosya tanımlayıcısı bütün süreç için geçerlidir; başka bir iş parçacığı aynı tanımlayıcıyla okuyup yazabilir, hatta dosyayı kapatabilir. Sinyal işleyicileri de süreç geneldir: bir iş parçacığının `SIGINT` için kurduğu işleyici, sinyal hangi iş parçacığına teslim edilirse edilsin çalışır.

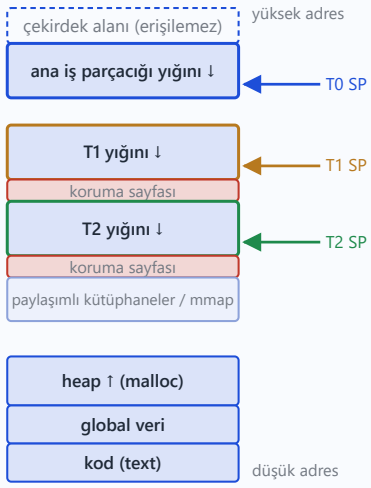
Sağ sütundaki öğeler yürütmenin kendisine aittir. İki iş parçacığı aynı fonksiyonu aynı anda çalıştırabilir; her birinin yerel değişkenleri kendi yığınında durduğu için birbirine karışmaz. `errno` gibi "global gibi görünen ama iş parçacığına özel" değişkenler ise iş parçacığına yerel depolama ile sağlanır (4.6.5).

> **Sık yapılan hata**
>
> "Yığın iş parçacığına özeldir, öyleyse başka iş parçacığı ona dokunamaz" diye düşünmek. Yığın özeldir ama **korunmaz**. Bütün yığınlar aynı adres alanındadır; hatalı bir işaretçi başka bir iş parçacığının yığınındaki veriyi okuyabilir ya da bozabilir. Donanım bunu engellemez, çünkü sayfa tablosu bütün iş parçacıkları için aynıdır.

Bu bölümün geri kalanı için özet bir kural yeterlidir: **paylaşılan veriye erişim senkronizasyon gerektirir**. Birden çok iş parçacığının yazdığı her global değişken, her heap nesnesi ve her açık dosya, bir kilitle ya da atomik işlemle korunmadıkça yarış durumuna açıktır (4.6.7).

4.1.5 Her iş parçacığının kendi yığını

Fonksiyon çağrıları ve yerel değişkenler her akış için ayrı tutulmak zorundadır. İş parçacığı A bir `f` fonksiyonunun içindeyken iş parçacığı B `g`'yi çağırabilir. İkisi aynı yığını kullansaydı B'nin çerçevesi A'nın çerçevesinin üstüne yazılır, A `f`'ten döndüğünde yanlış adrese atlardı. Bu yüzden her iş parçacığının kendi yığını vardır. Şekil 4.4 üç iş parçacıklı bir sürecin adres alanında yığınların nasıl yerleştiğini gösterir.



Şekil 4.4: Üç iş parçacıklı bir sürecin adres alanı. En üstte erişilemeyen çekirdek alanı, altında ana iş parçacığının yığını, sonra aralarında koruma sayfaları bulunan T1 ve T2 yığınları yer alır. Ortada paylaşımlı kütüphaneler ve `mmap` bölgesi, altta yukarı doğru büyüyen heap, global veri ve kod vardır. Her iş parçacığının SP'si kendi yığınını gösterir.

Linux'ta glibc kütüphanesi, yeni bir iş parçacığının yığınını `mmap` ile ayrılmış anonim bir bellek bölgesine yerleştirir. Varsayılan yığın boyutu genellikle `ulimit -s` değeri kadardır; çoğu dağıtımda bu değer 8 MiB'dir. Bu 8 MiB'nin tamamı hemen fiziksel belleğe yerleşmez: sanal bellek sayesinde yalnız dokunulan sayfalar gerçekten bellek tüketir.

Yığınlar aşağı doğru büyür ve iki yığın arasında başka bir şey yoksa taşan yığın komşusunun üstüne yazabilir. Bunu önlemek için her yığının ucuna bir **koruma sayfası** (*guard page*) konur. Koruma sayfası erişilemez olarak işaretlenmiştir; taşan bir yığın bu sayfaya dokunduğu anda sayfa hatası olur ve süreç `SIGSEGV` sinyaliyle sonlanır. Böylece hata sessizce komşu yığını bozmak yerine hemen fark edilir.

Yığın boyutu ayarlanabilir. Pthreads'te iş parçacığı oluşturulmadan önce `pthread_attr_setstacksize` ile istenen boyut verilir. Java'da `-Xss` seçeneği her platform iş parçacığının yığın boyutunu belirler. Derin özyineleme yapan bir iş parçacığı için yığını büyütmek, binlerce kısa ömürlü iş parçacığı için ise küçültmek gerekebilir.

Java 21 ile gelen **sanal iş parçacıkları** bu durumu değiştirir. Sanal iş parçacığının yığın çerçeveleri sabit büyüklükte bir bellek bölgesinde değil, heap'teki nesnelerde tutulur ve gerektiği kadar büyür. Bir sanal iş parçacığının birkaç yüz baytlık yığınla başlayabilmesi, milyonlarca sanal iş parçacığının aynı anda var olabilmesini sağlar (4.3.8).

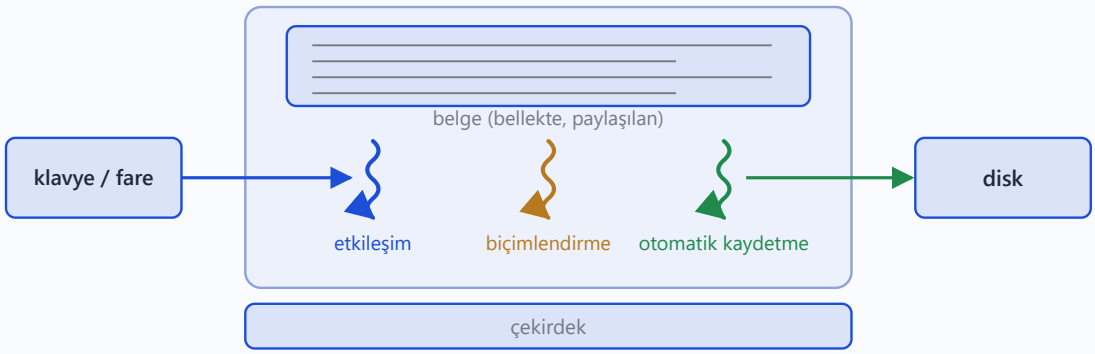
Örnek 4.1

Bir sunucu her bağlantı için bir platform iş parçacığı oluşturuyor; varsayılan yığın boyutu 8 MiB. 10.000 bağlantı için kaç bayt sanal adres alanı yalnız yığınlara ayrılır?

1. Bir yığın 8 MiB = 8×2^{20} bayttır.
2. 10.000 yığın: 10.000 × 8 MiB = 80.000 MiB ≈ 78,1 GiB.
3. 64 bitlik bir adres alanı bunu rahatça taşır ve dokunulmayan sayfalar fiziksel bellek tüketmez. Ama her yığının en az birkaç sayfası kullanılır; her iş parçacığının çekirdekte de bir TCB'si ve çekirdek yığını vardır. Bu yüzden on binlerce platform iş parçacığı, sanal iş parçacıklarına ya da olay güdümlü tasarıma göre belirgin biçimde daha çok kaynak harcar.

4.1.6 Örnek: üç iş parçacıklı kelime işlemci

İş parçacıklarının neden yararlı olduğunu göstermek için Tanenbaum'un klasik örneği bir kelime işlemcidir. Yüzlerce sayfalık bir kitap düzenlenirken kullanıcı birinci sayfada bir satırı siliyor. Bu satırın silinmesi, ardından gelen bütün sayfaların yeniden biçimlendirilmesini gerektirir. Aynı zamanda kullanıcı yazmaya devam etmek ister, program da belgeyi belirli aralıklarla diske kaydetmelidir. Şekil 4.5 bu üç işi üç iş parçacığına dağıtır.



Şekil 4.5: Üç iş parçacıklı kelime işlemci. Belge bellekte tek bir kopya olarak durur ve üç iş parçacığı onu paylaşır: etkileşim iş parçacığı klavye ve fare olaylarını işler, biçimlendirme iş parçacığı sayfaları yeniden düzenler, otomatik kaydetme iş parçacığı belgeyi diske yazar.

Etkileşim
: Tuş ve fare olaylarını hemen işler. Arka planda ne kadar iş olursa olsun arayüz donmaz.

Biçimlendirme
: Bir satır silinince yüzlerce sayfayı arka planda yeniden düzenler. Kullanıcı 600. sayfaya atladığında o sayfa büyük olasılıkla hazırdır.

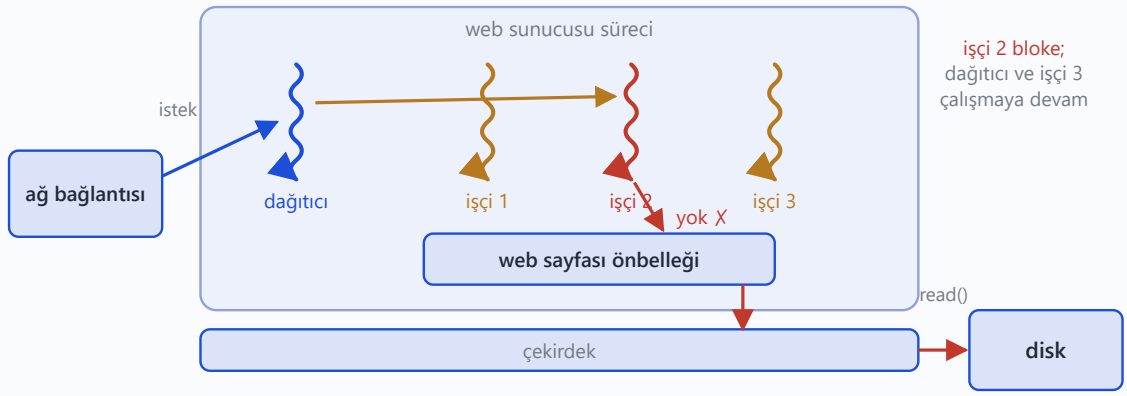
Otomatik kaydetme
: Belgeyi belirli aralıklarla diske yazar. Disk G/Ç'si beklenirken kullanıcıyı bekletmez.

Aynı tasarım üç ayrı süreçle kurulamazdı. Üç işin de **aynı belge** üzerinde çalışması gerekir; belge üç ayrı adres alanına kopyalansaydı her değişiklik ötekilere iletilmek zorunda kalırdı. Ortak adres alanı burada temel avantajdır. Tek iş parçacıklı bir tasarım ise iki kötü seçenek arasında kalırdı: ya biçimlendirme bitene kadar klavye girdisine yanıt verilmez, ya da biçimlendirme kodu her birkaç satırda bir durup klavyeyi yoklar. İkinci seçenek kodu okunmaz hâle getirir.

4.1.7 Çok iş parçacıklı web sunucusu

İş parçacıklarının en tanıdık kullanım yeri sunuculardır. Bir web sunucusu her saniye çok sayıda istek alır. İsteklerin bir kısmı sık erişilen sayfalar içindir ve bu sayfalar bellekteki bir **web sayfası önbelleğinde** hazır durur. Öteki istekler için sayfanın diskten okunması gerekir; bu da milisaniyeler sürer. Sunucu her isteği sırayla işlerse diski bekleyen bir istek, arkasındaki bütün istekleri bekletir.

Tanenbaum'un tasarımında sunucu sürecinde iki tür iş parçacığı vardır. **Görev dağıtıcı** (*dispatcher*) iş parçacığı ağdan gelen istekleri okur. Her istek için boşta bekleyen bir **işçi** (*worker*) iş parçacığı seçer, isteği ona devreder ve onu uyandırır. İşçi sayfayı önce önbellekte arar; bulamazsa diskten okur ve yanıtı gönderir. Şekil 4.6 bu düzeni, işçilerden birinin diskte bloke olduğu an için gösterir.



Şekil 4.6: Çok iş parçacıklı web sunucusu. Dağıtıcı ağdan gelen isteği işçi 2'ye devretmiştir. İşçi 2 sayfayı önbellekte bulamamış (X) ve diskten okumak için `read()` çağrısında bloke olmuştur. Bu sırada dağıtıcı ve işçi 3 çalışmaya devam eder.

Bu tasarımda dört an öne çıkar. İstek gelir ve dağıtıcı onu ağdan okur. Dağıtıcı boştaki bir işçiyi seçer, isteği ona verir ve uyandırır. İşçi 1 sayfayı önbellekte bulursa yanıtı hemen gönderir; diske gitmez. Yeni bir istek işçi 2'ye gider ve sayfa önbellekte yoksa işçi 2 diskte bloke olur. Önemli olan son durumdur: işçi 2'nin bloke olması yalnız işçi 2'yi durdurur. Dağıtıcı yeni istekleri kabul etmeye, işçi 3 de başka bir isteği işlemeye devam eder.

İki iş parçacığının kodu son derece sadedir. Dağıtıcı sonsuz bir döngüde bir sonraki isteği okur ve bir işçiye verir:

```
while (TRUE) {
  get_next_request(&buf);  /* ağdan oku */
  handoff_work(&buf);      /* işçiye ver */
}
```

`get_next_request(&buf)` ağdan bir istek gelene kadar bloke olur ve isteği `buf` ara belleğine yazar. `handoff_work(&buf)` isteği boştaki bir işçiye aktarır ve o işçiyi uyandırır. Dağıtıcı bundan sonra işin ne olacağıyla ilgilenmez; hemen döngünün başına döner. İşçinin kodu da benzer bir döngüdür:

```
while (TRUE) {
  wait_for_work(&buf);
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page))
    read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

İşçi `wait_for_work` içinde, dağıtıcı ona bir iş verene kadar bloke olur. İş gelince sayfayı önbellekte arar (`look_for_page_in_cache`). Sayfa önbellekte yoksa `read_page_from_disk` ile diskten okur; bu çağrı disk okuması bitene kadar bloke eder. Son olarak `return_page` sayfayı istemciye gönderir ve işçi yeni bir iş beklemek üzere döngünün başına döner.

İki kod da sıralıdır ve okunaklıdır. Bloke eden `read` çağrısı yalnız onu çağıran işçiyi durdurur. Tek iş parçacıklı bir sunucuda durum farklıdır: önbellekte olmayan bir sayfa için bütün süreç bloke olur, disk beklenirken işlemci boşta kalır ve o sırada gelen istekler kuyrukta bekler. Çok iş parçacıklı sunucuda iş bir işçiye aktarılır, sunucu yeni istekleri kabul etmeyi sürdürür.

4.1.8 Kod izleme: dağıtıcı ve işçiler

Tasarımın nasıl çalıştığını somut bir senaryoyla izleyelim. Sunucu sürecinde üç iş parçacığı var: dağıtıcı ve iki işçi, W1 ile W2. Başlangıçta dağıtıcı ağı dinliyor, iki işçi de `wait_for_work` içinde bloke. Önbellekte yalnız `a.html` sayfası var. Önce `b.html` , ardından `a.html` için birer istek geliyor. İzlenen kod, iki döngünün bir araya getirilmiş biçimidir:

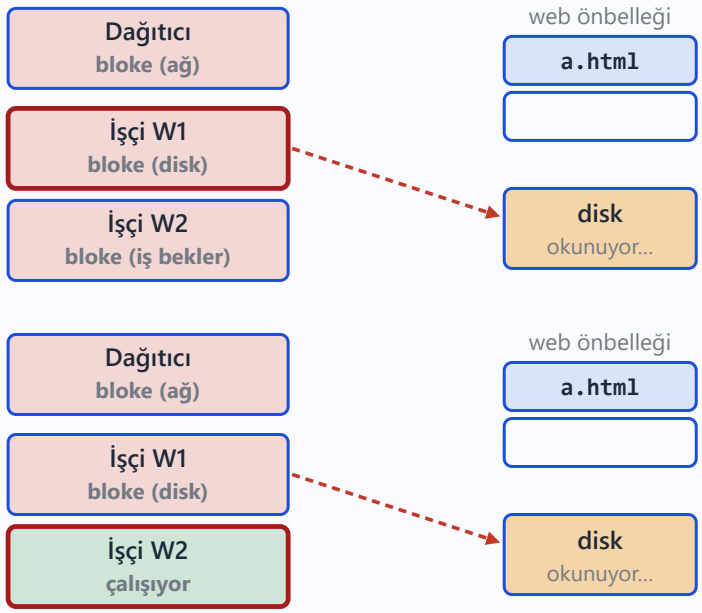
```
/* dağıtıcı iş parçacığı */
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
/* işçi iş parçacığı */
while (TRUE) {
  wait_for_work(&buf);
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page))
    read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

Adım	Olay	Dağıtıcı	W1	W2	Disk	Önbellek
0	başlangıç	çalışıyor	bloke (iş bekler)	bloke (iş bekler)	boşta	a.html
1	istek 1 gelir: `b.html`	çalışıyor	bloke (iş bekler)	bloke (iş bekler)	boşta	a.html
2	`handoff_work` : istek 1 → W1	çalışıyor	hazır	bloke (iş bekler)	boşta	a.html
3	W1 önbellekte arar: yok	bloke (ağ)	çalışıyor	bloke (iş bekler)	boşta	a.html
4	W1 `read_page_from_disk`	bloke (ağ)	bloke (disk)	bloke (iş bekler)	okunuyor	a.html
5	istek 2 gelir: `a.html`	çalışıyor	bloke (disk)	bloke (iş bekler)	okunuyor	a.html

Adım	Olay	Dağıtıcı	W1	W2	Disk	Önbellek
6	W1 meşgul: istek 2 → W2	çalışıyor	bloke (disk)	hazır	okunuyor	a.html
7	W2 `a.html`'i önbellekte bulur	bloke (ağ)	bloke (disk)	çalışıyor	okunuyor	a.html
8	W2 `return_page`: istek 2 yanıtlandı	bloke (ağ)	bloke (disk)	çalışıyor	okunuyor	a.html
9	disk biter, W1 uyanır, istek 1 yanıtlanır	bloke (ağ)	çalışıyor	bloke (iş bekler)	boşta	a.html, b.html
10	iki işçi yeniden iş bekler	bloke (ağ)	bloke (iş bekler)	bloke (iş bekler)	boşta	a.html, b.html

Tablo 4.2: Dağıtıcı ve iki işçinin izlemesi. Her satır bir adımdan sonraki durumu gösterir.

İzlemedeki kilit adım 4. adımdır. W1 diskten okumak için bloke olur, ama yalnız W1 durur; aynı süreçteki öteki iş parçacıkları çalışmaya devam eder. Disk beklenirken dağıtıcı ikinci isteği alır ve W1 meşgul olduğu için onu W2'ye verir. W2 sayfayı önbellekte bulur ve yanıtlar. Sonuçta **istek 2, istek 1'den önce yanıtlanır**. Disk okuması bitince W1 uyanır, sayfayı önbelleğe de koyar ve birinci istemciye gönderir. Şekil 4.7 4. ve 8. adımları yan yana gösterir.



Şekil 4.7: İzlemenin iki anı. (a) 4. adım: W1 diskte bloke, disk "okunuyor", dağıtıcı ağda yeni istek bekliyor. (b) 8. adım: W1 hâlâ diskte bloke iken W2 önbellekteki `a.html`'i döndürüyor.

Tek iş parçacıklı bir sunucuda istek 2, disk okuması bitene kadar bekleyecekti; oysa kendi sayfası bellekte hazırdı. Çok iş parçacıklı sunucu, bekleme ile hesaplamayı örtüştürerek bu boşa geçen süreyi kullanır. Her kod parçası da yine sıralı ve okunaklı kalır.

Sık yapılan hata

İstek 2'nin istek 1'den önce yanıtlanmasını bir hata saymak. Çok iş parçacıklı bir sunucuda istekler geliş sırasıyla değil, hazır oldukları sırayla tamamlanır. Her istek kendi bağlantısına yanıt verdiği için sıranın değişmesi doğruluğu bozmaz; tersine, ortalama yanıt süresini kısaltır.

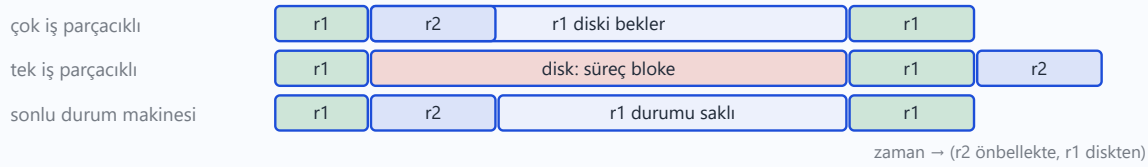
4.1.9 Sunucu tasarımının üç yolu

Bir sunucunun bloke eden işlemlerle nasıl başa çıkacağı konusunda üç temel seçenek vardır. Tanenbaum ve Bos bunları Tablo 4.3'teki gibi karşılaştırır.

Model	Paralellik / örtüşme	Sistem çağrıları	Programlama
Çok iş parçacıklı süreç	var	bloke eden	sıralı kod, kolay
Tek iş parçacıklı süreç	yok	bloke eden	kolay ama yavaş
Sonlu durum makinesi (olay güdümlü)	var (G/Ç örtüşür)	bloke etmeyen + bildirim / kesme	zor: her isteğin durumu elle saklanır

Tablo 4.3: Sunucu tasarımının üç yolu (Tanenbaum ve Bos, *Modern Operating Systems*, Şekil 2-10'daki karşılaştırma).

Üçüncü model biraz açıklama ister. Olay güdümlü sunucuda tek bir iş parçacığı vardır, ama bu iş parçacığı hiçbir zaman bloke olmaz. Diskten okuma gerektiğinde bloke etmeyen bir çağrı yapar ve okuma bitince bir bildirim alacağını bilerek başka işlere geçer. Bu sırada yarım kalan isteğin nerede kaldığını, hangi sayfayı beklediğini, kime yanıt verileceğini bir tabloya kendisi yazar. Bildirim gelince tablodan o isteği bulur ve kaldığı yerden sürdürür. Program böylece bir **sonlu durum makinesine** dönüşür: her istek bir durumdan ötekine geçen küçük bir makinedir. G/Ç ile hesaplama örtüşür, ama iş parçacığı modelinin sağladığı sıralı kod düzeni kaybolur.



Şekil 4.8: İki istek için üç modelin zaman çizelgesi; r1 diskten, r2 önbellekten karşılanıyor. Çok iş parçacıklı sunucuda r1 diski beklerken r2 işlenir. Tek iş parçacıklı sunucuda disk beklenirken süreç bloke olur ve r2 en sona kalır. Sonlu durum makinesinde r1'in durumu saklanır ve r2 araya girer.

Bugün olay güdümlü model yaygındır: nginx, Node.js ve Redis bu modeli kullanır. Linux'ta `epoll` ve `io_uring`, BSD ve macOS'ta `kqueue` çok sayıda bağlantıyı tek iş parçacığıyla izlemeye yarayan arayüzlerdir. Çoğu gerçek sunucu iki yaklaşımı birleştirir: her çekirdek için bir olay döngüsü çalışır, bloke edebilecek işler (ör. disk erişimi, sıkıştırma) ise bir iş parçacığı havuzuna gönderilir.

Örnek 4.2

Tek işlemcili bir sunucuda isteklerin 3/4'ü önbellekten karşılanıyor ve 10 ms işlemci zamanı harcıyor. Kalan 1/4'ünde 10 ms işlemci zamanına ek olarak 50 ms disk beklemesi var. Disk aynı anda birçok isteğe hizmet verebiliyor. Sunucu tek iş parçacıklıysa ve çok iş parçacıklıysa saniyede kaç isteğe yanıt verir?

1. **Tek iş parçacığı.** Disk beklenirken süreç bloke olur, işlemci boşta kalır. Bir isteğin ortalama süresi 3/4 × 10 + 1/4 × (10 + 50) = 7,5 + 15 = 22,5 ms'dir. Saniyede 1000 / 22,5 ≈ 44,4 istek yanıtlanır.
2. **Çok iş parçacığı.** Diski bekleyen işçi bloke olurken başka bir işçi işlemciyi kullanır; disk beklemesi hesaplamayla örtüşür. Darboğaz yalnız işlemcidir ve her istek 10 ms işlemci zamanı ister. Saniyede 1000 / 10 = 100 istek yanıtlanır.
3. **Yorum.** Tek işlemci olduğu için kazanç paralellikten gelmez; bloke eden çağrıların hesaplamayla örtüşmesinden gelir. Aynı örtüşme olay güdümlü tek iş parçacığıyla da sağlanabilir, ama kod karmaşıklaşır.

4.1.10 Çok iş parçacığının faydaları

Buraya kadarki örnekler çok iş parçacıklı programlamanın dört temel faydasını ortaya koyar.

Duyarlılık (*responsiveness*)
: Programın bir kısmı bloke olsa ya da uzun bir hesap yapsa bile program yanıt vermeye devam eder. Kullanıcı arayüzlerinde bu özellik belirleyicidir: düğmeye basıldığında başlayan uzun bir işlem ayrı bir iş parçacığında yürürse pencere donmaz.

Kaynak paylaşımı
: İş parçacıkları sürecin belleğini ve dosyalarını kendiliğinden paylaşır. Süreçler arasında aynı paylaşım için paylaşımlı bellek bölgesi ya da ileti geçirme düzeneği kurmak gerekir; iş parçacıklarında ek bir şey yapılmaz.

Ekonomi
: İş parçacığı oluşturmak ve iş parçacıkları arasında geçiş yapmak süreçlere göre ucuzdur: adres alanı kopyalanmaz, sayfa tabloları değişmez.

Ölçeklenebilirlik
: Çok iş parçacıklı tek bir süreç birden çok çekirdekte aynı anda çalışabilir. Tek iş parçacıklı bir süreç, makinede kaç çekirdek olursa olsun yalnız birini kullanır.

Ders kitaplarında sık aktarılan “iş parçacığı oluşturmak süreç oluşturmaktan 30 kat, iş parçacıkları arasında geçiş 5 kat hızlıdır” rakamları eski bir Solaris ölçümüne dayanır. Bu oranlar her sistemde geçerli değildir. Linux'ta süreç de iş parçacığı da aynı `clone()` çağrısıyla oluşturulur (4.6.9), bu yüzden oluşturma maliyetleri arasındaki fark daha küçüktür. Asıl kazanç, bellek eşlemesinin kopyalanmaması ve geçiş sırasında adres çeviri önbelleğinin korunmasıdır.

4.1.11 Bağlam değiştirme maliyeti

İşlemcinin bir yürütme akışından ötekine geçmesine **bağlam değiştirme** (*context switch*) denir. Geçişin iki türü arasındaki fark, iş parçacıklarının neden daha ucuz olduğunu açıklar. Şekil 4.9 iki türü karşılaştırır.



Şekil 4.9: Bağlam değiştirmenin iki türü. Aynı süreçteki iki iş parçacığı arasında yalnız yazmaçlar, PC ve SP kaydedilip yüklenir; adres alanı aynı kaldığı için TLB ve önbellek sıcaktır. Süreçler arası geçişte buna ek olarak sayfa tablosu tabanı değişir (x86'da CR3 yazmacı), PCID/ASID desteği yoksa TLB girdileri geçersiz olur ve önbellekler soğur.

Her iki geçişte de çekirdek, çalışan akışın yazmaçlarını, PC'sini ve SP'sini kaydeder ve yeni akışınkileri yükler. Süreçler arası geçişte buna bir adım daha eklenir: sayfa tablosunun taban adresi değiştirilir. x86 işlemcilerde bu adres CR3 yazmacında tutulur. Yeni adres alanına geçildiğinde, sanal adresleri fiziksel adreslere çeviren kayıtları saklayan **adres çeviri önbelleğindeki** (*translation lookaside buffer*, TLB) eski girdiler artık geçersizdir. İşlemci girdileri adres alanı kimliğiyle (PCID ya da ASID) etiketlemiyorsa TLB boşaltılır. Ardından yeni sürecin ilk bellek erişimleri TLB'yi ve veri önbelleklerini yeniden doldurana kadar yavaş çalışır.

Maliyetin iki bileşeni vardır. **Doğrudan maliyet**, yazmaçları kaydedip yüklemenin ve çizelgeleyiciyi çalıştırmanın süresidir; bugünkü işlemcilerde mikrosaniyeler mertebesindedir. **Dolaylı maliyet** ise geçişten sonra soğuk önbellekler ve TLB ıskaları yüzünden yavaşlayan bellek erişimlerinden gelir. Dolaylı maliyet çoğu zaman doğrudan maliyetten büyüktür. Aynı süreçteki iş parçacıkları arasında geçişte adres alanı değişmediği için dolaylı maliyetin büyük kısmı ortadan kalkar.

Kullanıcı düzeyi iş parçacıklarında (4.3.2) geçiş daha da ucuzdur. Geçişi çekirdek değil, kullanıcı alanındaki bir kütüphane fonksiyonu yapar. Tuzak yoktur, çekirdeğe giriş yoktur; geçiş birkaç komutluk sıradan bir fonksiyon çağrısına iner.

Sık yapılan hata

Bağlam değiştirmenin maliyetini yalnız yazmaç kaydetme süresi sanmak. Yazmaçları kaydetmek birkaç yüz nanosaniye sürebilir, ama ardından gelen önbellek ve TLB ıskaları programı çok daha uzun süre yavaşlatabilir. Gereğinden fazla iş parçacığı çalıştıran bir program, sık ge-

çişler yüzünden işlemciyi yararlı iş yerine önbelleği yeniden doldurmaya harcar.

4.2 Çok çekirdekli programlama

Yaklaşık 2005'e kadar işlemciler her yıl daha yüksek saat hızlarıyla piyasaya çıktı ve tek iş parçacıklı programlar kendiliğinden hızlandı. Güç tüketimi ve ısı bu artışı durdurunca üreticiler aynı yonga üzerine birden çok çekirdek koymaya yöneldi. Bugün bir dizüstü bilgisayarda 8–16, bir sunucuda 64–128 çekirdek bulunur. Bu çekirdeklerden yararlanmanın yolu, programı aynı anda çalışabilecek parçalara bölmektir. İş parçacıkları bunun en doğrudan aracıdır.

4.2.1 Eş zamanlılık ve paralellik

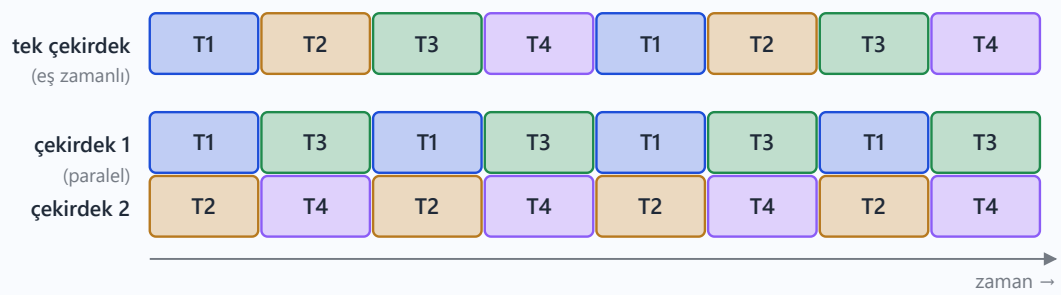
Gündelik dilde birbirinin yerine kullanılan iki sözcük, işletim sistemlerinde farklı anlamlar taşır.

Tanım 4.2

Eş zamanlılık (*concurrency*): birden çok görevin aynı zaman aralığında ilerlemesidir. Görevlerin aynı anda çalışması gerekmez; tek bir çekirdekte iç içe geçerek de ilerleyebilirler.

Paralellik (*parallelism*): birden çok görevin aynı anda yürütülmesidir. En az iki çekirdek ya da işlem birimi gerekir.

Tek çekirdekli bir makinede dört görev olduğunu düşünelim. Çizelgeleyici her birine kısa bir zaman dilimi verir: önce T1, sonra T2, T3, T4 ve yeniden T1. Dilimler kısa olduğu için insan gözüne dört görev birlikte ilerliyormuş gibi görünür. Gerçekte herhangi bir anda yalnız biri çalışır. Bu, eş zamanlılıktır ama paralellik değildir. Aynı dört görev iki çekirdekli bir makinede çalışırsa T1 ile T3 birinci çekirdeği, T2 ile T4 ikinci çekirdeği paylaşır. Artık her an iki görev gerçekten aynı anda çalışır: bu, paralel yürütmedir. Şekil 4.10 iki durumu karşılaştırır.



Şekil 4.10: Üstte tek çekirdekte zaman paylaşımı: T1–T4 sırayla iç içe çalışır; her an yalnız bir görev çalışır (eş zamanlı). Altta iki çekirdek: T1/T3 birinci, T2/T4 ikinci çekirdekte aynı anda çalışır (paralel).

Paralellik eş zamanlılığı gerektirir: aynı anda çalışan iki görev, aynı zaman aralığında da ilerliyor demektir. Tersi doğru değildir: eş zamanlı bir program tek çekirdekte paralellik olmadan çalışabilir. Aynı program iki farklı donanımda farklı davranır. Tek çekirdekte iş parçacıkları yalnız iç içe geçer; çok çekirdekte gerçekten aynı anda çalışır ve paylaşılan veriye gerçekten aynı anda erişebilir.

> **Sık yapılan hata**
>
> Tek çekirdekte iş parçacığı kullanmanın anlamsız olduğunu düşünmek. Paralellik olmasa da eş zamanlılık değerlidir: bloke olan bir iş parçacığı beklerken öteki çalışır (Örnek 4.2'deki sunucu tek işlemcide iki kattan fazla hızlanmıştı). Benzer biçimde, tek çekirdekte yarış durumu olmayacağını sanmak da yanlıştır: zaman dilimi bir oku-artır-yaz dizisinin ortasında bitebilir.

4.2.2 Çok çekirdekli programlamanın zorlukları

Çekirdek sayısının artması programları kendiliğinden hızlandırmaz. Programcının ya da derleyicinin işi çekirdeklere bölmesi gerekir. Silberschatz bu işin zorluklarını beş başlıkta toplar; bunlara bellek etkilerini de eklemek gerekir.

Görevleri belirleme
: Uygulamada birbirinden bağımsız, paralel yürüyebilecek etkinlikleri bulmak. Bazı algoritmalar doğal olarak bölünür (bir dizinin toplamı), bazıları adım adım ilerler ve bölünmez.

Denge (*balance*)
: Görevler benzer büyüklükte olmalıdır. Dört çekirdekten biri ötekilerin dört katı iş alırsa toplam süreyi o çekirdek belirler. Çok küçük bir görev için ayrı bir çekirdek ayırmak da kazandırmaz; görevi başlatma maliyeti kazancı aşar.

Veri bölme
: Görevlerin eriştiği veri de çekirdeklere bölünmelidir. Bütün çekirdekler aynı veriye erişirse paralellik çekişmeye dönüşür.

Veri bağımlılığı
: Bir görev ötekinin sonucunu bekliyorsa erişim senkronize edilmelidir. Bağımlılıklar paralel çalışabilecek kısmı küçültür.

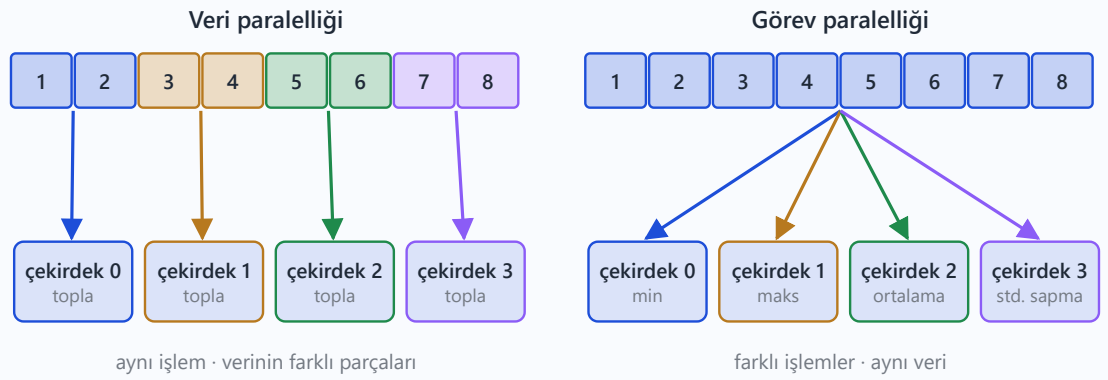
Test ve hata ayıklama
: Yürütme sırası her çalıştırmada değişebilir. Bir hata yüz çalıştırmada bir kez ortaya çıkabilir ve hata ayıklayıcı altında hiç görünmeyebilir.

Bellek etkileri
: Farklı değişkenler aynı önbellek satırındaysa ve iki çekirdek bunlara yazıyorsa, önbellek tutarlılık düzeneği satırı çekirdekler arasında sürekli taşır. Mantıksal olarak hiçbir şey paylaşılmadığı hâlde çekirdekler birbirini yavaşlatır. Bu duruma **yanlış paylaşım** (*false sharing*) denir.

Yarışları bulmaya yardım eden araçlar vardır: GCC ve Clang'da `-fsanitize=thread` seçeneğiyle etkinleşen ThreadSanitizer, Valgrind'in Helgrind aracı, Java Flight Recorder. Ama hiçbir araç bütün yarışları yakalamaz; araçlar yalnız o çalıştırmada gerçekleşen erişimleri inceler. Paylaşımı tasarım aşamasında azaltmak en etkili yöntemdir.

4.2.3 Veri ve görev paralelliği

İşi çekirdeklere bölmenin iki temel yolu vardır. **Veri paralelliğinde** (*data parallelism*) veri parçalara bölünür ve her çekirdek kendi parçasında aynı işlemi yapar. **Görev paralelliğinde** (*task parallelism*) iş farklı görevlere bölünür ve her çekirdek farklı bir işlem yapar; görevler aynı veriyi ya da farklı verileri kullanabilir. Şekil 4.11 iki yolu aynı sekiz öğeli dizi üzerinde gösterir.



Şekil 4.11: Solda veri paralelliği: 1–8 dizisi dört parçaya bölünür ve dört çekirdek kendi parçasında aynı işlemi (topla) yapar. Sağda görev paralelliği: dört çekirdek aynı dizinin tamamında farklı işlemler (min, maks, ortalama, standart sapma) yapar.

N öğeli bir diziyi dört çekirdekle toplamak veri paralelliğinin tipik örneğidir: her çekirdek N/4 öğe toplar, sonda dört kısmi toplam birleştirilir. Grafik işlemciler bu modelin uç örneğidir; binlerce basit çekirdek aynı komutu farklı piksellere ya da matris öğelerine uygular. Görev paralelliğine örnek olarak bir dizinin istatistiklerini hesaplayan bir program verilebilir: bir çekirdek en küçüğü, biri en büyüğü, biri ortalamayı, biri standart sapmayı bulur. Gerçek uygulamalar çoğunlukla ikisini birlikte kullanır.

4.2.4 Amdahl yasası

Bir programı çekirdek sayısını artırarak ne kadar hızlandırabiliriz? Hemen her programın paralelleştirilemeyen bir kısmı vardır: girdiyi okumak, sonuçları birleştirmek, bir kilidin arkasında sırayla yapılan işler. Bu kısım ne kadar çekirdek eklenirse eklensin aynı sürede çalışır. Gene Amdahl'ın 1967'de dile getirdiği gözlem, bu seri kısmın hızlanmaya kesin bir üst sınır koyduğudur.

Programın tek çekirdekteki çalışma süresini 1 birim alalım. Bu sürenin S kadarı seridir, $1 - S$ kadarı paralelleştirilebilir. N çekirdekte seri kısım yine S sürer; paralel kısım ise N çekirdeğe eşit bölünür ve $(1 - S) / N$ sürer. **Hızlanma** (*speedup*), eski sürenin yeni süreye oranıdır.

Teorem 4.1 (Amdahl yasası)

Seri oranı S olan bir program N çekirdekte en fazla

$$\text{hızlanma} \le 1 / (S + (1 - S) / N)$$

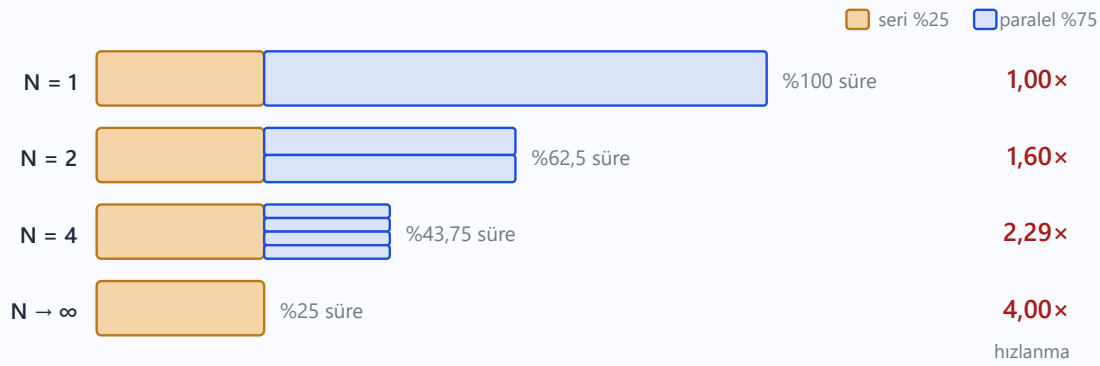
kat hızlanır. $N \to \infty$ iken paralel terim sıfıra gider ve hızlanmanın üst sınırı $1 / S$ olur.

Eşitsizlik işareti, formülün iyimser bir sınır olduğunu söyler. Formül paralel kısmın kusursuz bölündüğünü, iş parçacıkları arasında hiçbir iletişim ve senkronizasyon maliyeti olmadığını varsayar. Gerçek hızlanma bu sınırın altında kalır. Bir başka ölçü de **verimdir**: hızlanmanın çekirdek sayısına oranı, yani her çekirdeğin ne kadar iyi kullanıldığı.

Örnek 4.3

Bir programın çalışma süresinin %25'i seridir ($S = 0{,}25$). 1, 2 ve 4 çekirdekte süre ve hızlanma nedir? Hızlanmanın üst sınırı kaçtır?

1. **N = 1:** süre 0,25 + 0,75 / 1 = 1, yani %100. Hızlanma 1×.
2. **N = 2:** süre 0,25 + 0,75 / 2 = 0,25 + 0,375 = 0,625, yani %62,5. Hızlanma 1 / 0,625 = 1,60×. Çekirdek sayısı iki katına çıktı, hızlanma 2× değil.
3. **N = 4:** süre 0,25 + 0,75 / 4 = 0,25 + 0,1875 = 0,4375, yani %43,75. Hızlanma 1 / 0,4375 ≈ 2,29×. Verim 2,29 / 4 ≈ %57.
4. **N → ∞:** paralel kısım sıfıra yaklaşır, seri %25 kalır. Hızlanma en fazla 1 / 0,25 = 4×.



Şekil 4.12: Örnek 4.3'ün süre çubukları. Turuncu bölüm seri %25, mavi bölüm paralel %75'tir. N = 2'de paralel kısım ikiye, N = 4'te dörde bölünür; seri kısım hiç kısalmaz. $N \to \infty$ iken yalnız seri kısım kalır ve hızlanma 4×'te durur.

Şekil 4.12 yasanın özünü görsel olarak verir. Çekirdek eklendikçe yalnız mavi bölüm kısalır. Bir noktadan sonra çubuğun neredeyse tamamı turuncu bölümden oluşur ve yeni çekirdekler süreyi pek değiştirmez. Küçük görünen bir seri kısım bile ölçeklenmeyi belirgin biçimde sınırlar.

Örnek 4.4

Sekiz çekirdekli bir makinede bir hesaplamanın en az 6 kat hızlanması isteniyor. Seri kısım en fazla ne kadar olabilir?

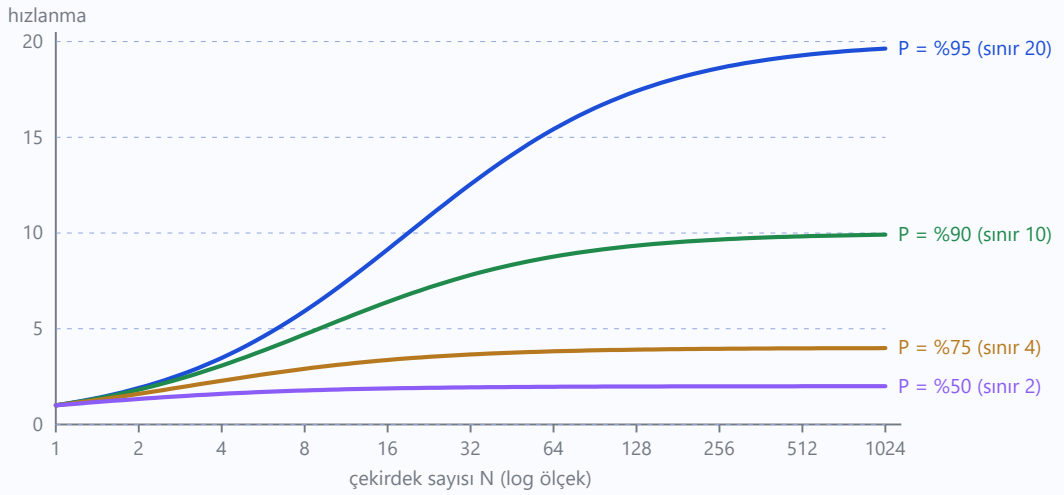
1. Hızlanma formülünü eşitliğe çevirelim: 6 = 1 / (S + (1 − S) / 8). Buradan S + (1 − S) / 8 = 1/6 olur.
2. İki tarafı 8 ile çarpalım: 8S + 1 − S = 8/6, yani 7S = 8/6 − 1 = 1/3.
3. S = 1/21 ≈ 0,0476. Seri kısım en fazla yaklaşık %4,8 olabilir; kodun en az %95,2'si paralel olmalıdır.
4. Sağlama: 1 / (1/21 + (20/21) / 8) = 1 / (1/21 + 5/42) = 1 / (7/42) = 6.

Sık yapılan hata

Seri oran ile paralel oranı karıştırmak. Formüldeki *S* seri (paralelleştirilemeyen) orandır. Soru “programın %80'i paralelleştirilebiliyor” diyorsa *S = 0,2*'dir, 0,8 değil. Bir başka yaygın hata da hızlanmayı süreyle karıştırmaktır: 0,625 süre, 1,60 ise hızlanmadır.

4.2.5 Amdahl eğrileri ve Gustafson yasası

Amdahl yasasının sonuçları farklı seri oranlar için çizildiğinde daha da çarpıcıdır. Şekil 4.13 paralel oranı %50, %75, %90 ve %95 olan dört program için hızlanmayı çekirdek sayısının logaritmik ölçeğinde gösterir.



Şekil 4.13: Amdahl eğrileri. Yatay eksen çekirdek sayısıdır (1'den 1024'e, logaritmik ölçek). Paralel oran P = %95, %90, %75 ve %50 için hızlanma sırasıyla 20, 10, 4 ve 2'ye yaklaşır ve orada düzleşir.

Seri oran S	N = 2	N = 4	N = 8	N = 16	N = 64	N = 1024	Sınır 1/S
0,50	1,33	1,60	1,78	1,88	1,97	2,00	2
0,25	1,60	2,29	2,91	3,37	3,82	3,99	4
0,10	1,82	3,08	4,71	6,40	8,77	9,91	10
0,05	1,90	3,48	5,93	9,14	15,42	19,64	20

Tablo 4.4: Farklı seri oranlar için hızlanma (Python ile hesaplandı, iki ondalığa yuvarlandı).

Tablodan iki sonuç çıkar. Birincisi, sınır seri orana çok duyarlıdır: seri kısmı %10'dan %5'e indirmek, sınırı 10'dan 20'ye çıkarır. İkincisi, çekirdek eklemenin getirisi hızla azalır. Programın %95'i paralel olsa bile 1024 çekirdekte hızlanma yalnız 19,6× olur; çekirdeklerin %98'inden fazlası boşa çalışıyor demektir. Gerçekte senkronizasyon ve iletişim yükü yüzünden hızlanma bu değerlerin de altında kalır.

Amdahl yasası problem boyutunun sabit kaldığını varsayar: aynı iş daha çok çekirdekle yapılır. John Gustafson 1988'de farklı bir soruyu sordu: çekirdek sayısı artınca genellikle problem de büyür. Hava tahmini yapan biri 1024 çekirdeğe kavuştuğunda aynı tahmini daha hızlı yapmakla yetinmez; daha ince bir ızgarayla daha ayrıntılı bir tahmin yapar. Bu durumda paralel kısım çekirdeklerle birlikte büyür, seri kısım ise aşağı yukarı sabit kalır. **Gustafson yasası**, bu varsayımla **ölçeklenmiş hızlanmanın**

> ölçeklenmiş hızlanma = S + (1 − S) · N

olduğunu söyler. Burada *S*, büyütülmüş problemin N çekirdekteki çalışma süresinin seri payıdır. Örneğin *S = 0,1* ve N = 16 için ölçeklenmiş hızlanma 0,1 + 0,9 · 16 = 14,5 olur; Amdahl yasası aynı değerlerle 6,4 verir. İki yasa çelişmez; farklı soruları yanıtlar. Amdahl, “bu işi ne kadar hızlı bitirebilirim?” sorusunu; Gustafson, “aynı sürede ne kadar büyük bir iş yapabilirim?” sorusunu cevaplar.

4.2.6 Kod izleme: Amdahl hızlanması

Aşağıdaki kısa C programı, seri oranı %10 olan bir program için 1, 2, 4, 8 ve 16 çekirdekte hızlanmayı yazdırır ve sonda üst sınırı verir:

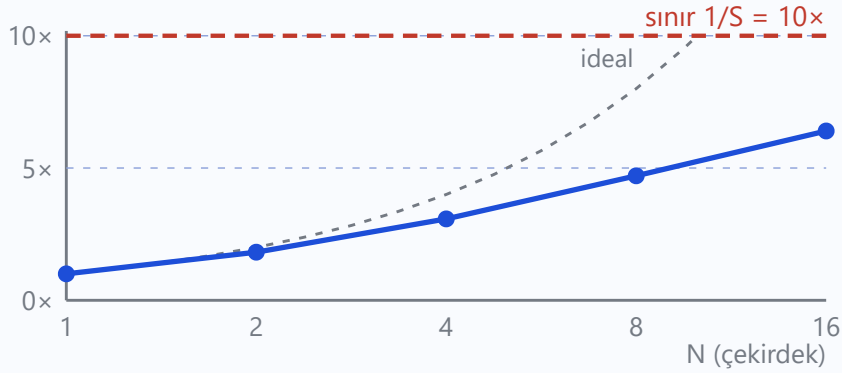
```c
double S = 0.1;            /* seri oran */
int N = 1;                 /* çekirdek */
while (N <= 16) {
  double h = 1 / (S + (1 - S) / N);
  printf("N=%2d  %.2fx\n", N, h);
  N = N * 2;
}
printf("sınır %.0fx\n", 1 / S);
```

Program `S = 0.1` ve `N = 1` ile başlar. `while` döngüsü N 16'yı aşmadığı sürece döner. Her turda Amdahl formülüyle hızlanma `h` hesaplanır, `printf` ile iki ondalık basamakla yazdırılır ve N iki katına çıkarılır. N = 32 olduğunda koşul yanlış olur, döngü biter ve son satır `1 / S` sınırını yazdırır. Tablo 4.5 döngünün her turunu izler.

Tur	N	`S + (1 - S) / N`	h	Çıktı satırı	Önceki tura göre artış
1	1	0,1 + 0,9 = 1	1,00	`N= 1 1.00x`	—
2	2	0,1 + 0,45 = 0,55	1,82	`N= 2 1.82x`	×1,82
3	4	0,1 + 0,225 = 0,325	3,08	`N= 4 3.08x`	×1,69
4	8	0,1 + 0,1125 = 0,2125	4,71	`N= 8 4.71x`	×1,53 (%53)
5	16	0,1 + 0,05625 = 0,15625	6,40	`N=16 6.40x`	×1,36 (%36)
—	32	32 ≤ 16 yanlış: döngü biter; son satır `sınır 10x`			

Tablo 4.5: Amdahl programının izlemesi (`S = 0.1`). Değerler Python ile yeniden hesaplanarak doğrulandı.

Son sütun, azalan getiriyi sayılarla gösterir. Çekirdek sayısı her turda iki katına çıkar, ama hızlanmadaki artış oranı 1,82'den 1,36'ya düşer. 8'den 16 çekirdeğe geçmek hızlanmayı yalnız %36 artırır. Çıktıdaki ondalık ayırıcının nokta olması C'nin `printf` biçimlendirmesinden gelir; metin içinde aynı değerler virgülle yazılır.



Şekil 4.14: İzlemenin sonunda hızlanma grafiği. Mavi eğri 1, 2, 4, 8 ve 16 çekirdekteki hızlanmayı (1,00×, 1,82×, 3,08×, 4,71×, 6,40×) gösterir. Kesikli gri eğri ideal hızlanmadır (h = N); kırmızı kesikli çizgi 1/S = 10× sınırıdır. Gerçek eğri sınıra yaklaşır ama onu asla geçemez.

N sonsuza giderken paralel kısım sıfıra iner, seri %10 kalır ve hızlanma 10×'u asla geçemez. 16 çekirdekte bile hızlanma 6,4×'tür; verim 6,4 / 16 = %40'tır. Çekirdeklerin %60'ı fiilen seri kısmı beklemekle geçer.

4.3 İş parçacığı modelleri

İş parçacıklarını kimin yönettiği sorusunun iki uç cevabı vardır. Ya bütün yönetim kullanıcı alanındaki bir kütüphanede yapılır ve çekirdek iş parçacıklarından habersizdir, ya da çekirdek her iş parçacığını tanır ve kendisi çizelgeler. Bu iki uç ve aralarındaki karma çözümler, iş parçacıklarının maliyetini, bloke olma davranışını ve çok çekirdekten yararlanıp yararlanamayacağını belirler.

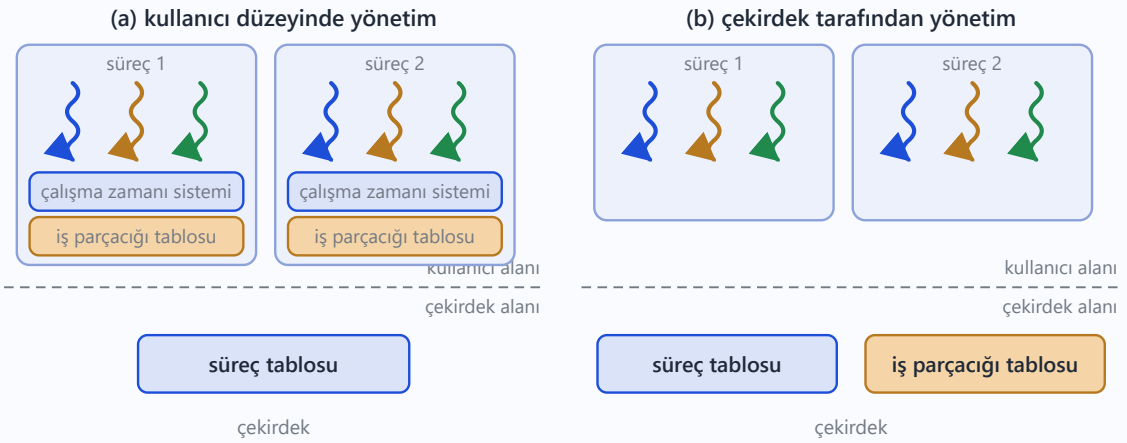
4.3.1 Kullanıcı ve çekirdek düzeyi iş parçacıkları

Tanım 4.3

Kullanıcı düzeyi iş parçacıkları (*user-level threads*), kullanıcı alanında çalışan bir kütüphanede, yani bir **çalışma zamanı sisteminde** (*run-time system*) yönetilir. Çekirdek her süreci tek akışlı sanır.

Çekirdek düzeyi iş parçacıkları (*kernel-level threads*), çekirdeğin tanıdığı ve kendisinin çizelgelediği iş parçacıklarıdır. Linux, Windows ve macOS bu yaklaşımı kullanır.

Şekil 4.15 iki yaklaşımda iş parçacığı tablosunun nerede durduğunu gösterir. Kullanıcı düzeyinde her sürecin kendi çalışma zamanı sistemi ve kendi iş parçacığı tablosu vardır; çekirdek yalnız süreç tablosunu bilir. Çekirdek düzeyinde iş parçacığı tablosu çekirdektedir ve bütün süreçlerin iş parçacıklarını içerir.



Şekil 4.15: (a) Kullanıcı düzeyinde yönetim: süreç 1 ve süreç 2'nin her birinde bir çalışma zamanı sistemi ve bir iş parçacığı tablosu vardır; çekirdek yalnız süreç tablosunu tutar. (b) Çekirdekte yönetim: süreç tablosu da iş parçacığı tablosu da çekirdektedir.

4.3.2 Kullanıcı düzeyi iş parçacıkları

İş parçacıklarını tamamen kullanıcı alanında yönetmek, ilk bakışta tuhaf görünebilir; ama bu yaklaşımın ciddi avantajları vardır.

- **Çekirdek desteği gerekmez.** İş parçacığı desteği olmayan bir çekirdekte bile iş parçacıkları kütüphane olarak gerçeklenebilir. 1990'lara kadar pek çok UNIX sürümü böyleydi.
- **Her süreç kendi tablosunu tutar.** Tablo, her iş parçacığı için PC'yi, yazmaçları, SP'yi ve durumu saklar. Çekirdeğin süreç tablosuna benzer, ama kullanıcı belleğindedir.
- **Geçiş çok hızlıdır.** Bir iş parçacığından ötekine geçmek için tuzak (*trap*) gerekmez, çekirdeğe girilmez, bellek eşlemesi değişmez, TLB boşaltılmaz. Geçiş birkaç komutluk bir fonksiyon çağrısıdır.
- **Çizelgeleme algoritması seçilebilir.** Her süreç kendine uygun bir çizelgeleme algoritması kullanabilir; örneğin bir çöp toplayıcı iş parçacığını en uygun anda çalıştırmak için.

Dezavantajlar ise daha temeldir ve kullanıcı düzeyi iş parçacıklarının işletim sistemi düzeyinde neden terk edildiğini açıklar.

- **Bloke eden sistem çağrısı bütün süreci durdurur.** Bir iş parçacığı `read` çağırıp diskte bloke olursa çekirdek bütün süreci bloke eder; çünkü çekirdek için süreçte tek bir akış vardır. Öteki iş parçacıkları hazır olsa bile çalışamaz. Çözümler zarif değildir: çağrıyı önce `select` ile sınayan ve okunacak veri yoksa başka iş parçacığına geçen sarmalayıcılar (*jacket*) kullanmak ya da bütün çağrıları bloke etmeyen sürümleriyle değiştirmek.
- **Sayfa hatası da bütün süreci bloke eder.** Bir iş parçacığının eriştiği sayfa bellekte değilse çekirdek sayfayı diskten getirirken süreci bekletir; kütüphanenin bundan haberi bile olmaz.
- **Saat kesmesi kütüphaneye gelmez.** Çekirdek zaman dilimi dolunca süreci kesebilir, ama süreç içinde hangi iş parçacığının çalışacağına kütüphane karar verir. Kütüphanenin elinde bir saat kesmesi olmadığı için bir iş parçacığı CPU'yu gönüllü bırakmazsa (`yield`) öteki iş parçacıkları hiç ça-

lışmaz.

- **Tek süreç birden çok çekirdeği kullanamaz.** Çekirdek süreci tek bir akış olarak gördüğü için onu aynı anda tek bir çekirdeğe yerleştirir.

4.3.3 Kod izleme: kullanıcı düzeyinde `thread_yield`

Kullanıcı düzeyinde geçişin ne kadar basit olduğunu görmek için bir kütüphanenin `thread_yield` fonksiyonunu adım adım izleyelim. Aşağıdaki kod, gerçek bir kütüphanenin basitleştirilmiş hâlidir. `PC` ve `SP`, işlemcinin program sayacını ve yığın işaretçisini temsil eden sözde değişkenlerdir; gerçek bir kütüphane bunları birkaç satır assembly ile okur ve yazar.

```
enum { HAZIR, CALISIYOR, BEKLIYOR };
struct tcb { int pc, sp, durum; } T[3];
int cur = 0;               /* çalışan */
void thread_yield(void) {  /* kütüphanede */
  T[cur].pc = PC;  T[cur].sp = SP;
  T[cur].durum = HAZIR;
  do cur = (cur + 1) % 3;
  while (T[cur].durum != HAZIR);
  T[cur].durum = CALISIYOR;
  PC = T[cur].pc;  SP = T[cur].sp;
}
```

`struct tcb` her iş parçacığının kaydedilmiş PC'sini, SP'sini ve durumunu tutar; `T[3]` üç iş parçacıklı bir tablodur. `cur` o anda çalışan iş parçacığının indisidir. `thread_yield` önce çalışan iş parçacığının PC ve SP değerlerini tabloya kaydeder ve onu HAZIR yapar. Ardından `do … while` döngüsü `cur`'u dairesel olarak ilerletir ve HAZIR bir iş parçacığı bulana kadar döner. Bulunan iş parçacığı ÇALIŞIYOR yapılır, kaydedilmiş PC ve SP değerleri işlemciye yüklenir. Fonksiyonun dönüş komutu artık yeni iş parçacığının yığınından döner; yürütme yeni iş parçacığının kaldığı yerden sürer.

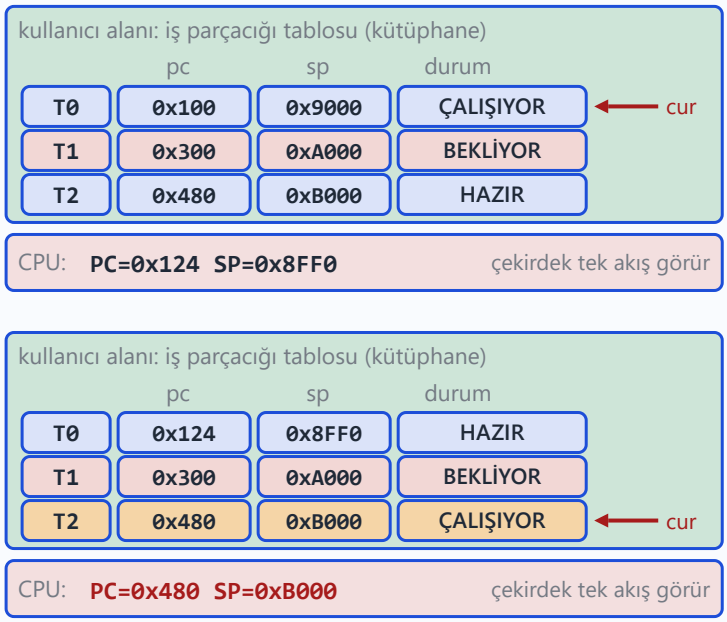
Başlangıçta T0 çalışıyor; işlemcide PC = 0x124, SP = 0x8FF0. Tablodaki T0 satırında eski değerler (0x100, 0x9000) duruyor; bunlar T0'ın son kaydedildiği andan kalmıştır. T1 kütüphanedeki bir kilidi bekliyor (BEKLİYOR), T2 hazır ve en son PC = 0x480, SP = 0xB000 iken bırakmış. T0 gönüllü olarak `thread_yield()` çağırıyor.

Adım	Yürüyen satır	T0 (pc, sp, durum)	T1	T2	`cur`	CPU
0	(başlangıç)	0x100, 0x9000, Ç	0x300, 0xA000, B	0x480, 0xB000, H	0	PC=0x124 SP=0x8FF0
1	`T[cur].pc = PC; T[cur].sp = SP;`	0x124, 0x8FF0, Ç	aynı	aynı	0	aynı
2	`T[cur].durum = HAZIR;`	0x124, 0x8FF0, H	aynı	aynı	0	aynı
3	`cur = (cur + 1) % 3;`	aynı	aynı	aynı	1	aynı

Adım	Yürüyen satır	T0 (pc, sp, durum)	T1	T2	cur	CPU
4	`while (T[1].durum != HAZIR)` : B, döngü sürer	aynı	aynı	aynı	1	aynı
5	`cur = (cur + 1) % 3;`	aynı	aynı	aynı	2	aynı
6	`while (T[2].durum != HAZIR)` : H, döngü biter	aynı	aynı	aynı	2	aynı
7	`T[cur].durum = CALISIYOR; PC = …; SP = …;`	0x124, 0x8FF0, H	0x300, 0xA000, B	0x480, 0xB000, Ç	2	PC=0x480 SP=0xB000
8	T2 biraz çalışır, PC = 0x48C'de `thread_yield()` çağırır: kaydet, HAZIR, `cur` = (2 + 1) % 3 = 0, T0 HAZIR	0x124, 0x8FF0, H	aynı	0x48C, 0xAFF8, H	0	PC=0x48C SP=0xAFF8
9	T0 ÇALIŞIYOR yapılır, yazmaçları yüklenir	0x124, 0x8FF0, Ç	aynı	0x48C, 0xAFF8, H	0	PC=0x124 SP=0x8FF0

Tablo 4.6: `thread_yield` izlemesi. Durumlar: Ç = ÇALIŞIYOR, H = HAZIR, B = BEKLİYOR.

Adım 1 ve 2'de T0'ın durumu kaydedilir. Bu sıradan bir fonksiyon çağrısıdır: tuzak yoktur, çekirdeğe girilmez. Adım 3–6'da tablo taranır; T1 beklediği için atlanır, T2 hazır olduğu için seçilir. Adım 7'de T2'nin yazmaçları yüklenir. Bütün geçiş birkaç komutla biter. Adım 8'de T2 de gönüllü olarak bırakır ve sıra dairesel olarak yeniden T0'a gelir. Adım 9'da T0, 0x124 adresinden, yani kaldığı yerden sürer. Şekil 4.16 başlangıcı ve 7. adımı gösterir.



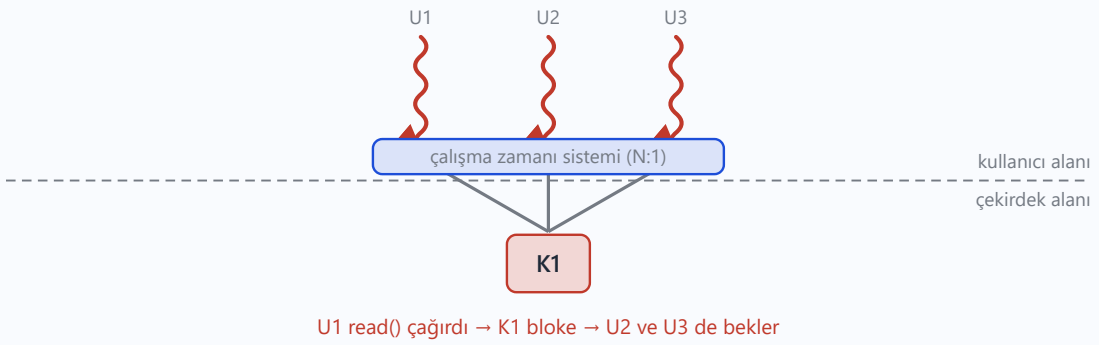
Şekil 4.16: Kullanıcı alanındaki iş parçacığı tablosu. (a) Başlangıç: T0 çalışıyor, `cur` T0'ı gösteriyor, işlemcide PC = 0x124. (b) 7. adım: T0'ın değerleri kaydedilmiş ve HAZIR, T1 hâlâ BEKLİYOR, T2 ÇALIŞIYOR ve işlemciye PC = 0x480, SP = 0xB000 yüklenmiş. Çekirdek bütün bu süre boyunca tek bir akış görür.

Sık yapılan hata

Kullanıcı düzeyi iş parçacıklarının da zaman dilimiyle kesileceğini varsaymak. T2 sonsuz bir hesap döngüsüne girip hiç `thread_yield()` çağırmasaydı T0 sonsuza dek bekleyecekti. Saat kesmesi çekirdeğe gelir; çekirdek süreci kesip başka bir süreç çalıştırabilir, ama süreç yeniden çalıştığında yine T2 kaldığı yerden sürer. Kütüphanenin araya girme şansı yoktur.

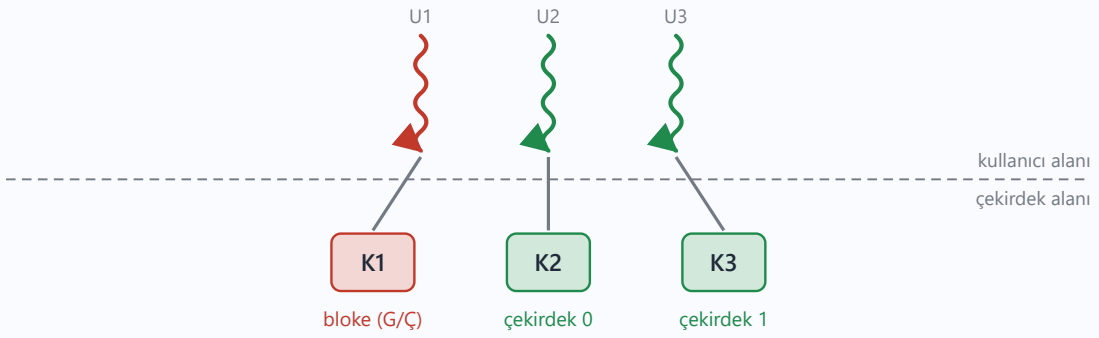
4.3.4 Bloke eden çağrı: N:1 ve 1:1

Kullanıcı düzeyi iş parçacıklarının en ağır sorunu, bloke eden sistem çağrılarıdır. Sorunu görünür kılmak için iki modeli yan yana koyalım. **Çoktan bire** (N:1) modelde bir sürecin bütün kullanıcı iş parçacıkları tek bir çekirdek iş parçacığı üzerinde çalışır. **Bire bir** (1:1) modelde her kullanıcı iş parçacığının kendi çekirdek iş parçacığı vardır.



Şekil 4.17: Çoktan bire model. U1, U2 ve U3 kullanıcı iş parçacıkları çalışma zamanı sistemi üzerinden tek bir çekirdek iş parçacığına (K1) eşlenmiştir. U1 bloke eden `read()` çağırır; çekirdek yalnız K1'i görür ve onu bloke eder, bu yüzden U2 ve U3 de bekler.

N:1 modelde U1'in `read()` çağrısı çekirdeğe ulaştığında çekirdek bu sürece ait tek bir çizelgelenebilir varlık görür: K1. Okuma bitene kadar K1'i bloke eder. U2 ve U3 hazır olsa da onları çalıştıracak bir çekirdek iş parçacığı yoktur; bütün süreç durur. 1:1 modelde aynı çağrı yalnız U1'in çekirdek iş parçacığını, K1'i bloke eder. U2 ve U3'ün kendi çekirdek iş parçacıkları (K2 ve K3) vardır; çizelgeleyici onları iki çekirdekte paralel olarak çalıştırmayı sürdürür.

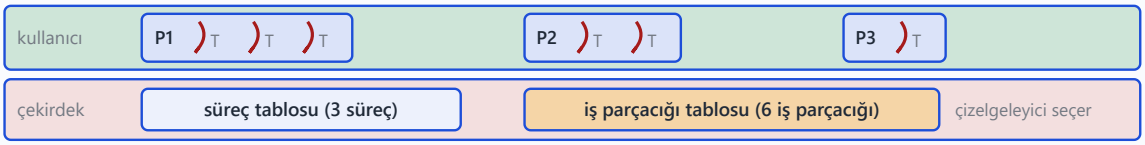


Şekil 4.18: Bire bir model. U1 G/Ç'de bloke olunca yalnız K1 bloke olur; U2 ve U3, K2 ve K3 üzerinden çekirdek 0 ve çekirdek 1'de çalışmaya devam eder.

4.3.5 Çekirdek düzeyi iş parçacıkları

Çekirdek düzeyi iş parçacıklarında çekirdek, kullanıcı düzeyindeki tablonun aynı bilgilerini (yazmaçlar, durum ve benzerleri) içeren bir iş parçacığı tablosu tutar. Bu düzenin sonuçları şöyle sıralanabilir.

- İş parçacığı oluşturmak ve sonlandırmak bir sistem çağrısıdır. Bu yüzden kullanıcı düzeyine göre daha pahalıdır.
- Bir iş parçacığı bloke olursa çekirdek başka birini seçer. Seçilen iş parçacığının aynı süreçten olması gerekmez.
- Sayfa hatası yalnız hatayı yapan iş parçacığını bekletir.
- Bir sürecin iş parçacıkları farklı çekirdeklerde gerçekten paralel çalışır.
- Her iş parçacığı çekirdek belleğinde yer kaplar: TCB ve çekirdek yığını. Bu yüzden sayısı sınırsız değildir.

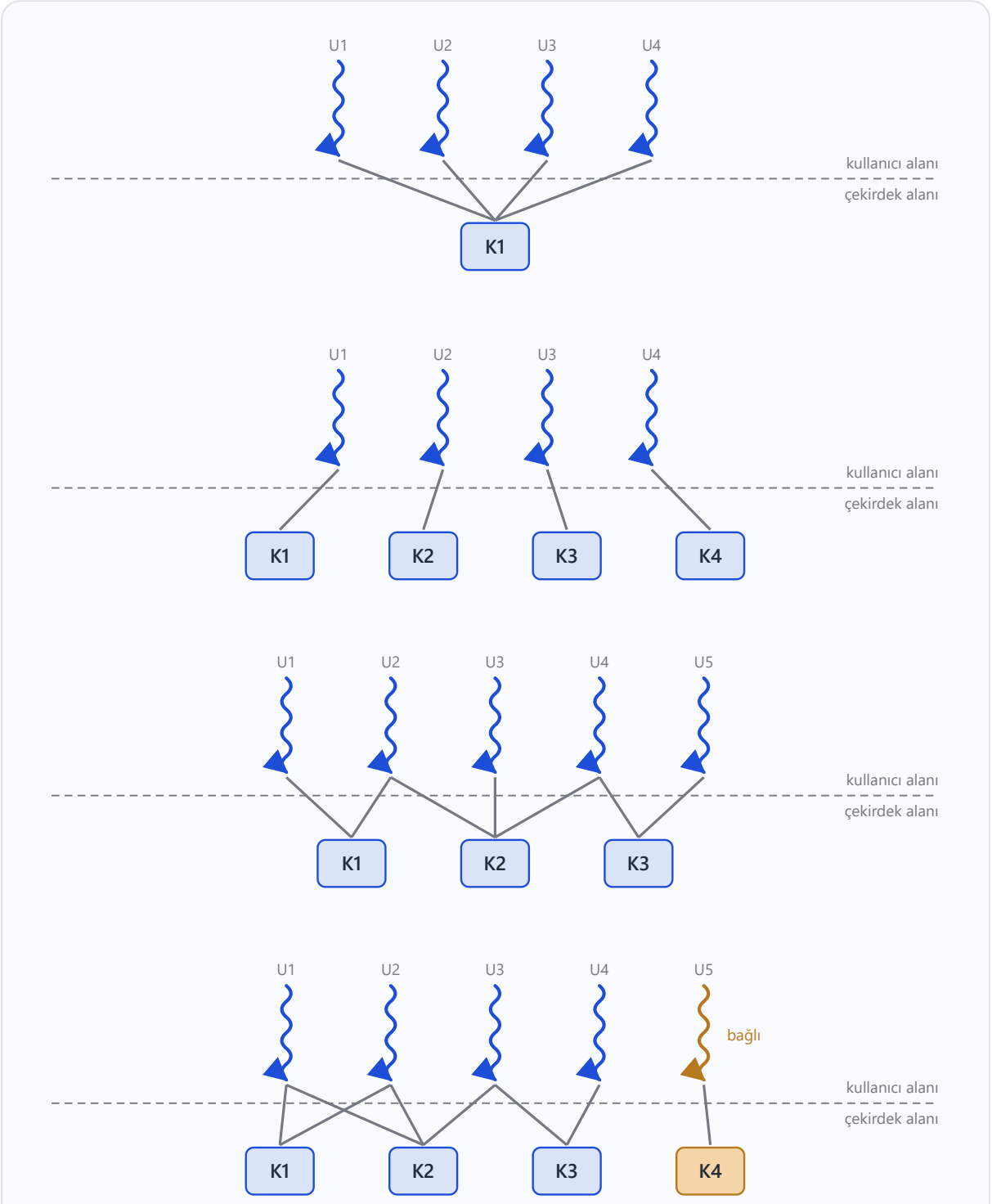


Şekil 4.19: Çekirdek düzeyi iş parçacıkları. Kullanıcı alanında P1'in üç, P2'nin iki, P3'ün bir iş parçacığı vardır. Çekirdekte süreç tablosu üç süreci, iş parçacığı tablosu altı iş parçacığını tutar; çizelgeleyici iş parçacıkları arasından seçim yapar.

Bugün Linux (NPTL, *Native POSIX Thread Library*), Windows ve macOS 1:1 modelini kullanır. Çekirdek iş parçacığı oluşturmanın maliyeti yıllar içinde azaldığı için on binlerce çekirdek iş parçacığı sorun değildir. Linux'ta iş parçacığı sayısını üç sınır belirler: sistem geneli `/proc/sys/kernel/threads-max`, kullanıcı başına `ulimit -u` ve denetim grubu başına `pids.max`. Bir sürecin iş parçacıklarını gözlemlemek için `ps -eLf` (her iş parçacığı bir satır), `top -H` ya da `/proc/<pid>/task/` dizini kullanılır; bu dizinde her iş parçacığı için bir alt dizin bulunur.

4.3.6 Eşleme modelleri

Kullanıcı iş parçacıklarının çekirdek iş parçacıklarına nasıl eşlendiği, dört model altında toplanır. Şekil 4.20 dördünü yan yana gösterir.



Şekil 4.20: Eşleme modelleri. (a) Çoktan bire: dört kullanıcı iş parçacığı tek çekirdek iş parçacığına eşlenir. (b) Bire bir: her kullanıcı iş parçacığının ayrı bir çekirdek iş parçacığı vardır. (c) Çoktan çoğa: beş kullanıcı iş parça-

cığı üç çekirdek iş parçacığına çoğullanır. (d) İki düzeyli: çoktan çoğa eşlemeye ek olarak U5 doğrudan K4'e bağlanmıştır.

Çoktan bire (N:1)
: Geçiş hızlıdır, ama bir bloke eden çağrı bütün süreci durdurur ve paralellik yoktur. Örnekler: Solaris'in yeşil iş parçacıkları (*green threads*) ve GNU Portable Threads; ikisi de bugün tarihîdir.

Bire bir (1:1)
: Gerçek paralellik vardır ve her iş parçacığı bağımsız olarak bloke olabilir. Bedeli, her iş parçacığının çekirdek kaynağı harcamasıdır. Linux, Windows ve macOS bu modeli kullanır.

Çoktan çoğa (M:N)
: Çok sayıda kullanıcı iş parçacığı daha az sayıda çekirdek iş parçacığına **çoğullanır**. Kullanıcı iş parçacığı oluşturmak ucuzdur, birden çok çekirdek kullanılır ve bir iş parçacığı bloke olunca çalışma zamanı ötekileri başka çekirdek iş parçacıklarında çalıştırabilir. Sorun gerçeklemenin zorluğudur: kullanıcı alanında ve çekirdekte iki ayrı çizelgeleyici birbirinden habersiz karar verir.

İki düzeyli (hibrit)
: M:N'e ek olarak bir kullanıcı iş parçacığı tek bir çekirdek iş parçacığına kalıcı olarak **bağlanabilir** (*bound*). Gecikmeye duyarlı bir iş parçacığı böylece kendi çekirdek iş parçacığına kavuşur.

4.3.7 Modellerin karşılaştırılması

Ölçüt	N:1	1:1	M:N
Geçiş maliyeti	çok düşük	orta (çekirdek)	düşük
Bloke eden çağrı	bütün süreç durur	yalnız o iş parçacığı	yalnız o iş parçacığı*
Çok çekirdek	kullanılamaz	kullanılır	kullanılır
İş parçacığı sayısı	çok fazla	çekirdek belleğiyle sınırlı	çok fazla
Gerçekleme	kolay (kütüphane)	kolay	zor (iki çizelgeleyici)

Tablo 4.7: Eşleme modellerinin karşılaştırılması.

Yıldızlı hücre bir koşula bağlıdır: M:N modelde bir iş parçacığının bloke olması ötekileri ancak çalışma zamanı bloke olan çağrıyı fark edip öteki kullanıcı iş parçacıklarını başka bir çekirdek iş parçacığına aktarabiliyorsa etkilemez.

İşletim sistemi düzeyinde M:N modeli terk edildi. Solaris 9 ve FreeBSD 7, iki çizelgeleyicinin uyum sorunları ve gerçeklemenin karmaşıklığı yüzünden 1:1 modeline geçti. Ancak M:N modeli dil çalışma zamanlarında geri döndü. Go dilinin hafif iş parçacıkları (*goroutines*), Erlang süreçleri ve Java 21 sanal iş parçacıkları M:N modelini çekirdekte değil, dilin çalışma zamanında uygular. Çalışma zamanı bloke edebilecek çağrıları zaten kendisi yönettiği için eski M:N gerçeklemelerinin en büyük sorunu ortadan kalkar.

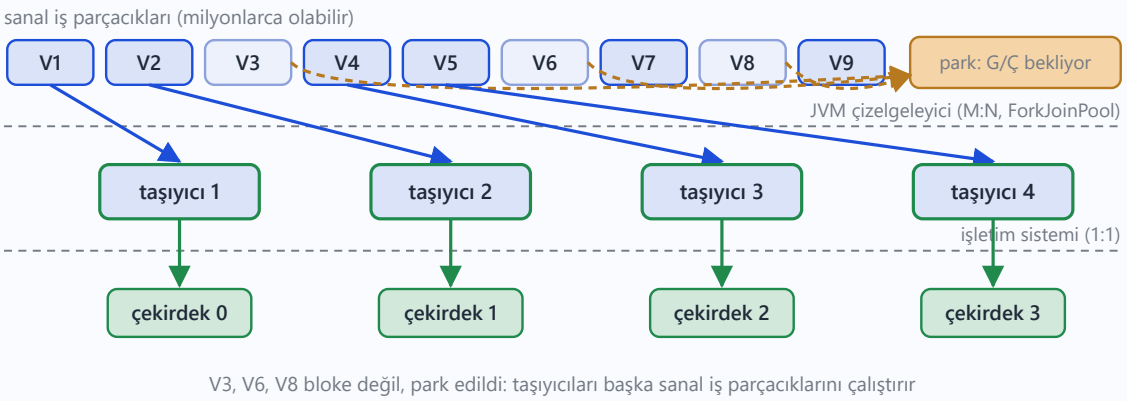
Örnek 4.5

Bir program çalışması boyunca 500.000 iş parçacığı geçişi yapıyor. Kullanıcı düzeyinde (N:1) bir geçiş 50 ns, çekirdek düzeyinde (1:1) 2 µs sürüyor. İki modelde geçişlere toplam ne kadar zaman gider? İş parçacıklarından biri 10 ms süren bir disk okuması yaptığında ne olur?

1. **N:1:** 500.000 × 50 ns = 25.000.000 ns = 0,025 s.
2. **1:1:** 500.000 × 2 µs = 1.000.000 µs = 1 s. Kullanıcı düzeyi geçiş 40 kat ucuzdur, çünkü tuzak ve çekirdeğe giriş yoktur.
3. **Disk okuması, N:1:** Çekirdek tek bir çekirdek iş parçacığı görür ve onu bloke eder. Bütün süreç 10 ms bekler; bu süre 200.000 kullanıcı düzeyi geçişe eşittir.
4. **Disk okuması, 1:1:** Yalnız okuyan iş parçacığı bekler. Öteki iş parçacıkları o 10 ms içinde, başka çekirdeklerde paralel olarak bile çalışır.
5. **Sonuç.** Geçiş maliyeti tek başına model seçimini belirlemez. G/Ç yapan bir programda tek bir bloke eden çağrı, N:1 modelinin geçişlerde kazandırdığı zamanın büyük kısmını yok edebilir. M:N modeli iki üstünlüğü birleştirmeye çalışır: ucuz geçiş ve bloke olmayan çalışma zamanı.

4.3.8 Güncel M:N: sanal iş parçacıkları ve Go

Java 21 ile kalıcı hâle gelen **sanal iş parçacıkları** (*virtual threads*, Loom projesi), M:N modelinin bugünkü en yaygın örneğidir. JVM çok sayıda sanal iş parçacığını az sayıda platform iş parçacığına çoğullar. Sanal iş parçacığını çalıştıran platform iş parçacığına **taşıyıcı** (*carrier*) denir; taşıyıcılar sıradan çekirdek iş parçacıklarıdır ve işletim sistemi onları 1:1 modeliyle çizelgeler. Şekil 4.21 bu iki katmanı gösterir.



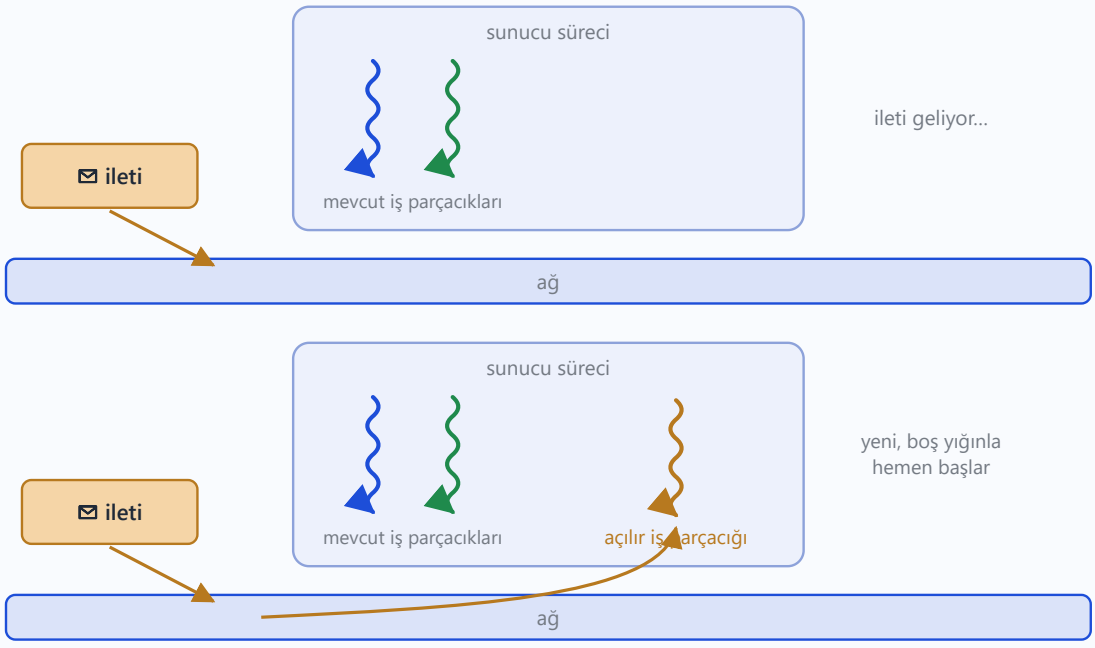
Şekil 4.21: Java sanal iş parçacıkları. V1–V9 sanal iş parçacıklarından bazıları dört taşıyıcıya bağlıdır; G/Ç bekleyen V3, V6 ve V8 park edilmiş ve heap'e alınmıştır. JVM çizelgeleyicisi (M:N, ForkJoinPool) sanal iş parçacıklarını taşıyıcılara yerleştirir; taşıyıcılar işletim sisteminde 1:1 modeliyle dört çekirdekte çalışır.

Sanal iş parçacığı bloke eden bir G/Ç çağrısı yaptığında JVM onu taşıyıcısından **indirir** (*unmount*): yığın çerçevelerini heap'e kaydeder ve iş parçacığını park eder. Taşıyıcı serbest kalır ve başka bir sanal iş parçacığını çalıştırır. G/Ç bitince park edilmiş iş parçacığı yeniden bir taşıyıcıya bindirilir (*mount*) ve kaldığı yerden sürer. Şekildeki V3, V6 ve V8 bloke değildir, park edilmiştir; taşıyıcıları bu sırada başka sanal iş parçacıklarını çalıştırır. Çekirdek yalnız taşıyıcıları görür. Bu yüzden milyonlarca sanal iş parçacığı oluşturmak mümkündür. Sanal iş parçacığı `Thread.ofVirtual().start(r)` ile başlatılır.

Go dilinin çalışma zamanı aynı fikri üç harfle anlatır. **G** bir Go iş parçacığıdır (`go f()` ile başlatılır). **M** bir işletim sistemi iş parçacığıdır (*machine*). **P**, Go kodu çalıştırma hakkını temsil eden mantıksal işlemcidir; sayısı `GOMAXPROCS` ile belirlenir ve varsayılan olarak çekirdek sayısına eşittir. Her P'nin çalıştırılmayı bekleyen G'lerden oluşan bir kuyruğu vardır. Kuyruğu boşalan bir P, başka bir P'nin kuyruğundan iş **çalar**. Bir G bloke eden bir sistem çağrısına girerse çalışma zamanı P'yi o M'den ayırıp başka bir M'ye verir; öteki G'ler çalışmaya devam eder.

4.3.9 Açılır iş parçacıkları

Dağıtık sistemlerde gelen iletileri işlemenin geleneksel yolu, `receive` çağrısında bloke olmuş bir iş parçacığının ileti gelince uyanmasıdır. Bu iş parçacığının durumu, yani yazmaçları ve yığını, her iletide geri yüklenmelidir. Daha hızlı bir seçenek, ileti geldiğinde onu işlemek için yeni bir iş parçacığı oluşturmaktır. Bu iş parçacığına **açılır iş parçacığı** (*pop-up thread*) denir.



Şekil 4.22: Açılır iş parçacığı. (a) İleti gelmeden önce: sunucu sürecinde bu iletiyi bekleyen bir iş parçacığı yoktur. (b) İleti gelince sistem onu işleyecek yeni bir iş parçacığı oluşturur; iş parçacığı boş bir yığınla hemen başlar.

Açılır iş parçacığı yenidir; geri yüklenecek bir geçmişi, yazmaçları ya da yığın içeriği yoktur. Bu yüzden iletinin gelişiyle işlenmeye başlaması arasındaki gecikme düşüktür. Aynı fikir çekirdek içinde de kullanılır: bazı çekirdekler kesmeleri, kesme geldiğinde oluşturulan kısa ömürlü iş parçacıklarında işler. Pthreads ile bir açılır iş parçacığı şöyle oluşturulabilir:

```
/* mesaj gelince */
pthread_t t;
pthread_create(&t, NULL, isle, msg);
pthread_detach(t); /* join yok */
```

`pthread_create` iletiyi `isle` fonksiyonuna argüman olarak verir. `pthread_detach` ise iş parçacığını **ayrık** yapar: iş parçacığı bitince kaynakları kendiliğinden geri verilir ve kimsenin onu `pthread_join` ile beklemesi gerekmez. Sunucunun gelen her ileti için oluşturduğu iş parçacıklarını tek tek beklemesi anlamsız olacağı için ayrık iş parçacığı burada doğal seçimdir.

4.4 İş parçacığı kütüphaneleri

Programcı iş parçacıklarını bir **iş parçacığı kütüphanesi** aracılığıyla oluşturur ve yönetir. Kütüphane, iş parçacığı oluşturma, bekleme, sonlandırma ve senkronizasyon için bir uygulama programlama arayüzü (*application programming interface*, API) sunar. Kütüphane tamamen kullanıcı alanında olabilir; bu durumda bir kütüphane fonksiyonu çağırmak sıradan bir fonksiyon çağrısıdır. Ya da kütüphane çekirdeğin desteğine dayanır ve fonksiyonlarının çoğu bir sistem çağrısıyla sonuçlanır.

4.4.1 Kütüphaneler ve iki kullanım biçimi

POSIX Pthreads
: Linux, macOS ve BSD sistemlerinin C API'sidir. İş parçacığı oluşturma ve beklemenin yanında karşılıklı dışlama kilidi (*mutex*) ve koşul değişkeni de sağlar.

Windows API
: `CreateThread` ile oluşturur, `WaitForSingleObject` ile bekler. Çekirdek düzeyidir ve 1:1 modelini kullanır.

Java
: `Thread`, `Runnable` ve `ExecutorService` sınıflarıyla çalışır. Java sanal makinesi (JVM) işletim sisteminin iş parçacıklarını, Java 21'den beri de sanal iş parçacıklarını kullanır.

C ve C++ standardı
: C11 `<threads.h>` başlığını; C++11 `std::thread`'i, C++20 `std::jthread`'i getirdi. Bunlar eskiden yaygın olan Boost.Thread kütüphanesinin yerini aldı.

Python
: `threading` modülünü sunar. Ancak standart yorumlayıcıdaki global yorumlayıcı kilidi (*global interpreter lock*, GIL) bir anda yalnız bir iş parçacığının Python kodu çalıştırmasına izin verir;

bu yüzden işlemci yoğun işlerde paralellik elde edilmez. Python 3.13'te deneysel olarak gelen GIL'siz derleme, 3.14'te desteklenen bir seçenek oldu.

Go

`go f()` ifadesi yeni bir Go iş parçacığı başlatır. Çalışma zamanı M:N modelini kullanır; iş parçacıkları kanallarla iletişim kurar.

İş parçacıkları iki farklı biçimde kullanılır. **Eş zamansız** (*asynchronous*) kullanımda ebeveyn iş parçacığı çocuğu başlatır ve kendi işine devam eder; ikisi arasında az veri paylaşılır. Çok iş parçacıklı web sunucusu buna örnektir: dağıtıcı işçiye işi verir ve sonucunu beklemez. **Senkron** kullanımda ise ebeveyn çocuklarının bitmesini bekler (`join`) ve sonuçlarını birleştirir. Bu düzene **fork-join** denir: iş çatallanır (*fork*), parçalar paralel çalışır, sonra yeniden birleşir (*join*).

4.4.2 POSIX Pthreads

Pthreads, IEEE 1003.1c (POSIX.1c) standardının tanımladığı iş parçacığı oluşturma ve senkronizasyon API'sidir. Önemli bir ayrıntı, Pthreads'in bir **tanımlama** olmasıdır, gerçekleme değil. Standart kütüphanenin nasıl davranacağını söyler; bunun kullanıcı düzeyinde mi, çekirdek düzeyinde mi yapılacağını gerçekleyiciye bırakır. Linux'ta glibc içindeki NPTL, Pthreads'i 1:1 modeliyle gerçekler; macOS ve BSD'lerde de Pthreads çekirdek iş parçacıkları üzerinde çalışır. Pthreads kullanan bir program `gcc prog.c -pthread` komutuyla derlenir.

Çağrı	Açıklama
`pthread_create`	Yeni iş parçacığı oluşturur ve verilen fonksiyonu onda çalıştırır.
`pthread_exit`	Çağıran iş parçacığını sonlandırır.
`pthread_join`	Belirli bir iş parçacığının bitmesini bekler.
`pthread_detach`	İş parçacığını ayrık yapar: bitince kaynakları kendiliğinden geri verilir; join edilmez.
`pthread_self`	Çağıran iş parçacığının kendi kimliğini döndürür.
`sched_yield`	İşlemciyi bırakır. (`pthread_yield` standart dışıdır ve glibc 2.34'te kullanımdan kaldırıldı.)
`pthread_attr_init` / `_destroy`	Yığın boyutu, ayrık durum gibi ayarları tutan öznitelik yapısını ilklendirir ya da kaldırır.

Tablo 4.8: Temel Pthreads çağrıları.

`pthread_create` 'in imzası Pthreads'in genel tarzını gösterir: `int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*fonk)(void *), void *arg)` . İlk parametreye yeni iş parçacığının kimliği yazılır. İkinci parametre öznitelikleri verir; `NULL` varsayılan ayarlar demektir. Üçüncü parametre iş parçacığının çalıştıracağı fonksiyondur; bu fonksiyon tek bir `void *` alır ve bir `void *` döndürür. Dördüncü parametre o fonksiyona geçirilecek argümandır. Çağrı başarıda 0, hatada bir hata kodu döndürür.

4.4.3 Pthreads: oluştur ve bekle

İlk örnek Silberschatz'ın Şekil 4.11'deki programından uyarlanmıştır. Program, komut satırından gelecek bir üst sınıra kadar olan tam sayıları ayrı bir iş parçacığında toplar; burada üst sınır doğrudan `"5"` dizgisi olarak verilmiştir. `#include <pthread.h>`, `<stdio.h>` ve `<stdlib.h>` satırları gösterilmemiştir.

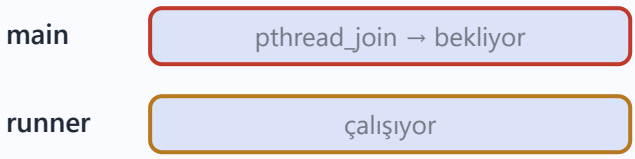
```c
int toplam;           /* paylaşılan */
void *runner(void *param) {
  int ust = atoi(param);
  toplam = 0;
  for (int i = 1; i <= ust; i++)
    toplam += i;
  pthread_exit(0);
}
int main(void) {
  pthread_t tid;
  pthread_create(&tid, NULL, runner, "5");
  pthread_join(tid, NULL);
  printf("toplam = %d\n", toplam);
  return 0;
}
```

`toplam` global bir değişkendir; bu yüzden iki iş parçacığı da onu görür. `runner`, iş parçacığının çalıştıracağı fonksiyondur. Parametresini `atoi` ile tam sayıya çevirir, kendi yığınındaki `ust` değişkenine koyar, 1'den `ust`'e kadar sayıları `toplam`'a ekler ve `pthread_exit(0)` ile sonlanır. `main`, `pthread_create` ile yeni iş parçacığını başlatır, `pthread_join` ile onun bitmesini bekler ve sonucu yazdırır.

Adım	main	runner	toplam	Açıklama
1	çalışıyor	henüz yok	0	Süreç tek iş parçacığıyla başlar; global `toplam` sıfır ilklendirilmiştir.
2	`pthread_create`	hazır	0	Yeni iş parçacığı oluşturulur; `runner("5")` çalışmaya adaydır.
3	`pthread_join` → bekliyor	çalışıyor	0	`runner`'ın ne zaman başlayacağını çizelgeleyici belirler.
4	bekliyor	`ust = atoi("5")` = 5	0	`ust` runner'ın kendi yığınındadır.
5	bekliyor	döngü: 1 + 2 + 3 + 4 + 5	15	Paylaşılan `toplam` güncellenir.
6	uyanıyor	`pthread_exit(0)` → bitti	15	runner sonlanır; join döner.
7	`printf`	bitti	15	Çıktı: `toplam = 15`.

Tablo 4.9: Oluştur-bekle programının izlemesi.

Programın çıktısı her çalıştırmada aynıdır, çünkü `main` `toplam`'ı yalnız `pthread_join` döndükten sonra okur. Join, runner'ın bütün yazmalarını bitirdiğini garanti eder. Join olmasaydı `main`, runner daha başlamadan `toplam`'ı okuyup 0 yazdırabilirdi. Daha kötüsü, `main`'den dönmek bütün süreci sonlandırır; runner işini bitiremeden ölürdü. 3. adımdaki gözlem de önemlidir: `pthread_create` döndüğünde yeni iş parçacığının çalışmaya başladığının hiçbir garantisi yoktur. Hangisinin önce çalışacağına çizelgeleyici karar verir.



Şekil 4.23: İzlemenin 3. adımı: `main` `pthread_join`'de bekliyor, runner çalışıyor.

4.4.4 Birden çok iş parçacığı

Veri paralelliğini Pthreads ile göstermek için 1'den 1000'e kadar sayıların toplamını dört iş parçacığına bölelim. Her iş parçacığı 250 sayı toplar.

```
#define N 4
long kismi[N];          /* her biri kendi hücresine */

void *topla(void *arg) {
  long k = (long)arg;    /* 0 .. N-1 */
  for (long i = k*250 + 1; i <= (k+1)*250; i++)
    kismi[k] += i;
  return NULL;
}
int main(void) {
  pthread_t t[N];
  for (long k = 0; k < N; k++)
    pthread_create(&t[k], NULL, topla, (void *)k);
  long toplam = 0;
  for (int k = 0; k < N; k++) {
    pthread_join(t[k], NULL);
    toplam += kismi[k];
  }
  printf("%ld\n", toplam);  /* 500500 */
}
```

İş parçacığı numarası `k`, işaretçi olarak değil, **değer** olarak geçirilir: `(void *)k` ifadesi sayıyı doğrudan işaretçi alanına koyar, `topla` içindeki `(long)arg` onu geri çıkarır. `long` ile işaretçi aynı genişlikte olduğu için bu dönüşüm güvenlidir. `k` numaralı iş parçacığı *250k + 1*'den *250(k + 1)*'e kadar sa-

yıları kendi `kismi[k]` hücresine ekler. `main` dört iş parçacığını oluşturur, sonra sırayla her birini join ile bekleyip kısmi toplamı ana toplama ekler.

k	Aralık	`kismi[k]`
0	1 … 250	31.375
1	251 … 500	93.875
2	501 … 750	156.375
3	751 … 1000	218.875
Toplam		500.500

Tablo 4.10: Dört iş parçacığının aralıkları ve kısmi toplamları (Python ile doğrulandı).

Programda yarış durumu yoktur, çünkü her iş parçacığı yalnız kendi hücresine yazar ve `main` bir hücreyi ancak o hücrenin sahibi bittikten sonra okur. İş parçacıkları hangi sırayla çalışırsa çalışsın sonuç her zaman 500.500'dür.

Sık yapılan hata

Argüman olarak döngü değişkeninin adresini, `&k` 'yi geçirmek. Dört iş parçacığı da aynı `k` değişkenini gösteren bir işaretçi alır. Döngü bu sırada `k` 'yi artırmaya devam ettiği için bir iş parçacığı `*arg` 'ı okuduğunda `k` çoktan 2 ya da 4 olmuş olabilir; iki iş parçacığı aynı aralığı toplar, bir aralık hiç toplanmaz. Döngü bitince `k` 'nin ömrü de sona erer. Çözüm, değeri doğrudan geçirmek ya da her iş parçacığına ayrı bir bellek hücresi vermektir.

Kodda bir performans tuzağı da vardır. `kismi` dizisinin dört hücresi bellekte yan yanadır ve aynı önbellek satırına düşer. Dört çekirdek bu satıra sürekli yazdığı için satır çekirdekler arasında gidip gelir: **yanlış paylaşım**. Her iş parçacığı toplamı önce yerel bir değişkende biriktirip sonda bir kez `kismi[k]` 'ye yazarsa sorun ortadan kalkar.

4.4.5 Argüman geçirme ve sonuç alma

İş parçacığı fonksiyonunun tek bir `void *` parametresi ve tek bir `void *` dönüş değeri vardır. Birden çok değer geçirmek ya da almak için değerler bir yapıya konur ve yapının adresi geçirilir. Aşağıdaki program bir aralığın alt ve üst sınırını yapı içinde verir, toplamı ve sayı adedini de yapı içinde geri alır.

```
typedef struct { int alt, ust; } aralik_t;
typedef struct { long toplam; int adet; } sonuc_t;

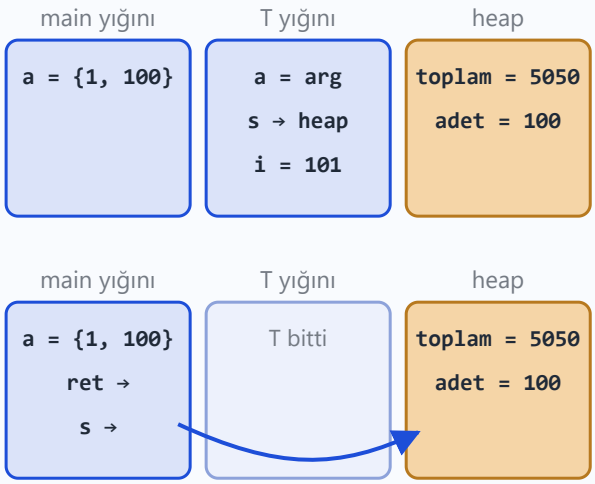
void *topla(void *arg) {
  aralik_t *a = arg;          /* void* → tür */
  sonuc_t *s = malloc(sizeof *s); /* heap'te */
  s->toplam = 0; s->adet = 0;
  for (int i = a->alt; i <= a->ust; i++) {
    s->toplam += i; s->adet++;
  }
  return s;         /* = pthread_exit(s) */
}
int main(void) {
  aralik_t a = { 1, 100 };
  pthread_t t;  void *ret;
  pthread_create(&t, NULL, topla, &a);
  pthread_join(t, &ret);          /* ret = s */
  sonuc_t *s = ret;
  printf("%d sayı, toplam %ld\n", s->adet, s->toplam);
  free(s);
  return 0;
}
```

`main`, `a` yapısını kendi yığınında oluşturur ve adresini argüman olarak verir. `topla`, `void *`'ı gerçek türüne, `aralik_t *`'a çevirir. Sonuç için heap'te `malloc` ile bir `sonuc_t` ayırır, alanlarını sıfırlar ve döngüde 1'den 100'e kadar sayıları toplarken adedi de sayar. `return s;` satırı `pthread_exit(s)` ile aynı etkiyi yapar: işaretçi, bu iş parçacığını join ile bekleyene teslim edilir. `main`'deki `pthread_join(t, &ret)` bu işaretçiyi `void **` türündeki ikinci parametre üzerinden `ret`'e yazar.

Adım	main yığını	T yığını	Heap
1	`a = {1, 100}`	yok	—
2	`a` ; `pthread_create`	`arg` → `a`	—
3	`a`	`a = arg`, `s` → heap	`toplam = 0, adet = 0`
4	`a`	`i = 101` (döngü bitti)	`toplam = 5050, adet = 100`
5	`a`	çerçeve silindi	`toplam = 5050, adet = 100`
6	`ret` → heap, `s = ret`	T bitti	`toplam = 5050, adet = 100`
7	`printf`, `free(s)`	T bitti	(geri verildi)

Tablo 4.11: Argüman ve sonuç programının izlemesi.

Program `100 sayı, toplam 5050` yazdırır. İzlemede iki ömür sorusu öne çıkar. Birincisi argümanın ömrüdür: `a` `main`'in yığınındadır ve `main` T'yi join edene kadar bu çerçeve yaşar. Bu yüzden adresini vermek güvenlidir. İkincisi sonucun ömrüdür: T'nin yığını T bitince yok olur, ama heap'teki nesne yaşamaya devam eder. Sahiplik join ile `main`'e geçer ve belleği geri vermek (`free`) artık `main`'in görevidir. Şekil 4.24 iki önemli anı gösterir.



Şekil 4.24: (a) 4. adım: T'nin yığınında `a` ve `s` işaretçileri, heap'teki yapıda `toplam = 5050` ve `adet = 100`. (b) 6. adım: T bitmiş ve yığını gitmiş; `main`'in `ret` ve `s` işaretçileri heap'teki yapıyı gösteriyor.

4.4.6 Yaygın hata: yığındaki sonucu döndürmek

Sonucu heap'te ayırmak yerine iş parçacığının kendi yığınında tutup adresini döndürmek, Pthreads programlarında sık görülen bir hatadır:

```
void *topla(void *arg) {
  sonuc_t s = { 0, 0 };  /* T'nin yığınında */
  /* ... s doldurulur ... */
  return &s;             /* HATA */
}
```

Hatalı sürümde sonuç, T'nin yığınındaki yerel `s` değişkenidir ve `&s` döndürülür. Fonksiyon dönünce çerçevesi silinir. İş parçacığı bitince yığınının tamamı geri verilir ya da daha sonra oluşturulacak başka bir iş parçacığına verilebilir. `main` join'den aldığı adresi okuduğunda sonuç tanımsızdır: çöp bir değer, `SIGSEGV` ya da en kötüsü, bazen doğru görünen bir değer. Sonuncusu en tehlikelisidir, çünkü hata testlerde görünmeyip kullanımda ortaya çıkar. Doğru sürüm heap'i kullanır:

```
void *topla(void *arg) {
  sonuc_t *s = malloc(sizeof *s);
  /* ... s doldurulur ... */
  return s;    /* heap'te: yaşar */
}
```



Şekil 4.25: Hatalı sürümün sonu. T dönmüş, yığın çerçevesi silinmiş ve yığın geri verilmiştir. `main`'deki `ret` hâlâ eski adresi (0x7f3a...e40) tutar; `ret->toplam`'ı okumak tanımsız davranıştır.

Aynı kural argüman için de geçerlidir: iş parçacığına verilen adres, iş parçacığı onu kullandığı sürece yaşamalıdır. GCC çoğu durumda yerel bir değişkenin adresini döndüren fonksiyonlar için `-Wreturn-local-addr` uyarısı verir; bu uyarı hata gibi ele alınmalıdır.

4.4.7 Java: `Thread`, `start` ve `join`

Java'da her program en az bir iş parçacığıyla, `main` metodunu çalıştıran ana iş parçacığıyla başlar. Yeni bir iş parçacığı oluşturmanın en kısa yolu, `Thread` kurucusuna çalıştırılacak kodu bir lambda ifadesi olarak vermektir. Aşağıdaki program, Pthreads örneğindeki toplamı Java ile yapar.

```
public class Topla {
  static int toplam = 0;
  public static void main(String[] a)
      throws InterruptedException {
    Thread t = new Thread(() -> {
      for (int i = 1; i <= 5; i++) toplam += i;
    });
    t.start();
    System.out.println("main: bekliyor");
    t.join();
    System.out.println("toplam = " + toplam);
  }
}
```

`new Thread(lambda)` yalnız bir nesne oluşturur; henüz hiçbir iş parçacığı çalışmaz ve nesnenin durumu `NEW`'dur. `t.start()` yeni bir işletim sistemi iş parçacığı başlatır ve lambda o iş parçacığında çalışır. `main` bu sırada yazmaya devam eder; iki iş parçacığı aynı anda çalışabilir. `t.join()` çağrısında `main`, `t` bitene kadar `WAITING` durumunda bekler. `t` döngüyü bitirince `toplam` 15 olur ve `t`'nin durumu `TERMINATED`'a geçer.

Satır	main	t
`new Thread(…)`	RUNNABLE	NEW (oluşturuldu, başlamadı)
`t.start()`	RUNNABLE	RUNNABLE
`println("main: bekliyor")`	RUNNABLE	RUNNABLE

Satır	main	t
`t.join()`	WAITING (join)	RUNNABLE
(t döngüyü bitirir)	uyanıyor	TERMINATED
`println("toplam = " + toplam)`	RUNNABLE	TERMINATED

Tablo 4.12: Java programındaki iş parçacıklarının `Thread.State` değerleri.

Program JDK 21'in `javac` ve `java` komutlarıyla derlenip çalıştırıldı. İki satırlık çıktısı her seferinde aynıdır: önce `main: bekliyor`, sonra `toplam = 15`. Durumları gözlemlemek için programa `t.getState()` yazdıran iki satır eklendiğinde de `start`'tan önce `NEW`, join'den sonra `TERMINATED` görüldü.

`toplam` değişkeni ne `volatile` ne de bir kilitle korunuyor; buna rağmen `main`'in 15'i görmesi garantidir. Java bellek modeli, bir iş parçacığının yaptığı bütün yazmaların, o iş parçacığını join ile bekleyen iş parçacığına görünür olacağını söyler. Bu ilişkiye **önce-olur** (*happens-before*) ilişkisi denir. Join, beklemenin yanında görünürlüğü de sağlar.

> **Sık yapılan hata**
>
> `start()` yerine `run()` çağırmak. `t.run()` sıradan bir metot çağrısıdır: lambda yeni bir iş parçacığında değil, çağıran iş parçacığının, yani `main`'in içinde çalışır. Program doğru sonucu verir ama hiçbir eş zamanlılık yoktur. `Thread.currentThread().getName()` yazdırıldığında fark hemen görülür: `run()` içinde ad `main`'dir. Bir `Thread` nesnesi de yalnız bir kez başlatılabilir; ikinci `start()` çağrısı `IllegalThreadStateException` fırlatır (JDK 21 ile denendi).

4.4.8 Java 21: sanal iş parçacıkları

Sanal iş parçacıklarının neden önemli olduğunu göstermek için 10.000 görev oluşturan bir program yazalım. Her görev kendi sanal iş parçacığında bir saniye uyur; uyku, bir ağ yanıtını beklemenin yerine geçer.

```
import java.time.Duration;
import java.util.concurrent.*;
import java.util.stream.IntStream;

public class Sanal {
  public static void main(String[] args) {
    try (var ex = Executors.newVirtualThreadPerTaskExecutor()) {
      IntStream.range(0, 10_000).forEach(i ->
        ex.submit(() -> {
          Thread.sleep(Duration.ofSeconds(1));
          return i;
        }));
    }  // close(): tüm görevleri bekler
    System.out.println("bitti");
  }
}
```

`Executors.newVirtualThreadPerTaskExecutor()` her gönderilen görev için yeni bir sanal iş parçacığı oluşturan bir yürütücü döndürür. `IntStream.range(0, 10_000)` 0'dan 9999'a kadar sayıları üretir ve her biri için `ex.submit` bir görev gönderir. Görev `Thread.sleep` ile bir saniye uyur ve numarasını döndürür. `try` bloğu kaynaklı biçimde (*try-with-resources*) yazılmıştır: blok bitince yürütücünün `close()` metodu kendiliğinden çağrılır ve bu metot gönderilen bütün görevler bitene kadar bekler. Son olarak `bitti` yazdırılır.

Program sıralı çalışsaydı 10.000 saniye, yani yaklaşık 2 saat 47 dakika sürerdi. İki farklı makinede JDK 21 ile yapılan ölçümlerde bütün görevler yaklaşık 2 saniyede ve 1,2 saniyede bitti. İkinci makinede 12 mantıksal çekirdek vardı ve görevler 12 taşıyıcı iş parçacığı üzerinde çalıştı. Ölçülen süre makineye göre değişir, ama büyüklük mertebesi değişmez: on bin görevin her biri beklerken taşıyıcısını bırakır, birkaç taşıyıcı bütün görevlere yeter.

Aynı programı 10.000 platform iş parçacığıyla çalıştırmak, 10.000 çekirdek iş parçacığı ve 10.000 yığın demektir. Bu çoğu sistemde çalışır, ama bellek ve çekirdek kaynağı açısından pahalıdır; görev sayısı yüz binlere çıkınca sistem sınırlarına takılır. Sanal iş parçacıklarının uygun olduğu alan, çok sayıda G/Ç bekleyen görevdir: binlerce eş zamanlı bağlantıya hizmet veren sunucular. İşlemci yoğun işte kazanç yoktur, çünkü o durumda darboğaz çekirdek sayısıdır. Sanal iş parçacıkları klasik yolla da oluşturulabilir: `Thread.ofVirtual().start(r)` sanal bir iş parçacığı başlatır; `Thread.ofPlatform()` ise platform iş parçacığı oluşturan bir kurucu döndürür.

4.5 Örtük iş parçacığı

4.5.1 Görevi belirtmek, iş parçacığını çalışma zamanına bırakmak

İş parçacığı sayısı yüzlere ya da binlere çıkınca her birini elle oluşturmak, beklemek ve senkronize etmek hem zorlaşır hem de hataya açık hâle gelir. **Örtük iş parçacığı** (*implicit threading*) bu yükü programcının üzerinden alır. Programcı yalnız **görevleri**, yani paralel çalışabilecek işleri belirtir. İş parçacıklarını derleyici ve çalışma zamanı kütüphaneleri oluşturur, görevleri onlara dağıtır ve yönetir. Tablo 4.13 yaygın yaklaşımları listeler.

Yaklaşım	Nerede	Fikir
İş parçacığı havuzu	Java `ExecutorService`, Windows iş parçacığı havuzu	Hazır iş parçacıkları kuyruktaki görevleri alır.
Fork-join	Java `ForkJoinPool`, paralel akışlar	Görevi böl, parçaları paralel çöz, sonuçları birleştir.
OpenMP	C, C++, Fortran	Derleyici yönergeleriyle paralel bölgeler.
Grand Central Dispatch	macOS, iOS (libdispatch)	Bloklar dağıtım kuyruklarına konur.
oneTBB (eski adı Intel TBB)	C++ şablon kütüphanesi	`parallel_for`, görev çizelgeleme, iş çalma.

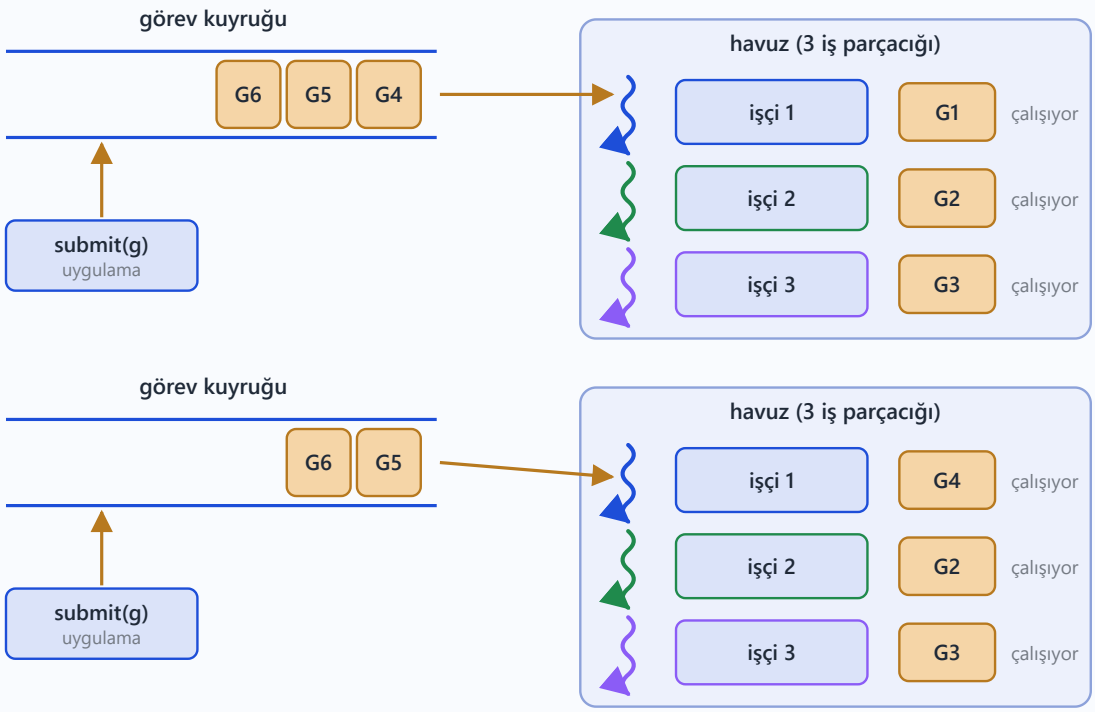
Tablo 4.13: Örtük iş parçacığı yaklaşımları.

4.5.2 İş parçacığı havuzu

Çok iş parçacıklı web sunucusunun her istek için yeni bir iş parçacığı oluşturduğunu düşünelim. Bu tasarımın iki sorunu vardır. Birincisi, her istekte iş parçacığı oluşturup sonra yok etmenin maliyetidir; iş kısa sürüyorsa bu maliyet işin kendisinden büyük olabilir. İkincisi, eş zamanlı iş parçacığı sayısına bir sınır yoktur: ani bir istek dalgası binlerce iş parçacığı oluşturup belleği ve işlemciyi tüketebilir.

Tanım 4.4

İş parçacığı havuzu (*thread pool*), başlangıçta oluşturulan ve kuyruğa konan görevleri sırayla alıp çalıştıran sabit (ya da sınırlı) sayıda iş parçacığıdır. Görevini bitiren iş parçacığı yok olmaz; kuyruktaki bir sonraki görevi alır.



Şekil 4.26: Üç iş parçacıklı havuz. (a) İşçiler kuyruktan G1, G2 ve G3'ü almış ve çalıştırıyor; kuyrukta G4, G5 ve G6 bekliyor. (b) İşçi 1, G1'i bitirince yok olmamış, sıradaki G4'ü almıştır.

Havuz başlangıçta sabit sayıda iş parçacığı oluşturur; uygulama görevleri kuyruğa koyar. Boştaki işçiler kuyruktan görev alır; yeni iş parçacığı oluşturma maliyeti ödenmez. İşçi bir görevi bitirince hemen sıradakini alır. Kuyruk boşalınca işçiler bir sonraki görev gelene kadar bekler. Eş zamanlı çalışan iş parçacığı sayısı hiçbir zaman havuz boyutunu aşmaz. Havuzun faydaları üç maddede toplanır:

- **Daha hızlı hizmet.** Hazır bir iş parçacığı görevi hemen alır; oluşturma maliyeti bir kez ödenmiştir.
- **Sistemi korur.** Havuz boyutu, aynı anda var olabilecek iş parçacığı sayısına üst sınır koyar. İstek dalgası iş parçacığı sayısını değil, kuyruk uzunluğunu artırır.
- **Görevi çalıştırma biçiminden ayırır.** Görev yalnız ne yapılacağını söyler. Gecikmeli mi, periyodik mi, hangi öncelikle çalışacağına havuz karar verir.

Havuz boyutunu seçmek için kaba bir kural kullanılır. İşlemci yoğun görevlerde boyut çekirdek sayısı kadar olmalıdır; Java'da bu sayı `Runtime.getRuntime().availableProcessors()` ile öğrenilir. Fazlası paralellik kazandırmaz, yalnız bağlam değiştirme ve önbellek çekişmesi ekler. G/Ç yoğun görevlerde iş parçacıklarının çoğu beklediği için daha büyük bir havuz gerekir. Bir görevin bekleme süresi B, hesap süresi H ise yaklaşık $N_{iş} \approx N_{çekirdek} \times (1 + B / H)$ iş parçacığı çekirdekleri dolu tutar.

Örnek 4.6

Dört çekirdekli bir makinede çalışan bir görev 10 ms hesap yapıyor, sonra 30 ms bir uzak servisin yanıtını bekliyor. Havuz kaç iş parçacıklı olmalıdır?

1. Bekleme/hesap oranı B / H = 30 / 10 = 3'tür.
2. $N_{iş} \approx 4 \times (1 + 3) = 16$.
3. Yorum: her çekirdek için dört iş parçacığı vardır. Bir iş parçacığı 10 ms hesap yapıp 30 ms beklerken öteki üçü sırayla aynı çekirdeği kullanır ve çekirdek sürekli dolu kalır. Bu sayı bir başlangıç noktasıdır; gerçek değer ölçümle ayarlanır. Çok büyük havuz bellek tüketir, çok küçük havuz işlemciyi boş bırakır.

4.5.3 Java `ExecutorService`

Java'nın `java.util.concurrent` paketi havuzları `ExecutorService` arayüzüyle sunar. Aşağıdaki kod parçası dört görevi iki iş parçacıklı bir havuza verir; her görev kendi numarasının karesini hesaplar.

```
ExecutorService havuz = Executors.newFixedThreadPool(2);
List<Future<Integer>> sonuclar = new ArrayList<>();
for (int g = 1; g <= 4; g++) {
  final int n = g;
  sonuclar.add(havuz.submit(() -> n * n));
}
int toplam = 0;
for (Future<Integer> f : sonuclar)
  toplam += f.get();      // sonucu bekler
System.out.println("toplam = " + toplam);
havuz.shutdown();
```

`Executors.newFixedThreadPool(2)` iki iş parçacıklı sabit bir havuz oluşturur. Döngü `g` = 1, 2, 3, 4 için birer görev gönderir. Lambda içinde döngü değişkeni doğrudan kullanılamaz, çünkü Java lambdaların yakaladığı yerel değişkenlerin sonradan değişmemesini şart koşar; bu yüzden her turda `final int n = g;` ile yeni bir kopya alınır. `submit` görevi kuyruğa koyar ve hemen bir `Future` nesnesi döndürür. `Future`, henüz hazır olmayabilecek bir sonucun yer tutucusudur. İkinci döngüde `f.get()` her görevin sonucunu, hazır değilse hazır olana kadar bekleyerek alır. Son satırdaki `shutdown()` havuzun yeni görev kabul etmesini durdurur; çalışan görevler biter ve iş parçacıkları sonlanır.

Görev	`n`	`f.get()`	`toplam`
1	1	1	1
2	2	4	5
3	3	9	14
4	4	16	30

Tablo 4.14: `Future` sonuçlarının toplanması. Görevlerin hangi işçide ve hangi sırayla çalıştığı değişebilir; toplama sırası değişmez.

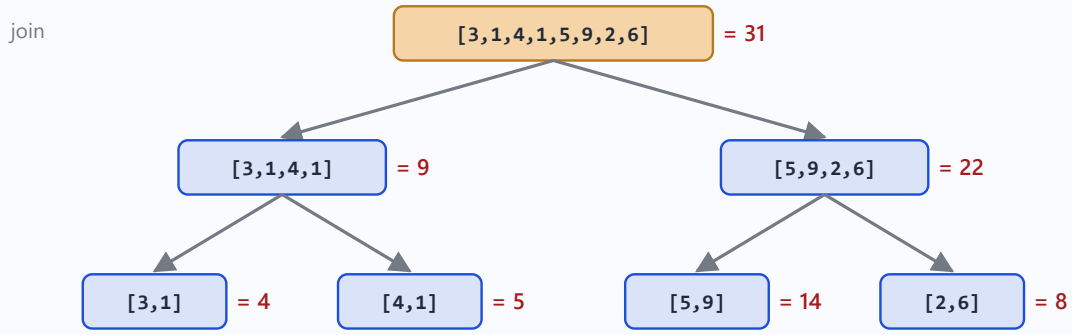
JDK 21 ile çalıştırıldığında program şunu yazdırır:

```
toplam = 30
```

Sonuç deterministiktir: görevler iki işçiye hangi sırayla dağılırsa dağılsın, `get()` çağrıları listedeki sırayla yapılır ve her biri kendi görevinin sonucunu döndürür. `Executors` sınıfının başka fabrika metotları da vardır. `newCachedThreadPool` gerektikçe yeni iş parçacığı oluşturur ve boşta kalanları bir süre sonra kapatır. `newScheduledThreadPool` gecikmeli ve periyodik görevler için kullanılır. `newVirtualThreadPerTaskExecutor` ise her görev için yeni bir sanal iş parçacığı oluşturur (4.4.8).

4.5.4 Fork-join

Fork-join, böl ve fethet (*divide and conquer*) algoritmalarının paralel karşılığıdır. Görev bir eşikten büyükse ikiye bölünür; parçalar ayrı görevler olarak çatallanır (fork) ve havuzdaki farklı iş parçacıklarında çalışabilir. Eşiğe inen küçük görevler doğrudan, sıralı olarak hesaplanır. Ebeveyn görev çocuklarını bekler (join) ve sonuçlarını birleştirir. Şekil 4.27 sekiz sayının toplamını bu yöntemle hesaplar.



Şekil 4.27: [3, 1, 4, 1, 5, 9, 2, 6] dizisinin fork-join ile toplanması. Kök görev [3,1,4,1] ve [5,9,2,6] olarak ikiye çatallanır; bunlar da ikişer öğelik yapraklara bölünür. Yapraklar doğrudan hesaplanır (4, 5, 14, 8), join adımları 4 + 5 = 9 ve 14 + 8 = 22'yi, kök de 9 + 22 = 31'i verir.

Yöntemin üç aşaması şekilden okunabilir. Önce büyük görev, eşikten büyük olduğu için bölünür. Fork adımında iki alt görev oluşturulur; bunlar havuzdaki farklı iş parçacıklarında çalışabilir. Alt görevler yine bölünür ve eşiğe (iki öğe) inen yapraklar doğrudan hesaplanır: [3,1] = 4, [4,1] = 5, [5,9] = 14, [2,6] = 8. Join adımında her ebeveyn çocuklarını bekler ve birleştirir: 4 + 5 = 9, 14 + 8 = 22. Kökte sonuç 9 + 22 = 31'dir. Fork-join, senkron iş parçacığı kullanımının tipik örneğidir: her ebeveyn çocuklarının sonucunu bekler.

4.5.5 `ForkJoinPool` ve iş çalma

Java, fork-join için `java.util.concurrent` paketinde `ForkJoinPool` havuzunu ve `RecursiveTask<V>` sınıfını sunar. Sonuç döndüren bir görev `RecursiveTask`'tan türetilir ve `compute()` metodunu yazar. Aşağıdaki sınıf bir dizinin `[bas, son)` aralığını toplar; `a`, `bas`, `son`

alanları ve kurucu kısa olsun diye gösterilmemiştir.

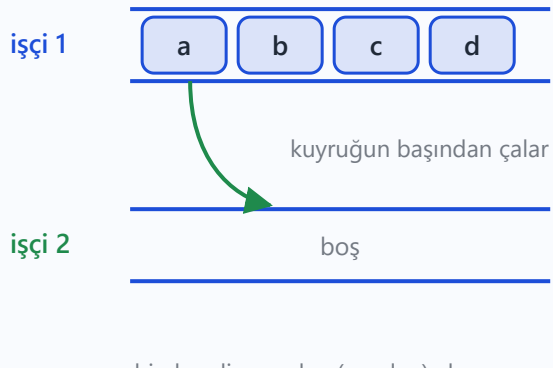
```
class Toplam extends RecursiveTask<Long> {
  protected Long compute() {
    if (son - bas <= ESIK) {      // küçükse doğrudan
      long s = 0;
      for (int i = bas; i < son; i++) s += a[i];
      return s;
    }
    int orta = (bas + son) / 2;
    Toplam sol = new Toplam(a, bas, orta);
    Toplam sag = new Toplam(a, orta, son);
    sol.fork();                   // eş zamansız başlat
    long sagSonuc = sag.compute();
    return sol.join() + sagSonuc;
  }
}
long s = ForkJoinPool.commonPool()
           .invoke(new Toplam(a, 0, a.length));
```

`compute()` önce aralığın uzunluğunu eşikle karşılaştırır. Aralık küçükse sıradan bir döngüyle toplar ve döndürür. Büyükse ortadan ikiye böler ve iki alt görev oluşturur. `sol.fork()` sol yarıyı, o anda çalışan işçinin kuyruğuna koyar ve hemen döner; görev henüz çalışmamıştır. Sağ yarı aynı iş parçacığında, doğrudan `sag.compute()` ile hesaplanır. Son satırda `sol.join()` sol yarının sonucunu bekler ve iki sonuç toplanır. `ForkJoinPool.commonPool()`, JVM'in paylaşılan havuzudur; `invoke` kök görevi başlatır ve sonucunu bekler.

Sağ yarıyı fork etmek yerine aynı iş parçacığında doğrudan çalıştırmak bilinçli bir seçimdir. İki yarı da fork edilseydi iş parçacığı hiçbir şey yapmadan ikisini beklerdi. Böylece her bölünmede yalnız bir görev kuyruğa girer ve iş parçacığı boş durmaz.

1'den 100.000'e kadar sayılarla doldurulmuş bir dizi için program JDK 21'de `toplam = 5000050000` yazdırır. Bu değer *100.000 × 100.001 / 2*'ye eşittir. 1000 öğelik bir eşik ve 11 iş parçacıklı ortak havuzla yapılan denemede de sonuç aynıdır.

Havuzun yükü nasıl dengelediği de önemlidir. `ForkJoinPool`'da her işçinin kendi **çift uçlu kuyruğu** (*deque*) vardır. İşçi fork ettiği görevleri kendi kuyruğunun bir ucuna koyar ve yine o uçtan alır; bu, en son bölünen, en küçük görevle devam etmesini sağlar. Kuyruğu boşalan bir işçi boşta beklemez; başka bir işçinin kuyruğunun **öbür ucundan** görev alır. Buna **iş çalma** (*work stealing*) denir. Öbür uçta genellikle en eski, en büyük görevler durur; tek bir çalma işlemi bu yüzden çok iş taşır ve sahibiyle çakışma olasılığı düşüktür.



Şekil 4.28: İş çalma. İşçi 1'in kuyruğunda a, b, c, d görevleri var; sahip kendi ucundan (sondan) alır. İşçi 2'nin kuyruğu boş; işçi 1'in kuyruğunun başından görev çalar.

4.5.6 Kod izleme: fork-join ile toplam

Fork ve join'in tam olarak ne yaptığını görmek için küçük bir diziyi tek işçili bir havuzda izleyelim: `a = {2, 7, 1, 8, 2, 8, 1, 8}`, `ESIK = 2` ve çağrı `new ForkJoinPool(1).invoke(new Toplam(a, 0, 8))`. Tek işçi seçmek izlemeyi deterministik yapar: kimse görev çalmaz. İzlenen metot şudur:

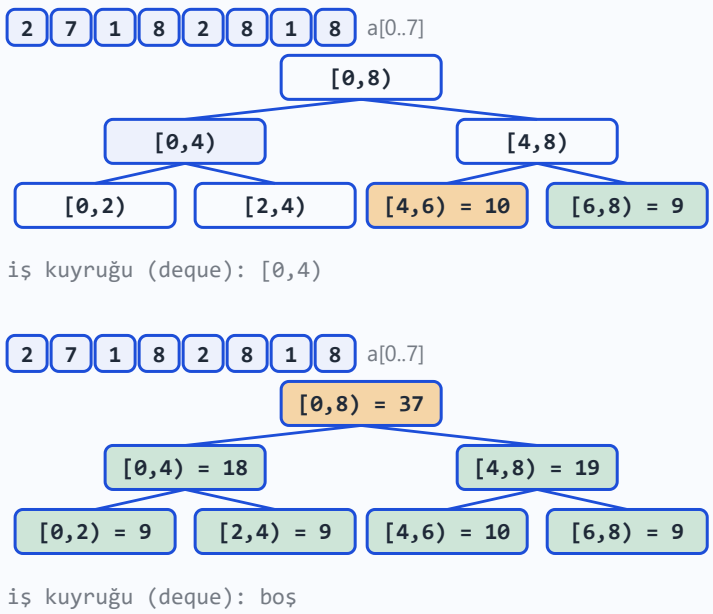
```
protected Long compute() {    // ESIK = 2
  if (son - bas <= ESIK) {
    long s = 0;
    for (int i = bas; i < son; i++)
      s += a[i];
    return s;
  }
  int orta = (bas + son) / 2;
  Toplam sol = new Toplam(a, bas, orta);
  Toplam sag = new Toplam(a, orta, son);
  sol.fork();                    // kuyruğa
  long sagSonuc = sag.compute();
  return sol.join() + sagSonuc;
}
```

Adım	Olay	Kuyruk	Hesaplanan
1	Kök [0,8): 8 − 0 = 8 > ESIK, bölünür	boş	—
2	`sol.fork()` : [0,4) kuyruğa; hemen çalışmaz	[0,4)	—
3	`sag.compute()` : [4,8) aynı iş parçacığında	[0,4)	—
4	[4,8) bölünür: [4,6) kuyruğa	[0,4) [4,6)	—
5	[6,8): 2 öğe, eşikte; doğrudan toplanır	[0,4) [4,6)	[6,8) = 1 + 8 = 9

Adım	Olay	Kuyruk	Hesaplanan
6	`sol.join()` : [4,6) kuyruğun ucunda, kimse çalmamış; işçi onu çıkarıp kendisi çalıştırır	[0,4)	[4,6) = 2 + 8 = 10
7	[4,8) birleşir ve köke döner	[0,4)	[4,8) = 10 + 9 = 19
8	Kök `sol.join()` : [0,4) kuyruktan çıkar ve çalışır; bölünür, [0,2) kuyruğa	[0,2)	—
9	[2,4) yaprak	[0,2)	[2,4) = 1 + 8 = 9
10	join [0,2) yaprağını çalıştırır; [0,4) birleşir	boş	[0,2) = 2 + 7 = 9; [0,4) = 9 + 9 = 18
11	Kök birleşir	boş	[0,8) = 18 + 19 = 37

Tablo 4.15: Tek işçili havuzda fork-join izlemesi. Kuyruk, işçinin çift uçlu kuyruğudur; sağdaki öğe kuyruğun işçiye yakın ucudur.

Program JDK 21 ile çalıştırıldı ve `toplam = 37` yazdırdı. Her görevin bitişinde aralığı ve sonucu yazdıran bir izleme satırı eklendiğinde görevler tam olarak tablodaki sırayla bitti: [6,8) = 9, [4,6) = 10, [4,8) = 19, [2,4) = 9, [0,2) = 9, [0,4) = 18, [0,8) = 37. Toplam yedi görev oluşur; dördü yapraktır.



Şekil 4.29: Fork-join izlemesinin iki anı. (a) 6. adım: [6,8) = 9 ve [4,6) = 10 hesaplanmış; kuyrukta yalnız [0,4) kalmış. (b) Son: bütün düğümler hesaplanmış, kök [0,8) = 37, kuyruk boş.

Havuzda boşta bir işçi daha olsaydı 2. adımda kuyruğa giren [0,4) görevini çalabilirdi. O durumda [0,4) ile [4,8) gerçekten paralel hesaplanır ve görevlerin bitiş sırası değişirdi. Sonuç ise değişmezdi: her düğüm yine iki çocuğunun toplamıdır. Çok iş parçacıklı programlarda bu ayrım sık karşımıza çıkar: **yürütme sırası belirlenimci olmayabilir, ama doğru senkronize edilmiş bir programın sonucu belirlenimcidir**.

4.5.7 OpenMP

OpenMP, C, C++ ve Fortran için paylaşımlı bellekli sistemlerde paralel programlamayı kolaylaştıran bir derleyici yönergeleri (*compiler directives*) ve kütüphane kümesidir. Programcı paralel çalışabilecek bölgeleri `#pragma omp` ile işaretler; iş parçacıklarını derleyici ve OpenMP çalışma zamanı oluşturur.

```
#include <omp.h>
#include <stdio.h>

int main(void) {
  #pragma omp parallel
  printf("merhaba, ben %d\n", omp_get_thread_num());

  double a[1000], b[1000], c[1000], s = 0;
  /* ... a ve b doldurulur ... */
  #pragma omp parallel for reduction(+:s)
  for (int i = 0; i < 1000; i++) {
    c[i] = a[i] + b[i];
    s += c[i];
  }
  return 0;
}
```

`#pragma omp parallel` bir **paralel bölge** açar: yönergeyi izleyen deyim, bir iş parçacığı ekibinin her üyesinde ayrı ayrı çalışır. Ekipteki iş parçacığı sayısı varsayılan olarak çekirdek sayısı kadardır; `OMP_NUM_THREADS` ortam değişkeniyle ya da yönergeye eklenen `num_threads(n)` ile değiştirilir. Bu örnekte bölge tek bir `printf` deyimidir; her iş parçacığı `omp_get_thread_num()` ile kendi numarasını alır ve bir "merhaba" satırı yazdırır. Satırların sırası her çalıştırmada değişebilir. Bölgenin sonunda örtük bir bariyer vardır: bütün iş parçacıkları bitmeden program devam etmez.

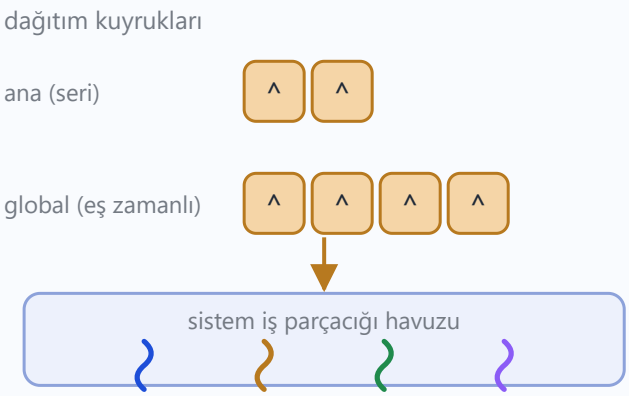
İkinci yönerge `#pragma omp parallel for`, ardından gelen `for` döngüsünün yinelemelerini iş parçacıklarına böler. Dört iş parçacığıyla her biri yaklaşık 250 yinelemeyi alır. Döngüdeki `s += c[i]` satırı ise sorunludur: `s` bütün iş parçacıklarınca paylaşılan tek bir değişkendir ve birden çok iş parçacığının aynı anda ona eklemesi bir yarış durumu yaratır. `reduction(+:s)` kısmı bunu çözer: her iş parçacığına `s`'nin özel bir kopyası verilir, kopyalar döngü bitince toplanıp asıl `s`'ye eklenir. Program `gcc -fopenmp` ile derlenir; bu seçenek verilmezse yönergeler yok sayılır ve program sıralı çalışır.

4.5.8 Grand Central Dispatch ve oneTBB

Grand Central Dispatch (GCD), Apple'ın macOS ve iOS için geliştirdiği bir çalışma zamanı kütüphanesi (libdispatch) ile C ve Objective-C dillerine eklenen bir dil özelliğinin birleşimidir. Dil özelliği **bloklardır**: `^{ printf("Ben bir bloğum"); }` gibi, şapka işaretiyle başlayan ve süslü parantez içinde yazılan bağımsız iş birimleri.

```
dispatch_queue_t q = dispatch_get_global_queue(
                        QOS_CLASS_USER_INITIATED, 0);
dispatch_async(q, ^{ puts("Ben bir bloğum"); });
```

Bloklar **dağıtım kuyruklarına** (*dispatch queues*) konur. Seri kuyruk blokları sırayla, birer birer çalıştırır; her uygulamanın ana kuyruğu seridir ve kullanıcı arayüzü işleri orada yapılır. Eş zamanlı (global) kuyruk ise blokları paralel çalıştırır. Global kuyruklar **hizmet kalitesi** (*quality of service*, QoS) sınıflarına ayrılır; örnekteki `QOS_CLASS_USER_INITIATED`, kullanıcının başlattığı ve sonucunu beklediği işler içindir. `dispatch_async` bloğu kuyruğa koyar ve hemen döner. Kuyruktan çıkan blok, sistemin yönettiği havuzdaki boş bir iş parçacığına atanır. İş parçacığı yönetimi tamamen GCD'dedir; programcı hiçbir iş parçacığı oluşturmaz.



Şekil 4.30: Grand Central Dispatch. Bloklar (^) ana (seri) kuyruğa ya da global (eş zamanlı) kuyruğa konur; sistem iş parçacığı havuzundaki iş parçacıkları kuyruklardan blok alır.

oneTBB, Intel Threading Building Blocks kütüphanesinin açık kaynak devamıdır ve C++ şablonları olarak gelir. `tbb::parallel_for(0, n, [&](int i) { f(v[i]); });` gibi bir çağrı, döngünün aralığını parçalara böler ve parçaları kendi çizelgeleyicisiyle, iş çalma kullanarak iş parçacıklarına dağıtır. Programcı yalnız "bu döngünün yinelemeleri birbirinden bağımsızdır" der; kaç iş parçacığı kullanılacağına ve parçaların büyüklüğüne kütüphane karar verir. Apple'ın Swift dilinde ise GCD'nin yerini büyük ölçüde dilin kendi `async` / `await` düzeneği almıştır.

4.6 İş parçacığı sorunları

İşletim sisteminin pek çok arayüzü, iş parçacıklarından önce, her sürecin tek akışlı olduğu varsayımıyla tasarlandı. `fork()`, sinyaller, global `errno` değişkeni ve statik ara bellek kullanan kütüphane fonksiyonları bunun örnekleridir. Bir süreçte birden çok akış olunca bu arayüzlerin her biri için yeni bir soru doğar.

4.6.1 İş parçacıklı süreçte `fork()` ve `exec()`

Üç iş parçacıklı bir süreçte iş parçacıklarından biri `fork()` çağırırsa çocuk süreçte kaç iş parçacığı olmalıdır? Bazı UNIX sistemleri bu soruya iki cevap verip iki ayrı `fork` sürümü sundu: biri bütün iş parçacıklarını, öteki yalnız çağıranı kopyalıyordu. POSIX'in cevabı tektir: **yalnız `fork`'u çağıran iş parçacığı kopyalanır**. Adres alanının tamamı kopyalanır, ama iş parçacıklarından yalnız biri.



Şekil 4.31: Üç iş parçacıklı ebeveyn süreçte T2 `fork()` çağırır. POSIX'e göre çocuk süreçte yalnız T2'nin kopyası (T2′) bulunur. T1'in o sırada tuttuğu bir kilit çocukta sonsuza dek kilitli kalabilir.

`fork`'tan hemen sonra `exec` çağrılacaksa bu davranış doğrudur: yeni program bütün adres alanını zaten değiştirir ve tek bir iş parçacığıyla başlar. Öteki iş parçacıklarını kopyalamak boşuna iş olurdu. Sorun, çocuğun `exec` çağırmadan çalışmaya devam ettiği durumdadır. Kopyalanan adres alanında, ebeveynin öteki iş parçacıklarının `fork` anında tuttuğu kilitler de kilitli olarak bulunur. Örneğin T1 o anda `malloc`'un iç kilidini tutuyorsa, çocukta bu kilidi açacak bir T1 yoktur ve çocuğun ilk `malloc` çağrısı sonsuza dek bekler.

Bu yüzden POSIX, çok iş parçacıklı bir sürecin çocuğunda `exec`'e kadar yalnız **async-signal-safe** fonksiyonların (sinyal işleyicisinden güvenle çağrılabilen, iç kilit kullanmayan fonksiyonlar) çağrılmasına izin verir. `pthread_atfork`, `fork`'tan önce ve sonra çalışacak işleyiciler kaydederek kütüphanelerin kilitlerini düzenli bir duruma getirmesine yarar. Amaç yeni bir program başlatmaksa, çoğu zaman `fork` ile `exec`'i tek adımda yapan `posix_spawn` daha iyi bir seçimdir.

4.6.2 Sinyal işleme

Sinyal, bir sürece bir olayın olduğunu bildiren yazılım bildirimidir. Tek iş parçacıklı süreçte sinyalin nereye gideceği bellidir. Çok iş parçacıklı süreçte ise soru şudur: sinyal hangi iş parçacığına teslim edilmeli? Sinyalin türüne göre dört seçenek vardır.

Seçenek	Örnek
Sinyalin ilgili olduğu iş parçacığına	senkron sinyaller: `SIGSEGV`, `SIGFPE`
Süreçteki her iş parçacığına	bazı sonlandırma durumları
Sinyali engellemeyen herhangi bir iş parçacığına	POSIX süreç sinyali: `SIGINT`

Seçenek	Örnek
Bütün sinyalleri alan özel bir iş parçacığına	`sigwait` döngüsü

Tablo 4.16: Çok iş parçacıklı süreçte sinyal teslimi seçenekleri.

Senkron sinyaller, bir iş parçacığının kendi yaptığı bir işlemden doğar: geçersiz bellek erişimi `SIGSEGV`, sıfıra bölme `SIGFPE` üretir. Bu sinyaller doğal olarak hatayı yapan iş parçacığına gider. Eş zamansız sinyaller ise dışarıdan gelir: kullanıcı Ctrl+C'ye bastığında süreç `SIGINT` alır. POSIX böyle bir süreç sinyalini, o sinyali engellemeyen herhangi bir iş parçacığına teslim eder.

Burada iki ayrım önemlidir. Sinyal **işleyicileri** süreç geneldir: bir sinyale karşı hangi fonksiyonun çalışacağı bütün iş parçacıkları için aynıdır. Sinyal **maskesi** ise iş parçacığına özeldir: her iş parçacığı hangi sinyalleri engellediğini `pthread_sigmask` ile kendisi belirler. Belirli bir iş parçacığına sinyal göndermek için `pthread_kill(tid, sig)` kullanılır.



Şekil 4.32: `SIGINT` (Ctrl+C) sürece gelir. T1 ve T2 sinyali maskeyle engellemiştir; sinyal yalnız engellemeyen ve `sigwait()` ile bekleyen T3'e teslim edilir.

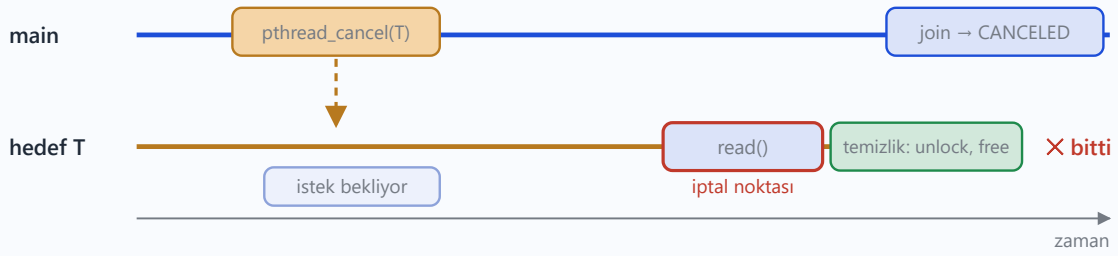
Çok iş parçacıklı programlarda iyi uygulama, şekildeki düzendir. Bütün iş parçacıkları süreç sinyallerini engeller; tek bir iş parçacığı `sigwait` ile bu sinyalleri bir döngüde bekler. Sinyal böylece eş zamansız bir işleyici yerine sıradan bir fonksiyon dönüşü olarak işlenir ve işleyicinin kısıtlarından (yalnız async-signal-safe fonksiyonlar) kurtulunur. Kullanıcı düzeyi iş parçacıklarında bu inceliklerin hiçbiri mümkün değildir: çekirdek yalnız süreci tanır ve doğru iş parçacığını adresleyemez. Windows'ta sinyal kavramı yoktur; benzer bir görevi eş zamansız yordam çağrıları (*asynchronous procedure call*, APC) görür.

4.6.3 İş parçacığı iptali

Bir iş parçacığını işi bitmeden sonlandırmak gerekebilir. Bir veritabanında paralel arama yapan birkaç iş parçacığından biri sonucu bulunca ötekiler gereksiz hâle gelir. Tarayıcıda “dur” düğmesine basıldığında sayfayı yükleyen iş parçacıkları durmalıdır. Sonlandırılacak iş parçacığına **hedef iş parçacığı** denir. İptalin iki biçimi vardır.

Eş zamansız iptal
: Hedef hemen sonlandırılır. Hedef o anda bir kilit tutuyor ya da bir veri yapısını yarım güncellemiş olabilir; kilit hiç açılmaz, veri tutarsız kalır. Bu yüzden tehlikelidir.

Ertelenmiş iptal (varsayılan)
: Hedef, iptal isteğini belirli **iptal noktalarında** denetler ve istek varsa düzenli biçimde sonlanır.



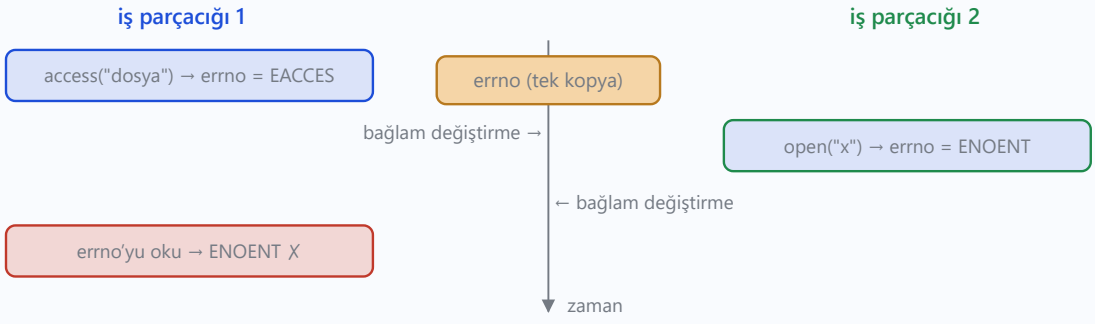
Şekil 4.33: Ertelenmiş iptal. `main` `pthread_cancel(T)` çağırır; istek bekletilir. T bir iptal noktasına (`read()`) gelince temizlik işleyicileri çalışır (kilit açılır, bellek geri verilir), T sonlanır ve join `PTHREAD_CANCELED` döndürür.

Ertelenmiş modda `pthread_cancel(T)` yalnız bir istektir; T çalışmaya devam eder. T bir iptal noktasına geldiğinde istek işlenir. İptal noktaları, POSIX'in belirlediği ve genellikle bloke edebilen çağrılardır: `read` , `sleep` , `pthread_cond_wait` ve benzerleri. Uzun bir hesap döngüsü hiç iptal noktasından geçmiyorsa, programcı döngüye `pthread_testcancel()` çağrısı ekleyerek yapay bir iptal noktası koyar. İptal işlenirken önce `pthread_cleanup_push` ile kaydedilmiş temizlik işleyicileri, kayıt sırasının tersiyle çalışır: kilitler açılır, ayrılmış bellek geri verilir. Sonra T sonlanır ve onu join ile bekleyen iş parçacığı dönüş değeri olarak `PTHREAD_CANCELED` alır.

Java'da iptalin karşılığı işbirlikçi `interrupt()` düzeneğidir. `t.interrupt()` hedefe bir kesme bayrağı koyar; hedef bayrağı `Thread.interrupted()` ile denetler ya da bloke eden bir çağrıdan `InterruptedException` ile çıkar. Eş zamansız iptalin Java'daki karşılığı olan `Thread.stop()` , JDK 20'den beri çalışmaz ve çağrıldığında istisna fırlatır.

4.6.4 Global değişken çakışması

Tek iş parçacıklı programlar için tasarlanmış global değişkenler, iş parçacıkları arasında çakışır. En bilinen örnek UNIX'in `errno` değişkenidir. Bir sistem çağrısı başarısız olunca hata kodunu `errno` 'ya yazar; program çağrıdan sonra bu değişkeni okuyarak hatanın nedenini öğrenir.



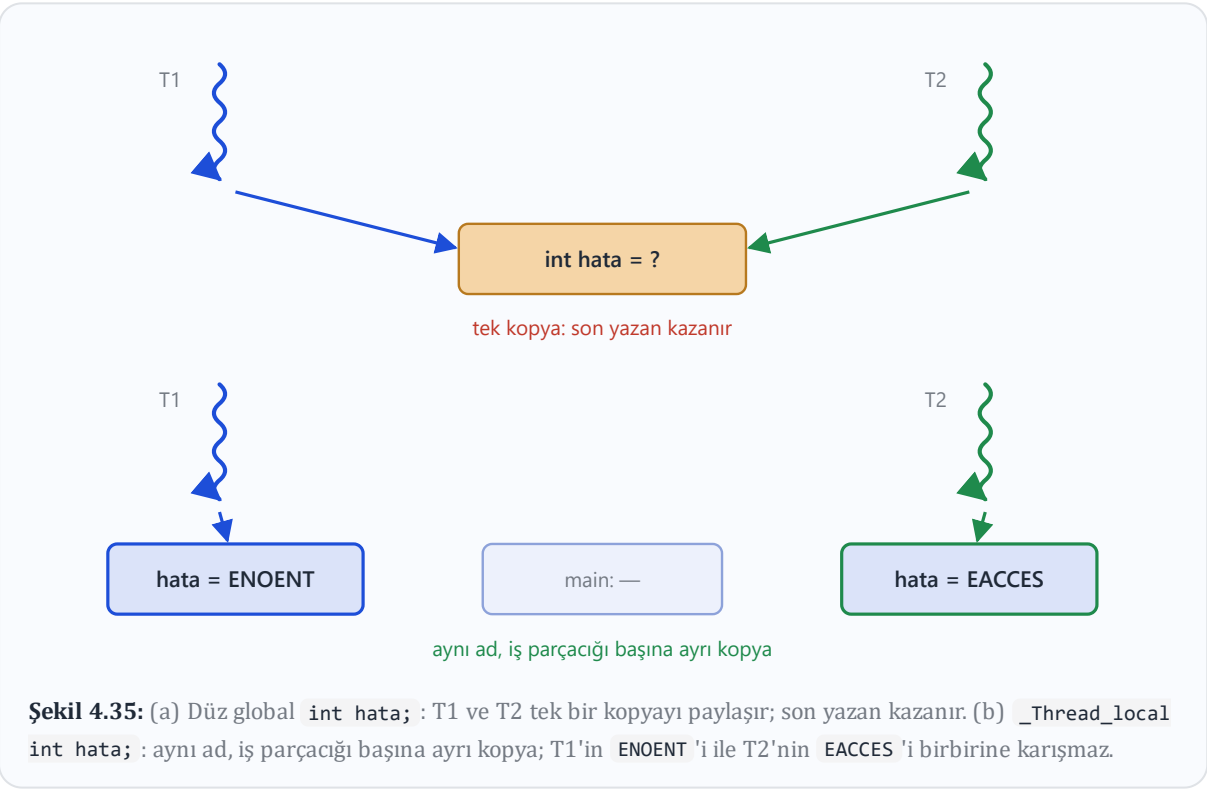
Şekil 4.34: Global `errno` çakışması. İş parçacığı 1 `access("dosya")` çağırır, `errno = EACCES` olur. Bağlam değişir; iş parçacığı 2 `open("x")` çağırır ve `errno` 'nun üzerine `ENOENT` yazar. İş parçacığı 1 yeniden çalıştığında yanlış değeri (`ENOENT`) okur.

Sorunun iki çözümü vardır: değişkeni yasaklamak ya da her iş parçacığına kendi kopyasını vermek. İlki eski kodla uyumsuzdur; bu yüzden ikincisi seçildi. Bugünkü C kütüphanelerinde `errno` bir makrodur ve iş parçacığına yerel bir değişkene karşılık gelir. Her iş parçacığı kendi `errno` 'sunu görür.

4.6.5 İş parçacığına yerel depolama

> **Tanım 4.5**
>
> **İş parçacığına yerel depolama** (*thread-local storage*, TLS), aynı adla erişilen ama her iş parçacığında ayrı bir kopyası bulunan değişkenlerdir. Bir iş parçacığının TLS değişkenine yazdığını öteki iş parçacıkları görmez.

TLS'nin yerel değişkenden farkı ömrüdür. Yerel değişken yalnız tanımlandığı fonksiyon çalışırken yaşar ve başka fonksiyonlar ona erişemez. TLS değişkeni ise iş parçacığının ömrü boyunca, fonksiyon çağrıları boyunca yaşar: global bir değişken gibidir, ama iş parçacığına özeldir. C11'de `_Thread_local`, C23 ve C++'ta `thread_local`, GCC'de `__thread` anahtar sözcüğüyle tanımlanır. Pthreads aynı işi fonksiyon çağrılarıyla yapar: `pthread_key_create` bir anahtar oluşturur, `pthread_setspecific` ve `pthread_getspecific` o anahtarın çağıran iş parçacığındaki değerini yazar ve okur. Tipik kullanımlar `errno`, bir işlemin kimliği ve iş parçacığı başına rastgele sayı üreteçidir.



Şekil 4.35: (a) Düz global `int hata;` : T1 ve T2 tek bir kopyayı paylaşır; son yazan kazanır. (b) `_Thread_local int hata;` : aynı ad, iş parçacığı başına ayrı kopya; T1'in `ENOENT` 'i ile T2'nin `EACCES` 'i birbirine karışmaz.

Java'da TLS, `ThreadLocal` sınıfıyla sağlanır. Aşağıdaki program iki iş parçacığında aynı `ThreadLocal` değişkenine farklı değerler yazar ve sonda `main`'in kendi kopyasını okur:

```
public class Yerel {
  static ThreadLocal<Integer> no =
    ThreadLocal.withInitial(() -> 0);
  public static void main(String[] args) throws Exception {
    Thread a = new Thread(() -> { no.set(1); System.out.println("A: " + no.get()); });
    Thread b = new Thread(() -> { no.set(2); System.out.println("B: " + no.get()); });
    a.start(); a.join();
    b.start(); b.join();
    System.out.println("main: " + no.get());
  }
}
```

`ThreadLocal.withInitial(() -> 0)`, her iş parçacığının kopyasının ilk okunduğunda 0 ile başlamasını sağlar. A iş parçacığı kendi kopyasına 1, B iş parçacığı kendi kopyasına 2 yazar. JDK 21 ile çalıştırıldığında çıktı `A: 1`, `B: 2`, `main: 0` olur. `main` hiçbir şey yazmadığı için başlangıç değerini görür. A ile B burada join ile sırayla çalıştırıldığı için satır sırası da her seferinde aynıdır; iki iş parçacığı birlikte başlatılsaydı A ve B satırlarının sırası değişebilirdi, ama değerler değişmezdi.

4.6.6 Yeniden girilir ve iş parçacığı güvenli kod

Kütüphane yordamlarının bir kısmı iç durumlarını statik değişkenlerde saklar. Böyle bir yordama **yeniden girilemez** (*non-reentrant*) denir. Bir iş parçacığı bir iletiyi kütüphanenin ortak ara belleğine koyarken bağlam değişir ve başka bir iş parçacığı aynı ara belleğe yazarsa ilk ileti kaybolur. `malloc` da tarihsel olarak böyleydi: serbest bellek listesini güncellerken listeyi kısa bir süre tutarsız bırakırdı, araya giren iş parçacığı geçersiz bir işaretçi alabilirdi. Bugünkü `malloc` gerçeklemeleri iç kilitler ve iş parçacığı başına bellek alanları (*arenas*) kullanır.

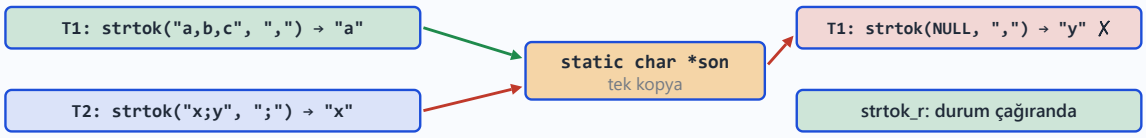
> **Tanım 4.6**
>
> **İş parçacığı güvenli** (*thread-safe*) kod, birden çok iş parçacığından aynı anda çağrılabilir; paylaşılan durumu bir kilitle korur ya da hiç durum tutmaz. **Yeniden girilir** (*reentrant*) kod ise hiç paylaşılan durum kullanmaz; gereken bütün durum çağıranın verdiği parametrelerde ve yerel değişkenlerdedir.

Klasik C kütüphanesindeki pek çok fonksiyonun güvensiz bir sürümü ve sonradan eklenmiş güvenli bir sürümü vardır. Güvenli sürümler genellikle `_r` (*reentrant*) ekiyle biter ve durumu statik bir değişkende değil, çağıranın verdiği bir işaretçide tutar.

Güvensiz	Güvenli sürüm
`strtok`	`strtok_r`

Güvensiz	Güvenli sürüm
`localtime`	`localtime_r`
`rand`	`rand_r` ya da yerel üreteç
`asctime`, `ctime`	`strftime`
Java `SimpleDateFormat`	`DateTimeFormatter`

Tablo 4.17: Güvensiz fonksiyonlar ve güvenli karşılıkları.



Şekil 4.36: `strtok`'un statik işaretçisi yüzünden iş parçacıklarının karışması. T1 `"a,b,c"`'yi parçalamaya başlar ve `"a"`'yı alır. T2 `"x;y"` ile çağırır ve tek kopya olan `static char *son`'u kendi dizgisine yöneltir. T1'in sonraki `strtok(NULL, ",")` çağrısı `"b"` yerine `"y"` döndürür. `strtok_r`'de bu durum çağıranda tutulur.

`strtok`, bir dizgiyi ayırıcılara göre parçalara böler. İlk çağrıda dizgi verilir, sonraki çağrılarda `NULL` verilir ve fonksiyon kaldığı yeri statik `son` işaretçisinden hatırlar. Şekildeki senaryoda T2'nin çağrısı bu işaretçiyi kendi dizgisine taşır; T1 artık kendi dizgisinin geri kalanını değil, T2'nin dizgisinden kalan `"y"`'yi alır. `strtok_r` aynı işaretçiyi çağıranın verdiği bir `char **` parametresinde tutar; her iş parçacığının kendi işaretçisi olduğu için karışma olmaz.

Sık yapılan hata

İş parçacığı güvenli ile yeniden girilir kavramlarını eş anlamlı saymak. Kilitle korunan bir fonksiyon iş parçacığı güvenlidir ama yeniden girilir değildir: aynı iş parçacığı fonksiyonun içindeyken bir sinyal işleyicisinden aynı fonksiyonu yeniden çağırırsa, kilidi zaten kendisi tuttuğu için sonsuza dek bekleyebilir. Yeniden girilir bir fonksiyon ise paylaşılan durum kullanmadığı için her iki durumda da güvenlidir.

4.6.7 Yarış durumu: `sayac++`

Paylaşılan veriye erişimin neden senkronizasyon gerektirdiğini en sade biçimde gösteren örnek, bir sayacı artırmaktır. C ya da Java'da `sayac++` tek bir satırdır, ama makinede üç adımdan oluşur:

```
LOAD  R, sayac
ADD   R, 1
STORE sayac, R
```

Değer bellekten bir yazmaca okunur (`LOAD`), yazmaçta bir artırılır (`ADD`) ve belleğe geri yazılır (`STORE`). İki iş parçacığı bu üç adımı iç içe yürütürse bir artış kaybolabilir.

Tanım 4.7

Yarış durumu (*race condition*), birden çok iş parçacığının paylaşılan veriye eriştiği ve sonucun erişimlerin zamanlamasına bağlı olduğu hatalı durumdur.

Başlangıçta `sayac = 5` olsun; T1 ve T2 sayacı birer kez artırıyor. Beklenen sonuç 7'dir. Tablo 4.18 bir artışın kaybolduğu sıralamayı izler.

Adım	T1	T2	R1	R2	`sayac`
1	`LOAD R1 ← sayac`		5	—	5
2		`LOAD R2 ← sayac`	5	5	5
3		`ADD R2, 1` ; `STORE`	5	6	6
4	`ADD R1, 1` ; `STORE`		6	6	**6**

Tablo 4.18: Kayıp güncelleme. T1 çekirdek 0'da, T2 çekirdek 1'de.

T1 değeri yazmacına okur: R1 = 5. Araya T2 girer (tek çekirdekte bir bağlam değişimiyle, çok çekirdekte ise aynı anda çalışarak) ve o da 5 okur. T2 artırıp yazar: `sayac` = 6. T1 eski değerinden devam eder, 5 + 1 = 6 yazar. T2'nin artışı kaybolmuştur; bu hataya **kayıp güncelleme** (*lost update*) denir. İki iş parçacığının dört adımı *C(4, 2) = 6* farklı sırayla iç içe geçebilir. Python ile bütün sıralamalar denendiğinde ikisinde sonuç 7, dördünde 6 çıkar: bir iş parçacığı ötekinin okuması ile yazması arasına girdiğinde artış kaybolur.



Şekil 4.37: Çözüm: oku-artır-yaz dizisi bir kilitle korunur. T1 kilidi alır, 5 okur, 6 yazar ve kilidi bırakır; sonra T2 kilidi alır, 6 okur, 7 yazar. Kritik bölge sırayla yürür ve sonuç 7'dir. C'de `pthread_mutex_lock` ya da `atomic_fetch_add` , Java'da `synchronized` ya da `AtomicInteger` kullanılır.

Çözüm, oku-artır-yaz dizisini **atomik** yapmaktır: dizi ya bir kilitle korunur ve bir anda yalnız bir iş parçacığı onu yürütür, ya da donanımın tek komutta oku-değiştir-yaz yapan atomik işlemleri kullanılır. Kilitler, atomik komutlar ve kritik bölge problemi senkronizasyonun konusudur; burada önemli olan, sorunun iş parçacıklarının paylaşılan adres alanından doğduğunu görmektir.

Örnek 4.7

Paylaşılan `sayac` 0'dır. Üç iş parçacığı sayacı birer kez artırıyor; her artırma iki adımdır: oku (`r = sayac`) ve yaz (`sayac = r + 1`). Olası son değerler nelerdir?

1. **En büyük değer:** İş parçacıkları sırayla çalışırsa her biri bir öncekinin yazdığını okur: 0 → 1 → 2 → 3.
2. **En küçük değer:** Üçü de yazmadan önce okursa üçü de 0 okur ve üçü de 1 yazar. Sonuç 1'dir.
3. **Ara değer:** İkisi aynı değeri okur, üçüncüsü onlardan birinin yazmasından sonra okursa sonuç 2 olur.
4. **Sayım.** Altı adımın iç içe geçebileceği *6! / (2! · 2! · 2!) = 90* sıralama vardır. Python ile hepsi denendiğinde 6'sında 3, 36'sında 2, 48'inde 1 çıkar. Doğru sonuç olan 3, sıralamaların yalnız %6,7'sinde elde edilir.

4.6.8 Yarışı gözlemlemek

Kayıp güncelleme kuramsal bir olasılık değildir; basit bir Java programıyla gözlenebilir. Aşağıdaki programda iki iş parçacığı paylaşılan `sayac`'ı birer milyon kez artırır.

```
public class Yaris {
  static int sayac = 0;
  public static void main(String[] a) throws Exception {
    Runnable is = () -> {
      for (int i = 0; i < 1_000_000; i++) sayac++;
    };
    Thread t1 = new Thread(is), t2 = new Thread(is);
    t1.start(); t2.start();
    t1.join();  t2.join();
    System.out.println("sayac = " + sayac);
  }
}
```

Beklenen sonuç 2.000.000'dur. Program JDK 21 ile üç kez çalıştırıldığında şu sonuçlar elde edildi:

```
sayac = 1456465
sayac = 1544210
sayac = 1910165
```

Aynı program başka bir makinede JDK 21.0.3 ile beş kez daha çalıştırıldığında 1.024.058, 1.047.114, 1.929.112, 1.055.456, 1.079.458 sonuçları elde edildi. **Bu sayılar her çalıştırmada değişir**; aynı makinede bile iki çalıştırma aynı sonucu vermez. Değişmeyen tek şey, sonucun her seferinde 2.000.000'dan küçük olmasıdır: artışların bir kısmı kaybolur.

Sonuç belirlenimci değildir: çizelgelemeye, çekirdek sayısına ve anında derleyicinin (*just-in-time compiler*, JIT) kodu nasıl derlediğine bağlıdır. JIT, döngüyü sayacı yazmaçta tutacak biçimde derleyebilir ve yazmaları seyrekleştirebilir; bu da kaybolan artış sayısını değiştirir. Bu yüzden böyle bir programın çıktısı adım adım izlenemez; yalnız olası değer aralığı söylenebilir.

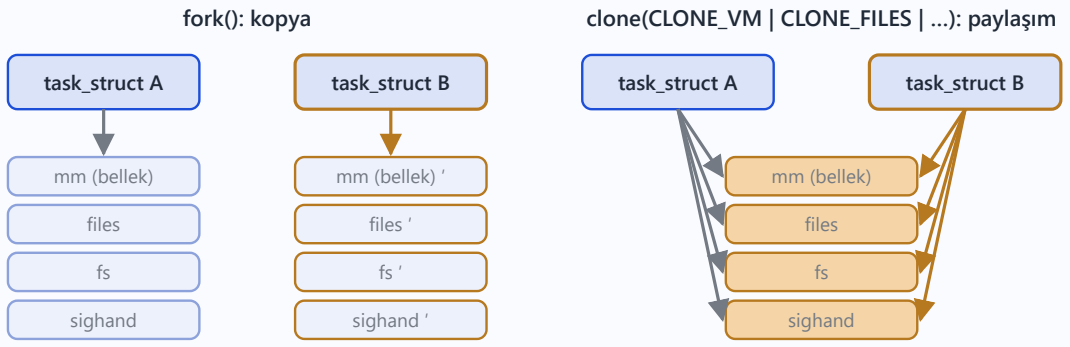
Düzeltmenin üç yolu vardır. Artırma bir `synchronized` blok içine alınabilir, `AtomicInteger.incrementAndGet()` kullanılabilir ya da çok yoğun güncellemeler için `LongAdder` seçilebilir. İlk ikisi JDK 21 ile denendiğinde iki iş parçacığı birer milyon artırma yaptı ve çıktı her seferinde şuydu:

```
synchronized: 2000000  atomik: 2000000
```

En iyi çözüm ise paylaşımı hiç yapmamaktır: her iş parçacığı yerel bir değişkende sayar, sonda sonuçlar birleştirilir. Bu, fork-join'in ve `kismi[k]` örneğinin fikridir; kilit maliyeti de ortadan kalkar.

4.6.9 Linux'ta iş parçacıkları: `clone()`

Linux çekirdeği "süreç" ile "iş parçacığı" arasında ayrım yapmaz. İkisi de bir **görevdir** (*task*) ve aynı `task_struct` yapısıyla temsil edilir. Yeni bir görev `clone()` sistem çağrısıyla oluşturulur; yeni görevin ebeveyniyle neyi paylaşacağını `clone()`'a verilen bayraklar belirler. Hiç paylaşım bayrağı verilmezse sonuç `fork` gibidir: her şeyin kopyası çıkarılır.



Şekil 4.38: Solda `fork()`: B görevi bellek (`mm`), `files`, `fs` ve `sighand` yapılarının kopyalarını alır. Sağda

`clone(CLONE_VM | CLONE_FILES | …)` : A ve B aynı yapıları gösterir.

`CLONE_VM` adres alanını, `CLONE_FS` çalışma dizini ve kök dizin gibi dosya sistemi bilgilerini, `CLONE_FILES` açık dosya tablosunu, `CLONE_SIGHAND` sinyal işleyicilerini paylaştırır. `pthread_create` aşağı yukarı `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD | …)` çağrısına karşılık gelir. `CLONE_THREAD` yeni görevi ebeveynle aynı **iş parçacığı grubuna** koyar. Grubun kimliği (TGID, *thread group ID*), kullanıcıya süreç kimliği olarak görünür: `getpid()` gruptaki herkes için aynı değeri döndürür. `gettid()` ise her görevin kendi kimliğini, yani iş parçacığı kimliğini döndürür. Bu esneklik, Linux'un tek bir düzenekle hem süreçleri hem iş parçacıklarını, hem de ikisinin arasındaki karma durumları (ör. belleği paylaşan ama dosya tablosu ayrı görevler) desteklemesini sağlar.

4.7 Süreç mi, iş parçacığı mı?

İş parçacıklarının avantajları şöyle özetlenir:

- **Duyarlılık:** biri bloke olunca öteki kullanıcı G/Ç'sini işler.
- **Kaynak paylaşımı:** bellek, açık dosyalar ve soketler kendiliğinden ortaktır.
- **Hız:** oluşturma ve geçiş süreçlerden ucuzdur.
- **Donanım paralelliği:** tek bir süreç birçok çekirdeği kullanır.
- **Kolay programlama modeli:** bloke eden çağrılarla bile paralellik sağlanır (web sunucusu); her akış sıralı kod olarak yazılır.

Dezavantajlar ise aynı paylaşımın öbür yüzüdür:

- **Senkronizasyon:** paylaşılan veriye erişim denetlenmelidir; yarış ve kilitlenme riski vardır.
- **Bağımlılık:** aynı süreçteki iş parçacıkları bağımsız değildir; biri çökerse bütün süreç çöker.
- **Bellek koruması yok:** bir iş parçacığı başkasının yığınının üzerine yazabilir.
- **Hata ayıklama ve test zordur:** hatalar zamanlamaya bağlıdır ve her çalıştırmada ortaya çıkmaz.

Seçim, yalıtım ile paylaşım arasındaki dengeye göre yapılır. Yalıtım gerekiyorsa, örneğin tarayıcı sekmeleri birbirinin çökmesinden etkilenmemeli ya da bir güvenlik sınırı çizilmeliyse süreç seçilir. Sıkı veri paylaşımı ve düşük maliyet gerekiyorsa iş parçacığı seçilir. Chrome ikisini birlikte kullanır: her sekme ve uzantı ayrı bir süreçte çalışır, her sürecin içinde de çizim, ağ ve betik yürütme için birden çok iş parçacığı vardır.

4.8 Özet

- **Süreç** kaynakların sahibidir; **iş parçacığı** süreç içindeki bir yürütme akışıdır ve çizelgelemenin temel birimidir. Kod, global veri, heap, açık dosyalar ve sinyal işleyicileri paylaşılır; PC, yazmaçlar, yığın, durum, sinyal maskesi ve TLS her iş parçacığına özeldir. Yığın özeldir ama korunmaz.
- Her iş parçacığının kendi yığını vardır; Linux'ta varsayılan boyut çoğunlukla 8 MiB'dir ve yığınlar arasındaki koruma sayfaları taşmayı `SIGSEGV` ile yakalar.
- Çok iş parçacıklı **web sunucusunda** dağıtıcı istekleri işçilere verir; diskte bloke olan işçi yalnız kendini durdurur. Kazanç, tek işlemcide bile bekleme ile hesaplamanın örtüşmesinden gelir. Olay güdümlü tasarım aynı örtüşmeyi tek iş parçacığıyla sağlar ama durumu elle saklamayı gerektirir.
- Faydalar: duyarlılık, kaynak paylaşımı, ekonomi, ölçeklenebilirlik. İş parçacıkları arası geçişte adres alanı değişmez, TLB ve önbellek sıcak kalır.
- **Eş zamanlılık** görevlerin aynı zaman aralığında ilerlemesidir; **paralellik** aynı anda yürütülmeleridir ve en az iki çekirdek gerektirir. Paralellik eş zamanlılığı gerektirir; tersi doğru değildir.
- **Amdahl yasası:** hızlanma ≤ 1 / (S + (1 − S) / N); üst sınır 1 / S. S = 0,1 için 16 çekirdekte hızlanma 6,4×, sınır 10×'tur. **Gustafson yasası** problem büyüdüğünde ölçeklenmiş hızlanmayı S + (1 − S) · N olarak verir.
- **Kullanıcı düzeyi** iş parçacıklarında geçiş çok ucuzdur, ama bloke eden çağrı ve sayfa hatası bütün süreci durdurur, saat kesmesi kütüphaneye gelmez ve tek süreç çok çekirdek kullanamaz. **Çekirdek düzeyi** iş parçacıkları bağımsız bloke olur ve paralel çalışır, ama her biri çekirdek kaynağı harcar.
- **N:1** ucuz ama paralelliksizdir; **1:1** (Linux, Windows, macOS) gerçek paralellik sağlar; **M:N** işletim sistemlerinde terk edildi, dil çalışma zamanlarında (Go, Erlang, Java 21 sanal iş parçacıkları) geri döndü.
- **Pthreads:** `pthread_create`, `pthread_join`, `pthread_exit`, `pthread_detach`. Argüman ve sonuç `void *` ile geçer; iş parçacığına verilen adres yeterince uzun yaşamalı, sonuç heap'te döndürülmelidir. **Java:** `start()` yeni iş parçacığı başlatır, `run()` başlatmaz; `join` hem bekler hem görünürlüğü sağlar.
- **Örtük iş parçacığı:** havuzlar oluşturma maliyetini bir kez öder ve iş parçacığı sayısını sınırlar (G/Ç yoğun işte $N_{\text{çekirdek}} \times (1 + B / H)$); fork-join görevi böler ve **iş çalma** ile yükü dengeler; OpenMP derleyici yönergeleriyle, GCD bloklar ve kuyruklarla, oneTBB şablonlarla çalışır.
- POSIX'te `fork()` yalnız çağıran iş parçacığını kopyalar. Sinyal işleyicileri süreç geneli, sinyal maskesi iş parçacığına özeldir. **Ertelenmiş iptalde** iş parçacığı iptal noktalarında güvenle sonlanır. `errno` gibi değişkenler **TLS** ile iş parçacığına özel yapılır.
- `sayac++` oku-artır-yaz üç adımıdır; korunmazsa artışlar kaybolur ve sonuç her çalıştırmada değişir. Çözüm kilit, atomik işlem ya da paylaşımsız tasarımdır. Linux'ta süreç ve iş parçacığı aynı `clone()` çağrısıyla, farklı paylaşım bayraklarıyla oluşturulur.

4.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

iş parçacığı *thread* · çok iş parçacıklı süreç *multithreaded process* · bağlam değiştirme *context switch* · eş zamanlılık *concurrency* · paralellik *parallelism* · yanlış paylaşım *false sharing* · Amdahl yasası *Amdahl's law* · kullanıcı düzeyi / çekirdek düzeyi iş parçacığı *user-level / kernel-level thread* · iş parçacığı havuzu *thread pool* · iş çalma *work stealing* · sinyal *signal* · iş parçacığına yerel depolama (TLS) *thread-local storage* · yeniden girilir / iş parçacığı güvenli *reentrant / thread-safe* · yarış durumu *race condition* · atomik komut *atomic instruction*

4.10 Kendini Dene

1. Aynı süreçteki iki iş parçacığı için şu öğelerden hangileri ortaktır, hangileri her birine özeldir: yığın işaretçisi, açık dosya tablosu, `static` bir değişken, `_Thread_local` bir değişken, heap'te `malloc` ile ayrılmış bir dizi?

   Cevap

   Ortak: açık dosya tablosu, `static` değişken ve heap'teki dizi; üçü de sürecin adres alanına ya da kaynaklarına aittir. Özel: yığın işaretçisi (her iş parçacığının kendi yığını vardır) ve `_Thread_local` değişken (her iş parçacığında ayrı kopya bulunur).

2. Bir programın %70'i paralelleştirilebiliyor. Dört çekirdekte hızlanma ve teorik üst sınır nedir?

   Cevap

   S = 0,3. Dört çekirdekte süre 0,3 + 0,7 / 4 = 0,3 + 0,175 = 0,475; hızlanma 1 / 0,475 ≈ 2,11×. Üst sınır 1 / 0,3 ≈ 3,33×'tür.

3. Tek çekirdekli bir makinede çalışan çok iş parçacıklı bir programda yarış durumu olabilir mi, neden?

   Cevap

   Evet. Paralellik olmasa da eş zamanlılık vardır. Bir iş parçacığının zaman dilimi `LOAD` ile `STORE` arasında bitebilir; çizelgeleyici öteki iş parçacığını çalıştırır, o da aynı eski değeri okur. Kayıp güncelleme tek çekirdekte de olur; yalnız daha seyrek görülür.

4. N:1 modelde çalışan bir programın iş parçacıklarından biri, bellekte olmayan bir sayfaya erişiyor. Ne olur? Aynı durum 1:1 modelde nasıl sonuçlanır?

> **Cevap**
>
> N:1 modelde sayfa hatası çekirdeğe gider ve çekirdek sayfayı diskten getirirken sürecin tek çekirdek iş parçacığını bekletir; kütüphane başka bir iş parçacığına geçemez ve bütün süreç durur. 1:1 modelde yalnız hatayı yapan iş parçacığı bekler; ötekiler çalışmaya devam eder.

5. `a = {4, 1, 3, 5, 2, 6}` dizisi, `ESIK = 2` olan fork-join `Toplam` göreviyle toplanıyor. Görev ağacını ve her düğümün sonucunu bulun.

> **Cevap**
>
> Kök [0,6) bölünür: orta = 3. Sol [0,3) üç öğelidir ve eşikten büyüktür; orta = 1 ile [0,1) = 4 ve [1,3) = 1 + 3 = 4'e bölünür, toplam 8. Sağ [3,6) da bölünür: [3,4) = 5 ve [4,6) = 2 + 6 = 8, toplam 13. Kök 8 + 13 = 21'dir. Yedi görev oluşur; dördü yapraktır. (Tek işçili `ForkJoinPool` ile JDK 21'de çalıştırıldı: 21.)

6. Dört çekirdekli bir makinede her görev 10 ms hesap yapıp 30 ms veritabanı yanıtı bekliyor. Havuz boyutu için kaba kural ne verir? Görevler yalnız hesap yapsaydı ne olurdu?

> **Cevap**
>
> $N_{iş} \approx 4 \times (1 + 30 / 10) = 16$. Yalnız hesap yapan görevlerde boyut çekirdek sayısı kadar, yani 4 olmalıdır; fazlası yalnız bağlam değiştirme ve önbellek çekişmesi ekler.

7. Dört iş parçacıklı bir süreçte bir iş parçacığı `fork()`, çocuk da hemen ardından `exec("/bin/ls")` çağırıyor. `exec` 'ten önce ve sonra çocukta kaç iş parçacığı vardır?

> **Cevap**
>
> `exec` 'ten önce bir: POSIX'te yalnız `fork` 'u çağıran iş parçacığı kopyalanır. `exec` 'ten sonra da bir: yeni program bütün adres alanını değiştirir ve tek bir iş parçacığıyla başlar.

8. Üç iş parçacığı başlangıçta 0 olan bir sayacı birer kez, korumasız `sayac++` ile artırıyor. Olası son değerler nelerdir?

> **Cevap**
>
> 1, 2 ya da 3. Üçü de yazmadan önce 0 okursa sonuç 1; ikisi aynı değeri okursa 2; hiç çakışma olmazsa 3'tür. 90 olası sıralamanın 48'inde 1, 36'sında 2, yalnız 6'sında 3 çıkar.

9. Bir Java programı `Thread t = new Thread(r);` satırından sonra `t.run();` çağırıyor ve hiç

`start()` çağırmıyor. Kaç iş parçacığı çalışır ve `t.getState()` ne döndürür?

Cevap

Yalnız `main` iş parçacığı çalışır; `r`'nin kodu `main`'in içinde sıradan bir metot gibi yürür. `t` hiç başlatılmadığı için `getState()` `NEW` döndürür.

4.11 İleri okuma

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, "Concurrency: An Introduction" ve "Interlude: Thread API" bölümleri: yarış durumunu ve Pthreads API'sini küçük, çalıştırılabilir C örnekleriyle anlatır; ostep.org'da ücretsizdir.
- A. S. Tanenbaum ve H. Bos, *Modern Operating Systems*, Kesim 2.2: kelime işlemci, web sunucusu, kullanıcı ve çekirdek düzeyi iş parçacıkları, açılır iş parçacıkları ve tek iş parçacıklı kodu çok iş parçacıklı hâle getirme sorunlarını bu nottaki örneklerin kaynağı olarak ayrıntılı işler.
- OpenJDK, *JEP 444: Virtual Threads*: sanal iş parçacıklarının neden tasarlandığını, taşıyıcıya bağlanma ve indirme düzeneğini ve hangi durumlarda kazanç sağlamadığını resmî belgede anlatır.
- M. D. Hill ve M. R. Marty, "Amdahl's Law in the Multicore Era", *IEEE Computer*, 41(7), 2008: Amdahl yasasını farklı çekirdek tasarımları için genişletir ve seri kısmın çok çekirdekli yongaların tasarımını nasıl etkilediğini gösterir.

4.12 Kaynaklar

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 4: Threads & Concurrency.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Kesim 2.2: Threads.
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Concurrency: An Introduction", "Thread API".
4. POSIX.1-2017 (IEEE Std 1003.1); `pthreads(7)` ve `clone(2)` Linux kılavuz sayfaları.
5. JEP 444: Virtual Threads (Java 21); OpenMP 5.2 belirtimi.
6. G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", *AFIPS Spring Joint Computer Conference*, 1967, s. 483–485.
7. J. L. Gustafson, "Reevaluating Amdahl's Law", *Communications of the ACM*, 31(5), 1988, s. 532–533.
8. M. D. Hill, M. R. Marty, "Amdahl's Law in the Multicore Era", *IEEE Computer*, 41(7), 2008, s. 33–38.

KISIM II · BÖLÜM 5

CPU Çizelgeleme

Bellekte çalışmaya hazır onlarca süreç bulunur, ama bir çekirdek aynı anda yalnız birini çalıştırabilir. CPU her boşaldığında hangi sürecin çalışacağına işletim sisteminin çizelgeleyicisi karar verir. Bu karar, kullanıcının bir tuşa bastıktan sonra ne kadar beklediğini, sunucunun saatte kaç iş bitirdiğini ve bir fren denetleyicisinin son tarihini kaçırıp kaçırmadığını belirler. Not, kararın hangi anlarda alındığını ve iyi bir kararın hangi ölçütlerle tartıldığını anlatarak başlar. Ardından FCFS, SJF, SRTF, öncelik ve Round Robin algoritmaları aynı dört süreçlik küme üzerinde adım adım çalıştırılır; böylece sonuçlar doğrudan karşılaştırılabilir. Çok düzeyli kuyruklar, piyango ve stride gibi orantılı pay yöntemleri, çok işlemcili sistemlerde yük dengeleme ve gerçek zamanlı sistemlerde RM ile EDF bunları izler. Son kesimler Linux'un CFS ve EEVDF çizelgeleyicilerini, Windows'un öncelik düzeylerini ve algoritmaları değerlendirme yöntemlerini ele alır. Bütün Gantt şemaları ve ortalamalar algoritmalar gerçekten çalıştırılarak denetlenmiştir.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum05` · **Laboratuvar:** `github.com/sercankulcu/operating-systems-labs/lab05`

ÖĞRENME ÇIKTILARI

Kavramlar
CPU–G/Ç patlama döngüsünü, çizelgeleme kararının alındığı dört durumu, kesmeli ve kesmesiz çizelgelemeyi ve görev dağıtıcının işini açıklamak.

Ölçütler
Bir Gantt şemasından bekleme, dönüş ve tepki sürelerini hesaplamak; verim, CPU kullanımı ve adalet hedeflerini birbirine karşı tartmak.

Algoritmalar
FCFS, SJF, SRTF, öncelik, Round Robin, MLQ ve MLFQ algoritmalarını aynı süreç kümesine adım adım uygulamak ve sonuçlarını karşılaştırmak.

Tahmin ve adalet
Sonraki CPU patlamasını üstel ortalamayla tahmin etmek; açlığı yaşlandırma, piyango, stride ve adil paylaşımla önlemek.

Çok işlemci ve gerçek zaman
Yük dengelemeyi ve işlemci afinitesini açıklamak; RM ve EDF çizelgelerini çizmek, çizelgelenebilirliği kullanım sınırıyla sınamak.

Gerçek sistemler
Linux CFS'i ve onun yerini alan EEVDF'i, Windows öncelik sınıflarını ve dinamik öncelik artırmalarını yorumlamak.

5.1 Temel kavramlar

Tek görevli eski bir sistemde çizelgeleme sorunu yoktur: bellekte tek program vardır ve CPU onu çalıştırır. Program diskten veri beklediğinde CPU da boşta bekler. Çok programlı (*multiprogramming*) sistemler bu boşluğu doldurmak için bellekte aynı anda birçok süreç tutar. Bir süreç beklemeye geçince CPU başka birine verilir. Fikir basittir, ama hemen bir soru doğurur: hazır bekleyenlerden hangisi seçilmelidir? Bu bölümün konusu o sorudur.

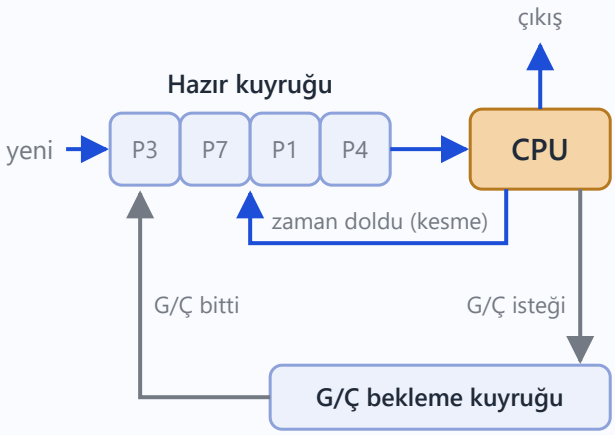
5.1.1 Çizelgeleme nedir?

Tanım 5.1

Kısa vadeli çizelgeleyici (*CPU scheduler*), CPU boşaldığında hazır kuyruğundaki süreçlerden hangisinin çalışacağını seçen çekirdek bileşenidir. **Hazır kuyruğu** (*ready queue*) çalışmaya hazır, yalnız CPU bekleyen süreçleri tutar. G/Ç ya da başka bir olay bekleyen süreçler ise **bekleme kuyruklarında** durur.

Çizelgeleyicinin amacı iki parçalıdır. Birincisi CPU'yu boş bırakmamaktır: hazır süreç varken CPU boşta kalmamalıdır. İkincisi, sistemin hedefini iyileştirmektir. Hedef ortama göre değişir: masaüstünde hızlı tepki, sunucuda yüksek verim, paylaşımlı makinede adalet, gömülü denetleyicide son tarihe uymak. Aynı algoritma bir hedefte iyi, ötekinde kötü olabilir; bölüm boyunca bu ödünleşim tekrar tekrar karşımıza çıkacak.

Bir süreç hayatı boyunca kuyruklar arasında dolaşır (Şekil 5.1). Yeni süreç hazır kuyruğuna girer. Çizelgeleyici onu seçince CPU'da çalışır. Zaman dilimi dolarsa saat kesmesiyle durdurulur ve hazır kuyruğuna geri döner. G/Ç isterse bekleme kuyruğuna geçer, G/Ç bitince yeniden hazır olur. İşi bitince sistemden çıkar. Her çekirdekte aynı anda en fazla bir süreç çalışır; çalışanlar dışındaki bütün süreçler bir kuyruktadır.



Şekil 5.1: Kuyruk diyagramı. Yeni süreç hazır kuyruğuna girer ve CPU'ya seçilir. Zaman dolunca hazır kuyruğuna döner, G/Ç isteyince G/Ç bekleme kuyruğuna geçer, G/Ç bitince yeniden hazır olur.

Hazır kuyruğu adına rağmen FIFO olmak zorunda değildir. Algoritmaya göre bir öncelikli kuyruk, dengeli bir ağaç ya da sırasız bir bağlı liste olabilir. Linux'un adil çizelgeleyicisi örneğin kırmızı-siyah ağaç kullanır. Kuyruktaki kayıtlar da süreçlerin kendisi değil, süreç denetim bloklarıdır (PCB).

Bir ayrıntı daha: iş parçacıklarını destekleyen modern çekirdekler aslında süreçleri değil, çekirdek iş parçacıklarını çizelgeler. Çok iş parçacıklı bir süreçte her iş parçacığı hazır kuyruğunda ayrı bir kayıttır. Ders kitapları yine de “süreç çizelgeleme” deyimini kullanır; bu notta da süreç sözcüğü “çizelgelenen birim” anlamındadır.

5.1.2 CPU–G/Ç patlama döngüsü

Çizelgelemenin başarısı süreçlerin gözlenen bir özelliğine dayanır: yürütme, CPU çalışması ile G/Ç beklemesi arasında gidip gelir. Süreç bir süre hesap yapar, sonra bir dosya okur ve bekler, sonra yine hesap yapar.

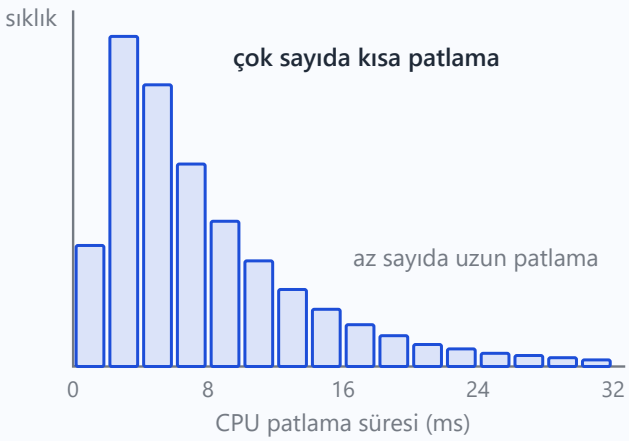
Tanım 5.2

CPU patlaması (*CPU burst*), sürecin iki G/Ç isteği arasında CPU'yu kesintisiz kullandığı süredir. **G/Ç patlaması** ise sürecin bir G/Ç işleminin bitmesini beklediği süredir. Süreç bir CPU patlamasıyla başlar ve son CPU patlamasında sonlanma çağrısıyla biter (Şekil 5.2).



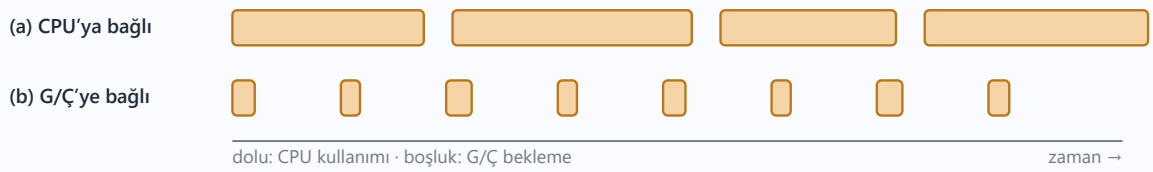
Şekil 5.2: CPU ve G/Ç patlamalarının dönüşümlü dizisi. `load`, `store`, `add` gibi komutlardan oluşan bir CPU patlamasını dosyadan okuma ve bekleme izler; ardından yeni bir CPU patlaması gelir.

Gerçek sistemlerde ölçülen CPU patlama sürelerinin dağılımı hep aynı biçimdedir (Şekil 5.3). Çok sayıda kısa patlama, az sayıda uzun patlama vardır. Grafik üstel ya da ona benzer, sağa doğru uzun kuyruklu bir eğridir. Bu gözlem, kısa patlamalı süreçlere öncelik veren algoritmaların neden işe yaradığını açıklar.



Şekil 5.3: CPU patlama sürelerinin histogramı. Kısa patlamalar çok sık, uzun patlamalar seyrektir.

Süreçler patlama yapılarına göre iki gruba ayrılır (Şekil 5.4). **CPU'ya bağlı** (*CPU-bound*) süreç uzun CPU patlamaları ve seyrek G/Ç yapar; video kodlama, derleme ya da bilimsel hesap böyledir. **G/Ç'ye bağlı** (*I/O-bound*) süreç kısa CPU patlamalarından sonra uzun G/Ç beklemelerine girer; metin düzenleyici, kabuk ya da veritabanı istemcisi buna örnektir. Bir süreç zamanla gruptan gruba geçebilir: önce dosyaları okuyan, sonra uzun hesap yapan bir program iki evreyi de yaşar.



Şekil 5.4: (a) CPU'ya bağlı süreç uzun CPU patlamaları yapar. (b) G/Ç'ye bağlı süreç kısa patlamalardan sonra uzun süre bekler. Dolu kutular CPU kullanımını, aradaki boşluklar G/Ç beklemesini gösterir.

Çizelgeleme açısından belirleyici olan G/Ç beklemesinin uzunluğu değil, CPU patlamasının kısalığıdır. G/Ç'ye bağlı bir süreç hazır olduğunda hemen çalıştırılırsa birkaç milisaniyede yeni bir G/Ç isteği verir. Böylece disk ya da ağ kartı da meşgul kalır. Bu süreç uzun bir CPU işinin arkasında bekletilirse aygıt boşta kalır. Disk teknolojisi CPU'dan çok daha yavaş geliştiği için süreçlerin büyük kısmı zamanla G/Ç'ye bağlı hâle gelmiştir. Bu yüzden G/Ç'ye bağlı süreçleri bekletmemek modern çizelgeleyicilerin ortak tasarım ilkesidir.

5.1.3 G/Ç ile örtüşme: patlamaları ayrı iş saymak

Bir sürecin her CPU patlamasını ayrı, kısa bir iş saymak, çizelgeleyiciye G/Ç ile hesaplamayı örtüştürme fırsatı verir. Bu fikir bölümdeki algoritmaların çoğunun arkasında yatar; en iyi bir örnekle anlaşılır.

Örnek 5.1

A ve B süreçleri 50'şer ms CPU zamanı istiyor. A, 10 ms çalışıp 10 ms disk bekliyor; bu döngü beş kez tekrarlanıyor. B hiç G/Ç yapmıyor. İki çizelgeyi karşılaştıralım.

1. **Örtüşmesiz çizelge.** A'nın 50 ms'lik işi tek parça sayılır; B ancak A bitince başlar. A beş CPU patlaması ve dört G/Ç beklemesiyle 5 × 10 + 4 × 10 = 90'da biter. A diskteyken CPU boştur, çünkü sıradaki iş hâlâ A'dır. B 90–140 arasında çalışır. Ortalama dönüş (90 + 140) / 2 = 115 ms, CPU kullanımı 100 / 140 ≈ %71'dir.
2. **Örtüşmeli çizelge.** A'nın her 10 ms'lik patlaması ayrı, kısa bir iş sayılır. A diske gidince B çalışır. A'nın G/Ç'si bitince kalan süresi 10 ms olan yeni A işi, kalan süresi daha uzun olan B'yi keser. Bu, kalan süresi en kısa olanı seçen kuralın (STCF ya da SRTF) davranışıdır. Çizelge A 0–10, B 10–20, A 20–30, … biçiminde ilerler; A yine 90'da, B ise 100'de biter. Ortalama dönüş (90 + 100) / 2 = 95 ms, CPU kullanımı 100 / 100 = %100'dür.
3. **Yorum.** A'nın bitiş zamanı değişmedi, ama B 40 ms erken bitti ve CPU hiç boş kalmadı. Kazanç, A'nın disk beklemesinin B'nin hesaplamasıyla örtüşmesinden geldi (Şekil 5.5).



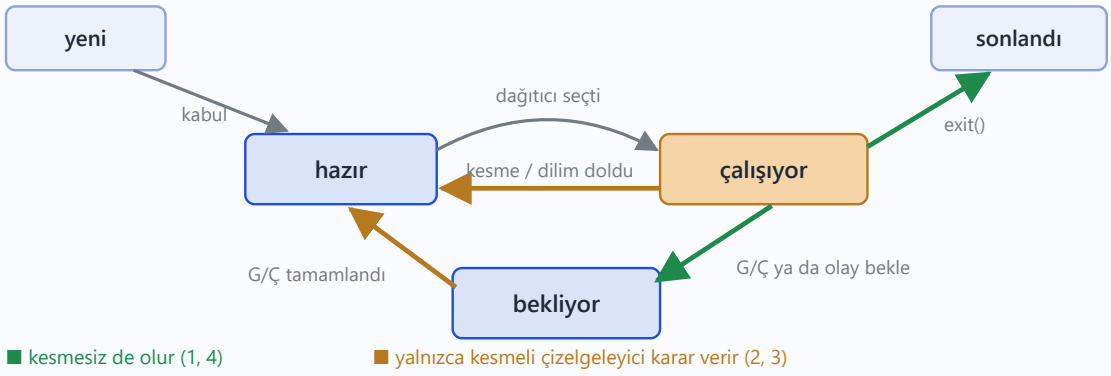
Şekil 5.5: Üstte örtüşmesiz çizelge: A'nın G/Ç beklemelerinde CPU boş kalır, B 140'ta biter. Altta örtüşmeli çizelge: A diskteyken B çalışır, iki süreç de 100 ms içinde biter.

Gerçek bir sistem patlama sürelerini önceden bilmez. Bu yüzden "kısa patlamalıyı öne al" kuralı ancak dolaylı olarak uygulanabilir. Çok düzeyli geri beslemeli kuyruk tam olarak bunu yapar: süreçlerin davranışını gözleyip kısa patlamalıları üst kuyruklarda tutar.

5.1.4 Çizelgeleme kararları ne zaman alınır?

Çizelgeleyici sürekli çalışan bir döngü değildir; belirli olaylarda çağrılır. Süreç durum diyagramında dört geçiş bir karar gerektirebilir (Şekil 5.6):

1. **Çalışıyor → bekliyor.** Süreç G/Ç ister ya da bir çocuğunu `wait()` ile bekler. CPU boşalır; yeni bir süreç seçmek zorunludur.
2. **Çalışıyor → hazır.** Saat kesmesi gelir, sürecin zaman dilimi dolmuştur. Çalışanı durdurup başkasını seçmek isteğe bağlıdır.
3. **Bekliyor → hazır.** Bir G/Ç tamamlanır ve bekleyen süreç uyanır. Uyanan süreç o an çalışandan daha önemliyse çalışanı kesebilir.
4. **Sonlanma.** Süreç `exit()` ile biter. CPU boşalır; seçim yine zorunludur.



Şekil 5.6: Süreç durum diyagramında çizelgeleme kararlarının alındığı dört geçiş. Yeşil geçişlerde (1 ve 4) her çizelgeleyici karar verir; turuncu geçişlerde (2 ve 3) yalnız kesmeli çizelgeleyici karar verir.

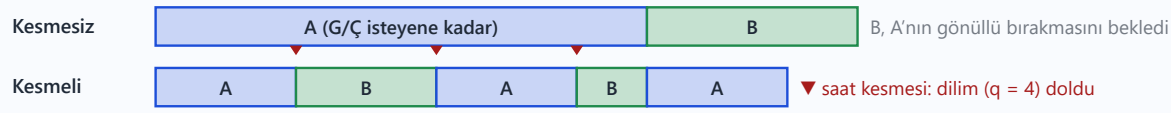
1. ve 4. durumlarda başka seçenek yoktur: CPU boştur ve birine verilmelidir. 2. ve 3. durumlarda ise CPU hâlâ bir süreci çalıştırıyor; çizelgeleyici onu çalışmaya bırakmayı da seçebilir. Çizelgeleyiciler bu iki durumda karar verip vermediklerine göre ikiye ayrılır.

5.1.5 Kesmeli ve kesmesiz çizelgeleme

Tanım 5.3

Kesmesiz (*non-preemptive*) ya da işbirlikçi çizelgelemede karar yalnız 1. ve 4. durumlarda alınır. Seçilen süreç bloke olana, sonlanana ya da CPU'yu gönüllü bırakana (`yield`) kadar çalışır. **Kesmeli** (*preemptive*) çizelgelemede 2. ve 3. durumlarda da karar alınır. Süreç, zaman dilimi dolunca ya da daha önemli bir süreç hazır olunca askıya alınabilir.

Kesmesiz çizelgeleme saat kesmesine ihtiyaç duymaz ve bağlam değiştirme sayısı azdır. Bedeli, sistemin her programın iyi niyetine bağlı kalmasıdır. Windows 3.x ve klasik Mac OS (9 ve öncesi) işbirlikçi çoklu görev kullanıyordu. Sonsuz döngüye giren tek bir program CPU'yu hiç bırakmadığı için bütün sistemi kilitleyebiliyordu. Kesmeli çizelgeleme bunu saat kesmesiyle önler: çekirdek belirli aralıklarla CPU'yu geri alır. Bugün bütün genel amaçlı işletim sistemleri kesmelidir (Şekil 5.7).



Şekil 5.7: Aynı iki süreç. Kesmesizde A, G/Ç isteyene kadar çalışır ve B, A'nın CPU'yu gönüllü bırakmasını bekler. Kesmelide saat kesmesi A'yı her q = 4 birimin sonunda keser ve A ile B dönüşümlü çalışır.

Kesmenin de bir bedeli vardır. İki süreç ortak bir veriyi paylaşıyorsa biri veriyi güncellerken kesilebilir. Öteki süreç verinin yarım kalmış, tutarsız hâlini okuyabilir. Bu **yarış durumu** (*race condition*) riski, kesmeli sistemlerde senkronizasyon araçlarını zorunlu kılar. Ayrıca her kesme bir bağlam değiştirme demektir ve bu da CPU zamanı harcar.

Aynı soru çekirdeğin kendi kodu için de sorulur. Bir süreç sistem çağrısının ortasında çekirdek kodunu çalıştırırken kesilebilir mi? Eski UNIX çekirdekleri çağrının bitmesini beklerdi. Linux 2.6 sürümünden beri çekirdek kodu da kesilebilir; kesme yalnız çalışan kod hiçbir kilit tutmuyorsa yapılır. Çekirdek bunu her görev için tuttuğu `preempt_count` sayacıyla izler: sayaç sıfırsa kesmek güvenlidir. Gerçek zamanlı kullanım için geliştirilen PREEMPT_RT yamaları yaklaşık yirmi yıllık çalışmadan sonra 2024'te Linux 6.12 ile ana çekirdeğe girdi. Bu yapılandırmada kesme işleyicileri de iş parçacığı olarak çalışır ve kesilebilir; en kötü durum gecikmesi mikrosaniyeler düzeyine iner.

Hangi yaklaşımın seçileceği ortama bağlıdır. Gece çalışan toplu işlerde kesmesiz ya da uzun dilimli kesmeli algoritmalar yeterlidir, çünkü kimse ekran başında beklemez. Etkileşimli sistemlerde ve sunucularda kesme zorunludur: bir sürecin CPU'yu tekeline alması öteki kullanıcıların isteklerini daki-

kalarca bekletebilir.

Sık yapılan hata

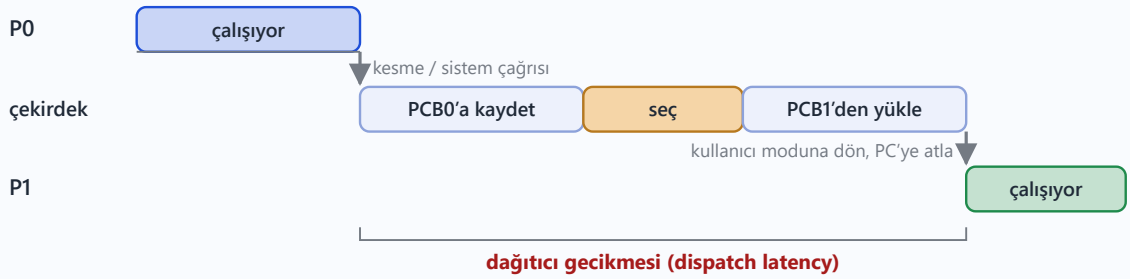
"Kesmeli çizelgeleyici G/Ç isteğinde karar vermez" sanmak. Kesmeli çizelgeleyici dört durumun hepsinde karar verir; kesmesiz olandan farkı, 2. ve 3. durumlarda da karar vermesidir. Bir disk okumasının bitmesi (bekliyor → hazır) yalnız kesmeli çizelgeleyicide çalışan süreci durdurabilir. Kesmesiz çizelgeleyicide uyanan süreç hazır kuyruğuna girer ve sırasını bekler.

5.1.6 Görev dağıtıcı ve bağlam değiştirme

Seçim yapmak ile seçilen süreci çalıştırmak iki ayrı iştir. Seçimi çizelgeleyici yapar; bu bir politikadır. Seçilen süreci CPU'ya yerleştirmek ise bir mekanizmadır ve onu görev dağıtıcı üstlenir.

Tanım 5.4

Görev dağıtıcı (*dispatcher*), çizelgeleyicinin seçtiği sürece CPU'yu veren çekirdek modülüdür. Üç iş yapar: bağlamı değiştirir, kullanıcı moduna geçer ve programdaki doğru adrese atlar. Dağıtıcının bir süreci durdurup ötekini başlatması için geçen süreye **dağıtım gecikmesi** (*dispatch latency*) denir (Şekil 5.8).



Şekil 5.8: Dağıtıcının işi. Bir kesme ya da sistem çağrısıyla çekirdeğe girilir. Dağıtıcı P0'ın durumunu PCB0'a kaydeder ve çizelgeleyici P1'i seçer. Dağıtıcı P1'in durumunu PCB1'den yükler, kullanıcı moduna döner ve P1'in kaldığı yere atlar. Aradaki süre dağıtım gecikmesidir.

Dağıtıcı her bağlam değiştirmede çalıştığı için çok hızlı olmalıdır. Bugünkü işlemcilerde doğrudan maliyet birkaç mikrosaniyedir. Dolaylı maliyet daha büyük olabilir: yeni sürecin verisi önbellekte yoktur ve ilk erişimler ıskalarla yavaşlar. Tablo 5.1 süreçler arası bir geçişte nelerin değiştiğini özetler.

Bileşen	Süreçler arası geçişte
Yazmaçlar, PC, SP	eski sürecin PCB'sine kaydedilir, yenisininki yüklenir
Bellek haritası	sayfa tablosu tabanı değişir (x86'da CR3 yazmacı)
TLB	işlemci PCID/ASID etiketlerini destekliyorsa boşaltılmaz

Bileşen	Süreçler arası geçişte
Önbellekler	boşaltılmaz, ama yeni süreç için soğuktur

Tablo 5.1: Süreçler arası bağlam değiştirmede donanım durumunun bileşenleri.

Linux'ta bağlam değiştirmeler doğrudan ölçülebilir. `vmstat 1` komutu her saniye bir satır basar; `cs` sütunu o saniyedeki toplam bağlam değiştirme sayısıdır. Tek bir süreç için `/proc/<pid>/status` dosyası iki sayaç tutar. `voluntary_ctxt_switches`, sürecin bloke olarak CPU'yu kendisinin bıraktığı geçişleri sayar. `nonvoluntary_ctxt_switches` ise çizelgeleyicinin süreci zorla kestiği geçişleri sayar. G/Ç'ye bağlı bir süreçte ilk sayaç, CPU'ya bağlı bir süreçte ikincisi büyür.

5.1.7 Ortama göre hedefler

Tek bir "en iyi" çizelgeleyici yoktur, çünkü sistemlerin istedikleri farklıdır. Tanenbaum hedefleri ortamlara göre gruplar (Tablo 5.2). Bazı hedefler her sistemde geçerlidir. **Adalet**, benzer süreçlerin benzer CPU payı almasıdır. **Politika uygulama**, sistemin koyduğu kurala uymaktır; örneğin güvenlik denetim süreçlerinin önce çalışması. **Denge** ise CPU'nun ve G/Ç aygıtlarının birlikte meşgul tutulmasıdır.

Ortam	Öncelikli hedefler	Uygun yaklaşım
Tüm sistemler	Adalet, politika uygulama, denge	—
Toplu iş (*batch*)	Verim (iş/saat), kısa dönüş süresi, yüksek CPU kullanımı	Kesmesiz ya da uzun dilimli kesmeli
Etkileşimli	Kısa tepki süresi, orantılılık (gecikme kullanıcının beklentisine uygun)	Kesmeli, kısa zaman dilimi
Gerçek zamanlı	Son tarihlere uymak (veri kaybı yok), öngörülebilirlik (ses ve görüntü kalitesi düşmez)	Öncelik ya da son tarih tabanlı; görevler kısa çalışıp bloke olur

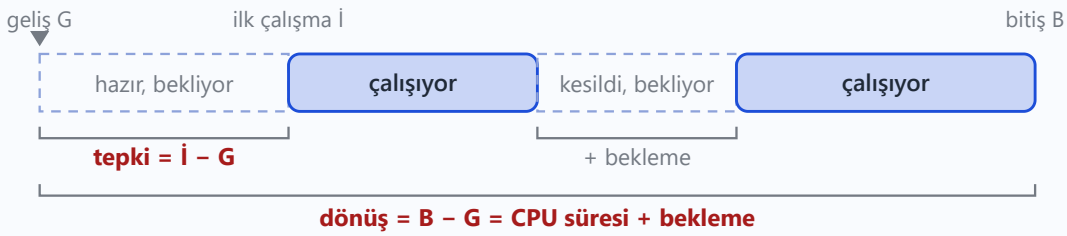
Tablo 5.2: Ortama göre çizelgeleme hedefleri ve uygun yaklaşımlar.

Orantılılık ilginç bir hedeftir. Kullanıcı büyük bir dosyayı sıkıştırmanın uzun süreceğini bilir ve bekler. Ama bir pencereyi kapatmanın birkaç saniye sürmesine katlanmaz. İyi bir etkileşimli çizelgeleyici gecikmeyi işin algılanan büyüklüğüyle orantılı tutar.

Toplu işler eski zamanlardan kalma bir kavram değildir. Bankaların gece çalışan bordro ve hesap kapatma işleri, bilimsel hesaplar, yapay zekâ modellerinin eğitimi ve sürekli tümleştirme (CI) derlemeleri bugün de toplu iştir. Yüksek başarımlı hesap merkezlerinde bu işler Slurm gibi kuyruk yöneticileriyle sıraya konur. Hedefler çoğu zaman birbiriyle çelişir: tepki süresini kısaltmak için sık bağlam değiştirmek, her geçişin maliyeti yüzünden verimi düşürür.

5.1.8 Çizelgeleme ölçütleri

Algoritmaları karşılaştırmak için hedefleri ölçülebilir büyüklüklere çevirmek gerekir. Bir sürecin hayatındaki üç an bu ölçütlerin hepsini tanımlamaya yeter: sisteme **geliş** anı G, CPU'ya **ilk çıkış** anı İ ve **bitiş** anı B (Şekil 5.9).



Şekil 5.9: Bir sürecin zaman çizgisinde geliş (G), ilk çalışma (İ) ve bitiş (B) anları. Tepki süresi İ − G, dönüş süresi B − G'dir; dönüş süresi CPU süresiyle hazır kuyruğunda geçen bekleme sürelerinin toplamıdır.

Tanım 5.5

CPU kullanımı, CPU'nun meşgul olduğu zamanın oranıdır. **Verim** (*throughput*), birim zamanda tamamlanan süreç sayısıdır. **Dönüş süresi** (*turnaround time*), sürecin girişinden tamamlanmasına kadar geçen süredir: B − G. **Bekleme süresi** (*waiting time*), sürecin hazır kuyruğunda geçirdiği toplam süredir; süreç G/Ç yapmıyorsa dönüş süresi eksi CPU süresine eşittir. **Tepki süresi** (*response time*), girişten CPU'ya ilk çıkışa kadar geçen süredir: İ − G.

İlk iki ölçütün büyük, son üçünün küçük olması istenir. Gerçek sistemlerde CPU kullanımı hafif yüklü bir makinede %40, yoğun bir sunucuda %90 dolayındadır. Ölçütlerin hiçbiri tek başına yeterli değildir; algoritmalar arasındaki farklar en açık ortalama bekleme, ortalama dönüş ve ortalama tepki sürelerinde görünür.

Bu ölçütlerden biri ötekilerden daha temeldir. Bir sürecin CPU süresi ve G/Ç süresi programın kendisine aittir; hangi algoritma seçilirse seçilsin değişmez. Çizelgeleyicinin etkileyebildiği tek şey sürecin hazır kuyruğunda beklediği süredir. Dönüş süresi = CPU süresi + G/Ç süresi + bekleme süresi olduğuna göre, iki algoritma arasındaki dönüş farkı bekleme farkının ta kendisidir.

Tepki süresi ile bekleme süresi kolay karışır. Tepki yalnız ilk beklemeyi ölçer: kullanıcı komutu yazdıktan sonra ekranda ilk hareketi ne zaman görür? Bekleme süresi ise sürecin hayatı boyunca kuyrukta geçirdiği bütün aralıkların toplamıdır. Kesmeli bir algoritmada süreç birçok kez kesilip kuyruğa dönebilir; tepki süresi küçük kalırken bekleme süresi büyüyebilir.

Etkileşimli sistemlerde ortalamadan çok tepki süresinin **varyansı** önemlidir. Komutların çoğu 0,1 saniyede, arada bir de 5 saniyede yanıtlanırsa kullanıcı bunu, hepsinin 0,5 saniyede yanıtlanmasından daha rahatsız edici bulur. Son olarak **ağırlıklı** (normalleştirilmiş) **dönüş süresi**, dönüş süresinin CPU süresine oranıdır. İdeal değeri 1'dir ve kısa işlerin ne kadar cezalandırıldığını gösterir.

Örnek 5.2

P1 (geliş 0, CPU 5), P2 (geliş 1, CPU 4) ve P3 (geliş 2, CPU 2) şu Gantt şemasıyla çizelgelenmiş olsun: P1 0–2, P2 2–4, P3 4–6, P1 6–8, P2 8–10, P1 10–11. Üç sürecin ölçütlerini bulalım.

1. **Bitiş anları.** Her sürecin Gantt şemasındaki son parçası: P1 11'de, P2 10'da, P3 6'da biter.
2. **Dönüş = bitiş − geliş.** P1 11 − 0 = 11, P2 10 − 1 = 9, P3 6 − 2 = 4. Ortalama (11 + 9 + 4) / 3 = 8.
3. **Bekleme = dönüş − CPU.** P1 11 − 5 = 6, P2 9 − 4 = 5, P3 4 − 2 = 2; ortalama 13 / 3 ≈ 4,33. P1'in beklemeleri doğrudan da sayılabilir: 2–6 ve 8–10 aralıklarında kuyruktadır, 4 + 2 = 6.
4. **Tepki = ilk çalışma − geliş.** P1 0 − 0 = 0, P2 2 − 1 = 1, P3 4 − 2 = 2; ortalama 1.

P1 hemen başladığı için tepki süresi 0'dır, ama iki kez kuyrukta beklediği için bekleme süresi 6'dır. Tepki ilk beklemeyi, bekleme tüm beklemeleri ölçer.

Sık yapılan hata

Bekleme süresini "bitiş − CPU süresi" diye hesaplamak. Formül *dönüş* − CPU süresidir; dönüş de bitiş − geliştir. Geliş anı sıfırdan farklıysa ikisi farklı sonuç verir. Örnek 5.2'de P2 için bitiş − CPU = 6 bulunur, doğru bekleme ise 10 − 1 − 4 = 5'tir.

5.2 Temel algoritmalar

Algoritmaları anlamanın en iyi yolu onları elle çalıştırmaktır. Bu kesimdeki algoritmalar aynı dört süreç üzerinde adım adım izlenir; her birinin sonunda bekleme, dönüş ve tepki ortalamaları ile bağlam değiştirme sayısı verilir. Böylece bir algoritmanın neyi iyileştirip neyi kötüleştirdiği sayılarla görülür.

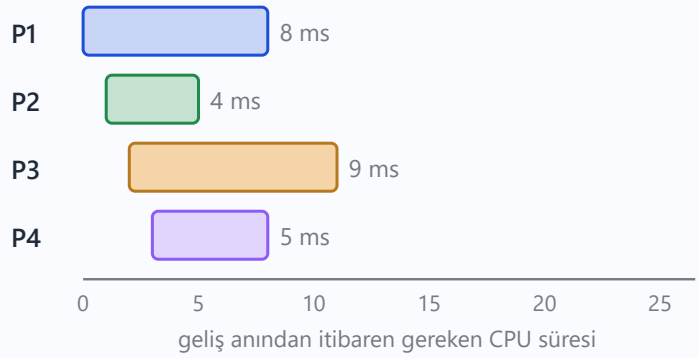
5.2.1 Ortak süreç kümesi

Tablo 5.3 bölüm boyunca kullanılan ortak kümeyi verir. Dört süreç birer milisaniye arayla gelir; CPU süreleri 4 ile 9 ms arasındadır. Öncelik sütunu yalnız öncelik algoritmasında kullanılır (Şekil 5.10).

Süreç	Geliş	CPU süresi	Öncelik
P1	0	8	2
P2	1	4	3
P3	2	9	1

Süreç	Geliş	CPU süresi	Öncelik
P4	3	5	4

Tablo 5.3: Ortak süreç kümesi (süreler ms).



Şekil 5.10: Ortak kümenin zaman çizgisi: her çubuk sürecin geliş anında başlar ve gereken CPU süresi kadar uzanır.

Hesaplarda dört kural geçerlidir. Küçük öncelik sayısı yüksek öncelik demektir. Eşitlikte önce gelen süreç kazanır. Round Robin'de bir sürecin geldiği anda başka bir sürecin dilimi doluyorsa yeni gelen kuyruğa önce girer. Bağlam değiştirme süresi sıfır kabul edilir ve süreçler G/Ç yapmaz. Bu varsayımlar gerçekçi değildir, ama algoritmaların farkını yalın biçimde gösterir. Aynı hesapları farklı kümelerle denemek için CPU çizelgeleme simülatörü kullanılabilir; simülatör Gantt şemasını ve ölçütleri anında çizer.

Bağlam değiştirme sayısı da şöyle sayılır: Gantt şemasında art arda gelen iki parçanın süreci farklıysa bir geçiş vardır. İlk sürecin CPU'ya çıkması geçiş sayılmaz. Bir süreç diliminin sonunda kuyrukta tek başınaysa yeni dilime geçmesi de geçiş değildir.

5.2.2 FCFS

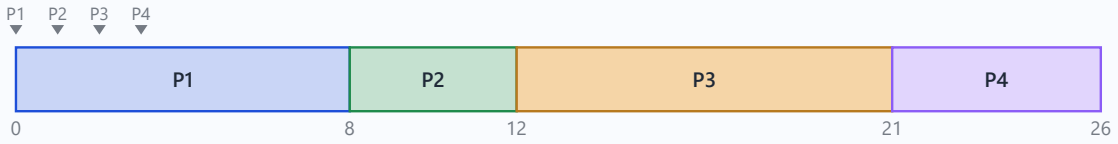
Tanım 5.6

İlk gelen ilk hizmet alır (FCFS, *first-come, first-served*) algoritmasında hazır kuyruğu FIFO'dur. CPU'yu alan süreç bitene ya da bloke olana kadar onu bırakmaz. FCFS kesmesizdir.

FCFS, gerçeklemesi en kolay algoritmadır: yeni süreç kuyruğun sonuna eklenir, CPU boşalınca baştaki çıkarılır. Her iki işlem O(1)'dir. Ortak kümede izleme şöyledir:

- **t = 0.** Kuyrukta yalnız P1 vardır. P1 CPU'yu alır ve 8 ms boyunca bırakmaz. Bu sürede P2, P3 ve P4 gelip sırayla kuyruğa girer.
- **t = 8.** P1 biter. Kuyruğun başındaki P2 çalışır. P2 1'de gelmişti, 8 − 1 = 7 ms bekledi.

- **t = 12.** P2 biter, P3 çalışır; 12 − 2 = 10 ms bekledi.
- **t = 21.** P3 biter, P4 çalışır; 21 − 3 = 18 ms bekledi. P4 26'da biter (Şekil 5.11; Tablo 5.4).



Şekil 5.11: FCFS Gantt şeması: süreçler geliş sırasıyla, kesilmeden çalışır. P1 0–8, P2 8–12, P3 12–21, P4 21–26.

Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	0	8	8	0	0
P2	1	4	8	12	11	7	7
P3	2	9	12	21	19	10	10
P4	3	5	21	26	23	18	18
Ort.					15,25	8,75	8,75

Tablo 5.4: FCFS ile ortak kümenin ölçütleri.

Ortalama bekleme 35 / 4 = 8,75 ms, ortalama dönüş 61 / 4 = 15,25 ms'dir. Kesmesiz algoritmalarda süreç ilk kez çalışmaya başladığında bitene kadar sürdüğü için tepki süresi bekleme süresine eşittir. Gantt şemasında üç geçiş vardır: P1→P2, P2→P3, P3→P4.

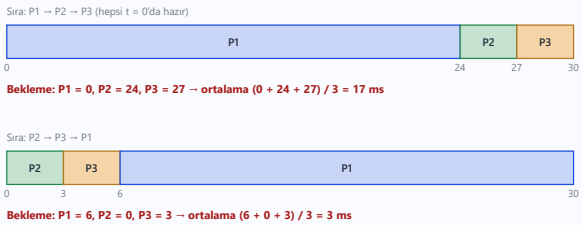
5.2.3 Konvoy etkisi

FCFS'nin ortalama bekleme süresi sabit değildir; süreçlerin geliş sırasına aşırı duyarlıdır. Silberschatz'ın klasik örneği bunu açıkça gösterir.

Örnek 5.3

P1 (24 ms), P2 (3 ms) ve P3 (3 ms) t = 0'da hazırdır. FCFS ile iki farklı geliş sırasında ortalama bekleme nedir?

1. **Sıra P1, P2, P3.** P1 0–24, P2 24–27, P3 27–30. Beklemeler 0, 24, 27; ortalama (0 + 24 + 27) / 3 = 17 ms.
2. **Sıra P2, P3, P1.** P2 0–3, P3 3–6, P1 6–30. Beklemeler P1 6, P2 0, P3 3; ortalama (6 + 0 + 3) / 3 = 3 ms.
3. **Yorum.** İşler aynı, toplam süre aynı (30 ms); yalnız sıra değişti ve ortalama bekleme 17'den 3'e düştü. Uzun iş öndeyse kısa işlerin hepsi onun süresi kadar bekler (Şekil 5.12).

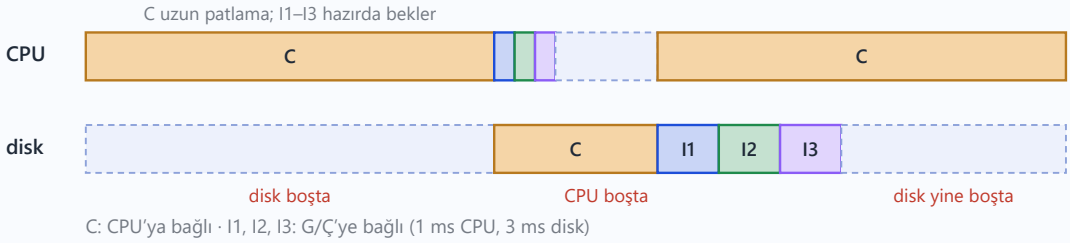


Şekil 5.12: Aynı üç işin iki sırası. Uzun P1 öndeyken ortalama bekleme 17 ms, kısa işler öndeyken 3 ms'dir.

Sorun, süreçlerin G/Ç yaptığı dinamik bir ortamda daha da büyür. Bir CPU'ya bağlı süreç C ile birkaç G/Ç'ye bağlı süreç (I1, I2, I3) düşünelim. C uzun patlamasıyla CPU'yu tutarken G/Ç'ye bağlı süreçlerin G/Ç'leri biter ve hepsi hazır kuyruğunda C'nin arkasına dizilir. Bu sırada disk boştadır. C sonunda G/Ç'ye gidince I1, I2, I3 kısa patlamalarını hızla bitirip diske gider. Bu kez CPU boşta kalır. Ardından C geri gelir ve döngü yinelenir.

Tanım 5.7

Konvoy etkisi (*convoy effect*), kısa patlamalı süreçlerin tek bir uzun patlamalı sürecin arkasında birikmesidir. Sonucunda CPU ve G/Ç aygıtları dönüşümlü olarak boşta kalır ve kullanımları düşer (Şekil 5.13).



Şekil 5.13: Konvoy etkisi. C uzun patlamasıyla CPU'yu tutarken I1–I3 hazır kuyruğunda bekler ve disk boştadır. C diske geçince G/Ç'ye bağlı süreçler kısa patlamalarını bitirip diske sıralanır ve bu kez CPU boşta kalır.

FCFS'de açlık yoktur: her süreç önündekiler bitince mutlaka çalışır. Ama kesmesiz olduğu için tek bir uzun iş sistemi uzun süre tutabilir. Bu yüzden FCFS etkileşimli sistemlere uygun değildir. Yine de birçok algoritmanın içinde yaşar: eşit öncelikli süreçler arasındaki bağı çözmek için genellikle geliş sırasına bakılır.

5.2.4 Kod izleme: FCFS bekleme süresi

FCFS'nin bekleme hesabı tek bir döngüyle yapılabilir. Aşağıdaki C fonksiyonu geliş sırasına göre sıralanmış süreç dizisini alır, her sürecin beklemesini yazar ve ortalamayı basar.

```c
/* p[] geliş sırasına göre sıralı */
void fcfs(Surec p[], int n) {
  int t = 0, top = 0;
  for (int i = 0; i < n; i++) {
    if (t < p[i].gelis) t = p[i].gelis;
    int bekleme = t - p[i].gelis;
    t += p[i].sure;          /* bitiş */
    top += bekleme;
    printf("%s: %d\n", p[i].ad, bekleme);
  }
  printf("ort = %.2f\n", (double) top / n);
}
```

`t`, CPU'nun boşaldığı anı tutan saattir; `top` beklemelerin toplamıdır. Döngü her süreç için önce saatin sürecin gelişinden geride olup olmadığına bakar. Geride ise CPU boşta beklemiştir ve saat geliş anına ileri alınır. Ortak kümede süreçler CPU boşalmadan geldiği için bu satır hiç çalışmaz, ama boşluklu kümelerde gereklidir. Sonra bekleme `t - p[i].gelis` olarak hesaplanır: süreç gelişinden CPU'yu alana kadar beklemiştir. `t += p[i].sure` saati sürecin bitişine taşır. En sonda toplam, süreç sayısına bölünür; `(double)` dönüşümü tam sayı bölmesini önler (Tablo 5.5).

i	Süreç	`t` (başta)	bekleme	`t` (sonda)	`top`	Çıktı
0	P1 (0, 8)	0	0 − 0 = 0	8	0	`P1: 0`
1	P2 (1, 4)	8	8 − 1 = 7	12	7	`P2: 7`
2	P3 (2, 9)	12	12 − 2 = 10	21	17	`P3: 10`
3	P4 (3, 5)	21	21 − 3 = 18	26	35	`P4: 18`
—						`ort = 8.75`

Tablo 5.5: `fcfs()` ortak kümede. Her satır döngü adımının sonundaki durumu gösterir.

Fonksiyon O(n) zamanda çalışır ve ek bellek kullanmaz. Dizinin sıralı olması gerekir; sıralama O(n log n) sürer. Programın çıktısında ondalık ayırıcı noktadır, çünkü `printf` C yerel ayarıyla çalışır.

5.2.5 SJF: en kısa iş önce

Konvoy örneği bir ipucu verir: kısa işleri öne almak ortalama beklemeyi düşürür. SJF bu fikri kurala dönüştürür.

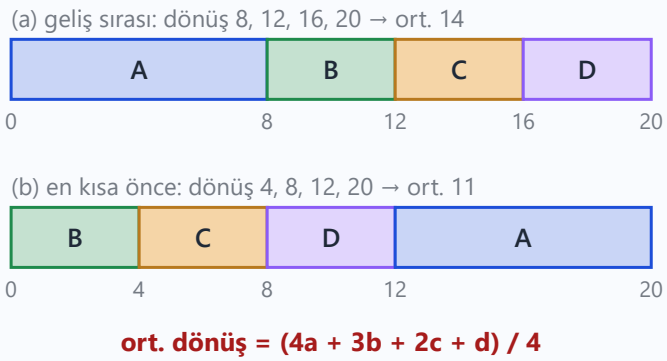
> **Tanım 5.8**
>
> **En kısa iş önce** (SJF, *shortest-job-first*) algoritmasında CPU boşalınca hazır süreçlerden *sonraki CPU patlaması* en kısa olan seçilir; eşitlikte FCFS uygulanır. Bu notta SJF adı kesmesiz sürümü belirtir; kesmeli sürümün adı SRTF'dir.

Algoritmanın adı biraz yanıltıcıdır. Seçimi işin toplam uzunluğu değil, bir sonraki CPU patlamasının uzunluğu belirler. Daha doğru ad “en kısa sonraki CPU patlaması” olurdu.

Teorem 5.1 (SJF'nin optimalliği)

Bütün işler aynı anda hazırsa ve süreleri biliniyorsa, SJF sırası ortalama bekleme süresini (ve ortalama dönüş süresini) en aza indirir.

Gerekçe şudur. Dört iş a, b, c, d sürelerinde ve bu sırayla çalışsın. İlk iş a sürede biter, ikincisi a + b'de, üçüncüsü a + b + c'de, dördüncüsü a + b + c + d'de. Ortalama dönüş (4a + 3b + 2c + d) / 4'tür. İlk işin süresi dört dönüş süresinin hepsine eklenir, sonuncununki yalnız birine. En büyük katsayı a'nındır; toplamı küçültmek için en kısa iş başa konmalıdır. Herhangi bir sırada komşu iki iş kısa olan önde olacak biçimde yer değiştirirse toplam ya azalır ya aynı kalır. Bu yer değiştirmeler sonunda sıra SJF sırasına varır (Şekil 5.14).



Şekil 5.14: Dört iş: A 8, B, C, D 4'er birim. (a) Geliş sırasıyla dönüş süreleri 8, 12, 16, 20, ortalama 14. (b) En kısa iş önce sırasıyla 4, 8, 12, 20, ortalama 11. Ortalama dönüş (4a + 3b + 2c + d) / 4'tür.

Teoremin iki koşulu vardır ve ikisi de önemlidir. Birincisi işlerin aynı anda hazır olmasıdır. İşler farklı zamanlarda geliyorsa kesmesiz SJF optimal değildir: uzun bir iş başladıktan hemen sonra kısa bir iş gelirse SJF onu bekletmek zorundadır. Bu durumda optimal olan, kesmeli sürüm SRTF'dir. İkinci koşul sürelerin bilinmesidir. Toplu sistemlerde kullanıcı işini gönderirken bir süre sınırı verir; Slurm'deki `--time` seçeneği buna örnektir. Kısa sınır veren işin önce çalışma şansı artar, ama süre aşılırsa iş sonlandırılır. Kısa vadeli çizelgelemede ise sonraki patlama hiçbir şekilde bilinemez; ancak geçmiş patlamalardan tahmin edilebilir.

SJF'nin bir de açlık sorunu vardır. Sürekli kısa işler gelirse uzun bir iş hiç seçilmeyebilir. SJF ortalama bekleme ve dönüş süresini en aza indirmeyi hedefler; CPU kullanımını artırmaz. CPU, bu bölümdeki her algoritmada hazır iş olduğu sürece zaten meşguldür.

Örnek 5.4

Dört süreç t = 0'da hazırdır: P1 = 6, P2 = 8, P3 = 7, P4 = 3 ms. SJF ile çizelgeyi ve ortalamaları bulalım; FCFS ile karşılaştıralım.

1. **t = 0.** Hepsi hazır; CPU süresi en kısa olan P4 (3) seçilir.
2. **t = 3.** P4 bitti. Hazır: P1 (6), P2 (8), P3 (7). En kısa P1 (6); 3–9 çalışır.
3. **t = 9.** Hazır: P3 (7), P2 (8). P3 9–16 çalışır.
4. **t = 16.** Yalnız P2 kaldı; 16–24 çalışır.
5. **Ortalamalar.** Beklemeler P1 3, P2 16, P3 9, P4 0; ortalama 28 / 4 = 7 ms. Dönüşler 9, 24, 16, 3; ortalama 13 ms. Aynı küme FCFS ile (P1, P2, P3, P4 sırasıyla) çalışsaydı beklemeler 0, 6, 14, 21 ve ortalama 41 / 4 = 10,25 ms olurdu (Şekil 5.15).

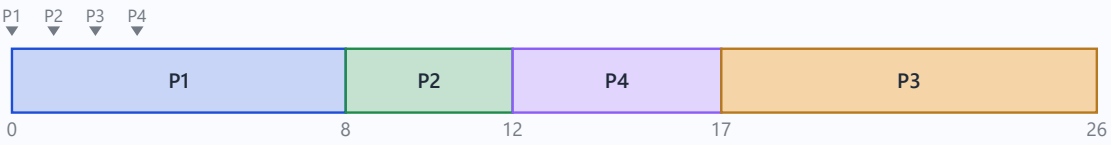


Şekil 5.15: SJF, bütün süreçler 0 anında hazırken: P4 0–3, P1 3–9, P3 9–16, P2 16–24.

5.2.6 SJF ortak kümede ve ağırlıklı dönüş

Ortak kümede süreçler farklı anlarda gelir ve kesmesiz SJF'nin zayıflığı görünür. t = 0'da yalnız P1 vardır ve CPU'yu alır. Sonradan gelen kısa işler P1'i durduramaz.

- **t = 0.** Yalnız P1 hazır; 0–8 çalışır.
- **t = 8.** P1 bitti. Hazır: P2 (4), P3 (9), P4 (5). En kısa P2; 8–12 çalışır.
- **t = 12.** Hazır: P3 (9), P4 (5). En kısa P4; 12–17 çalışır.
- **t = 17.** Yalnız P3 kaldı; 17–26 çalışır (Şekil 5.16; Tablo 5.6).



Şekil 5.16: Kesmesiz SJF ortak kümede: P1 kesilmez; ardından en kısa işler sırayla çalışır. P1 0–8, P2 8–12, P4 12–17, P3 17–26.

Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	0	8	8	0	0
P2	1	4	8	12	11	7	7

Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P3	2	9	17	26	24	15	15
P4	3	5	12	17	14	9	9
Ort.					14,25	7,75	7,75

Tablo 5.6: Kesmesiz SJF ile ortak kümenin ölçütleri.

SJF, FCFS'ye göre ortalama beklemeyi 8,75'ten 7,75'e indirdi. İlk iki süreç FCFS ile aynı çalıştı; fark, P4'ün P3'ün önüne geçmesinden geldi. P4'ün beklemesi 18'den 9'a düştü, P3'ünki 10'dan 15'e çıktı: kısa işe kazandırılan 9 ms, uzun işten alınan 5 ms'den fazladır.

Ağırlıklı dönüş süresi bu takası daha görünür kılar. Tablo 5.7 beş işlik başka bir kümede iki algoritmayı karşılaştırır.

		A	B	C	D	E	Ort.
	Geliş	0	1	2	3	4	
	CPU	4	3	5	2	4	
SJF	Bitiş	4	9	18	6	13	
	Dönüş	4	8	16	3	9	8
	Ağırlıklı	1	2,67	3,2	1,5	2,25	2,12
FCFS	Bitiş	4	7	12	14	18	
	Dönüş	4	6	10	11	14	9
	Ağırlıklı	1	2	2	5,5	3,5	2,8

Tablo 5.7: FCFS ve SJF ile bitiş, dönüş ve ağırlıklı dönüş (dönüş / CPU süresi).

SJF'de A 0–4 çalışır. t = 4'te B (3), C (5), D (2) ve E (4) hazırdır; en kısa D 4–6 çalışır. Sonra B 6–9, E 9–13 ve en son C 13–18 gelir. C'nin ağırlıklı dönüşü 16 / 5 = 3,2'dir. SJF'nin ortalaması 2,12, FCFS'ninki 2,8'dir. FCFS'de en büyük ceza 2 ms'lik D'ye düşer: 11 ms'lik dönüşü CPU süresinin 5,5 katıdır.

FCFS'nin kısa işleri cezalandırması uç bir kümede daha çarpıcıdır. A (geliş 0, CPU 1), B (1, 100), C (2, 1), D (3, 100) kümesinde 1 ms'lik C, B'nin arkasında 99 ms bekler. C'nin dönüşü 100, ağırlıklı dönüşü 100 / 1 = 100 olur. Aynı beklemeye uğrayan D'nin ağırlıklı dönüşü ise yalnız 199 / 100 = 1,99'dur. Uzun iş için 99 ms bekleme önemsizdir, kısa iş için felakettir.

5.2.7 Sonraki patlamayı tahmin: üstel ortalama

SJF'yi kısa vadeli çizelgelemede kullanmanın tek yolu sonraki patlamayı tahmin etmektir. Yaygın yöntem, geçmiş patlamaların üstel ortalamasıdır.

Tanım 5.9

Üstel ortalama (*exponential average*) ile sonraki patlamanın tahmini

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n,\ 0 \le \alpha \le 1$$

formülüyle hesaplanır. Burada t_n, n. patlamanın ölçülen gerçek süresidir; τ_n o patlama için yapılmış tahmindir; τ_0 ise başlangıçta seçilen sabit bir değerdir (örneğin sistem ortalaması).

Formül yeni tahmini iki bilginin karışımı olarak kurar: en son ölçüm ve geçmişin özeti olan eski tahmin. α ikisinin ağırlığını belirler. α = 0 ise son ölçüm hiç hesaba katılmaz ve tahmin hep τ_0 kalır. α = 1 ise yalnız son patlama sayılır; tahmin son ölçülen patlamaya eşittir. Yaygın seçim α = ½'dir: son ölçüm ile geçmiş eşit ağırlık alır.

Adı formül açıldığında anlaşılır:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\alpha\, t_{n-1} + (1-\alpha)^2\,\alpha\, t_{n-2} + \ldots + (1-\alpha)^{n+1}\,\tau_0$$

1 − α birden küçük olduğu için her eski ölçümün ağırlığı bir öncekinin (1 − α) katıdır. α = ½ için son ölçümün ağırlığı ½, bir öncekinin ¼, daha öncekinin ⅛'dir. Tanenbaum aynı fikri **yaşlandırma** (*aging*) adıyla anlatır: tahminler T_0, $T_0/2 + T_1/2$, $T_0/4 + T_1/4 + T_2/2$, … diye ilerler. Bu "yaşlandırma" sözcüğü öncelik çizelgelemedeki yaşlandırmayla karıştırılmamalıdır; ikincisi açlığı önleme yöntemidir.

Aşağıdaki program α = 0,5 ve τ_0 = 10 ile yedi patlamalık bir diziyi işler.

```c
double tau = 10.0;      /* τ0 */
const double a = 0.5;  /* α */
int t[] = { 6, 4, 6, 4, 13, 13, 13 };
for (int n = 0; n < 7; n++) {
  printf("%.1f %d\n", tau, t[n]);
  tau = a * t[n] + (1 - a) * tau;
}
printf("sonraki: %.1f\n", tau);
```

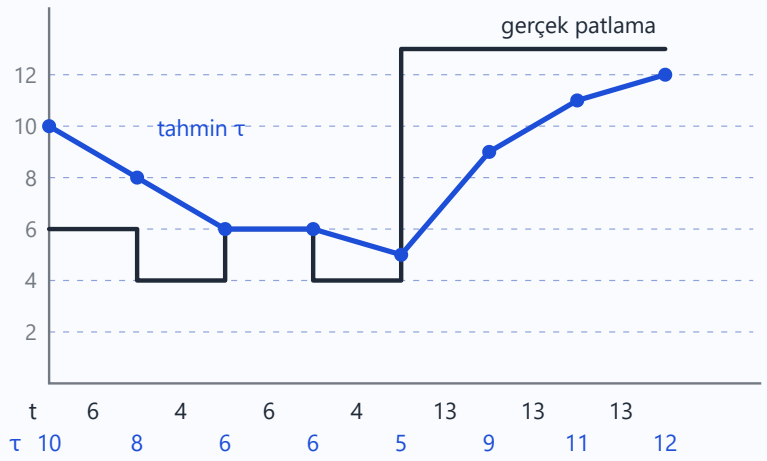
Döngü her adımda önce o patlama için yapılmış tahmini ve gerçek değeri basar, sonra tahmini günceller. Güncelleme satırı formülün birebir karşılığıdır: `a * t[n]` son ölçümün payı, `(1 - a) * tau` eski tahminin payıdır. Döngü bitince yedi ölçümün hepsini görmüş tahmin "sonraki" diye basılır (Tablo 5.8).

n	tahmin τ_n	gerçek t_n	hesap	yeni τ_{n+1}
0	10	6	0,5 · 6 + 0,5 · 10	8
1	8	4	0,5 · 4 + 0,5 · 8	6

n	tahmin τ_n	gerçek t_n	hesap	yeni τ_{n+1}
2	6	6	0,5 · 6 + 0,5 · 6	6
3	6	4	0,5 · 4 + 0,5 · 6	5
4	5	13	0,5 · 13 + 0,5 · 5	9
5	9	13	0,5 · 13 + 0,5 · 9	11
6	11	13	0,5 · 13 + 0,5 · 11	12

Tablo 5.8: Üstel ortalama izlemesi ($\alpha = 0{,}5$, $\tau_0 = 10$).

Programın çıktısı yedi satır `10.0 6`, `8.0 4`, `6.0 6`, `6.0 4`, `5.0 13`, `9.0 13`, `11.0 13` ve son olarak `sonraki: 12.0`'dır. Dizinin ilk yarısında patlamalar 4 ile 6 arasında gidip gelir ve tahmin bu aralığa iner. Patlamalar 13'e sıçrayınca tahmin 5 → 9 → 11 → 12 diye birkaç adımda yetişir; her adımda kalan farkın yarısını kapatır. Tahminin gerçeği bir gecikmeyle izlediği Şekil 5.17'de açıkça görülür.



Şekil 5.17: Üstel ortalama ($\alpha = 0{,}5$, $\tau_0 = 10$). Basamaklı çizgi gerçek patlamaları (6, 4, 6, 4, 13, 13, 13), noktalı eğri tahminleri (10, 8, 6, 6, 5, 9, 11, 12) gösterir.

α'nın seçimi bir ödünleşimdir. Aynı dizi $\alpha = 0{,}75$ ile çalıştırılırsa son tahmin ≈ 12,9 olur ve sıçramaya daha hızlı yetişilir; ama tek bir aykırı ölçüm de tahmini o kadar sarsar. $\alpha = 0{,}25$ ile son tahmin ≈ 10,3'te kalır: tahmin kararlıdır, ama değişime geç uyar.

Örnek 5.5

$\alpha = 0{,}5$ ve $\tau_0 = 12$ ms. Ölçülen ilk üç patlama 8, 6 ve 10 ms. τ_1, τ_2 ve τ_3 nedir? $\alpha = 1$ olsaydı τ_3 kaç olurdu?

1. $\tau_1 = 0{,}5 \cdot 8 + 0{,}5 \cdot 12 = 4 + 6 = 10$ ms.
2. $\tau_2 = 0{,}5 \cdot 6 + 0{,}5 \cdot 10 = 3 + 5 = 8$ ms.
3. $\tau_3 = 0{,}5 \cdot 10 + 0{,}5 \cdot 8 = 5 + 4 = 9$ ms.

4. $\alpha = 1$ ise yalnız son ölçüm sayılır: $\tau_3 = t_2 = 10$ ms. $\alpha = 0$ olsaydı tahmin hiç değişmez, 12'de kalırdı.

Sık yapılan hata

Formüldeki t_n ile τ_n'yi karıştırmak ya da yeni tahmini hep τ_0 ile karıştırarak hesaplamak. Her adımda bir önceki adımın *sonucu* kullanılır. Örnek 5.5'te τ_2 hesaplanırken 12 değil, $\tau_1 = 10$ kullanılır.

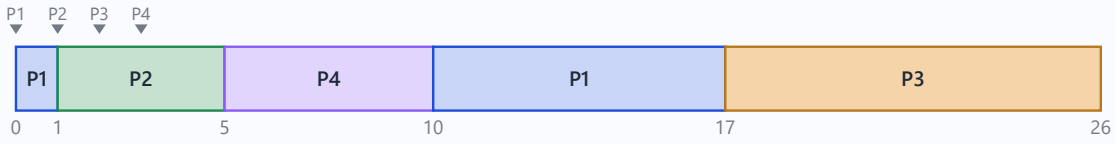
5.2.8 SRTF: en kısa kalan süre önce

Tanım 5.10

En kısa kalan süre önce (SRTF, *shortest-remaining-time-first*; SRTN ya da STCF diye de anılır) SJF'nin kesmeli sürümüdür. Yeni bir süreç geldiğinde onun CPU süresi çalışan sürecin *kalan* süresiyle karşılaştırılır; yeni gelen daha kısaysa çalışan kesilir.

Ortak kümede SRTF izlemesi:

- **t = 0.** Yalnız P1 vardır ve çalışmaya başlar.
- **t = 1.** P2 gelir (4 ms). P1'in kalanı 8 − 1 = 7'dir. 4 < 7 olduğu için P1 kesilir, P2 çalışır.
- **t = 2.** P3 gelir (9). P2'nin kalanı 3'tür ve 3 ≤ 9; kesme yoktur.
- **t = 3.** P4 gelir (5). P2'nin kalanı 2'dir ve 2 ≤ 5; kesme yoktur.
- **t = 5.** P2 biter. Kalan süreler P1 7, P3 9, P4 5; en kısa P4 seçilir ve 5–10 çalışır.
- **t = 10.** P4 biter. Kalanlar P1 7, P3 9; P1 10–17 kaldığı yerden devam eder.
- **t = 17.** P1 biter, P3 17–26 çalışır (Şekil 5.18; Tablo 5.9).



Şekil 5.18: SRTF ortak kümede: P1 0–1, P2 1–5, P4 5–10, P1 10–17, P3 17–26. P1 1'de kesilir ve 10'da kaldığı yerden devam eder.

Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	0	17	17	9	0
P2	1	4	1	5	4	0	0

Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P3	2	9	17	26	24	15	15
P4	3	5	5	10	7	2	2
Ort.					13	6,5	4,25

Tablo 5.9: SRTF ile ortak kümenin ölçütleri.

SRTF bu kümede ortalama beklemeyi 6,5 ms'ye indirir; incelediğimiz algoritmaların en iyisidir. Bedeli dört bağlam değiştirmedir (P1→P2, P2→P4, P4→P1, P1→P3). Kesmeli algoritmalarda tepki ile bekleme ayrışır: P1'in tepki süresi 0, ama bekleme süresi 9'dur, çünkü 1–10 aralığında kuyrukta beklemiştir.

SRTF, işler farklı zamanlarda geldiğinde ortalama bekleme açısından optimaldir; ama SJF'nin iki sorununu da devralır. Kalan süreleri bilmek gerekir ve uzun işler açlık çekebilir. Ayrıca her gelişte karşılaştırma yapıldığı için geçiş sayısı artabilir.

Örnek 5.6

P1 (geliş 0, CPU 7), P2 (2, 4), P3 (4, 1), P4 (5, 4) SRTF ile çizelgeleniyor. Gantt şeması ve ortalama bekleme nedir?

1. **t = 0–2.** Yalnız P1 hazır; 2 ms çalışır, kalanı 5 olur.
2. **t = 2.** P2 gelir: 4 < 5, P1 kesilir. P2 2–4 çalışır, kalanı 2 olur.
3. **t = 4.** P3 gelir: 1 < 2, P2 kesilir. P3 4–5 çalışıp biter.
4. **t = 5.** P4 gelir (4). Kalanlar P1 5, P2 2, P4 4; en kısa P2, 5–7'de biter.
5. **t = 7.** Kalanlar P1 5, P4 4; P4 7–11 çalışır. Son olarak P1 11–16.
6. **Beklemeler.** P1 16 − 0 − 7 = 9, P2 7 − 2 − 4 = 1, P3 0, P4 11 − 5 − 4 = 2. Ortalama 12 / 4 = 3 ms.

5.2.9 Öncelik çizelgeleme

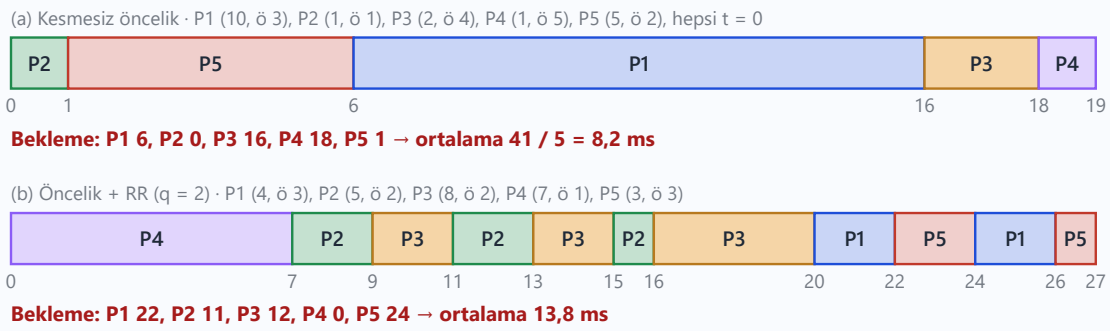
Tanım 5.11

Öncelik çizelgelemede (*priority scheduling*) her sürece bir öncelik sayısı verilir ve hazır süreçlerden önceliği en yüksek olan çalışır. Bu notta küçük sayı yüksek öncelik demektir.

Sayının yönü sistemden sisteme değişir ve bu sık bir karışıklık kaynağıdır. Linux'un `nice` değerinde ve çekirdek içi öncelik ölçeğinde küçük sayı yüksek önceliktir. Windows'ta ise büyük sayı yüksek önceliktir: 31 en yüksek düzeydir.

Öncelikler iç ya da dış etkenlerden gelir. İç öncelikler ölçülebilir büyüklüklere dayanır: bellek gereksinimi, açık dosya sayısı, ölçülen ortalama G/Ç patlamasının CPU patlamasına oranı. Dış öncelikler işletim sisteminin dışında belirlenir: işin önemi, ödenen ücret, kurumun politikası. SJF aslında öncelik çizelgelemenin özel bir durumudur; önceliği tahmin edilen sonraki patlama belirler. Bütün öncelikler eşitse algoritma FCFS'ye döner.

Öncelik çizelgeleme kesmeli ya da kesmesiz olabilir. Kesmeli sürümde yeni gelen süreç çalışandan daha öncelikliyse onu keser. Kesmesiz sürümde yeni süreç kuyruğun başına konur ama çalışanın bitmesini bekler. Aynı öncelik düzeyindeki süreçler arasında RR uygulanabilir; Şekil 5.19(b) bu birleşimi gösterir.



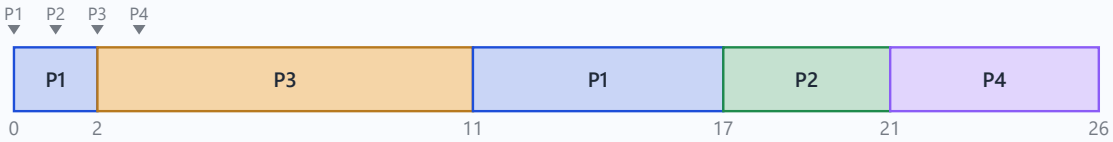
Şekil 5.19: (a) Kesmesiz öncelik, hepsi t = 0'da hazır: P2, P5, P1, P3, P4 sırası; ortalama bekleme 41 / 5 = 8,2 ms. (b) Öncelik ve eşit öncelikliler arasında RR (q = 2): önce tek başına en yüksek öncelikli P4, sonra P2 ile P3 dönüşümlü, en son P1 ile P5 dönüşümlü; ortalama bekleme 69 / 5 = 13,8 ms.

(b) şıkkında öncelik 1'deki P4 tek başına 0–7 çalışır. Öncelik 2'de P2 (5) ile P3 (8) iki milisaniyelik dilimlerle dönüşümlü çalışır: P2 7–9, P3 9–11, P2 11–13, P3 13–15, P2 15–16'da biter, P3 16–20'de biter. Öncelik 3'te P1 (4) ile P5 (3) aynı biçimde 20–27 arasını paylaşır. Beklemeler P1 22, P2 11, P3 12, P4 0, P5 24'tür.

Öncelik çizelgelemenin önemli bir yan etkisi **öncelik terslemesidir** (*priority inversion*). Yüksek öncelikli H süreci, düşük öncelikli L'nin tuttuğu bir kilidi beklesin. L kilidi bırakmak için çalışmalıdır, ama orta öncelikli M süreçleri sürekli L'nin önüne geçer. Sonuçta H, kendisinden düşük öncelikli M'lerin hepsini beklemek zorunda kalır. Çözüm **öncelik kalıtımıdır**: kilidi tutan L, kilit bırakılana kadar bekleyenin önceliğini geçici olarak devralır.

Ortak kümede kesmeli öncelik çizelgeleme şöyle ilerler (öncelikler P1 2, P2 3, P3 1, P4 4):

- **t = 0.** Yalnız P1 vardır ve çalışır.
- **t = 1.** P2 gelir (öncelik 3). Çalışan P1'in önceliği (2) daha yüksektir; kesme yok.
- **t = 2.** P3 gelir (öncelik 1). P1'den yüksektir; P1 kesilir, P3 çalışır.
- **t = 3.** P4 gelir (öncelik 4). P3'ten düşüktür; kesme yok.
- **t = 11.** P3 biter. Hazır: P1 (2), P2 (3), P4 (4). P1 11–17 kalan 6 ms'sini çalışır.
- **t = 17.** P2 17–21, ardından P4 21–26 çalışır (Şekil 5.20; Tablo 5.10).



Şekil 5.20: Kesmeli öncelik ortak kümede: P1 0–2, P3 2–11, P1 11–17, P2 17–21, P4 21–26. Yüksek öncelikli P3 gelince P1 kesilir.

Süreç	Geliş	CPU	Öncelik	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	2	0	17	17	9	0
P2	1	4	3	17	21	20	16	16
P3	2	9	1	2	11	9	0	0
P4	3	5	4	21	26	23	18	18
Ort.						17,25	10,75	8,5

Tablo 5.10: Kesmeli öncelik ile ortak kümenin ölçütleri.

Ortalama bekleme 10,75 ms ile FCFS'den bile kötüdür. Bu şaşırtıcı değildir: öncelik algoritması bekleme süresini değil, önemli işlerin bekletilmemesini hedefler. En önemli P3 hiç beklemedi; en önemsiz P4 18 ms bekledi.

Örnek 5.7

P1 (geliş 0, CPU 5, öncelik 3), P2 (2, 4, 1), P3 (3, 2, 4), P4 (5, 3, 2), P5 (6, 1, 5). Kesmeli öncelik ile Gantt şeması ve ortalamalar nedir?

1. **t = 0–2.** P1 çalışır. t = 2'de gelen P2 (öncelik 1) P1'i keser.
2. **t = 2–6.** P2 çalışır. P3 (4) ve P4 (2) gelir ama P2'den düşüktür; P2 6'da biter.
3. **t = 6.** Hazır: P1 (3), P3 (4), P4 (2), P5 (5). P4 6–9, P1 9–12, P3 12–14, P5 14–15.
4. **Ölçütler.** Bitişler P1 12, P2 6, P3 14, P4 9, P5 15. Dönüşler 12, 4, 11, 4, 9; ortalama 40 / 5 = 8 ms. Beklemeler 7, 0, 9, 1, 8; ortalama 25 / 5 = 5 ms.

Aynı küme kesmesiz öncelikle çalıştırılırsa P1 0–5'te kesilmeden biter, ardından P2 5–9, P4 9–12, P3 12–14, P5 14–15 gelir. Ortalama bekleme 24 / 5 = 4,8 ms'dir; kesmeliden daha iyidir. Kesme ortalamayı her zaman iyileştirmez, yalnız yüksek öncelikli işin bekletilmemesini sağlar.

5.2.10 Açlık ve yaşlandırma

Tanım 5.12

Açlık (*starvation*), bir sürecin sürekli başkalarına yenik düşüp hiç çalışamamasıdır. **Yaşlandırma** (*aging*), hazır kuyruğunda bekleyen sürecin önceliğini zamanla artırarak açlığı önleme yöntemidir.

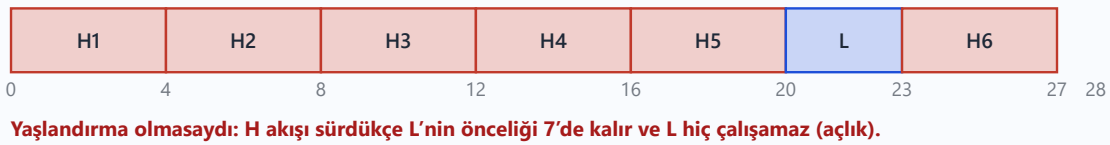
Öncelik çizelgelemede açlık gerçek bir sorundur. Yüksek öncelikli işler sürekli gelirse düşük öncelikli bir iş sonsuza kadar bekleyebilir. Bir söylentiye göre 1973'te MIT'deki IBM 7094 kapatıldığında 1967'de gönderilmiş ve hâlâ çalışmamış düşük öncelikli bir iş bulunmuştu. Yaşlandırma bu beklemeye üst sınır koyar.

Örnek 5.8

Kesmesiz öncelik çizelgeleme kullanılıyor. Kural: hazır kuyruğunda geçen her 4 ms için öncelik sayısı 1 azalır. L (3 ms, öncelik 7) t = 0'da gelir; her 4 ms'de bir yeni bir H işi (4 ms, öncelik 2) gelir. L ne zaman çalışır?

1. **t = 0.** L'nin önceliği 7, yeni gelen H1'inki 2. H1 çalışır. L 4 ms bekleyince önceliği 6 olur.
2. **t = 4, 8, 12, 16.** L'nin önceliği bu anlarda sırayla 6, 5, 4 ve 3'tür. Her seferinde yeni gelen H (öncelik 2) daha yüksektir ve çalışır; L'nin öncelik sayısı her turdan sonra bir azalır.
3. **t = 20.** L'nin önceliği 2'ye inmiştir ve yeni gelen H6 ile eşittir. Eşitlikte önce gelen kazanır: L çalışır ve 23'te biter. L 20 ms beklemiştir.

En kötü durumda L, öncelik farkı × 4 ms = (7 − 2) × 4 = 20 ms bekler. Yaşlandırma olmasaydı H akışı sürdükçe L'nin önceliği 7'de kalır ve L hiç çalışamazdı (Şekil 5.21).



Şekil 5.21: Yaşlandırma sayesinde L 20. ms'de çalışır: H1–H5 4'er ms çalışır, L 20–23, H6 23–27.

Yaşlandırma açlığı ortadan kaldırmaz, ama sonsuz beklemeyi sınırlı bir beklemeye çevirir. Sınır öncelik aralığıyla ve iki artış arasındaki süreyle orantılıdır. Bir sistemde öncelikler 0 ile 63 arasında olsun ve her 10 saniyede bir artırılsın. Önceliği 63 olan süreç hiç çalışmasa bile en geç 63 × 10 = 630 saniyede (10,5 dakika) en yüksek önceliğe ulaşır.

Sık yapılan hata

Yaşlandırma süresini hesaplarken öncelik değerlerini saymak. 63'ten 0'a inmek için 64 değer değil, 63 adım gerekir. Bekleme süresi adım sayısı ile adım aralığının çarpımıdır: 63 × 10 s = 630 s, 640 s değil.

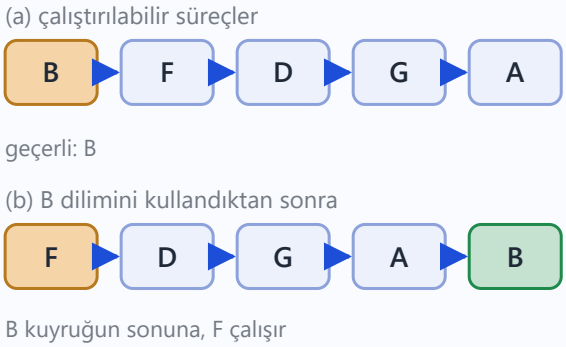
5.3 Round Robin

FCFS adildir ama kesmesizdir; SJF ve SRTF ortalamada iyidir ama süreleri bilmek ister. Etkileşimli sistemler ise süre bilgisi olmadan herkese çabuk tepki veren bir algoritmaya ihtiyaç duyar. Round Robin bu ihtiyaca verilen en eski ve en yaygın cevaptır.

5.3.1 Döngüsel sıra ve zaman dilimi

Tanım 5.13

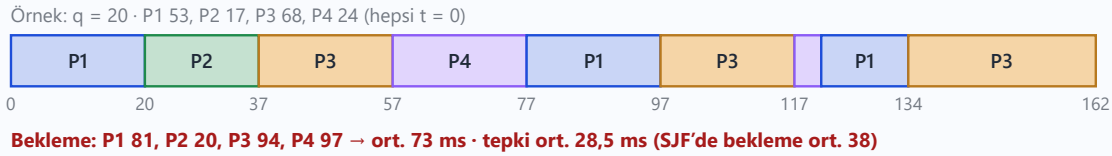
Round Robin (RR, döner sıralı çizelgeleme), FCFS'ye kesme eklenmiş hâlidir. Hazır kuyruğu FIFO'dur; her süreç en fazla bir **zaman dilimi** (*time slice, quantum*) q kadar çalışır. Dilim dolunca saat kesmesi gelir ve süreç kuyruğun sonuna gider. Süreç dilimi dolmadan biterse ya da bloke olursa CPU'yu hemen bırakır (Şekil 5.22).



Şekil 5.22: Round Robin. (a) Çalıştırılabilir süreçler listesi; geçerli süreç B. (b) B dilimini kullandıktan sonra listenin sonuna gider ve sıradaki F çalışır.

RR'nin en değerli özelliği, bekleme için kesin bir üst sınır vermesidir. Kuyrukta n süreç varsa ve dilim q ise hiçbir süreç bir sonraki dilimini $(n - 1) \cdot q$ süreden uzun beklemez. Çünkü önündeki her süreç en fazla bir dilim kullanır. Örneğin 5 süreç ve 20 ms dilimle her süreç en geç 80 ms'de bir kez CPU'ya çıkar. Tepki süresi sınırlıdır ve açlık yoktur.

Bedeli dönüş süresindedir. RR süreçleri dönüşümlü çalıştırdığı için hepsi işin sonuna kadar uzar; dönüş süresi genellikle SJF'den kötüdür. Buna karşılık tepki süresi çok daha iyidir. Silberschatz'ın klasik örneği farkı sayılarla gösterir (Şekil 5.23).



Şekil 5.23: RR (q = 20) ile P1 53, P2 17, P3 68, P4 24 ms; hepsi t = 0'da hazır. Sıra P1, P2, P3, P4, P1, P3, P4 (117–121, şemada etiketsiz dar parça), P1, P3. Ortalama bekleme 73 ms, ortalama tepki 28,5 ms; aynı küme SJF ile çalışsaydı ortalama bekleme 38 ms olurdu.

Bu örnekte P2 ilk turda 17 ms'lik işini bitirir. P4 ikinci turda kalan 4 ms'sini 117–121 arasında kullanır; parça çok dar olduğu için şemada etiketi yoktur. Bitişler P1 134, P2 37, P3 162, P4 121'dir. Beklemeler P1 134 − 53 = 81, P2 20, P3 162 − 68 = 94, P4 121 − 24 = 97; ortalama 292 / 4 = 73 ms. SJF sırası (P2, P4, P1, P3) beklemeleri 0, 17, 41, 94 yapar ve ortalama 38 ms'ye iner. Ama SJF'de P3 ilk kez 94. ms'de CPU'ya çıkar; RR'de ise 37. ms'de.

RR'nin bir zayıflığı da G/Ç'ye bağlı süreçlere karşı tutumudur. G/Ç'ye bağlı süreç dilimin küçük bir kısmını kullanıp bloke olur, G/Ç bitince kuyruğun sonuna döner. CPU'ya bağlı süreçler ise her seferinde dilimin tamamını kullanır. Sonuçta CPU'ya bağlılar CPU'dan daha büyük pay alır. Bu dengesizlik, MLFQ ve "sanal RR" (G/Ç'den dönen sürece ayrı, öncelikli bir yardımcı kuyruk) gibi iyileştirmelerin çıkış noktasıdır.

5.3.2 Ortak kümede q = 2 ve q = 5

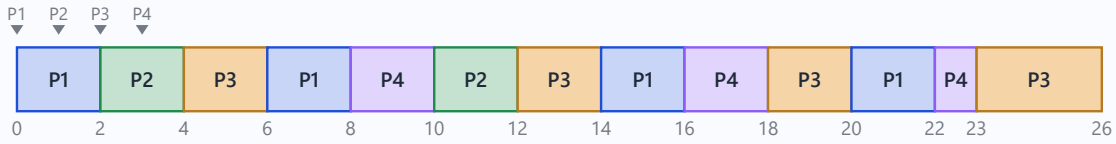
RR'yi ortak kümede iki farklı dilimle izleyelim. Kural hatırlanmalıdır: bir sürecin dilimi dolarken o arada yeni süreçler gelmişse yeniler kuyruğa önce, dilimi dolan süreç sonra girer.

Örnek 5.9

Ortak küme, q = 2 ms. Gantt şeması ve ölçütler nedir?

1. **t = 0–2.** Yalnız P1 vardır ve 2 ms çalışır (kalan 6). Bu arada gelen P2 ve P3 kuyruğa P1'den önce girer: kuyruk P2, P3, P1.
2. **t = 2–4.** P2 çalışır (kalan 2). t = 3'te gelen P4 ondan önce girer: kuyruk P3, P1, P4, P2.
3. **t = 4–6.** P3 (kalan 7). **t = 6–8.** P1 (kalan 4). **t = 8–10.** P4 (kalan 3).
4. **t = 10–12.** P2 son 2 ms'sini kullanır ve 12'de biter.
5. **t = 12–20.** P3 12–14 (kalan 5), P1 14–16 (kalan 2), P4 16–18 (kalan 1), P3 18–20 (kalan 3).

6. **t = 20–26.** P1 20–22'de biter, P4 22–23'te biter. P3 23–25 çalışır ve kalan 1 ms için kuyrukta başka süreç olmadığından hemen yeni dilim alır; 26'da biter (Şekil 5.24; Tablo 5.11).



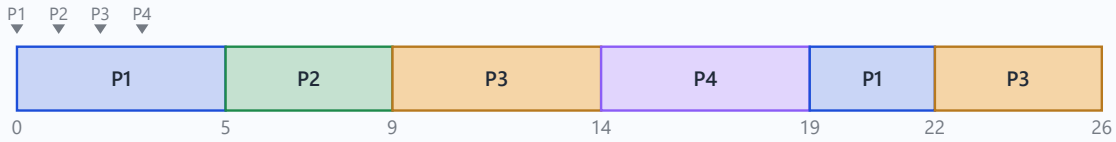
Şekil 5.24: RR q = 2 ortak kümede. Süreçler sık yer değiştirir; herkes kısa sürede ilk kez CPU'ya çıkar.

Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	0	22	22	14	0
P2	1	4	2	12	11	7	1
P3	2	9	4	26	24	15	2
P4	3	5	8	23	20	15	5
Ort.					19,25	12,75	2

Tablo 5.11: RR (q = 2) ile ortak kümenin ölçütleri.

Ortalama tepki 2 ms'ye düştü; bu, şimdiye kadarki en iyi değerdir. Ama ortalama bekleme 12,75 ve ortalama dönüş 19,25 ms ile en kötüsüdür. Gantt şemasında 13 parça ve 12 bağlam değiştirme vardır. Gerçek bir sistemde her geçiş birkaç mikrosaniye yer; bu küçük örnekte sıfır kabul edilse de dilim küçüldükçe yük büyür.

Aynı küme q = 5 ile çalıştırıldığında tablo değişir. P1 0–5 çalışır (kalan 3); bu sürede gelen P2, P3, P4 kuyruğa önce girer. P2 5–9'da dilim dolmadan biter. P3 9–14 (kalan 4), P4 14–19'da biter. P1 19–22'de, P3 22–26'da biter (Şekil 5.25; Tablo 5.12).



Şekil 5.25: RR q = 5 ortak kümede: P1 0–5, P2 5–9, P3 9–14, P4 14–19, P1 19–22, P3 22–26.

Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
P1	0	8	0	22	22	14	0
P2	1	4	5	9	8	4	4
P3	2	9	9	26	24	15	7
P4	3	5	14	19	16	11	11

Süreç	Geliş	CPU	İlk çalışma	Bitiş	Dönüş	Bekleme	Tepki
Ort.					17,5	11	5,5

Tablo 5.12: RR (q = 5) ile ortak kümenin ölçütleri.

Daha büyük dilimle geçiş sayısı 12'den 5'e iner; ortalama bekleme 11'e, dönüş 17,5'e iyileşir. Buna karşılık ortalama tepki 2'den 5,5 ms'ye çıkar. Dilim büyüdükçe RR FCFS'ye yaklaşır.

5.3.3 Kod izleme: `rr()`

RR'nin çekirdeği birkaç satırlık bir döngüdür. Aşağıdaki fonksiyon geliş sırasına göre sıralı diziyi alır ve her sürecin bitiş anını basar. `gelenleri_ekle(p, n, &k, t)` yardımcı fonksiyonu, `t` anına kadar gelmiş ama henüz kuyruğa girmemiş süreçleri sırayla kuyruğa ekler. `k`, dizide eklenecek sıradaki sürecin indisidir.

```
/* p[] geliş sırasına göre sıralı */
void rr(Surec p[], int n, int q) {
  int t = 0, k = 0;     /* k: sıradaki */
  gelenleri_ekle(p, n, &k, t);
  while (!bos() || k < n) {
    if (bos()) {         /* CPU boşta */
      t = p[k].gelis;    /* sonraki geliş */
      gelenleri_ekle(p, n, &k, t);
    }
    Surec *s = cikar();  /* baştaki */
    int dilim = min(q, s->kalan);
    t += dilim; s->kalan -= dilim;
    gelenleri_ekle(p, n, &k, t); /* önce */
    if (s->kalan > 0) ekle(s);   /* sonra */
    else printf("%s %d\n", s->ad, t);
  }
}
```

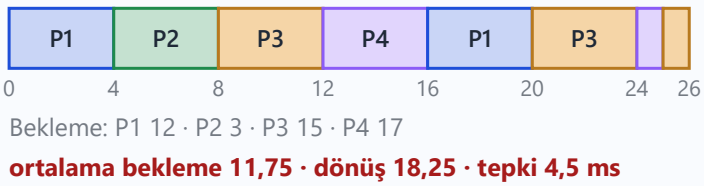
Döngü, kuyruk boşalsa bile henüz gelmemiş süreç kaldıkça (`k < n`) sürer. Kuyruk boşken CPU boşta kalır; kod saati doğrudan sıradaki sürecin geliş anına ilerletir ve o süreci kuyruğa ekler. Bu satırlar olmasaydı döngü erken biter ve sonra gelen süreçler hiç çalışmazdı. Ortak kümede CPU hiç boşta kalmadığı için bu dal çalışmaz. Döngünün her turu bir dağıtımdır. Kuyruğun başındaki süreç çıkarılır ve `min(q, s->kalan)` kadar çalıştırılır: kalan süre dilimden kısaysa süreç dilimi doldurmadan biter. Saat ilerletilir, kalan süre azaltılır. Ardından gelen iki satırın sırası RR'nin bağ çözme kuralını belirler. Önce bu dilim sırasında gelen yeni süreçler eklenir, sonra işi bitmeyen süreç kuyruğun sonuna döner. İki satır yer değiştirseydi dilimi dolan süreç yeni gelenlerin önüne geçerdi ve çizelge farklı olurdu. İşi biten süreç kuyruğa dönmez; adı ve bitiş anı basılır (Tablo 5.13).

Tur	Çalışan	`dilim`	`t`	Kalan	Kuyruk (baş → son)	Çıktı
başta	—	—	0	—	P1	

Tur	Çalışan	`dilim`	`t`	Kalan	Kuyruk (baş → son)	Çıktı
1	P1	4	4	4	P2, P3, P4, P1	
2	P2	4	8	0	P3, P4, P1	`P2 8`
3	P3	4	12	5	P4, P1, P3	
4	P4	4	16	1	P1, P3, P4	
5	P1	4	20	0	P3, P4	`P1 20`
6	P3	4	24	1	P4, P3	
7	P4	1	25	0	P3	`P4 25`
8	P3	1	26	0	boş	`P3 26`

Tablo 5.13: `rr()` ortak kümede, q = 4. Her satır döngü turunun sonundaki durumu gösterir.

Kuyruk boşalıp gelecek süreç kalmayınca döngü biter: 8 dağıtım ve 7 bağlam değiştirme. Beklemeler P1 20 − 8 = 12, P2 8 − 1 − 4 = 3, P3 26 − 2 − 9 = 15, P4 25 − 3 − 5 = 17; ortalama 11,75 ms. Ortalama dönüş 18,25, ortalama tepki (0 + 3 + 6 + 9) / 4 = 4,5 ms'dir (Şekil 5.26).



Şekil 5.26: `rr()` izlemesinin sonucu (q = 4): P1 0–4, P2 4–8, P3 8–12, P4 12–16, P1 16–20, P3 20–24, P4 24–25, P3 25–26.

Her tur O(1) zamanda çalışır: kuyruktan çıkarma ve kuyruğa ekleme sabit sürelidir. Toplam tur sayısı, her sürecin CPU süresinin q'ya bölünüp yukarı yuvarlanmasının toplamıdır. Bu örnekte 2 + 1 + 3 + 2 = 8'dir. Dolayısıyla q küçüldükçe dağıtım sayısı ters orantılı olarak büyür.

Sık yapılan hata

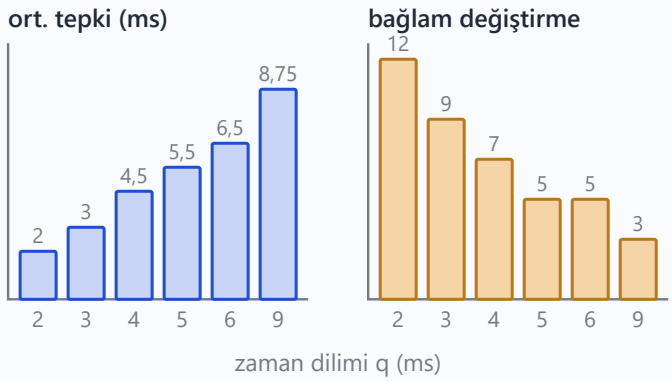
Dilimi dolan süreci, aynı anda gelen yeni süreçlerin önüne koymak. Bu notta ve sunumda yeni gelenler önce girer; kodda bu, `gelenleri_ekle` satırının `ekle(s)` satırından önce gelmesiyle sağlanır. Ders kitapları bu konuda farklı kurallar kullanabilir; bir RR sorusunda önce kural belirlenmelidir.

5.3.4 Zaman dilimi seçimi

RR'nin tek parametresi q'dur ve davranışı büyük ölçüde ona bağlıdır. Ortak küme farklı dilimlerle çalıştırıldığında Tablo 5.14 elde edilir (Şekil 5.27).

Zaman dilimi	Bekleme	Dönüş	Tepki	Geçiş
q = 2	12,75	19,25	2	12
q = 3	13,5	20	3	9
q = 4	11,75	18,25	4,5	7
q = 5	11	17,5	5,5	5
q = 6	12	18,5	6,5	5
q ≥ 9 (FCFS)	8,75	15,25	8,75	3

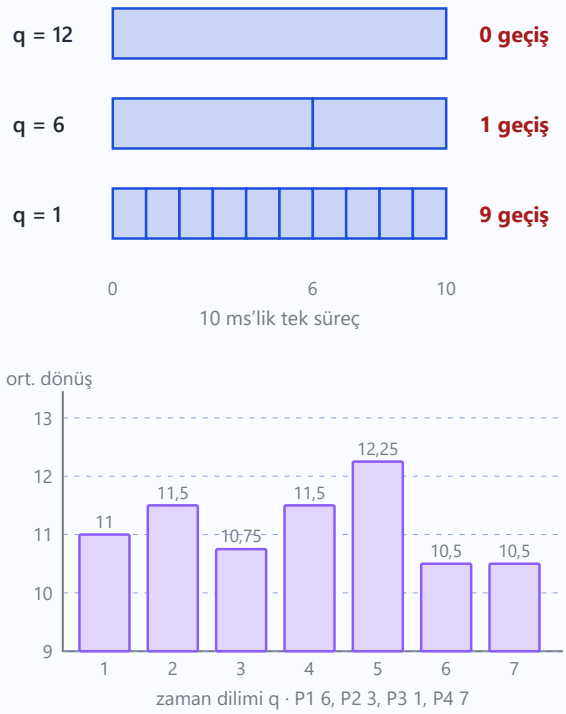
Tablo 5.14: Ortak kümede zaman dilimine göre ortalamalar (ms) ve bağlam değiştirme sayısı.



Şekil 5.27: Zaman dilimi büyüdükçe ortalama tepki süresi artar, bağlam değiştirme sayısı azalır.

Tablodan üç sonuç çıkar. Birincisi, q küçüldükçe tepki iyileşir ama geçiş sayısı artar. İkincisi, ortalama bekleme ve dönüş q ile tekdüze değişmez: q = 3, q = 2'den de kötüdür; q = 5, q = 6'dan iyidir. Üçüncüsü, q en uzun patlamaya (burada P3'ün 9 ms'si) eşit ya da ondan büyük olunca hiçbir süreç kesilmez ve RR tam olarak FCFS olur.

Dönüş süresinin q ile tekdüze değişmediği başka bir kümede de görülür. P1 6, P2 3, P3 1, P4 7 ms (hepsi t = 0'da) için ortalama dönüş q = 1'den 7'ye 11; 11,5; 10,75; 11,5; 12,25; 10,5; 10,5 olur (Şekil 5.28, sağ). Genel kural şudur: süreçlerin çoğu işini tek dilimde bitirebiliyorsa ortalama dönüş iyileşir.



Şekil 5.28: Solda 10 ms'lik tek bir süreç: q = 12 iken geçiş yok, q = 6 iken 1, q = 1 iken 9 geçiş. Sağda P1 6, P2 3, P3 1, P4 7 kümesinde ortalama dönüş süresinin q'ya göre değişimi.

Gerçek sistemde her bağlam değiştirme s süre tutarsa, dilimini sonuna kadar kullanan süreçlerle CPU'nun boşa giden oranı

> kayıp oranı = s / (q + s)

olur. Her turda q kadar yararlı iş ve s kadar geçiş vardır. q = 4 ms ve s = 1 ms ise kayıp 1 / 5 = %20'dir: CPU zamanının beşte biri yalnız süreç değiştirmeye gider. q = 100 ms'de kayıp ≈ %1'e iner, ama on süreçlik bir kuyrukta her süreç bir saniyeye yakın bekler ve sistem ağır tepki verir.

Sık yapılan hata

Kayıp oranını s / q diye hesaplamak. Payda turun tamamıdır, yani yararlı iş ile geçişin toplamı: q + s. q = 4, s = 1 için s / q = %25 bulunur; doğru değer %20'dir.

Bugün bağlam değiştirme birkaç mikrosaniye sürer, tipik dilimler 1–100 ms arasındadır. Bu yüzden doğrudan geçiş yükü küçüktür; asıl maliyet önbelleğin soğumasıdır. Linux'un EEVDF çizelgeleyicisinde temel dilim 0,75 ms × (1 + $\log_2$ CPU sayısı) formülüyle hesaplanır. Formülde CPU sayısı en fazla 8 alınır; bu yüzden 8 ve daha çok çekirdekte dilim 3 ms'dir. Windows istemci sürümlerinde dilim yakla-

şık iki saat tıkıdır. Silberschatz'ın pratik kuralı şudur: CPU patlamalarının yaklaşık %80'i q'dan kısa olmalıdır. Böylece süreçlerin çoğu dilimini doldurmadan G/Ç'ye gider; dilim yalnız uzun CPU işlerini sınırlar.

Bir örnek, iyi seçilmiş bir dilimin konvoy etkisini nasıl kırdığını gösterir. P1 24, P2 3, P3 3 ms kümesinde FCFS ortalama bekleme 17 ms idi. q = 4 ile P1 0–4, P2 4–7, P3 7–10 ve P1 10–30 çalışır. Beklemeler 6, 4 ve 7, ortalama 17 / 3 ≈ 5,67 ms'dir.

Örnek 5.10

A, B, C, D t = 0'da bu sırayla gelir; her biri 4 ms CPU ister. FCFS ve q = 1'lik RR için ortalama dönüş ve tepki süresi nedir?

1. **FCFS.** Bitişler 4, 8, 12, 16; ortalama dönüş 40 / 4 = 10 ms. Tepki (0 + 4 + 8 + 12) / 4 = 6 ms. Üç geçiş.
2. **RR (q = 1).** Süreçler her milisaniye yer değiştirir. Son tur 12–16 arasındadır; bitişler 13, 14, 15, 16 ve ortalama dönüş 58 / 4 = 14,5 ms. Tepki (0 + 1 + 2 + 3) / 4 = 1,5 ms. On beş geçiş.

Eşit uzunluktaki işlerde RR tepkide en iyi, dönüşte en kötü seçeneklerdendir: herkes işin sonuna kadar bekletilir.

5.4 Çok düzeyli kuyruklar

Gerçek bir sistemde süreçler türdeş değildir. Ses çalan bir süreç, kullanıcının yazdığı metin düzenleyici ve gece yedeği alan bir betik aynı kuyrukta adil biçimde yarışmamalıdır. Çok düzeyli kuyruklar süreçleri sınıflara ayırır ve her sınıfa kendi kuyruğunu ve kuralını verir.

5.4.1 Çok düzeyli kuyruk (MLQ)

Tanım 5.14

Çok düzeyli kuyruk (MLQ, *multilevel queue*) çizelgelemede hazır kuyruğu birkaç ayrı kuyruğa bölünür. Her süreç türüne göre (örneğin ön plan ya da arka plan) *kalıcı olarak* bir kuyruğa atanır. Her kuyruğun kendi algoritması vardır; kuyruklar arasında da ayrı bir çizelgeleme yapılır (Şekil 5.29).



Şekil 5.29: Çok düzeyli kuyruk: gerçek zamanlı, sistem, etkileşimli ve toplu süreçler için ayrı kuyruklar; öncelik yukarıdan aşağıya azalır.

Tipik bir düzende ön plandaki etkileşimli süreçler kısa dilimli RR ile, arka plandaki toplu süreçler FCFS ile çizelgelenir. Kuyruklar arası seçim için iki yol vardır. **Sabit öncelikte** üst kuyruk boşalmadan alt kuyruk hiç çalışmaz. Toplu bir iş, etkileşimli kuyrukta süreç olduğu sürece bekler ve açlık çekebilir. **Zaman paylaşımında** her kuyruğa CPU'nun belirli bir oranı verilir; örneğin %80 ön plana, %20 arka plana.

MLQ'nun yükü düşüktür: süreç kuyruğa bir kez atanır ve bir daha taşınmaz. Ama esnek değildir. Bir süreç zamanla davranış değiştirse bile kuyruğu değişmez. Sınıfı yanlış tahmin edilen bir süreç ya hak etmediği önceliği alır ya da hak ettiğinden yoksun kalır. Bu sabit atamalı yapı MLQ'dur; süreçlerin davranışına göre kuyruk değiştirdiği yapıya geri beslemeli kuyruk denir.

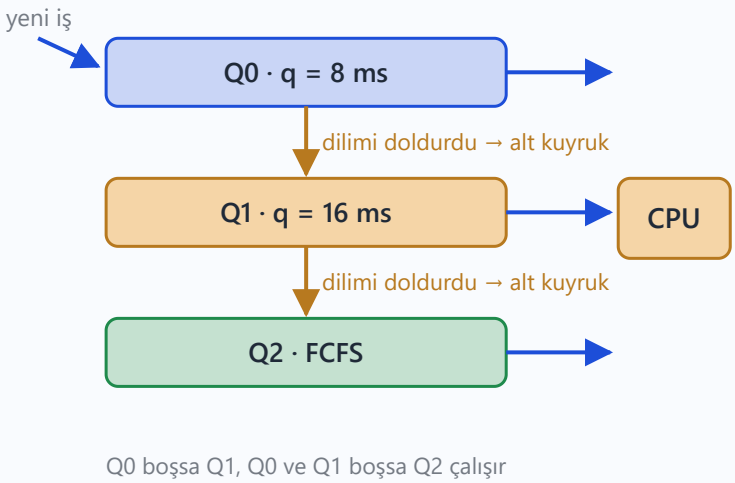
5.4.2 Geri beslemeli kuyruk (MLFQ) ve kuralları

MLFQ, bölümün en başında sorulan soruya pratik bir cevap verir: patlama sürelerini bilmeden kısa işleri nasıl öne alırız? Cevap, süreçleri gözlemektir. Her yeni iş önce kısa sayılır ve en üst kuyruğa konur. Gerçekten kısaysa orada hızla biter. Uzunsa dilimini doldurur ve bir alt kuyruğa iner. Böylece algoritma geçmişten öğrenir ve SJF'ye yaklaşır.

Tanım 5.15

Çok düzeyli geri beslemeli kuyruk (MLFQ, *multilevel feedback queue*), süreçlerin gözlenen davranışlarına göre kuyruklar arasında taşındığı çok düzeyli kuyruktur. OSTEP'in düzenlemesiyle beş kuralı vardır:

1. Öncelik(A) > Öncelik(B) ise A çalışır.
2. Öncelikler eşitse A ve B RR ile çalışır.
3. Yeni iş en üst kuyruğa girer.
4. Bir düzeydeki hakkını (*allotment*) bitiren iş, CPU'yu kaç kez bırakmış olursa olsun bir alt düzeye iner.
5. Her S sürede bir bütün işler en üst kuyruğa taşınır (**öncelik yükseltme**; Şekil 5.30).



Şekil 5.30: MLFQ: yeni iş Q0'a girer; dilimini dolduran iş bir alt kuyruğa iner. Q0 boşsa Q1, Q0 ve Q1 boşsa Q2 çalışır.

Kuralların her biri bir sorunu çözer. İlk ikisi temel çizelgeyi kurar. Üçüncüsü yeni işlere tanınan avantajdır: kısa bir komut hemen çalışıp biter. Dördüncüsü uzun işleri aşağı iter; hakkın düzeydeki *toplam* kullanım olarak sayılması kandırılmayı önler. Beşincisi açlığı önler ve davranışı değişen işe yeni bir şans verir.

MLFQ'yu tanımlayan parametreler şunlardır: kuyruk sayısı, her kuyruğun algoritması ve dilimi, işin ne zaman alt ya da üst kuyruğa geçeceği ve yeni işin hangi kuyruğa gireceği. Genellikle üst kuyruklarda dilim kısa, alt kuyruklarda uzundur: üstte kısa işler hızla döner, altta uzun işler daha seyrek ama daha uzun çalışır. Bu esneklik MLFQ'yu en genel ama aynı zamanda ayarlaması en zor algoritma yapar.

5.4.3 MLFQ adım adım

Örnek 5.11

Üç kuyruk: Q0 (q = 8), Q1 (q = 16), Q2 (FCFS). İşler (geliş, CPU): A (0, 30), B (2, 6), C (4, 20), D (25, 4). Üst kuyruğa iş gelince kesilen iş kendi kuyruğunun başına döner ve dilimini tamamlar.

1. **t = 0.** A gelir, Q0'a girer ve 8 ms'lik dilimle çalışır. Bu arada B (t = 2) ve C (t = 4) Q0'a eklenir.
2. **t = 8.** A dilimini doldurdu ve Q1'e iner (kalan 22). Q0'da sıradaki B çalışır.
3. **t = 14.** B 6 ms'de, dilimi dolmadan Q0'da biter. C Q0'da 8 ms çalışır.
4. **t = 22.** C de Q1'e iner (kalan 12). Q0 boştur; Q1'in başındaki A 16 ms'lik dilimle çalışmaya başlar.
5. **t = 25.** D gelir ve Q0'a girer. Q0, Q1'den üstte olduğu için A kesilir ve Q1'in başına döner; dilimin 3 ms'sini kullanmıştır, 13 ms'si kalmıştır.
6. **t = 29.** D 4 ms'de biter; hiç beklememiştir. A, Q1'de dilimin kalan 13 ms'sini kullanır.
7. **t = 42.** A Q1 dilimini de doldurdu ve Q2'ye iner (kalan 6). C Q1'de çalışır; kalan 12 ms'si 16 ms'lik dilime sığar ve 54'te biter.
8. **t = 54.** Q0 ve Q1 boştur. A, Q2'de FCFS ile kalan 6 ms'yi çalışır ve 60'ta biter (Şekil 5.31; Tablo 5.15).



Şekil 5.31: MLFQ izlemesinin Gantt şeması; her parçanın etiketi işi ve çalıştığı kuyruğu gösterir: A·Q0 0–8, B·Q0 8–14, C·Q0 14–22, A 22–25, D 25–29, A·Q1 29–42, C·Q1 42–54, A·Q2 54–60.

İş	Geliş	CPU	Bitiş	Dönüş	Bekleme	Tepki
A	0	30	60	60	30	0
B	2	6	14	12	6	6
C	4	20	54	50	30	10
D	25	4	29	4	0	0
Ort.				31,5	16,5	4

Tablo 5.15: MLFQ örneğinin ölçütleri.

Kısa B ve D üst kuyrukta hızla biter; tepki süreleri 6 ve 0'dır. Uzun A aşağı süzülür ve en son biter. Ortalama tepki 4 ms'dir. MLFQ hiçbir süreç süresini bilmeden kısa işleri öne almıştır: yaptığı tek şey her işin ne kadar CPU kullandığını saymaktır.

Örnek 5.12

Aynı kurallarla Q0 (q = 4), Q1 (q = 8), Q2 (FCFS) ve A (0, 14), B (1, 3), C (3, 9), D (16, 2). İşler ne zaman biter?

1. A 0–4 dilimini doldurur ve Q1'e iner. B 4–7'de dilim dolmadan biter (tepki 3). C 7–11 dilimini doldurur ve Q1'e iner.
2. Q0 boştur; Q1'in başındaki A 11'de 8 ms'lik dilimle başlar. t = 16'da D gelir; A dilimin 5 ms'sini kullanmışken kesilir ve Q1'in başına döner.
3. D 16–18'de biter. A dilimin kalan 3 ms'sini 18–21'de kullanır; Q0 ve Q1'de toplam 4 + 8 = 12 ms çalışmıştır ve Q2'ye iner.
4. C Q1'de kalan 5 ms'sini 21–26'da bitirir. A Q2'de kalan 2 ms'yi 26–28'de çalışır.

Bitişler: B 7, D 18, C 26, A 28. Ortalama tepki (0 + 3 + 4 + 0) / 4 = 1,75 ms'dir.

5.4.4 MLFQ uygulamada

MLFQ'nun kurallarının neden böyle olduğu, olmadığı zaman ortaya çıkan sorunlardan anlaşılır.

Oyun oynama (*gaming*). Kuralın ilk hâli "dilimini dolduran iş aşağı iner, dilimden önce CPU'yu bırakan aynı düzeyde kalır" olsaydı kötü niyetli bir program bundan yararlanabilirdi. Program dilim bitmeden hemen önce kısa bir G/Ç yapar, örneğin 10 ms'lik dilimin 9,9 ms'sini kullanır. Hiç alt kuyruğa inmez ve CPU'nun büyük kısmını tekeline alır. Çözüm hakkı düzeydeki toplam kullanım olarak saymaktır (kural 4). İş CPU'yu kaç parçada kullanırsa kullansın, toplam 10 ms dolunca aşağı iner (Şekil 5.32).



Şekil 5.32: MLFQ'yu kandırmak. Üstte iş her dilimin 9,9 ms'sini kullanıp G/Ç yapar ve hep Q0'da kalır. Altta kural 4 ile düzeydeki toplam 10 ms dolunca bir alt düzeye iner.

Açlık ve davranış değişimi. Çok sayıda etkileşimli iş varsa üst kuyruklar hiç boşalmaz ve alt kuyruktaki uzun işler aç kalır. Ayrıca uzun bir hesap işi sonradan etkileşimli hâle gelirse alt kuyrukta kaldığı için bu fark edilmez. Periyodik öncelik yükseltme (kural 5) iki sorunu birlikte çözer. S'nin seçimi zordur. Çok büyük seçilirse uzun işler yine aç kalır; çok küçük seçilirse etkileşimli işler CPU'yu yeterince alamaz. OSTEP bu tür değerleri "vudu sabitleri" diye anar.

Tarihçe. MLFQ fikrini ilk kez Corbató ve ekibi 1962'de MIT'deki CTSS sisteminde uyguladı. CTSS'te öncelik sınıfları 1, 2, 4 ve 8 dilim çalışırdı; dilimlerini bitiren iş bir alt sınıfa inerdi. Terminalde Enter tuşuna basılınca süreç en üst sınıfa taşınıyordu, çünkü sistem onun etkileşimli hâle geldiğini varsayı-

yordu. Kullanıcılar bunu kısa sürede fark etti ve uzun işlerini hızlandırmak için arada bir Enter'a bastı. Oyun oynama sorununun ilk örneklerinden biri budur.

Bugün. MLFQ fikri birçok gerçek sistemde yaşar. Solaris'in zaman paylaşımlı (TS) sınıfı her öncelik düzeyinin dilimini ve aşağı ya da yukarı geçiş kuralını bir tabloyla tanımlar. FreeBSD'nin ULE çizelgeleyicisi, Windows'un dinamik öncelikleri ve macOS da aynı fikri kullanır. Linux ise 2.6.23 sürümünden beri bu tür öncelik sezgileri yerine adil paylaşıma dayanır; önce CFS, sonra EEVDF ile.

Sık yapılan hata

Öncelik yükseltmenin MLFQ'nun kandırılmasını önlediğini sanmak. Öncelik yükseltme açlığı önler; bir işin dilimin hep 9,9 ms'sini kullanıp üstte kalmasını engelleyen kural, düzeydeki toplam kullanımın sayılmasıdır. G/Ç yapan her işi doğrudan en alta göndermek de çözüm değildir: gerçekten etkileşimli olan işler cezalandırılır.

5.5 Orantılı pay

Şimdiye kadarki algoritmalar dönüş ya da tepki süresini iyileştirmeye çalıştı. Orantılı pay (*proportional share*) çizelgeleyicileri başka bir hedef koyar: her süreç ya da kullanıcı CPU'nun önceden belirlenmiş bir payını almalıdır. Bulut sunucusunda müşteri parasını ödediği payı, paylaşımlı bir makinede her kullanıcı adil payını ister.

5.5.1 Garantili ve adil paylaşım

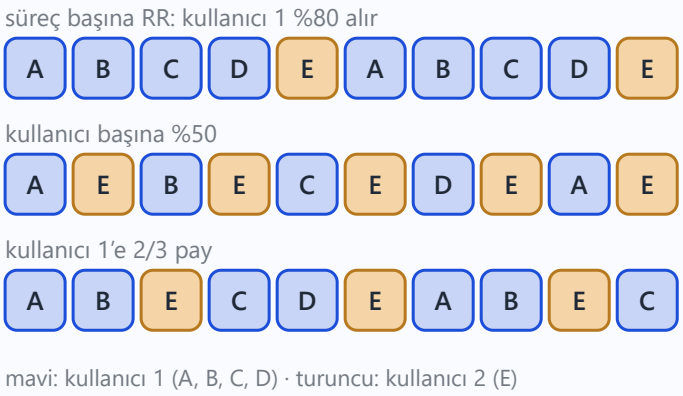
Tanım 5.16

Garantili çizelgelemede n kullanıcı (ya da süreç) varsa her birine CPU'nun 1/n'i söz verilir. Çizelgeleyici her biri için *oran = aldığı CPU / hak ettiği CPU* değerini hesaplar ve oranı en düşük olanı çalıştırır. **Adil paylaşımda** (*fair-share scheduling*) pay süreçlere değil, kullanıcılara ya da gruplara verilir.

Garantili çizelgelemede oranı 0,5 olan süreç payının yalnız yarısını almıştır ve öne geçer. Oranı 2,0 olan süreç payının iki katını almıştır ve oranı düşük olanlara yol verir. Hesap için çizelgeleyicinin her sürecin ne zamandan beri var olduğunu ve o zamandan beri ne kadar CPU kullandığını tutması yeterlidir. Tanenbaum'daki anlamıyla garantili çizelgeleme bu pay garantisidir; önceden belirlenmiş bir zaman programına göre çalışmak değildir.

Süreç başına adalet, çok süreç açan kullanıcıyı ödüllendirir. Kullanıcı 1 dört süreç (A, B, C, D), kullanıcı 2 tek süreç (E) çalıştırsın. Süreç başına RR'de beş süreç eşit pay alır ve kullanıcı 1 CPU'nun %80'ini alır. Adil paylaşımda pay önce kullanıcılara bölünür: her kullanıcıya %50. Kullanıcı 1'in payı kendi

dört süreci arasında bölünür, kullanıcı 2'ninki E'ye gider. Sıra A, E, B, E, C, E, D, E biçimini alır. Paylar politikayla da belirlenebilir; kullanıcı 1'e 2/3 pay verilirse her üç dilimin ikisi onun süreçlerine gider (Şekil 5.33).



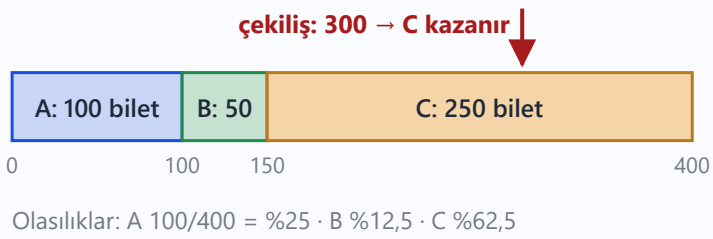
Şekil 5.33: Adil paylaşım. Kullanıcı 1 dört süreç (A–D), kullanıcı 2 tek süreç (E) çalıştırır. Süreç başına RR'de kullanıcı 1 %80 alır; kullanıcı başına paylaşımda ikisi %50 alır; politika gerektirirse kullanıcı 1'e 2/3 pay verilebilir.

Bugün adil paylaşım Linux'ta denetim grupları (*cgroups*) ile yapılır. cgroup v2'de her grubun `cpu.weight` değeri vardır ve CPU grup ağırlıklarına göre bölünür. systemd her kullanıcıyı `user-UID.slice` adlı bir gruba koyar; bu yüzden bir kullanıcının yüz süreç açması öteki kullanıcının payını azaltmaz. Kubernetes de kapsayıcılara verilen CPU isteklerini bu ağırlıklara çevirir.

5.5.2 Piyango çizelgeleme

Tanım 5.17

Piyango çizelgelemede (*lottery scheduling*) her sürece bilet verilir. Her dilimde rastgele bir bilet çekilir ve biletin sahibi çalışır. Uzun vadede bir sürecin CPU payı, biletlerinin toplam bilete oranıdır (Şekil 5.34).



Şekil 5.34: Piyango. A 0–99, B 100–149, C 150–399 numaralı biletleri tutar; çekilen 300 C'nin aralığındadır. Olasılıklar A 100/400 = %25, B %12,5, C %62,5.

Waldspurger ve Weihl'in 1994'te önerdiği yöntemin en çekici yanı basitliğidir. Gereken tek şey bir rastgele sayı üreteci ve süreç listesi üzerinde birikimli bir toplamdır. Önceliği ifade etmek de kolaydır: daha önemli süreç daha çok bilet alır. Yeni gelen süreç bilet alır almaz kazanabilir; algoritma bu anlamda hemen tepki verir.

Biletler, adil paylaşımın zor işlerini de çözer. **Bilet devri**nde bir istemci sunucudan yanıt beklerken kendi biletlerini geçici olarak sunucuya verir; böylece sunucu, en çok bekleyen istemcinin işine hız verir. **Bilet para birimi**, bir kullanıcının kendi biletlerini kendi süreçleri arasında istediği birimle bölmesine izin verir; sistem bunu küresel değere çevirir. Birbirine güvenen süreçlerin olduğu bir ortamda **enflasyon** da kullanılabilir: daha çok CPU'ya ihtiyacı olan süreç kendi bilet sayısını geçici olarak artırır.

Piyangonun zayıflığı olasılıksal olmasıdır. Kısa vadede adaletsiz olabilir: eşit biletli iki iş 10 dilimi 7–3 bölüşebilir. Pay ancak çekiliş sayısı arttıkça bilet oranına yaklaşır. Aynı oranı deterministik olarak sağlayan yöntem stride çizelgelemedir.

Örnek 5.13

A'nın 50, B'nin 30, C'nin 20 bileti var. C'ye 20 bilet daha verilirse 240 dilimde C'nin beklenen dilim sayısı kaçtır?

1. Toplam bilet 50 + 30 + 40 = 120 olur.
2. C'nin payı 40 / 120 = 1/3'tür; 240 dilimde beklenen değer 240 / 3 = 80'dir.

Eski payla (%20) hesaplamak 48, toplamı yanlışlıkla 100 sanmak %40 ve 96 verir. 80 bir beklenen değerdir; tek bir çalışmada 75 ya da 86 da görülebilir.

5.5.3 Kod izleme: çekiliş döngüsü

OSTEP piyangonun karar döngüsünü şöyle verir. Süreçler bilet sayılarıyla birlikte bir bağlı listededir; `rastgele_al(0, totaltickets)` 0 ile toplam bilet arasında bir sayı döndürür. OSTEP bu fonksiyona `getrandom` der; aynı adda bir Linux sistem çağrısı bulunduğu için burada Türkçe ad kullanılır.

```
/* her dilimde bir çekiliş */
int counter = 0;
int winner = rastgele_al(0, totaltickets);
node_t *current = head;
while (current) {
  counter = counter + current->tickets;
  if (counter > winner)
    break;          /* kazanan bulundu */
  current = current->next;
}
/* current: CPU'yu alacak süreç */
calistir(current);
```

Döngü listeyi baştan tarar ve `counter` değişkeninde o ana kadarki biletlerin toplamını tutar. Her düğümde düğümün biletleri toplama eklenir. Toplam çekilen sayıyı aştığı anda çekilen bilet bu düğümün aralığındadır ve döngü kırılır. Aşmıyorsa sonraki düğüme geçilir. İzleme için çekilen sayılar sırayla 47, 35 ve 12 olarak sabitlenmiştir; bilet listesi A 30, B 10, C 60'tır (toplam 100). A 0–29, B 30–39, C 40–99 numaralı biletleri tutar (Tablo 5.16).

Çekiliş	Düğüm	`counter`	`counter > winner` ?	Sonuç
47	A	30	hayır	sonraki düğüm
47	B	40	hayır	sonraki düğüm
47	C	100	evet	C kazanır
35	A	30	hayır	sonraki düğüm
35	B	40	evet	B kazanır
12	A	30	evet	A kazanır

Tablo 5.16: Üç çekilişte karar döngüsü (bilet listesi A 30, B 10, C 60).

Her yeni çekilişte sayaç sıfırdan başlar ve liste baştan taranır. Üç dilimde A, B ve C birer kez çalıştı. Oysa beklenen paylar %30, %10 ve %60'tır; kısa vadede pay biletlerle orantılı değildir, ancak uzun vadede 30 : 10 : 60'a yaklaşır.

Bir çekilişin maliyeti en kötü durumda O(n)'dir, çünkü kazanan listenin sonunda olabilir. Liste bilet sayısına göre azalan sıralanırsa en çok biletli süreçler başta olur ve tarama ortalamada kısalır. Çok süreçli sistemlerde birikimli toplamlar bir ağaçta tutularak seçim O(log n)'ye indirilebilir.

5.5.4 Stride çizelgeleme

Tanım 5.18

Stride çizelgelemede (*stride scheduling*) her sürecin bir **adımı** vardır: adım = büyük bir sabit / bilet sayısı. Her sürecin ayrıca bir **pass** sayacı tutulur ve başlangıçta sıfırdır. Her dilimde pass değeri en küçük süreç çalışır ve pass değerine kendi adımını ekler.

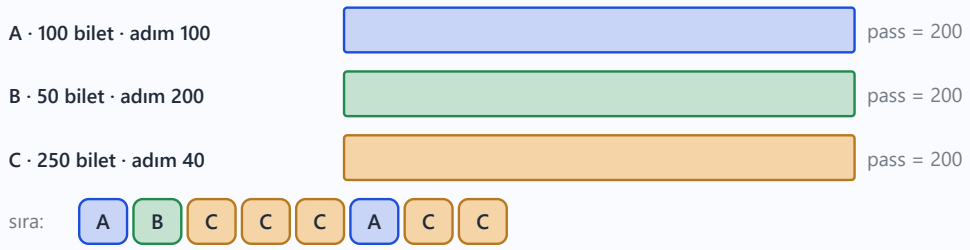
Çok bileti olan sürecin adımı küçüktür; pass değeri yavaş artar ve sık seçilir. Az bileti olanın adımı büyüktür; bir kez çalışınca uzun süre geride kalır. Waldspurger 1995'te bu yöntemi piyangonun deterministik eşi olarak önerdi (Tablo 5.17; Şekil 5.35).

Örnek 5.14

A'nın 100, B'nin 50, C'nin 250 bileti var; sabit 10.000. Adımlar A 10.000 / 100 = 100, B 200, C 40'tır. Eşitlikte A, B, C sırası uygulanır. İlk 8 dilimde çalışma sırası nedir?

Dilim	pass A	pass B	pass C	Çalışan	Güncelleme
1	0	0	0	A	pass(A) = 100
2	100	0	0	B	pass(B) = 200
3	100	200	0	C	pass(C) = 40
4	100	200	40	C	pass(C) = 80
5	100	200	80	C	pass(C) = 120
6	100	200	120	A	pass(A) = 200
7	200	200	120	C	pass(C) = 160
8	200	200	160	C	pass(C) = 200

Tablo 5.17: Stride izlemesi. pass değerleri dilim başındaki, yani seçim anındaki değerlerdir.



Şekil 5.35: Stride çizelgelemede 8 dilimin sonu. Sıra A, B, C, C, C, A, C, C; üç sürecin pass değeri de 200'e eşitlenmiştir.

8 dilimde A 2, B 1, C 5 kez çalıştı. Oran tam olarak 2 : 1 : 5'tir; bu da bilet oranı 100 : 50 : 250'nin kendisidir. Sekizinci dilimin sonunda üç pass değeri de 200'dür: bir tam döngü tamamlanmıştır ve aynı desen yeniden başlar. Piyangoda aynı oran ancak uzun vadede ve yaklaşık olarak tutar.

Stride'ın bir dezavantajı küresel durumdur. Sisteme yeni bir süreç girdiğinde pass değeri ne olmalıdır? Sıfır verilirse öteki süreçlerin pass değerleri yüzlerce olduğu için yeni süreç uzun süre CPU'yu tekeline alır. Bu yüzden yeni süreç o andaki en küçük pass değeriyle başlatılır. Piyangoda böyle bir sorun yoktur; yeni süreç yalnız biletlerini getirir ve hemen adil bir şans elde eder. Linux CFS'in sanal çalışma süresi fikri stride'a çok yakındır: `vruntime` bir pass sayacı, ağırlığın tersi de adımdır.

Örnek 5.15

A 50, B 100, C 25 bilet; adımlar A 200, B 100, C 400. İlk 7 dilimin sırası nedir?

1. Dilim 1: pass (0, 0, 0) → A; A 200. Dilim 2: (200, 0, 0) → B; B 100. Dilim 3: (200, 100, 0) → C; C 400.
2. Dilim 4: (200, 100, 400) → B; B 200. Dilim 5: (200, 200, 400) → eşitlikte A; A 400.
3. Dilim 6: (400, 200, 400) → B; B 300. Dilim 7: (400, 300, 400) → B; B 400.

Sıra A, B, C, B, A, B, B. A 2, B 4, C 1 dilim aldı: oran 2 : 4 : 1 = 50 : 100 : 25. Sonda üç pass değeri de 400'dür.

5.6 Çok işlemcili çizelgeleme

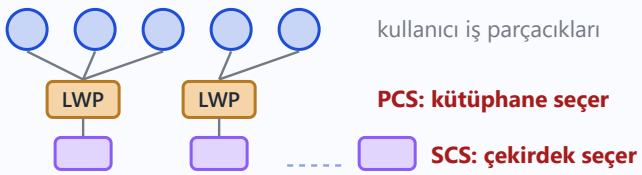
Bugün bir telefonda bile sekiz çekirdek bulunur. Birden çok CPU varken çizelgeleyici iki soruya cevap vermek zorundadır: hangi iş çalışsın ve hangi CPU'da çalışsın? İkinci soru, önbellekler ve bellek düzeni yüzünden göründüğünden zordur.

5.6.1 İş parçacığı çizelgeleme: PCS ve SCS

Çekirdek süreçleri değil, çekirdek iş parçacıklarını çizelgeler. Kullanıcı iş parçacıklarını ise bir kütüphane ya da dilin çalışma zamanı yönetir. Bu iki katmanlı yapı iki ayrı yarışma kapsamı doğurur.

Tanım 5.19

Süreç kapsamlı yarışmada (PCS, *process-contention scope*) iş parçacığı kütüphanesi aynı sürecin kullanıcı iş parçacıklarını süreç içindeki hafif süreçler (LWP) üzerinde yarıştırır. Bu kapsam N:1 ve M:N modellerinde görülür. **Sistem kapsamlı yarışmada** (SCS, *system-contention scope*) çekirdek, sistemdeki bütün iş parçacıkları arasından seçim yapar (Şekil 5.36).



Şekil 5.36: PCS ve SCS. Kullanıcı iş parçacıklarını kütüphane LWP'lere (PCS), çekirdek iş parçacıklarını çekirdek CPU'lara (SCS) eşler.

Pthreads kapsamı bir öznitelikle sorgulamaya ve ayarlamaya izin verir:

```
pthread_attr_t attr;
int scope;
pthread_attr_init(&attr);
pthread_attr_getscope(&attr, &scope);
if (scope == PTHREAD_SCOPE_PROCESS)
  printf("PCS\n");
/* Linux ve macOS yalnızca SYSTEM */
pthread_attr_setscope(&attr,
                      PTHREAD_SCOPE_SYSTEM);
pthread_create(&tid, &attr, runner, NULL);
```

Program bir öznitelik nesnesi oluşturur ve varsayılan kapsamı okur. Kapsam `PTHREAD_SCOPE_PROCESS` ise `PCS` basar. Ardından kapsamı sisteme ayarlar ve iş parçacığını bu öznitelikle oluşturur. Linux, Windows ve macOS 1:1 modelini kullandığı için yalnız SCS'yi destekler. Linux'ta varsayılan kapsam zaten `PTHREAD_SCOPE_SYSTEM`'dir ve program bir şey basmaz; PCS istemek `ENOTSUP` hatası verir.

PCS fikri işletim sistemlerinde terk edilse de dil çalışma zamanlarında geri döndü. Java 21'in sanal iş parçacıkları ve Go'nun goroutine'leri, çalışma zamanının kendi çizelgeleyicisiyle az sayıda taşıyıcı iş parçacığına eşlenir. Çekirdek yalnız taşıyıcıları görür; hangi sanal iş parçacığının hangi taşıyıcıda çalışacağına çalışma zamanı karar verir.

5.6.2 Ortak kuyruk ve çekirdek başına kuyruk

Çok işlemcili çizelgelemenin iki temel düzeni vardır. **Asimetrik çok işlemlemede** (AMP) tek bir ana işlemci bütün çizelgeleme kararlarını ve G/Ç işlemlerini yapar; ötekiler yalnız kullanıcı kodu çalıştırır. Basittir, çünkü çekirdek veri yapılarına tek işlemci erişir, ama ana işlemci darboğaz olur. **Simetrik çok işlemlemede** (SMP) her işlemci kendi çizelgeleme kararını verir. Bütün modern sistemler SMP'dir (Şekil 5.37).



Şekil 5.37: (a) Kilitle korunan tek ortak hazır kuyruğu: dört çekirdek de işini aynı kuyruktan alır. (b) Her çekirdeğin kendi hazır kuyruğu.

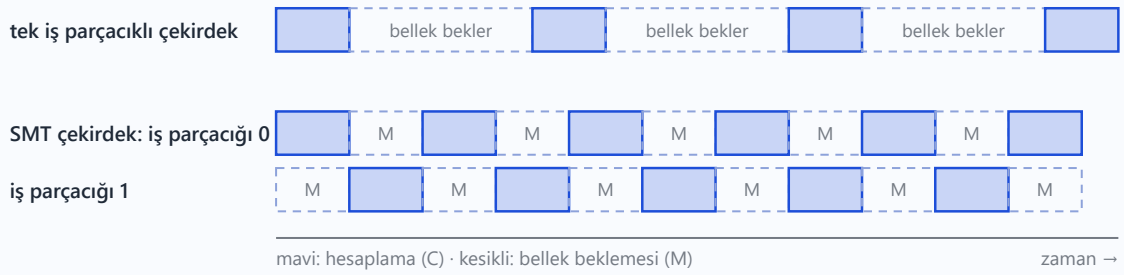
SMP'de hazır süreçleri tutmanın iki yolu vardır. **Ortak kuyrukta** bütün çekirdekler aynı kuyruktan iş alır. Yük kendiliğinden dengelenir, çünkü boşalan her çekirdek kuyruktaki sıradaki işi alır. Ama kuyruğa her erişim bir kilit ister ve çekirdek sayısı arttıkça kilit çekişmesi büyür. Ayrıca bir iş her seferinde

farklı bir çekirdekte çalışabilir; önbelleğindeki veri işe yaramaz.

Çekirdek başına kuyrukta her çekirdeğin kendi hazır kuyruğu vardır. Buna iki düzeyli çizelgeleme de denir: iş önce bir çekirdeğe atanır, sonra o çekirdeğin kuyruğundan çalışır. Kilit neredeyse gerekmez ve iş aynı çekirdekte kaldığı için önbelleği sıcak kalır. Linux, Windows ve FreeBSD bu düzeni kullanır. Bedeli, kuyrukların zamanla dengesizleşebilmesidir: bir çekirdekte beş iş beklerken öteki boşta kalabilir. Bu yüzden yük dengeleme gerekir.

5.6.3 Çok çekirdek ve donanım iş parçacıkları

Modern işlemcilerde bir çekirdek zamanının büyük kısmını bellek beklerken geçirebilir. Önbellek ıskası olduğunda veri ana bellekten gelene kadar yüzlerce çevrim geçer. Silberschatz bu **bellek beklemesinin** (*memory stall*) çekirdek zamanının %50'sine varabildiğini belirtir (Şekil 5.38).



Şekil 5.38: Bellek beklemesi. Tek iş parçacıklı çekirdek zamanının önemli bir kısmında bekler (C: hesaplama, M: bellek beklemesi). SMT'de iki donanım iş parçacığı birbirinin beklemelerini doldurur.

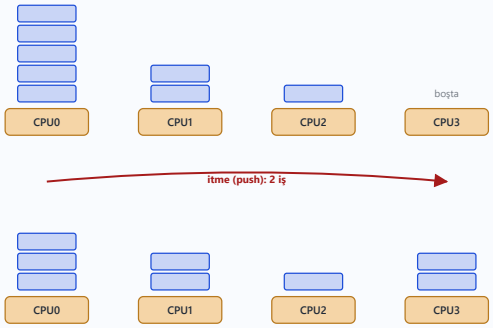
Eş zamanlı çoklu iş parçacığı (SMT; Intel'de Hyper-Threading) bu boşluğu doldurur. Bir çekirdekte iki donanım iş parçacığı bulunur; her birinin kendi yazmaç kümesi vardır, yürütme birimleri ise ortaktır. Biri bellek beklerken öteki komut yürütür. İşletim sistemi her donanım iş parçacığını ayrı bir **mantıksal CPU** olarak görür; dört çekirdekli ve SMT'li bir işlemci sekiz mantıksal CPU gösterir.

Böylece iki düzeyli bir çizelgeleme ortaya çıkar. İşletim sistemi hangi yazılım iş parçacığının hangi mantıksal CPU'da çalışacağını seçer. Çekirdeğin donanımı ise o an hangi donanım iş parçacığının yürüyeceğini seçer. İşletim sistemi bu ayrımı bilmelidir. Kardeş mantıksal CPU'lar yürütme birimlerini paylaşır. Bu yüzden boş bir fiziksel çekirdek varken iki yoğun işi aynı çekirdeğin iki mantıksal CPU'suna koymak kötü bir karardır; ikisi de yavaşlar.

5.6.4 Yük dengeleme

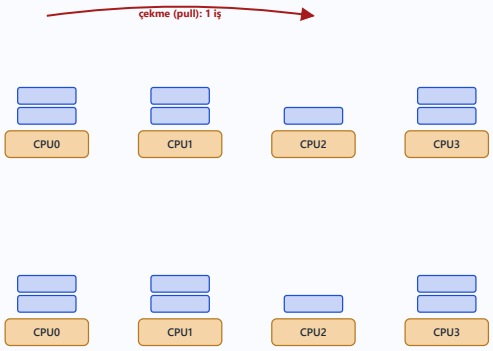
Tanım 5.20

Yük dengeleme (*load balancing*), işleri çekirdekler arasında eşit dağıtma işidir. **İtme göçünde** (*push migration*) periyodik bir denetleyici yükleri karşılaştırır ve aşırı yüklü çekirdekten işleri daha az yüklülere iter. **Çekme göçünde** (*pull migration*) boşta kalan çekirdek yüklü bir çekirdeğin kuyruğundan iş çeker (Şekil 5.39).



Şekil 5.39: Solda başlangıç: CPU0'ın kuyruğunda beş iş var, CPU3 boşta. Sağda itme göçü: periyodik dengeleyici CPU0'dan CPU3'e iki iş iter.

Çekirdek başına kuyrukta dengesizlik kendiliğinden düzelmez; birinin müdahale etmesi gerekir. İki yöntem birbirini tamamlar ve Linux ikisini de kullanır. Periyodik dengeleyici birkaç milisaniyede bir çalışıp büyük farkları giderir. Bir çekirdeğin kuyruğu boşaldığında ise beklemeden çekme göçü yapılır: boştaki çekirdek en yüklü kuyruktan bir iş çalar (Şekil 5.40).



Şekil 5.40: Solda çekme göçü: CPU2'deki tek iş bitince boşta kalan CPU2 en yüklü kuyruktan bir iş çalar. Sağda sonuç: kuyruklar dengelenmiştir.

Denge sağlansa da bedeli vardır. Taşınan işin önbelleği yeni çekirdekte soğuktur ve ilk erişimleri ıskayla başlar. Linux bu yüzden dengelemeyi katmanlı yapar. Bu katmanlara çizelgeleme alanları (*scheduling domains*) denir. Önce aynı fiziksel çekirdeğin SMT kardeşleri arasında, sonra ortak son düzey önbelleği paylaşan çekirdekler arasında dengeler. NUMA düğümleri arasındaki dengeleme daha seyrek ve daha büyük bir dengesizlik eşiğiyle yapılır, çünkü orada göçün bedeli en büyüktür. İşlerin çekirdeklere dağıtımı ve göçün etkisi çok çekirdekli çizelgeleme simülatöründe adım adım izlenebilir.

5.6.5 Kod izleme: `dengele()`

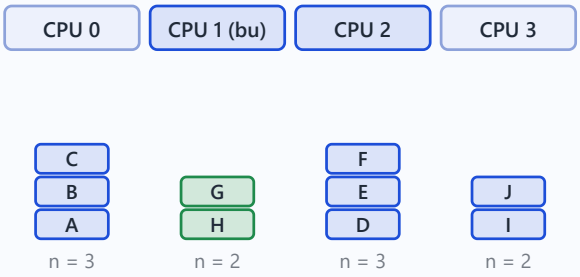
Çekme dengelemesinin özü kısa bir fonksiyondur. `q[c].n`, c numaralı çekirdeğin kuyruk uzunluğudur; `gocur(kaynak, hedef)` kaynak kuyruğun sonundaki işi hedef kuyruğa taşır.

```
void dengele(int bu) {   /* boştaki CPU */
  int en = bu;
  for (int c = 0; c < NCPU; c++)
    if (q[c].n > q[en].n) /* daha yüklü */
      en = c;
  int t = (q[en].n - q[bu].n) / 2;
  for (int k = 0; k < t; k++)
    gocur(&q[en], &q[bu]); /* sondan */
}
```

Fonksiyon, kuyruğu boşalan çekirdeğin numarasıyla (`bu`) çağrılır. İlk döngü en yüklü kuyruğu bulur: `en` başlangıçta `bu`'dur ve daha uzun bir kuyruk görüldükçe güncellenir. Ardından taşınacak iş sayısı farkın yarısı olarak hesaplanır. Farkın tamamı taşınsaydı dengesizlik yalnız yön değiştirirdi; yarısı taşınınca iki kuyruk birbirine yaklaşır. Son döngü o kadar işi kaynağın sonundan alır (Tablo 5.18; Şekil 5.41).

Adım	Olay	`en`	Kuyruk uzunlukları
başta	`en = bu = 1`	1	3, 0, 5, 2
c = 0	3 > 0, daha yüklü	0	3, 0, 5, 2
c = 1	0 > 3 değil	0	3, 0, 5, 2
c = 2	5 > 3, daha yüklü	2	3, 0, 5, 2
c = 3	2 > 5 değil	2	3, 0, 5, 2
`t`	(5 − 0) / 2 = 2 (tam sayı bölmesi)	2	3, 0, 5, 2
k = 0	H, CPU2'den CPU1'e göçer	2	3, 1, 4, 2
k = 1	G, CPU2'den CPU1'e göçer	2	3, 2, 3, 2

Tablo 5.18: `dengele(1)`; başlangıçta kuyruklar CPU0 3 (A, B, C), CPU1 0, CPU2 5 (D–H), CPU3 2 (I, J).



Şekil 5.41: `dengele(1)` sonrası kuyruklar: CPU1'e H ve G göçmüştür; uzunluklar 3, 2, 3, 2.

İşlerin kuyruğun sonundan alınmasının bir nedeni vardır. Sondakiler en uzun süre beklemiş olanlar değil, en son gelenlerdir; önbelleğe en az veri yüklemiş, yani önbellekleri en soğuk olan işlerdir. Bunları taşımak afinite kaybını en aza indirir. Fonksiyonun zamanı O(NCPU + t)'dir. Gerçek çekirdeklerde yalnız kuyruk uzunluğuna değil, işlerin ağırlıklarına ve son zamanlardaki CPU kullanımlarına da bakılır.

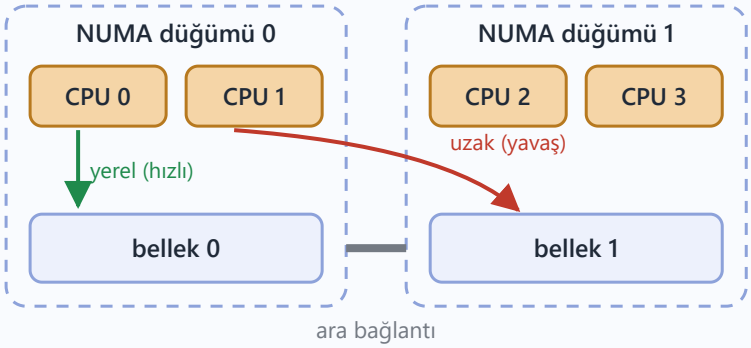
5.6.6 Afinite, NUMA ve heterojen çekirdekler

Tanım 5.21

İşlemci afinitesi (*processor affinity*), bir iş parçacığını önbelleği ve TLB'si sıcak olan CPU'da tutma eğilimidir. **Yumuşak afinitede** çizelgeleyici işi aynı CPU'da tutmaya çalışır ama garanti etmez. **Katı afinitede** iş parçacığının çalışabileceği CPU kümesi açıkça belirlenir.

Linux katı afiniteyi `sched_setaffinity()` sistem çağrısıyla sunar; kabukta `taskset -c 0-3 ./prog` komutu programı 0–3 numaralı CPU'larla sınırlar. Veritabanları ve gerçek zamanlı uygulamalar bunu sıkça kullanır.

Afinite sorusu NUMA (tekdüze olmayan bellek erişimi) mimarisinde daha da büyür. Çok soketli sunucularda her işlemcinin kendine yakın bir belleği vardır. Yerel belleğe erişim hızlı, öteki düğümün belleğine ara bağlantı üzerinden erişim yavaştır. İyi bir çizelgeleyici iş parçacığını belleğinin bulunduğu düğümde çalıştırır. Linux'un otomatik NUMA dengelemesi sayfa erişimlerini örnekler; görevleri verilerine ya da sayfaları görevlerine yaklaştırır (Şekil 5.42).



Şekil 5.42: NUMA: her düğümün CPU'ları kendi yerel belleğine hızlı, öteki düğümün belleğine ara bağlantı üzerinden yavaş erişir.

Yük dengeleme ile afinite birbiriyle çelişir. Göç dengeyi sağlar ama önbelleği ve bellek yerelliğini bozar. Afinite yerelliği korur ama çekirdeklerin bir kısmı boşta kalırken ötekiler aşırı yüklenebilir. Gerçek çizelgeleyiciler ikisi arasında eşiklerle denge kurar: dengesizlik küçükse işler yerinde kalır, büyükse taşınır.

Son yıllarda çekirdekler de türdeş olmaktan çıktı. **Heterojen çok işlemcili** yongalarda başarımı yüksek ama çok enerji harcayan çekirdeklerle yavaş ama verimli çekirdekler bir aradadır. ARM'ın big.LITTLE ve DynamIQ düzenleri, Intel'in 12. nesilden beri P ve E çekirdekleri, Apple M serisi buna örnektir. Intel'in Thread Director donanımı işletim sistemine hangi iş parçacığının hangi tür çekirdeğe uygun olduğu konusunda ipuçları verir. Linux'un enerji farkında çizelgelemesi (EAS, *Energy Aware Scheduling*) her işi, başarım hedefini karşılayan ve en az enerji harcayan çekirdeğe koyar. Arka plan işleri verimli çekirdeklere, kullanıcının beklediği işler güçlü çekirdeklere gider.

Sık yapılan hata

Göçün bedava olduğunu sanmak. Yük dengeleme işleri eşitler ama taşınan iş yeni çekirdekte önbelleği ve TLB'yi soğuk bulur; NUMA düğümleri arasında taşındıysa belleğine de uzaktan erişir. Boşta kalan bir çekirdeğin iş çekmesi (çekme göçü) de afiniteyi bozar; katı afinitede ise iş hiç taşınmaz.

5.7 Gerçek zamanlı çizelgeleme

Genel amaçlı bir sistemde işin geç bitmesi can sıkar ama yanlış değildir. Gerçek zamanlı bir sistemde ise doğru sonuç geç gelirse yanlış sayılır. Bir hava yastığı denetleyicisinin çarpışmayı 40 ms sonra fark etmesi hiç fark etmemesiyle aynıdır. Bu sistemlerde çizelgelemenin hedefi ortalamayı iyileştirmek değil, her işin son tarihini garanti etmektir.

5.7.1 Katı ve yumuşak gerçek zaman, gecikme

Tanım 5.22

Katı gerçek zamanlı (*hard real-time*) sistemde bir son tarihin kaçırılması sistemin başarısız olması demektir. **Yumuşak gerçek zamanlı** (*soft real-time*) sistemde kritik görevler önceliklidir, ama garanti yoktur; son tarihin kaçırılması yalnız hizmet kalitesini düşürür.

ABS fren sistemi, hava yastığı, endüstriyel robot denetimi ve uçuş denetimi katı gerçek zamanlıdır. Video akışı, internet üzerinden ses (VoIP), oyunlar ve ses çalma yumuşak gerçek zamanlıdır: bir çerçeve geç gelirse görüntü bir an takılır, ama sistem çalışmaya devam eder.

Gerçek zamanlı sistemin tepkisini ölçen büyüklük **olay gecikmesidir** (*event latency*): bir olayın gerçekleşmesinden ona hizmet eden sürecin çalışmaya başlamasına kadar geçen süre. Bu süre birkaç bileşenin toplamıdır (Şekil 5.43). **Kesme gecikmesi**, kesmenin gelmesinden kesme hizmet yordamının (ISR) başlamasına kadar geçen süredir. Ardından ISR çalışır. Son olarak **dağıtım gecikmesi** gelir: çizelgeleyicinin çalışan süreci durdurup gerçek zamanlı süreci başlatması.



Şekil 5.43: Olay gecikmesi: kesme gecikmesi, kesme hizmet yordamı (ISR) ve çakışma ile dağıtım evrelerinden oluşan dağıtım gecikmesinin toplamıdır.

Dağıtım gecikmesinin çakışma evresinde iki iş yapılır. Çekirdekte çalışan herhangi bir iş kesilir ve gerçek zamanlı sürecin ihtiyaç duyduğu, düşük öncelikli bir sürecin tuttuğu kaynak bırakılır. Bu evrenin kısa olması için çekirdeğin kesmeli olması gerekir; çekirdek kodu kesilemeseydi uzun bir sistem çağrısının bitmesi beklenirdi. Bu yüzden gerçek zamanlı sistemler kesmeli, öncelik tabanlı bir çekirdek ister. Yalnız bu da katı garanti için yetmez. Sistem yeni bir görev kabul etmeden önce, görev eklendiğinde bütün son tarihlerin hâlâ karşılanıp karşılanamayacağını sınamalıdır. Bu sınamaya **kabul denetimi** (*admission control*) denir.

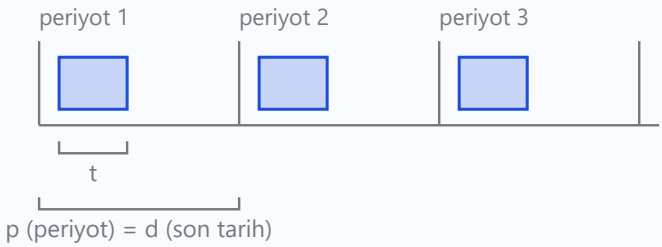
Gerçek zamanlı algoritmalar statik ya da dinamik olabilir. Statik algoritmalarda öncelikler sistem çalışmadan önce bir kez hesaplanır; hız monoton çizelgeleme böyledir. Dinamik algoritmalarda kararlar çalışma anında verilir; en erken son tarih önce böyledir.

5.7.2 Periyodik görevler ve hız monoton çizelgeleme

Gerçek zamanlı görevlerin çoğu periyodiktir. Bir sensör her 10 ms'de bir okunur, bir kontrol döngüsü her 50 ms'de bir yeni çıkış hesaplar.

Tanım 5.23

Periyodik görev her p sürede (periyot) bir, t süre CPU ister ve her işini d son tarihine kadar bitirmelidir; $0 \le t \le d \le p$. Görevin **kullanımı** $U = t / p$'dir. Bu notta son tarih periyodun sonudur: $d = p$ (Şekil 5.44).



Şekil 5.44: Periyodik görev: her p sürede bir t süre CPU ister ve son tarih d'ye (burada periyot sonuna) kadar bitmelidir.

Tanım 5.24

Hız monoton çizelgelemede (RM, *rate-monotonic scheduling*) her periyodik göreve periyoduyla ters orantılı, statik bir öncelik verilir: periyot kısaldıkça öncelik yükselir. Kesmelidir; daha öncelikli görevin yeni işi geldiğinde çalışan görev kesilir.

RM'nin arkasındaki sezgi basittir: daha sık çalışması gereken görev daha acildir. RM, statik öncelikli algoritmalar arasında optimaldir. Bir görev kümesi herhangi bir statik öncelik atamasıyla çizelgelenebiliyorsa RM ile de çizelgelenebilir. RM'nin çizelgeleyemediği kümeyi başka hiçbir statik öncelik ataması çizelgeleyemez.

Teorem 5.2 (Liu–Layland sınırı)

n bağımsız periyodik görevin son tarihleri periyot sonu olsun. Toplam kullanım

> $$\Sigma U \le n\,(2^{1/n} - 1)$$

koşulunu sağlıyorsa RM bütün son tarihleri karşılar.

Sınır n arttıkça azalır. Tablo 5.19 birkaç değeri verir; n sonsuza giderken sınır ln 2 ≈ 0,693'e yaklaşır. Yani görev sayısı ne olursa olsun, toplam kullanım %69,3'ü aşmıyorsa RM'nin başarısı garantidir.

n	1	2	3	4	5	10	∞
sınır	1	0,8284	0,7798	0,7568	0,7435	0,7177	ln 2 ≈ 0,6931

Tablo 5.19: Liu–Layland sınırı $n(2^{1/n} - 1)$.

Teoremin yönü önemlidir: koşul **yeterlidir, gerekli değildir**. Kullanım sınırın altındaysa RM kesin başarır. Sınırın üstündeyse RM yine başarabilir ya da başaramayabilir; testten sonuç çıkmaz. Karar vermek için çizelgeyi bir hiperperiyot boyunca (periyotların en küçük ortak katı kadar) çizmek gerekir. Toplam kullanım 1'i aşıyorsa hiçbir algoritma kümeyi çizelgeleyemez: CPU'dan saniyede bir saniyeden fazla iş isteniyordur.

Örnek 5.16

P1 (p = 50, t = 20) ve P2 (p = 100, t = 35). RM ile çizelgelenebilir mi?

1. **Test.** U = 20/50 + 35/100 = 0,40 + 0,35 = 0,75. n = 2 için sınır 0,828'dir; 0,75 ≤ 0,828 olduğu için RM bütün son tarihleri karşılar.
2. **Çizelge.** P1'in periyodu kısa, önceliği yüksektir. P1 0–20, P2 20–50 (30 ms). t = 50'de P1'in yeni işi gelir ve P2'yi keser: P1 50–70. P2 kalan 5 ms'sini 70–75'te bitirir; son tarihi 100'dür. 75–100 arasında CPU boştur.
3. **İkinci hiperperiyot.** 100–200 arasında aynı desen yinelenir. Hiçbir son tarih kaçırılmaz (Şekil 5.45).



Şekil 5.45: Hız monoton çizelgeleme: P1 (p = 50, t = 20) her 50 ms'de önce çalışır, P2 (p = 100, t = 35) kalan boşlukları kullanır; bütün son tarihler karşılanır.

5.7.3 RM ile EDF aynı kümede

Tanım 5.25

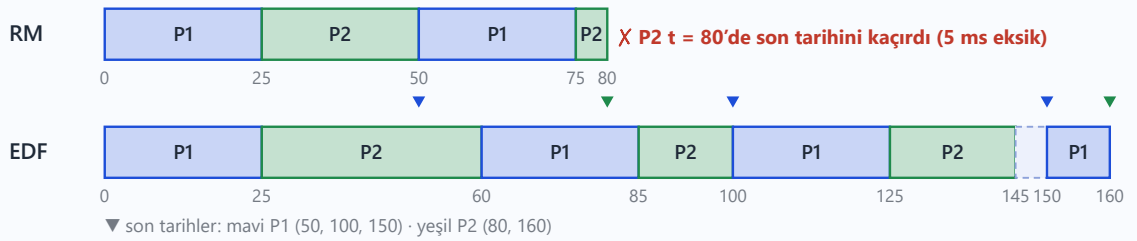
En erken son tarih önce (EDF, *earliest deadline first*) çizelgelemede öncelik dinamiktir: hazır işler arasında son tarihi en yakın olan çalışır. Bir işin önceliği her yeni periyotta, yeni son tarihine göre yeniden belirlenir.

EDF'nin çizelgelenebilirlik koşulu çok daha cömerttir: kesmeli, bağımsız ve son tarihi periyot sonu olan görevler için ΣU ≤ 1 yeterli ve gereklidir. EDF CPU'yu teorik olarak %100'e kadar kullanabilir. İki algoritmanın farkını, RM'nin testi geçemediği bir kümede görelim.

Örnek 5.17

P1 (p = 50, t = 25) ve P2 (p = 80, t = 35). U = 25/50 + 35/80 = 0,5 + 0,4375 = 0,9375 > 0,828: RM için garanti yoktur; ama U ≤ 1 olduğu için EDF yeterlidir.

1. **RM.** P1 (kısa periyot) önce 0–25, P2 25–50 çalışır; P2 25 ms tamamlamıştır, 10 ms'si kalmıştır. t = 50'de P1'in yeni işi gelir ve statik önceliği yüksek olduğu için P2'yi keser; P1 50–75. P2 75–80'de 5 ms daha çalışır ama t = 80'de süresi biter: P2 son tarihini 5 ms eksikle kaçırır.
2. **EDF, t = 50.** P1'in yeni işinin son tarihi 100, P2'ninki 80'dir. Son tarihi daha yakın olan P2 devam eder ve 60'ta biter. P1 60–85 çalışır.
3. **EDF, t = 80.** P2'nin yeni işi gelir; son tarihi 160'tır. P1'inki 100 olduğu için P1 85'e kadar sürer. P2 85–100 çalışır. t = 100'de P1'in yeni işinin son tarihi 150 < 160'tır ve P1 öne geçer: P1 100–125, P2 125–145'te biter. 145–150 arası boştur.
4. **Sonuç.** 400 ms'lik hiperperiyot (50 ile 80'in en küçük ortak katı) boyunca EDF hiçbir son tarihi kaçırmaz (Şekil 5.46).



Şekil 5.46: P1 (50, 25) ve P2 (80, 35). Üstte RM: P2 t = 80'de son tarihini 5 ms eksikle kaçırır. Altta EDF: P1 0–25, P2 25–60, P1 60–85, P2 85–100, P1 100–125, P2 125–145, P1 150–; bütün son tarihler karşılanır.

EDF'nin üstünlüğünün bedeli vardır. Öncelikler dinamik olduğu için her karar anında son tarihleri karşılaştırmak gerekir. Daha önemlisi aşırı yükte davranışıdır. Toplam kullanım geçici olarak 1'i aşarsa RM'de yalnız düşük öncelikli görevler kaçırır ve hangisinin kaçıracağı öngörülebilir. EDF'de ise kaçırmalar bir zincir gibi yayılabilir ve hangi görevin etkileneceğini önceden söylemek zordur.

EDF'ye yakın bir fikir, **orantılı pay** gerçek zamanlı çizelgelemedir. Toplam T pay uygulamalar arasında dağıtılır; N pay alan uygulama CPU'nun N/T'sini alır. Kabul denetimi toplamı aşan isteği reddeder. Linux'un `SCHED_DEADLINE` politikası (3.14 sürümünden beri) EDF'yi sabit bant genişliği sunucusu (CBS) korumasıyla birleştirir. Her görev bir çalışma süresi, son tarih ve periyot bildirir. Görev bildirdi-

ğinden fazla çalışmaya kalkarsa bir sonraki periyoda kadar durdurulur; böylece kötü davranan bir görev ötekilerin garantisini bozamaz. Periyodik görev kümeleri için RM ve EDF çizelgeleri, kullanım oranı ve kaçırılan son tarihler gerçek zamanlı çizelgeleme simülatöründe denenebilir.

Örnek 5.18

T1 (çalışma 2, periyot 5) ve T2 (çalışma 4, periyot 7). Toplam kullanım nedir? İlk son tarihler RM ve EDF ile karşılanır mı?

1. U = 2/5 + 4/7 = 34/35 ≈ 0,97. RM sınırı 0,828 aşılıyor; EDF için U ≤ 1 yeterli.
2. **RM.** T1 0–2, T2 2–5 (3 birim). t = 5'te T1'in yeni işi statik önceliğiyle T2'yi keser: T1 5–7. T2'nin son tarihi 7'dir ve 1 birimi eksik kalır; son tarih kaçırılır.
3. **EDF.** t = 5'te T1'in yeni son tarihi 10, T2'ninki 7'dir; T2 devam eder ve 6'da biter. 35 birimlik hiperperiyot boyunca EDF hiç son tarih kaçırmaz.

5.7.4 Kod izleme: Liu–Layland testi

RM'nin yeterlilik testi kısa bir fonksiyondur. Fonksiyon üç değer döndürür: 1 (RM kesin yeter), 0 (test karar veremez) ve −1 (kullanım 1'i aştığı için hiçbir algoritma çizelgeleyemez).

```
int rm_test(Gorev g[], int n) {
  double U = 0;
  for (int i = 0; i < n; i++)
    U += (double) g[i].t / g[i].p;
  double sinir = n * (pow(2, 1.0/n) - 1);
  if (U <= sinir) return 1; /* yeter */
  if (U <= 1.0) return 0;   /* belirsiz */
  return -1;            /* imkânsız */
}
```

Döngü her görevin kullanımını toplar; `(double)` dönüşümü tam sayı bölmesini engeller. Ardından sınır hesaplanır. `pow(2, 1.0/n)` ifadesindeki `1.0` önemlidir: `1/n` yazılsaydı n ≥ 2 için tam sayı bölmesi 0 verir, sınır da 0 çıkardı. Son iki karşılaştırma üç sonucu ayırır (Tablo 5.20).

Küme	Adım	Eklenen	`U`	`sinir`	Dönüş
T1 (20, 5), T2 (50, 10), T3 (100, 30)	i = 0	5 / 20 = 0,25	0,25		
	i = 1	10 / 50 = 0,2	0,45		
	i = 2	30 / 100 = 0,3	0,75		
	sınır		0,75	$3(2^{1/3} - 1) = 0{,}7798$	1
P1 (50, 25), P2 (80, 35)	i = 0	25 / 50 = 0,5	0,5		
	i = 1	35 / 80 = 0,4375	0,9375		

Küme	Adım	Eklenen	`U`	`sinir`	Dönüş
	sınır		0,9375	0,8284	0

Tablo 5.20: `rm_test()` iki görev kümesinde.

Birinci kümede 0,75 ≤ 0,7798 olduğu için fonksiyon 1 döndürür ve RM bütün son tarihleri karşılar. İkinci küme, Örnek 5.17'deki kümedir. Kullanım sınırı aşar ama 1'in altındadır; fonksiyon 0 döndürür. Bu kümede RM gerçekten son tarih kaçırır, EDF ise karşılar. 0 sonucu "RM başarısız olur" demek değildir: P1 (p = 50, t = 25) ve P2 (p = 100, t = 50) kümesinin kullanımı 0,5 + 0,5 = 1 > 0,828'dir ve test 0 verir, ama periyotlar katlı olduğu için RM bütün son tarihleri karşılar (P1 0–25, P2 25–50, P1 50–75, P2 75–100). Sınırı biraz aşan başka bir küme de RM ile çizelgelenebilir olabilir; bunu yalnız çizelgenin kendisi gösterir. Fonksiyonun zamanı O(n)'dir.

5.7.5 Kod izleme: EDF çizelgeleyici

EDF'nin karar kuralı her zaman adımında uygulanabilir. Aşağıdaki program üç periyodik görevi 12 ms boyunca, milisaniyede bir karar vererek çizelgeler.

```
/* A(P=3,C=1) B(P=4,C=1) C(P=6,C=2) */
for (int t = 0; t < 12; t++) {
    for (int i = 0; i < n; i++)
        if (t % P[i] == 0) {      /* yeni iş */
            kalan[i] = C[i];
            teslim[i] = t + P[i];
        }
    int s = -1;
    for (int i = 0; i < n; i++)
        if (kalan[i] > 0 &&
            (s < 0 || teslim[i] < teslim[s]))
            s = i;
    if (s >= 0) kalan[s]--;     /* 1 ms */
}
```

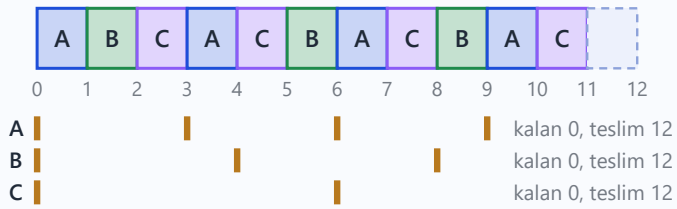
`P[i]` periyot, `C[i]` çalışma süresi, `kalan[i]` geçerli işin kalan süresi, `teslim[i]` de geçerli işin son tarihidir. Her milisaniyenin başında periyodu yeni başlayan görevlerin işleri yenilenir: kalan süre `C[i]`'ye, son tarih `t + P[i]`'ye ayarlanır. İkinci döngü işi kalmış görevler arasından son tarihi en küçük olanı seçer. Karşılaştırma katı küçüktür (`<`) olduğu için eşitlikte dizideki ilk görev kazanır. Seçilen görevin kalan süresi 1 azalır. Hiç iş yoksa `s` −1 kalır ve CPU o milisaniye boştur.

Görevler A (P = 3, C = 1), B (P = 4, C = 1) ve C (P = 6, C = 2)'dir. Toplam kullanım 1/3 + 1/4 + 2/6 = 11/12 ≈ 0,917 ≤ 1'dir. Periyotların en küçük ortak katı 12 olduğu için 12 ms'lik bir pencere bütün deseni gösterir (Tablo 5.21; Şekil 5.47).

`t`	Yeni iş	Hazır (son tarih)	Çalışan
0	A, B, C	A 3, B 4, C 6	A

`t`	Yeni iş	Hazır (son tarih)	Çalışan
1	—	B 4, C 6	B
2	—	C 6	C
3	A	A 6, C 6	A (eşitlikte ilk)
4	B	B 8, C 6	C
5	—	B 8	B
6	A, C	A 9, C 12	A
7	—	C 12	C
8	B	B 12, C 12	B (eşitlikte ilk)
9	A	A 12, C 12	A (eşitlikte ilk)
10	—	C 12	C
11	—	yok	boş

Tablo 5.21: EDF kod izlemesi. “Hazır” sütunu işi kalmış görevleri ve son tarihlerini gösterir.



Şekil 5.47: EDF Gantt şeması: A, B, C, A, C, B, A, C, B, A, C, boş. Her milisaniyede son tarihi en yakın hazır iş çalışır.

12 ms'de 11 ms iş yapılır ve 1 ms boşluk kalır; bu, 1 − 11/12 kadardır. Hiçbir iş son tarihini kaçırmaz. t = 4 anı EDF'yi RM'den ayırır: B'nin periyodu C'ninkinden kısadır ve RM olsaydı B çalışırdı. EDF ise son tarihlere bakar: C'nin işi 6'da, B'ninki 8'de bitmelidir; C seçilir. Bu küme RM ile de kaçırmasız çizelgelenir, ama kullanımı (0,917) üç görevli Liu–Layland sınırını (0,780) aştığı için RM'nin başarısını yalnız çizelge gösterebilir.

Programın zamanı adım başına O(n)'dir. Gerçek EDF uygulamaları her milisaniye karar vermez; yalnız iş geldiğinde ya da bittiğinde karar verir ve hazır işleri son tarihe göre sıralı bir öncelikli kuyrukta tutar. Böylece her karar O(log n) sürer.

5.7.6 POSIX gerçek zamanlı çizelgeleme

POSIX'in gerçek zamanlı uzantısı (POSIX.1b) iş parçacıklarına çizelgeleme politikası atamayı sağlar. Tablo 5.22 Linux'ta kullanılabilen politikaları özetler.

Politika	Davranış
`SCHED_FIFO`	Sabit öncelik (1–99), dilim yok; iş parçacığı bloke olana ya da CPU'yu bırakana kadar çalışır.
`SCHED_RR`	`SCHED_FIFO` ile aynı, ama eşit öncelikliler arasında zaman dilimi uygulanır.

Politika	Davranış
`SCHED_OTHER`	Varsayılan zaman paylaşımlı politika (Linux'ta EEVDF); ayrıca `SCHED_BATCH` ve `SCHED_IDLE`.
`SCHED_DEADLINE`	Yalnız Linux: çalışma süresi, son tarih ve periyot ile EDF.

Tablo 5.22: POSIX ve Linux çizelgeleme politikaları.

```
pthread_attr_t attr;
struct sched_param sp;
int policy;
pthread_attr_init(&attr);
pthread_attr_getschedpolicy(&attr, &policy);
/* kalıtım kapatılmazsa politika yok sayılır */
pthread_attr_setinheritsched(&attr,
                     PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
sp.sched_priority = 50;          /* 1..99 */
pthread_attr_setschedparam(&attr, &sp);
pthread_create(&tid, &attr, runner, NULL);
```

Program önce öznitelik nesnesini hazırlar ve varsayılan politikayı okur. Ardından önemli bir satır gelir: `pthread_attr_setinheritsched` ile kalıtım kapatılır. Varsayılan durumda yeni iş parçacığı politikasını oluşturan iş parçacığından devralır; kalıtım kapatılmazsa öznitelikte ayarlanan politika sessizce yok sayılır. Sonra politika `SCHED_FIFO`, öncelik 50 yapılır ve iş parçacığı oluşturulur.

Gerçek zamanlı politikalar ayrıcalık ister: Linux'ta `CAP_SYS_NICE` yetkisi gerekir; yetkisiz bir süreçte `pthread_create` `EPERM` döndürür. Kabukta aynı iş `chrt -f 50 ./prog` komutuyla yapılır. Gerçek zamanlı bir iş parçacığı sonsuz döngüye girerse sistemi kilitleyebilir. Linux buna karşı varsayılan olarak gerçek zamanlı işlere her saniyenin en fazla 0,95 saniyesini verir; kalan 0,05 saniye normal süreçlere kalır. Bu sınır `sched_rt_runtime_us` ayarıyla değiştirilebilir. İşlev adlarında bir ayrıntı da sık yanlış yazılır: adlar `pthread_attr_getschedpolicy` ve `pthread_attr_setschedpolicy`'dir; “sched” ile “policy” arasında alt çizgi yoktur.

5.8 İşletim sistemi örnekleri

Gerçek çizelgeleyiciler bölümdeki fikirlerin birleşimidir. Linux gerçek zamanlı işler için öncelikli kuyrukları, normal işler için orantılı payı kullanır. Windows ise MLFQ'ya benzeyen dinamik önceliklerle çalışır.

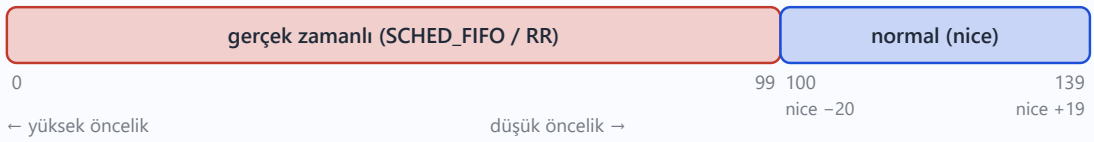
5.8.1 Linux çizelgeleme sınıfları

Linux çizelgeleyicisi modülerdir: her görev bir **çizelgeleme sınıfına** aittir ve sınıflar sabit bir öncelik sırasıyla sorulur. Çizelgeleyici önce en üst sınıfa “çalıştıracak görevin var mı?” diye sorar; yoksa bir alttakine geçer. Üst sınıfta hazır görev varken alttakiler hiç çalışmaz (Tablo 5.23).

Sınıf	Politikalar	Seçim
`stop`	— (çekirdek içi)	CPU durdurma, görev göçü; her şeyi keser
`deadline`	`SCHED_DEADLINE`	EDF + CBS
`rt`	`SCHED_FIFO`, `SCHED_RR`	100 öncelik düzeyli dizi ve biteşlem, O(1) seçim
`fair`	`SCHED_NORMAL`, `SCHED_BATCH`, `SCHED_IDLE`	CFS (2.6.23–6.5), EEVDF (6.6 ve sonrası)
`ext`	`SCHED_EXT`	BPF ile yazılan çizelgeleyici (sched_ext, 6.12 ve sonrası)
`idle`	—	hiç iş yoksa boşta döngüsü

Tablo 5.23: Linux çizelgeleme sınıfları (öncelik sırasıyla).

Bu sıra, aynı CPU'da hazır üç görevden hangisinin önce çalışacağını doğrudan belirler. `SCHED_DEADLINE` görevi, öncelik 50'li `SCHED_FIFO` görevinden önce gelir. O da nice değeri −20 olan normal görevden önce gelir. nice değeri yalnız adil sınıfın içindeki payı etkiler; bir normal görevi gerçek zamanlı görevin önüne geçiremez (Şekil 5.48).



Şekil 5.48: Linux çekirdek öncelikleri: 0–99 gerçek zamanlı görevler, 100–139 normal görevler (nice −20 ile +19). Küçük sayı yüksek önceliktir.

Çekirdek içinde tek bir öncelik ölçeği kullanılır ve küçük sayı yüksek önceliktir. 0–99 aralığı gerçek zamanlı görevlerindir; kullanıcının verdiği `rt_priority` (1–99) çekirdek önceliğine 99 − `rt_priority` olarak çevrilir. 100–139 aralığı normal görevlerindir: nice −20 → 100, nice 0 → 120, nice 19 → 139. `SCHED_RR` politikasının varsayılan dilimi 100 ms'dir.

5.8.2 CFS: sanal çalışma süresi

Linux 2.6.23'ten 6.5'e kadar normal görevleri Tamamen Adil Çizelgeleyici (*Completely Fair Scheduler*, CFS) çizelgeledi. CFS'in amacı ideal bir çok görevli işlemciyi taklit etmektir. İdeal işlemcide n görev varsa her biri aynı anda CPU'nun 1/n'i hızında çalışırdı. Gerçek CPU bir anda tek görev çalıştırabildiği için CFS her görevin bu idealden ne kadar geride kaldığını izler ve en çok geride kalanı çalıştırır.

Tanım 5.26

Sanal çalışma süresi (`vruntime`), bir görevin aldığı CPU süresinin ağırlığına göre ölçeklenmiş hâlidir. Görev Δ süre çalıştığında

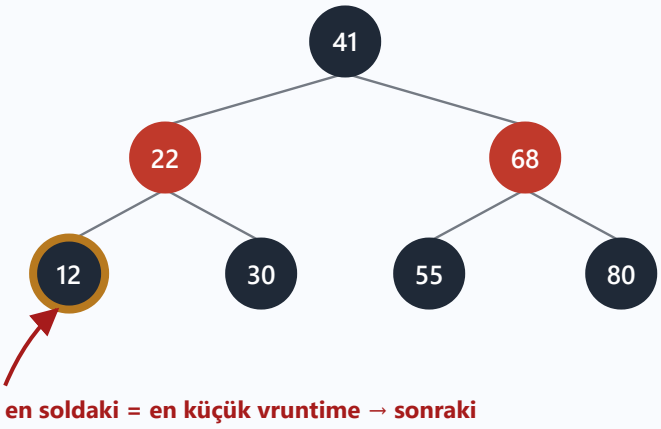
> vruntime += Δ × 1024 / ağırlık

olur. CFS her zaman `vruntime` değeri en küçük görevi çalıştırır.

Ağırlık nice değerinden gelir. nice 0'ın ağırlığı 1024'tür; her nice adımı ağırlığı yaklaşık 1,25 kat değiştirir. nice 5'in ağırlığı 335, nice −5'inki 3121'dir. Yaklaşık 1,25 katlık fark, iki CPU'ya bağlı görev arasında aşağı yukarı %10'luk CPU payı farkına karşılık gelir. nice 0 bir görev için vruntime gerçek süreyle aynı hızda artar. Ağırlığı küçük (nice'ı büyük) görevin vruntime'ı daha hızlı artar; bu yüzden daha seyrek seçilir.

İki CPU'ya bağlı görev düşünelim: A nice 0 (ağırlık 1024), B nice 5 (ağırlık 335). Uzun vadede ikisinin vruntime'ı aynı hızda ilerlemelidir; bu da paylarının ağırlıklarıyla orantılı olmasını gerektirir. A 1024 / (1024 + 335) = 1024 / 1359 ≈ %75, B ≈ %25 alır; oran yaklaşık 3 : 1'dir. B 10 ms çalışınca vruntime'ı 10 × 1024 / 335 ≈ 30,6 ms artar.

G/Ç'ye bağlı görevler bu düzende kendiliğinden ödüllenir. Uyuyan görev CPU kullanmadığı için vruntime'ı artmaz; uyandığında CPU'ya bağlı görevlerin gerisinde kalmıştır ve hemen öne geçer. Uzun süre uyuyan bir görevin geri döndüğünde CPU'yu uzun süre tekeline alması da istenmez. CFS bu yüzden uyanan görevin vruntime'ını en küçük vruntime'ın biraz gerisine çeker; geride kalmış göreve verilen avantaj sınırlıdır (Şekil 5.49).



Şekil 5.49: CFS hazır görevleri vruntime değerine göre kırmızı-siyah ağaçta tutar. En soldaki düğüm en az CPU almış görevdir ve sonraki seçimdir.

Hazır görevler vruntime anahtarıyla bir kırmızı-siyah ağaçta tutulur. Ekleme ve silme O(log n) sürer. En soldaki düğüm, yani vruntime'ı en küçük görev, ayrıca bir işaretçide önbelleğe alınır; bu yüzden seçim O(1)'dir.

5.8.3 Kod izleme: `vruntime`

Aşağıdaki kod CFS'in seçim kuralını basitleştirilmiş bir tık işleyicisi olarak gösterir. Her tıkta en soldaki görev 3 ms (3000 µs) çalıştırılır, vruntime'ı güncellenir ve ağaca yeniden eklenir.

```
#define DELTA 3000  /* 3 ms, µs olarak */
/* nice 0 → 1024, nice 5 → 335 */
void tik(void) {
  Gorev *g = en_soldaki(); /* en solda */
  calistir(g, DELTA);
  g->vruntime += DELTA * 1024 / g->agirlik;
  agaca_ekle(g);      /* yeniden sırala */
}
```

`en_soldaki()` vruntime'ı en küçük görevi döndürür; eşitlikte ağaca önce eklenmiş görev soldadır. `calistir` görevi 3 ms çalıştırır. Güncelleme satırı tam sayı aritmetiğiyle yapılır: A için 3000 × 1024 / 1024 = 3000, B için 3.072.000 / 335 = 9170 (kalan atılır). Son satır görevi yeni vruntime'ına göre ağaçtaki yerine koyar. Başlangıçta iki görevin vruntime'ı da 0'dır (Tablo 5.24).

Tık	A.vruntime	B.vruntime	Seçilen	Eklenen
1	0	0	A (eşitlikte önce eklenen)	3000
2	3.000	0	B	9170
3	3.000	9.170	A	3000
4	6.000	9.170	A	3000
5	9.000	9.170	A	3000
6	12.000	9.170	B	9170
7	12.000	18.340	A	3000
8	15.000	18.340	A	3000
9	18.000	18.340	A	3000
son	21.000	18.340		

Tablo 5.24: `tik()` izlemesi; A nice 0 (ağırlık 1024), B nice 5 (ağırlık 335). vruntime değerleri µs.

9 tıkta A 7, B 2 kez çalıştı. Kısa vadede oran 7 : 2'dir; uzun vadede 1024 : 335 ağırlık oranına, yani ≈ %75 – %25'e yaklaşır. Aynı kural 10.000 tık çalıştırıldığında A 7535, B 2465 tık alır. CFS'te sabit bir zaman dilimi yoktur; pay doğrudan ağırlıktan çıkar. Gerçek CFS süreyi nanosaniye ile ölçer ve sabit tık

yerine bir "zamanlayıcı gecikmesi" penceresini görevler arasında ağırlıklarına göre böler; ama seçim kuralı aynıdır. Bu kural stride çizelgelemenin pass sayacıyla aynı fikirdir: `vruntime` pass, 1024 / ağırlık adımdır.

5.8.4 EEVDF

CFS adil payı iyi sağlıyordu, ama gecikmeye duyarlı görevler için birçok sezgisel ayara ihtiyaç duyuyordu. Linux 6.6 sürümünde normal görevlerin çizelgeleyicisi EEVDF ile değiştirildi. EEVDF (*Earliest Eligible Virtual Deadline First*), Stoica ve Abdel-Wahab'ın 1995'te önerdiği bir algoritmadır. CFS'teki sezgilerin yerine iki net ölçüt koyar: uygunluk ve sanal son tarih.

Tanım 5.27

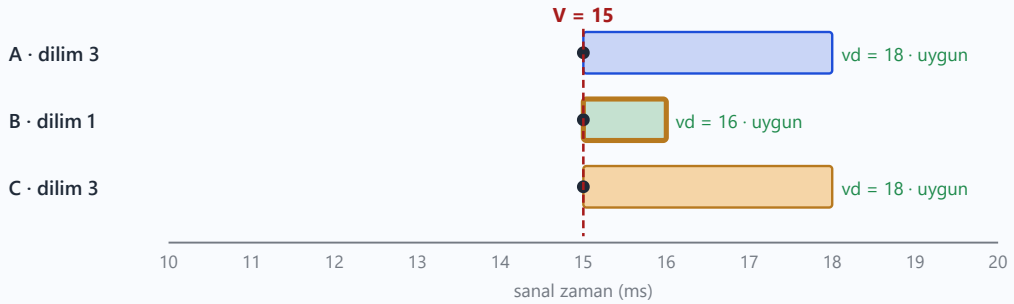
EEVDF'de V, görevlerin sanal çalışma sürelerinin (ağırlıklı) ortalamasıdır. Bir görevin **gecikmesi** (*lag*) payından ne kadar geride olduğunu gösterir. Görev payından geride ya da tam payındaysa **uygundur**: lag ≥ 0, yani v ≤ V. Görevin **sanal son tarihi** vd = v + dilim / ağırlık'tır. Uygun görevler arasında vd değeri en erken olan çalışır.

Uygunluk koşulu adaleti, son tarih ise gecikmeyi yönetir. Payından fazlasını almış görev (v > V) uygun değildir ve bekler; böylece kimse payını aşamaz. Uygun görevler arasında kısa dilim isteyen görevin son tarihi daha yakındır ve önce çalışır. Kısa dilim isteyen görev daha sık ama daha kısa çalışır; uzun vadede payı değişmez.

Örnek 5.19

Üç eşit ağırlıklı görev: A (v = 12, dilim 3), B (v = 12, dilim 1), C (v = 15, dilim 3). Ağırlık eşit olduğu için vd = v + dilim alınır. Beş seçimi izleyelim.

1. V = (12 + 12 + 15) / 3 = 13. C önde (v = 15 > V): uygun değil. Uygun A ve B; vd A 15, B 13 → B seçilir.
2. B 1 ms çalıştı: v = 13, yeni vd = 14. V = 13,33; uygun A ve B; en erken vd B'nin (14) → B.
3. B 14'e ilerledi; V = 13,67. B ve C ortalamanın önünde; yalnız A uygun → A 3 ms'lik dilimini çalışır.
4. A: v = 15, yeni vd = 18. V = 14,67; uygun yalnız B (v = 14) → B.
5. B: v = 15, vd = 16. V = 15; üçü de uygun. vd A 18, B 16, C 18 → B (Şekil 5.50).



Şekil 5.50: Beşinci seçim anı: V = 15 ve üç görev de uygundur; sanal son tarihi en erken olan B (vd = 16) seçilir.

Örnekte kısa dilim isteyen B beş seçimin dördünü aldı, ama her seferinde yalnız 1 ms çalıştı. A ise bir kez, 3 ms çalıştı. Uygunluk koşulu B'nin öne kaçmasını sınırlar: B ortalamanın önüne geçtiğinde (üçüncü adım) uygunluğunu kaybetmiş ve A'ya yol vermiştir. Linux 6.12'den beri gecikmeye duyarlı bir görev `sched_setattr()` çağrısıyla kendisi için kısa dilim isteyebilir. Ağırlıklar nice ile CFS'tekiyle aynı biçimde hesaplanır.

5.8.5 Windows öncelikleri

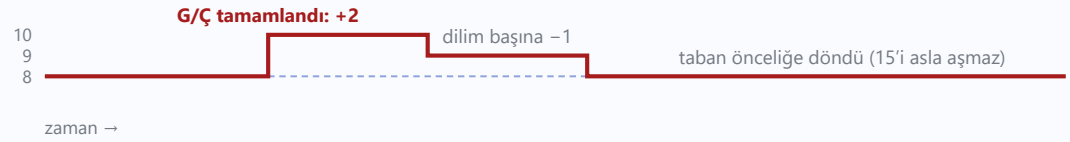
Windows kesmeli, öncelik tabanlı bir çizelgeleyici kullanır. Hazır iş parçacıklarından önceliği en yüksek olan çalışır. Bloke olana, dilimi bitene ya da daha yüksek öncelikli bir iş parçacığı hazır olana kadar CPU'da kalır. 32 öncelik düzeyi vardır ve büyük sayı yüksek önceliktir. 1–15 değişken sınıf, 16–31 gerçek zamanlı sınıftır. 0 düzeyi, boş bellek sayfalarını sıfırlayan sistem iş parçacığına (*zero-page thread*) ayrılmıştır. Her düzeyin kendi kuyruğu vardır; hiç hazır iş parçacığı yoksa boşta (*idle*) iş parçacığı çalışır.

Bir iş parçacığının sayısal önceliği iki bilgiden çıkar: sürecinin öncelik sınıfı ve iş parçacığının o sınıf içindeki göreli önceliği (Tablo 5.25).

Göreli öncelik	REALTIME	HIGH	ABOVE_NORMAL	NORMAL	BELOW_NORMAL	IDLE
zaman-kritik	31	15	15	15	15	15
en yüksek	26	15	12	10	8	6
normalin üstü	25	14	11	9	7	5
normal	**24**	**13**	**10**	**8**	**6**	**4**
normalin altı	23	12	9	7	5	3
en düşük	22	11	8	6	4	2
boşta	16	1	1	1	1	1

Tablo 5.25: Windows öncelik sınıfları (sütunlar) ve göreli öncelikler (satırlar). Kalın satır her sınıfın taban önceliğidir.

Değişken sınıftaki iş parçacıklarının önceliği zamanla değişir; bu, MLFQ'nun Windows'taki karşılığıdır. Bir G/Ç tamamlandığında iş parçacığının önceliği geçici olarak artırılır. Artış klavye ve fare gibi etkileşimli aygıtlarda disk G/Ç'sinden daha büyüktür. Ön plandaki pencerenin iş parçacıkları ve GUI olaylarıyla uyanan iş parçacıkları da artış alır. Artırılmış öncelik her dilimde 1 azalarak tabana döner ve hiçbir zaman 15'i aşmaz. Açlığa karşı da bir önlem vardır: yaklaşık 4 saniye boyunca hazır olduğu hâlde çalışamamış iş parçacığının önceliği bir dilimlik süre için 15'e çıkarılır. Dilim istemci sürümlerinde yaklaşık 2, sunucu sürümlerinde 12 saat tıkıdır; sunucuda uzun dilim bağlam değiştirmeyi azaltıp verimi artırır (Şekil 5.51).



Şekil 5.51: Windows dinamik öncelik artırma: taban önceliği 8 olan iş parçacığı G/Ç tamamlanınca 10'a çıkar, her dilimde 1 azalarak tabana döner.

Sık yapılan hata

Linux'ta nice −20 olan bir görevin `SCHED_FIFO` görevinin önüne geçeceğini sanmak. nice yalnız adil sınıfın içindeki ağırlığı değiştirir; gerçek zamanlı sınıfta hazır görev varken adil sınıftaki hiçbir görev çalışmaz. Windows'ta da dinamik artırmalar değişken sınıfla sınırlıdır: bir iş parçacığı artırmayla 15'i geçip gerçek zamanlı sınıfa giremez.

5.9 Algoritmaları değerlendirme

Bu kadar algoritma arasından belirli bir sistem için hangisi seçilmelidir? Önce ölçüt belirlenir: örneğin “en uzun tepki süresi 300 ms'yi aşmasın, bu koşulla verim en büyük olsun”. Sonra algoritmalar bu ölçüte göre değerlendirilir. Değerlendirmenin dört yolu vardır.

5.9.1 Dört yöntem ve Little yasası

Deterministik modelleme. Önceden verilmiş belirli bir iş yükü için her algoritmanın sonucu hesaplanır. Bu bölümdeki bütün Gantt şemaları deterministik modellemedir. Yöntem kesin ve basittir, sonuçlar karşılaştırılabilir sayılardır. Zayıflığı, sonucun yalnız o girdi için geçerli olmasıdır: başka bir kümede sıralama değişebilir.

Kuyruk modelleri. Gerçek sistemde iş yükü her gün değişir, ama süreçlerin geliş ve hizmet sürelerinin dağılımı ölçülebilir. Kuyruk kuramı bu dağılımlardan ortalama kuyruk uzunluğu, ortalama bekleme süresi ve kullanım gibi büyüklükleri hesaplar. En temel araç Little yasasıdır.

Teorem 5.3 (Little yasası)

Kararlı durumdaki bir sistemde ortalama süreç sayısı n, ortalama geliş hızı λ ile bir sürecin sistemde ortalama kalma süresi W'nin çarpımıdır:

$$n = \lambda \times W$$

Kararlı durumda sisteme giren süreç sayısı çıkan süreç sayısına eşittir. Saniyede λ süreç gelir ve her biri W saniye kalırsa, herhangi bir anda içeride ortalama λ × W süreç bulunur. Saniyede 7 süreç gelir ve her biri ortalama 2 saniye kalırsa sistemde ortalama 14 süreç vardır. Yasa geliş ve hizmet dağılımlarından bağımsızdır; bu yüzden çok geniş bir alanda kullanılır. Herhangi iki büyüklük biliniyorsa üçüncüsü bulunur. Bir sunucuya saniyede 6 istek geliyor ve sistemde ortalama 18 istek bulunuyorsa her istek ortalama W = 18 / 6 = 3 saniye kalır (Şekil 5.52).



Şekil 5.52: Little yasası: saniyede λ = 7 süreç gelir, her süreç sistemde ortalama W = 2 s kalır; sistemde ortalama n = 7 × 2 = 14 süreç bulunur.

Kuyruk modellerinin sınırı, matematiksel olarak çözülebilmeleri için yapılan varsayımlardır. Gelişlerin Poisson dağılımına uyduğu gibi varsayımlar gerçek iş yüklerinde tam tutmaz; karmaşık algoritmaların modeli de çoğu zaman çözülemez.

Simülasyon. Sistemin programlanmış bir modeli kurulur: saat, kuyruklar, çizelgeleyici. Model rastgele üretilmiş ya da gerçek bir sistemden kaydedilmiş iz (*trace*) verisiyle çalıştırılır. İz verisi gerçek olayların sırasını koruduğu için sonuçlar gerçeğe yakındır. Bedeli zamandır: ayrıntılı bir simülasyon saatler sürebilir ve kayıtlar büyük yer tutar.

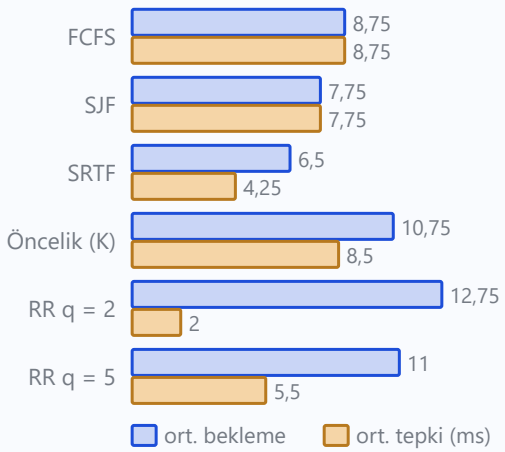
Gerçekleme. En kesin yol algoritmayı gerçek çekirdeğe koyup denemektir. Bu yol eskiden pahalıydı: çekirdeği değiştirmek, derlemek ve yeniden başlatmak gerekiyordu. Bugün Linux'un `sched_ext` altyapısı BPF ile yazılmış çizelgeleyicileri çekirdeği yeniden derlemeden yüklemeye izin verir; büyük şirketler farklı çizelgeleyicileri üretimde A/B testiyle karşılaştırır. Gerçeklemenin ince bir sorunu da vardır: kullanıcılar ve programcılar algoritmaya uyum sağlar. CTSS'te Enter tuşuna basan kullanıcılar bunun ilk örneğiydi; ölçülen davranış algoritma yüzünden değişebilir.

5.9.2 Ortak kümede karşılaştırma

Ortak kümede incelenen bütün algoritmaların sonuçları Tablo 5.26'da bir araya getirilmiştir (Şekil 5.53).

Algoritma	Bekleme	Dönüş	Tepki	Geçiş
FCFS	8,75	15,25	8,75	3
SJF	7,75	14,25	7,75	3
SRTF	6,5	13	4,25	4
Öncelik (K)	10,75	17,25	8,5	4
RR q = 2	12,75	19,25	2	12
RR q = 5	11	17,5	5,5	5

Tablo 5.26: Ortak kümede algoritmaların ortalamaları (ms) ve bağlam değiştirme sayıları. K: kesmeli.



Şekil 5.53: Ortak kümede algoritmaların ortalama bekleme ve ortalama tepki süreleri.

SRTF ortalama beklemede en iyisidir (6,5), ama sürelerin bilinmesini ister. RR q = 2 tepkide en iyisidir (2), ama en çok geçişi yapar ve beklemede en kötüsüdür. Kesmeli öncelik bekleme süresini değil önemi optimize eder; ortalaması FCFS'den kötü, ama en önemli P3 hiç beklemez. Tek bir “en iyi” algoritma yoktur. Seçim hedefe bağlıdır: tepki, verim, adalet ya da son tarih. Gerçek sistemlerin MLFQ ya da EEVDF gibi karma yapılar kullanmasının nedeni budur: kısa işleri öne alırken kimseyi aç bırakmamaya ve tepkiyi sınırlı tutmaya çalışırlar.

5.9.3 Dönüş, tepki ve adalet üzerine üç alıştırma

Örnek 5.20

A, B, C üçü de t = 0'da gelir ve her biri 5 ms CPU ister. SJF ile RR (q = 1) için ortalama dönüş ve ortalama tepki süresi nedir?

1. **SJF.** Süreler eşit olduğu için SJF burada FCFS ile aynıdır. Bitişler 5, 10, 15; ortalama dönüş 10 ms. Tepki (0 + 5 + 10) / 3 = 5 ms.

2. **RR (q = 1).** A, B, C her milisaniye yer değiştirir; son tur 12–15 arasındadır ve bitişler 13, 14, 15'tir. Ortalama dönüş 14 ms, tepki (0 + 1 + 2) / 3 = 1 ms.

Örnek 5.21

A (geliş 0, süre 100), B (10, 10), C (10, 10). Kesmesiz SJF ve STCF (SRTF) için ortalama dönüş ve tepki süresi nedir?

1. **SJF.** t = 0'da yalnız A vardır ve kesilemez: A 0–100, B 100–110, C 110–120. Dönüşler 100, 100, 110; ortalama 310 / 3 ≈ 103,3 ms. Tepkiler 0, 90, 100; ortalama ≈ 63,3 ms.
2. **STCF.** t = 10'da B gelir; süresi 10, A'nın kalanı 90'dır ve A kesilir. B 10–20, C 20–30, A 30–120. Dönüşler 120, 10, 20; ortalama 150 / 3 = 50 ms. Tepkiler 0, 0, 10; ortalama ≈ 3,3 ms.

Kesme, kısa işlerin uzun işin arkasında beklemesini, yani konvoy etkisini ortadan kaldırdı.

Örnek 5.22

Adaletsizlik ölçütü U, ilk biten işin bitiş zamanının ikinci biten işin bitiş zamanına oranı olsun. Eşit biletli iki iş, her biri L dilim istiyor. RR ile U kaçtır? Piyango çizelgelemede L büyüdükçe U nasıl değişir?

1. **RR (ve stride).** İşler dönüşümlü çalışır; biri 2L − 1'de, öteki 2L'de biter. U = (2L − 1) / 2L; L = 10 için 19 / 20 = 0,95. İş uzadıkça U 1'e yaklaşır.
2. **Piyango.** Her dilimde iki iş eşit olasılıkla kazanır; biri L kazanca ulaşınca biter, öteki kalan dilimlerini tek başına çalışır ve 2L'de biter. İlk işin bitiş anının beklenen değeri hesaplanırsa beklenen U, L = 10 için ≈ 0,82, L = 100 için ≈ 0,94 çıkar.

Piyango kısa işlerde belirgin bir adaletsizlik yaratır; iş uzadıkça şansın etkisi azalır ve U 1'e yaklaşır. Deterministik stride bu sorunu taşımaz.

5.10 Özet

- **Kısa vadeli çizelgeleyici** CPU boşalınca hazır kuyruğundan bir süreç seçer; **görev dağıtıcı** bağlamı değiştirir, kullanıcı moduna geçer ve programa atlar. Aradaki süre dağıtım gecikmesidir.
- Süreçler **CPU ve G/Ç patlamaları** arasında gidip gelir. Patlamaların çoğu kısadır; G/Ç'ye bağlı süreçleri bekletmemek aygıtları meşgul tutar. Her patlamayı ayrı iş saymak G/Ç ile hesaplamayı örtüştürür.
- Karar dört durumda alınır: çalışıyor → bekliyor, çalışıyor → hazır, bekliyor → hazır ve sonlanma. **Kesmesiz** çizelgeleyici yalnız 1. ve 4. durumda, **kesmeli** olan dördünde de karar verir.

- Ölçütler: CPU kullanımı ve verim büyük; dönüş (B − G), bekleme (dönüş − CPU) ve tepki (İ − G) süreleri küçük olmalıdır. Çizelgeleyici yalnız bekleme süresini etkiler.
- **FCFS** basit ve açlıksızdır, ama geliş sırasına duyarlıdır ve konvoy etkisi yaratır. **SJF**, işlerin hepsi aynı anda hazırken ortalama beklemeyi en aza indirir; sonraki patlama **üstel ortalamayla** tahmin edilir: $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n$. **SRTF** kesmeli sürümdür ve ortak kümede en düşük beklemeyi (6,5 ms) verir.
- **Öncelik** çizelgelemede açlık **yaşlandırmayla**, öncelik terslemesi öncelik kalıtımıyla çözülür.
- **RR** her sürece en fazla q süre verir; hiçbir süreç $(n - 1) \cdot q$'dan uzun beklemez. q küçüldükçe tepki iyileşir, geçiş artar; kayıp oranı $s / (q + s)$'dir. q en uzun patlamayı aşınca RR FCFS olur.
- **MLQ** süreçleri kalıcı sınıflara ayırır. **MLFQ** davranışa göre taşır: yeni iş üstte başlar, düzeydeki hakkını bitiren iner, belirli aralıklarla herkes en üste çıkar.
- Orantılı pay: **garantili** ve **adil paylaşım** (Linux cgroup `cpu.weight`), olasılıksal **piyango** ve deterministik **stride** (adım = sabit / bilet, en küçük pass çalışır).
- Çok işlemcide modern sistemler SMP ve **çekirdek başına kuyruk** kullanır; dengesizlik itme ve çekme göçüyle giderilir, göç ise **afiniteyi** bozar. SMT çekirdekleri, NUMA ve heterojen çekirdekler kararı karmaşıklaştırır.
- Gerçek zamanda **RM** statik önceliklidir; $\Sigma U \le n(2^{1/n} - 1)$ yeterli koşuldur. **EDF** dinamiktir; $\Sigma U \le 1$ yeterli ve gereklidir. Linux `SCHED_DEADLINE` EDF ile CBS'yi birleştirir.
- Linux sınıfları sırayla sorar: stop, deadline, rt, fair, ext, idle. **CFS** en küçük `vruntime`'ı seçer (vruntime += Δ × 1024 / ağırlık); **EEVDF** uygun görevler arasında en erken sanal son tarihi seçer. Windows 32 düzeyli öncelik ve dinamik artırma kullanır.
- Değerlendirme yöntemleri: deterministik modelleme, kuyruk modelleri (**Little yasası** $n = \lambda \times W$), simülasyon ve gerçekleme.

5.11 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

çizelgeleyici *scheduler* · hazır kuyruğu *ready queue* · CPU patlaması *CPU burst* · kesmeli / kesmesiz çizelgeleme *preemptive / nonpreemptive scheduling* · görev dağıtıcı *dispatcher* · bağlam değiştirme *context switch* · verim *throughput* · dönüş süresi *turnaround time* · tepki süresi *response time* · konvoy etkisi *convoy effect* · en kısa iş önce *shortest-job-first* · üstel ortalama *exponential average* · açlık *starvation* · yaşlandırma *aging* · öncelik terslemesi *priority inversion* · zaman dilimi *time slice (quantum)* · çok düzeyli geri beslemeli kuyruk *multilevel feedback queue* · piyango çizelgeleme *lottery scheduling* · yük dengeleme *load balancing* · işlemci afinitesi *processor affinity* · hız monoton çizelgeleme *rate-monotonic scheduling* · en erken son tarih önce *earliest deadline first*

5.12 Kendini Dene

1. P1 (geliş 0, CPU 7), P2 (1, 3), P3 (3, 2), P4 (6, 1). Kesmesiz SJF ve SRTF ile ortalama bekleme süreleri nedir?

   Cevap

   SJF: P1 0–7 kesilmez; t = 7'de hazır P2 (3), P3 (2), P4 (1); sıra P4 7–8, P3 8–10, P2 10–13. Beklemeler 0, 9, 5, 1; ortalama 15 / 4 = 3,75 ms. SRTF: t = 1'de P2 (3) < P1'in kalanı 6, P1 kesilir; P2 1–4, P3 4–6, P4 6–7, P1 7–13. Beklemeler 6, 0, 1, 0; ortalama 7 / 4 = 1,75 ms.

2. A (geliş 0, CPU 5), B (1, 4), C (2, 3); RR, q = 2. Yeni gelen süreç, dilimi biten süreçten önce kuyruğa girer. Gantt şeması ve ortalama bekleme nedir?

   Cevap

   A 0–2, B 2–4, C 4–6, A 6–8, B 8–10 (biter), C 10–11 (biter), A 11–12 (biter). Bitişler A 12, B 10, C 11. Beklemeler A 12 − 0 − 5 = 7, B 10 − 1 − 4 = 5, C 11 − 2 − 3 = 6; ortalama 6 ms. Ortalama tepki (0 + 1 + 2) / 3 = 1 ms.

3. $\alpha = 0{,}25$ ve $\tau_0 = 10$ ms. Ölçülen patlamalar 4, 4 ve 16 ms. τ_3 nedir? $\alpha = 0{,}25$ seçmenin bu dizideki etkisi nedir?

   Cevap

   $\tau_1 = 0{,}25 \cdot 4 + 0{,}75 \cdot 10 = 8{,}5$; $\tau_2 = 0{,}25 \cdot 4 + 0{,}75 \cdot 8{,}5 = 7{,}375$; $\tau_3 = 0{,}25 \cdot 16 + 0{,}75 \cdot 7{,}375 \approx 9{,}53$ ms. Küçük α geçmişe çok ağırlık verir: iki kısa patlamadan sonra tahmin hâlâ 7'nin üstündedir ve 16'ya sıçrayan patlamaya da yavaş yetişir.

4. MLFQ: Q0 (q = 3), Q1 (q = 6), Q2 (FCFS). A (geliş 0, CPU 12), B (2, 2), C (5, 6). İşler ne zaman biter?

   Cevap

   A 0–3 dilimini doldurur ve Q1'e iner. B 3–5'te Q0'da biter. t = 5'te gelen C 5–8 Q0 dilimini doldurur ve Q1'e iner. A Q1'de 8–14 çalışır, dilimini doldurup Q2'ye iner (kalan 3). C Q1'de 14–17'de biter. A Q2'de 17–20'de biter. Bitişler B 5, C 17, A 20.

5. Bir sistemde kesmesiz çizelgeleyici kullanılıyor. Çalışan süreç bir dosya okuması başlatıyor ve bu sırada daha önce başlatılmış bir disk okuması bitiyor. Bu iki olaydan hangisinde çizelgeleyici çağrı-

lır?

Cevap

Yalnız ilkinde. Çalışan süreç G/Ç isteyince bekleme durumuna geçer ve CPU boşalır (çalışıyor → bekliyor); seçim zorunludur. Disk okumasının bitmesi bir süreci bekliyor → hazır durumuna geçirir; kesmesiz çizelgeleyici çalışan süreci durdurmaz, uyanan süreç hazır kuyruğuna girer.

6. Stride çizelgelemede A'nın 40, B'nin 20, C'nin 80 bileti var; adım = 10.000 / bilet. Eşitlikte A, B, C sırası. İlk 7 dilimde çalışma sırası nedir?

Cevap

Adımlar A 250, B 500, C 125. Sıra A, B, C, C, A, C, C. A 2, B 1, C 4 dilim alır; oran 2 : 1 : 4 = 40 : 20 : 80. Yedinci dilimin sonunda üç pass değeri de 500'dür.

7. Üç periyodik görev: T1 (p = 4, t = 1), T2 (p = 5, t = 2), T3 (p = 10, t = 3). Liu–Layland testi ne söyler? RM gerçekten son tarih kaçırır mı?

Cevap

U = 0,25 + 0,4 + 0,3 = 0,95. Üç görev için sınır ≈ 0,780; test karar veremez (sonuç 0). 20 ms'lik hiperperiyot boyunca RM çizelgesi çizilirse (T1 0–1, T2 1–3, T3 3–4, T1 4–5, T2 5–7, T3 7–8, T1 8–9, T3 9–10, …) hiçbir son tarih kaçırılmaz. Test yeterli koşuldur, gerekli değil.

8. Linux CFS'te CPU'ya bağlı iki görev var: A nice −5 (ağırlık 3121), B nice 0 (ağırlık 1024). Payları yaklaşık nedir? Hangisinin vruntime'ı daha hızlı artar?

Cevap

A 3121 / 4145 ≈ %75, B ≈ %25. vruntime += Δ × 1024 / ağırlık olduğundan B'nin vruntime'ı aynı çalışma süresinde yaklaşık 3 kat hızlı artar; B daha seyrek seçilir.

9. Çekirdek başına kuyruklu bir sistemde neden ortak tek kuyruk kullanılmaz? Çekirdek başına kuyruğun bedeli nedir?

Cevap

Ortak kuyrukta her erişim bir kilit ister; çekirdek sayısı arttıkça kilit çekişmesi büyür ve iş her seferinde farklı çekirdekte çalışabildiği için önbelleği soğuk kalır. Çekirdek başına kuyrukta kilit gerekmez, afinite korunur; ama kuyruklar dengesizleşebilir ve yük dengeleme (itme ve çekme göçü) gerekir. Göç eden iş yeni çekirdekte önbelleği soğuk bulur.

10. RR'de q = 9 ms ve her bağlam değiştirme 1 ms. Süreçler dilimlerini sonuna kadar kullanıyor. CPU zamanının yüzde kaçı geçişe gider? q = 4'e göre ne değişir?

    Cevap

    1 / (9 + 1) = %10. q = 4'te kayıp 1 / 5 = %20 idi. Dilim büyüdükçe kayıp azalır, ama kuyruktaki her süreç bir sonraki dilimini daha uzun bekler ve tepki süresi kötüleşir.

5.13 İleri okuma

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, "Scheduling: Introduction", "The Multi-Level Feedback Queue", "Lottery Scheduling" ve "Multiprocessor Scheduling" bölümleri: MLFQ kurallarını sorunlardan yola çıkarak kurar, piyango ve stride çizelgelemeyi kısa C kodlarıyla anlatır; ostep.org'da ücretsizdir.
- C. L. Liu ve J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, 20(1), 1973: RM'nin optimalliğini ve kullanım sınırını, EDF'nin $\Sigma U \le 1$ koşulunu kanıtlayan temel makaledir.
- Linux çekirdek belgeleri, `Documentation/scheduler/sched-eevdf.rst` ve `sched-deadline.rst`: EEVDF'in uygunluk ve sanal son tarih kurallarını ve `SCHED_DEADLINE`'ın kabul denetimini çekirdeğin kendi belgesinde anlatır.
- P. Yosifovich ve diğerleri, *Windows Internals*, 7. baskı, Kısım 1, Bölüm 4: Windows'un öncelik düzeylerini, dilim hesabını, dinamik artırmaları ve çok işlemcili iş parçacığı seçimini ayrıntılı verir.

5.14 Kaynaklar

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 5: CPU Scheduling.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Kesim 2.4: Scheduling.
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — Bölüm 7–10: Scheduling, MLFQ, Lottery, Multiprocessor.
4. P. Yosifovich, A. Ionescu, M. Russinovich, D. Solomon, *Windows Internals*, 7. baskı, Kısım 1, 2017 — Bölüm 4: Thread scheduling.
5. Linux çekirdek belgeleri: `sched-eevdf.rst`, `sched-design-CFS.rst`, `sched-deadline.rst`, `sched-ext.rst`; `sched(7)` kılavuz sayfası.
6. I. Stoica, H. Abdel-Wahab, "Earliest Eligible Virtual Deadline First", 1995.
7. C. L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, 20(1), 1973, s. 46–61.

8. C. A. Waldspurger, W. E. Weihl, “Lottery Scheduling: Flexible Proportional-Share Resource Management”, *OSDI*, 1994.

KISIM II · BÖLÜM 6

Senkronizasyon

Aynı adres alanını paylaşan iş parçacıkları ya da ortak bir bellek bölgesini kullanan süreçler, birbirinden habersiz olarak aynı veriyi okuyup yazabilir. Çizelgeleyici bir iş parçacığını herhangi bir komutun ardından kesebilir; çok çekirdekli bir makinede iki iş parçacığı gerçekten aynı anda çalışır. Bu yüzden tek başına doğru olan iki kod parçası birlikte çalışınca yanlış sonuç verebilir. Senkronizasyon, paylaşılan veriye erişimi düzenleyerek bu hataları önleme işidir. Not, sorunun kendisiyle başlar: yarış durumu ve kritik bölge. Ardından yalnız okuma ve yazmayla kurulan yazılım çözümlerini, modern işlemcilerin bellek modelini ve donanımın sunduğu atomik komutları inceler. Bu temel üzerine spinlock, mutex, semafor ve monitör kurulur. Klasik problemler bu araçları sınar. Son kesimler POSIX ve Java arayüzlerini, öncelik terslemesini ve kilitsiz veri yapılarının temelini ele alır.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum06` · **Kod (Java):** `github.com/sercankulcu/operating-systems-java/Bolum06/src` · **Laboratuvar:** `github.com/sercankulcu/operating-systems-labs/lab06`

ÖĞRENME ÇIKTILARI

Problem

Yarış durumunu ve kritik bölgeyi tanımlamak; bir çözümün sağlaması gereken üç koşulu (karşılıklı dışlama, ilerleme, sınırlı bekleme) açıklamak.

Yazılım ve bellek modeli

Kilit değişkeni, katı sıra ve Peterson çözümünü bu üç koşula göre değerlendirmek; modern işlemcilerde yeniden sıralamanın etkisini ve bellek bariyerlerini açıklamak.

Donanım desteği

`test_and_set`, `compare_and_swap`, XCHG ve atomik değişkenlerle kilit ve sayaç kurmak; bir ara yerleştirmeyi adım adım izlemek.

Araçlar

Spinlock, mutex, semafor, monitör ve koşul değişkenini karşılaştırmak; Mesa anlamındaki `while` kuralını uygulamak.

Klasik problemler
Sınırlı tampon, okur-yazar ve yemek yiyen filozoflar problemlerini çözmek; açlık ve kilitlenme risklerini görmek.

Gerçek sistemler
POSIX ve Java arayüzlerini kullanmak; öncelik terslemesini (Mars Pathfinder) ve kilitsiz yapıların temelini yorumlamak.

6.1 Yarış durumu

Süreçler ve iş parçacıkları tek başlarına çalıştıklarında davranışları kolay öngörülür: aynı girdiyle aynı sonucu üretirler. Ortak bir veriyi paylaştıklarında bu kolaylık kaybolur. Sonuç artık yalnız koda değil, iki akışın komutlarının hangi sırayla iç içe geçtiğine de bağlıdır. Bu kesim sorunun nereden çıktığını üç örnekle gösterir: çekirdekteki bir sayaç, bir yazıcı kuyruğu ve tek satırlık bir `count++`.

6.1.1 Neden senkronizasyon?

Tanım 6.1

Senkronizasyon (*synchronization*), eş zamanlı yürüyen süreçlerin ya da iş parçacıklarının paylaşılan verilere erişimini düzenleyerek verinin tutarlılığını korumaktır.

Bir süreç, komutlarının herhangi birinin ardından kesilebilir. Zamanlayıcı kesmesi gelir, çizelgeleyici başka bir süreci seçer ve kesilen süreç yaptığı güncellemeyi yarıda bırakır. Yarım kalmış güncellemeyi bu sırada başka bir süreç okursa tutarsız bir değer görür. Çok çekirdekli sistemlerde kesme bile gerekmez: iki çekirdek gerçekten aynı anda aynı bellek sözcüğüne yazabilir. Şekil 6.1 bu durumu gösterir. İki iş parçacığı aynı `count` değişkeni üzerinde “oku, değiştir, yaz” adımlarını yürütür; araya giren bir bağlam değiştirme güncellemeyi yarıda bırakır.



Şekil 6.1: İki iş parçacığı aynı paylaşılan değişkeni okuyup yazar. Araya giren bir kesme ya da bağlam değiştirme, güncellemeyi yarıda bırakabilir.

Senkronizasyon hatalarının belirtileri çeşitlidir:

- **Tutarsız veri ve kayıp güncelleme.** İki artırmadan biri kaybolur; bağlı listenin bir düğümü listeden kopar.
- **Ezilen ortak tampon.** Ortak bir statik tampon kullanan, yani yeniden girilemeyen (*non-reentrant*) bir fonksiyon iki iş parçacığından çağrılınca biri ötekinin sonucunu ezer.
- **Kilitlenme.** İki iş parçacığı birbirinin bıraktığını beklerken sonsuza dek bekler.
- **Açlık.** Bir iş parçacığına hiç sıra gelmez; ötekiler hep önüne geçer.

Bu hataların ortak özelliği *ara sıra* ortaya çıkmalarıdır. Program yüz kez doğru çalışır, yüz birincide yanlış sonuç verir. Hata ayıklayıcıyla adım adım yürütmek zamanlamayı değiştirdiği için hata çoğu zaman kaybolur. Bu yüzden senkronizasyon hataları test ederek değil, kodu baştan doğru kurarak önlenir.

6.1.2 Çekirdekte bir yarış: fork() ve pid

Yarışlar yalnız kullanıcı programlarında görülmez; çekirdeğin kendisi de paylaşılan verilerle doludur. Silberschatz'ın örneğinde çekirdek, sıradaki boş süreç kimliğini `next_available_pid` adlı bir değişkende tutar. Değer 2615'tir. P0 ve P1 süreçleri neredeyse aynı anda `fork()` çağırır. Her çağrıda çekirdek kodu üç iş yapar: değeri okur, çocuğa verir ve bir artırıp geri yazar.

1. **P0 okur.** P0'ın çekirdek kodu `next_available_pid` değerini okur: 2615. Artırıp yazmadan önce kesilir.
2. **P1 okur ve yazar.** P1 de 2615 okur, çocuğuna 2615 verir ve değişkene 2616 yazar.
3. **P0 sürer.** P0 kaldığı yerden devam eder. Elindeki eski değerle çocuğuna yine 2615 verir ve o da 2616 yazar.



Şekil 6.2: `fork()` içinde yarış. İki süreç 2615 değerini okur; ikisinin çocuğu da 2615 kimliğini alır ve iki artıştan biri kaybolur (Silberschatz Şekil 6.2'den uyarlanmıştır).

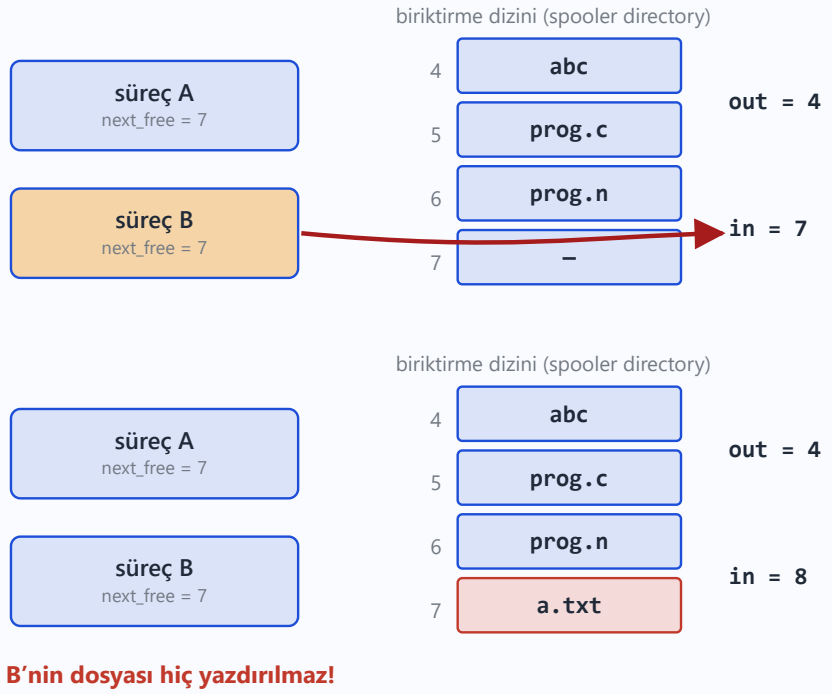
Sonuçta iki farklı sürece aynı kimlik verilmiştir ve sayaç iki yerine bir artmıştır (Şekil 6.2). İşlemler başka bir sırayla yürüseydi, örneğin P0 kesilmeden üç adımını bitirseydi, çocuklar 2615 ve 2616 kimliklerini alacaktı. Sonucun işlemlerin sırasına bağlı olması bu bölümün temel kavramının ta kendisidir.

6.1.3 Yazıcı biriktirme dizini

Tanenbaum aynı sorunu bir yazıcı kuyruğuyla anlatır. Yazdırılacak dosyaların adları bir **biriktirme dizinindeki** (*spooler directory*) numaralı yuvalara sırayla yazılır. Yazıcı süreci dizini tarar ve dosyaları yazdırır. Paylaşılan iki değişken vardır: `out`, sıradaki yazdırılacak yuvayı; `in`, ilk boş yuvayı gösterir. Başlangıçta 4, 5 ve 6 numaralı yuvalar doludur; `out = 4`, `in = 7`'dir.

A ve B süreçleri birer dosya yazdırmak ister. Her biri `in`'i yerel bir `next_free` değişkenine okur, dosya adını o yuvaya yazar ve `in`'i bir artırır. Şekil 6.3 şu ara yerleştirmeyi gösterir:

1. A, `in`'i okur ve `next_free = 7` yapar. Tam o anda zaman dilimi biter.
2. B de `in`'i okur: yine 7.
3. B, `b.txt` adını 7. yuvaya yazar ve `in = 8` yapar.
4. A kaldığı yerden devam eder. Yerel değişkeni hâlâ 7 olduğu için `a.txt`'yi 7. yuvaya yazar, B'nin girdisini ezer ve `in = 8` yapar.



Şekil 6.3: Yazıcı biriktirme dizininde yarış. (a) A ve B, `in` 'i okumuş ve ikisinin de `next_free` değeri 7'dir. (b) Önce B 7. yuvaya `b.txt` 'yi yazar, sonra A aynı yuvaya `a.txt` 'yi yazar. Dizin tutarlı görünür ama B'nin dosyası hiç yazdırılmaz.

Dizin dışarıdan bakınca tutarlıdır: `in = 8` ve yuvaların hepsi dolu. Yazıcı süreci hiçbir tutarsızlık fark etmez. Ama B'nin çıktısı kaybolmuştur ve B kullanıcısı yazıcının başında boşuna bekler.

Tanım 6.2

Yarış durumu (*race condition*), iki ya da daha çok süreç paylaşılan bir veriye eriştiğinde nihai sonucun erişimlerin yürütme sırasına bağlı olmasıdır.

6.1.4 Tek satır, üç komut: count++

Yarış durumu için uzun bir kod gerekmez. C dilindeki tek bir `count++` satırı, makine düzeyinde üç komuta derlenir:

1. `reg = count` : değeri bellekten bir yazmaca yükle,
2. `reg = reg + 1` : yazmacı artır,
3. `count = reg` : yazmacı belleğe geri yaz.

`count--` da aynı üç adımdan oluşur; yalnız ortadaki işlem çıkarmadır. Her iş parçacığının kendi yazmaç kümesi vardır. Bir iş parçacığı kesildiğinde yazmaçları PCB'sine kaydedilir; yazmaçtaki ara değer belleğe yansımaz.

Örnek 6.1

Başta `count = 5` 'tir. Bir üretici `count++`, bir tüketici `count--` yürütür. Doğru sonuç 5'tir. Aşağıdaki ara yerleştirmede sonuç kaçtır?

An	Kim	Komut	reg_1	reg_2	count
T_0	üretici	`reg1 = count`	5	–	5
T_1	üretici	`reg1 = reg1 + 1`	6	–	5
T_2	tüketici	`reg2 = count`	6	5	5
T_3	tüketici	`reg2 = reg2 - 1`	6	4	5
T_4	üretici	`count = reg1`	6	4	6
T_5	tüketici	`count = reg2`	6	4	**4**

Tablo 6.1: `count++` ve `count--` komutlarının bir ara yerleştirmesi.

1. **T_0–T_1.** Üretici değeri yazmacına yükler ve artırır. Yazmaçta 6 vardır ama bellekte hâlâ 5 durur. Tam bu anda kesilir.
2. **T_2–T_3.** Tüketici bellekten 5 okur ve kendi yazmacında 4'e indirir.
3. **T_4.** Üretici yeniden çalışır ve belleğe 6 yazar.
4. **T_5.** Tüketici kendi yazmacındaki 4'ü belleğe yazar ve üreticinin 6'sını ezer. Sonuç 4'tür; doğrusu 5'ti.

T_4 ile T_5 yer değiştirseydi son yazan üretici olurdu ve sonuç 6 çıkardı. İki iş parçacığının altı komutu, her birinin kendi sırası korunarak 20 farklı biçimde iç içe geçebilir. Bu 20 ara yerleştirmenin hepsi tek tek denendiğinde yalnız üç sonuç çıkar: 4, 5 ve 6. Doğru sonuç 5, ancak bir iş parçacığının üç komutu ötekinin okumasından önce tamamlanırsa elde edilir.

Burada geçen **ara yerleştirme** (*interleaving*), eş zamanlı akışların komutlarının, her akışın kendi içindeki sırası korunarak tek bir sıraya dizilmesidir. Tek çekirdekte bu sırayı çizelgeleyici belirler. Çok çekirdekte ise farklı çekirdeklerdeki komutların belleğe ulaşma sırası belirler. m ve n komutluk iki akış için olası ara yerleştirme sayısı $C(m + n, m)$'dir; üçer komutluk iki akış için $C(6, 3) = 20$'dir. Bu sayı akışlar uzadıkça çok hızlı büyür ve hepsini elle denemek olanaksızlaşır.

Yarışın bir başka yüzü, aynı programın her çalıştırmada farklı sonuç üretmesidir. Aşağıdaki iki süreç `n` adlı ortak bir sayaç kullanır: birinci süreç sürekli `n` 'i artırır, ikinci süreç ara sıra `n` 'i yazdırıp sıfırlar. Amaç belli bir sürede kaç olay sayıldığını raporlamaktır.

```
/* Süreç 1 */     /* Süreç 2 */
repeat           repeat
  n = n + 1;       print(n);
                   n = 0;
```

`n = 5` iken iki sürecin adımları üç farklı biçimde iç içe geçebilir:

- `n = n + 1; print(n); n = 0` → ekrana 6 yazılır, sayaç 0 olur. Doğru davranış budur.
- `print(n); n = 0; n = n + 1` → 5 yazılır, sayaç 1 olur. Bu da doğrudur: artış bir sonraki rapora kalmıştır.
- `print(n); n = n + 1; n = 0` → 5 yazılır ve sayaç 0 olur. Artış hiçbir raporda görünmez: *kaybolmuştur*.

Bu tür bir yürütmeye **tekrarlanamaz** (*non-reproducible*) yürütme denir. Hatalı sonuç ancak belli bir zamanlamada ortaya çıkar ve aynı girdiyle yeniden çalıştırmak onu çoğu zaman yinelemez.

Aynı yarışı gerçek bir makinede görmek kolaydır. Aşağıdaki Java programında iki iş parçacığı korumasız bir sayacı 100.000'er kez artırır. Doğru sonuç 200.000'dir.

```
public class Yaris {
  static int sayac = 0;                 /* korumasız */
  public static void main(String[] a) throws InterruptedException {
    Runnable is = () -> { for (int n = 0; n < 100_000; n++) sayac++; };
    Thread t1 = new Thread(is), t2 = new Thread(is);
    t1.start(); t2.start(); t1.join(); t2.join();
    System.out.println("sayac = " + sayac);
  }
}
```

Program JDK 21 ile aynı makinede üç kez çalıştırıldığında şu çıktıları verdi:

```
sayac = 170429
sayac = 150258
sayac = 135129
```

Bu sayılar her çalıştırmada değişir; başka bir makinede, başka bir yük altında çok farklı değerler çıkabilir. Değişmeyen tek şey, sonucun hemen hiçbir zaman 200.000 olmamasıdır. Kaybolan her artış, Tablo 6.1'deki gibi bir ara yerleştirmenin sonucudur.

> **Sık yapılan hata**
>
> Tek satırlık bir C ifadesinin bölünmez olduğunu sanmak. `count++`, `x += 2` ya da `a[i] = a[i] + 1` gibi ifadeler kaynak kodda tek satırdır ama makinede oku-değiştir-yaz dizisidir ve her adımın arasında kesilebilir. Bir başka yanılgı da "çok kısa bir işlem, araya bir şey girmez" düşüncesidir. Yarışın olasılığı düşük olabilir; ama saniyede milyonlarca kez çalışan bir kodda düşük olasılıklı olay da sık sık gerçekleşir.

6.2 Kritik bölge problemi

Yarış durumlarının ortak bir yapısı vardır: her süreç kodunun belli bir parçasında paylaşılan veriye dokunur. Sorunu çözmek için bütün programı korumak gerekmez; yalnız bu parçaların aynı anda çalışmasını engellemek yeter. Bu kesim o parçayı ve bir çözümden beklenenleri tanımlar.

6.2.1 Giriş, kritik, çıkış ve kalan bölüm

Sistemde *n* süreç olsun: $\{P_0, P_1, \ldots, P_{n-1}\}$. Her birinin kodunda paylaşılan veriye eriştiği bir bölüm vardır: bir global değişkene yazmak, bir tabloyu güncellemek, bir dosyaya yazmak gibi.

> **Tanım 6.3**
>
> **Kritik bölge** (*critical section*), bir sürecin paylaşılan veriye eriştiği ve aynı anda yalnız bir sürecin bulunması gereken kod parçasıdır. **Karşılıklı dışlama** (*mutual exclusion*) ise bir süreç kendi kritik bölgesindeyken başka hiçbir sürecin kendi kritik bölgesinde bulunmaması koşuludur.

Bir süreç kritik bölgesine girmeden önce izin ister; bu isteği yapan koda **giriş bölümü** (*entry section*) denir. Kritik bölgeden çıkarken başkalarına bunu duyurur; bu da **çıkış bölümüdür** (*exit section*). Kodun geri kalanı **kalan bölümdür** (*remainder section*). Tipik bir süreç bu dört parçayı bir döngü içinde tekrarlar (Şekil 6.4):

```
while (true) {
    /* giriş bölümü  (entry)     */
        /* kritik bölge          */
    /* çıkış bölümü  (exit)      */
        /* kalan bölüm (remainder) */
}
```



Şekil 6.4: Bir sürecin genel yapısı: izin iste (giriş), kullan (kritik bölge), bırak (çıkış) ve kalan bölüm. Döngü sürekli tekrarlanır.

Kritik bölge problemi, süreçlerin işbirliği yapmasını sağlayacak bir giriş ve çıkış protokolü tasarlamaktır. Kritik bölgenin ve kalan bölümün içeriği protokolü ilgilendirmez. Önemli olan, giriş ve çıkış bölümlerinin hangi zamanlamada olursa olsun doğru çalışmasıdır.

6.2.2 Bir çözümün üç koşulu

Karşılıklı dışlama tek başına yetmez. Hiç kimseyi kritik bölgeye sokmayan bir protokol de dışlamayı "sağlar" ama işe yaramaz. Bu yüzden Silberschatz bir çözümden üç koşul ister.

Tanım 6.4

1. **Karşılıklı dışlama.** P_i kritik bölgesindeyken başka hiçbir süreç kendi kritik bölgesinde yürütülemez.
2. **İlerleme** (*progress*). Kritik bölge boşsa ve girmek isteyen süreçler varsa, kimin gireceğine yalnız kalan bölümünde olmayan süreçler karar verir. Bu karar süresiz ertelenemez.
3. **Sınırlı bekleme** (*bounded waiting*). Bir süreç girme isteğini yaptıktan sonra, isteği karşılanana kadar öteki süreçlerin kritik bölgeye kaç kez girebileceğinin bir üst sınırı vardır.

Birinci koşul bir **güvenlik** (*safety*) özelliğidir: "kötü bir şey asla olmaz". İkinci ve üçüncü koşul **canlılık** (*liveness*) özellikleridir: "iyi bir şey sonunda olur". İlerleme koşulu sistemin bütün olarak takılmamasını, sınırlı bekleme ise tek tek her sürecin sonunda sıra almasını güvenceye alır. Sınırlı bekleme sağlanırsa hiçbir süreç **açlığa** (*starvation*) düşmez.

Her üç koşul da süreçlerin hızı hakkında bir varsayım yapmaz. Yalnız her sürecin sıfırdan büyük bir hızla ilerlediği kabul edilir; süreçlerin birbirine göre göreli hızları bilinmez. Tanenbaum aynı gereksinimleri dört madde olarak yazar. Tablo 6.2 bu maddelerin üç koşulla eşleşmesini verir.

Tanenbaum'un dört koşulu	Karşılığı
İki süreç aynı anda kritik bölgede olmamalı.	Karşılıklı dışlama
CPU hızı ve sayısı hakkında varsayım yapılmamalı.	Her süreç sıfırdan büyük hızla ilerler; göreli hız bilinmez.
Kritik bölge dışındaki süreç ötekileri engellememeli.	İlerleme
Hiçbir süreç sonsuza dek beklememeli.	Sınırlı bekleme

Tablo 6.2: Tanenbaum'un dört koşulu ve karşılıkları (Silberschatz 10e §6.2; Tanenbaum MOS §2.3.2).

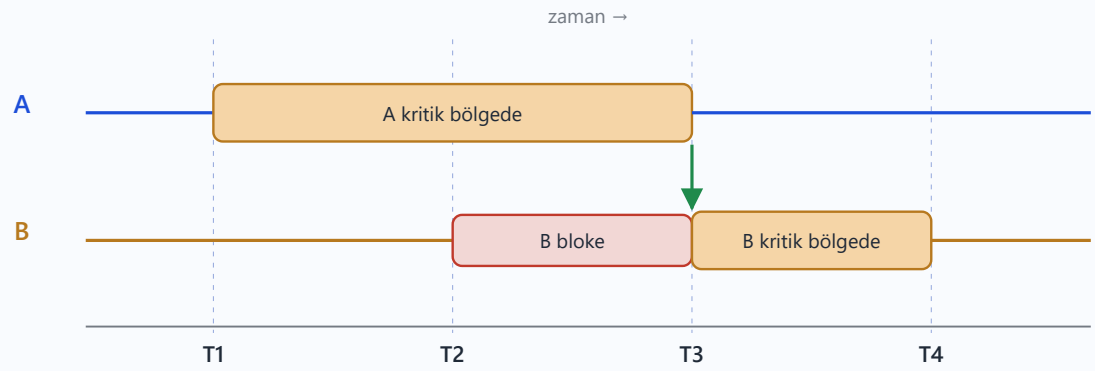
Sık yapılan hata

İlerleme ile sınırlı beklemeyi karıştırmak. İlerleme, *kritik bölge boşken* girmek isteyenlerden birinin sonlu sürede girebilmesidir. Sınırlı bekleme ise *belli bir süreç* için kaç kez başkasına yol verileceğinin sınırıdır. Bir kilit ilerleme koşulunu sağlayıp sınırlı beklemeyi bozabilir: her seferinde biri girer ama hep aynı şanssız süreç dışarıda kalır.

6.2.3 Kritik bölge zaman çizelgesi

İyi bir çözümün davranışı Tanenbaum'un zaman çizelgesinde açıkça görülür (Şekil 6.5):

1. **T1.** A kritik bölgesine girer.
2. **T2.** B de girmek ister. Bölge dolu olduğu için B bloke olur ya da meşgul bekler.
3. **T3.** A çıkar ve bekleyen B hemen girer.
4. **T4.** B çıkar.



Şekil 6.5: Kritik bölgeyle karşılıklı dışlama. A, T1–T3 arasında kritik bölgededir; B, T2'de gelir ve T3'e kadar bekler, sonra T4'e kadar kritik bölgede kalır (Tanenbaum MOS Şekil 2-22'den uyarlanmıştır).

Çizelgede hiçbir anda iki süreç birlikte kritik bölgede değildir. İyi bir çözüm bu davranışı yalnız bu zamanlamada değil, olası bütün zamanlamalarda garanti etmelidir. Üç koşuldan birini bozan tek bir ara yerleştirme, bir çözümü geçersiz kılmaya yeter.

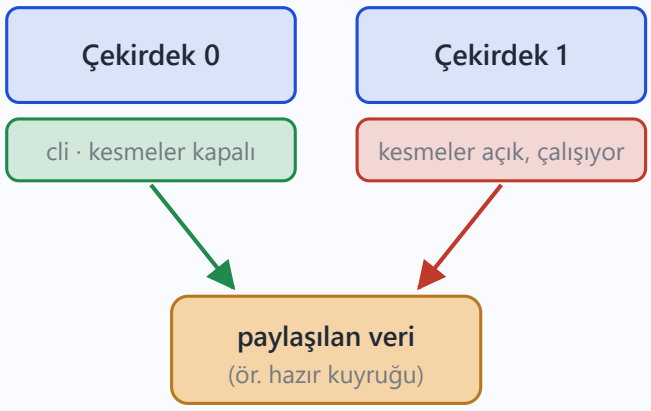
6.3 Yazılım yaklaşımları

İlk çözüm denemeleri özel bir donanım desteği kullanmaz; yalnız sıradan okuma ve yazma komutlarıyla kurulur. Bu denemelerin çoğu başarısızdır, ama neden başarısız olduklarını görmek üç koşulu anlamanın en iyi yoludur. Kesim, işletim sistemine özgü bir yolla, kesmeleri kapatmakla başlar ve Peterson algoritmasıyla biter.

6.3.1 Kesmeleri kapatmak

Tek çekirdekli bir makinede bir süreç ancak bir kesme yüzünden işlemciyi kaybeder. Süreç kritik bölgeye girmeden önce kesmeleri kapatır ve çıkınca yeniden açarsa zamanlayıcı kesmesi gelmez, bağlam değiştirme de olmaz. x86'da bu iş `cli` (kesme bayrağını temizle) ve `sti` (kesme bayrağını kur) komutlarıyla yapılır. Fikir basittir ama kullanıcı programları için uygun değildir:

- Kesmeleri açmayı unutan ya da kritik bölgede sonsuz döngüye giren bir süreç bütün sistemi kilitler. İşletim sistemi denetimi bir daha geri alamaz.
- Kesmeler kapalıyken G/Ç tamamlanma ve zamanlayıcı kesmeleri bekler. Uzun bir kritik bölge sistemi tepkisiz bırakır.
- Çok çekirdekli makinede işe yaramaz. `cli` yalnız onu yürüten çekirdeğin kesmelerini kapatır; öteki çekirdekler paylaşılan veriye erişmeyi sürdürür (Şekil 6.6).
- `cli` ve `sti` ayrıcalıklı komutlardır. Kullanıcı modundaki bir program bunları çalıştıramaz; çalıştırmaya kalkarsa işlemci bir tuzak üretir.



kesme kapatmak yalnız yerel çekirdeği durdurur

Şekil 6.6: Kesme kapatmak yalnız yerel çekirdeği durdurur. Çekirdek 0'da kesmeler kapalıyken Çekirdek 1 ça-

lışmayı ve paylaşılan veriye (örneğin hazır kuyruğuna) erişmeyi sürdürür.

Yöntem çekirdeğin içinde ise yaygındır; ama kısa süreli ve yerel olarak kullanılır. Linux'ta `local_irq_save()` yerel çekirdeğin kesmelerini kapatır ve eski durumu saklar. Bir kesme işleyicisiyle paylaşılan veride `spin_lock_irqsave()` hem kesmeleri kapatır hem de öteki çekirdeklere karşı bir spinlock alır. İki önlem birlikte gerekir: kesme kapatma aynı çekirdekteki işleyiciye, spinlock öteki çekirdeklere karşı korur.

6.3.2 Kilit değişkeni

İlk akla gelen yazılım çözümü paylaşılan bir **kilit değişkenidir**. `lock` değeri 0 ise bölge boştur, 1 ise biri içeridedir. Giren süreç değer 0 olana kadar bekler, sonra 1 yazar; çıkarken yeniden 0 yazar.

```
int lock = 0;           /* paylaşılan */

void gir(void) {
  while (lock != 0)
    ;                   /* meşgul bekle */
  lock = 1;
}
void cik(void) {
  lock = 0;
}

/* her süreç:  gir(); kritik_bolge(); cik(); */
```

Kodun hatası, “test et” ile “ata” adımlarının ayrı olmasıdır. Satır 4'teki sınama ile satır 6'daki atama arasında süreç kesilebilir. Şu ara yerleştirme karşılıklı dışlamayı bozar:

1. **Başlangıç.** `lock = 0`; bölge boştur.
2. **P0 sınar.** P0 `lock`'u okur: 0. Döngüden çıkar.
3. **P0 kesilir.** P0, satır 6'daki `lock = 1` atamasını yapmadan önce kesilir. P1 çalışır ve o da 0 okur.
4. **P1 girer.** P1 kilidi 1 yapar ve kritik bölgeye girer.
5. **P0 girer.** P0 kaldığı yerden devam eder. Sınamayı zaten yaptığı için değeri yeniden okumaz; o da 1 yazar ve girer.



Şekil 6.7: Kilit değişkeninin başarısızlığı. (a) P0 sınamayı geçmiş ama satır 6'dan önce kesilmiştir; P1 de `lock`'u 0 okur. (b) İki süreç birden kritik bölgededir.

Durum uzayının tamamı bir programla gezildiğinde aynı sonuç çıkar: iki sürecin aynı anda kritik bölgede olduğu bir durum erişilebilirdir. Sorun özünde döngüseldir. Kilidin kendisi paylaşılan bir değişkendir ve onu güvenle sınayıp değiştirmek için yine bir kilide ihtiyaç duyulur. Bu yüzden sıradan okuma ve yazmayla kurulan bir kilit değişkeni tek başına yetmez. Çözüm, sınama ve atamayı tek bir bölünmez komutta birleştirmektir; bu komutu donanım sağlar. Yalnız okuma ve yazmayla daha ileri gidilebildiğini ise katı sıra ve Peterson algoritması gösterir.

6.3.3 Katı sırayla değişim

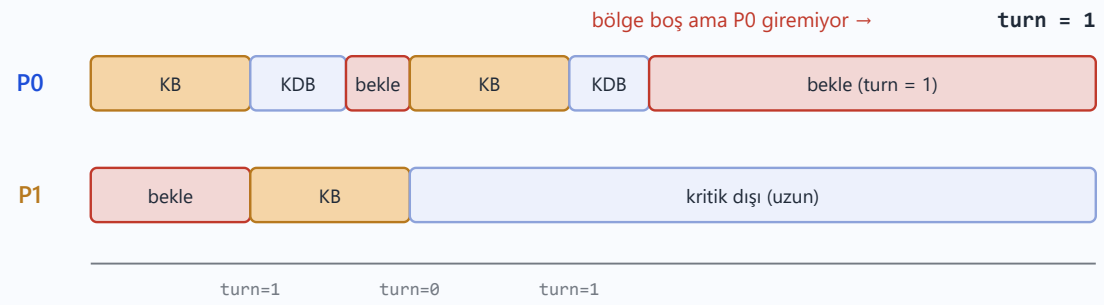
İkinci deneme kilit yerine bir sıra değişkeni kullanır. Paylaşılan `turn` değişkeni kritik bölgeye sıranın kimde olduğunu gösterir. P0 ancak `turn == 0` iken, P1 ancak `turn == 1` iken girer. Çıkan süreç sırayı ötekine verir.

```
/* P0 */
while (true) {
  while (turn != 0) ;  /* bekle */
  kritik_bolge();
  turn = 1;
  kritik_disi();
}
```

```
/* P1 */
while (true) {
  while (turn != 1) ;  /* bekle */
  kritik_bolge();
  turn = 0;
  kritik_disi();
}
```

Bu çözüm karşılıklı dışlamayı sağlar: `turn` aynı anda hem 0 hem 1 olamaz ve yalnız kritik bölgeden çıkan süreç onu değiştirir. Sınırlı bekleme de sağlanır: sıra her çıkışta öteki sürece geçer, bu yüzden bir süreç en fazla bir giriş bekler. Sorun ilerleme koşulundadır. Şu senaryoda bu açıkça görülür (Şekil 6.8):

1. `turn = 0` iken P0 girer; P1 meşgul bekler.
2. P0 çıkarken `turn = 1` yapar; P1 girer.
3. P1 çıkarken `turn = 0` yapar ve uzun bir kritik dışı işe başlar. P0 bir kez daha girip çıkar ve `turn = 1` yapar.
4. P0 yeniden girmek ister ama sıra P1'dedir. Bölge boştur, P1 de kritik bölgeye ilgi göstermez; buna rağmen P0 giremez.



Şekil 6.8: Katı sırayla değişimde ilerleme koşulunun bozulması. KB kritik bölge, KDB kritik dışı bölümdür. Son aşamada bölge boş olduğu hâlde kritik bölge dışındaki P1, P0'ı engeller.

Katı sıra süreçleri *zorla dönüşümlü* çalıştırır. Hızları farklı iki süreçten hızlı olanı, yavaş olanın kalan bölümünü bitirmesini beklemek zorunda kalır. Kalan bölümünde takılan ya da hiç geri dönmeyen bir süreç, ötekini sonsuza dek dışarıda bırakır. Ayrıca bekleyen süreç `while (turn != i) ;` döngüsünde boşuna işlemci harcar.

Tanım 6.5

Meşgul bekleme (*busy waiting*), bir koşulu döngü içinde sürekli sınayarak beklemektir. Bekleyen süreç bloke olmaz; işlemciyi kullanmayı sürdürür. Meşgul bekleyen kilide **spinlock** denir.

6.3.4 Peterson algoritması

G. L. Peterson 1981'de, yalnız okuma ve yazmayla iki süreç için doğru çalışan kısa bir çözüm verdi. Algoritma kilit değişkeni fikriyle katı sıra fikrini birleştirir. Her sürecin bir `flag` bayrağı vardır: `flag[i] = true`, "P_i girmek istiyor" demektir. Ortak `turn` değişkeni ise iki süreç birlikte istediğinde kimin önce gireceğini belirler.

```
/* paylaşılan */
bool flag[2] = { false, false };
int  turn;

/* P_i  (j = 1 - i) */
while (true) {
  flag[i] = true;    /* girmek istiyorum */
  turn = j;          /* önce sen buyur  */
  while (flag[j] && turn == j)
    ;                /* meşgul bekle    */
  /* kritik bölge */
  flag[i] = false;
  /* kalan bölüm */
}
```

Giriş bölümü iki satırdır. Süreç önce niyetini bildirir (`flag[i] = true`), sonra nezaket gösterip sırayı ötekine verir (`turn = j`, yani "önce sen buyur"). Ardından öteki süreç girmek istediği ve sıra onda olduğu sürece bekler. Çıkışta yalnız kendi bayrağını indirir.

Algoritmanın üç koşulu nasıl sağladığı birkaç cümleyle gösterilebilir:

- **Karşılıklı dışlama.** İki süreç birden kritik bölgedeyse `flag[0]` ve `flag[1]` ikisi de `true` olmalıdır. Bu durumda her biri döngüden ancak `turn` kendi lehine olduğunda çıkabilir. Ama `turn` aynı anda hem 0 hem 1 olamaz. `turn`'e son yazan süreç sırayı ötekine bırakmıştır ve beklemek ona düşer.
- **İlerleme.** P_j girmek istemiyorsa `flag[j] = false`'tur ve P_i'nin döngü koşulu hemen yanlış olur: P_i beklemeden girer. Kalan bölümdeki bir süreç ötekini engellemez.
- **Sınırlı bekleme.** P_i beklerken P_j çıkar ve hemen yeniden girmek isterse, giriş bölümünde `turn = i` yazmak zorundadır. Bu atama P_i'nin önünü açar. P_i en fazla bir giriş bekler.

Peterson algoritması yalnız iki süreç içindir. *n* süreç için Lamport'un fırıncı (*bakery*) algoritması vardır: her süreç bir sıra numarası alır ve en küçük numaralı süreç girer. Bu algoritmaların iki önemli bedeli vardır. Birincisi meşgul beklemedir; bekleyen süreç işlemci yakar. İkincisi daha önemlidir: doğruluk kanıtı, bütün işlemcilerin bellek işlemlerini tek bir ortak sırada gördüğünü varsayar. Bu varsayıma **sıralı tutarlılık** denir ve modern donanım onu kendiliğinden sağlamaz.

Sık yapılan hata

Giriş bölümündeki iki atamanın sırasını önemsiz sanmak. `turn = j` önce, `flag[i] = true` sonra yazılırsa algoritma bozulur. P0 `turn = 1` yazar ve kesilir. P1 `turn = 0` ve `flag[1] = true` yazar; `flag[0]` henüz `false` olduğu için girer. P0 devam edip `flag[0] = true` yazar; `turn` 0 olduğu için koşul yanlıştır ve P0 da girer. Durum uzayının tamamı gezildiğinde bu sıranın iki süreci birlikte kritik bölgeye soktuğu doğrulanır; doğru sırayla ise böyle bir durum hiç erişilemez.

6.3.5 Kod izleme: Peterson

Algoritmanın davranışını en iyi bir ara yerleştirmeyi adım adım izlemek gösterir. Aşağıdaki kod giriş ve çıkış bölümlerini `gir(i)` ve `cik(i)` fonksiyonlarına ayırır. Bellek sıralı tutarlı varsayılır: her yazma hemen bütün süreçlere görünür.

```
bool flag[2] = { false, false };
int turn;
void gir(int i) {
  int j = 1 - i;
  flag[i] = true;   /* istiyorum */
  turn = j;         /* sen buyur */
  while (flag[j] && turn == j)
    ;               /* meşgul bekle */
}
void cik(int i) {
  flag[i] = false;
}
```

Tablo 6.3 iki sürecin neredeyse aynı anda girmek istediği bir senaryoyu izler. Her satır bir adımın ardından paylaşılan değişkenleri ve süreçlerin durumunu verir. “Giriyor” giriş bölümünde olmayı, “dönüyor” bekleme döngüsünde olmayı, KB ise kritik bölgede olmayı gösterir.

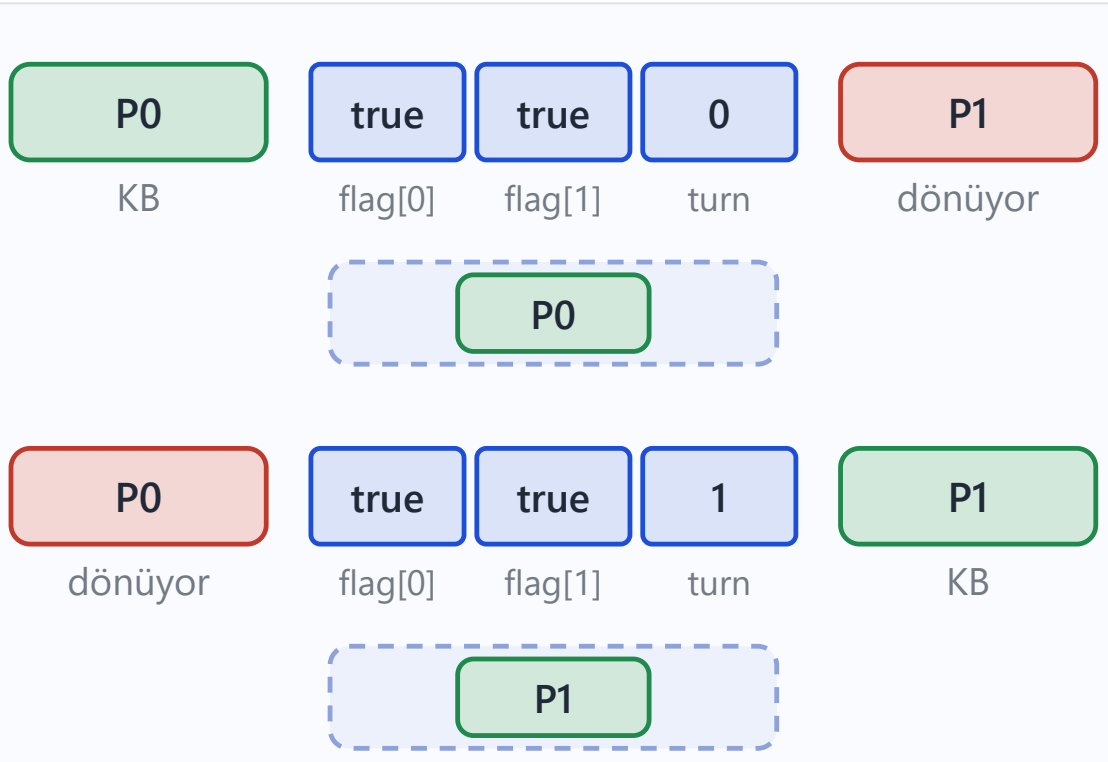
Adım	Olay	flag[0]	flag[1]	turn	P0	P1
1	başlangıç	false	false	?	kalan	kalan
2	P0: `flag[0] = true`	true	false	?	giriyor	kalan
3	P1: `flag[1] = true`	true	true	?	giriyor	giriyor
4	P0: `turn = 1`	true	true	1	giriyor	giriyor
5	P1: `turn = 0`	true	true	0	giriyor	giriyor
6	P0 sınar: `turn == 1` yanlış	true	true	0	KB	giriyor
7	P1 sınar: koşul doğru	true	true	0	KB	dönüyor
8	P0: `flag[0] = false`	false	true	0	kalan	dönüyor
9	P1 sınar: `flag[0]` false	false	true	0	kalan	KB

Adım	Olay	flag[0]	flag[1]	turn	P0	P1
10	P0: `flag[0] = true`, `turn = 1`	true	true	1	giriyor	KB
11	P0 sınar: koşul doğru	true	true	1	dönüyor	KB
12	P1: `flag[1] = false`	true	false	1	dönüyor	kalan
13	P0 sınar: `flag[1]` false	true	false	1	KB	kalan

Tablo 6.3: Peterson algoritmasının adım adım izlenmesi (sıralı tutarlı bellek).

Adım 4 ve 5 izlemenin kilit noktasıdır. İki süreç de sırayı ötekine verir; ama `turn` tek bir değişkendir ve son yazanın değeri kalır. P1 en son yazdığı için `turn = 0` olur ve beklemek P1'e düşer. Adım 6'da P0'ın koşulu `flag[1] && turn == 1`, yani `true && false` olur; P0 girer. Adım 7'de P1'in koşulu `flag[0] && turn == 0` doğrudur ve P1 döner. Karşılıklı dışlama korunur.

Adım 8'de P0 çıkar ve bayrağını indirir. P1'in koşulu yanlışa döner ve P1 girer (adım 9): ilerleme sağlanır. Adım 10'da P0 yeniden girmek ister; bu kez `turn`'e son yazan P0'dır ve beklemek ona düşer. P1 çıkınca (adım 12) P0 girer. P1 hemen yeniden isteseydi `turn = 0` yazıp bekleyecekti. Böylece P0 en fazla bir giriş bekler: sınırlı bekleme de sağlanır.



Şekil 6.9: Peterson izlemesinden iki an. (a) Adım 7: `turn = 0`, P0 kritik bölgede, P1 dönüyor. (b) Adım 11: `turn = 1`, roller değişmiştir; P1 kritik bölgede, P0 dönüyor.

Bu izleme tek bir ara yerleştirmedir; doğruluk ise hepsi için geçerli olmalıdır. İki süreçlik Peterson'un küçük durum uzayı bir programla tamamen gezildi. Sıralı tutarlı bellekte ne dışlama ihlali ne de kilitlenme bulundu.

6.4 Bellek modeli

Peterson algoritmasının kanıtı, bir süreç bir değişkene yazdığında bu yazmanın hemen ve aynı sırayla herkese göründüğünü varsayar. Modern bir çok çekirdekli işlemcide bu varsayım doğru değildir. Hem işlemci hem derleyici, tek iş parçacığının sonucunu değiştirmediği sürece bellek erişimlerinin sırasını değiştirebilir. Bu kesim, bu yeniden sıralamanın Peterson'u nasıl bozduğunu ve nasıl önlendiğini anlatır.

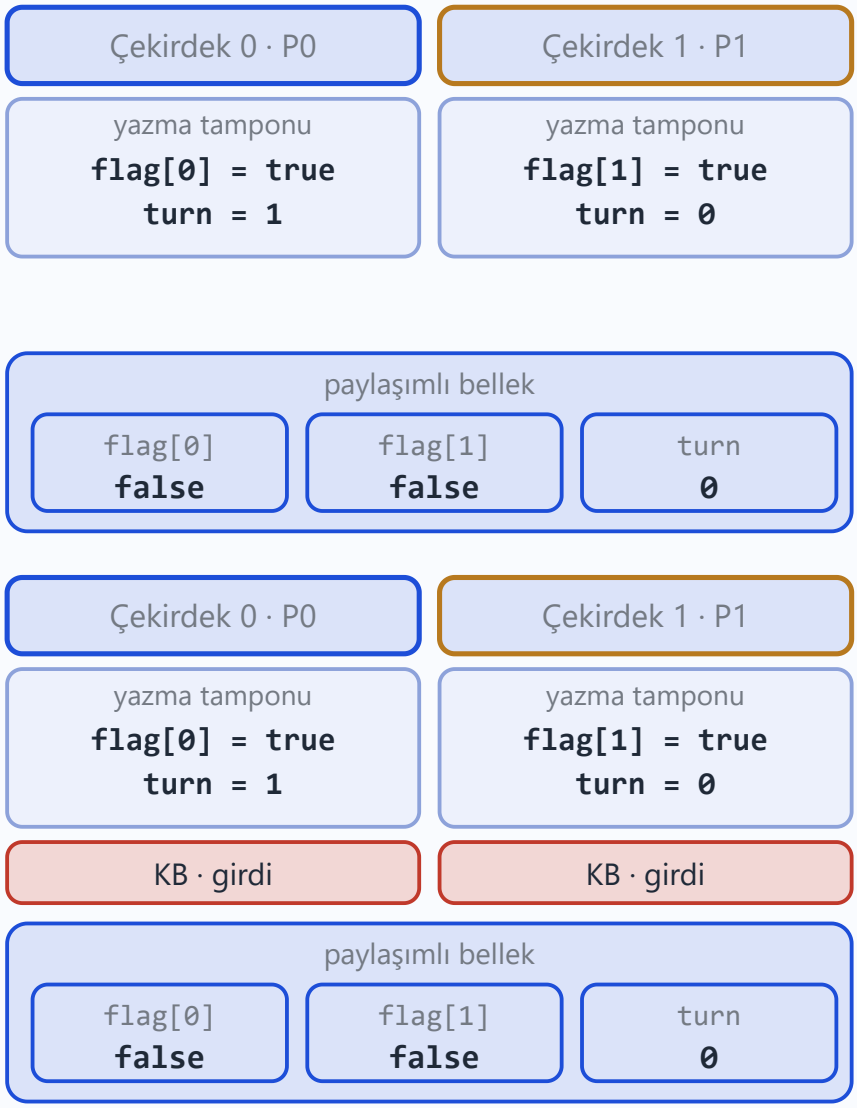
6.4.1 Peterson modern işlemcide neden bozulur?

Her çekirdeğin bir **yazma tamponu** (*store buffer*) vardır. Çekirdek bir yazma komutu yürüttüğünde değer önce bu tampona gider. Çekirdek yazmanın belleğe ulaşmasını beklemeden sonraki komutlara geçer; tampon arka planda belleğe boşaltılır. Bu düzen yazmaları çok hızlandırır. Aynı çekirdek sonra o adresi okursa değeri kendi tamponundan alır. Öteki çekirdekler ise tampon boşalana kadar eski değeri görür.

```
/* P_i  (i = 0 ya da 1, j = 1 - i) */
flag[i] = true;
turn = j;
while (flag[j] && turn == j)
  ;
/* kritik bölge */
flag[i] = false;
```

Peterson'un giriş bölümünde her süreç önce iki yazma (`flag[i]` ve `turn`), sonra bir okuma (`flag[j]`) yapar. Yazma tamponu varken şu yürütme mümkündür (Şekil 6.10):

1. **Başlangıç.** Bellekte iki bayrak da `false`'tur.
2. **P0 yazar.** P0 `flag[0] = true` ve `turn = 1` yazar. İki yazma da Çekirdek 0'ın tamponunda bekler; bellekte henüz görünmez.
3. **P1 yazar.** P1 de aynısını yapar; yazmaları Çekirdek 1'in tamponundadır.
4. **P0 okur.** P0, `flag[1]`'i bellekten okur: `false`. Döngüden çıkar ve kritik bölgeye girer.
5. **P1 okur.** P1 de `flag[0]`'ı bellekten `false` okur ve girer. İki süreç birden kritik bölgededir.
6. **Tamponlar boşalır.** Bellek sonunda `flag[0] = flag[1] = true` olur. Ama iş işten geçmiştir.



Şekil 6.10: Yazma tamponu Peterson'u bozar. (a) İki sürecin yazmaları kendi tamponlarında bekler; bellekte bayraklar hâlâ `false`'tur. (b) İki süreç de karşı bayrağı bellekten `false` okur ve kritik bölgeye girer.

Burada hiçbir komut "yanlış" çalışmamıştır. Her çekirdek kendi bakış açısından programı sırasıyla yürütmüştür. Sorun, bir çekirdeğin yazmasının öteki çekirdeğe, o çekirdeğin sonraki okumasından *sonra* ulaşmasıdır. Dışarıdan bakan için etki, "yazma, sonraki okumadan sonraya kaydı" gibidir. Derleyici de aynı etkiyi yaratabilir: `flag[j]` okumasını yazmaların önüne alabilir ya da döngüde değişmeyen bir değeri bir kez okuyup yazmaçta tutabilir. Bu not için yazma tamponlarıyla kurulan modelde Peterson'un bütün yürütmeleri gezildi. Bariyersiz sürümde iki sürecin birlikte girdiği bir yürütme bulundu; tam bariyerli sürümde bulunmadı.

6.4.2 Bellek modelleri ve yeniden sıralama

Bir işlemcinin hangi yeniden sıralamalara izin verdiğini onun **bellek modeli** belirler. Farkı en kısa yoldan gösteren deney, "yazma tamponu testi" (*store-buffer litmus test*) olarak anılır:

```
/* x = y = 0 */
/* T1 */        /* T2 */
x = 1;          y = 1;
r1 = y;         r2 = x;

/* r1 == 0 && r2 == 0 ?  */
```

Örnek 6.2

Başlangıçta `x = y = 0` 'dır. T1 ve T2 yukarıdaki iki satırı yürütür. Hangi (`r1` , `r2`) sonuçları mümkündür?

1. **Ara yerleştirme sayısı.** Her iş parçacığının iki komutu vardır. Kendi sıraları korunarak C(4, 2) = 6 ara yerleştirme oluşur.
2. **Sıralı tutarlılık.** Altı sıranın hepsi denenir. Her iş parçacığında yazma okumadan önce geldiği için, ilk okuma yapıldığında en az bir yazma bitmiştir. Sonuçlar (0, 1), (1, 0) ve (1, 1)'dir; (0, 0) hiçbir sırada çıkmaz.
3. **Yazma tamponlu işlemci.** T1'in `x = 1` yazması tamponda beklerken T1 `y` 'yi bellekten 0 okuyabilir. T2 için de aynısı geçerlidir. Bu yüzden x86, ARM ve RISC-V üzerinde `r1 = r2 = 0` da gözlenebilir.
4. **Tam bariyerle.** Her iş parçacığında yazma ile okuma arasına tam bellek bariyeri konursa (0, 0) yeniden imkânsız olur. Yazma tamponlu modelde bütün yürütmeler gezildiğinde yalnız SC'deki üç sonuç kalır.

Tanım 6.6

Sıralı tutarlılık (*sequential consistency*, SC), bütün iş parçacıklarının bellek işlemlerinin tek bir ortak sıraya dizilebilmesidir. Bu sırada her iş parçacığının kendi program sırası korunur ve her çekirdek aynı sırayı görür (Lamport, 1979).

Model	İzin verilen yeniden sıralama
Sıralı tutarlılık (SC)	Hiçbiri; bütün çekirdekler tek bir ortak sıra görür.
x86-64 (TSO)	Yalnız “yazma → sonraki okuma”; yazmalar kendi aralarında sıralıdır.
ARMv8, RISC-V (zayıf)	Aralarında bağımlılık yoksa okumalar ve yazmalar birbirini geçebilir.
Derleyici	Tek iş parçacığı için eşdeğerse her erişimi taşıyabilir, yazmaçta tutabilir ya da silebilir.

Tablo 6.4: Bellek modelleri ve izin verdikleri yeniden sıralamalar.

x86'nın modeline **toplam yazma sırası** (*total store order*, TSO) denir. Bu model SC'ye oldukça yakındır; yazma tamponundan doğan tek bir gevşemeye izin verir. Peterson'u bozan da tam olarak bu gevşemedir. ARM ve RISC-V gibi zayıf modellerde daha çok yeniden sıralama mümkündür. Örneğin iki

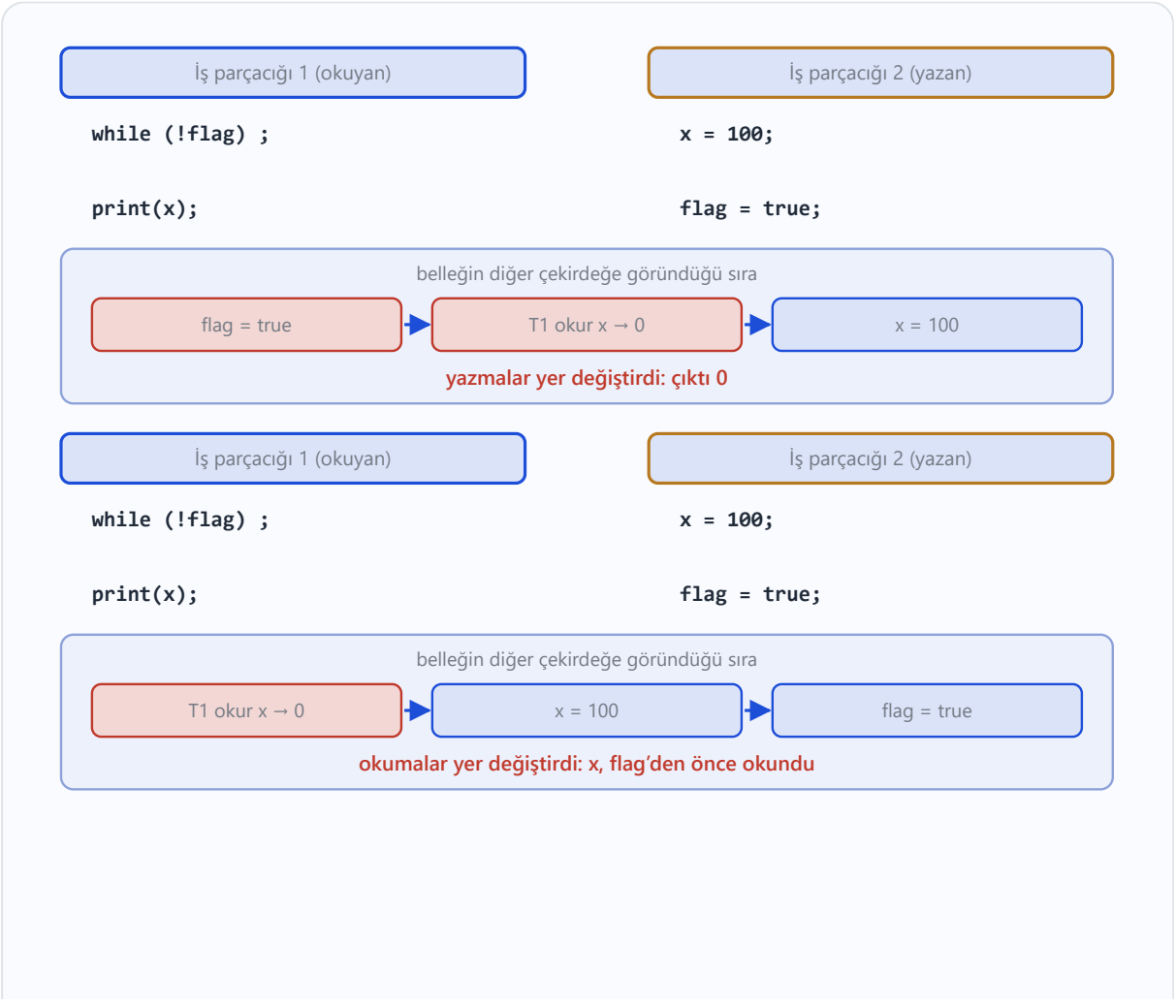
yazma, başka bir çekirdeğe yazıldıkları sıranın tersiyle görünebilir.

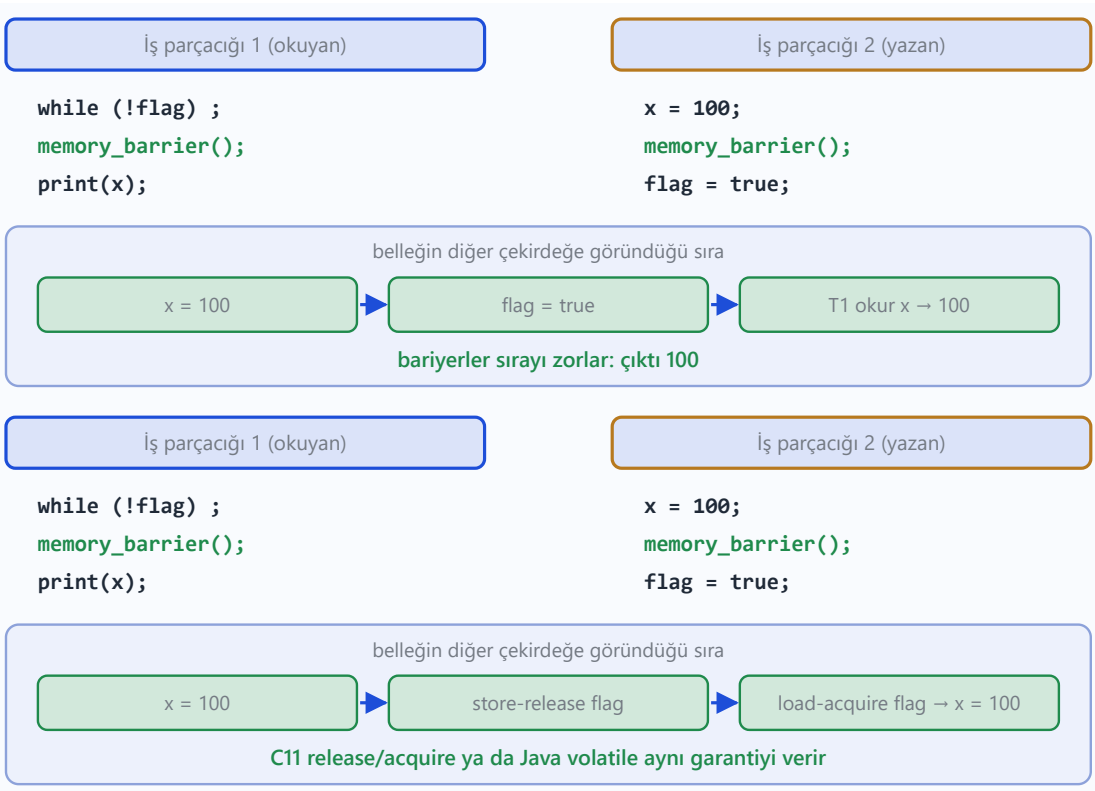
Bu farklar yüzünden eş zamanlı kod "işlemci ne yapar" sorusuna göre değil, dilin bellek modeline göre yazılır. C'deki `volatile` anahtar sözcüğü bu iş için bir araç değildir. `volatile` yalnız derleyicinin o değişkene her erişimi gerçekten yapmasını ister. İş parçacıkları için ne atomiklik ne de sıralama sağlar. Doğru araçlar C11'in `<stdatomic.h>` başlığı, Java'nın `volatile` alanları ve `java.util.concurrent` paketi ile kilitlerdir.

6.4.3 Bellek bariyerleri

Silberschatz'ın örneğinde iki iş parçacığı bir bayrakla haberleşir. İş parçacığı 2 önce `x = 100`, sonra `flag = true` yazar. İş parçacığı 1, `flag` doğru olana kadar bekler ve sonra `x`'i yazdırır. Programcının beklentisi 100 yazılmasıdır. Zayıf bir bellek modelinde iki şey bu beklentiyi bozar (Şekil 6.11):

- **Yazmalar yer değiştirir.** İş parçacığı 2'nin yazmaları öteki çekirdeğe ters sırayla ulaşır. İş parçacığı 1 `flag`'i `true` görür ama `x` hâlâ 0'dır.
- **Okumalar yer değiştirir.** İş parçacığı 1'in kendisi `x`'i `flag`'den önce okuyabilir. Döngüden çıktığında elindeki `x` eski değerdir.





Şekil 6.11: Bellek bariyerleri. (a) Bariyer yokken yazmalar yer değiştirir ve çıktı 0 olur. (b) Okumalar yer değiştirir: `x`, `flag`'den önce okunur. (c) İki tarafa da bariyer konunca sıra zorlanır ve çıktı 100 olur. (d) C11 release/acquire atomikleri ya da Java `volatile` aynı garantiyi daha ucuza verir.

Tanım 6.7

Bellek bariyeri (*memory barrier* ya da *fence*), kendinden önceki bütün bellek erişimlerinin, sonrakilerden önce tamamlanmasını ve öteki çekirdeklere görünmesini zorlayan komuttur.

Örnekte iş parçacığı 2 iki yazmanın arasına bir bariyer koyar: `x`'in yazılması kesinlikle `flag`'den önce görünür. İş parçacığı 1 de döngüden sonra bir bariyer koyar: `x` okuması kesinlikle `flag` okumasından sonra yapılır. İki bariyer birlikte çıktının 100 olmasını garanti eder. Gerçek kodda bariyerin adı platforma göre değişir: x86'da `mfence`, ARM'da `dmb`, Linux çekirdeğinde `smp_mb()`, C11'de `atomic_thread_fence`.

Tam bariyer pahalı bir komuttur, çünkü yazma tamponunun boşalmasını bekler. Çoğu durumda daha hafif bir garanti yeter. Yazan taraf `flag`'i **release** anlamıyla yazar: önceki bütün erişimler bu yazmadan önce görünür. Okuyan taraf `flag`'i **acquire** anlamıyla okur: sonraki erişimler bu okumadan önce yapılmaz. Kilit ve semafor gerçeklemeleri bu bariyerleri zaten içerir. Bu yüzden kilitle korunan kodda ayrıca bariyer yazmak gerekmez. Bariyerler, kilitleri ve atomik değişkenleri yazanların aracıdır.

Sık yapılan hata

Peterson algoritmasını C'de `volatile` değişkenlerle yazıp x86 üzerinde doğru çalışacağını sanmak. `volatile` derleyicinin okumayı yazmaçta tutmasını engeller ama işlemcinin yazma tamponunu etkilemez. Program uzun süre doğru çalışır gibi görünür, sonra ara sıra iki iş parçacığı birlikte kritik bölgeye girer. Bekleme döngüsüne `usleep()` eklemek de çare değildir; yalnız zamanlamayı değiştirir ve hatayı daha seyrek hâle getirir. Çare, `flag[i] = true` yazması ile `flag[j]` okuması arasına tam bariyer koymak ya da değişkenleri sıralı tutarlı atomikler (`_Atomic` , varsayılan `memory_order_seq_cst`) olarak tanımlamaktır.

6.4.4 Peterson'un Java'da doğru yazılışı

Java bellek modelinde `volatile` alanlara yapılan okuma ve yazmalar sıralı tutarlıdır. Derleyici ve JVM bu erişimlerin sırasını korumak için gereken bariyerleri kendisi ekler. Bu yüzden Peterson algoritması ayrı `volatile` alanlarla Java'da doğru çalışır:

```
public class Peterson {
  static volatile boolean flag0, flag1;
  static volatile int turn;
  static int sayac = 0;

  static void kilitle(int i) {
    if (i == 0) { flag0 = true; turn = 1;
      while (flag1 && turn == 1) Thread.onSpinWait(); }
    else        { flag1 = true; turn = 0;
      while (flag0 && turn == 0) Thread.onSpinWait(); }
  }
  static void birak(int i) {
    if (i == 0) flag0 = false; else flag1 = false;
  }
  /* main: i = 0 ve i = 1 için iki iş parçacığı,
     her biri 100_000 kez: kilitle(i); sayac++; birak(i);
     ikisi join edilir, sayac yazdırılır */
}
```

Kodda iki bayrak ayrı alanlardır ve `turn` de `volatile` 'dır. `Thread.onSpinWait()` (Java 9 ve sonrası) işlemciye “meşgul bekliyorum” ipucu verir; x86'da `pause` komutuna dönüşür. Bu komut döngünün güç tüketimini ve aynı çekirdeği paylaşan donanım iş parçacığına verdiği zararı azaltır. Sunumdaki kodun `main` kısmı yorum olarak özetlenmişti; çalıştırılabilir tam hâli şöyledir:

```
public class Peterson {
  static volatile boolean flag0, flag1;
  static volatile int turn;
  static int sayac = 0;

  static void kilitle(int i) {
    if (i == 0) { flag0 = true; turn = 1;
      while (flag1 && turn == 1) Thread.onSpinWait(); }
    else        { flag1 = true; turn = 0;
      while (flag0 && turn == 0) Thread.onSpinWait(); }
  }
  static void birak(int i) {
    if (i == 0) flag0 = false; else flag1 = false;
  }
  public static void main(String[] a) throws InterruptedException {
    Thread[] t = new Thread[2];
    for (int k = 0; k < 2; k++) {
      final int i = k;
      t[k] = new Thread(() -> {
        for (int n = 0; n < 100_000; n++) { kilitle(i); sayac++; birak(i); }
      });
    }
    for (Thread x : t) x.start();
    for (Thread x : t) x.join();
    System.out.println("sayac = " + sayac);
  }
}
```

Program JDK 21 ile üç kez çalıştırıldı ve her seferinde şu çıktıyı verdi:

```
sayac = 200000
sayac = 200000
sayac = 200000
```

Sayaç bu kez her çalıştırmada 200.000'dir: kritik bölge korunmuştur. Aynı sayacın korumasız sürümü ise her çalıştırmada farklı ve daha küçük değerler veriyordu. Bu programın tek bir zamanlamada doğru çalışması kanıt değildir; doğruluğun güvencesi Java bellek modelinin `volatile` için verdiği sıralı tutarlılık garantisidir.

Sık yapılan hata

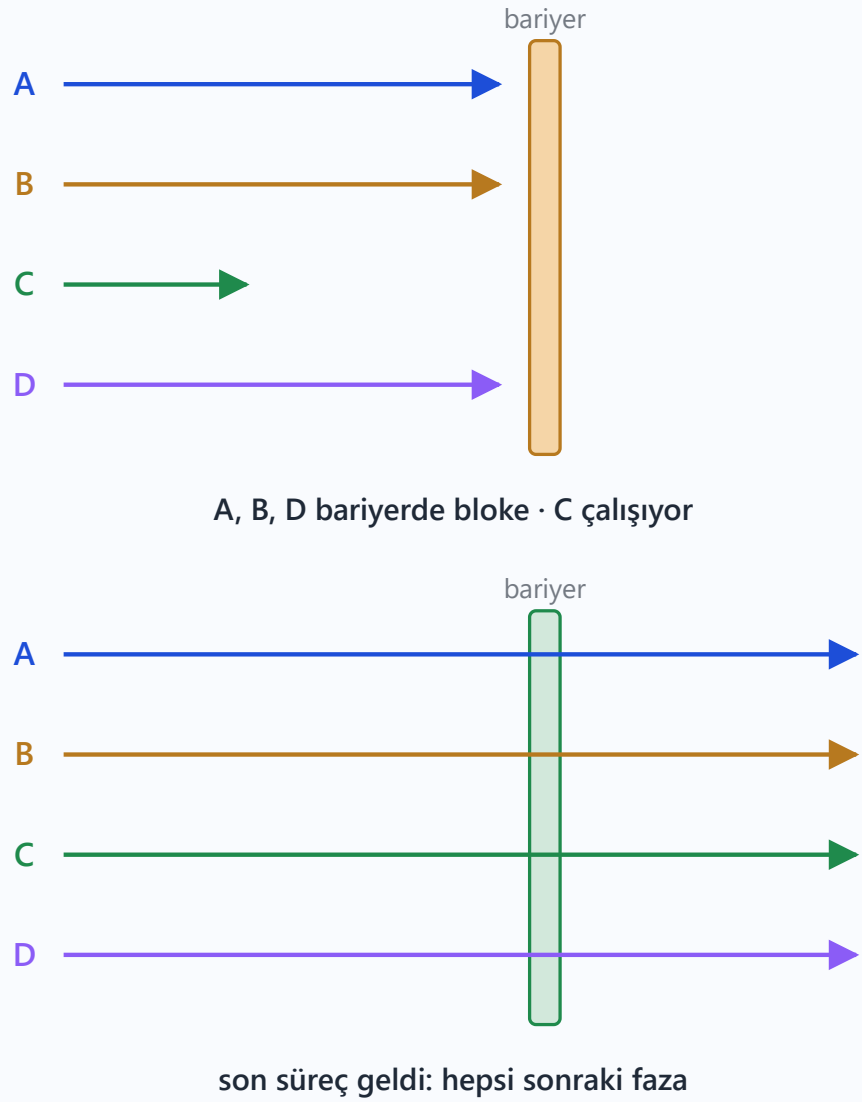
Java'da Peterson yazarken üç tuzak sık görülür. Birincisi `volatile boolean[] flag` tanımıdır: bu tanım yalnız dizi *referansını* volatile yapar, elemanları değil. Eleman okuma ve yazmaları sıradan erişim olarak kalır. İkincisi süreç numarasını `getId() % N` ile üretmektir; iki iş

parçacığı aynı `i` değerini alabilir ve ikisi de aynı bayrağı kullanır. Üçüncüsü kritik bölgeye `System.out.println` koymaktır. Yazdırma işlemi içte kendi kilidini kullanır ve zamanlamayı o kadar değiştirir ki ölçüm anlamını yitirir.

6.4.5 Senkronizasyon bariyeri

"Bariyer" sözcüğü senkronizasyonda iki farklı kavramın adıdır. Bellek bariyeri, tek bir çekirdeğin bellek erişimlerinin sırasını zorlar. **Senkronizasyon bariyeri** (*barrier*) ise bir süreç grubunu belli bir noktada buluşturur.

Fazlara bölünmüş paralel hesaplar bu buluşmaya ihtiyaç duyar. Bir ızgara simülasyonunda her süreç ızgaranın bir dilimini hesaplar. Bir sonraki faz, komşu dilimlerin bu fazdaki değerlerini kullanır. Bu yüzden hiçbir süreç, ötekiler bu fazı bitirmeden sonraki faza geçmemelidir. Fazı bitiren süreç `barrier_wait()` çağırır ve bloke olur. N'inci süreç geldiğinde bariyer açılır ve hepsi sonraki faza birlikte geçer (Şekil 6.12).



Şekil 6.12: Senkronizasyon bariyeri. (a) A, B ve D fazlarını bitirip bariyerde bloke olmuştur; en yavaş süreç C beklenir. (b) Son süreç gelince bariyer açılır ve hepsi sonraki faza geçer.

POSIX'te bu iş için `pthread_barrier_wait`, Java'da `CyclicBarrier` vardır. Aşağıdaki Java parçasında üç iş parçacığı iki faz hesaplar. Bariyere verilen ikinci argüman bir eylemdir; son iş parçacığı geldiğinde, bariyer açılmadan önce bir kez çalışır.

```
var b = new CyclicBarrier(3,
  () -> System.out.println(
          "faz " + faz[0]++ + " tamam"));
/* 3 iş parçacığı, her biri:
   for (f = 0; f < 2; f++) {
     hesapla(f); b.await();
   } */
```

Tam program her iş parçacığına fazlar arasında rastgele süre uyutarak JDK 21 ile çalıştırıldı:

```
import java.util.concurrent.CyclicBarrier;
public class Bariyer {
  public static void main(String[] a) throws Exception {
    int[] faz = {0};
    var b = new CyclicBarrier(3,
      () -> System.out.println(
              "faz " + faz[0]++ + " tamam"));
    Thread[] t = new Thread[3];
    for (int k = 0; k < 3; k++) {
      t[k] = new Thread(() -> {
        try { for (int f = 0; f < 2; f++) { Thread.sleep((long) (Math.random() * 20));
b.await(); } }
        catch (Exception e) { }
      });
      t[k].start();
    }
    for (Thread x : t) x.join();
  }
}
```

Çıktı her çalıştırmada aynıdır, çünkü eylem her fazda tam olarak bir kez ve yalnız bütün iş parçacıkları geldikten sonra çalışır:

```
faz 0 tamam
faz 1 tamam
```

İş parçacıklarının bariyere hangi sırayla vardığı ise her çalıştırmada değişir. Programın çıktısı bu sıraya bağlı olmadığı için değişkenlik dışarıdan görünmez.

6.5 Donanım desteği

Yazılım çözümlerinin ortak sorunu, bir değeri okumak ile ona yazmanın iki ayrı adım olmasıdır. Arada her şey olabilir. Donanım bu boşluğu kapatmak için bir bellek sözcüğünü kesintisiz olarak sınayıp değiştiren komutlar sunar. Bu komutlar bütün kilitlerin, semaforların ve kilitsiz veri yapılarının temelidir.

6.5.1 Atomik donanım komutları

Tanım 6.8

Atomik komut (*atomic instruction*), bir bellek sözcüğünü okuyup değiştirip yazan ve bu üç adımı bölünmeden yapan donanım komutudur. Komut yürürken başka hiçbir çekirdek o sözcüğe erişemez; komutun etkisi ya bütünüyle görülür ya hiç görülmez.

Silberschatz bu komutların davranışını C benzeri bir fonksiyonla tanımlar. Aşağıdaki `test_and_set` fonksiyonunun üç satırı gerçekte tek bir makine komutudur:

```
/* tümü TEK atomik komuttur */
bool test_and_set(bool *hedef) {
  bool eski = *hedef;
  *hedef = true;
  return eski;
}

/* kullanım: basit spinlock */
bool lock = false;
while (test_and_set(&lock))
  ;            /* meşgul bekle */
/* kritik bölge */
lock = false;
```

Fonksiyon hedefin eski değerini döndürür ve hedefe her durumda `true` yazar. Kilit boşsa (`false`) çağıran `false` alır ve kilit artık `true`'dur: kilit alınmıştır. Kilit doluysa çağıran `true` alır ve hiçbir şey değişmez: döngü yeniden dener. Kilit değişkenindeki "sınama ile atama arası" pencere ortadan kalkmıştır.

Atomiklik çok çekirdekte de geçerlidir. Eski işlemciler komut boyunca bellek veri yolunu kilitlerdi. Bugün önbellek tutarlılığı protokolü, ilgili önbellek satırını komut süresince tek bir çekirdeğe ayırır; öteki çekirdekler bu sürede o satıra erişemez. Tablo 6.5 yaygın mimarilerdeki karşılıkları verir.

Mimari	Atomik okuma-değiştirme-yazma
x86-64	`XCHG`, `LOCK CMPXCHG`, `LOCK XADD`, `LOCK BTS`
ARMv8	`LDXR` / `STXR` (LL/SC); v8.1 LSE ile `CAS`, `LDADD`, `SWP`
RISC-V "A" uzantısı	`LR` / `SC`, `AMOSWAP`, `AMOADD`
Tanenbaum'un sanal işlemcisi	`TSL REGISTER,LOCK`

Tablo 6.5: Mimarilere göre atomik okuma-değiştirme-yazma komutları.

ARM ve RISC-V'deki LL/SC çifti farklı bir yoldan aynı sonuca varır. `LDXR` (*load-linked*) değeri okur ve adresi izlemeye alır. `STXR` (*store-conditional*) ancak araya başka bir yazma girmemişse yazar; girmişse başarısız olur ve kod yeniden dener.

6.5.2 TSL ile kritik bölge

Tanenbaum aynı fikri `TSL` (*test and set lock*) adlı bir komutla anlatır. `TSL REGISTER,LOCK`, `LOCK` sözcüğünü yazmaca okur ve `LOCK`'a 1 yazar; ikisini tek bölünmez adımda yapar. Kritik bölgeye giriş ve çıkış birkaç satırlık bir assembly koduyla yazılır:

```
enter_region:
  TSL REGISTER,LOCK  | R = LOCK; LOCK = 1
  CMP REGISTER,#0    | LOCK 0 mıydı?
  JNE enter_region   | değilse: tekrar dene
  RET                | kritik bölgeye girildi

leave_region:
  MOVE LOCK,#0       | LOCK = 0
  RET
```

`enter_region`, `LOCK`'un eski değerini sınar. Değer 0 idiyse kilit boştu ve artık bu çekirdeğe aittir; fonksiyon döner. Değer 0 değilse kilit doludur ve döngü baştan dener. `leave_region` kilidi sıradan bir yazmayla 0 yapar. Tablo 6.6 iki işlemcinin aynı anda girmeye çalıştığı bir senaryoyu izler.

Adım	Olay	LOCK	R0	R1
1	kilit boş	0	–	–
2	CPU 0: `TSL` (eski 0 yazmaca, LOCK'a 1)	1	0	–
3	CPU 1: `TSL` (eski 1 yazmaca)	1	0	1
4	CPU 0: `CMP` → 0, kritik bölgede; CPU 1: `JNE` → döngü	1	0	1
5	CPU 0: `MOVE LOCK,#0`, çıktı	0	0	1
6	CPU 1: `TSL` → 0 gördü, giriyor	1	0	0

Tablo 6.6: TSL ile kritik bölge izlemesi (R0, R1: iki işlemcinin yazmaçları).

Üçüncü adım önemlidir. CPU 1'in `TSL`'i de `LOCK`'a 1 yazar; ama değer zaten 1 olduğu için bir şey değişmez. CPU 1'in yazmacına gelen 1, “kilit doluydu” bilgisidir. İki işlemci hiçbir an birlikte içeride değildir. TSL çok çekirdekte de çalışır: kesme kapatmanın aksine öteki işlemcilerin o sözcüğe erişimini de engeller (Tanenbaum MOS Şekil 2-25).

6.5.3 XCHG komutu

`XCHG` iki işlenenin içeriğini tek adımda takas eder. Yazmaca önce 1 konur, sonra yazmaç `LOCK` ile takas edilir. Takastan sonra `LOCK` her durumda 1'dir ve yazmaçta `LOCK`'un eski değeri vardır (Şekil 6.13). Etki TSL ile aynıdır.

```
enter_region:
  MOVE REGISTER,#1    | yazmaca 1 koy
  XCHG REGISTER,LOCK  | yazmaç ile LOCK'u takas et
  CMP REGISTER,#0     | LOCK 0 mıydı?
  JNE enter_region    | değilse: döngü
  RET                 | kritik bölgeye girildi
leave_region:
  MOVE LOCK,#0        | LOCK = 0
  RET
```



Şekil 6.13: `XCHG` ile atomik takas. Sonrasında yazmaçta `LOCK`'un eski değeri, `LOCK`'ta 1 vardır. Eski değer 0 ise kilit alınmıştır.

Intel x86 işlemcilerinde kilitleme için kullanılan temel komut budur. Bellek işlenenli `XCHG` örtük olarak `LOCK` önekli sayılır, yani her zaman atomiktir. C11'deki karşılığı `atomic_exchange(&lock, 1)` çağrısıdır.

6.5.4 compare_and_swap ile kilit

En esnek atomik komut **karşılaştır ve değiştir**dir (*compare-and-swap*, CAS). Komut üç işlenen alır: bir adres, beklenen değer ve yeni değer. Adresteki değer beklenene eşitse yeni değeri yazar. Her durumda adresteki eski değeri döndürür.

```
int compare_and_swap(int *v, int bekl, int yeni) {
  int eski = *v;          /* üç satır birlikte */
  if (eski == bekl)       /* TEK atomik komut  */
    *v = yeni;
  return eski;
}

int lock = 0;
void gir(void) {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ;                     /* meşgul bekle */
}
void cik(void) { lock = 0; }
```

Kilit olarak kullanıldığında beklenen değer 0, yeni değer 1'dir. Kilit boşsa CAS 1 yazar ve 0 döndürür: döngü biter. Kilit doluysa CAS hiçbir şey yazmaz ve 1 döndürür: döngü sürer. İzleme şöyle ilerler:

1. **Kilit boş.** `lock = 0`.
2. **T0 kazanır.** T0'ın CAS'ı `lock`'u 0 bulur, 1 yazar ve 0 döndürür. T0 döngüden çıkar ve kritik bölgeye girer.
3. **T1 kaybeder.** T1'in CAS'ı 1 bulur. Bu beklenen değer değildir; hiçbir şey yazmaz ve 1 döndürür.
4. **T1 bekler.** T1 meşgul bekler; her deneme bellek sözcüğünü değiştirmeden döner.
5. **T0 çıkar.** T0 kilidi 0 yapar. Gerçek kodda bu yazma release sıralamalı yapılır.
6. **T1 girer.** T1'in sıradaki CAS'ı başarılı olur.

x86'da CAS'ın karşılığı `LOCK CMPXCHG`'dir. C11'de `atomic_compare_exchange_strong`, Java'da `AtomicInteger.compareAndSet` aynı işi yapar (Silberschatz 10e §6.4.2).

> **Sık yapılan hata**
>
> Atomik komutla kurulan basit spinlock'un üç koşulu da sağladığını düşünmek. Karşılıklı dışlama atomiklik sayesinde sağlanır. İlerleme de sağlanır: kilit boşaldığında denemelerden biri mutlaka başarılı olur. Ama kilit boşaldığında hangi iş parçacığının CAS'ının önce çalışacağı belirsizdir. Şanssız bir iş parçacığı her yarışı kaybedebilir; **sınırlı bekleme** garanti edilmez.

6.5.5 CAS ile atomik artırma

CAS kilit kurmanın ötesinde, kilit kullanmadan güncelleme yapmayı da sağlar. Paylaşılan bir sayacı artırmak için iş parçacığı önce değeri okur. Sonra CAS ile "değer hâlâ okuduğum gibiyse bir fazlasını yaz" der. Arada başka biri değeri değiştirdiyse CAS başarısız olur ve döngü güncel değerle yeniden dener.

```c
void artir(int *v) {
  int temp;
  do {
    temp = *v;              /* oku              */
  } while (temp != compare_and_swap(v, temp, temp + 1));
}                           /* değişmediyse yaz */

/* C11: sayaç atomic_int ise tek satır: atomic_fetch_add(v, 1); */
```

Örnek 6.3

Başta `v = 5` 'tir. T1 ve T2 aynı anda `artir(&v)` çağırır. Sıra şöyledir: T1 okur, T2 okur, T2 CAS yapar, T1 CAS yapar, T1 yeniden dener. Her adımda `v` ve yerel değerler nedir?

Adım	Olay	v	T1.temp	T2.temp
1	T1 okur	5	5	–
2	T2 okur	5	5	5
3	T2: CAS(v, 5, 6) → 5 döner, başarılı	6	5	5
4	T1: CAS(v, 5, 6) → 6 döner, başarısız	6	5	–
5	T1 yeniden okur	6	6	–
6	T1: CAS(v, 6, 7) → 6 döner, başarılı	7	6	–

Tablo 6.7: CAS ile artırma izlemesi.

1. **Adım 1–2.** İki iş parçacığı da 5 okur. Düz `v++` olsaydı burada bir artış kaybolacaktı.
2. **Adım 3.** T2'nin CAS'ı `v` 'yi hâlâ 5 bulur ve 6 yazar. Dönen 5, `temp` 'e eşittir; T2'nin döngüsü biter.
3. **Adım 4.** T1'in CAS'ı `v` 'yi 6 bulur. Beklenen 5 değildir; CAS yazmaz ve 6 döndürür. 6 ≠ 5 olduğu için döngü tekrarlanır.
4. **Adım 5–6.** T1 güncel değeri (6) okur ve CAS'ı bu kez başarılı olur: `v = 7` .

Hiçbir artış kaybolmadı ve kimse kilit tutmadı. İki iş parçacığının bütün ara yerleştirmeleri programla gezildiğinde son değer her zaman 7 çıkar.

Bu “oku → hesapla → CAS → başarısızsa yeniden dene” döngüsü, kilitsiz veri yapılarının temel kalıbıdır. Sayaç `atomic_int` olarak tanımlanırsa C11 aynı artırmayı tek satırla yapar: `atomic_fetch_add(v, 1)` . x86 bunu döngüsüz tek bir `LOCK XADD` komutuyla yürütür.

6.5.6 Sınırlı beklemeli CAS kilidi

Basit CAS kilidinin eksiği sınırlı beklemeydi. Silberschatz bu eksiği bir `waiting[n]` dizisiyle giderir. Bekleyen her süreç kendi bayrağını kaldırır. Kritik bölgeden çıkan süreç kilidi doğrudan bırakmaz; önce süreçleri halka sırasıyla, kendinden sonrakinden başlayarak tarar. İlk bekleyeni bulursa kilidi ona devreder.

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare_and_swap(&lock, 0, 1);
  waiting[i] = false;
  /* kritik bölge */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = 0;            /* bekleyen yok */
  else
    waiting[j] = false;  /* sırayı devret */
  /* kalan bölüm */
}
```

Kodun en ince yeri devretme adımıdır. Çıkan süreç bekleyen P_j'yi bulunca `lock`'a dokunmaz; yalnız `waiting[j] = false` yapar. P_j'nin döngü koşulu `waiting[j] && key == 1` olduğu için P_j döngüden çıkar ve girer. Kilit bu sırada hep 1'de kalır, yani araya başka biri giremez. Kilit ancak bekleyen kimse yoksa gerçekten bırakılır (`lock = 0`).



Şekil 6.14: Döngüsel tarama. (a) P1 çıkarken `j = 2, 3, …` sırasıyla tarar; ilk bekleyen P3'tür. (b) P1 kilidi bırakmaz, `waiting[3] = false` yapar; P3 girer ve `lock` hâlâ 1'dir. Beş süreçte en fazla $n - 1 = 4$ sıra beklenir.

Halka taraması sınırlı beklemeyi garanti eder. Bekleyen bir süreç, kendisinden önce taranacak en fazla $n - 1$ süreç kadar giriş bekler. Tablo 6.8 dört süreçlik bir senaryoyu izler. Bu kodda satır numaraları şöyledir: 2 bayrak kaldırma, 3–4 bekleme döngüsü, 5 bayrak indirme, 7–9 tarama, 10–13 bırakma ya da devretme.

```
/* P_i; n = 4 süreç */
waiting[i] = true; key = 1;
while (waiting[i] && key == 1)
  key = compare_and_swap(&lock, 0, 1);
waiting[i] = false;
/* kritik bölge */
j = (i + 1) % n;
while ((j != i) && !waiting[j])
  j = (j + 1) % n;
if (j == i)
  lock = 0;            /* bekleyen yok */
else
  waiting[j] = false;  /* sırayı devret */
```

Adım	Olay	lock	key	j	waiting	KB
1	kilit boş, bekleyen yok	0	–	–	F F F F	–
2	P2 gelir; CAS 0 döndürür	1	0	–	F F T F	–
3	P2 `waiting[2] = false`, girer	1	0	–	F F F F	P2
4	P0 gelir; CAS 1 döndürür, döner	1	1	–	T F F F	P2
5	P3 gelir ve döner	1	1	–	T F F T	P2
6	P2 çıkar; tarama `j = 3`'te durur	1	–	3	T F F T	P2
7	P2 `waiting[3] = false`; P3 girer	1	–	3	T F F F	P3
8	P3 çıkar; `j = (3 + 1) % 4 = 0`	1	–	0	T F F F	P3
9	sıra P0'a devredilir	1	–	0	F F F F	P0
10	P0 çıkar; `j = 1, 2, 3` boş, `j` yeniden 0	1	–	0	F F F F	P0
11	`j == i`: bekleyen yok, `lock = 0`	0	–	–	F F F F	–

Tablo 6.8: Sınırlı beklemeli CAS kilidinin izlenmesi ($n = 4$; waiting dizisi P0 P1 P2 P3 sırasıyla, T/F).

Dokuzuncu adım ilginçtir. P0, P3'ten önce gelmişti ama halka sırası P3'ü önce seçti. Bu kilit gelme sırasına göre (FIFO) çalışmaz; yalnız bekleyişin bir üst sınırı olmasını sağlar. P2 çıkarken tarama kendinden sonraki `j = 3`'ten başladığı için P3 öne geçmiştir. Yine de P0 en fazla $n - 1 = 3$ giriş bekler. Kilidin gerçekten boşaldığı tek an 11. adımdır.

6.6 Kilitler

Atomik komutlar kendi başlarına kullanışsızdır; programcılar onları doğrudan değil, üzerlerine kurulmuş araçlar aracılığıyla kullanır. Bu kesim bu araçların en temel olanlarını ele alır: atomik değişkenler, spinlock ve mutex. Ardından kilit kullanan veri yapılarının nasıl ölçeklendirildiğine bakar.

6.6.1 Atomik değişkenler

Dil ve kütüphaneler atomik komutları **atomik değişkenler** olarak sunar. C11'de `_Atomic` türündeki bir değişkene yapılan her erişim atomiktir ve varsayılan olarak sıralı tutarlıdır (`seq_cst`). Bu türde `sayac++` ifadesi bile atomiktir; x86'da `lock xadd` , ARMv8.1'de `ldadd` komutuna derlenir.

```
class="kw">#include <stdatomic.h>
atomic_int sayac = 0;     /* _Atomic int */

void *is(void *arg) {
  for (int n = 0; n < 100000; n++)
    atomic_fetch_add(&sayac, 1);
  return NULL;
}
/* 2 iş parçacığı + join → 200000 */
```

Java'nın karşılığı `java.util.concurrent.atomic` paketidir. `AtomicInteger.incrementAndGet()` atomik artırma, `compareAndSet(beklenen, yeni)` ise doğrudan CAS'tır:

```
static AtomicInteger atom = new AtomicInteger();
/* iki iş parçacığı, her biri 100_000 kez: */
atom.incrementAndGet();

var v = new AtomicInteger(5);
v.compareAndSet(5, 6);   /* true,  v = 6 */
v.compareAndSet(5, 7);   /* false, v = 6 */
```

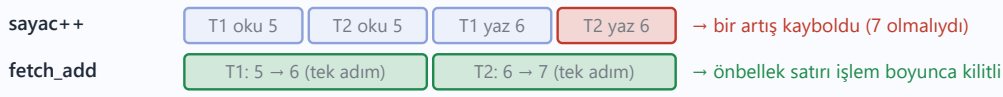
Sunumdaki parçalar tek bir programda birleştirildi. Program aynı sayacı ayrıca `synchronized` bir metotla da artırır:

```
import java.util.concurrent.atomic.AtomicInteger;
public class Atom {
  static AtomicInteger atom = new AtomicInteger();
  static int duz = 0;
  static synchronized void artir() { duz++; }
  public static void main(String[] a) throws InterruptedException {
    Runnable is = () -> { for (int n = 0; n < 100_000; n++) { atom.incrementAndGet();
artir(); } };
    Thread t1 = new Thread(is), t2 = new Thread(is);
    t1.start(); t2.start(); t1.join(); t2.join();
    System.out.println("AtomicInteger = " + atom.get());
    System.out.println("synchronized  = " + duz);
    var v = new AtomicInteger(5);
    System.out.println(v.compareAndSet(5, 6) + " " + v.get());
    System.out.println(v.compareAndSet(5, 7) + " " + v.get());
  }
}
```

JDK 21 ile üç kez çalıştırıldığında her seferinde aynı çıktıyı verdi:

```
AtomicInteger = 200000
synchronized  = 200000
true 6
false 6
```

İki yöntem de 200.000 sonucunu verir. `compareAndSet(5, 6)` değer 5 olduğu için başarılıdır. `compareAndSet(5, 7)` ise değer artık 6 olduğu için başarısızdır ve `v` 6'da kalır.



Şekil 6.15: Düz `sayac++` ile `fetch_add`. Üst satırda iki okuma aynı değeri görür ve bir artış kaybolur. Alt satırda her artış tek adımdır; önbellek satırı işlem boyunca kilitlidir.

Sık yapılan hata

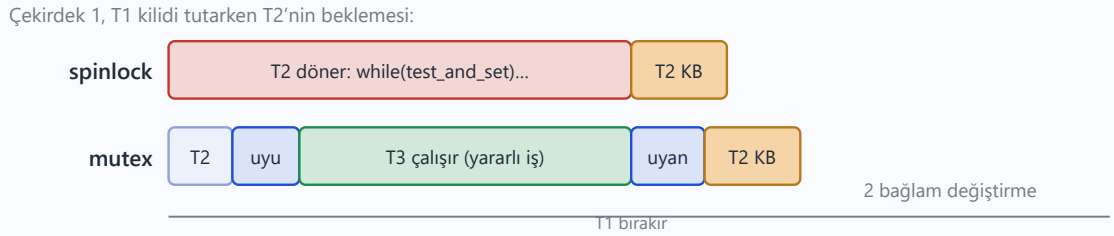
Atomik değişkenlerin birden çok değişkeni birlikte koruyacağını sanmak. Atomik değişken yalnız *tek* bir değişkenin güncellemesini bölünmez yapar. Bir tampon dizisi ile içindeki öğe sayısını birlikte tutarlı tutmak gerekiyorsa, sayacı atomik yapmak yetmez. Dizi güncellendiği hâlde sayaç henüz artmamış olabilir. Birden çok değişkenli tutarlılık için yine bir kilit ya da monitör gerekir.

6.6.2 Spinlock ve mutex

Bir kilit doluyken bekleyen iş parçacığının ne yapacağı iki ayrı kilit türü doğurur. **Spinlock** meşgul bekler: işlemciyi bırakmaz, kilidi sürekli sınar. Kilit boşalır boşalmaz bağlam değiştirme olmadan girer. **Uyuyan mutex** ise bekleyeni bloke eder; çekirdek işlemciyi başka bir iş parçacığına verir. Kilit boşalınca bekleyen uyandırılır.

Tanım 6.9

Mutex (*mutual exclusion lock*), karşılıklı dışlama için kullanılan ve sahibi olan ikili bir kilittir. `lock` kilidi alır ya da kilit boşalana kadar çağıranı bekletir; `unlock` kilidi bırakır ve yalnız kilidin sahibi çağırabilir.



Şekil 6.16: Spinlock ve uyuyan mutex. Spinlock'ta T2, T1 kilidi bırakana kadar döner ve hemen girer. Mutex'te T2 uyur, işlemci T3'e geçer ve yararlı iş yapılır; bedeli iki bağlam değiştirmedir.

Seçim, beklemenin ne kadar süreceğine bağlıdır. Kritik bölge iki bağlam değiştirmeden kısaysa dönmek uyumaktan ucuzdur. Kritik bölge uzunsa ya da içinde bloke olunabiliyorsa (örneğin disk okuması), dönmek işlemciyi boşa yakar. Tek çekirdekli bir makinede spinlock anlamsızdır: kilidi tutan iş parçacığı çalışamaz, çünkü işlemciyi dönen iş parçacığı kullanır. Bekleyen bütün dilimini yakar ve kilit ancak bir sonraki bağlam değiştirmede boşalabilir.

	Spinlock	Mutex (uyuyan)
Uygun	Çok çekirdek; kritik bölge iki bağlam değiştirmeden kısa	Uzun ya da bloke olabilen kritik bölge
Tek çekirdek	Anlamsız: tutan çalışamaz, bekleyen dilimini yakar	Doğru seçim
Örnek	Linux `spin_lock`, `pthread_spin_lock`	`pthread_mutex_t`, Java `synchronized`

Tablo 6.9: Spinlock ve uyuyan mutex karşılaştırması.

Gerçek mutex gerçeklemelerinin çoğu iki yöntemi birleştirir. Önce kısa bir süre döner, kilit bu sürede boşalmazsa uyur.

6.6.3 Mutex nasıl yazılır: yield ve futex

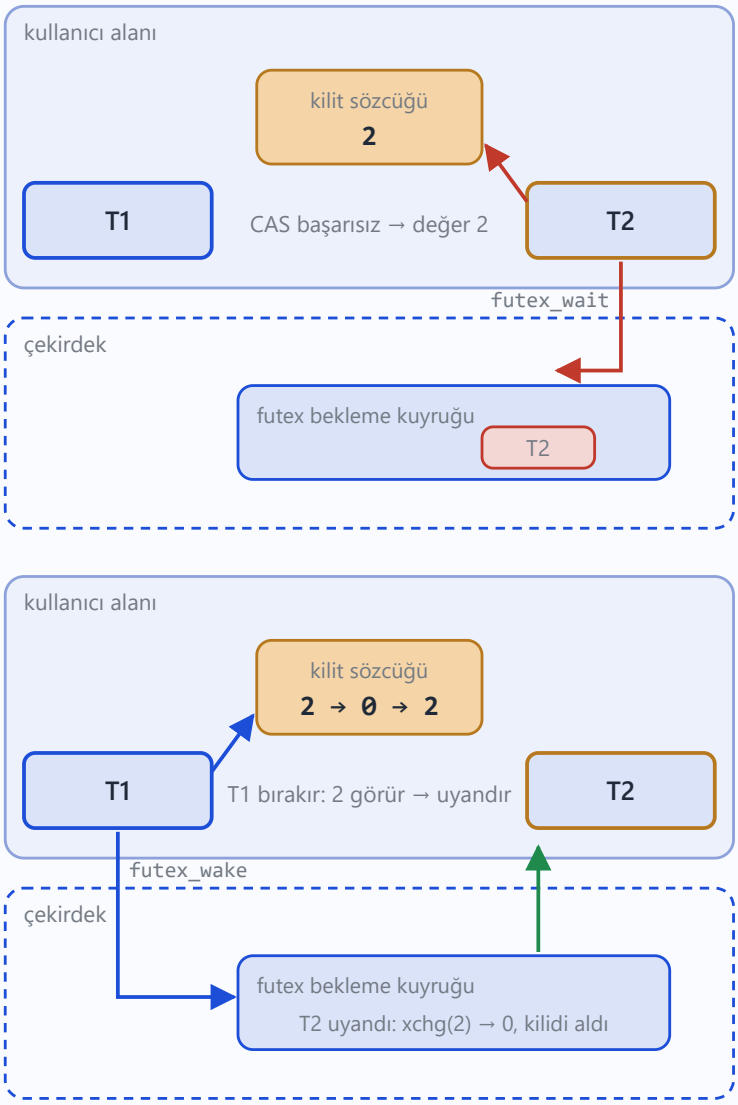
Tanenbaum, kullanıcı düzeyi iş parçacıkları için TSL ile bir mutex yazar. Kod, TSL'li `enter_region`'a çok benzer. Tek farkı, kilit doluysa döngüye dönmek yerine `thread_yield` çağırmasıdır:

```
mutex_lock:
  TSL REGISTER,MUTEX | eskisini al, 1 yap
  CMP REGISTER,#0    | boş muydu?
  JZE ok             | evet: alındı
  CALL thread_yield  | hayır: CPU'yu ver
  JMP mutex_lock     | sonra yine dene
ok: RET
mutex_unlock:
  MOVE MUTEX,#0
  RET
```

Kullanıcı düzeyi iş parçacıklarında zamanlayıcı kesmesiyle iş parçacığı değiştirilmez. Meşgul bekleyen bir iş parçacığı işlemciyi hiç bırakmaz ve kilidi tutan iş parçacığı hiç çalışamaz. `thread_yield` işlemciyi aynı süreçteki öteki iş parçacıklarına verir; kilidin sahibi çalışıp kilidi bırakabilir (Tanenbaum Şekil 2-29).

Linux'taki çözüm **futex**'tir (*fast userspace mutex*). Kilit, kullanıcı alanında bir tam sayı sözcüğüdür. Çekişme yoksa kilit yalnız bir atomik komutla alınıp bırakılır; çekirdeğe hiç girilmez. Çekirdek ancak bir iş parçacığının gerçekten uyuması gerektiğinde devreye girer. glibc'nin `pthread_mutex`'i, Java kilitleri ve Go'nun eş zamanlılık araçları bu mekanizmanın üstüne kuruludur. Ulrich Drepper'ın “Futexes Are Tricky” makalesindeki yaygın tasarımda sözcüğün üç değeri vardır: 0 boş, 1 kilitli ve bekleyen yok, 2 kilitli ve bekleyen olabilir.

1. **Çekişme yok.** T1 sözcüğü tek bir CAS ile 0'dan 1'e çevirir. Sistem çağrısı yapılmaz.
2. **T2 gelir.** T2 kilidi dolu bulur, değeri 2 (“bekleyen var”) yapar ve `futex_wait` sistem çağrısıyla çekirdekteki bekleme kuyruğunda uyur.
3. **T1 bırakır.** T1 sözcüğü 0 yapar. Eski değer 2 olduğu için `futex_wake` çağırır.
4. **T2 uyanır.** T2 sözcüğü atomik takasla (`xchg`) yine 2 yapar. Takasın döndürdüğü eski değer 0 olduğu için kilit artık T2'nindir. T2 başka bekleyen olup olmadığını bilemez; bu yüzden değeri temkinli olarak 1 değil 2 yapar.



Şekil 6.17: futex ile mutex. (a) T2 kilidi dolu bulur, sözcüğü 2 yapar ve çekirdekteki futex kuyruğunda uyur. (b) T1 bırakırken 2 görür, sözcüğü 0 yapar ve `futex_wake` çağırır; uyanan T2 sözcüğü yeniden 2 yapar ve kilidi alır.

Değer 1 ile 2'nin ayrılmasının nedeni, gereksiz `futex_wake` çağrılarından kaçınmaktır. Değer 1 ise kimse uyumuyordur ve bırakan iş parçacığı çekirdeğe hiç girmez. Uyanan iş parçacığının değeri 2 yapması ise bazen gereksiz bir `futex_wake` çağrısına yol açar; ama hiçbir bekleyenin unutulmamasını güvenceye alır. Böylece çekişmesiz durumda alma ve bırakma birer atomik komuttan ibarettir.

6.6.4 Ölçeklenebilir sayaç

Tek bir kilitle korunan bir sayaç doğrudur ama ölçeklenmez. Her artış aynı kilit için yarışır ve aynı önbellek satırı çekirdekler arasında sürekli taşınır. Çekirdek sayısı arttıkça toplam süre kısalmaz, uzar. Arpaci-Dusseau'ların **yaklaşık sayacı** bu sorunu çekirdek başına sayımla çözer. Her çekirdeğin kendi yerel sayacı ve kendi kilidi vardır. Yerel sayaç bir `S` eşiğine ulaşınca değeri genel sayaca aktarılır.

```
typedef struct {
  int global;      pthread_mutex_t gkilit;
  int yerel[NCPU]; pthread_mutex_t ykilit[NCPU];
  int S;                    /* aktarma eşiği */
} ycount_t;

void ekle(ycount_t *c, int cpu) {
  pthread_mutex_lock(&c->ykilit[cpu]);
  c->yerel[cpu]++;
  if (c->yerel[cpu] >= c->S) {        /* aktar */
    pthread_mutex_lock(&c->gkilit);
    c->global += c->yerel[cpu];
    pthread_mutex_unlock(&c->gkilit);
    c->yerel[cpu] = 0;
  }
  pthread_mutex_unlock(&c->ykilit[cpu]);
}
int oku(ycount_t *c) {              /* yaklaşık */
  pthread_mutex_lock(&c->gkilit);
  int v = c->global;
  pthread_mutex_unlock(&c->gkilit);
  return v;
}
```

`ekle` yalnız çağıran çekirdeğin yerel kilidini alır. Farklı çekirdekler farklı kilitler kullandığı için birbirlerini beklemez. Genel kilit yalnız aktarma sırasında, yani her `S` artışta bir alınır. `oku` yalnız genel sayacı döndürür; bu yüzden sonuç *yaklaşıktır*.

Örnek 6.4

Dört çekirdek ve `S = 4` vardır. Çekirdekler aşağıdaki sırayla `ekle` çağırır. Her adımdan sonra yerel sayaçlar (Y1–Y4), genel sayaç G ve gerçek toplam nedir?

t	Artıran çekirdekler	Y1	Y2	Y3	Y4	G	gerçek
0	–	0	0	0	0	0	0
1	1, 3	1	0	1	0	0	2
2	1, 2, 3	2	1	2	0	0	5
3	1, 3, 4	3	1	3	1	0	8
4	1, 2	4 → 0	2	3	1	4	10
5	3, 4	0	2	4 → 0	2	8	12

Tablo 6.10: Yaklaşık sayaç izlemesi (S = 4).

1. **t = 1–3.** Sekiz artış olur ama hiçbir yerel sayaç 4'e ulaşmaz. Genel kilide hiç dokunulmaz; G = 0'dır.
2. **t = 4.** Y1 eşiğe ulaşır. Çekirdek 1 genel kilidi alır, 4'ü G'ye ekler ve yerel sayacını sıfırlar.

3. **t = 5.** Y3 de eşiğe ulaşır ve aktarır: G = 8.
4. **Okuma.** `oku()` 8 döndürür. Gerçek değer 12'dir; aradaki 4, Y2 ile Y4'te bekler.

En kötü durumda her yerel sayaç `S − 1`'de durur ve hiç aktarılmamıştır. Bu yüzden `oku()` gerçek değerden en fazla NCPU × (S − 1) kadar geride kalabilir; örnekte 4 × 3 = 12'dir. `S`, doğruluk ile ölçeklenme arasındaki ödünleşimi belirler. Büyük `S` genel kilide seyrek gider ve iyi ölçeklenir ama okunan değer daha geride kalır. `S = 1`'de her artış genel sayaca gider: sonuç kesindir ama sayaç tek kilitli sayaç kadar yavaştır. Linux çekirdeğindeki `percpu_counter` aynı fikre dayanır.

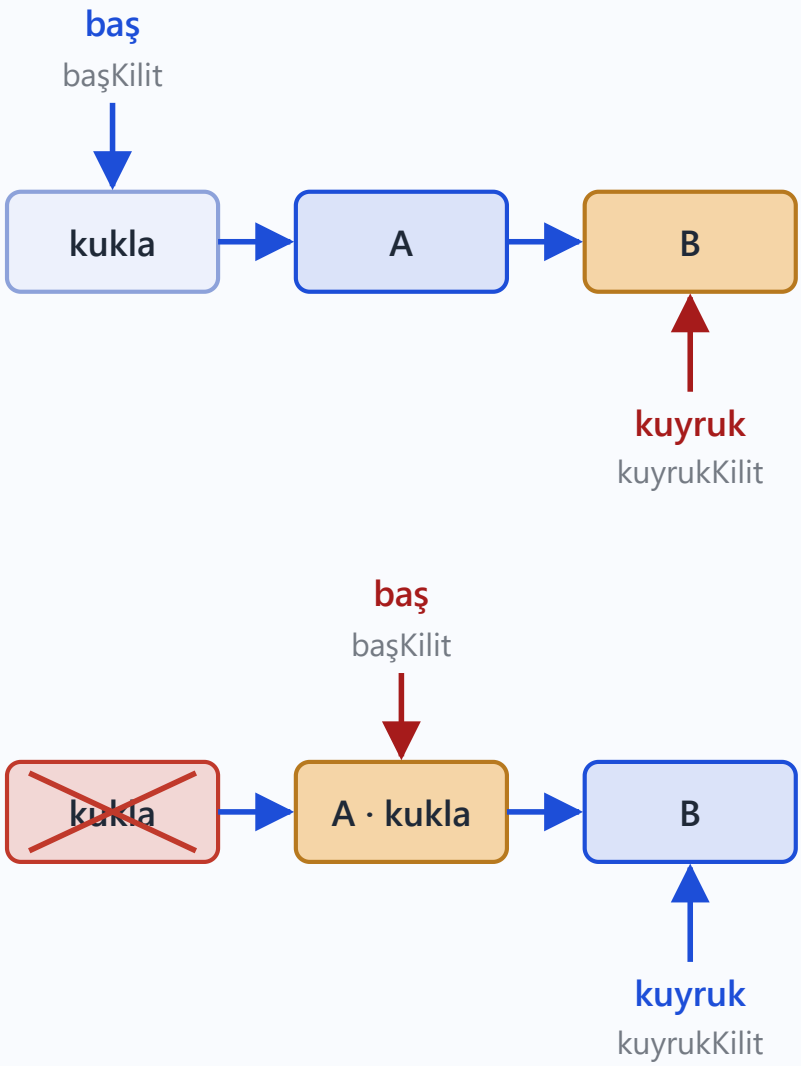
6.6.5 Eş zamanlı veri yapılarında kilit düzeni

Bir veri yapısını iş parçacığı güvenli yapmanın en kolay yolu, her işlemi tek bir kilitle sarmaktır. Bu yol doğrudur ama bütün işlemleri sıraya sokar. Kilidi bölmek paralelliği artırabilir; Tablo 6.11 yaygın düzenleri özetler.

Yapı	Kilit düzeni	Sonuç
Bağlı liste	Tek mutex	Basittir. `malloc` kilit dışında yapılmalı; her dönüş yolunda `unlock` unutulmamalıdır.
El ele kilitleme	Düğüm başına kilit: önce sonrakini al, sonra öncekini bırak	Gezinmeler paralel ilerler; ama her düğümdeki kilit bedeli çoğu zaman kazancı yer.
Kuyruk	Baş ve kuyruk için ayrı kilit, kukla düğüm	Ekleme ile çıkarma aynı anda yapılabilir.
Hash tablosu	Kova başına kilitli liste	Farklı kovalara giden işlemler birbirini beklemez; çok iyi ölçeklenir.

Tablo 6.11: Eş zamanlı veri yapılarında kilit düzenleri.

Michael ve Scott'ın 1996'da yayımladığı iki kilitli kuyruk, kilit bölmenin güzel bir örneğidir. Kuyruğun başında her zaman bir **kukla düğüm** bulunur. Başlangıçta baş da kuyruk ucu da bu kuklayı gösterir. `ekle` yalnız `kuyrukKilit`'i alır ve kuyruk ucunu ilerletir. `çıkar` yalnız `başKilit`'i alır: eski kuklayı serbest bırakır, ilk gerçek düğümün değerini döndürür ve o düğüm yeni kukla olur (Şekil 6.18). Kukla düğüm sayesinde iki işlem hiçbir zaman aynı düğümü değiştirmez.



Şekil 6.18: İki kilitli kuyruk. (a) `ekle(A)` ve `ekle(B)` yalnız `kuyrukKilit`'i alır. (b) `çıkar()` yalnız `başKilit`'i alır; eski kukla serbest bırakılır, A'nın değeri döner ve A yeni kukla olur.

Pratik kural şudur: kod önce tek kilitle doğru yazılır. Ölçüm kilidin gerçekten darboğaz olduğunu gösterirse kilit bölünür. Daha çok kilit her zaman daha hızlı demek değildir. Her kilit ek bir atomik komut ve kilitlenme riski getirir.

6.7 Semaforlar

Mutex yalnız "bir kişi girsin" der. Pek çok eşgüdüm sorunu ise daha fazlasını ister: N özdeş kaynaktan birini almak ya da bir olay gerçekleşene kadar beklemek gibi. Dijkstra'nın semaforu bu iki ihtiyacı da tek bir araçla karşılar.

6.7.1 Semafor

Tanım 6.10

Semafor (*semaphore*), yalnız iki atomik işlemle erişilen bir tam sayıdır (Dijkstra, 1965):

- `wait(S)`, P ya da `down`: değer 0 ise çağıran süreç bloke olur (uyur); değilse değer 1 azaltılır ve süreç devam eder.
- `signal(S)`, V ya da `up`: değer 1 artırılır; bekleyen bir süreç varsa biri uyandırılır.

Semaforun değeri, kullanılabilir kaynak sayısı gibi düşünülebilir. Değer 3 ise üç süreç daha beklemeden geçebilir. P ve V harfleri Dijkstra'nın Felemenkçe sözcüklerinden gelir; Tanenbaum `down` ve `up`, Silberschatz `wait` ve `signal` adlarını kullanır.

`wait` ve `signal`'ın kendileri de birer kısa kritik bölgedir: değeri okuyup değiştirirler. İşletim sistemi bu iki işlemi tek çekirdekte kesmeleri kapatarak, çok çekirdekte bir spinlock ile korur. Meşgul bekleme böylece tamamen ortadan kalkmaz; ama uygulamanın uzun kritik bölgesinden semafor işleminin birkaç komutuna iner. Değer 0 iken `down` çağıran süreç gerçek sistemlerde uyur. Kitaplarda görülen meşgul bekleyen sürüm yalnız kavramsal ilk tanımdır.

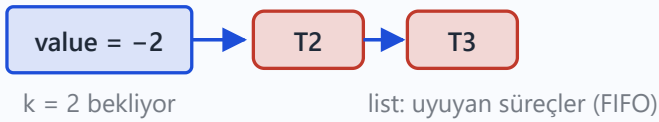
6.7.2 Bekleme kuyruklu semafor

Silberschatz'ın gerçeklemesinde semafor bir değer ve bekleyen süreçlerin listesinden oluşur:

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

```
void wait(semaphore *S) {     /* P / down */
  S->value--;
  if (S->value < 0) {
    ekle(S->list, bu_surec);
    sleep();                   /* bloke ol */
  }
}
void signal(semaphore *S) {   /* V / up */
  S->value++;
  if (S->value <= 0) {
    P = cikar(S->list);        /* FIFO */
    wakeup(P);
  }
}
```

Bu gerçeklemede `wait` önce değeri azaltır, sonra sınar. Bu yüzden değer negatif olabilir. Değer $-k$ ise k süreç kuyrukta bekliyordur (Şekil 6.19). `signal` değeri artırır; sonuç hâlâ 0 ya da negatifse kuyrukta bekleyen vardır ve baştaki uyandırılır.



Şekil 6.19: Değeri −2 olan semafor. Mutlak değer bekleyen süreç sayısıdır: T2 ve T3 listede uyur.

Örnek 6.5

Tek bir kaynak vardır ve semafor 1 ile başlar (ikili semafor, mutex gibi çalışır). T1, T2 ve T3 sırayla `wait`, sonra aynı sırayla `signal` yapar.

Olay	S.value	S.list	Kritik bölgede
başlangıç	1	boş	–
T1 `wait`	0	boş	T1
T2 `wait`	−1	T2	T1
T3 `wait`	−2	T2, T3	T1
T1 `signal`	−1	T3	T2
T2 `signal`	0	boş	T3
T3 `signal`	1	boş	–

Tablo 6.12: Bekleme kuyruklu semafor izlemesi (Silberschatz gerçeklemesi).

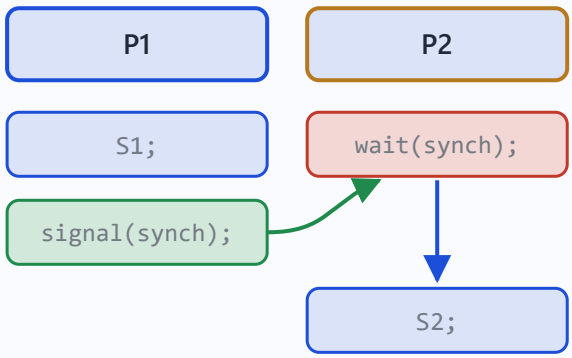
1. **T1 wait.** Değer 0 olur. Negatif olmadığı için T1 devam eder.
2. **T2 ve T3 wait.** Değer önce −1, sonra −2 olur; ikisi de kuyruğa eklenir ve uyur.
3. **T1 signal.** Değer −1 olur. Hâlâ ≤ 0 olduğu için kuyruğun başındaki T2 uyandırılır ve kritik bölgeye girer.
4. **T2 signal.** Değer 0 olur; T3 uyandırılır.
5. **T3 signal.** Değer 1 olur ve kimse beklemez.

Kuyruğun sırası semaforun tanımında belirtilmez. FIFO kuyruk kullanılırsa her bekleyen sonunda sıra alır ve bekleme sınırlıdır. Başka bir sıra (örneğin LIFO ya da önceliğe göre) açlığa yol açabilir. Tanenbaum'un tanımında ise değer hiç negatif olmaz: `up`, uyuyan biri varsa değeri artırmadan doğrudan birini uyandırır. İki tanım dışarıdan aynı davranışı verir; yalnız iç değerin okunuşu farklıdır.

6.7.3 Kullanım kalıpları

Semaforun başlangıç değeri, onun hangi iş için kullanıldığını belirler:

- **İkili semafor (mutex gibi).** Başlangıç 1'dir. Her süreç `wait` ... kritik bölge ... `signal` yapar.
- **Sayma semaforu.** Başlangıç N'dir ve N özdeş kaynağı temsil eder: bir bağlantı havuzunun bağlantıları ya da bir tamponun boş yuvaları gibi. Örneğin bir veritabanına en fazla 5 bağlantı açılabiliyorsa semafor 5 ile başlar; altıncı istek bloke olur.
- **Sinyal ya da sıralama.** Başlangıç 0'dır ve bir olay gerçekleşene kadar beklemek için kullanılır. P2'nin S2 deyimi, P1'in S1 deyiminden sonra çalışmalıysa P1 S1'den sonra `signal(synch)`, P2 S2'den önce `wait(synch)` yapar (Şekil 6.20).



Şekil 6.20: 0 ile başlayan semaforla sıralama. P2'nin `wait(synch)`'i, P1 `signal(synch)` yapana kadar bekler; böylece S2, S1'den sonra çalışır.

Sıralama kalıbında olayın önce gerçekleşmesi sorun değildir. P1 önce `signal` yaparsa değer 1 olur; P2 sonra `wait` yaptığında beklemeden geçer. Semafor uyandırmayı bir sayaç olarak saklar ve hiçbir sinyal kaybolmaz. Bu özellik semaforu koşul değişkeninden ayıran en önemli farktır.

Semafor güçlü ama yapısal olmayan bir araçtır: doğruluk, her `wait` / `signal` çiftinin doğru yazılmasına bağlıdır. Tablo 6.13 tipik hataları listeler. Tek bir unutulmuş satır, derleyicinin hiç uyarmadığı ve testte nadiren görülen bir hata yaratır.

Kod	Sonuç
`signal(m)` ... KB ... `wait(m)`	Sıra ters: birden çok süreç kritik bölgeye girer.
`wait(m)` ... KB ... `wait(m)`	Süreç kendini bekler: kalıcı olarak bloke olur.
`wait` ya da `signal` unutulur	Dışlama bozulur ya da kilitlenme olur.
İki semafor farklı sırayla alınır	Kilitlenme.

Tablo 6.13: Semaforla yapılan yaygın hatalar.

6.7.4 Kod izleme: semaforla bariyer

Senkronizasyon bariyeri iki semafor ve bir sayaçla kurulabilir. `mutex` sayacı korur. `turnike` 0 ile başlar, yani kapalıdır. Son gelen iş parçacığı turnikeyi açar. Turnikeden geçen her iş parçacığı arkasındakini geçirir:

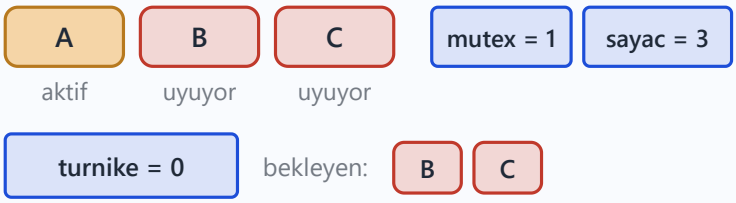
```
#define N 3
semaphore mutex = 1, turnike = 0;
int sayac = 0;
void bariyer(void) {
  down(&mutex);
  sayac = sayac + 1;
  if (sayac == N) up(&turnike); /* son */
  up(&mutex);
  down(&turnike);  /* turnikeden geç */
  up(&turnike);    /* arkadakini geçir */
}
```

Aşağıdaki izlemede Tanenbaum'un semafor anlamı kullanılır: `up`, uyuyan biri varsa değeri artırmadan onu uyandırır. Üç iş parçacığı A, B ve C sırayla bariyere gelir.

Adım	Olay	mutex	turnike	sayac	turnikede uyuyan
1	üçü de hesaplıyor	1	0	0	–
2	A gelir, mutex'i alır, `sayac = 1`	0	0	1	–
3	1 ≠ 3: A mutex'i bırakır, turnikede uyur	1	0	1	A
4	B gelir, `sayac = 2`, turnikede uyur	1	0	2	A, B
5	C gelir, `sayac = 3 = N`	0	0	3	A, B
6	C `up(&turnike)`: A uyanır	0	0	3	B
7	C mutex'i bırakır, turnikede uyur	1	0	3	B, C
8	A geçer, `up` ile B'yi uyandırır	1	0	3	C
9	B geçer, `up` ile C'yi uyandırır	1	0	3	–
10	C geçer, `up`: bekleyen yok	1	1	3	–

Tablo 6.14: Semaforla bariyer izlemesi (N = 3).

Altıncı adımda C turnikeyi açar ama bekleyen olduğu için değer artmaz; A uyandırılır. C de hemen arkasından turnikede uyur (adım 7), çünkü turnikeden ancak biri onu geçirirse geçebilir. Turnike bir kişilik döner kapı gibi çalışır: her geçen, arkadakini geçirir. Son geçen C'nin `up`'ı kimseyi bulamaz ve değer 1 olur.



Şekil 6.21: Bariyer izlemesinin 7. adımı. A uyandırılmış ve etkindir; B ve C turnikede uyur. `sayac = 3`, turnike hâlâ 0'dır.

Üç iş parçacığının bütün ara yerleştirmeleri bir programla gezildi. Her yürütmede iş parçacıkları turnikeden ancak `sayac = 3` olduktan sonra geçti ve hiçbir yürütme kilitlenmedi. Bu bariyer tek kullanımlıktır: sonunda turnike 1'de kalır ve sayaç 3'tür. Döngü içinde yeniden kullanmak için turnikenin yeniden kapatılması gerekir. Tekrar kullanılabilir sürüm iki turnike ister; Downey'nin kitabı bu sürümü ayrıntılı işler.

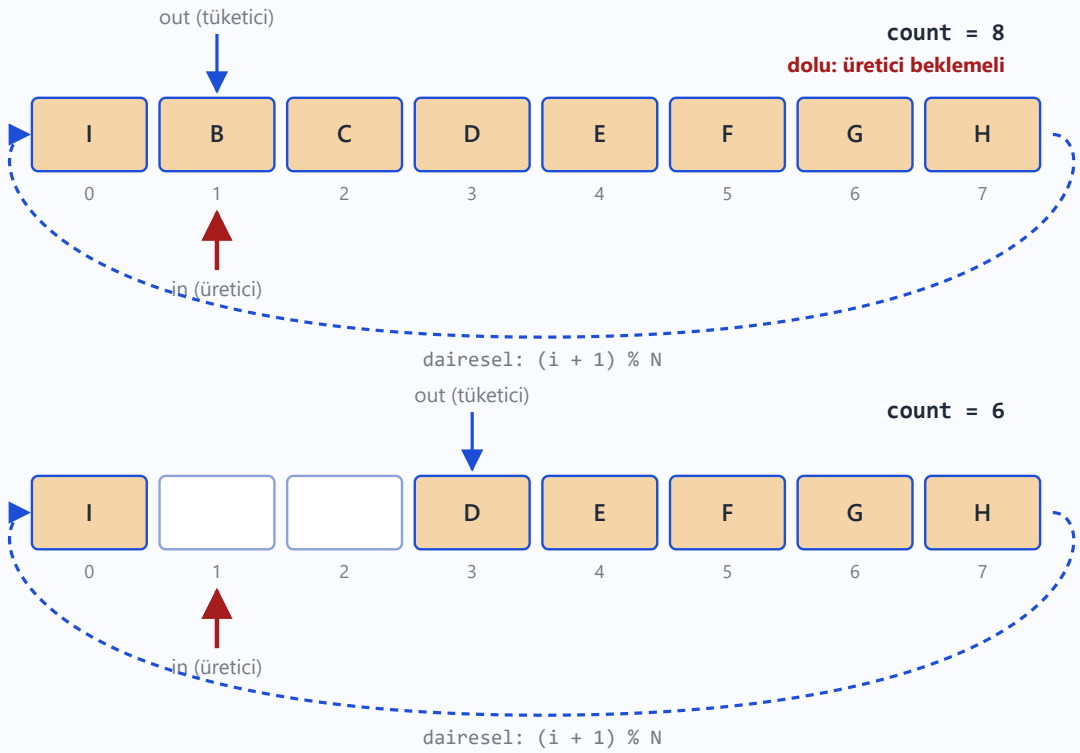
> **Sık yapılan hata**
>
> Turnikeden geçtikten sonraki `up(&turnike)` satırını gereksiz sanmak. Son gelen iş parçacığının `up`'ı yalnız tek bir uyuyanı uyandırır. Geçen her iş parçacığı arkasındakini geçirmezse turnikeden yalnız biri geçer ve öteki ikisi sonsuza dek uyur.

6.8 Klasik problemler

Senkronizasyon araçları, gerçek sistemlerdeki pek çok durumun modeli olan birkaç klasik problemle sınanır. Yeni bir eşgüdüm mekanizması önerildiğinde bu problemleri ne kadar sade çözebildiğine bakılır.

6.8.1 Üretici–tüketici problemi

İki süreç sabit boyutlu (N yuvalı) bir tamponu paylaşır. **Üretici** tampona öğe yazar, **tüketici** tampondan öğe okur. Tampon doluysa üretici, boşsa tüketici beklemelidir. Bu düzene **sınırlı tampon** (*bounded buffer*) denir. Tampon sınırsız olsaydı yalnız tüketici beklerdi. Tampon dairesel bir dizidir: `in` üreticinin yazacağı, `out` tüketicinin okuyacağı yuvayı gösterir ve ikisi de `(i + 1) % N` ile ilerler.



Şekil 6.22: N = 8 yuvalı dairesel tampon. (a) A alındıktan sonra D...I eklenmiş, `in` başa sarmıştır: `count = 8`, tampon doludur ve üretici beklemelidir. (b) Tüketici B ve C'yi alır: `out = 3`, `count = 6`.

Örnekte üretici önce A, B ve C'yi 0, 1 ve 2 numaralı yuvalara koyar; `in = 3` olur. Tüketici A'yı alır: `out = 1`, `count = 2`. Üretici D...I öğelerini 3–7 ve 0 numaralı yuvalara koyar; `in` başa sarıp 1 olur ve `count = 8 = N`'dir. Tüketici B ile C'yi alınca `out = 3`, `count = 6` olur ve üretici yeniden yazabilir.

6.8.2 Ölümcül yarış: kayıp uyandırma

İlk akla gelen çözüm, ortak bir `count` sayacı ile `sleep` ve `wakeup` çağrılarını kullanır. `sleep()` çağıranı bloke eder; `wakeup(p)` uyuyan `p`'yi hazır yapar. Tanenbaum'un kodu şöyledir:

```
#define N 100
int count = 0;
void producer(void) {
    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}
void consumer(void) {
    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }
}
```

Kod masum görünür ama ölümcül bir yarış içerir. Aşağıdaki sırada iki süreç de sonsuza dek uyur:

1. Tampon boştur. Tüketici `count`'u okur: 0. Tam `sleep()` çağıracakken kesilir.
2. Üretici bir öğe üretir ve tampona koyar; `count` 1 olur.
3. `count == 1` olduğu için üretici tüketicinin uyuduğunu varsayar ve `wakeup(consumer)` gönderir. Tüketici henüz uyumadığı için sinyal *kaybolur*.
4. Tüketici kaldığı yerden sürer. Eski kararına göre `sleep()` çağırır ve uyur.
5. Üretici tamponu doldurur; tüketici uyuduğu için kimse okumaz. `count == N` olunca üretici de uyur.

Sorun, `count` sınaması ile `sleep` çağrısının bölünmez olmamasıdır. Arada gelen uyandırma, uyuyan kimse olmadığı için iz bırakmadan kaybolur. Semafor bu sorunu çözer: uyandırmayı bir sayaç olarak sakladığı için, önce gelen bir `up` sonraki `down`'u beklemeden geçirir (Tanenbaum MOS Şekil 2-27).

6.8.3 Semaforlarla sınırlı tampon

Doğru çözüm üç semafor kullanır:

```
#define N 100
semaphore mutex = 1;  /* tampona erişim     */
semaphore empty = N;  /* boş yuva sayısı    */
semaphore full  = 0;  /* dolu yuva sayısı   */

void producer(void) {
  while (true) {
    item = produce_item();
    down(&empty);   /* boş yuva bekle   */
    down(&mutex);   /* KB'ye gir        */
    insert_item(item);
    up(&mutex);     /* KB'den çık       */
    up(&full);      /* dolu sayısı + 1  */
  }
}
```

```
void consumer(void) {
  while (true) {
    down(&full);    /* dolu yuva bekle  */
    down(&mutex);
    item = remove_item();
    up(&mutex);
    up(&empty);     /* boş sayısı + 1   */
    consume_item(item);
  }
}
```

- `mutex = 1` tampona erişimi karşılıklı dışlar. 0 ile başlasaydı hiç kimse giremezdi.
- `empty = N` boş yuvaları sayar. Tampon doluyken üretici `down(&empty)` 'de bloke olur.
- `full = 0` dolu yuvaları sayar. Tampon boşken tüketici `down(&full)` 'de bloke olur.

`mutex` dışlama, `empty` ve `full` ise sıralama (senkronizasyon) görevi görür. İki görev ayrı semaforlarla yapılır.

Örnek 6.6

N = 2 ve tampon boştur. Sıra şöyledir: T bir öğe almaya çalışır; Ü A'yı ve B'yi koyar; Ü C için dener; T bir öğe alır; Ü C'yi koyar; T bir öğe daha alır. Semaforlar Tanenbaum anlamındadır: `up` , uyuyan varsa birini uyandırır ve değer değişmez.

Olay	mutex	empty	full	Tampon	Uyuyan
başlangıç	1	2	0	–	–
T: `down(&full)` , full = 0 → uyur	1	2	0	–	T
Ü: A'yı koyar; `up(&full)` T'yi uyandırır	1	1	0	A	–

Olay	mutex	empty	full	Tampon	Uyuyan
Ü: B'yi koyar	1	0	1	A B	–
Ü: C için `down(&empty)` → uyur	1	0	1	A B	Ü
T: A'yı alır; `up(&empty)` Ü'yü uyandırır	1	0	1	B	–
Ü: C'yi koyar	1	0	2	B C	–
T: B'yi alır	1	1	1	C	–

Tablo 6.15: Semaforlarla sınırlı tampon izlemesi (N = 2).

1. **Tüketici önce çalışır.** `full = 0` olduğu için uyur. Meşgul bekleme yoktur.
2. **A konur.** `up(&full)` uyuyan tüketiciyi uyandırır. Uyanan süreç bu "1"i kullandığı için `full` 0'da kalır.
3. **B konur.** Tüketici henüz çalışmamıştır; `empty = 0`, `full = 1` olur.
4. **C için yer yok.** Üretici uyur; tampon taşmaz.
5. **A alınır.** Tüketicinin `up(&empty)` 'si uyuyan üreticiyi uyandırır; `empty` 0'da kalır.
6. **Son durum.** C konur ve B alınır. Tamponda C vardır: `mutex = 1`, `empty = 1`, `full = 1`.

Hiçbir işlem yürümezken her zaman `empty + full = N` olur; bir işlemin ortasındayken aradaki fark "yoldaki" öğedir.

Dört öğeli bir senaryonun bütün ara yerleştirmeleri bir programla gezildi: öğeler hep A, B, C, D sırasıyla alındı, tampon hiç taşmadı ve kilitlenme olmadı.

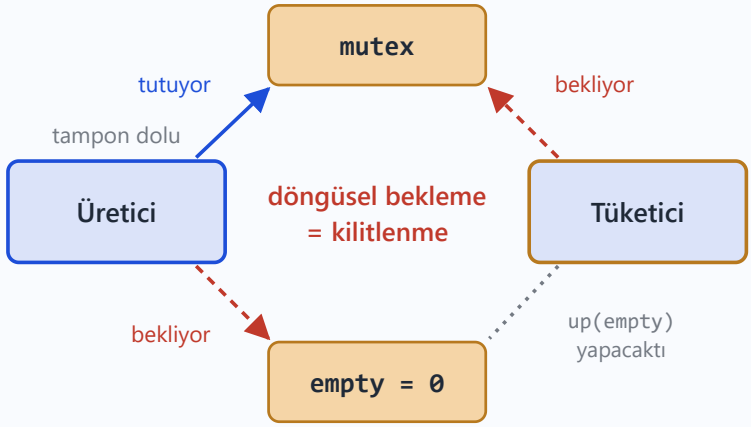
6.8.4 down sırası değişirse

Çözümün doğruluğu iki `down` 'ın sırasına bağlıdır. Üreticide sıra ters çevrilirse ne olur?

```
void producer(void) {
  while (true) {
    item = produce_item();
    down(&mutex);  /* ← önce */
    down(&empty);  /* ← sonra */
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```

1. Tampon tamamen doludur: `empty = 0`.
2. Üretici önce `mutex` 'i alır, sonra `down(&empty)` 'de uyur. Mutex hâlâ elindedir.
3. Tüketici `down(&full)` 'ü geçer ama `down(&mutex)` 'te uyur. Oysa `up(&empty)` yapacak olan oydu.

4. Herkes ötekini bekler: **kilitlenme**.



Şekil 6.23: Ters sıralı `down`'ın yarattığı döngüsel bekleme. Üretici mutex'i tutup `empty`'yi, tüketici `full`'ü geçmiş olarak mutex'i bekler.

Programla yapılan durum uzayı taramasında da kilitlenen bir durum bulunur: `mutex = 0`, `empty = 0` ve tamponda iki öğe. Kural şudur: önce sayma semaforu, sonra mutex beklenir; mutex tutulurken uyunmaz. `up`'ların sırası ise önemsizdir, çünkü `up` hiçbir zaman bloke olmaz.

6.8.5 Okur-yazar problemi

Bir veritabanını birçok süreç paylaşır. Okurlar veriyi yalnız okur, yazarlar değiştirir. Okurlar birbirini dışlamak zorunda değildir: on okur aynı anda okuyabilir. Ama bir yazar yazarken ne başka bir yazar ne de bir okur içeride olmalıdır. Tanenbaum'un çözümü iki semafor ve bir sayaç kullanır:

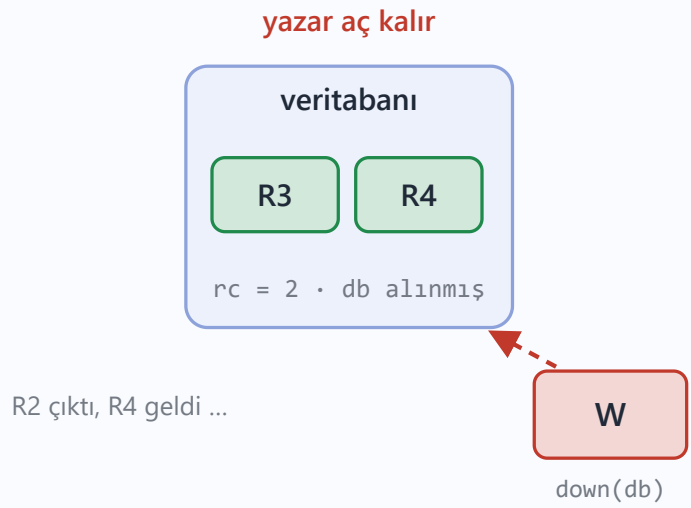
```
semaphore mutex = 1;  /* rc'yi korur          */
semaphore db = 1;     /* veritabanını korur */
int rc = 0;           /* okuyan sayısı       */

void writer(void) {
  while (true) {
    think_up_data();
    down(&db);  write_data_base();  up(&db);
  }
}
void reader(void) {
  while (true) {
    down(&mutex);
    if (++rc == 1) down(&db);  /* ilk okur  */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* son okur  */
    up(&mutex);
    use_data_read();
  }
}
```

`rc` içerideki okur sayısıdır ve `mutex` onu korur. `db` veritabanını korur. Yazar doğrudan `db`'yi alır. Okurlardan yalnız *ilk* giren `db`'yi alır (`++rc == 1`) ve yalnız *son* çıkan bırakır (`--rc == 0`). Aradaki okurlar `db`'ye hiç dokunmadan girip çıkar.

Çözüm dışlamayı doğru sağlar. İki okur ve bir yazarın bütün ara yerleştirmeleri gezildiğinde yazar ile bir okurun birlikte içeride olduğu bir durum çıkmadı; iki okur ise birlikte okuyabildi. Ama çözümün ciddi bir zayıflığı vardır (Şekil 6.24):

1. R1 girer ve `db`'yi alır.
2. Bir yazar gelir ve `down(&db)`'de bekler.
3. Yeni okur R2 `db`'ye dokunmadan girer: `rc = 2`.
4. R1 çıkarken R3 gelir. `rc` 1'e inip yeniden 2 olur; hiç 0 olmaz.
5. Okur akışı örtüşerek sürdükçe `up(&db)` hiç çağrılmaz ve yazar süresiz bekler.



Şekil 6.24: Okur öncelikli çözümde yazar açlığı. Okurlar birbirinin ardından geldikçe `rc` hiç 0 olmaz ve yazar `down(db)`'de bekler.

Bu çözüm **okur önceliklidir** ve yazar açlığına açıktır. Problemin varyantları farklı öncelikleri seçer (Tablo 6.16). Downey'nin açlıksız çözümünde gelen herkes önce bir turnikeden geçer. Bekleyen yazar turnikeyi tutar ve arkasından yeni okur giremez:

```
/* Downey: açlıksız çözüm */
/* yazar */
wait(turnike);
wait(odaBos);
  yaz();
signal(turnike);
signal(odaBos);

/* okur */
wait(turnike); signal(turnike);
/* ilk okur odaBos'u alır,
   son okur bırakır (rc ile) */
  oku();
```

Varyant	Davranış
1. okur-yazar (okur öncelikli)	Yazar ancak hiç okur yokken girer; yazarlar aç kalabilir.
2. okur-yazar (yazar öncelikli)	Bekleyen yazar varsa yeni okur giremez; okurlar aç kalabilir.
Adil (turnike)	Gelen herkes önce aynı turnikeden geçer; yazar beklerken turnikeyi tutar.
`pthread_rwlock_t`	glibc'nin varsayılanı okur önceliklidir; yazar önceliği ayarlanabilir.
Java `ReentrantReadWriteLock`	`new ReentrantReadWriteLock(true)` adil, FIFO'ya yakın sıralama verir.
Linux seqlock, RCU	Okurların hiç kilit almadığı, okuma ağırlıklı çekirdek yapıları.

Tablo 6.16: Okur-yazar varyantları ve gerçek sistemlerdeki karşılıkları.

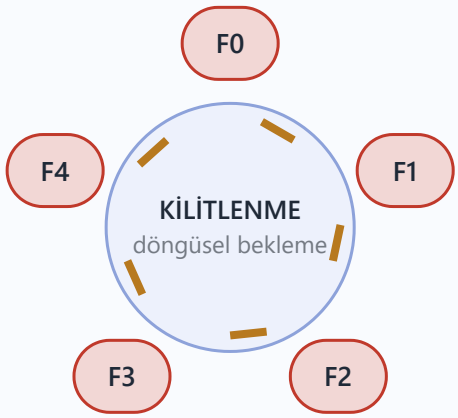
6.8.6 Yemek yiyen filozoflar

Dijkstra 1965'te şu problemi ortaya koydu. Yuvarlak bir masada beş filozof oturur; aralarında beş çatal vardır. Her filozof düşünür, acıkınca sağındaki ve solundaki iki çatalı alıp yer. Problem, sınırlı kaynakların süreçler arasında kilitlenmesiz ve açlıksız paylaştırılmasının modelidir. Naif çözüm her çatalı bir semaforla korur:

```
semaphore catal[5];        /* hepsi 1 */

void filozof(int i) {
  while (true) {
    dusun();
    wait(&catal[i]);             /* sol */
    wait(&catal[(i + 1) % 5]);   /* sağ */
    ye();
    signal(&catal[(i + 1) % 5]);
    signal(&catal[i]);
  }
}
```

Hepsi aynı anda acıkır ve sol çatalını alırsa her filozof sağ çatalı bekler ve kimse bırakmaz: kilitlenme (Şekil 6.25). Her çatal semaforla korunsa bile çözüm yanlıştır. Beş filozofun bütün ara yerleştirmeleri gezildiğinde tam bu durum bulunur: beşi de ilk `wait`'i geçmiş, bütün çatal semaforları 0'dır.



Şekil 6.25: Naif çözümde kilitlenme. Her filozof sol çatalını tutar ve sağ çatalı bekler; döngüsel bekleme oluşur.

Dört çözüm yolu vardır:

- **Masaya en fazla dört.** Başlangıcı 4 olan bir oda semaforu, aynı anda en fazla dört filozofun çatal almaya çalışmasına izin verir. Beş çatal dört kişiye dağıldığında en az biri iki çatala ulaşır ve döngüsel bekleme kırılır.

- **Kaynak sıralaması (asimetri).** Her filozof önce numarası küçük çatalı alır. F4'ün çatalları 4 ve 0'dır; F4 önce 0'ı ister. Genel kural: kilitler hep aynı global sırayla alınır.
- **İki çatalı birlikte almak.** Tanenbaum'un çözümü bir durum dizisi ve filozof başına bir semafor kullanır. Filozof ancak iki komşusu da yemiyorsa yemeye geçer.
- **Monitör.** Silberschatz'ın çözümü `pickup(i)` ve `putdown(i)` prosedürleri ile filozof başına bir koşul değişkeni (`self[i]`) kullanır; mantığı Tanenbaum'unkiyle aynıdır.

Tanenbaum'un çözümü aşağıdadır. Her filozofun durumu `THINKING`, `HUNGRY` ya da `EATING` olabilir. `s[i]` semaforları 0 ile başlar; `LEFT` ve `RIGHT` komşuların numaralarıdır.

```
void take_forks(int i) {
  down(&mutex);
  state[i] = HUNGRY;
  test(i);
  up(&mutex);
  down(&s[i]);    /* çatal yoksa bloke */
}
void put_forks(int i) {
  down(&mutex);
  state[i] = THINKING;
  test(LEFT);     /* (i+4) % 5 */
  test(RIGHT);    /* (i+1) % 5 */
  up(&mutex);
}
void test(int i) {
  if (state[i] == HUNGRY && state[LEFT] != EATING
                         && state[RIGHT] != EATING) {
    state[i] = EATING;
    up(&s[i]);
  }
}
```

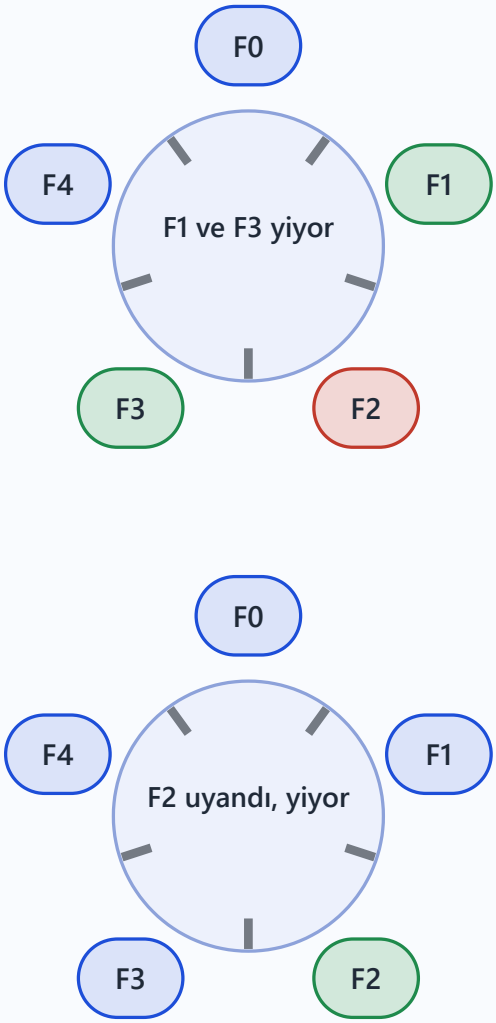
`take_forks` durumu `HUNGRY` yapar ve `test(i)` çağırır. Komşular yemiyorsa `test` durumu `EATING` yapar ve `up(&s[i])` ile filozofa izin verir; sonraki `down(&s[i])` hemen geçer. Komşulardan biri yiyorsa izin verilmez ve filozof `down(&s[i])` 'de uyur. `put_forks` durumu `THINKING` yapar ve iki komşuyu sınar: bekleyen bir komşu artık yiyebiliyorsa onu uyandırır.

Örnek 6.7

Başta herkes düşünür (durum dizisi TTTTT; T düşünüyor, H aç, E yiyor). Sırayla F1, F2 ve F3 acıkır; sonra F1, ardından F3 yemeyi bitirir.

1. **F1 acıkır.** `test(1)` : F0 ve F2 yemiyor → F1 `EATING` olur, `up(&s[1])` ; `down(&s[1])` hemen geçer. Durum: TETTT.
2. **F2 acıkır.** Sol komşu F1 yiyor → test başarısız; F2 `down(&s[2])` 'de uyur. Durum: TEHTT.

3. **F3 acıkır.** F2 aç ama yemiyor, F4 düşünüyor → F3 yer. Durum: TEHET. Aynı anda iki filozof yiyebilir.
4. **F1 bırakır.** `test(0)` ve `test(2)` yapılır. `test(2)` başarısızdır, çünkü sağdaki F3 hâlâ yiyor. Durum: TTHET.
5. **F3 bırakır.** `test(2)` bu kez başarılıdır: F2 `EATING` olur ve `up(&s[2])` onu uyandırır. Durum: TTETT.



Şekil 6.26: Tanenbaum çözümünden iki an. (a) F1 ve F3 yer, F2 bloke bekler. (b) F3 bırakınca F2 uyanır ve yer (Tanenbaum MOS Şekil 2-47).

Beş filozofun bütün ara yerleştirmeleri bir programla gezildi. Hiçbir yürütmede kilitlenme olmadı, iki komşu hiçbir zaman birlikte yemedi ve aynı anda en fazla iki filozof yedi. İki sayısı rastgele değildir: yiyen her filozof iki çatal tutar ve yan yana iki filozof aynı anda yiyemez. n filozoflu bir masada aynı anda en fazla $\lfloor n/2 \rfloor$ filozof yiyebilir; beş için 2, yedi için 3'tür.

Sık yapılan hata

Kilitlenmesiz bir çözümün açlıksız da olduğunu sanmak. Tanenbaum'un çözümünde kilitlenme yoktur. Ama F1 ve F3 sırayla yemeye devam ederse, ikisinden biri her zaman yiyor olur ve F2'nin `test` 'i hiç başarılı olmaz. F2 teorik olarak sonsuza dek aç kalabilir. Açlığı önlemek için ek bir kural gerekir: bir sıra numarası, bekleme süresine göre öncelik artırma (yaşlandırma) gibi.

6.9 Monitörler

Semaforla yazılan kodun doğruluğu her `wait` / `signal` çiftine bağlıdır ve iki `down` 'ın yerini değiştirmek bile kilitlenme yaratır. Monitör, bu hata sınıfını programlama dili düzeyinde azaltmak için geliştirildi.

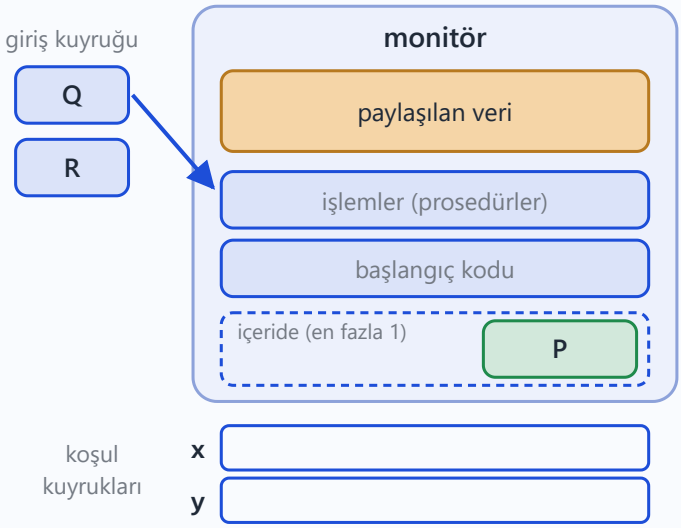
6.9.1 Monitör

Tanım 6.11

Monitör (*monitor*), paylaşılan veriyi, o veri üzerinde çalışan prosedürleri ve başlangıç kodunu bir arada tutan, dil düzeyinde bir soyut veri tipidir. Aynı anda monitörün içinde yalnız bir süreç etkin olabilir (Hoare, 1974; Brinch Hansen, 1975).

```
monitor ornek
  integer i;
  condition c;
  procedure uretici(); … end;
  procedure tuketici(); … end;
end monitor;
```

Süreçler monitörün prosedürlerini çağırabilir ama iç verisine doğrudan erişemez. Karşılıklı dışlamayı programcı değil derleyici sağlar: her prosedürün başına ve sonuna giriş kilidini derleyici ekler. Programcı kilidi almayı ya da bırakmayı unutamaz. Monitör doluysa çağıran süreç bir **giriş kuyruğunda** bekler (Şekil 6.27).



Şekil 6.27: Monitörün yapısı. İçeride en fazla bir süreç (P) etkindir; Q ve R giriş kuyruğunda bekler. `x` ve `y` koşul değişkenlerinin kendi kuyrukları vardır.

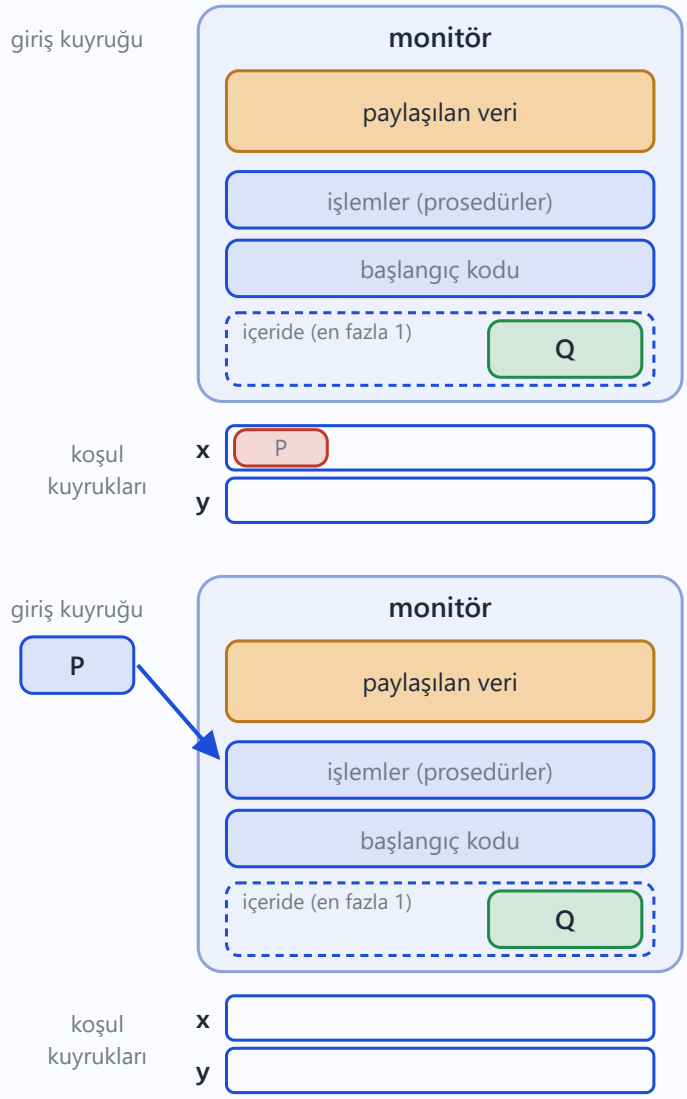
6.9.2 Koşul değişkenleri

Dışlama yetmez; monitörün içindeki süreç bazen bir koşulun sağlanmasını beklemek zorundadır. Örneğin tüketici tamponu boş bulmuştur. Monitörü tutarak beklerse üretici hiç giremez. Bu yüzden monitörler **koşul değişkenleri** sunar.

Tanım 6.12

Koşul değişkeni (*condition variable*), bir koşulun gerçekleşmesini beklemek için kullanılan bir bekleme kuyruğudur. `x.wait()` çağıranı askıya alır ve monitörü serbest bırakır. `x.signal()`, `x`'te bekleyen bir süreci (varsa) devam ettirir. `broadcast` ya da `notifyAll` bekleyenlerin hepsini uyandırır.

1. P monitördedir; tampon boş olduğu için devam edemez. Q giriş kuyruğundadır.
2. P `x.wait()` çağırır: monitörü bırakır ve `x` kuyruğunda uyur. Böylece Q girebilir.
3. Q koşulu sağlar ve `x.signal()` çağırır; `x` kuyruğundan bir süreç uyanır. Kimin hemen çalışacağı ise anlamsal bir seçimdir.
4. `x` kuyruğunda kimse yokken `x.signal()` etkisizdir ve hiçbir iz bırakmaz.



Şekil 6.28: Koşul değişkeni. (a) P `x.wait()` ile monitörü bırakır ve `x` kuyruğunda uyur; Q içeri girer. (b) Q `x.signal()` çağırır ve P uyanır.

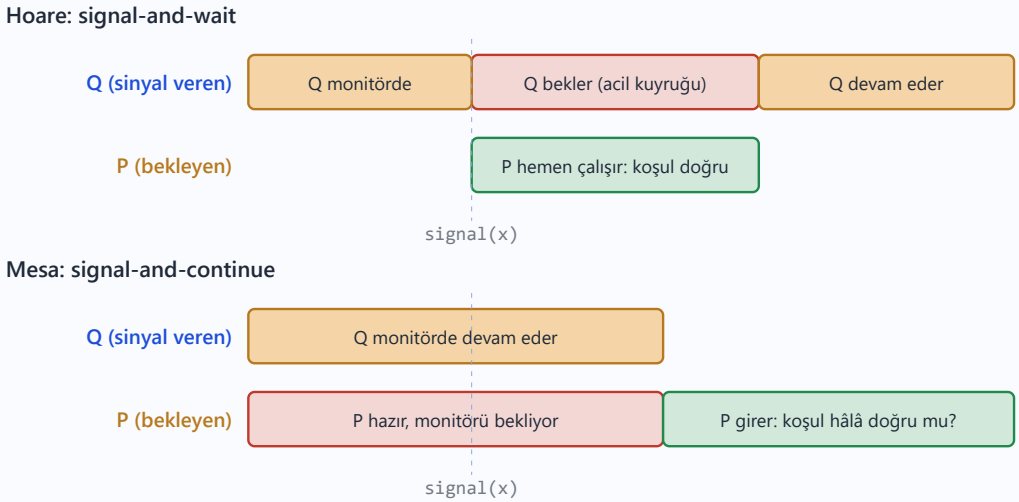
Son adım, koşul değişkeni ile semafor arasındaki temel farktır. Semaforun `signal`'ı değeri artırır ve sonraki `wait` bu artışı kullanır; sinyal hatırlanır. Koşul değişkeninin ise değeri yoktur; yalnız bir kuyruktur. Bekleyen yoksa sinyal kaybolur. Bu yüzden beklenen koşul (örneğin `count > 0`) her zaman paylaşılan veride ayrıca tutulur ve beklemeden önce sınanır.

6.9.3 Hoare ve Mesa anlamı

Q, `x.signal()` ile P'yi uyandırdığında monitörde iki süreç çalışmaya hazırdır. Ama içeride yalnız biri etkin olabilir. Hangisinin süreceği monitörün **anlamını** belirler:

- **Signal-and-wait (Hoare).** Uyanan P hemen çalışır; Q bir "acil" kuyrukta bekler. P, uyandığı anda koşulun doğru olduğundan emindir: beklemeyi bir `if` ile yazmak yeter.

- **Signal-and-continue (Mesa, 1980).** Q devam eder; P yalnız "hazır" olur ve monitörü yeniden almak için bekler. P çalışana kadar başka bir süreç koşulu bozabilir. P uyandığında koşulu yeniden sınamak zorundadır.



Şekil 6.29: (a) Hoare anlamı: `signal`'dan sonra P hemen çalışır ve koşul doğrudur; Q acil kuyrukta bekler. (b) Mesa anlamı: Q monitörde devam eder; P hazırdır ve girdiğinde koşulun hâlâ doğru olup olmadığını sınamalıdır.

Anlam	Kim hemen çalışır?	Kullananlar
Signal-and-wait (Hoare)	Uyandırılan	Ders kitabı monitörleri
Signal-and-exit (Brinch Hansen)	Uyandırılan; `signal` prosedürün son komutu olmalı	Concurrent Pascal
Signal-and-continue (Mesa)	Sinyal veren	Pthreads, Java, C#, Go `sync.Cond`, Linux çekirdeği

Tablo 6.17: Monitör anlamları.

Gerçek sistemlerin neredeyse hepsi Mesa anlamını kullanır. Mesa anlamında ise `signal` yalnız bir "ipucudur": koşul belki doğrudur, uyanan bunu kendisi sınar.

6.9.4 Neden if değil while?

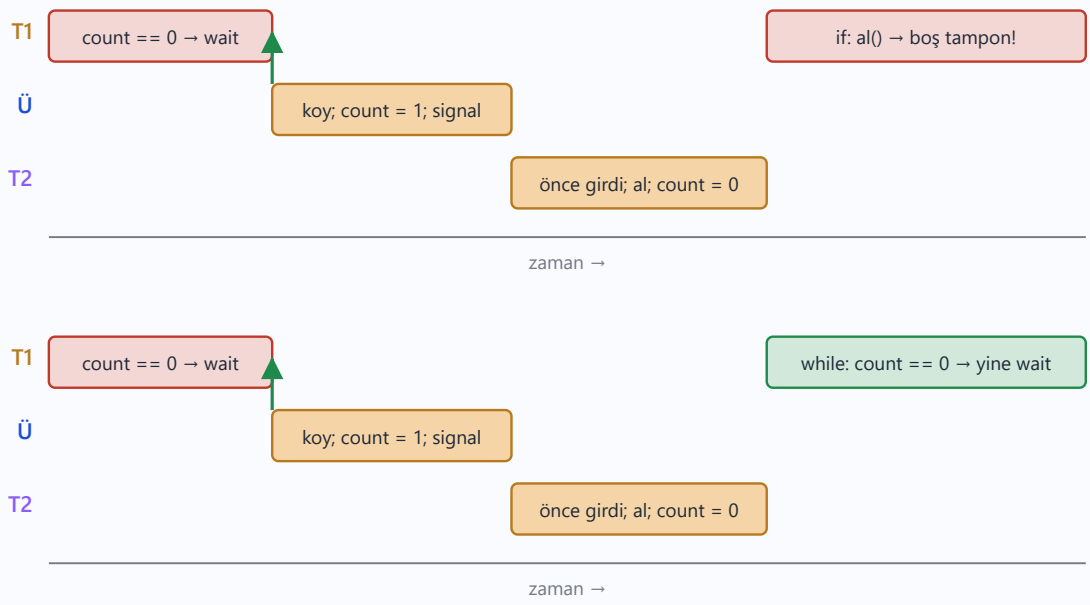
Mesa anlamında beklemenin `if` ile yazılması bir hatadır:

```
/* YANLIŞ (Mesa'da) */
lock(&m);
if (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* boş olabilir */
unlock(&m);
```

```
/* DOĞRU */
lock(&m);
while (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* count > 0 kesin */
unlock(&m);
```

Hatanın oluşma yolu şöyledir (Şekil 6.30):

1. Tampon boştur. Tüketici T1 `dolu` koşul değişkeninde bekler.
2. Üretici bir öğe koyar, `count = 1` yapar ve `signal` çağırır. T1 uyanır ama önce kilidi geri alması gerekir; yalnız hazır olur.
3. Bu arada yeni gelen tüketici T2 kilidi önce kapar (*barging*) ve öğeyi alır: `count = 0`.
4. T1 sonunda kilidi alır. `if` ile yazıldıysa koşulu yeniden sınamadan boş tampondan okur: hata.
5. `while` ile yazıldıysa T1 koşulu yeniden sınar, `count == 0` görür ve tekrar bekler.



Şekil 6.30: (a) `if` ile bekleyen T1, T2 öğeyi aldıktan sonra boş tampondan okur. (b) `while` ile T1 koşulu yeniden sınar ve yeniden bekler.

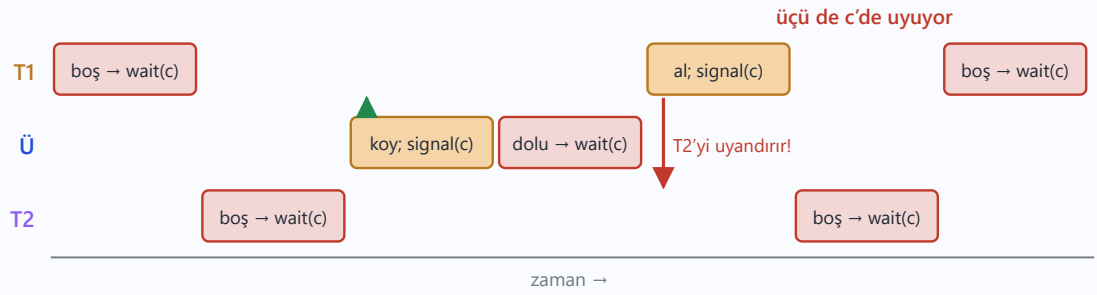
Mesa anlamlı bir monitörün modeliyle bütün ara yerleştirmeler gezildiğinde, `if`'li sürümde boş tampondan okunan bir yürütme bulunur. `while`'lı sürümde böyle bir yürütme yoktur. POSIX ayrıca **sahte uyanmalara** (*spurious wakeup*) izin verir: `pthread_cond_wait`, hiç `signal` gelmeden de dönebilir. Bu yüzden `while` her durumda zorunludur.

6.9.5 Tek koşul değişkeni yetmez

`while` gerekli ama yeterli değildir. Arpaci-Dusseau'ların örneğinde tampon tek yuvalıdır; iki tüketici (T1, T2) ve bir üretici (Ü) vardır. Hepsi aynı `c` koşul değişkenini kullanır:

```
/* YANLIŞ: tek koşul c */
T: while (count == 0) wait(&c, &m);
   x = al();  signal(&c);
Ü: while (count == 1) wait(&c, &m);
   koy(x);    signal(&c);
```

1. Tampon boştur; T1 ve T2 `c`'de uyur.
2. Ü bir öğe koyar ve `signal(c)` ile T1'i uyandırır.
3. Ü yeniden dener; tampon dolu olduğu için o da `c`'de uyur. Artık `c`'de T2 ve Ü bekler.
4. T1 öğeyi alır ve `signal(c)` yapar. Hangi bekleyenin uyanacağı belli değildir; bu kez T2 uyanır.
5. T2 tamponu boş bulur ve yeniden uyur. T1 de bir sonraki turda uyur. Üreticiyi uyandıracak kimse kalmamıştır: program durur.



Şekil 6.31: Tek koşul değişkeniyle kilitlenme. T1'in sinyali üreticiyi değil T2'yi uyandırır; sonunda üçü de `c`'de uyur.

Hata, tüketicilerle üreticinin aynı kuyrukta beklemesidir. Tüketici yalnız üreticiyi, üretici yalnız tüketiciyi uyandırmalıdır. Çözüm iki koşul değişkenidir:

```
/* DOĞRU: iki koşul */
T: while (count == 0) wait(&dolu, &m);
   x = al();  signal(&bos);
Ü: while (count == 1) wait(&bos, &m);
   koy(x);    signal(&dolu);
```

Modelle yapılan taramada tek koşullu sürümde, `while`'a rağmen üç iş parçacığının da uyuduğu bir durum erişilebilir. İki koşullu sürümde böyle bir durum yoktur.

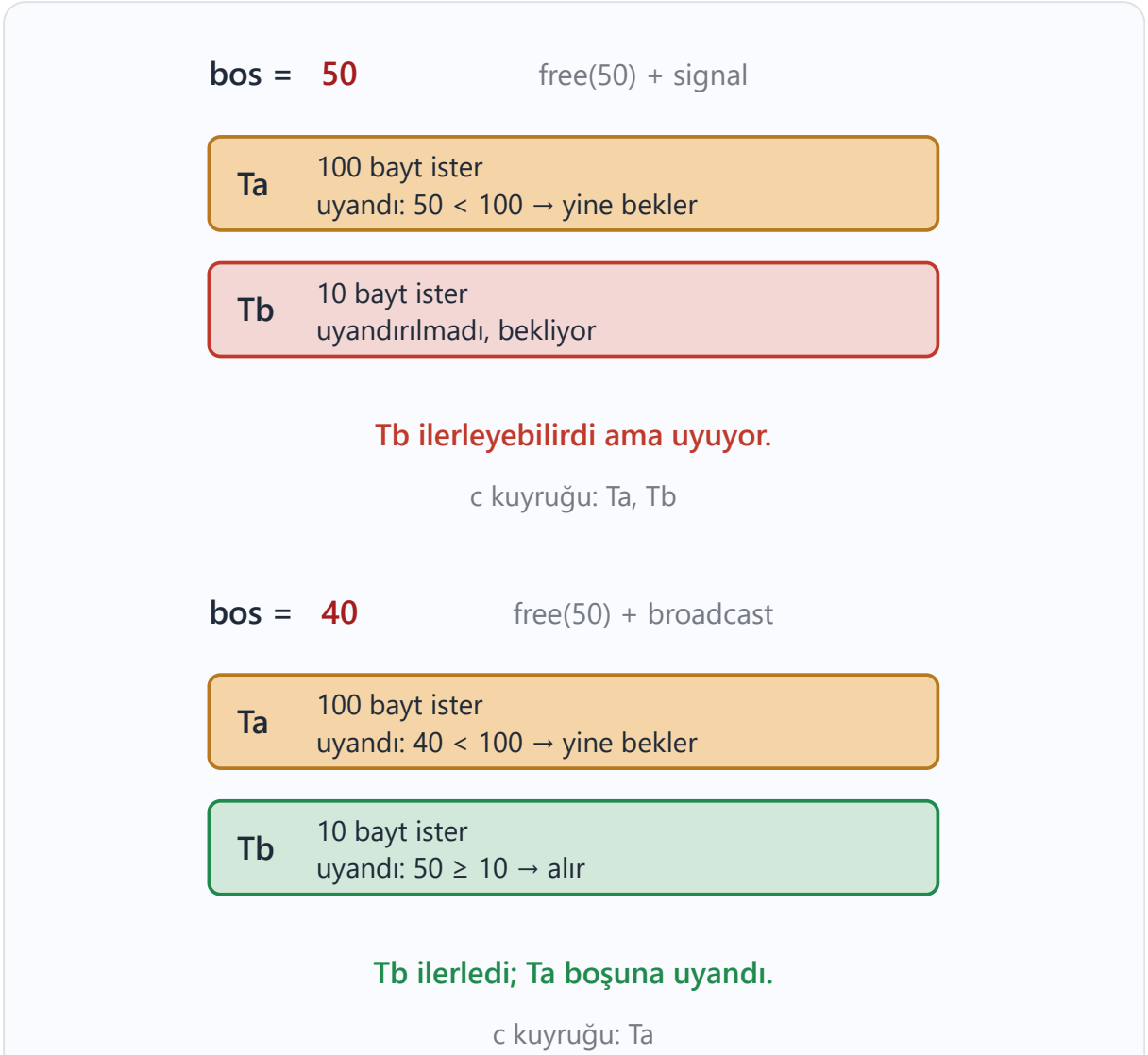
6.9.6 signal ve broadcast

Bazen hangi bekleyenin ilerleyebileceği bilinemez. Bir bellek ayırıcıda iş parçacıkları farklı miktarlarda bellek ister:

```
int bos = 0;            /* boş bayt */

void bellek_al(int n) {
  lock(&m);
  while (bos < n)
    wait(&c, &m);
  bos -= n;  /* ... ayır */
  unlock(&m);
}
void bellek_ver(int n) {
  lock(&m);
  bos += n;
  broadcast(&c); /* signal yetmez */
  unlock(&m);
}
```

Bellek tükenmiştir. Ta 100, Tb 10 bayt ister; ikisi de `while` döngüsünde uyur. Tc 50 bayt bırakır. `signal` yalnız birini uyandırır. Bu Ta olursa, 50 < 100 olduğu için yine uyur; Tb ilerleyebilecekken uyumaya devam eder. `broadcast` ise herkesi uyandırır. Her biri koşulunu yeniden sınar: Tb 10 baytı alır ve `bos` 40 olur; Ta 40 < 100 görüp yeniden uyur.



Şekil 6.32: (a) `signal` yalnız Ta'yı uyandırır; Ta yine bekler ve Tb'nin hakkı boşa gider. (b) `broadcast` ile Tb ilerler, Ta boşuna uyanmıştır.

Kural şudur: hangi bekleyenin ilerleyebileceği bilinmiyorsa `pthread_cond_broadcast` (Java'da `notifyAll`) kullanılır. Bu kalıba **kapsayan koşul** (*covering condition*) denir. Bedeli gereksiz uyanmalardır. Bekleyenler ayrı koşullara bölünebiliyorsa `signal` daha ucuzdur. Her iki durumda da bekleme `while` içinde yapılır.

6.9.7 Monitörle üretici–tüketici

Tanenbaum'un Pidgin Pascal ile yazdığı monitör, sınırlı tamponu iki koşul değişkeniyle çözer:

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove := remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;

  count := 0;
end monitor;
```

`count`'a erişim monitör sayesinde zaten karşılıklı dışlanır. Tampon doluysa üretici `wait(full)` ile bloke olur ve monitörü bırakır. Tüketici bir öğe alınca `count = N − 1` olur ve `signal(full)` üreticiyi uyandırır. Üreticiyi bloke eden `wait`'tir; `signal` yalnız uyandırır. Burada `if` yeterlidir, çünkü Tanenbaum Brinch Hansen anlamını varsayar: `signal` prosedürün son komutudur ve sinyal veren hemen çıkar. Pthreads ya da Java'da aynı kod `while` ile yazılmalıdır. Kayıp uyandırma da olmaz: sınama ile `wait` aynı monitör içinde, bölünmeden yapılır.

Örnek 6.8

N = 2 ve tampon boştur. T önce `remove` çağırır; Ü sonra sırayla a, b, c, d öğelerini koyar; T arada iki kez `remove` çağırır.

Olay	count	Tampon	İçeride	full	empty
başlangıç	0	–	–	–	–
T `remove` : count = 0 → `wait(empty)`	0	–	–	–	T
Ü `insert(a)`	1	a	Ü	–	T
count = 1 → `signal(empty)` ; Ü çıkar, T sürer	1	a	T	–	–
T a'yı alır; 0 ≠ N − 1, sinyal yok	0	–	–	–	–
Ü `insert(b)` ; `signal(empty)` boşa gider	1	b	–	–	–
Ü `insert(c)` ; tampon doldu	2	b c	–	–	–
Ü `insert(d)` : count = N → `wait(full)`	2	b c	–	Ü	–

Olay	count	Tampon	İçeride	full	empty
T b'yi alır; count = 1 = N − 1 → `signal(full)`	1	c	Ü	–	–
Ü `wait` 'ten döner, d'yi koyar	2	c d	–	–	–

Tablo 6.18: Monitörle üretici–tüketici izlemesi (N = 2, Brinch Hansen anlamı).

b konduğunda `count = 1` olur ve `signal(empty)` çağrılır; ama bekleyen yoktur ve sinyal kaybolur. Bu kayıp zararsızdır, çünkü bekleyen biri olsaydı zaten uyandırılacaktı. Sonunda ne veri kaybolur ne de bir süreç sonsuza dek uyur.

6.10 POSIX ve Java

Ders kitabı monitörleri bir dil özelliğidir ve C'de yoktur. Gerçek programlarda aynı yapı bir mutex ile koşul değişkenlerinin birlikte kullanılmasıyla kurulur. Bu kesim POSIX iş parçacıkları (Pthreads) ve Java'daki araçları özetler.

6.10.1 Pthreads: mutex ve koşul değişkeni

Mutex çağrısı	Görevi	Koşul çağrısı	Görevi
`pthread_mutex_init`	mutex oluşturur	`pthread_cond_init`	koşul değişkeni oluşturur
`pthread_mutex_destroy`	mutex'i yok eder	`pthread_cond_destroy`	koşul değişkenini yok eder
`pthread_mutex_lock`	kilidi alır ya da bloke olur	`pthread_cond_wait`	mutex'i atomik olarak bırakıp uyur; dönerken yeniden alır
`pthread_mutex_trylock`	alır ya da hemen `EBUSY` döner	`pthread_cond_signal`	bekleyenlerden en az birini uyandırır
`pthread_mutex_unlock`	kilidi bırakır (yalnız sahibi)	`pthread_cond_broadcast`	bekleyenlerin hepsini uyandırır

Tablo 6.19: Pthreads mutex ve koşul değişkeni çağrıları.

Kural: `pthread_cond_wait` her zaman mutex tutulurken ve bir `while` döngüsü içinde çağrılır. Mutex'in atomik olarak bırakılıp uyunması, kayıp uyandırma penceresini kapatır. Statik başlatma için `PTHREAD_MUTEX_INITIALIZER` ve `PTHREAD_COND_INITIALIZER` vardır. Mutex türleri `NORMAL`, `ERRORCHECK` ve `RECURSIVE` 'dir; öncelik kalıtımı `PTHREAD_PRIO_INHERIT` özniteliğiyle açılır. Ek arayüzler arasında `pthread_rwlock_*`, `pthread_spin_*`, `pthread_barrier_*` ve C11'in `mtx_t` / `cnd_t` türleri bulunur.

Tanenbaum ve Bos'un Pthreads ile yazdığı üretici–tüketici programında tampon tek bir tam sayıdır (0 boş demektir). Üretici 1 ve 2 değerlerini koyar, tüketici onları yazdırır:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
               condp = PTHREAD_COND_INITIALIZER;
int buffer = 0;                    /* 0 = boş */

void *producer(void *p) {
  for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
    pthread_mutex_lock(&m);
    while (buffer != 0) pthread_cond_wait(&condp, &m);
    buffer = i;
    pthread_cond_signal(&condc);
    pthread_mutex_unlock(&m);
  } return NULL;
}
void *consumer(void *p) {
  for (int i = 1; i <= MAX; i++) {
    pthread_mutex_lock(&m);
    while (buffer == 0) pthread_cond_wait(&condc, &m);
    printf("%d\n", buffer); buffer = 0;
    pthread_cond_signal(&condp);
    pthread_mutex_unlock(&m);
  } return NULL;
}
```

1. Tüketici önce çalışır ve mutex'i alır. Tampon boştur; `cond_wait` mutex'i atomik olarak bırakır ve uyur.
2. Üretici mutex'i alır. Tampon boş olduğu için döngüyü atlar ve 1 yazar. `signal` tüketiciyi uyandırır; ama tüketici mutex'i geri almadan dönemez (Mesa).
3. Üretici yarışı kazanıp mutex'i yeniden almış olabilir. `i = 2` için tamponu dolu bulur, `condp`'de bekler ve mutex'i bırakır.
4. Tüketici mutex'i alır, `while` koşulunu yeniden sınar ($1 \neq 0$), 1'i yazdırır ve tamponu boşaltır.
5. Tüketici `condp`'ye sinyal verir. Üretici uyanır, tamponu boş bulur ve 2 yazar. İkinci turda tüketici 2'yi yazdırır.

Mutex ve koşul değişkeninin Mesa anlamlı modeliyle bütün ara yerleştirmeler gezildi. Her yürütmede çıktı “1, 2” oldu ve hiçbiri kilitlenmedi. Sıra değişse de çıktı değişmez; değişen yalnız kimin ne zaman uyuduğudur.

6.10.2 POSIX semaforları

```
class="kw">#include <semaphore.h>

/* adsız: aynı sürecin iş parçacıkları */
sem_t bos, dolu;
sem_init(&bos,  0, N);   /* 0: süreç içi */
sem_init(&dolu, 0, 0);
sem_wait(&bos);   /* … koy … */   sem_post(&dolu);
sem_destroy(&bos);

/* adlandırılmış: ilgisiz süreçler arası */
sem_t *s = sem_open("/os06", O_CREAT, 0644, 1);
sem_wait(s);  /* KB */  sem_post(s);
sem_close(s);
sem_unlink("/os06");
```

POSIX iki tür semafor sunar. **Adsız** semafor `sem_init` ile bir değişken olarak oluşturulur. İkinci argüman 0 ise semafor aynı sürecin iş parçacıkları arasında kullanılır. 1 ise süreçler arasında paylaşılır; o zaman semafor paylaşımlı bellekte durmalıdır. **Adlandırılmış** semafor `sem_open` ile bir adla açılır ve ilgisiz süreçler aynı adı kullanarak ona erişir. `sem_wait` P, `sem_post` V işlemidir. `sem_post` bir sinyal işleyicisinden de güvenle çağrılabilir. `sem_trywait` beklemeden, `sem_timedwait` süre sınırıyla dener; `sem_getvalue` anlık değeri yalnız bilgi amacıyla verir.

macOS adsız semaforları desteklemez; orada `sem_init` kullanımdan kaldırılmıştır. Adlandırılmış semafor ya da `dispatch_semaphore` kullanılır. Java'nın karşılığı `java.util.concurrent.Semaphore`'dur: `acquire()`, `release()` ve isteğe bağlı adil mod.

6.10.3 Java: synchronized, wait ve notify

Java'da her nesne bir monitördür: bir **içsel kilidi** ve tek bir **bekleme kümesi** (*wait set*) vardır. `synchronized` metot ya da blok bu kilidi alır ve çıkışta, istisna fırlatılsa bile, bırakır. Kilit yeniden girilebilir: sahibi aynı kilidi ikinci kez alabilir. `wait()` kilidi bırakıp uyur; `notify()` bekleyenlerden birini, `notifyAll()` hepsini uyandırır. Anlam Mesa'dır; bekleme `while` içinde yapılır.

```
public class TekYuva {
  private int yuva = 0;            /* 0 = boş */

  synchronized void koy(int x) throws InterruptedException {
    while (yuva != 0) wait();      /* kilidi bırakır */
    yuva = x;
    notifyAll();
  }
  synchronized int al() throws InterruptedException {
    while (yuva == 0) wait();
    int x = yuva;
    yuva = 0;
    notifyAll();
    return x;
  }
  /* main: üretici 10, 20, 30, 40 koyar;
     main dört kez al() ile okuyup yazdırır */
}
```

Sunumdaki `main` açıklaması çalıştırılabilir bir programa dönüştürüldü:

```
public class TekYuva {
  private int yuva = 0;            /* 0 = boş */

  synchronized void koy(int x) throws InterruptedException {
    while (yuva != 0) wait();      /* kilidi bırakır */
    yuva = x;
    notifyAll();
  }
  synchronized int al() throws InterruptedException {
    while (yuva == 0) wait();
    int x = yuva;
    yuva = 0;
    notifyAll();
    return x;
  }
  public static void main(String[] a) throws InterruptedException {
    TekYuva t = new TekYuva();
    Thread u = new Thread(() -> {
      try { for (int x = 10; x <= 40; x += 10) t.koy(x); } catch (InterruptedException e) {
}
    });
    u.start();
    for (int k = 0; k < 4; k++) System.out.println("alındı " + t.al());
    u.join();
  }
}
```

Program JDK 21 ile her çalıştırmada aynı çıktıyı verdi:

```
alındı 10
alındı 20
alındı 30
alındı 40
```

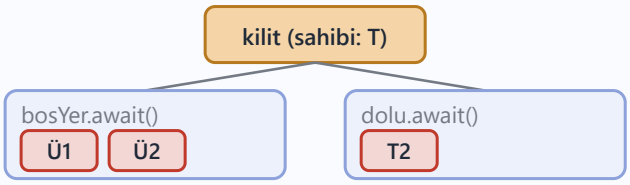
Üretici ve tüketici aynı bekleme kümesini paylaşır. Bu yüzden `notify` yerine `notifyAll` daha güvenlidir: `notify` yanlış türden bir bekleyeni uyandırabilir ve tek koşul değişkeniyle görülen kilitlenme yaşanabilir.

6.10.4 Java: ReentrantLock ve Condition

`java.util.concurrent.locks` paketi daha esnek bir kilit sunar. Bir `ReentrantLock`'a birden çok `Condition` bağlanabilir. Böylece üreticiler ve tüketiciler ayrı kuyruklarda bekler ve `signal` doğru grubu uyandırır. Kilit ayrıca `tryLock()`, süre sınırlı ve kesilebilir kilitleme ile isteğe bağlı adil mod sunar. `unlock()` mutlaka `finally` içinde çağrılır; yoksa bir istisna kilidi sonsuza dek tutulu bırakır.

```
class Tampon {
  private final int[] buf = new int[2];
  private int bas = 0, son = 0, adet = 0;
  private final ReentrantLock kilit = new ReentrantLock();
  private final Condition bosYer = kilit.newCondition();
  private final Condition dolu   = kilit.newCondition();

  void koy(int x) throws InterruptedException {
    kilit.lock();
    try {
      while (adet == buf.length) bosYer.await();
      buf[son] = x; son = (son + 1) % buf.length; adet++;
      dolu.signal();
    } finally { kilit.unlock(); }
  }
  /* al(): while (adet == 0) dolu.await(); …
           bosYer.signal();                    */
}
/* üretici 1..5 koyar; main 5 kez al() */
```



Şekil 6.33: Bir kilide bağlı iki koşul. Üreticiler (Ü1, Ü2) `bosYer`'de, tüketici T2 `dolu`'da bekler; kilidin sahibi T'dir.

Sunumdaki sınıf `al()` metodu tamamlanarak çalıştırıldı:

```
import java.util.concurrent.locks.*;
class Tampon {
  private final int[] buf = new int[2];
  private int bas = 0, son = 0, adet = 0;
  private final ReentrantLock kilit = new ReentrantLock();
  private final Condition bosYer = kilit.newCondition();
  private final Condition dolu   = kilit.newCondition();

  void koy(int x) throws InterruptedException {
    kilit.lock();
    try {
      while (adet == buf.length) bosYer.await();
      buf[son] = x; son = (son + 1) % buf.length; adet++;
      dolu.signal();
    } finally { kilit.unlock(); }
  }
  int al() throws InterruptedException {
    kilit.lock();
    try {
      while (adet == 0) dolu.await();
      int x = buf[bas]; bas = (bas + 1) % buf.length; adet--;
      bosYer.signal();
      return x;
    } finally { kilit.unlock(); }
  }
  public static void main(String[] a) throws InterruptedException {
    Tampon t = new Tampon();
    Thread u = new Thread(() -> {
      try { for (int x = 1; x <= 5; x++) t.koy(x); } catch (InterruptedException e) { }
    });
    u.start();
    int toplam = 0; StringBuilder sb = new StringBuilder();
    for (int k = 0; k < 5; k++) { int x = t.al(); toplam += x; sb.append(x).append(' '); }
    u.join();
    System.out.println(sb + "| toplam = " + toplam);
  }
}
```

JDK 21 ile üç çalıştırmanın üçü de aynı çıktıyı verdi:

```
1 2 3 4 5 | toplam = 15
1 2 3 4 5 | toplam = 15
1 2 3 4 5 | toplam = 15
```

İki yuvalı tamponda üretici tüketiciden en fazla iki öğe önde olabilir; tampon dolunca `bosYer`'de bekler. Öğeler ise hep konduğu sırayla alınır. İş parçacıklarının ne zaman uyuduğu çalıştırmadan çalıştırmaya değişir, çıktı değişmez. Sanal iş parçacıkları için bir not: Java 21'de sanal bir iş parçacığı `synchronized` içinde bloke olunca onu taşıyan platform iş parçacığını "sabitliyordu" (*pinning*). Bu yüzden o sürümde `ReentrantLock` önerilirdi. JDK 24'te JEP 491 bu sorunu giderdi.

6.11 Canlılık

Doğru bir senkronizasyon önce güvenliği sağlar: veri bozulmaz. Ama sistemin ilerlemesi de gerekir. Bu kesim, dışlama doğru olsa bile ortaya çıkabilen canlılık sorunlarını ele alır.

6.11.1 Öncelik terslemesi

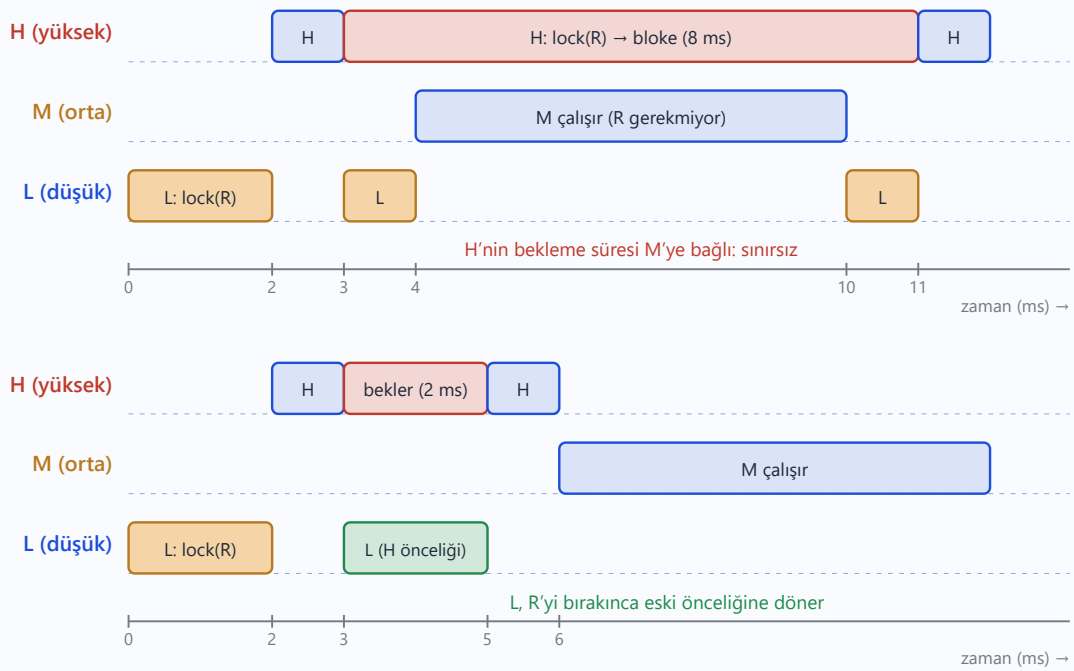
Öncelikli çizelgelemede yüksek öncelikli bir görev hazır olduğu an çalışır. Kilitler bu kuralı beklenmedik biçimde bozabilir. Üç görev düşünelim: yüksek öncelikli H, orta öncelikli M ve düşük öncelikli L. H ile L paylaşılan bir R kaynağını kullanır; M'nin R ile ilgisi yoktur. Tek bir işlemci vardır ve süreler milisaniyedir.

1. **t = 0.** L, R'nin kilidini alır. Kritik bölgesi toplam 4 ms sürecektir.
2. **t = 2.** H gelir ve L'yi keser. t = 3'te R'yi ister ve bloke olur. Kısa bir bekleme beklenir: L kalan 2 ms'yi bitirince H girer.
3. **t = 4.** M gelir. M, L'den önceliklidir; L'yi keser ve t = 10'a kadar çalışır. L ancak t = 10'da yeniden çalışır ve R'yi t = 11'de bırakır.

H, R için 8 ms beklemiştir; bu sürenin 6 ms'si M'nin çalışmasıdır. En yüksek öncelikli görev, kendisiyle hiçbir ilgisi olmayan orta öncelikli bir görevi bekler. M uzadıkça H'nin beklemesi de sınırsız uzar.

> **Tanım 6.13**
>
> **Öncelik terslemesi** (*priority inversion*), yüksek öncelikli bir görevin düşük öncelikli bir görevin tuttuğu kilidi beklemesidir. Bu sırada araya giren orta öncelikli görevler yüzünden yüksek öncelikli görev dolaylı olarak onları da bekler. **Öncelik kalıtımı** (*priority inheritance*) ise kilidi tutan görevin, o kilidi bekleyen en yüksek öncelikli görevin önceliğini geçici olarak almasıdır.

Öncelik terslemesi bir çözüm değil, sorundur. Öncelik kalıtımında H, R'yi beklerken L geçici olarak H'nin önceliğini alır. M gelse de L'yi kesemez. L t = 3'ten t = 5'e kadar çalışıp R'yi bırakır ve eski önceliğine döner. H hemen çalışır ve yalnız 2 ms bekler (Şekil 6.34). Bir başka çözüm **öncelik tavanıdır** (*priority ceiling*): kilidi alan görev, o kilidi kullanabilecek en yüksek öncelikli görevin önceliğine hemen yükseltilir.



Şekil 6.34: (a) Öncelik terslemesi: H, R'yi beklerken M çalışır; H'nin beklemesi M'nin süresine bağlıdır. (b) Öncelik kalıtımı: L, H'nin önceliğiyle çalışır, M araya giremez ve L R'yi bırakınca H hemen çalışır.

Bu iki senaryo, kesmeli öncelikli bir tek işlemci çizelgeleyicisinin küçük bir modeliyle birim birim yeniden çalıştırıldı. Kalıtımsız durumda H'nin kilit beklemesi 8, kalıtımlı durumda 2 birim çıktı. Kilidi spinlock ile bekleyen H ise tek çekirdekte daha kötü bir sonuç doğurur. H işlemciyi hiç bırakmadığı için L hiç çalışamaz ve kilidi hiç bırakamaz: sistem kilitlenir.

Örnek 6.9

Tek CPU'da öncelikler H > M > L'dir. L t = 0'da kilidi alır ve kritik bölgesi 4 birim sürer. H t = 1'de gelir, hemen aynı kilidi ister; kilidi alınca 2 birim çalışır. M t = 2'de gelir, 5 birim çalışır ve kilit kullanmaz. H kilit için kaç birim bekler ve ne zaman biter?

1. **Kalıtım yok.** L 0–1 arası çalışır. H t = 1'de bloke olur. L 1–2 arası sürer. M t = 2'de gelir ve L'yi keserek 2–7 arası çalışır. L kalan 2 birimi 7–9 arası bitirir ve kilidi bırakır. H 9–11 arası çalışır. H 1'den 9'a kadar 8 birim bekler ve t = 11'de biter.
2. **Kalıtım var.** H t = 1'de bloke olunca L, H'nin önceliğini alır. M t = 2'de gelse de L'yi kesemez. L kalan 3 birimi 1–4 arası bitirir ve kilidi bırakır. H 4–6 arası çalışır. H 3 birim bekler ve t = 6'da biter; M ise 6–11 arası çalışır.

6.11.2 Mars Pathfinder (1997)

Öncelik terslemesinin en ünlü örneği Mars'ta yaşandı. Temmuz 1997'de Mars yüzeyindeki Pathfinder aracı, VxWorks gerçek zamanlı işletim sistemi altında çalışırken tekrar tekrar kendini sıfırladı. Düşük öncelikli meteoroloji görevi, bilgi veri yolunu koruyan bir mutex'i tutuyordu. Yüksek öncelikli veri yolu yönetim görevi bu mutex'te bloke oldu. Bu sırada orta öncelikli ve uzun süren iletişim görevi araya girdi (Şekil 6.35).



Şekil 6.35: Pathfinder'daki öncelik terslemesi. Meteoroloji görevi (düşük) mutex'i tutar, veri yolu yönetimi (yüksek) bekler, iletişim görevi (orta) uzun süre çalışır ve bekçi zamanlayıcı sistemi sıfırlar.

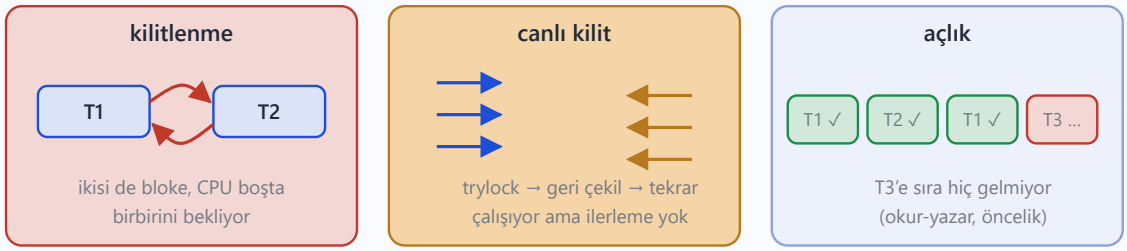
Veri yolu yönetim görevi zamanında çalışamayınca bir **bekçi zamanlayıcı** (*watchdog*) "sistem asılı kaldı" sonucuna vardı ve tam bir sıfırlama yaptı. JPL mühendisleri hatayı yerdeki kopya üzerinde yeniden üretti. Mutex'in öncelik kalıtımı seçeneği VxWorks'te vardı ama kapalıydı; uzaktan yüklenen küçük bir yamayla açıldı ve sıfırlamalar durdu. Bugün aynı mekanizma POSIX'te `PTHREAD_PRIO_INHERIT` özniteliği, Linux'ta öncelik kalıtımlı futex'ler ve `rt_mutex` olarak bulunur. Linux 6.12'de ana dala giren PREEMPT_RT gerçek zamanlı yaması da bu mekanizmaya dayanır.

6.11.3 Kilitlenme, canlı kilit ve açlık

Üç canlılık sorunu sık karıştırılır (Şekil 6.36):

Tanım 6.14

- **Kilitlenme** (*deadlock*): bir süreç kümesinin her üyesi, kümedeki başka bir üyenin tuttuğu kaynağı bekler. Süreçler bloke olur ve işlemci boşta kalabilir. Dört gerekli koşul aynı anda sağlanınca oluşur.
- **Canlı kilit** (*livelock*): süreçler bloke değildir; birbirine tepki verip durmadan aynı adımları tekrarlar ama hiçbiri ilerlemez.
- **Açlık** (*starvation*): sistem bütün olarak ilerler ama bir süreç sürekli başkalarına yol verir ve hiç sıra almaz.



Şekil 6.36: Kilitlenmede herkes bloke, canlı kilitte herkes çalışıyor ama ilerleme yok, açlıkta ise ötekiler ilerlerken T3'e sıra hiç gelmez.

Canlı kilidin tipik örneği iki "kibar" iş parçacığıdır. Her biri iki kilit ister. İkinci kilidi `trylock` ile alamayan, elindekini bırakır ve hep aynı süre sonra yeniden dener. İki iş parçacığı tam aynı ritimde çalışırsa sonsuza dek birbirine yol verir. Çare, rastgele (genellikle üstel artan) bir süre geri çekilmektir; Ethernet çarpışmalarında kullanılan yöntem budur. Açlığın çaresi ise FIFO kuyruklar, adil kilitler ve yaşlandırmadır.

6.12 Kilitsiz yapılara giriş

Kilit kullanan her yapı bir zayıflık taşır: kilidi tutan iş parçacığı kesilir, bir sayfa hatası bekler ya da çökerse herkes durur. **Kilitsiz** (*lock-free*) yapılar kilit yerine doğrudan CAS gibi atomik komutları kullanır. İlerleme garantileri üç düzeyde tanımlanır:

Tür	Garanti
Bloke eden	Kilidi tutan durursa herkes durur.
Kilitsiz	Sistem bütün olarak her zaman ilerler; bazı iş parçacıkları tekrar tekrar başarısız olabilir.
Beklemesiz (*wait-free*)	Her iş parçacığı kendi işlemini sınırlı sayıda adımda bitirir.

Tablo 6.20: İlerleme garantileri.

En basit kilitsiz yapı Treiber yığınıdır. Yığının tepesi tek bir atomik işaretçidir. `push` yeni düğümü hazırlar, düğümün `next` alanını mevcut tepeye bağlar ve tepeyi CAS ile değiştirmeye çalışır:

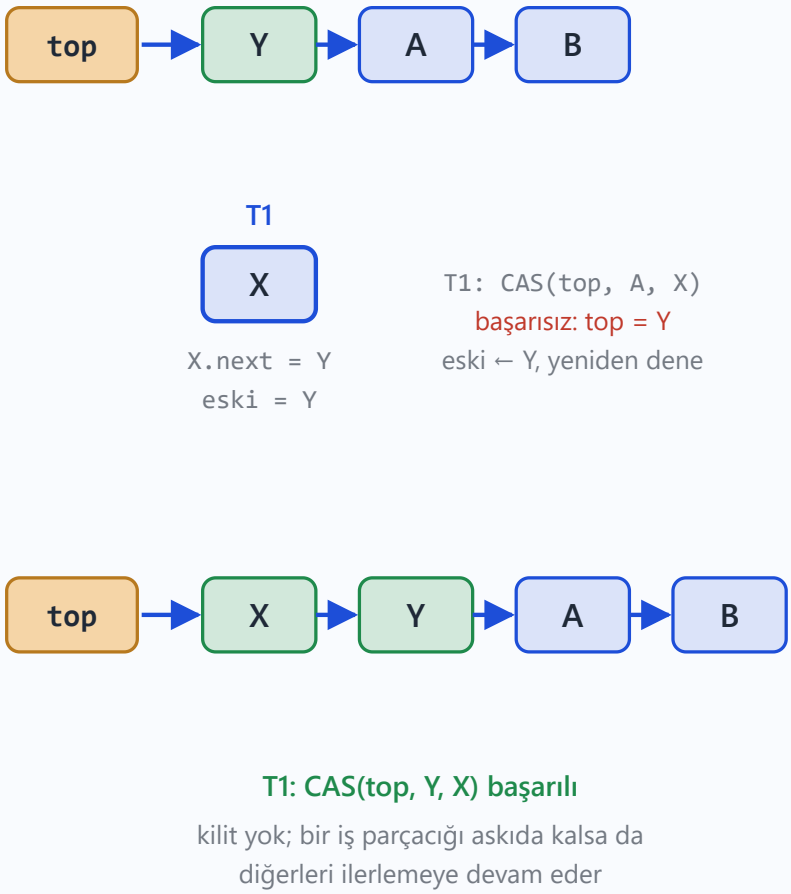
```
void push(node_t *n) {
  node_t *eski = atomic_load(&top);
  do {
    n->next = eski;
  } while (!atomic_compare_exchange_weak(
              &top, &eski, n));
}  /* başarısızsa eski = güncel top */
```

Örnek 6.10

Yığın top → A → B'dir. T1, X'i; T2, Y'yi eklemek ister.

1. **T1 hazırlanır.** `eski = A` okur ve `X.next = A` yapar. Bu sırada kesilir.
2. **T2 ekler.** `CAS(top, A, Y)` başarılıdır; yığın top → Y → A → B olur.
3. **T1'in CAS'ı başarısız olur.** `top` artık A değil Y'dir. C11'in `compare_exchange` fonksiyonu başarısızlıkta güncel değeri `eski` 'ye yazar: `eski = Y`. Döngü `X.next = Y` yapar ve yeniden dener.
4. **İkinci CAS başarılıdır.** Yığın top → X → Y → A → B olur.

Kimse kilit tutmadı. T1 adım 1'den sonra sonsuza dek askıda kalsaydı bile T2 işini bitirebilirdi.



Şekil 6.37: Treiber yığınında `push`. (a) T2'nin eklemesinden sonra T1'in `CAS(top, A, X)`'i başarısız olur; T1 `eski`'yi Y yapar. (b) İkinci deneme başarılıdır.

Kilitsiz programlamanın ünlü tuzağı **ABA sorunudur**. CAS yalnız değerin eşitliğine bakar; değerin arada değişip geri dönüp dönmediğini bilemez. `pop` sırasında T1, `top = A` ve `A.next = B` okuyup kesilsin. Bu arada T2, A'yı ve B'yi çıkarır, B'yi serbest bırakır ve A'yı yeniden ekler. `top` yeniden A olmuştur. T1'in `CAS(top, A, B)` çağrısı başarılı olur ve tepeye artık yığında olmayan, belleği serbest bırakılmış B düğümünü koyar. Çare, işaretçiyle birlikte her değişiklikte artan bir sürüm sayacını karşılaştırmak ya da serbest bırakılan düğümleri güvenli bellek geri kazanımıyla (*hazard pointer*, *epoch*) geciktirmektir.

6.13 Özet

- **Yarış durumu**, paylaşılan veriye erişen süreçlerin sonucunun yürütme sırasına bağlı olmasıdır. Tek bir `count++` bile makinede oku-değiştir-yaz dizisidir ve arasında kesilebilir. Bu tür hatalar ara sıra ortaya çıkar ve testle yakalanması zordur.
- **Kritik bölge** paylaşılan veriye erişen koddur. Bir çözüm üç koşulu sağlamalıdır: **karşılıklı dışlama** (güvenlik), **ilerleme** ve **sınırlı bekleme** (canlılık).
- **Kesme kapatma** çok çekirdekte işe yaramaz ve kullanıcıya açık değildir. **Kilit değişkeni** dışlamayı, **katı sıra** ilerlemeyi bozar. **Peterson algoritması** iki süreç için üç koşulu yalnız okuma ve yazmayla sağlar; ama sıralı tutarlı bellek varsayar.
- Modern işlemcilerde yazma tamponu ve derleyici yeniden sıralaması Peterson'u bozar. x86 (TSO) yalnız "yazma → sonraki okuma" yeniden sıralamasına izin verir. **Bellek bariyeri** ya da sıralı tutarlı atomikler (C11 `_Atomic`, Java `volatile`) doğru sırayı geri getirir; C'deki `volatile` getirmez.
- Donanım **atomik komutlar** sunar: `test_and_set`/TSL, XCHG ve CAS. Basit atomik spinlock sınırlı beklemeyi garanti etmez; `waiting[]` dizisi ve döngüsel devretme bunu sağlar. CAS döngüsü kilitsiz güncellemenin temelidir.
- **Spinlock** meşgul bekler ve kısa kritik bölgeler için çok çekirdekte uygundur. **Mutex** bekleyeni uyutur. Linux'ta mutex'ler **futex** ile kurulur: çekişme yoksa çekirdeğe girilmez. Çekirdek başına sayaç, doğruluktan biraz ödün vererek ölçeklenir.
- **Semafor** `wait`/`signal` ile kullanılan bir sayaçtır. Başlangıç değeri 1 ise dışlama, N ise kaynak sayma, 0 ise sıralama kalıbıdır. Semafor uyandırmayı saklar; sinyal kaybolmaz.
- **Sınırlı tampon** `mutex`, `empty` ve `full` semaforlarıyla çözülür; `down` sırası ters olursa kilitlenme olur. Okur öncelikli **okur-yazar** çözümü yazar açlığına açıktır. **Yemek yiyen filozofların** naif çözümü kilitlenir; oda semaforu, kaynak sıralaması ya da durum dizisiyle çözülür.
- **Monitör** dışlamayı dil düzeyinde sağlar; **koşul değişkeni** değeri olmayan bir bekleme kuyruğudur. Gerçek sistemler Mesa anlamını kullanır: bekleme her zaman `while` içindedir, ayrı gruplar için ayrı koşul değişkenleri kullanılır, hangi bekleyenin ilerleyeceği bilinmiyorsa `broadcast` yapılır.
- Pthreads'te `pthread_cond_wait` mutex'i atomik olarak bırakıp uyur. Java'da her nesne bir monitördür (`synchronized`, `wait`, `notifyAll`); `ReentrantLock` birden çok `Condition` sunar.
- **Öncelik terslemesi** yüksek öncelikli görevin dolaylı olarak orta öncelikliyi beklemesidir; **öncelik kalıtımı** ya da tavanı ile çözülür (Mars Pathfinder). Kilitlenme, canlı kilit ve açlık farklı canlılık sorunlarıdır.
- **Kilitsiz** yapılar CAS döngüsüyle çalışır ve kilidi tutanın durmasından etkilenmez; ama ABA sorununa karşı sürüm sayacı ya da güvenli bellek geri kazanımı gerekir.

Araç	Ne sağlar?	Ne zaman?
Peterson, katı sıra	Yalnız okuma/yazmayla dışlama (SC varsayımıyla)	Kavramsal; gerçek kodda kullanılmaz

Araç	Ne sağlar?	Ne zaman?
Bellek bariyeri	Okuma/yazma sırası ve görünürlük	Kilit ve atomik gerçeklemelerinin içinde
TAS, XCHG, CAS	Atomik okuma-değiştirme-yazma	Kilitlerin ve kilitsiz yapıların temeli
Atomik değişken	Tek değişkenin bölünmez güncellemesi	Sayaç, bayrak, referans sayımı
Spinlock	Dışlama, uyumadan	Çok çekirdek, çok kısa kritik bölge, çekirdek içi
Mutex	Dışlama, bekleyen uyur	Genel amaçlı kritik bölgeler
Semafor	Sayma, dışlama ve sıralama	N özdeş kaynak, olay bildirimi
Monitör + koşul değişkeni	Yapısal dışlama + koşul bekleme	Karmaşık koşullar (Java, Pthreads)
Okur-yazar kilidi	Çoklu okur, tek yazar	Okuma ağırlıklı veriler

Tablo 6.21: Senkronizasyon araçlarının özeti.

6.14 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

senkronizasyon *synchronization* · yarış durumu *race condition* · kritik bölge *critical section* · karşılıklı dışlama *mutual exclusion* · sınırlı bekleme *bounded waiting* · meşgul bekleme *busy waiting* · Peterson algoritması *Peterson's algorithm* · bellek bariyeri *memory barrier* · atomik komut *atomic instruction* · spinlock *spinlock* · mutex *mutex* · semafor *semaphore* · üretici–tüketici problemi *producer–consumer problem* · okur-yazar problemi *readers–writers problem* · yemek yiyen filozoflar problemi *dining philosophers problem* · monitör *monitor* · koşul değişkeni *condition variable* · öncelik terslemesi *priority inversion* · öncelik kalıtımı *priority inheritance* · kilitlenme *deadlock* · canlı kilit *livelock* · açlık *starvation*

6.15 Kendini Dene

1. Paylaşılan `count = 10`'dur. T1 `count = count + 3`, T2 `count = count - 1` yürütür; her biri LOAD, ADD, STORE olmak üzere üç komuttur. Olası bütün son değerler nelerdir?

   Cevap

   12, 13 ve 9. Biri ötekinin okumasından önce bitirirse 12 çıkar. T2'nin güncellemesi kaybolursa 13, T1'inki kaybolursa 9 çıkar. 20 ara yerleştirmenin hepsi bu üç değerden birini verir.

2. Peterson algoritmasının çıkış bölümündeki `flag[i] = false` satırı unutulursa üç koşuldan han-

gisi bozulur? P0 bir kez girip çıktıktan sonra kalan bölümde kalırsa P1'e ne olur?

Cevap

İlerleme bozulur. `flag[0]` sonsuza dek `true` kalır. P1 girmek isteyince `turn = 0` yazar; koşulu `flag[0] && turn == 0` doğru olur ve P1 döner. P0 kalan bölümde dururken P1'in önünü açacak tek şey P0'ın yeniden girmek isteyip `turn = 1` yazmasıdır. Kritik bölge boş olduğu hâlde kalan bölümdeki bir süreç ötekini engeller.

3. Başlangıç değeri 3 olan bir semaforda (Silberschatz gerçeklemesi, FIFO kuyruk) P1–P5 sırayla `wait`, ardından iki `signal` yapılıyor. Beş `wait`'ten sonra ve iki `signal`'dan sonra değer ve kuyruk nedir?

Cevap

Beş `wait`'ten sonra değer 3 − 5 = −2'dir; P1, P2, P3 geçmiştir, P4 ve P5 kuyruktadır. İlk `signal` değeri −1 yapar ve P4'ü, ikincisi 0 yapar ve P5'i uyandırır. Son durumda değer 0 ve kuyruk boştur.

4. Semaforlu sınırlı tampon çözümünde N = 3'tür ve tüketici hiç çalışmıyor. Üretici beş öğe koymaya çalışırsa kaç öğe konur, `empty` ve `full` kaç olur, üretici hangi öğede uyur?

Cevap

Üç öğe konur. Her öğe konduğunda `empty` bir azalır, `full` bir artar: sonunda `empty = 0`, `full = 3`'tür. Dördüncü öğe için `down(&empty)` değeri 0 bulur ve üretici uyur. Tampon taşmaz.

5. Semaforla bariyer kodunda turnikeden geçtikten sonraki `up(&turnike)` satırı silinirse, N = 3 iken kaç iş parçacığı bariyeri geçer?

Cevap

Yalnız biri. Son gelen iş parçacığının `up`'ı turnikedeki tek bir iş parçacığını geçirir. Geçen arkadakini geçirmediği için öteki ikisi turnikede sonsuza dek uyur. Bütün ara yerleştirmeler gezildiğinde kilitlenen her yürütmede geçen sayısı 1'dir.

6. `v = 0` iken üç iş parçacığı CAS döngüsüyle (`artir`) birer kez artırma yapıyor. Son değer kaçtır? Toplam CAS çağrısı en az ve en çok kaç olabilir?

Cevap

Son değer her zaman 3'tür. En az 3 CAS yapılır: her biri ötekiler okumadan önce bitirirse hiç başarısızlık olmaz. En çok 3 + 2 + 1 = 6 CAS yapılır. Üçü de aynı değeri okursa biri başarılı olur ve ikisi yeniden dener. Sonraki turda yine biri başarılı, biri başarısız olur.

7. Tek CPU'da H > M > L'dir. L t = 0'da kilidi alır ve kritik bölgesi 3 birimdir. H t = 1'de gelir ve hemen aynı kilidi ister; kilidi alınca 1 birim çalışır. M t = 2'de gelir ve 4 birim çalışır, kilit kullanmaz. Öncelik kalıtımı yokken ve varken H kaç birim bekler, ne zaman biter?

   Cevap

   Kalıtım yokken çizelge L(0–1), L(1–2), M(2–6), L(6–7), H(7–8)'dir. H 1'den 7'ye kadar 6 birim bekler ve t = 8'de biter. Kalıtımla L, H'nin önceliğiyle 1–3 arası çalışır ve kilidi bırakır; H 3–4 arası çalışır. H 2 birim bekler ve t = 4'te biter; M 4–8 arası çalışır.

8. İki çekirdekli makinede eşik `S = 5` olan yaklaşık sayaç kullanılıyor. Çekirdek 0 on iki, çekirdek 1 yedi kez `ekle()` çağırıyor. Sonunda `oku()` kaçı verir? Gerçek değer kaçtır? En kötü durumda fark en fazla kaç olabilir?

   Cevap

   Çekirdek 0 iki kez 5 aktarır ve yerelde 2 kalır; çekirdek 1 bir kez 5 aktarır ve yerelde 2 kalır. `oku()` = G = 15, gerçek değer 19'dur. En kötü durumda fark 2 × (5 − 1) = 8'dir.

9. Altı filozoflu masada aynı anda en fazla kaç filozof yiyebilir? Naif çözüm (önce sol, sonra sağ çatal) altı filozofta da kilitlenir mi?

   Cevap

   ⌊6/2⌋ = 3: örneğin F0, F2 ve F4 birlikte yiyebilir. Naif çözüm filozof sayısından bağımsız olarak kilitlenir: hepsi sol çatalını alırsa döngüsel bekleme oluşur.

10. Yazma tamponu testinde T1 `x = 1; r1 = y;`, T2 `y = 1; r2 = x;` yürütür. Tam bellek bariyeri yalnız T1'de, yazma ile okuma arasına konuyor. x86 (TSO) üzerinde `r1 = r2 = 0` hâlâ gözlenebilir mi?

    Cevap

    Evet. T1'in bariyeri yalnız T1'in kendi yazmasının okumasından önce görünmesini sağlar. T2'nin `y = 1` yazması tamponda beklerken T2 `x`'i bellekten 0 okuyabilir; T1 de `y`'yi bellekten 0 okuyabilir. Yazma tamponlu modelde bütün yürütmeler gezildiğinde (0, 0) yine çıkar. Bariyer iki tarafa da konmalıdır.

6.16 İleri okuma

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, “Locks”, “Condition Variables”, “Semaphores” ve “Common Concurrency Problems” bölümleri: kilitlerin donanımdan nasıl kurulduğunu, yaklaşık sayacı ve koşul değişkenlerindeki tuzakları kısa ve ölçümlü

örneklerle anlatır.

- A. B. Downey, *The Little Book of Semaphores*: turnikeli bariyerden açlıksız okur-yazara kadar onlarca problemi semaforlarla çözer; her çözümden önce okurun kendisinin denemesine yer bırakır. Kitap ücretsizdir.
- A. Silberschatz, P. B. Galvin ve G. Gagne, *Operating System Concepts* (10. baskı), Bölüm 6 ve 7: kritik bölge koşullarını, donanım desteğini ve gerçek sistemlerdeki araçları ayrıntılı işler.
- B. Goetz ve arkadaşları, *Java Concurrency in Practice*: Java bellek modelini, `volatile`'ın anlamını ve `java.util.concurrent` araçlarının doğru kullanımını uygulamalı olarak anlatır.

6.17 Kaynaklar

1. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Locks", "Condition Variables", "Semaphores", "Common Concurrency Problems".
2. A. B. Downey, *The Little Book of Semaphores*, 2. baskı, Green Tea Press.
3. B. Goetz ve ark., *Java Concurrency in Practice*, Addison-Wesley, 2006; Java SE 21 API: `java.util.concurrent`; JEP 491.
4. U. Drepper, "Futexes Are Tricky", Red Hat teknik raporu, sürüm 1.6, 2011 (ilk sürüm 2004), akkadia.org/drepper/futex.pdf.
5. M. Jones, "What really happened on Mars?", 1997.
6. M. M. Michael, M. L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms", *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing* (PODC '96), ss. 267–275, 1996, doi:10.1145/248052.248106.
7. G. L. Peterson, "Myths About the Mutual Exclusion Problem", *Information Processing Letters*, 1981.
8. POSIX.1-2017 (IEEE Std 1003.1): `pthread_mutex_lock`, `pthread_cond_wait`, `sem_overview(7)`; Linux `futex(2)` kılavuz sayfası.
9. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 6: Synchronization Tools, Bölüm 7: Synchronization Examples.
10. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı, Pearson, 2014 (5. baskı 2023) — §2.3 Interprocess Communication, §2.5 Classical IPC Problems.

KISIM II · BÖLÜM 7

Kilitlenme

İki iş parçacığı iki kilidi ters sırada almaya çalışırsa ikisi de sonsuza dek bekleyebilir. Hiçbiri çalışmadığı için hiçbiri elindeki kilidi bırakmaz. Bu duruma **kilitlenme** denir. Kilitlenen program hata vermez, çökmez; yalnızca durur. Üstelik çoğu zaman yalnız belirli bir zamanlamada ortaya çıkar ve testlerde görülmeyebilir. Not önce kaynak kavramıyla ve kilitlenmenin dört koşuluyla başlar. Ardından kilitlenmeyi çizgelerle modeller ve dört başa çıkma stratejisini sırayla işler: görmezden gelme, tespit edip kurtarma, kaçınma ve önleme. Tespit için döngü arama ve matris tabanlı algoritma, kaçınma için güvenli durum ve bankacı algoritması adım adım çözülür. Son kesimler canlı kilidi, açlığı, kilitlenme dışı eş zamanlılık hatalarını ve Java programlarında kilitlenmenin nasıl bulunduğunu anlatır.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum07` · **Kod (Java):** `github.com/sercankulcu/operating-systems-java/Bolum07/src` · **Laboratuvar:** `github.com/sercankulcu/operating-systems-labs/lab07`

ÖĞRENME ÇIKTILARI

Tanım ve kaynaklar
Kilitlenmeyi tanımlamak; kesilebilir ve kesilemez kaynakları, iste–kullan–bırak döngüsünü ayırt etmek.

Dört koşul
Karşılıklı dışlama, tut ve bekle, kesme yok ve döngüsel bekleme koşullarını bir örnek üzerinde göstermek.

Modelleme
Kaynak tahsis ve bekleme çizgelerini çizmek; çizgedeki döngünün ne zaman kilitlenme anlamına geldiğini söylemek.

Tespit ve kurtarma
Döngü arama ve matris tabanlı tespit algoritmalarını elle çalıştırmak; kesme, geri sarma ve sonlandırmayla kurtarmayı karşılaştırmak.

Kaçınma
Güvenli ve güvensiz durumları ayırt etmek; bankacı algoritmasıyla bir isteğin verilip verilmeyeceğine karar vermek.

Önleme ve pratik
Kilit sıralamasını, iki aşamalı kilitlemeyi, canlı kilidi ve açlığı tanımak; Java'da kilitlenmeyi `jstack` ile bulmak.

7.1 Kilitlenme ve kaynaklar

Eş zamanlı çalışan süreçler kaynakları paylaşır: bir yazıcıyı, bir dosyayı, bir veritabanı kaydını, bellekteki bir veri yapısını koruyan kilidi. Paylaşımı düzenlemenin olağan yolu, kaynağı kullanan süreç işini bitirene kadar ötekileri bekletmektir. Bekletmek çoğu zaman sorunsuzdur; bekleyen süreç sırası gelince devam eder. Ama bekleme bir halka oluşturursa sıra hiç gelmez. Bu kesim kilitlenmenin tanımını, ilgili kaynak türlerini ve en küçük kilitlenme örneğini verir.

7.1.1 Kilitlenme nedir?

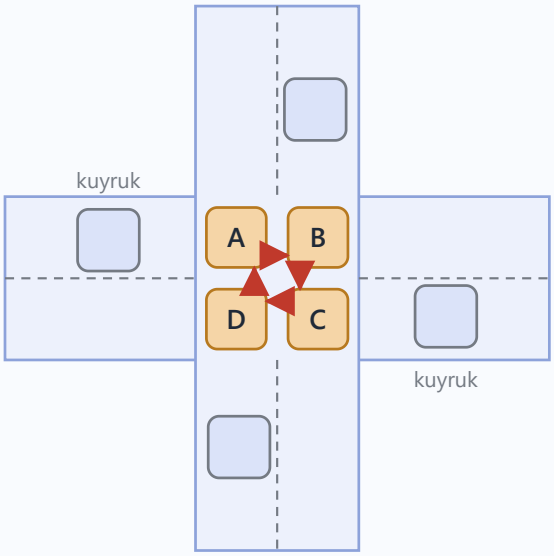
En basit durumu düşünelim. S1 süreci K1 kaynağını tutuyor ve K2'yi bekliyor. S2 süreci de K2'yi tutuyor ve K1'i bekliyor. S1, K2'yi alamadığı için ilerleyemez ve K1'i bırakmaz. S2 de aynı nedenle K2'yi bırakmaz. İki süreç de bloke olmuştur. Onları uyandıracak tek olay ötekinin kaynağı bırakmasıdır; o olay da hiç gerçekleşmeyecektir.

Tanım 7.1

Bir süreç kümesindeki her süreç, yalnız kümedeki başka bir sürecin tetikleyebileceği bir olayı bekliyorsa küme **kilitlenmiştir** (*deadlock*). Beklenen olay çoğu zaman bir kaynağın serbest bırakılmasıdır.

Tanımdaki “yalnız kümedeki başka bir süreç” ifadesi önemlidir. Bekleyen süreci dışarıdan biri kurtarabiliyorsa, örneğin kaynağı tutan süreç kümenin dışındaysa ve bir gün işini bitirecekse, ortada kilitlenme yoktur; yalnızca uzun bir bekleme vardır. Kilitlenmede ise kurtarıcı kümenin içindedir ve kümedeki herkes beklediği için kurtarıcı hiç çalışmaz.

Gündelik hayatta da benzer durumlar vardır. Dört aracın aynı anda girdiği bir kavşakta her araç sağındakinin önünü keser; hiçbiri ilerleyemez (*gridlock*). Tek hatlı bir rayda karşılıklı gelen iki tren de birbirini bekler. Şekil 7.1 kavşak örneğini gösterir.



Şekil 7.1: Kavşakta kilitlenme. Dört araç kavşağın dört karesini tutar; her biri bir sonrakinin tuttuğu kareyi bekler. Arkadaki kuyruklar yüzünden hiçbiri geri de çekilemez.

Kilitlenen süreçler işlemci harcamaz, çünkü bloke durumdadırlar. Asıl zarar, tuttukları kaynakların sonsuza dek kullanılamamasıdır. O kaynakları isteyen başka süreçler de zamanla beklemeye başlar ve kilitlenme yayılır. Konu yalnız işletim sistemine özgü değildir: veritabanı işlemleri, çok iş parçacıklı uygulamalar ve dağıtık sistemler de kilitlenir. Bu notta "süreç" sözcüğü iş parçacığını da kapsar; anlatılan her şey iki durumda da geçerlidir.

7.1.2 Kesilebilir ve kesilemez kaynaklar

Kilitlenmenin konusu **kaynaklardır**. Kaynak, bir sürecin kullanmak için edinmesi gereken her şeydir: bir donanım aygıtı, bir bellek bölgesi, bir dosya, bir kilit, bir veritabanı kaydı. Kaynakları kilitlenme açısından ikiye ayırmak gerekir.

> **Tanım 7.2**
>
> **Kesilebilir kaynak** (*preemptable resource*), sahibinden zarar vermeden alınıp sonra geri verilebilen kaynaktır. **Kesilemez kaynak** (*nonpreemptable resource*) ise sahibinden alınırsa yürüyen işi bozar; yalnız sahibi kendi isteğiyle bırakabilir.

İşlemci kesilebilir bir kaynaktır. Çizelgeleyici çalışan süreci durdurur, yazmaçlarını kaydeder ve işlemciyi başka sürece verir; süreç sonra kaldığı yerden devam eder. Ana bellek de kesilebilir: bir sayfa diske yazılıp çerçeve başka sürece verilebilir, sayfa gerektiğinde geri yüklenir. Bu tür kaynaklarda bekleme halkası oluşsa bile işletim sistemi kaynağı yeniden dağıtarak halkayı kırabilir.

Yarısı yazılmış bir Blu-ray diski kesilemez: yazma işini yarıda kesip sürücüyü başka sürece vermek diski bozar. Yazıcıda basılmakta olan bir iş de böyledir; başka bir işin sayfaları araya girer. Bir mutex ya da semafor ile korunan veri, kilit zorla alınırsa tutarsız bir ara durumda kalır. Kilitlenmeler bu tür kesilemez kaynaklarla ilgilidir. Tablo 7.1 yaygın kaynakları türlerine göre sıralar.

Kaynak	Örnek sayısı	Tür	Not
CPU çekirdeği	çok	kesilebilir	zamanlayıcı kesmesiyle el değiştirir
Bellek sayfası	çok	kesilebilir	takas ya da sayfalama ile geri alınır
Mutex, semafor, dosya kilidi	1	kesilemez	yalnız sahibi bırakır
Veritabanı satır kilidi	1	kesilemez*	*işlem geri sarılırsa bırakılır

Tablo 7.1: Kaynakların kilitlenme açısından sınıflandırılması.

Tablodaki "örnek sayısı" sütunu da önemlidir. Bir kaynak türünün birden çok **özdeş örneği** (*instance*) olabilir. Sistemde iki teyp sürücüsü varsa ve bir süreç "bir teyp sürücüsü" istiyorsa, iki sürücüden hangisinin verildiği fark etmez. Bir istek o türün herhangi bir örneğiyle karşılanabiliyorsa örnekler özdeştir. Genel olarak R_i kaynak türünün W_i tane örneği vardır. Örnek sayısı, kilitlenmenin nasıl tespit edileceğini doğrudan etkiler.

> **Sık yapılan hata**
>
> Veritabanı kaydı kilidinin kesilebilir olduğunu düşünmek, çünkü veritabanı kilitlenmeyi işlemi iptal ederek çözer. İptal, kilidi zorla alıp başkasına vermek değildir; işlemin bütün değişikliklerini geri alınır ve işlem baştan başlar. Kaynak kesilmiş olmaz, iş kaybedilir. Bu yüzden tabloda yıldızla işaretlenmiştir.

7.1.3 Kaynak kullanım döngüsü

Bir süreç bir kaynağı her zaman aynı üç adımla kullanır:

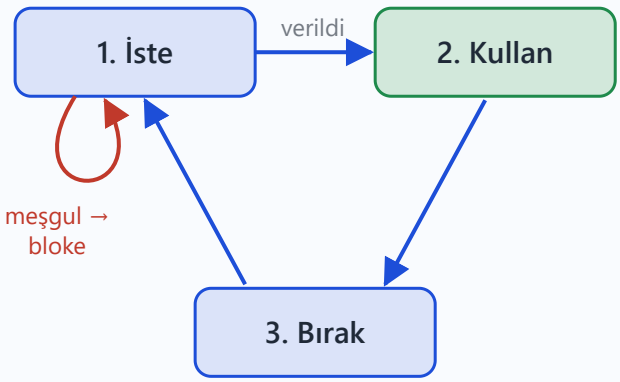
1. **İste** (*request*). Kaynak boştaysa sürece verilir. Boş değilse süreç bloke olur ve kaynak boşalınca uyandırılır. Bazı sistemlerde süreç bloke olmak yerine bir hata kodu alır ve biraz sonra yeniden dener.
2. **Kullan** (*use*). Süreç kaynak üzerinde işlem yapar: yazıcıya basar, dosyaya yazar, korunan veriyi günceller.
3. **Bırak** (*release*). Kaynak serbest kalır; onu bekleyen süreçlerden biri uyandırılır.

İşletim sistemi bir tabloda hangi kaynağın kime verildiğini ve kimin hangi kaynağı beklediğini tutar. Kilitlenme tespiti bu tablodaki bilgiyle yapılır. İsteme ve bırakma işlemleri çoğu zaman sistem çağrılarıdır ya da kütüphane fonksiyonlarıdır; Tablo 7.2 en bilinen çiftleri gösterir.

İste	Bırak
`open()`	`close()`

İste	Bırak
`malloc()` / `mmap()`	`free()` / `munmap()`
`pthread_mutex_lock()`	`pthread_mutex_unlock()`
`sem_wait()` (down)	`sem_post()` (up)

Tablo 7.2: İsteme ve bırakma işlemlerinin karşılıkları.



Şekil 7.2: Kaynak kullanım döngüsü. Kaynak meşgulse süreç istek adımında bloke olur; kaynak bırakılınca uyanır ve kullanım adımına geçer.

7.1.4 Aynı sıra ve ters sıra

Kullanıcı süreçlerinin kendi kaynaklarını semaforlarla koruduğunu varsayalım. Her kaynağa bir ikili semafor karşılık gelsin: kaynağa erişmeden önce `down`, iş bitince `up` yapılır. Tek süreç ya da tek kaynak varken sorun çıkmaz. İki süreç ve iki kaynak olunca sonucu kaynakların alınma sırası belirler. Aşağıdaki iki süreç kaynakları aynı sırada alır:

```
void process_A(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
void process_B(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
```

Bu kod kilitlenmez. İki süreç de önce `resource_1`'i ister. Biri onu alır, öteki bekler. `resource_1`'i alan süreç `resource_2`'yi de rahatça alır, çünkü onu isteyebilecek tek öteki süreç hâlâ `resource_1`'in önünde bekler. İşini bitirip iki kaynağı bırakınca bekleyen süreç devam eder.

Şimdi B'nin sırasını ters çevirelim:

```
void process_A(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
void process_B(void) {
  down(&resource_2);  /* ters sıra */
  down(&resource_1);
  use_both_resources();
  up(&resource_1);
  up(&resource_2);
}
```

Bu sürümde kilitlenme mümkündür. A `resource_1`'i, B de `resource_2`'yi alırsa her biri ötekinin elindekini bekler. Bırakma sırasının bir önemi yoktur; belirleyici olan edinme, yani `down` sırasıdır. Programcının gözünden iki sürüm neredeyse aynıdır. İki satırın yerini değiştirmek, hiç kilitlenmeyen bir programı ara sıra kilitlenen bir programa dönüştürür.

7.1.5 Kod izleme: ters sırada edinim

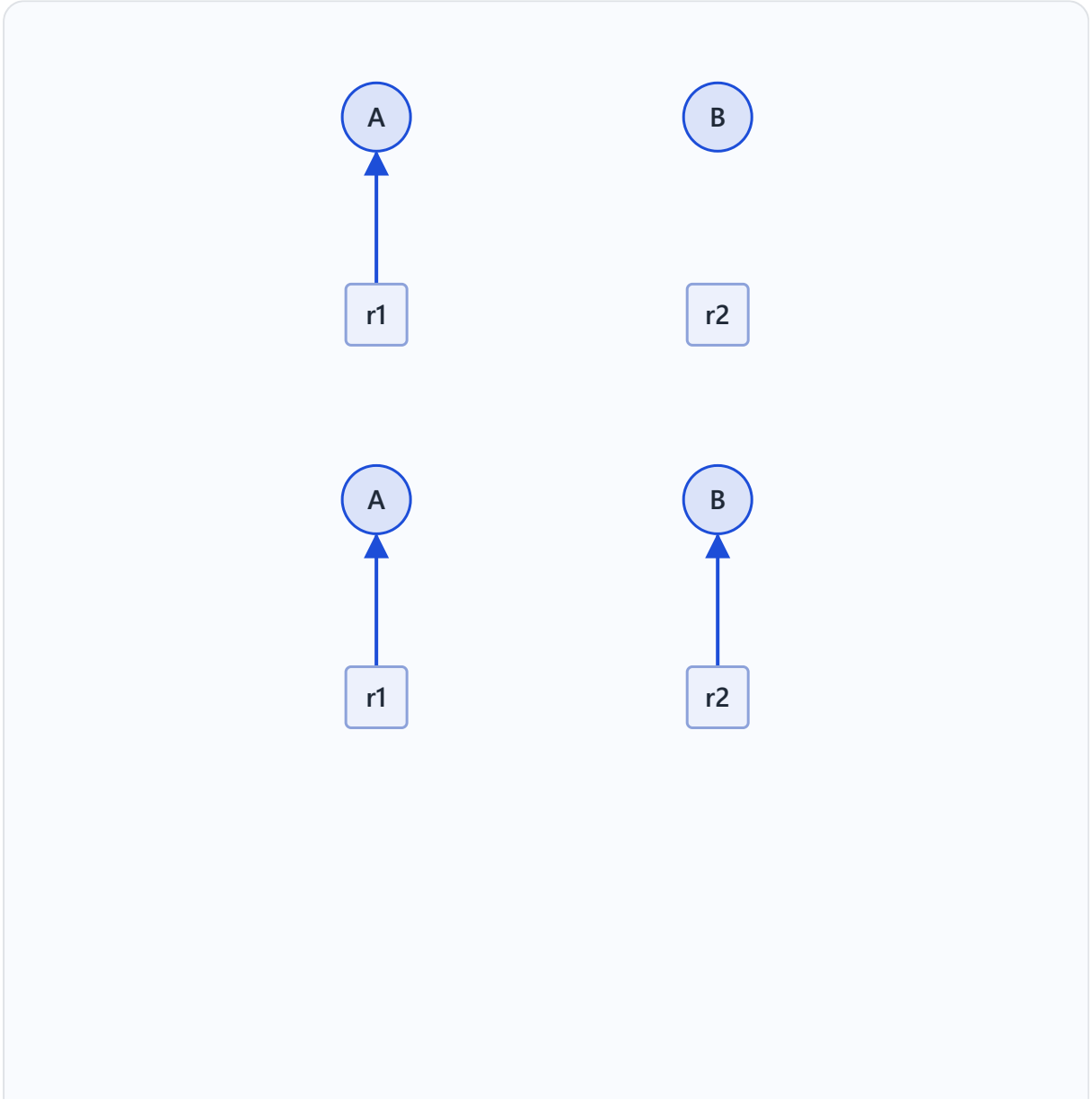
Ters sıralı kodun kısaltılmış hâli aşağıdadır. İki ikili semafor 1 ile başlar; yani iki kaynak da boştur.

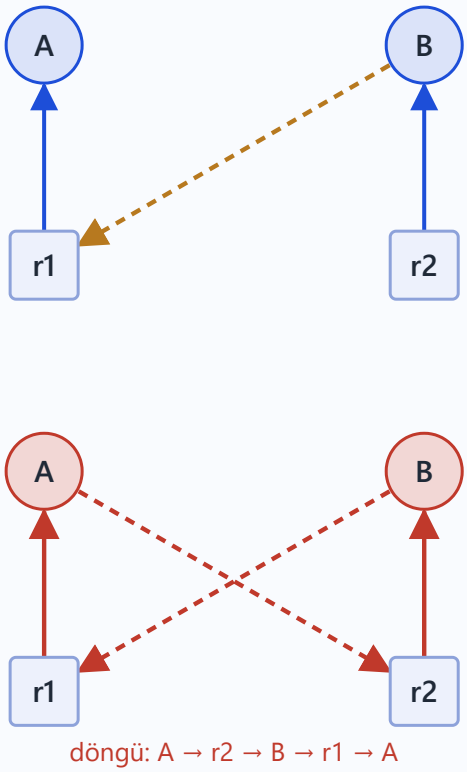
```
semaphore r1 = 1, r2 = 1;
void A(void) {
  down(&r1);
  down(&r2);
  use_both();
  up(&r2); up(&r1);
}
void B(void) {
  down(&r2);
  down(&r1);
  use_both();
  up(&r1); up(&r2);
}
```

Tek bir işlemcide çizelgeleyicinin şu sırayı seçtiğini düşünelim: A'nın ilk `down`'ı, B'nin ilk `down`'ı, A'nın ikinci `down`'ı, B'nin ikinci `down`'ı. Tablo 7.3 her adımda semaforların ve süreçlerin durumunu, Şekil 7.3 de aynı adımların çizgesini gösterir. Çizgede daire süreci, kare kaynağı gösterir. Kaynaktan sürece giden düz ok "bu kaynak bu sürecindir", süreçten kaynağa giden kesikli ok "bu süreç bu kaynağı bekliyor" anlamına gelir.

Adım	Olay	`r1`	`r2`	A	B
0	başlangıç	1	1	hazır	hazır
1	A: `down(&r1)`	0	1	r1'i tutar	hazır
2	B: `down(&r2)`	0	0	r1'i tutar	r2'yi tutar
3	B: `down(&r1)`	0	0	r1'i tutar	bloke (r1)
4	A: `down(&r2)`	0	0	bloke (r2)	bloke (r1)

Tablo 7.3: Ters sırada edinimin izlemesi.





Şekil 7.3: Ters sırada edinim. (a) A `r1`'i alır. (b) Çizelgeleyici B'ye geçer; B `r2`'yi alır. (c) B `r1`'i ister ve bloke olur. (d) A da `r2`'yi ister ve bloke olur: çizgede A → r2 → B → r1 → A döngüsü oluşmuştur.

Dördüncü adımdan sonra iki süreç de bloke durumdadır ve `up` satırlarına hiçbiri ulaşamaz. Çizgedeki döngü kilitlenmenin görsel işaretidir. Kilitlenme her çalıştırmada oluşmaz. A iki `down`'ı da B başlamadan bitirirse B ilk `down`'ında bekler ve A işini bitirince devam eder. Kilitlenme yalnız iki süreç de ilk `down`'ını ötekinin ikinci `down`'ından önce yaparsa oluşur. Bu pencere dar olduğu için hata testlerde nadiren görülür, sistem yük altındayken ortaya çıkar.

Sık yapılan hata

Programı birkaç kez çalıştırıp kilitlenmediğini görerek kodun doğru olduğuna karar vermek. Kilitlenme belirli bir zamanlamaya bağlıdır; bin çalıştırmada bir kez bile görülmeyebilir. Doğruluk, mümkün olan bütün sıralamalar düşünülerek gösterilir. Pratikte en güvenilir yol kilitleri her yerde aynı sırada almaktır.

7.2 Kilitlenmenin dört koşulu

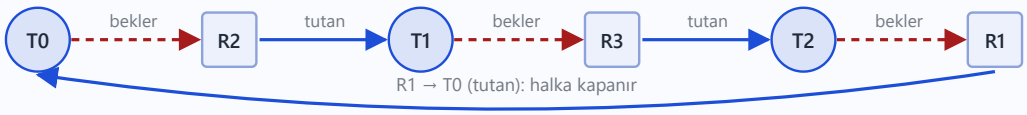
Ters sıralı örnekte kilitlenmeyi mümkün kılan neydi? Kaynaklar paylaşılamıyordu, süreçler bir kaynağı tutarken ötekini bekliyordu, kimse kaynağı zorla geri alamıyordu ve bekleme bir halka oluşturuyordu. 1971'de Coffman, Elphick ve Shoshani bu dört özelliğin kilitlenme için gerekli olduğunu gösterdi.

7.2.1 Coffman koşulları

Teorem 7.1 (Kilitlenmenin dört koşulu)

Kilitlenme olabilmesi için şu dört koşulun aynı anda sağlanması gerekir:

1. **Karşılıklı dışlama** (*mutual exclusion*): her kaynak ya tam olarak bir sürece atanmıştır ya da boştur; aynı anda paylaşılamaz.
2. **Tut ve bekle** (*hold and wait*): en az bir kaynak tutan bir süreç, başkalarının tuttuğu yeni kaynakları ister ve bekler.
3. **Kesme yok** (*no preemption*): verilmiş bir kaynak zorla geri alınamaz; yalnız tutan süreç kendi isteğiyle bırakır.
4. **Döngüsel bekleme** (*circular wait*): T_0, T_1'in; T_1, T_2'nin; ...; T_n de T_0'ın tuttuğu bir kaynağı bekler.



Şekil 7.4: Döngüsel bekleme. T0, R1'i tutar ve R2'yi bekler; T1, R2'yi tutar ve R3'ü bekler; T2, R3'ü tutar ve R1'i bekler. Son ok halkayı kapatır.

Koşulların "birlikte gerekli" olması, birinin bile eksik olduğu bir sistemde kilitlenmenin imkânsız olduğu anlamına gelir. Kaynak paylaşılabiliyorsa kimse onu beklemez. Hiçbir süreç kaynak tutarken beklemiyorsa halka kurulamaz. Kaynaklar geri alınabiliyorsa halka kırılabilir. Halka hiç oluşmuyorsa da bekleyen her süreç sonunda sırasını alır. Kilitlenmeyi **önleme** stratejisi tam olarak bu gözleme dayanır: dört koşuldan birini yapısal olarak imkânsız kılmak.

Dört koşulun doğası aynı değildir. İlk üçü sistemin *politikasıdır*: kaynakların paylaşılıp paylaşılmadığı, süreçlerin nasıl kaynak istediği, kaynakların geri alınıp alınamadığı tasarımda belirlenir. Bu üç politika geçerli olduğunda kilitlenme *olabilir*, ama olmak zorunda değildir. Dördüncü koşul ise sistemin *o anki durumudur*: süreçlerin belirli bir sırayla kaynak istemesi sonucunda halka ya oluşur ya oluşmaz. Her kaynak türünden yalnız bir örnek varsa dört koşul aynı zamanda yeterlidir: dördü de sağlanıyorsa sistem kilitlenmiştir. Çok örnekli kaynaklarda bu yeterlilik ortadan kalkar: döngüdeki bir kaynağın başka bir örneği döngü dışındaki bir süreçte olabilir.

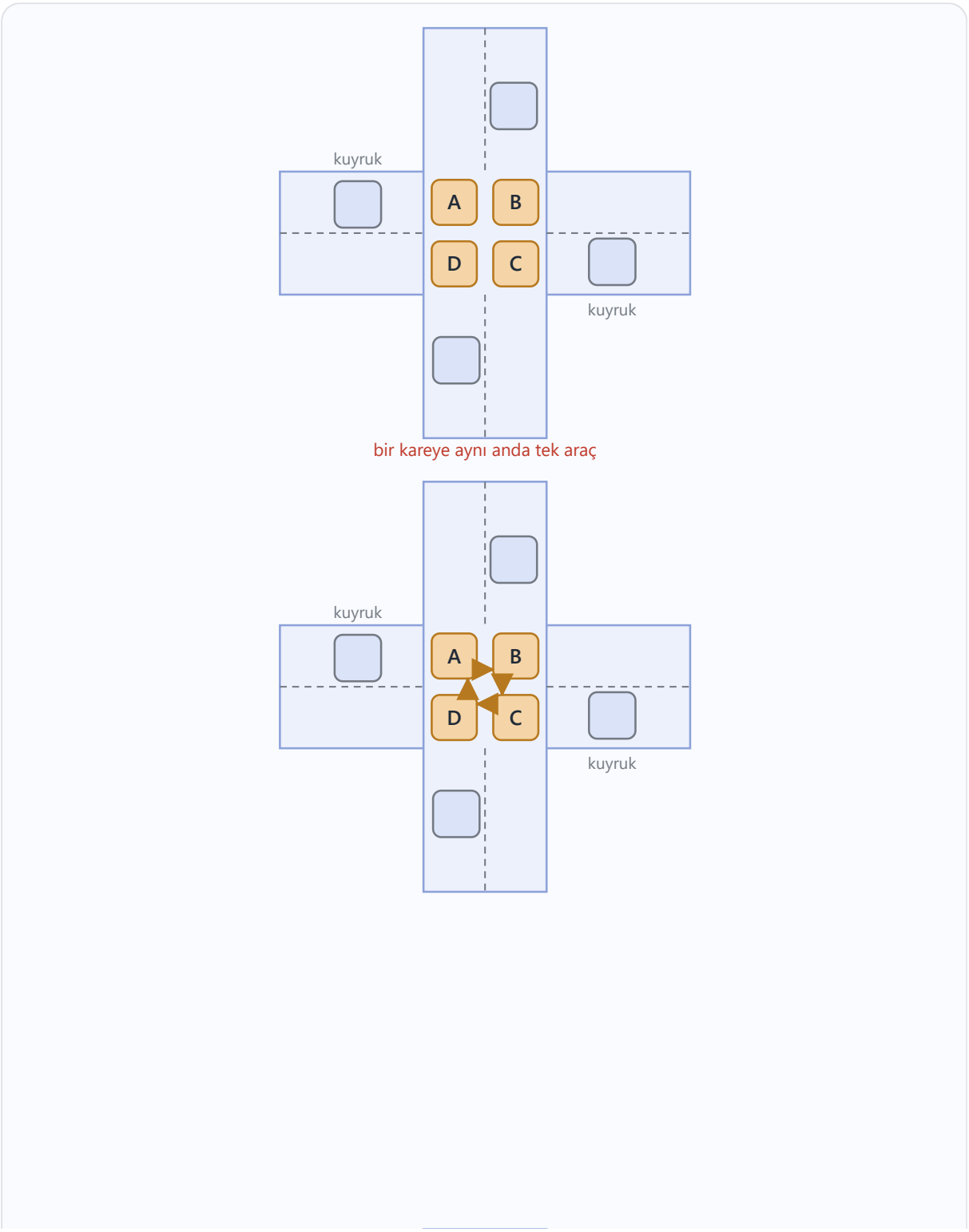
Sık yapılan hata

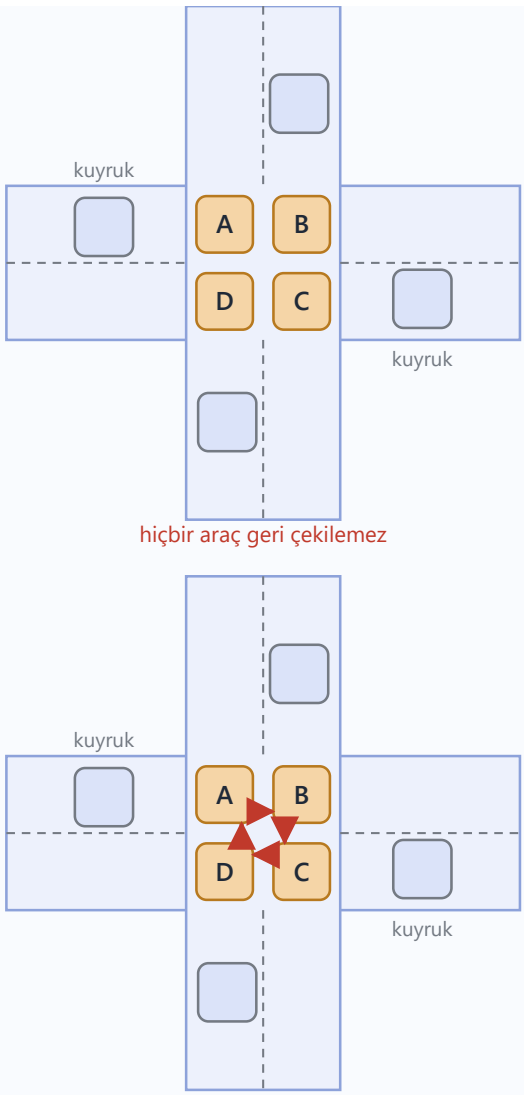
Koşulları birbirinin yerine kullanmak. "Kesme yok" koşulu kaynağın zorla geri alınamamasıdır; "tut ve bekle" ise sürecin bir kaynağı tutarken başka bir kaynak beklemesidir. Süreçten yeni istekten önce elindekileri bırakmasını istemek tut ve beklemeyi kırar. Elindeki kaynağın

işletim sistemince zorla alınması ise kesme yok koşulunu kırar.

7.2.2 Kavşak örneği

Dört koşulu somutlaştırmak için kavşağa dönelim. Kavşağın dört karesi dört kesilemez kaynaktır, araçlar da süreçlerdir. Şekil 7.5 koşulları sırayla işaretler.





Şekil 7.5: Kavşakta dört koşul. (a) Karşılıklı dışlama: bir karede aynı anda tek araç bulunur. (b) Tut ve bekle: her araç bulunduğu kareyi tutarken önündeki kareyi bekler. (c) Kesme yok: arkada kuyruk vardır, hiçbir araç geri çekilemez. (d) Döngüsel bekleme: A → B → C → D → A.

1. **Karşılıklı dışlama.** İki araç aynı kareye sığmaz. Bir kare ya boştur ya tek bir araca aittir.
2. **Tut ve bekle.** Her araç girdiği kareyi tutar ve ilerlemek için önündeki kareyi bekler.
3. **Kesme yok.** Araçların arkasında kuyruk vardır; hiçbiri geri çekilip kareyi boşaltamaz.
4. **Döngüsel bekleme.** A, B'nin karesini; B, C'ninkini; C, D'ninkini; D de A'nınkini bekler. Halka kapanmıştır ve trafik kilitlenir.

Trafikteki çözümler de koşullardan birini kırar. Trafik polisi bir aracı geri çektirirse kesme yok koşulu kırılır. “Kavşak boşalmadan girme” kuralı, aracın bir kareyi tutarken başka kareyi beklemesini engeller; böylece tut ve bekle ortadan kalkar. Trafik ışığı ise yönlere sıra verir ve halkanın kurulmasını önler. Bilgisayar sistemlerindeki önleme yöntemleri de bu üç fikrin karşılığıdır.

7.3 Kilitlenmenin modellenmesi

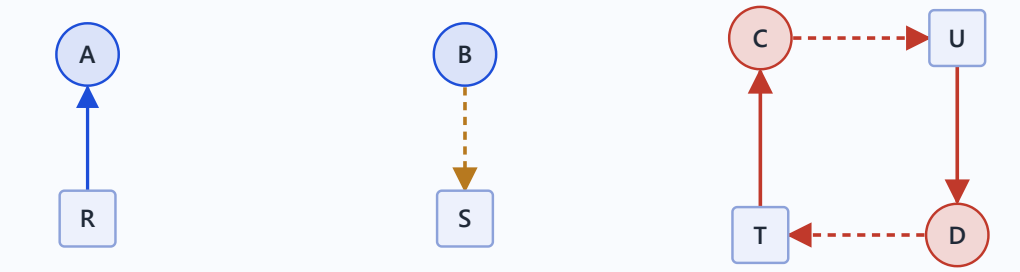
Kilitlenmeyi konuşmak için kimin neyi tuttuğunu ve kimin neyi beklediğini bir bakışta gösteren bir araca ihtiyaç vardır. Holt'un 1972'de önerdiği kaynak tahsis çizgesi bu işi görür. Çizge hem kilitlenmenin nasıl oluştuğunu anlamayı kolaylaştırır hem de tespit algoritmalarının çalıştığı veri yapısıdır.

7.3.1 Kaynak tahsis çizgesi

Tanım 7.3

Kaynak tahsis çizgesi (*resource allocation graph*), iki tür düğümü olan yönlü bir çizgedir. Süreçler daireyle, kaynaklar kareyle çizilir. Kaynaktan sürece giden **atama kenarı** R → P, R'nin P'ye verildiğini gösterir. Süreçten kaynağa giden **istek kenarı** P → R, P'nin R'yi beklediğini gösterir.

Bir istek karşılandığında istek kenarı atama kenarına dönüşür, yani okun yönü ters çevrilir. Kaynak bırakıldığında atama kenarı silinir. Çok örnekli bir kaynak türünde karenin içine her örnek için bir nokta konur. Atama kenarı bu noktalardan birinden çıkar; istek kenarı ise karenin kendisine gider, çünkü süreç belirli bir örneği değil, türün herhangi bir örneğini bekler. Bu notta istek kenarları kesikli ve turuncu çizilir.



(a) A, R'yi tutuyor (b) B, S'yi istiyor (c) Kilitlenme: C → U → D → T → C

Şekil 7.6: Kaynak tahsis çizgesinin gösterimi. (a) R'den A'ya atama kenarı: A, R'yi tutuyor. (b) B'den S'ye istek kenarı: B, S'yi istiyor. (c) C, U'yu bekliyor; U, D'de; D, T'yi bekliyor; T, C'de. Çizgedeki C → U → D → T → C döngüsü kilitlenmedir.

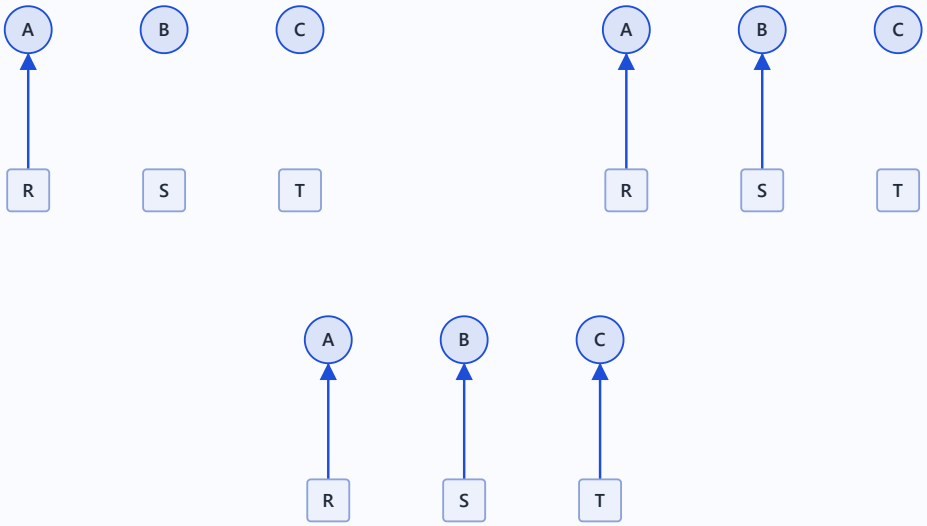
Tanenbaum aynı modeli süreç düğümleriyle çizer. Silberschatz ise düğümleri iş parçacığı olarak adlandırır ve T harfiyle gösterir. İki gösterim arasında anlam farkı yoktur. Bu notta örnekler hangi kaynaktan alındıysa o kaynağın adlandırması korunur.

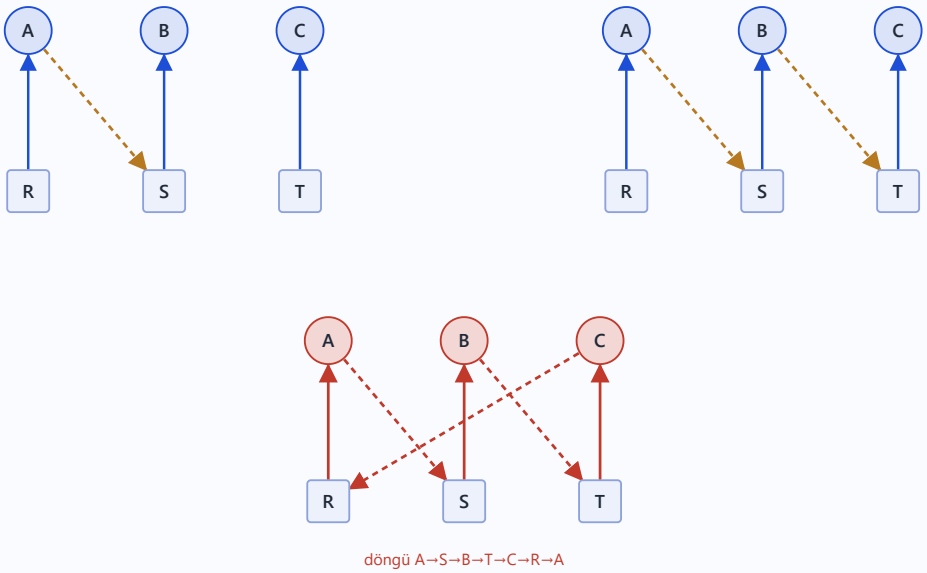
7.3.2 Kilitlenme nasıl oluşur?

Üç süreç (A, B, C) ve üç kaynak (R, S, T) düşünelim. Her süreç iki kaynak ister ve ikisini de bırakır:

```
A        B        C
R iste   S iste   T iste
S iste   T iste   R iste
R bırak  S bırak  T bırak
S bırak  T bırak  R bırak
```

İşletim sistemi süreçleri döner sıralı (*round robin*) çizelgeliyor ve istekleri geldikleri sırayla karşılıyor olsun. Süreçlerin ilk istekleri sırayla gelir ve üç kaynak da boş olduğu için hepsi karşılanır. İkinci istekler geldiğinde her süreç bir sonrakinin elindeki kaynağı ister. Şekil 7.7 altı isteği ve çizgenin her adımdaki hâlini gösterir.





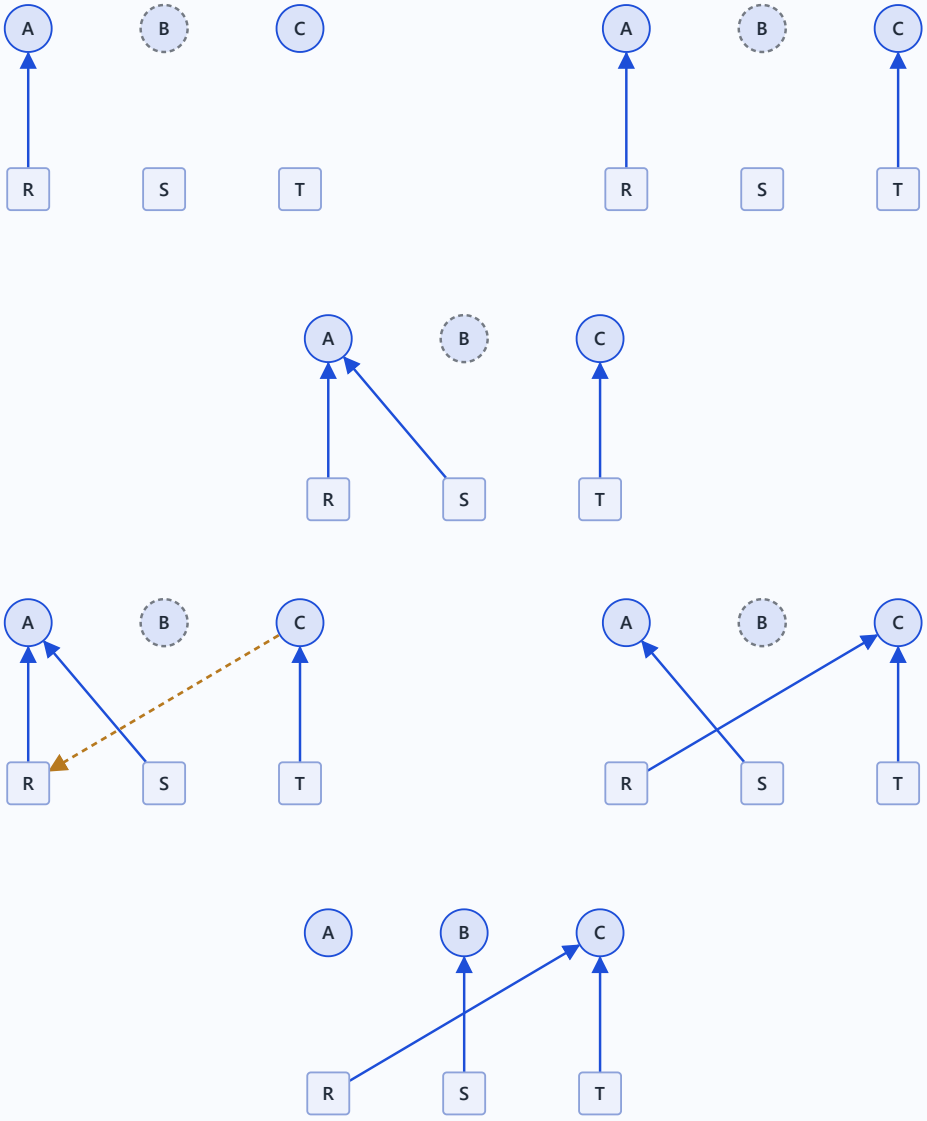
Şekil 7.7: Kilitlenmenin oluşumu (Tanenbaum Şekil 6-4). (a) A, R'yi alır. (b) B, S'yi alır. (c) C, T'yi alır. (d) A, S'yi ister ve bloke olur. (e) B, T'yi ister ve bloke olur. (f) C, R'yi ister ve bloke olur: A → S → B → T → C → R → A döngüsü kapanır.

1. **A, R'yi ister.** R boştur; atama kenarı R → A eklenir.
2. **B, S'yi ister.** S boştur; S → B eklenir.
3. **C, T'yi ister.** T boştur; T → C eklenir.
4. **A, S'yi ister.** S, B'dedir. A bloke olur; istek kenarı A → S eklenir.
5. **B, T'yi ister.** T, C'dedir. B bloke olur; B → T eklenir.
6. **C, R'yi ister.** R, A'dadır. C bloke olur; C → R eklenir. Çizgede A → S → B → T → C → R → A döngüsü vardır ve üç süreç de kilitlenmiştir.

Bu sonuç kaçınılmaz değildi. Çizelgeleyici başka bir sıra seçebilirdi. Örneğin A'yı iki isteğini de yapıp kaynaklarını bırakana kadar çalıştırsaydı, ardından B'yi, sonra C'yi çalıştırsaydı hiçbir istek beklemezdi. Paralellik kaybolurdu ama kilitlenme olmazdı. Kilitlenme, isteklerin sırasına bağlıdır.

7.3.3 Kilitlenme nasıl önlenebilir?

İşletim sistemi bir isteği vermenin kilitlenmeye yol açacağını önceden bilseydi, tehlikeli isteği bekleterek kilitlenmeyi önleyebilirdi. Aynı örnekte işletim sisteminin B'nin isteklerini bir süre askıya aldığını, yani B'yi çizelgelemediğini varsayalım. Yalnız A ve C çalışır. Şekil 7.8 bu sıralamayı izler.



Şekil 7.8: B ertelendiğinde kilitlenme olmaz (Tanenbaum Şekil 6-4). (a) A, R'yi alır. (b) C, T'yi alır. (c) A, S'yi alır. (d) C, R'yi ister ve bloke olur; döngü yoktur. (e) A, R'yi bırakır; R, C'ye verilir. (f) A, S'yi bırakıp biter; artık B çalıştırılabilir.

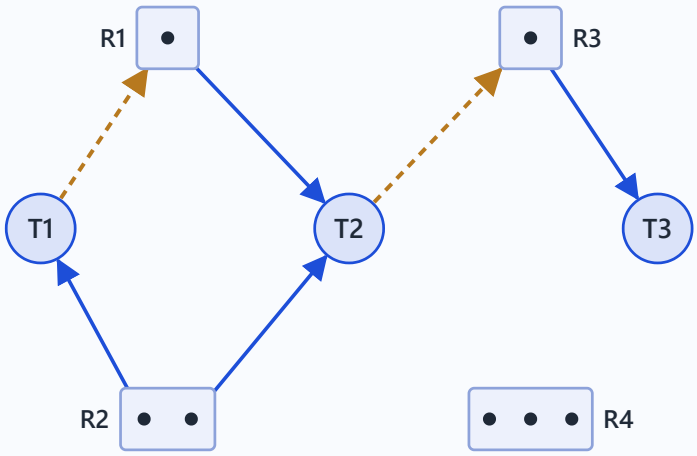
1. **A, R'yi alır; C, T'yi alır.** İki kaynak da boştur.
2. **A, S'yi alır.** S boştur, çünkü B henüz onu istememiştir.
3. **C, R'yi ister.** R, A'dadır; C bloke olur. Çizgede bir istek kenarı vardır ama döngü yoktur. A'nın beklediği hiçbir şey yoktur, ilerleyebilir.
4. **A, R'yi bırakır.** R, bekleyen C'ye verilir ve C ilerler.
5. **A, S'yi bırakıp biter.** Şimdi B çalıştırılır. B, S'yi alır; T'yi de C bitince alır. Kilitlenme olmaz.

İşletim sisteminin burada yaptığı şey kaçınma stratejisinin özüdür: hangi istek ertelenmeli? Ertelenecek isteği seçmek için her sürecin ne kadar kaynak isteyebileceğini bilmek gerekir. Bankacı algoritması bu bilgiyi kullanır.

7.3.4 Çok örnekli kaynaklar

Şimdiye kadarki örneklerde her kaynak türünden tek örnek vardı. Silberschatz'ın klasik örneği çok örnekli türleri de içerir. Kaynak kümeleri şöyledir: R1 bir örnekli, R2 iki örnekli, R3 bir örnekli, R4 üç örnekli. Üç iş parçacığının durumu da şöyledir:

- T1, bir R2 örneğini tutar ve R1'i bekler.
- T2, R1'i ve bir R2 örneğini tutar, R3'ü bekler.
- T3, R3'ü tutar.



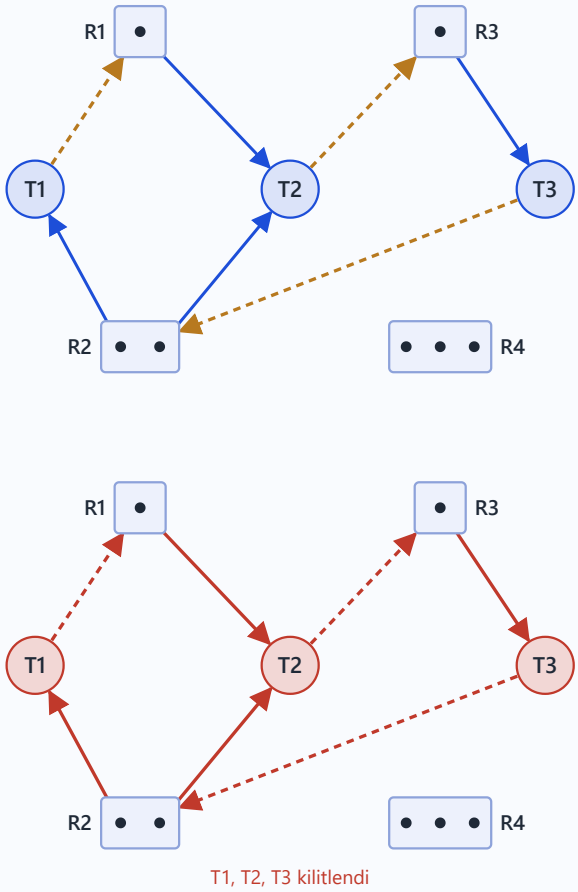
Şekil 7.9: Çok örnekli kaynaklarla tahsis çizgesi (Silberschatz Şekil 8.4). Atama kenarı karedeki noktadan, yani belirli bir örnekten çıkar; istek kenarı karenin kendisine gider. R4'ün üç örneği de boştadır.

Çizgede döngü yoktur. T3 hiçbir şey beklemediği için işini bitirip R3'ü bırakır. R3, T2'ye verilir; T2 de biter ve R1 ile R2 örneğini bırakır. R1, T1'e verilir ve T1 de biter. Bitiş sırası T3, T2, T1'dir. Döngü yoksa kilitlenme de yoktur; bu kural örnek sayısından bağımsızdır.

7.3.5 Döngü ve kilitlenme

Aynı duruma tek bir istek ekleyelim: T3, bir R2 örneği istesin. R2'nin iki örneği de doludur (biri T1'de, biri T2'de); T3 bloke olur. Bu tek istek kenarı çizgede iki döngü oluşturur:

- Döngü 1: T1 → R1 → T2 → R3 → T3 → R2 → T1.
- Döngü 2: T2 → R3 → T3 → R2 → T2.

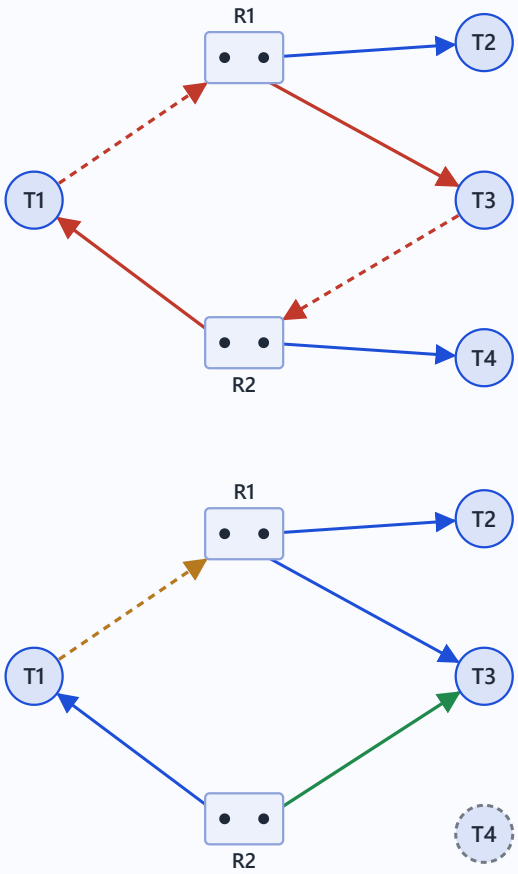


Şekil 7.10: Kilitlenen çok örnekli çizge (Silberschatz Şekil 8.5). (a) T3 bir R2 örneği ister; iki örnek de doludur ve T3 bloke olur. (b) İki döngü oluşur; R2'nin iki örneğinin sahibi de döngüdedir. T1, T2 ve T3 kilitlenmiştir.

Bu durumda kurtarıcı yoktur. T3, R2 örneklerinden birinin bırakılmasını bekler; ama iki örneğin sahibi de (T1 ve T2) döngüdedir. T2, T3'ün R3'ünü; T1, T2'nin R1'ini bekler. Döngüdeki her süreç döngüdeki başka bir süreci bekler ve dışarıdan kaynak bırakacak kimse yoktur. Üç iş parçacığı da kilitlenmiştir.

7.3.6 Döngü var, kilitlenme yok

Çok örnekli kaynaklarda döngü her zaman kilitlenme demek değildir. Silberschatz'ın ikinci örneğinde R1 ve R2 ikişer örneklidir. R1'in iki örneği T2 ve T3'te, R2'nin iki örneği T1 ve T4'tedir. T1 bir R1 örneği, T3 de bir R2 örneği ister. Çizgede T1 → R1 → T3 → R2 → T1 döngüsü vardır.



Şekil 7.11: Döngü var ama kilitlenme yok (Silberschatz Şekil 8.6). (a) T1 → R1 → T3 → R2 → T1 döngüsü; T2 ve T4 döngünün dışındadır ve hiçbir şey beklemez. (b) T4 işini bitirip R2 örneğini bırakır; örnek T3'e verilir ve döngü kırılır.

Fark döngünün dışındaki süreçlerdedir. T2 ve T4 hiçbir şey beklemez; işlerini bitirip örneklerini bırakırlar. T4'ün bıraktığı R2 örneği T3'e verilir, döngü kırılır. T2'nin bıraktığı R1 örneği de T1'e verilebilir. Sonunda herkes biter; sistem hiç kilitlenmemiştir.

Örnek 7.1

R1'in iki, R2 ile R3'ün birer örneği var. P1, R2'yi tutup R1'i istiyor. P2, bir R1 örneği tutup R3'ü istiyor. P3, R3'ü tutup R2'yi istiyor. P4, öteki R1 örneğini tutuyor ve bir şey istemiyor. Çizgede döngü var mı? Sistem kilitlenmiş mi?

1. **Döngü.** P1 → R1 → P2 → R3 → P3 → R2 → P1 bir döngüdür. Kaynaklar tek örnekli olsaydı bu kesin kilitlenme olurdu.
2. **Döngü dışındaki örnek.** R1'in ikinci örneği döngüde olmayan P4'tedir. P4 hiçbir şey beklemez, biter ve R1 örneğini bırakır.
3. **Zincirleme bitiş.** P1'in R1 isteği karşılanır; P1 biter, R1 ve R2'yi bırakır. P3, R2'yi alır ve biter; R2 ile R3 boşalır. P2, R3'ü alıp biter.

4. **Sonuç.** Bitiş sırası P4, P1, P3, P2'dir. Herkes bitebildiği için kilitlenme yoktur.

7.3.7 Döngü kuralı

Örneklerden çıkan sonuç Tablo 7.4'te özetlenmiştir. Döngü yoksa kilitlenme hiçbir durumda yoktur. Döngü varsa sonuç örnek sayısına bağlıdır.

Çizge	Her türde tek örnek	Bazı türlerde birden çok örnek
Döngü yok	kilitlenme yok	kilitlenme yok
Döngü var	kilitlenme var (döngü ⇔ kilitlenme)	kilitlenme olabilir; döngü gerekli ama yeterli değil

Tablo 7.4: Tahsis çizgesindeki döngü ile kilitlenme arasındaki ilişki.

Teorem 7.2 (Döngü kuralı)

Kaynak tahsis çizgesinde döngü yoksa sistem kilitlenmemiştir. Her kaynak türünün tek örneği varsa döngü kilitlenmenin hem gerekli hem yeterli koşuludur ve döngüdeki süreçler kilitlenmiştir. Bazı türlerin birden çok örneği varsa döngü gereklidir ama yeterli değildir.

İspat. Tek örnekli durumda döngüdeki her kaynağın tek sahibi vardır ve o sahip döngüdeki başka bir kaynağı bekler. Kaynağı bırakabilecek tek süreç beklediği için kaynak hiç boşalmaz; döngüdeki süreçlerin hiçbiri ilerleyemez. Çok örnekli durumda döngüdeki bir kaynağın başka bir örneği döngü dışındaki bir süreçte olabilir. O süreç bitip örneği bırakınca döngüdeki istek karşılanır. Bu yüzden döngü kilitlenmeyi garanti etmez.

Tek örnekli kaynaklarda döngüyü bulmak için derinlik öncelikli arama yeterlidir ve maliyeti $O(V + E)$'dir (V düğüm, E kenar sayısı). Çok örnekli kaynaklarda döngü yalnız bir uyarıdır. Kesin karar için tahsis ve istek matrisleriyle "herkes bir sırayla bitebilir mi?" sorusu sorulur. Teorem dört koşulla da tutarlıdır: döngüsel bekleme, döngü içeren çizgenin karşılığıdır.

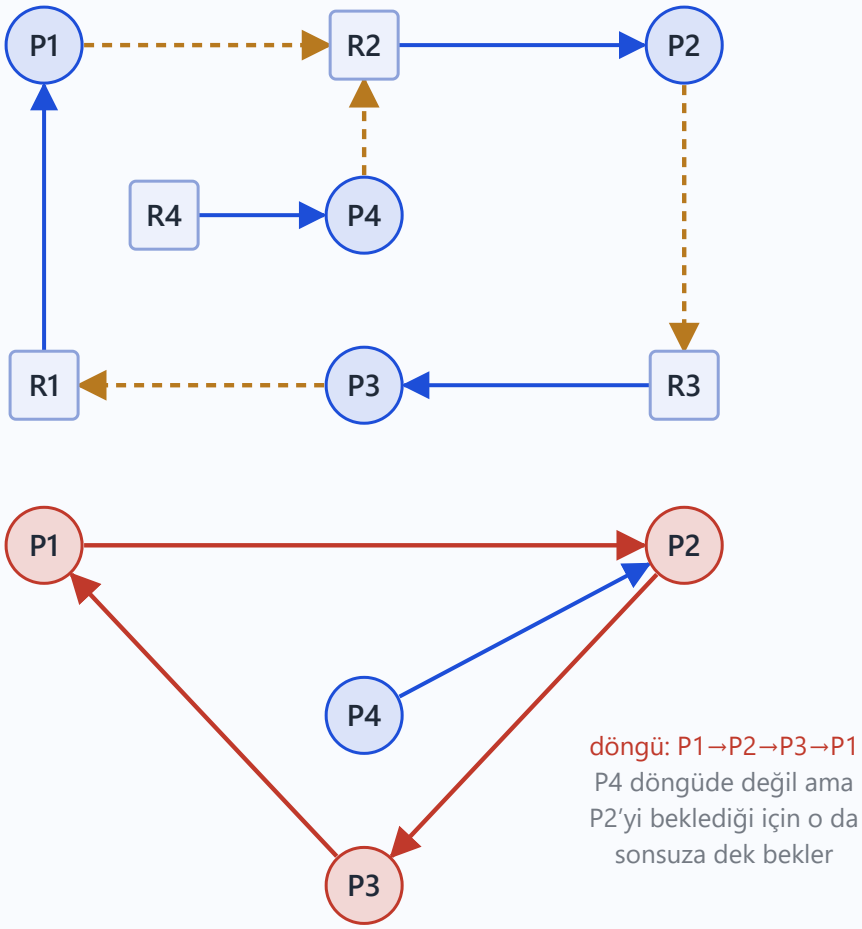
7.3.8 Bekleme çizgesi

Her kaynak türünün tek örneği varsa tahsis çizgesi daha da sadeleştirilebilir. Kaynağın tek bir sahibi olduğu için "P_i, R'yi bekliyor ve R, P_j'de" bilgisi kısaca "P_i, P_j'yi bekliyor" demektir.

Tanım 7.4

Bekleme çizgesi (*wait-for graph*), yalnız süreç düğümlerinden oluşan yönlü çizgedir. Tahsis çizgesindeki her $P_i \to R \to P_j$ yolu tek bir $P_i \to P_j$ kenarına daraltılır; kaynak düğümleri silinir.

Şekil 7.12 dönüşümü dört süreçli bir örnekte gösterir. P1, R1'i tutar ve R2'yi ister. P2, R2'yi tutar ve R3'ü ister. P3, R3'ü tutar ve R1'i ister. P4 de R4'ü tutar ve R2'yi ister.



Şekil 7.12: Tahsis çizgesinden bekleme çizgesine. Üstte tek örnekli kaynaklarla tahsis çizgesi; altta kaynak düğümleri silindikten sonraki bekleme çizgesi. P1 → P2 → P3 → P1 döngüsü kilitlenmedir; P4 döngüde değildir ama P2'yi beklediği için o da sonsuza dek bekler.

Kenarlar şöyle çıkar: P1, R2'yi bekler ve R2, P2'dedir; dolayısıyla P1 → P2. Aynı yolla P2 → P3, P3 → P1 ve P4 → P2 bulunur. Bekleme çizgesinde döngü ancak ve ancak kilitlenme varsa oluşur. İşletim sistemi bu çizgeyi tutar ve döngü arayan algoritmayı belirli aralıklarla çalıştırır. Silberschatz bu taramanın maliyetini $O(n^2)$ olarak verir (n: süreç sayısı).

Bekleme çizgesi yalnız her türden tek örnek olduğunda kullanılabilir; çok örnekli bir türde "R'nin sahibi" tek bir süreç değildir. Veritabanları bu yöntemi yaygın olarak kullanır: PostgreSQL ve MySQL InnoDB, satır kilitleri için bir bekleme çizgesi tutar ve döngü bulduğunda işlemlerden birini iptal eder.

Sık yapılan hata

Döngüde olmayan süreçlerin her zaman ilerleyebileceğini sanmak. Bekleme çizgesinde P4 döngüde değildir, ama beklediği P2 döngüdedir. P2 hiç bitmeyeceği için P4 de hiç ilerleyemez. Kilitlenmiş küme yalnız döngüdeki süreçlerdir; döngüye bir yolla bağlı olan süreçler ise kilit-

lenmenin dolaylı kurbanlarıdır.

7.4 Başa çıkma stratejileri

Kilitlenmeyle başa çıkmanın dört yolu vardır. Yollar birbirinden şu sorularla ayrılır: Sistem kilitlenmeye hiç izin verir mi? Karar ne zaman verilir, tasarımda mı, çalışma anında mı, kilitlenme olduktan sonra mı? Karar için hangi bilgi gerekir?

7.4.1 Dört strateji

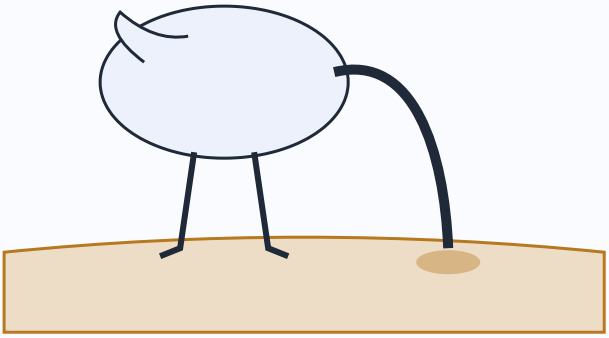
1. **Görmezden gelme.** "Devekuşu algoritması" da denir. Kilitlenme nadirse ve önleme maliyeti yüksekse sorun yok sayılır; kilitlenme olursa sistem ya da süreç yeniden başlatılır.
2. **Tespit ve kurtarma.** Tepkisel bir yaklaşımdır. Kilitlenmeye izin verilir; sistem belirli aralıklarla kilitlenme olup olmadığını denetler ve bulursa kaynak kesme, geri sarma ya da sonlandırma ile çözer.
3. **Dinamik kaçınma.** Önleyicidir ve çalışma anında karar verir. Sistemi güvensiz bir duruma sokacak istek ertelenir. Bankacı algoritması bu yaklaşımın örneğidir. Her sürecin azami ihtiyacının önceden bilinmesini gerektirir.
4. **Önleme.** Önleyicidir ve kararı tasarımda verir. Dört koşuldan en az biri yapısal olarak imkânsız kılınır; kilit sıralaması en bilinen örnektir.

Listede "kurtarma" ayrı bir strateji olarak geçmez, çünkü kurtarma tespitin ikinci yarısıdır. Önce kilitlenme bulunur, sonra çözülür. Kaçınma ile önleme arasındaki fark da sık karıştırılır. Önleme, kilitlenmenin koşullarından birini sistemden tümüyle kaldırır; koşullar zaten sağlanamadığı için kilitlenme imkânsızdır. Kaçınma ise dört koşulun hepsine izin verir, ama her istekte durumun gidişatını hesaplar ve tehlikeli istekleri bekletir.

7.4.2 Devekuşu algoritması

En basit strateji hiçbir şey yapmamaktır. Adını, tehlike karşısında başını kuma gömdüğüne inanılan devekuşundan alır. Kilitlenme olursa kullanıcı ya da yönetici takılan süreci öldürür, gerekirse sistemi yeniden başlatır.

"Kilitlenme mi? Ne kilitlenmesi?"



Şekil 7.13: Devekuşu algoritması: sorunu görmezden gelmek.

İlk bakışta sorumsuz görünse de bu seçim bir mühendislik dengesidir. Kilitlenme yılda bir kez oluyorsa, buna karşılık çökmeler ve donanım hataları çok daha sık görülüyorsa, pahalı ve kısıtlayıcı önlemler değmez. Her istekte çalışan bir kaçınma algoritması bütün sistemi yavaşlatır; kaynak isteklerine konan kurallar da programcının işini zorlaştırır.

Linux, Windows ve macOS gibi genel amaçlı işletim sistemleri kullanıcı süreçlerinin kilitleri, dosya kilitleri ve boru beklemeleri için bu yaklaşımı uygular. Sorumluluk uygulama geliştiricisindedir. Ama işletim sistemi kendi içinde kilitlenmeye karşı önlem alır: çekirdekteki kilitler belgelenmiş bir sırayla alınır ve Linux'taki `lockdep` aracı bu sırayı denetler. Gerçek zamanlı ve güvenlik açısından kritik sistemlerde devekuşu yaklaşımı kabul edilemez; bir uçuş denetim yazılımı "ara sıra yeniden başlatılır" diye tasarlanamaz.

Devekuşu yaklaşımı tespit ile kaçınma arasında bir orta yol değildir. Ne tespit ne kaçınma yapar; yalnızca kilitlenmenin olasılığını ve maliyetini önlemin maliyetiyle karşılaştırır.

7.5 Tespit ve kurtarma

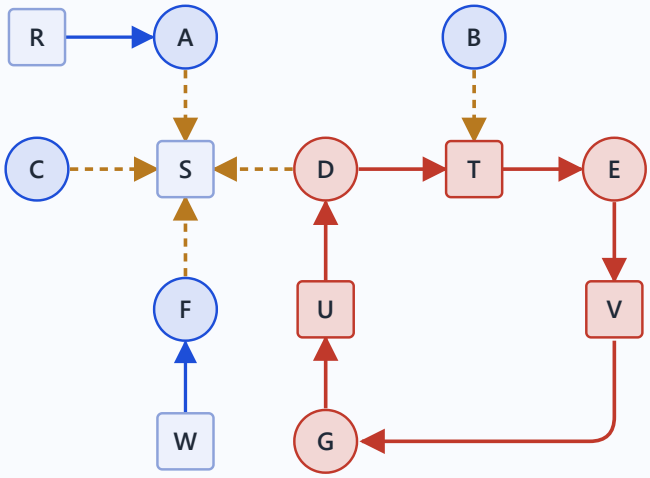
Tespit ve kurtarma stratejisinde sistem kaynak isteklerine kısıt koymaz. İstekler geldikçe karşılanır, kaynak boş değilse süreç bekletilir. Sistem zaman zaman bir algoritma çalıştırarak kilitlenme olup olmadığını denetler. Algoritma kaynak türlerinin tek ya da çok örnekli olmasına göre değişir.

7.5.1 Her türden tek kaynak

Her kaynak türünün tek örneği varsa kilitlenme, tahsis çizgesindeki döngüyle aynı şeydir. Tanenbaum'un örneğinde yedi süreç (A–G) ve altı kaynak (R–W) vardır:

- A, R'yi tutar ve S'yi ister.
- B hiçbir şey tutmaz ama T'yi ister.

- C hiçbir şey tutmaz ama S'yi ister.
- D, U'yu tutar; S'yi ve T'yi ister.
- E, T'yi tutar ve V'yi ister.
- F, W'yi tutar ve S'yi ister.
- G, V'yi tutar ve U'yu ister.



Şekil 7.14: Yedi süreç ve altı kaynakla tahsis çizgesi (Tanenbaum Şekil 6-5). Vurgulanan D → T → E → V → G → U → D döngüsü kilitlenmedir.

Soru şudur: sistem kilitlenmiş mi, kilitlendiyse hangi süreçler? Çizge çizildikten sonra cevap göz ile bulunabilir. D → T → E → V → G → U → D bir döngüdür; D, E ve G kilitlenmiştir. A, C ve F kilitli değildir. S boştadır; üçünden biri S'yi alır, işini bitirip bırakır, sonra öteki alır. B'nin durumu ise farklıdır: B döngüde değildir ama döngüdeki T'yi beklediği için o da ilerleyemez. Bu küçük çizgede göz yeter. Yüzlerce düğümlü bir çizgede ise sistematik bir algoritma gerekir.

7.5.2 Döngü arama algoritması

Tanenbaum'un verdiği algoritma çizgedeki her düğümü sırayla başlangıç noktası yapar ve o düğümden derinlik öncelikli ilerler. Gidilen yol bir *L* listesinde tutulur. Bir düğüm *L*'ye ikinci kez eklenirse döngü bulunmuştur.

1. Çizgedeki her *N* düğümü için 2–6. adımları *N*'den başlayarak uygula.
2. *L*'yi boş liste yap; bütün kenarları işaretsiz say.
3. Geçerli düğümü *L*'nin sonuna ekle. Düğüm *L*'de iki kez geçiyorsa çizgede döngü vardır: *L*'deki döngüyü raporla ve dur.
4. Geçerli düğümden çıkan işaretsiz bir kenar var mı? Varsa 5. adıma, yoksa 6. adıma git.
5. İşaretsiz kenarlardan birini seç, işaretle ve gösterdiği düğüme geç; 3. adıma dön.

6. Başlangıç düğümündeysek *N*'den erişilebilen bir döngü yoktur. Değilsek çıkmaz sokaktayız: düğümü *L*'den çıkar, önceki düğüme dön ve 4. adımdan devam et.

Algoritmanın özü her düğümden başlayan bir derinlik öncelikli aramadır (*depth-first search*, DFS); yığın görevini *L* listesi görür. Bu basit sürüm her düğümden yeniden başladığı için en kötü durumda $O(V \cdot (V + E))$ zaman alır. Düğümleri üç renge boyayan sürüm ise tek geçişte $O(V + E)$'de çalışır. Çıktıda *L*'nin tekrar eden düğümden sonraki kısmı döngüdür ve içindeki süreçler kilitlenmiştir.

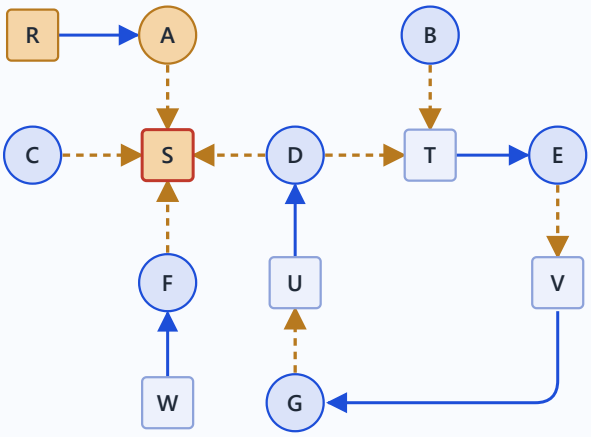
Altıncı adımda "önceki düğüme dön" denildiğinde o düğüm zaten *L*'dedir. Bu yüzden 3. adıma değil 4. adıma gidilir; 3. adıma gidilseydi düğüm ikinci kez eklenir ve sahte bir döngü raporlanırdı. Tanenbaum'un metni bu adımda kısaca "3. adıma git" der; anlamı, düğümü yeniden eklemeden çıkış kenarlarına bakmaktır.

7.5.3 Döngü arama adım adım

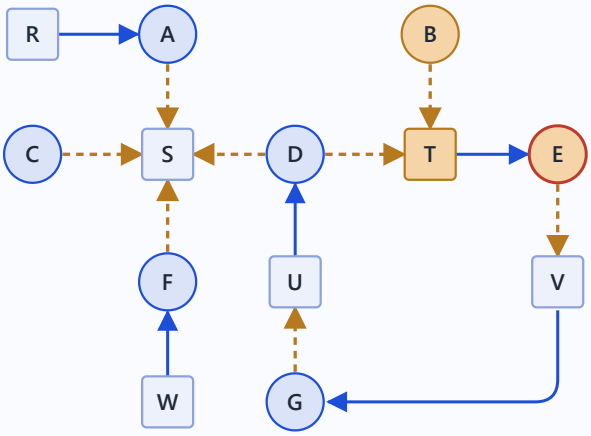
Algoritmayı Şekil 7.14'teki çizgede çalıştıralım. Başlangıç düğümleri şekildeki sırayla, yani soldan sağa ve yukarıdan aşağı seçilir: R, A, B, C, S, D, T, E, F, U, V, W, G. Bir düğümden birden çok kenar çıkıyorsa seçim keyfidir; burada D'den önce S'ye, sonra T'ye giden kenar seçilir. Tablo 7.5 *L*'nin her adımdaki hâlini verir.

Başlangıç	Olay	*L*
R	R eklenir; tek çıkış R → A	[R]
R	A eklenir; A → S	[R, A]
R	S eklenir; S'den çıkan kenar yok	[R, A, S]
R	çıkmaz sokak: S çıkar, A'ya dönülür; A'nın işaretsiz kenarı yok	[R, A]
R	A çıkar, R'ye dönülür; R başlangıçtır: R'den döngü yok	[R] → []
A	A → S, S çıkmaz sokak; A'dan döngü yok	[A] → [A, S] → []
B	B → T → E	[B, T, E]
B	E → V → G → U	[B, T, E, V, G, U]
B	U → D; D'den önce S denenir	[B, T, E, V, G, U, D, S]
B	S çıkmaz sokak, L'den çıkar; D'ye dönülür	[B, T, E, V, G, U, D]
B	D → T: T ikinci kez eklenir → döngü	[B, T, E, V, G, U, D, T]

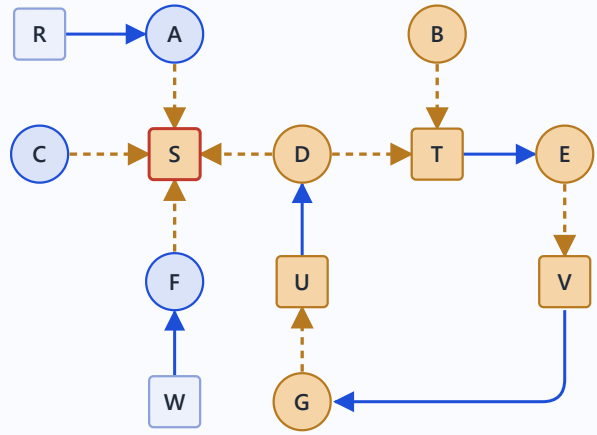
Tablo 7.5: Döngü arama algoritmasının izlemesi.



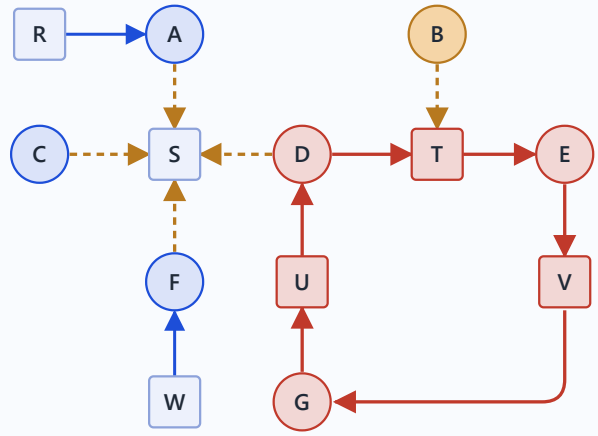

```
L = [R, A, S]
```




```
L = [B, T, E]
```




```
L = [B, T, E, V, G, U, D, S]
```




```
L = [B, T, E, V, G, U, D, T]
```

Şekil 7.15: Döngü aramanın dört anı. Turuncu düğümler o anki L listesinde, kırmızı çerçeveli düğüm geçerli düğümdür. (a) R → A → S çıkmaz sokağa varır. (b) B'den başlanır: B → T → E. (c) U → D'den sonra S denenir. (d) D → T ile T ikinci kez eklenir: döngü T, E, V, G, U, D.

Algoritma B'den başlayan aramada durur. L = [B, T, E, V, G, U, D, T] listesinde T iki kez geçer. Tekrar eden düğümün ilk geçişinden sonraki kısım, yani T, E, V, G, U, D döngüdür. Döngüdeki süreçler D, E ve G'dir; bunlar kilitlenmiştir. B listenin başındadır ama döngüde değildir: döngüye giden bir yolun üzerindedir. Çizgeyi göz ile incelerken bulduğumuz sonucun aynısı çıkar. Bu sonuçlar Python'da aynı algoritmanın yeniden çalıştırılmasıyla doğrulanmıştır.

7.5.4 Kod izleme: renkli derinlik öncelikli arama

Her düğümden yeniden başlamak gereksiz iş yapar. Renkli DFS her düğüme bir renk verir. **Beyaz** düğüm henüz ziyaret edilmemiştir. **Gri** düğüm şu an yığındadır, yani gidilen yolun üzerindedir. **Siyah** düğümün ise bütün komşuları işlenmiş ve düğüm bitmiştir. Gri bir düğüme giden kenara **geri kenar** denir ve döngü olduğunu gösterir. Aşağıdaki kod bekleme çizgesini `bekler[u][v]` komşuluk matrisiyle tutar: `bekler[u][v]` 1 ise u, v'yi bekliyordur.

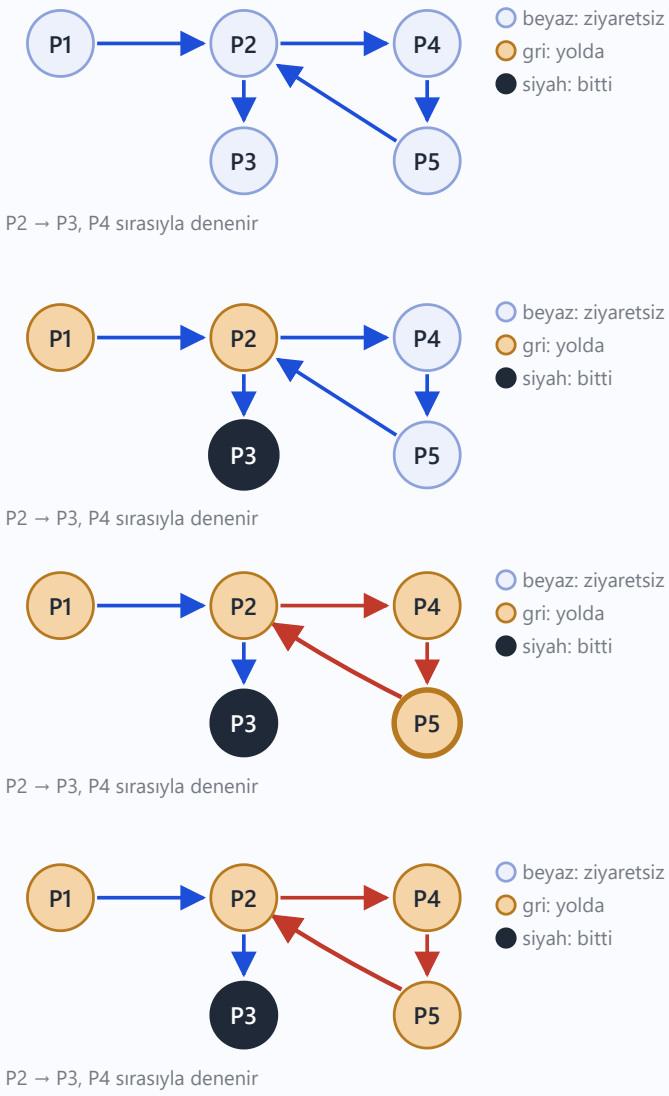
```
enum { BEYAZ, GRI, SIYAH };
int renk[N];          /* hepsi BEYAZ */
bool dongu(int u) {
  renk[u] = GRI;      /* yolda */
  for (int v = 0; v < N; v++) {
    if (!bekler[u][v]) continue;
    if (renk[v] == GRI)
      return true;    /* geri kenar */
    if (renk[v] == BEYAZ && dongu(v))
      return true;
  }
  renk[u] = SIYAH;    /* bitti */
  return false;
}
```

Kodun mantığı şöyledir. `dongu(u)` çağrıldığında u griye boyanır. Ardından u'nun beklediği her v denenir. v griyse yol kendi üstüne dönmüştür ve `true` döner. v beyazsa `dongu(v)` özyinelemeli olarak çağrılır. O çağrı döngü bulursa `true` yukarı taşınır. Siyah komşular atlanır, çünkü onlardan döngü çıkmadığı zaten bilinir. Bütün komşular bitince u siyaha boyanır ve `false` döner. Her düğüm en çok bir kez griye boyanır ve her kenar bir kez incelenir; maliyet $O(V + E)$'dir. Komşuluk matrisiyle kenarları bulmak her düğümde N adım sürdüğü için bu kodun gerçek maliyeti $O(N^2)$'dir.

Çizgede beş süreç vardır ve kenarlar şunlardır: P1 → P2, P2 → P3, P2 → P4, P4 → P5 ve P5 → P2. Arama P1'den başlar. `for` döngüsü komşuları numara sırasıyla denediği için P2'den önce P3'e, sonra P4'e gidilir. Tablo 7.6 her adımdaki renkleri, Şekil 7.16 da önemli anları gösterir.

Adım	Olay	P1	P2	P3	P4	P5	Çağrı yığını
0	başlangıç	B	B	B	B	B	—
1	`dongu(P1)` : P1 gri	G	B	B	B	B	P1
2	P1 → P2 beyaz: `dongu(P2)`	G	G	B	B	B	P1, P2
3	P2 → P3 beyaz: `dongu(P3)`	G	G	G	B	B	P1, P2, P3
4	P3'ün komşusu yok: siyah, `false`	G	G	S	B	B	P1, P2
5	P2 → P4 beyaz: `dongu(P4)`	G	G	S	G	B	P1, P2, P4
6	P4 → P5 beyaz: `dongu(P5)`	G	G	S	G	G	P1, P2, P4, P5
7	P5 → P2: P2 gri → geri kenar, `true`	G	G	S	G	G	P1, P2, P4, P5
8	`true` yığın boyunca yukarı taşınır	G	G	S	G	G	—

Tablo 7.6: Renkli DFS izlemesi (B: beyaz, G: gri, S: siyah).



Şekil 7.16: Renkli DFS. (a) Başlangıçta bütün düğümler beyazdır. (b) P3 kimseyi beklemez ve siyaha boyanır. (c) P5 → P2 kenarında P2 gridir: geri kenar, yani döngü. (d) Döngüdeki P2, P4 ve P5 kilitlenmiştir; P1 döngüde değildir ama P2'yi beklediği için takılıdır.

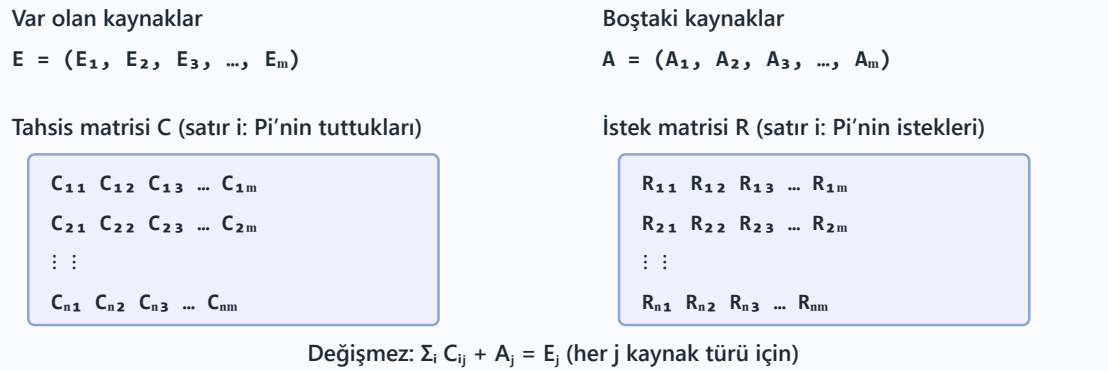
Döngü P2 → P4 → P5 → P2'dir ve bu üç süreç kilitlenmiştir. Gri düğümler o anda yığındaki yolu oluşturur; döngü, geri kenarın gösterdiği gri düğümden (P2) yığının tepesine (P5) kadar olan kısımdır. P1 de gridir, ama geri kenar P1'e değil P2'ye gittiği için P1 döngüde değildir. P1, P2'yi beklediği için yine de ilerleyemez.

Sık yapılan hata

Siyah bir komşuya giden kenarı da döngü saymak. Siyah düğüm daha önce bitmiştir ve ondan çıkan yollar incelenip döngüsüz bulunmuştur. Siyah düğüme giden kenar yalnızca iki yolun birleştiğini gösterir. Örneğin P1 → P3 kenarı da olsaydı, P3 siyahken bu kenar döngü oluşturmazdı. Döngünün işareti yalnız gri düğüme giden kenardır.

7.5.5 Her türden çok kaynak: vektörler ve matrisler

Kaynak türlerinin birden çok örneği varsa döngü aramak yetmez. Bunun yerine tahsis ve istek bilgisi matrislerle tutulur ve “herkes bir sırayla bitebilir mi?” sorusu sorulur. *n* süreç ve *m* kaynak türü olsun. Algoritma dört veri yapısı kullanır.



Şekil 7.17: Tespit algoritmasının veri yapıları (Tanenbaum Şekil 6-6): var olan kaynaklar *E*, boştaki kaynaklar *A*, tahsis matrisi *C* ve istek matrisi *R*. Her *j* türü için tutulan ve boştaki örneklerin toplamı var olan örnek sayısına eşittir.

- ***E*** (var olan kaynaklar): E_j, *j* türünden sistemde toplam kaç örnek olduğudur.
- ***A*** (boştaki kaynaklar): A_j, *j* türünden şu an boşta kaç örnek olduğudur.
- ***C*** (tahsis matrisi): *C*'nin *i*. satırı P_i'nin şu an tuttuğu kaynaklardır.
- ***R*** (istek matrisi): *R*'nin *i*. satırı P_i'nin şu an istediği ve beklediği kaynaklardır.

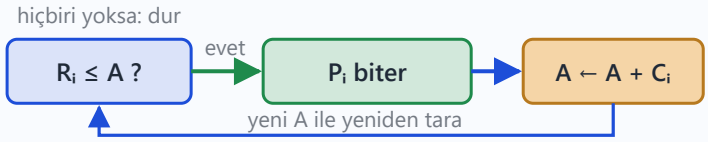
Bu yapılar arasında değişmeyen bir bağıntı vardır: her *j* türü için $\Sigma_i C_{ij} + A_j = E_j$. Bir kaynak örneği ya bir sürecin elindedir ya da boştadır. Silberschatz aynı yapıları *Available*, *Allocation* ve *Request* adlarıyla verir.

Algoritma satırları vektör olarak karşılaştırır. İki vektör için $X \le Y$ ancak ve ancak her *j* için $X_j \le Y_j$ ise doğrudur. Örneğin (1 0 1 0) ≤ (2 1 0 0) yanlıştır: ilk iki bileşen uyar ama üçüncü bileşende 1 > 0'dır. Vektörlerde “küçük” kavramı her bileşende ayrı ayrı sağlanmalıdır; toplamları karşılaştırmak yanlış sonuç verir.

7.5.6 Tespit algoritması

Algoritmanın fikri basittir. İsteği şu anki boş kaynaklarla karşılanabilen bir süreç çalışıp bitebilir. Bitince tuttuğu bütün kaynakları boş havuza geri verir; havuz büyüdüğü için başka bir sürecin isteği de karşılanabilir hâle gelebilir. Bu, hiçbir süreç kalmayana ya da kalanların hiçbiri karşılanamayana kadar tekrarlanır.

```
A ← boş kaynak vektörü
tüm Pi işaretsiz
tekrarla:
  Ri ≤ A olan işaretsiz bir Pi bul
  bulunamazsa: çık
  A ← A + Ci   /* Pi biter, bırakır */
  Pi'yi işaretle
işaretsiz Pi'ler kilitlenmiştir
```



Şekil 7.18: Tespit döngüsü. İsteği boş vektörle karşılanan süreç biter ve tuttuklarını A'ya ekler; yeni A ile tarama yeniden yapılır. Karşılanabilen satır kalmayınca algoritma durur.

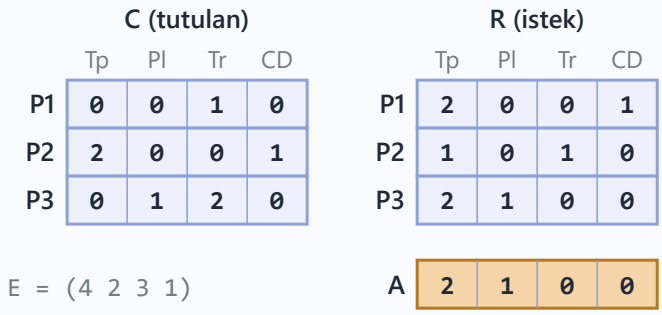
"$A \leftarrow A + C_i$" satırı, biten sürecin tuttuğu kaynakların boş havuza döndüğünü ifade eder. Algoritma sonunda işaretsiz kalan süreçler kilitlenmiştir. Bu sonuç iyimser bir varsayıma dayanır: isteği karşılanan süreç ek kaynak istemeden bitecek sayılır. Süreç gerçekte bitmeden yeni bir istek yaparsa ve o istek kilitlenmeye yol açarsa, durum algoritmanın bir sonraki çalıştırılmasında yakalanır.

Maliyet şöyle hesaplanır. Her turda en çok n satır taranır ve her satır karşılaştırması m bileşenlidir. Her başarılı turda bir süreç işaretlendiği için en çok n tur olur. Toplam maliyet $O(m \cdot n^2)$'dir.

7.5.7 Tespit örneği

Örnek 7.2

Dört kaynak türü vardır: teyp (Tp), çizici (Pl), tarayıcı (Tr) ve CD sürücüsü. Var olan kaynaklar E = (4 2 3 1), boştakiler A = (2 1 0 0)'dır. Tahsis ve istek matrisleri aşağıdaki şekildedir (Tanenbaum Şekil 6-7). Sistem kilitlenmiş mi?



Şekil 7.19: Tespit örneğinin başlangıç durumu: tutulan kaynaklar C, istekler R ve boş vektör A.

1. **Denetim.** C'nin sütun toplamları (2 1 3 1)'dir; E − (2 1 3 1) = (2 1 0 0) = A. Veriler tutarlıdır.
2. **P1.** R_1 = (2 0 0 1) ≤ (2 1 0 0)? CD bileşeninde 1 > 0: boş CD yoktur. P1 şimdilik bitemez.
3. **P2.** R_2 = (1 0 1 0) ≤ (2 1 0 0)? Tarayıcı bileşeninde 1 > 0: boş tarayıcı yoktur.
4. **P3.** R_3 = (2 1 0 0) ≤ (2 1 0 0) ✓. P3 çalışıp bitebilir. Bitince C_3 = (0 1 2 0) geri gelir: A = (2 1 0 0) + (0 1 2 0) = (2 2 2 0). P3 işaretlenir.
5. **Yeni tur, P1.** (2 0 0 1) ≤ (2 2 2 0)? CD hâlâ 0'dır; P1 yine bekler.
6. **P2.** (1 0 1 0) ≤ (2 2 2 0) ✓. P2 biter ve C_2 = (2 0 0 1) döner: A = (4 2 2 1).
7. **Yeni tur, P1.** (2 0 0 1) ≤ (4 2 2 1) ✓. P1 biter ve C_1 = (0 0 1 0) döner: A = (4 2 3 1) = E.
8. **Sonuç.** Bütün süreçler işaretlendi; sistem kilitlenmemiştir. Bulunan bitiş sırası P3, P2, P1'dir.

Örnekte her işaretlemeden sonra tarama baştan başlar. Bu yüzden toplam altı karşılaştırma yapılır: ilk turda üç, ikinci turda iki, üçüncü turda bir. Bulunan sıra, algoritmanın satırları hangi sırayla taradığına bağlıdır; ama kilitlenme kararı değişmez. Bir sürecin bitmesi boş havuzu yalnızca büyütür, hiçbir zaman küçültmez. Bu yüzden bir aşamada karşılanabilen bir istek sonraki aşamalarda da karşılanabilir. Hangi sırayla taranırsa taransın, işaretlenebilen bütün süreçler sonunda işaretlenir. Tablo 7.7 A vektörünün değişimini özetler; bütün değerler Python'da algoritmanın kendisiyle yeniden hesaplanmıştır.

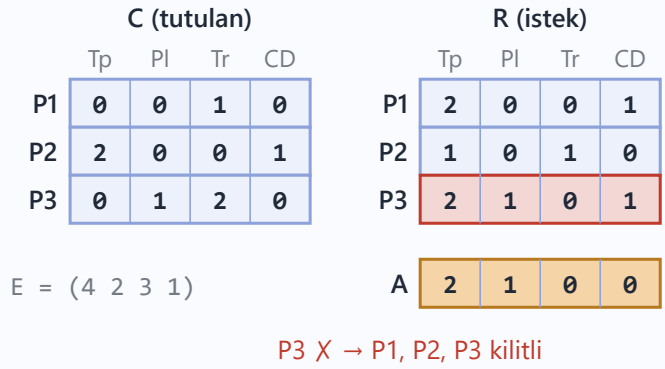
Tur	Karşılaştırma	Sonuç	A sonrası
1	P1: (2 0 0 1) ≤ (2 1 0 0)	✗ (CD)	(2 1 0 0)
1	P2: (1 0 1 0) ≤ (2 1 0 0)	✗ (tarayıcı)	(2 1 0 0)
1	P3: (2 1 0 0) ≤ (2 1 0 0)	✓ P3 biter	(2 2 2 0)
2	P1: (2 0 0 1) ≤ (2 2 2 0)	✗ (CD)	(2 2 2 0)
2	P2: (1 0 1 0) ≤ (2 2 2 0)	✓ P2 biter	(4 2 2 1)

Tur	Karşılaştırma	Sonuç	*A* sonrası
3	P1: (2 0 0 1) ≤ (4 2 2 1)	✓ P1 biter	(4 2 3 1)

Tablo 7.7: Tespit örneğinde boş vektörün değişimi.

7.5.8 Bir istek fazlası: kilitlenme

Aynı durumda tek bir hücreyi değiştirelim. P3 iki teyp ve bir çiziciye ek olarak bir CD de istesin: R_3 = (2 1 0 1). Boş vektör yine A = (2 1 0 0)'dır.



Şekil 7.20: P3'ün isteği (2 1 0 1) olunca hiçbir satır A = (2 1 0 0) ile karşılanamaz; P1, P2 ve P3 kilitlenmiştir.

1. **P1 ✗.** CD gerekir, boşta CD yoktur.
2. **P2 ✗.** Tarayıcı gerekir, boşta tarayıcı yoktur.
3. **P3 ✗.** CD gerekir. Tek CD P2'dedir ve P2 bitemediği için hiç boşalmayacaktır.

İlk turda hiçbir satır karşılanamaz ve algoritma hemen biter. Üç süreç de işaretsiz kalır; üçü de kilitlenmiştir. Tek bir hücrenin değişmesi sonucu tümüyle değiştirmiştir. Algoritma döngü aramaz, ama kilitlenmiş süreçleri doğrudan verir; bu süreçler arasındaki bekleme ilişkisi kendiliğinden bir döngü oluşturur. P3, P2'deki CD'yi; P2, P1'deki tarayıcıyı ya da P3'teki tarayıcıları; P1 de P2'deki CD'yi bekler.

Örnek 7.3

Üç kaynak türü vardır ve E = (5 3 4)'tür. Tahsisler P1 (1 0 1), P2 (2 1 0), P3 (0 1 2), P4 (1 1 0); istekler P1 (0 1 0), P2 (1 0 1), P3 (2 0 1), P4 (0 0 2)'dir. Satırlar P1'den başlayarak, kaldığı yerden sırayla taranıyor. Sistem kilitlenmiş mi? P2'nin isteği (1 1 1) olsaydı ne olurdu?

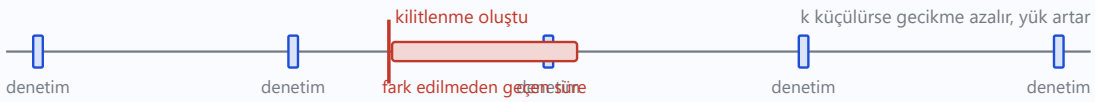
1. **Boş vektör.** Tahsislerin toplamı (4 3 3)'tür; A = (5 3 4) − (4 3 3) = (1 0 1).
2. **P1.** (0 1 0) ≤ (1 0 1) değildir; ikinci türden boş örnek yoktur.
3. **P2.** (1 0 1) ≤ (1 0 1) ✓ → A = (1 0 1) + (2 1 0) = (3 1 1).
4. **P3.** (2 0 1) ≤ (3 1 1) ✓ → A = (3 2 3).

5. **P4.** $(0\ 0\ 2) \le (3\ 2\ 3)$ ✓ → $A = (4\ 3\ 3)$.
6. **P1.** $(0\ 1\ 0) \le (4\ 3\ 3)$ ✓ → $A = (5\ 3\ 4) = E$. Herkes işaretlendi: kilitlenme yoktur.
7. **Değişiklik.** P2'nin isteği (1 1 1) olursa $A = (1\ 0\ 1)$ iken P1 ve P2 ikinci türden ister (boşta 0), P3 birinci türden 2 ister (1 var), P4 üçüncü türden 2 ister (1 var). İlk turda hiçbir satır karşılanamaz; dört süreç de kilitlenmiştir.

7.5.9 Tespit ne zaman çalıştırılır?

Tespit algoritmasının ne sıklıkla çalıştırılacağı ayrı bir tasarım kararıdır. Üç seçenek vardır:

- **Her istekte.** Bekletilen her istekten sonra algoritma çalıştırılır. Kilitlenme oluştuğu anda, onu tetikleyen istekle birlikte bulunur. Ama işlemci maliyeti yüksektir.
- **Periyodik.** Algoritma her k dakikada bir çalışır. Ucuzdur; ama kilitlenme bir süre fark edilmeden kalır ve o sürede döngüye başka süreçler de takılabilir.
- **Belirtiye göre.** İşlemci kullanımı belirli bir eşiğin altına düştüğünde ya da bir kilit beklemesi uzadığında çalışır. Kilitlenen süreçler işlemci kullanmadığı için düşük kullanım iyi bir belirtidir.



Şekil 7.21: Periyodik tespit. Kilitlenme iki denetim arasında oluşursa bir sonraki denetime kadar fark edilmez. Periyot k küçülürse gecikme azalır ama yük artar.

Doğru sıklık, kilitlenmenin ne kadar olası olduğuna ve oluştuğunda kaç sürecin etkileneceğine bağlıdır. Tespitin yükü ihmal edilemez ve çalıştırma sıklığıyla doğru orantılıdır. Kurtarmanın maliyeti, yani kaybolan iş de hesaba katılır. PostgreSQL ara bir yol seçer: bir işlem bir kilidi `deadlock_timeout` kadar (varsayılan 1 saniye) beklemişse bekleme çizgesini tarar. Döngü bulursa işlemlerden birini iptal eder ve istemciye `ERROR: deadlock detected` iletisini döndürür. Böylece kısa sürede kendiliğinden biten beklemeler için tarama yapılmaz.

7.5.10 Kilitlenmeden kurtarma

Kilitlenme bulunduktan sonra döngü bir şekilde kırılmalıdır. Kırmanın üç yolu vardır ve hiçbiri bedelsiz değildir.

Tanım 7.5

Kilitlenmeden kurtarma (*deadlock recovery*), kaynak kesme, geri sarma ya da süreç sonlandırmayla döngüyü kırmaktır. Döngüyü kırmak için seçilen sürece **kurban** denir.

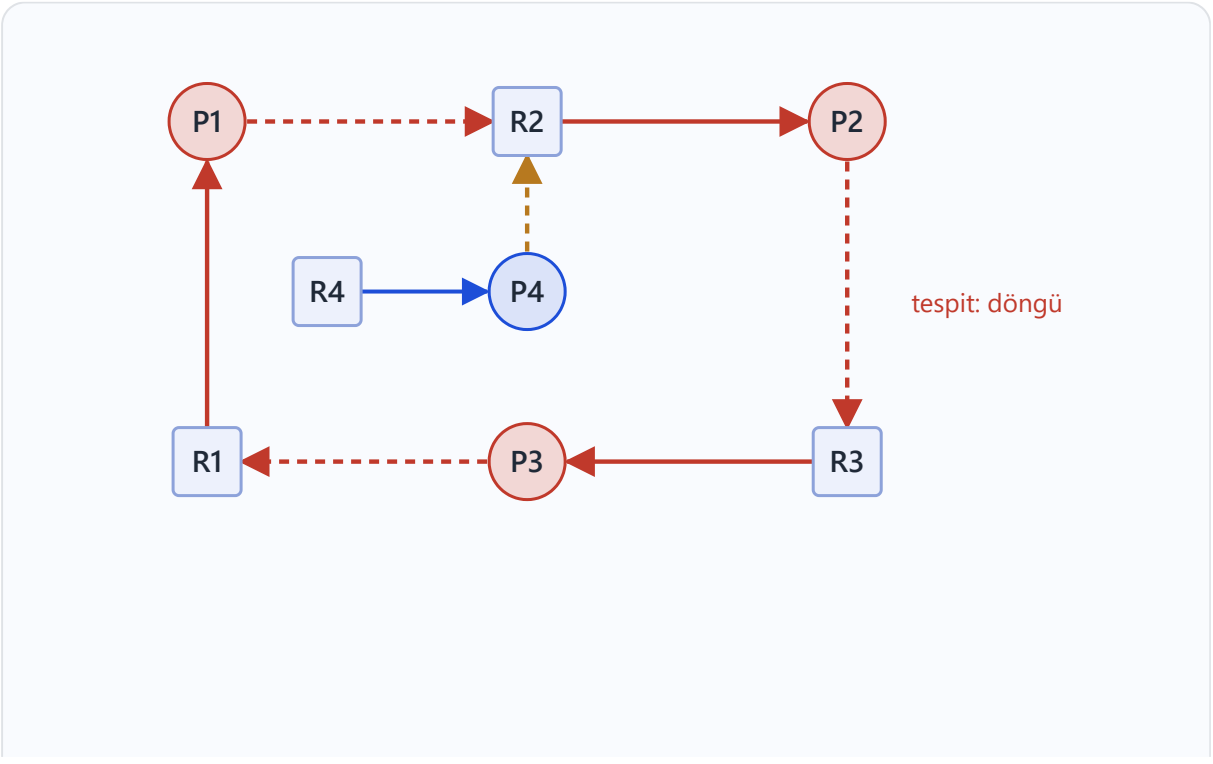
- **Kaynak kesme.** Bir kaynak sahibinden geçici olarak alınıp başka sürece verilir. Örneğin bir yazıcının basmakta olduğu iş kenara alınır, basılmış kâğıtlar ayrılır ve yazıcı başka işe verilir. Bu yöntem kaynağın doğasına bağlıdır; çoğu zaman elle yapılır ya da hiç mümkün değildir.
- **Geri sarma** (*rollback*). Süreçler belirli aralıklarla **denetim noktası** (*checkpoint*) kaydeder: bellek-lerini ve kaynak durumlarını bir dosyaya yazar. Kurban, çekişilen kaynağı almadan önceki son denetim noktasına döndürülür ve kaynak bırakılmış olur. O noktadan sonra yapılan iş kaybolur. Veritabanlarında işlemin iptal edilmesi (*abort*) bunun karşılığıdır.
- **Sonlandırma.** En kaba yöntemdir. Ya döngüdeki bütün süreçler öldürülür ya da döngü kırılana kadar teker teker öldürülür. İkinci seçenekte her öldürmeden sonra tespit algoritması yeniden çalışır. Döngünün dışından bir süreç de seçilebilir; yeter ki bıraktığı kaynaklar döngüyü kırsın.

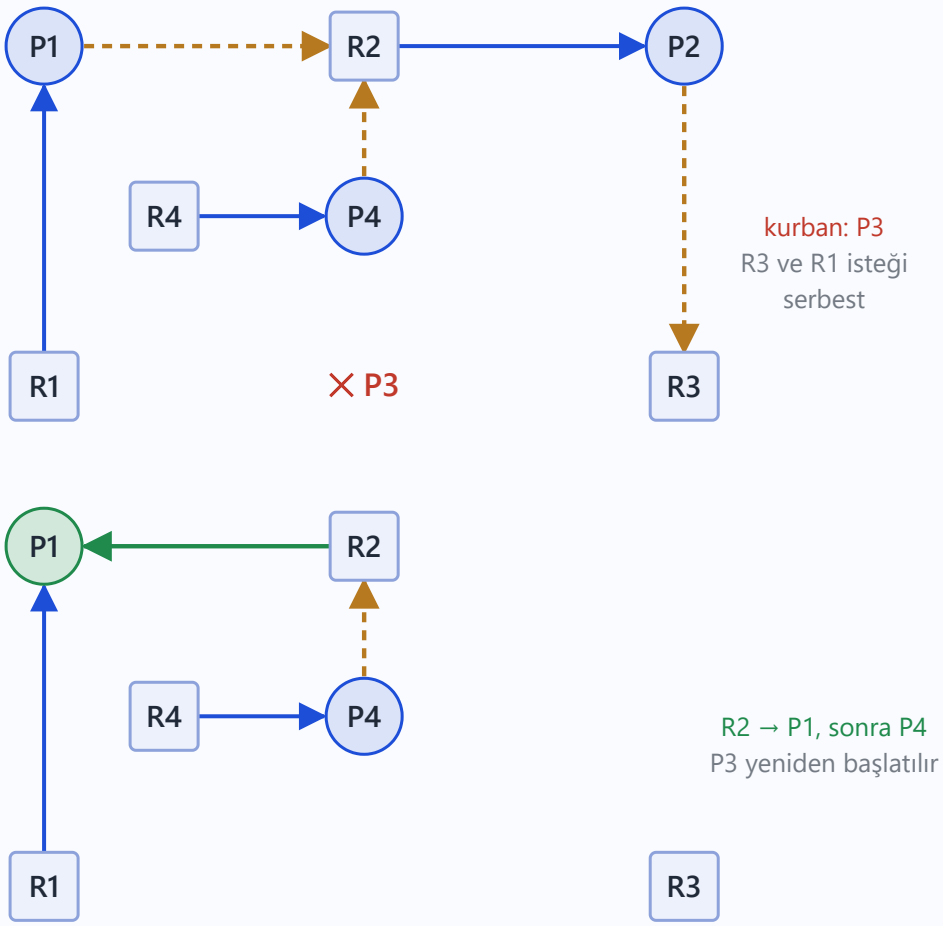
Kurban seçiminde amaç en düşük maliyettir. Maliyete katılan ilk etkenler sürecin önceliği, ne kadar süredir çalıştığı ve bitmesine ne kadar kaldığıdır. Tuttuğu kaynak sayısı ve bitmek için kaç kaynak daha istediği de hesaba girer. Sürecin etkileşimli mi yoksa toplu iş mi olduğuna ve güvenle yeniden başlatılıp başlatılamayacağına da bakılır. Bir derleme yeniden başlatılabilir; yarım kalmış bir veritabanı güncellemesi ise dikkat ister. Aynı süreç her seferinde kurban seçilirse hiç bitemez ve **açlık** oluşur. Bunu önlemek için sürecin kaç kez geri sarıldığı maliyete eklenir.

Önleme ve tespit, birer kurtarma yöntemi değildir. Hiçbir kurtarma kesintisiz de değildir: her kurtarma yöntemi kaybedilen bir iş ya da yeniden çalıştırma maliyeti getirir.

7.5.11 Kurtarma adım adım

Dört süreçli bir örnekte tespit algoritması P1 → R2 → P2 → R3 → P3 → R1 → P1 döngüsünü bulmuş olsun. P4 de R2'yi beklediği için takılıdır. Şekil 7.22 teker teker sonlandırmayla kurtarmayı gösterir.





Şekil 7.22: Kurtarma. (a) Tespit algoritması döngüyü bulur. (b) En az iş kaybettirecek süreç olan P3 kurban seçilir; tuttuğu R3 serbest kalır. (c) R3, P2'ye verilir; P2 bitince R2 önce P1'e, P1 bitince P4'e verilir. P3 baştan ya da denetim noktasından yeniden çalıştırılır.

1. **Kurban seçimi.** En az iş kaybettirecek süreç P3'tür. P3 sonlandırılır ya da denetim noktasına geri sarılır; tuttuğu R3 serbest kalır.
2. **Tespit yeniden.** P3'ün kenarları silindikten sonra çizgede döngü kalmamıştır.
3. **Zincirleme ilerleme.** R3, onu bekleyen P2'ye verilir. P2 işini bitirip R2 ile R3'ü bırakır. R2 sıradaki bekleyene, P1'e verilir; P1 bitince P4 de R2'yi alır.
4. **Kurbanın yeniden çalışması.** P3 baştan ya da son denetim noktasından yeniden başlatılır. Kaybedilen iş, P3'ün o noktadan sonra yaptığı iştir.

Teker teker sonlandırmada her kurbandan sonra tespit algoritması yeniden çalışır ve döngü kırılınca durulur. Bu örnekte tek kurban yetmiştir.

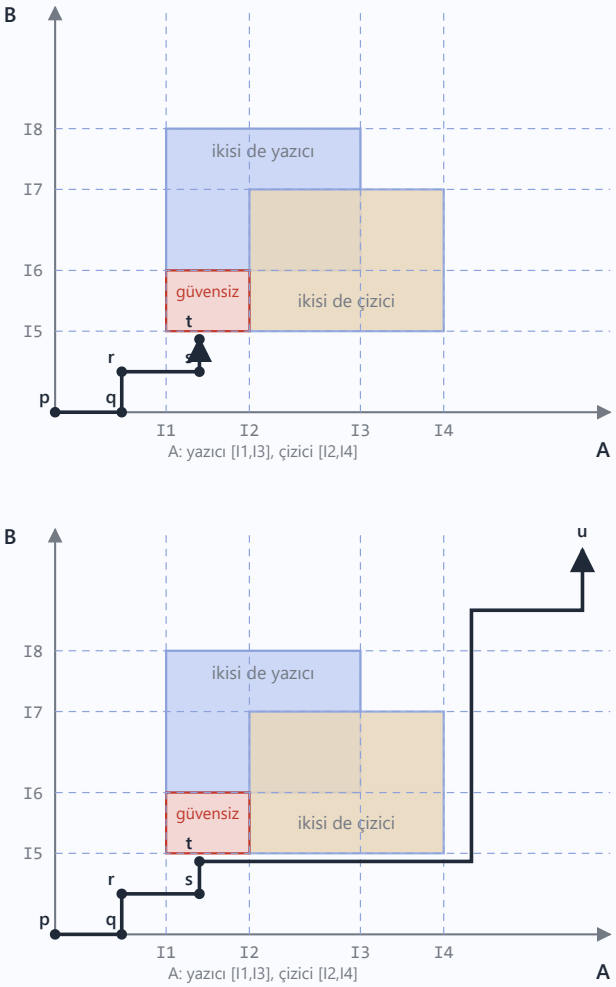
7.6 Kilitlenmeden kaçınma

Tespit stratejisi kilitlenmeye izin verir ve sonra bedelini öder. Kaçınma stratejisi ise kilitlenmeye hiç izin vermez. Sistem her kaynak isteğinde “bu isteği verirsem ileride kilitlenme kaçınılmaz olur mu?” diye sorar ve cevap evetse isteği erteler. Bu soruyu cevaplamak için her sürecin en fazla ne kadar kaynak isteyebileceği önceden bilinmelidir.

7.6.1 Kaynak yörüngeleri

Kaçınmanın fikri en açık biçimde iki süreçli bir çizimde görülür. Bu gösterim Dijkstra'ya dayanır. A ve B süreçleri bir yazıcı ve bir çizici kullanıyor olsun. A, I1 ile I3 komutları arasında yazıcıyı, I2 ile I4 arasında çiziciyi kullanır. B ise I5 ile I7 arasında çiziciyi, I6 ile I8 arasında yazıcıyı kullanır.

Yatay eksen A'nın, dikey eksen B'nin yürüttüğü komutları gösterir. Sistemin her anı düzlemde bir noktadır. Tek işlemci olduğu için yörünge yalnızca sağa (A çalışıyor) ya da yukarı (B çalışıyor) ilerler; çapraz gidemez. İki taralı bölge, iki sürecin aynı kaynağı aynı anda kullanacağı noktalardır. Karşılıklı dışlama nedeniyle bu bölgelere girilemez.



Şekil 7.23: İki sürecin kaynak yörüngeleri (Tanenbaum Şekil 6-8). (a) p → q → r → s → t yörüngesi; kırmızı kutu güvensiz bölgedir. (b) *t* noktasında B'nin çizici isteği ertelenir; A I4'e kadar ilerleyip iki kaynağı da bırakır, sonra B devam eder.

Yörüngeyi izleyelim. *p* → *q* arasında A çalışır; *q* → *r* arasında B çalışır. *r* → *s* arasında A yine çalışır ve I1'de yazıcıyı alır. *s* → *t* arasında B ilerler ve I5'te çiziciyi isteyecektir. Kırmızı kutu, A'nın yazıcıyı, B'nin de çiziciyi tuttuğu bölgedir. Yörünge bu kutuya girerse iki süreç de sağa ya da yukarı ilerlemek için taralı bölgelerden birine girmek zorunda kalır. Bu imkânsız olduğu için kilitlenme kaçınılmazdır. Kutunun içinde henüz kimse bloke değildir, ama sonuç kesindir. Bu yüzden bu bölgeye **güvensiz** denir.

Kaçınma, yörüngeyi güvensiz bölgeye sokacak isteği ertelemektir. *t* noktasında işletim sistemi B'nin çizici isteğini bekletir. A, I4'e kadar çalışıp iki kaynağı da bırakır; ardından B devam eder ve kilitlenme olmaz. Burada kritik olan, işletim sisteminin A'nın ileride çiziciyi isteyeceğini bilmesidir. Bu bilgi olmadan *t*'deki istek zararsız görünürdü.

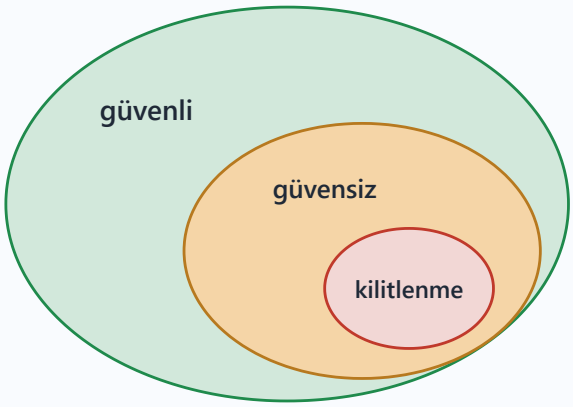
7.6.2 Güvenli ve güvensiz durumlar

Yörünge çizimi iki süreç ve iki kaynakla sınırlıdır. Genel durumda kaçınma, sistemin durumuna bakar. Durum, var olan kaynaklar E, boştakiler A, tahsis matrisi C ve her sürecin azami ihtiyacı (Max) ile belirlenir.

Tanım 7.6

Her süreç azami isteğini hemen yapsa bile bütün süreçleri bir sırayla sonuna kadar çalıştırabilecek bir **güvenli dizi** $\langle P_1, \ldots, P_n \rangle$ varsa durum **güvenlidir** (*safe state*). Dizide her P_i için kalan ihtiyaç, boştaki kaynaklarla dizide ondan önce gelen P_j'lerin ($j < i$) tuttuğu kaynakların toplamını aşmaz. Böyle bir dizi yoksa durum **güvensizdir** (*unsafe state*).

Tanımın mantığı tespit algoritmasınınkiyle aynıdır, tek fark bakılan satırdır. Tespitte süreçlerin *şu anki istekleri* karşılanabiliyor mu diye bakılır. Kaçınmada ise en kötü durum varsayılır: her süreç azamisine kadar isteyecek sayılır ve *kalan ihtiyacı* karşılanabiliyor mu diye bakılır. Dizideki ilk süreç kalan ihtiyacını boştaki kaynaklarla karşılar, biter ve her şeyi bırakır. İkinci süreç bu büyümüş havuzla karşılanır ve zincir böyle sürer.



Şekil 7.24: Durum uzayı. Kilitlenmiş durumlar güvensiz durumların bir alt kümesidir; güvenli durumlar ise kilitlenmeden tümüyle ayrıdır.

Güvensiz durum kilitlenmiş durum değildir. Güvensiz durumda henüz kimse kalıcı olarak bloke değildir. Süreçler azami isteklerini yapmazsa, örneğin biri ek isteğini yapmadan bir kaynağı bırakırsa, her şey yolunda gidebilir. Fark şudur: güvenli durumda sistem herkesin bitmesini garanti edebilir; güvensiz durumda bu garanti yoktur. Kaçınma algoritması sistemi her zaman güvenli durumda tutar ve güvensiz duruma götürecek her isteği erteler.

Sık yapılan hata

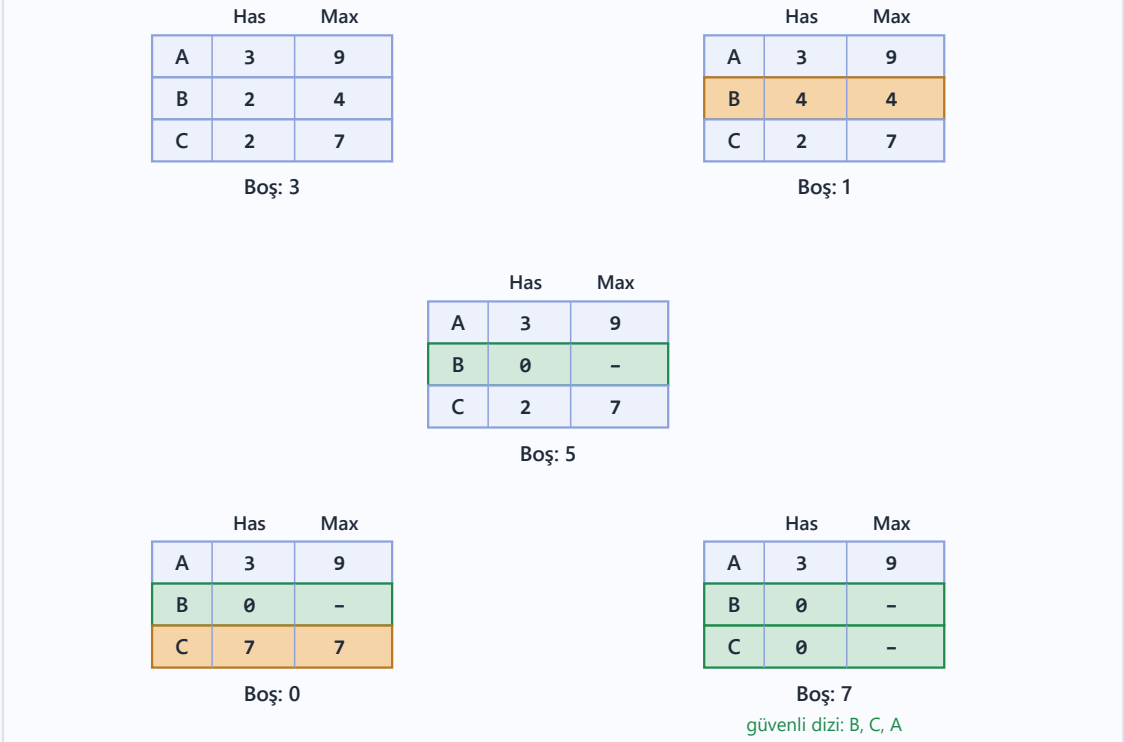
"Güvensiz" ile "kilitlenmiş" sözcüklerini eş anlamlı kullanmak. Güvensiz durum yalnızca bitiş garantisinin kaybolduğunu söyler. Bankacı algoritması güvensiz durumu kilitlenme olduğu için değil, kilitlenme *olabileceği* için reddeder. Sistem güvensiz durumda kalsa da süreçler azami isteklerini yapmadan bitebilir.

7.6.3 Güvenli durum örneği

Tek kaynak türünden 10 örnek olsun. Tablolarda "Has" sürecin şu an tuttuğu, "Max" bitene kadar gerekecek en çok örnek sayısıdır. Kalan ihtiyaç Max − Has'tir. Kaçınmada işletim sistemi isteklerin sırasını kendisi seçebilecek konumdadır: güvenli bir dizi varsa süreçleri o diziye göre bitirebilir.

Örnek 7.4

A 3 örnek tutuyor (Max 9), B 2 (Max 4), C 2 (Max 7). Toplam 10 örnekten 7'si dağıtılmış, 3'ü boştur. Durum güvenli mi? (Tanenbaum Şekil 6-9)



Şekil 7.25: Güvenli durum. (a) Başlangıç, boş 3. (b) B'ye 2 verilir, boş 1. (c) B biter, boş 5. (d) C'ye 5 verilir, boş 0. (e) C biter, boş 7; A'nın kalan 6'sı karşılanır.

1. **Kalan ihtiyaçlar.** A: 9 − 3 = 6; B: 4 − 2 = 2; C: 7 − 2 = 5.

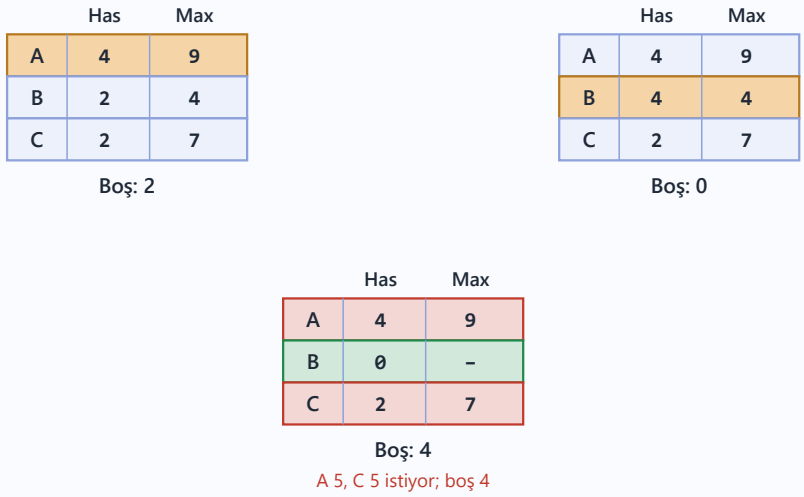
2. **B.** Kalan ihtiyacı 2 ≤ 3'tür. B'ye 2 örnek verilir; boş 1 kalır. B azamisine ulaşır, biter ve 4 örneği bırakır: boş 5.
3. **C.** Kalan ihtiyacı 5 ≤ 5'tir. C'ye 5 verilir; boş 0 kalır. C biter ve 7 örneği bırakır: boş 7.
4. **A.** Kalan ihtiyacı 6 ≤ 7'dir. A da biter.
5. **Sonuç.** ⟨B, C, A⟩ güvenli bir dizidir; durum güvenlidir. Bütün sıralar denendiğinde başka güvenli dizi olmadığı da görülür: ilk adımda yalnız B karşılanabilir, B'den sonra yalnız C.

7.6.4 Güvensiz durum örneği

Aynı başlangıçtan çıkalım; fark tek bir karardadır. A bir örnek daha ister ve işletim sistemi bunu verir.

Örnek 7.5

A 4 örnek tutuyor (Max 9), B 2 (Max 4), C 2 (Max 7); boş 2. Durum güvenli mi? (Tanenbaum Şekil 6-10)



Şekil 7.26: Güvensiz durum. (a) A 4/9, boş 2. (b) Yalnız B karşılanabilir; B'ye 2 verilir. (c) B biter, boş 4; A'nın 5, C'nin 5 ihtiyacı vardır ve hiçbiri garanti edilemez.

1. **Kalan ihtiyaçlar.** A: 5; B: 2; C: 5.
2. **B.** 2 ≤ 2; B'ye 2 verilir, B biter ve 4 örneği bırakır: boş 4.
3. **A ve C.** İkisinin de kalan ihtiyacı 5'tir ve 5 > 4. Hiçbiri garanti edilemez.
4. **Sonuç.** Güvenli dizi yoktur; A'ya fazladan örnek verildiği an durum güvensiz olmuştur. A'nın isteği ertelenmeliydi.

Burada henüz kilitlenme yoktur. A ek isteğini yapmadan bir örneği bırakırsa sorun çıkmayabilir. Ama işletim sistemi en kötü durumu varsayar: herkes azami isteğini yaparsa bitiş garanti değildir. Kaçınma, bu garantiyi kaybettiren isteği vermez.

7.6.5 Bankacı algoritması: tek kaynak türü

Dijkstra 1965'te kaçınma algoritmasını bir banka benzetmesiyle anlattı. Küçük bir kasaba bankacısı müşterilerine kredi limitleri açar. Limitlerin toplamı kasadaki paradan fazladır, çünkü bütün müşteriler aynı anda limitlerinin tamamını istemez. Bankacı her istekte “bu krediyi verirsem bütün müşterilerimin limitlerini sonunda karşılayabilir miyim?” diye sorar.

Tanım 7.7

Bankacı algoritması (*banker's algorithm*), her kaynak isteğinde isteği verilmiş gibi yeni durumu hesaplayan ve bu durum güvenliyse isteği veren, güvensizse süreci bekleten kaçınma algoritmasıdır. İstek sonra, kaynaklar bırakıldığında yeniden değerlendirilir.

Güvenlik sınaması şöyle yapılır: kalan ihtiyacı kasadaki parayla karşılanabilen bir müşteri bulunur; bu müşteri bitmiş sayılır ve kredisi kasaya eklenir. Bu işlem bütün müşteriler bitene kadar ya da kimse karşılanamayana kadar tekrarlanır.

Örnek 7.6

Kasada 10 birim var. Dört müşterinin kredi limitleri A 6, B 5, C 4, D 7'dir; toplam 22. Bir süre sonra A 1, B 1, C 2, D 4 birim kullanıyor. Durum güvenli mi? Bu durumda B bir birim daha isterse istek verilmeli mi? (Tanenbaum Şekil 6-11)



Şekil 7.27: Bankacı algoritması, tek kaynak türü. (a) Başlangıç: kimse kredi kullanmıyor, kasada 10 birim. (b) Kullanılan 8, kasada 2: güvenli. (c) B bir birim daha alırsa kasada 1 kalır: güvensiz.

1. **Başlangıç.** Limit toplamı 22 > 10 olsa da kimse kredi kullanmadığı sürece durum güvenlidir; müşteriler sırayla bitirilebilir.
2. **(b) durumu.** Kullanılan 1 + 1 + 2 + 4 = 8, kasada 2 birim vardır. Kalan ihtiyaçlar A 5, B 4, C 2, D 3'tür.
3. **Güvenli dizi.** C'nin ihtiyacı 2 ≤ 2: C biter ve 4 birim döner, kasada 4. D'nin ihtiyacı 3 ≤ 4: D biter, kasada 8. A (5 ≤ 8) bitince 9, B (4 ≤ 9) bitince 10. ⟨C, D, A, B⟩ güvenli bir dizidir.
4. **B'nin isteği.** B bir birim daha alırsa kasada 1 kalır. Kalan ihtiyaçlar A 5, B 3, C 2, D 3'tür; en küçüğü bile 1'den büyüktür. Durum güvensizdir; istek reddedilir ve B bekletilir.

Durum, yalnız bir sürecin değil bütün süreçlerin bir sırayla bitebildiği zaman güvenlidir. Bankacı algoritması kilitlenmeyi sonradan çözmez; sistemi hiç kilitlenmeye sokmaz. Bedeli de bellidir: (c) durumunda kasada fiilen para varken B bekletilir.

Örnek 7.7

Süreçler aynı türden özdeş kaynakları (teyp sürücüsü) teker teker istiyor ve hepsini işi bitince bırakıyor. (a) 4 süreç var ve her biri aynı anda en çok 3 sürücü kullanıyor. Kilitlenmenin hiç oluşmaması için en az kaç sürücü gerekir? (b) 6 süreç, süreç başına en çok 4 sürücü ve sistemde 18 sürücü varsa kilitlenme olabilir mi? (c) 18 sürücü ve süreç başına en çok 4 sürücüyle en çok kaç süreç güvenle çalışır?

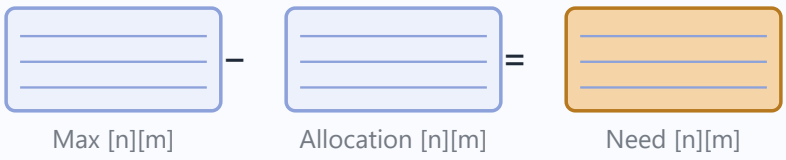
1. **En kötü durum.** n süreç, her biri en çok k sürücü istiyor olsun. En kötü durumda her süreç $k-1$ sürücü tutar ve bir tane daha bekler. Bir sürücü daha varsa süreçlerden biri ihtiyacını tamamlar, biter ve sürücülerini bırakır. Gereken en az sayı $n \cdot (k-1) + 1$'dir.
2. **(a)** $4 \cdot 2 + 1 = 9$ sürücü. 8 sürücüde dört sürecin her biri ikişer sürücü tutup üçüncüyü bekleyebilir.
3. **(b)** $6 \cdot 3 + 1 = 19 > 18$: kilitlenme olabilir. 18 sürücünün hepsi altı sürece üçer dağılırsa herkes dördüncüyü bekler.
4. **(c)** $3n + 1 \le 18$ olmalıdır; $n \le 5{,}67$, yani en çok 5 süreç.

Bu sınır, tut ve beklemenin döngüsel beklemeye dönüşemeyeceği kaynak sayısıdır ve bankacı algoritmasının tek kaynaklı en kötü durumuna karşılık gelir. Değerler bütün dağılımlar Python'da kaba kuvvetle denenerek doğrulanmıştır.

7.6.6 Bankacı algoritması: çok kaynak türü

Algoritma çok kaynak türüne doğrudan genellenir. n süreç ve m kaynak türü için Silberschatz'ın adlandırmasıyla dört veri yapısı tutulur:

- **Available[*m*]**: her türden boştaki örnek sayısı.
- **Max[*n*][*m*]**: her sürecin önceden bildirdiği azami istek.
- **Allocation[*n*][*m*]**: her sürecin şu an tuttuğu örnekler.
- **Need[*n*][*m*]** = Max − Allocation: kalan ihtiyaç.



Şekil 7.28: Kalan ihtiyaç matrisi, azami istek matrisinden tahsis matrisinin çıkarılmasıyla bulunur: Need = Max − Allocation.

Algoritmanın iki parçası vardır. **Güvenlik algoritması** bir durumun güvenli olup olmadığını söyler:

```
/* güvenlik algoritması */
Work ← Available
Finish[i] ← false  (tüm i)
while Finish[i] = false ve
      Need[i] ≤ Work olan i var:
  Work ← Work + Allocation[i]
  Finish[i] ← true
tüm i için Finish[i] ise GÜVENLİ
```

`Work`, sınama sırasında boş havuzu temsil eden geçici bir vektördür; `Available`'ın kendisi değiştirilmez. `Finish[i]`, *i*. sürecin bitmiş sayılıp sayılmadığını tutar. Kalan ihtiyacı `Work` ile karşılanabilen bitmemiş bir süreç bulundukça o süreç bitmiş sayılır ve tuttukları `Work`'e eklenir. Sonunda bütün `Finish` değerleri doğruysa durum güvenlidir.

İstek algoritması ise bir süreç `Request[i]` istediğinde çalışır:

```
/* Ti, Request[i] isterse */
if Request[i] > Need[i]:
  hata  /* Max aşıldı */
if Request[i] > Available:
  Ti bekler  /* kaynak yok */
/* varmış gibi yap: */
Available  -= Request[i]
Allocation[i] += Request[i]
Need[i]    -= Request[i]
if güvenli: isteği ver
else: geri al, Ti bekler
```

İlk sınama bir hata denetimidir: süreç bildirdiği azamiden fazlasını isteyemez. İkinci sınama kaynakların şu an fiilen boşta olup olmadığına bakar; değilse süreç zaten beklemek zorundadır. İki sınama da geçerse algoritma isteği *verilmiş gibi* yapar: üç yapıyı günceller ve yeni durumda güvenlik algoritmasını çalıştırır. Durum güvenliyse değişiklikler kalıcı olur. Güvensizse üç değişiklik geri alınır ve süreç bekletilir.

Güvenlik algoritmasının maliyeti tespit algoritmasınınkiyle aynıdır: $O(m \cdot n^2)$. Fark, bu sınamanın her istekte çalışmasıdır.

> **Sık yapılan hata**
>
> Güvenlik algoritmasında biten sürecin *Need* satırını `Work`'e eklemek. Biten süreç tuttuğu kaynakları, yani *Allocation* satırını bırakır. *Need* yalnızca sürecin bitebilmesi için gereken miktardır ve karşılaştırmada kullanılır; havuza eklenen ise *Allocation*'dır.

7.6.7 Güvenlik algoritması: örnek

Silberschatz'ın klasik örneğinde beş iş parçacığı (T0–T4) ve üç kaynak türü vardır: A türünden 10, B türünden 5, C türünden 7 örnek. Tahsis ve azami istekler şöyledir:

İş parçacığı	Allocation (A B C)	Max (A B C)	Need (A B C)
T0	0 1 0	7 5 3	7 4 3
T1	2 0 0	3 2 2	1 2 2
T2	3 0 2	9 0 2	6 0 0

İş parçacığı	Allocation (A B C)	Max (A B C)	Need (A B C)
T3	2 1 1	2 2 2	0 1 1
T4	0 0 2	4 3 3	4 3 1

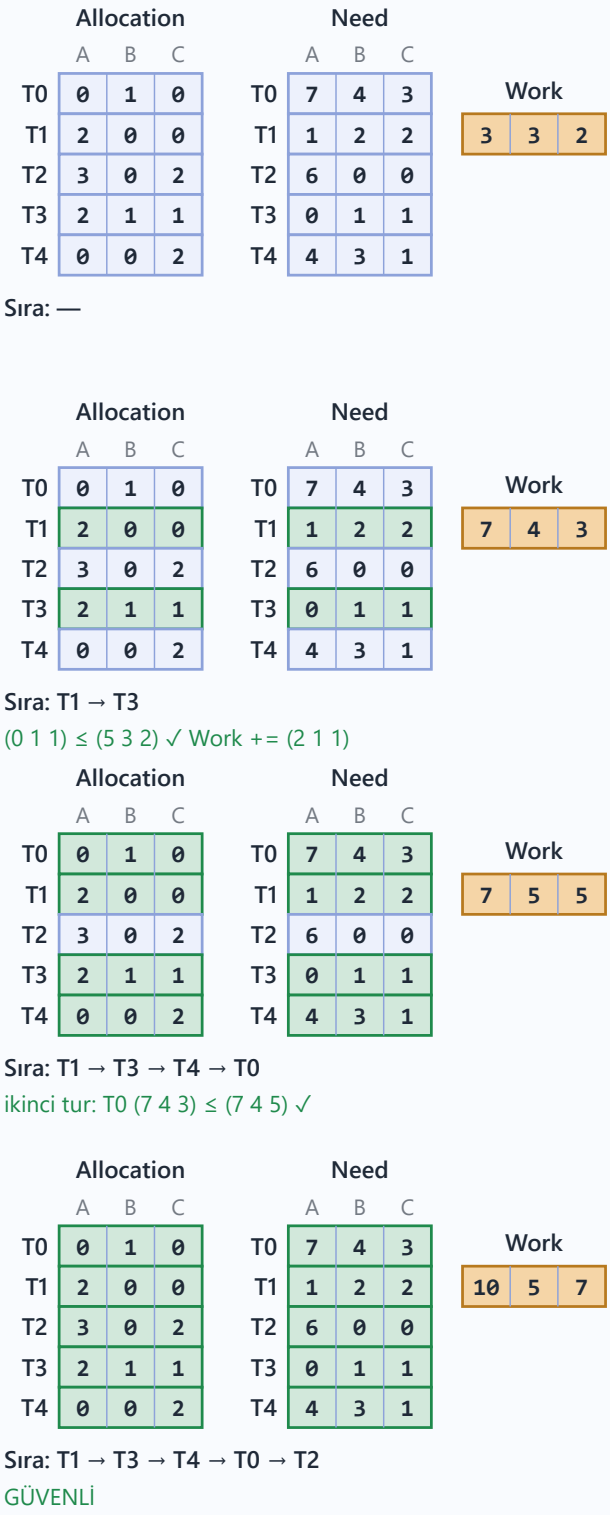
Tablo 7.8: Silberschatz örneğinin başlangıç durumu.

Tahsislerin sütun toplamı (7 2 5)'tir; Available = (10 5 7) − (7 2 5) = (3 3 2). Need sütunu Max − Allocation ile hesaplanmıştır. Güvenlik algoritması `Work` = (3 3 2) ile başlar ve satırları T0'dan başlayarak kaldığı yerden döngüsel olarak tarar.

```
Work ← Available
Finish[i] ← false
for i = T0 … T4 (tekrar tekrar):
  if Finish[i] = false and
     Need[i] ≤ Work:
    Work ← Work + Allocation[i]
    Finish[i] ← true
if tüm Finish[i]: GÜVENLİ
```

Adım	Satır	Need ≤ Work?	Sonuç	Work sonrası
1	T0	(7 4 3) ≤ (3 3 2)	✗ geç	(3 3 2)
2	T1	(1 2 2) ≤ (3 3 2)	✓ + (2 0 0)	(5 3 2)
3	T2	(6 0 0) ≤ (5 3 2)	✗ A yetmez	(5 3 2)
4	T3	(0 1 1) ≤ (5 3 2)	✓ + (2 1 1)	(7 4 3)
5	T4	(4 3 1) ≤ (7 4 3)	✓ + (0 0 2)	(7 4 5)
6	T0 (ikinci tur)	(7 4 3) ≤ (7 4 5)	✓ + (0 1 0)	(7 5 5)
7	T2	(6 0 0) ≤ (7 5 5)	✓ + (3 0 2)	(10 5 7)

Tablo 7.9: Güvenlik algoritmasının izlemesi.



Şekil 7.29: Güvenlik algoritması. (a) Başlangıç: Work = (3 3 2). (b) T1 ve T3 bittikten sonra Work = (7 4 3). (c) İkinci turda T0 biter, Work = (7 5 5). (d) Bütün Finish değerleri doğru: durum güvenlidir.

Bütün `Finish` değerleri doğrudur ve sonunda `Work` = (10 5 7), yani bütün kaynaklardır. Durum güvenlidir; bulunan güvenli dizi ⟨T1, T3, T4, T0, T2⟩'dir. Güvenli dizi tek değildir. Bütün sıralar denendiğinde bu durum için 16 güvenli dizi bulunur; ⟨T1, T3, T4, T2, T0⟩ de bunlardan biridir. Hepsinin ortak

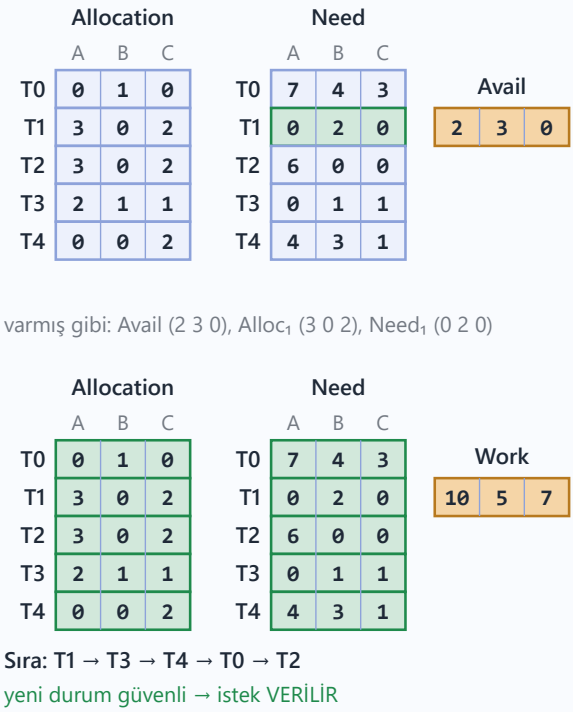
noktası ilk sürecin T1 ya da T3 olmasıdır, çünkü başlangıçtaki (3 3 2) ile yalnız bu ikisi karşılanabilir.

7.6.8 İstek kabulü

Aynı durumda T1, Request_1 = (1 0 2) istesin. İstek algoritması üç adımda karar verir.

Örnek 7.8

T1, (1 0 2) istiyor. İstek verilmeli mi?



Şekil 7.30: T1'in isteği. (a) Varmış gibi durum: Available (2 3 0), Allocation_1 (3 0 2), Need_1 (0 2 0). (b) Güvenlik algoritması ⟨T1, T3, T4, T0, T2⟩ dizisini bulur: istek verilir.

1. **Need denetimi.** (1 0 2) ≤ Need_1 = (1 2 2) ✓. T1 bildirdiği azamiyi aşmıyor.
2. **Available denetimi.** (1 0 2) ≤ Available = (3 3 2) ✓. Kaynaklar şu an boştadır. Ama bu tek başına yetmez.
3. **Varmış gibi durum.** Available = (3 3 2) − (1 0 2) = (2 3 0); Allocation_1 = (2 0 0) + (1 0 2) = (3 0 2); Need_1 = (1 2 2) − (1 0 2) = (0 2 0).
4. **Güvenlik testi.** Work = (2 3 0). T0 (7 4 3) ✗. T1 (0 2 0) ✓ → Work = (5 3 2). T2 (6 0 0) ✗. T3 (0 1 1) ✓ → (7 4 3). T4 (4 3 1) ✓ → (7 4 5). T0 (7 4 3) ✓ → (7 5 5). T2 (6 0 0) ✓ → (10 5 7).
5. **Karar.** Güvenli dizi ⟨T1, T3, T4, T0, T2⟩ vardır; istek verilir ve yeni durum kalıcı olur.

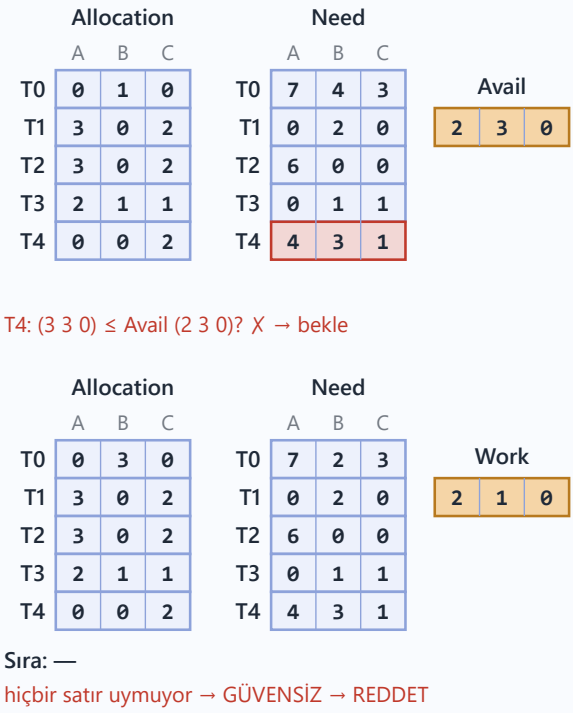
Güvenlik testi "varmış gibi" durumda yapılır. Durum güvensiz çıksaydı üç değişiklik de geri alınırdı ve T1 bekletilirdi.

7.6.9 İstek reddi

T1'in isteği verildikten sonra iki yeni istek gelsin. Bu iki istek, bir sürecin bekletilmesinin iki farklı nedenini gösterir.

Örnek 7.9

Durum T1'in isteği verildikten sonraki durumdur: Available = (2 3 0). (a) T4, (3 3 0) isterse ne olur? (b) T0, (0 2 0) isterse ne olur?



Şekil 7.31: İki ret. (a) T4'ün isteği için A türünden yalnız 2 örnek boştadır: T4 bekler. (b) T0'ın isteği verilmiş gibi yapılınca Available (2 1 0) olur ve hiçbir satır karşılanamaz: durum güvensizdir, istek reddedilir.

1. **(a) T4.** (3 3 0) ≤ Need_4 = (4 3 1) ✓. Ama (3 3 0) ≤ (2 3 0) değildir: A türünden yalnız 2 örnek boştadır. Kaynak yoktur, T4 bekler. Güvenlik testine hiç gelinmez.
2. **(b) T0, iki denetim.** (0 2 0) ≤ Need_0 = (7 4 3) ✓ ve (0 2 0) ≤ (2 3 0) ✓. Kaynak fiilen boştadır.
3. **Varmış gibi durum.** Available = (2 1 0); Allocation_0 = (0 3 0); Need_0 = (7 2 3).
4. **Güvenlik testi.** Work = (2 1 0). T0 (7 2 3): A yetmez. T1 (0 2 0): B yetmez. T2 (6 0 0): A yetmez. T3 (0 1 1): C yetmez. T4 (4 3 1): A yetmez. Hiçbir satır karşılanamaz.

5. **Karar.** Durum güvensizdir. Değişiklikler geri alınır ve T0 bekletilir; kaynak boşta olduğu hâlde.

Birinci nedende kaynak yoktur; ikincide kaynak vardır, ama vermek durumu güvensiz yapar. Kaçınmanın bedeli ikinci nedendir: kaynaklar bazen boşta beklerken istekler ertelenir ve kaynak kullanım oranı düşer.

7.6.10 Kod izleme: bankacı istek algoritması

İstek ve güvenlik algoritmalarını birlikte izlemek için başlangıç durumuna dönelim: Available = (3 3 2). Bu kez T3, Request = (0 1 1) istiyor. Kod iki algoritmayı bir arada verir:

```
/* Ti, Request isterse */
if Request > Need[i]: hata /* Max aşıldı */
if Request > Available: Ti bekler
Available -= Request  /* varmış gibi */
Allocation[i] += Request
Need[i] -= Request
if guvenli(): isteği VER
else: geri al, Ti bekler
guvenli():
  Work ← Available; Finish ← false
  for i = T0 … T4 (tekrar tekrar):
    if !Finish[i] and Need[i] ≤ Work:
      Work ← Work + Allocation[i]
      Finish[i] ← true
  return tüm Finish[i]
```

Adım	Satır	Değerlendirme	Değişen değerler
1	`Request > Need[3]` ?	(0 1 1) ≤ (0 1 1): azami aşılmıyor	—
2	`Request > Available` ?	(0 1 1) ≤ (3 3 2): kaynak var	—
3	varmış gibi	üç yapı güncellenir	Available (3 2 1), Allocation[3] (2 2 2), Need[3] (0 0 0)
4	`guvenli()`	Work ← (3 2 1); T0 (7 4 3), T1 (1 2 2), T2 (6 0 0) bitemez	Work (3 2 1)
5	T3	(0 0 0) ≤ (3 2 1) ✓	Work (5 4 3), Finish[3]
6	T4	(4 3 1) ≤ (5 4 3) ✓	Work (5 4 5), Finish[4]
7	T0, T1 (ikinci tur)	T0'da A yetmez; T1: (1 2 2) ≤ (5 4 5) ✓	Work (7 4 5), Finish[1]
8	T2	(6 0 0) ≤ (7 4 5) ✓	Work (10 4 7), Finish[2]
9	T0 (üçüncü tur)	(7 4 3) ≤ (10 4 7) ✓	Work (10 5 7), Finish[0]

Adım	Satır	Değerlendirme	Değişen değerler
10	sonuç	bütün Finish doğru: güvenli	dizi ⟨T3, T4, T1, T2, T0⟩; istek verilir

Tablo 7.10: T3'ün (0 1 1) isteğinin izlemesi.

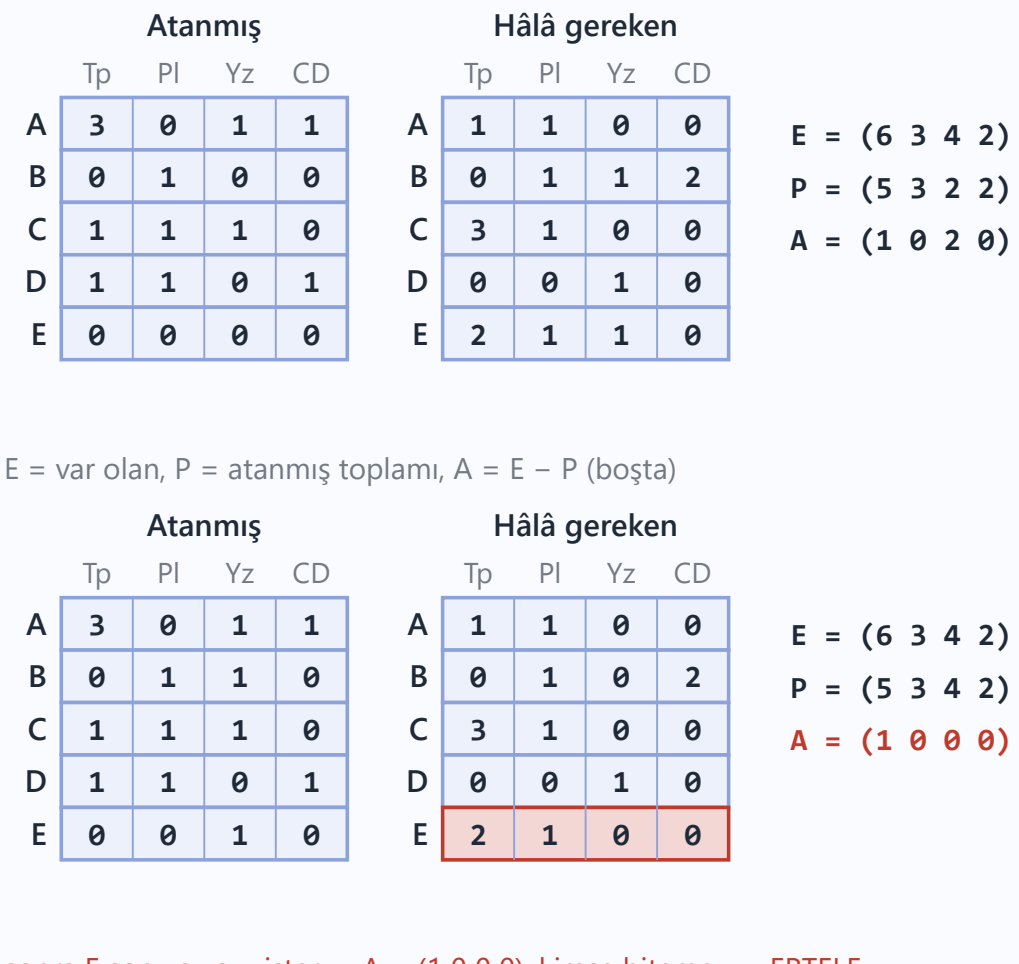


Şekil 7.32: T3'ün isteğinin izlemesi. (a) Varmış gibi durum: Allocation[3] = (2 2 2), Need[3] = (0 0 0). (b) T3 ve T4 bitti, Work = (5 4 5). (c) T1 ve T2 de bitti, Work = (10 4 7). (d) T0 da biter: bütün Finish değerleri doğru.

İzlemede dikkat çeken nokta, T1'in ilk turda bitememesidir. Başlangıç durumunda T1 hemen bitebiliyordu. Ama T3'e bir B ve bir C verildiği için `Work` = (3 2 1) olmuştur; T1'in C ihtiyacı ise 2 > 1'dir. T3'ün kalan ihtiyacı (0 0 0)'a indiği için T3 hemen biter ve tuttuğu (2 2 2)'yi geri verir; bundan sonra zincir kurulur. Bütün değerler Python'da istek ve güvenlik algoritmalarının kendisiyle yeniden hesaplanmıştır.

7.6.11 İkinci örnek

Tanenbaum'un çok kaynaklı örneğinde beş süreç (A–E) ve dört kaynak türü vardır: teyp (Tp), çizici (Pl), yazıcı (Yz) ve CD. Var olan kaynaklar E = (6 3 4 2), atanmış kaynakların toplamı P = (5 3 2 2)'dir. Boştakiler $A = E - P$ = (1 0 2 0)'dır.



Şekil 7.33: Çok kaynaklı bankacı algoritması (Tanenbaum Şekil 6-12). Üstte başlangıç: atanmış kaynaklar, hâlâ gereken kaynaklar ve E, P, A vektörleri. Altta B ve E'ye birer yazıcı verildikten sonraki durum: A = (1 0 0 0) ile hiçbir satır karşılanamaz.

Örnek 7.10

Hâlâ gereken kaynaklar A (1 1 0 0), B (0 1 1 2), C (3 1 0 0), D (0 0 1 0), E (2 1 1 0)'dır. Durum güvenli mi? B bir yazıcı isterse verilmeli mi? Ardından E son yazıcıyı isterse ne olur?

1. **Güvenlik.** D'nin ihtiyacı (0 0 1 0) ≤ (1 0 2 0)'dır. D biter ve (1 1 0 1) döner: A = (2 1 2 1). Sonra E (2 1 1 0) ≤ (2 1 2 1) biter; E hiçbir şey tutmadığı için A değişmez. A biter, A = (5 1 3 2); B biter, (5 2 3 2); C biter, (6 3 4 2). ⟨D, E, A, B, C⟩ güvenli bir dizidir.

2. **B'nin yazıcı isteği.** Verilmiş gibi yapılınca A = (1 0 1 0), P = (5 3 3 2) olur; B'nin ihtiyacı (0 1 0 2)'ye iner. D yine bitebilir ((0 0 1 0) ≤ (1 0 1 0)) ve aynı dizi korunur. İstek verilir.
3. **E'nin yazıcı isteği.** Verilmiş gibi yapılınca A = (1 0 0 0), P = (5 3 4 2) olur. A ve B'nin çizici ihtiyacı, C'nin teyp ve çizici ihtiyacı, D'nin yazıcı ihtiyacı, E'nin de çizici ihtiyacı karşılanamaz. Hiçbir satır ≤ A değildir: durum güvensizdir ve istek ertelenir.

"Her satırda ≤ A olan satırı ara" adımları burada güvenlik testidir. Aynı mantık tespit algoritmasında istek matrisiyle, kaçınmada kalan ihtiyaç matrisiyle kullanılır. İki algoritmanın kodu neredeyse aynıdır; değişen yalnızca karşılaştırılan satırın anlamıdır.

7.6.12 Bankacı algoritması pratikte

Bankacı algoritması zarif bir fikirdir, ama dört varsayıma dayanır:

- Her süreç azami ihtiyacını önceden bildirir.
- Süreç sayısı sabittir.
- Kaynak sayısı değişmez; örneğin bir aygıt bozulmaz.
- Süreçler kaynakları sonlu bir sürede bırakır.

Bu varsayımlar genel amaçlı işletim sistemlerinde tutmaz. Bir kelime işlemcinin ne kadar bellek ya da kaç dosya kilidi isteyeceği önceden bilinmez; süreçler sürekli doğar ve ölür; bir disk bozulabilir. Bu yüzden algoritma gerçek işletim sistemlerinde neredeyse hiç kullanılmaz. Benzer bir "kabul denetimi" fikri ise yaygındır. Gerçek zamanlı çizelgelemede yeni görevin öteki görevlerin zaman garantilerini bozup bozmadığına bakılır. Bulut sistemlerinde kotalar, bağlantı havuzlarında da üst sınırlar konur.

Kaçınma, her istekte $O(m \cdot n^2)$ maliyetli bir test ve gereksiz ertelemeler yüzünden tespit ve kurtarmadan genellikle daha pahalıdır. Avantajı kurtarma gerektirmemesidir. Algoritma arka planda sürekli çalışmaz: test yalnız bir istek geldiğinde ve bir kaynak bırakıldığında bekleyen istekler için yapılır.

Kilitlenme ve bankacı algoritması simülatörü, Allocation, Max, Need ve Available tablolarıyla güvenli dizi aramayı ve istek denetimini adım adım çalıştırır. Bu kesimdeki örnekler simülatörde de denenebilir.

7.7 Kilitlenmeyi önleme

Kaçınma, süreçlerin gelecekteki isteklerini bilmeyi gerektirir ve bu bilgi pratikte yoktur. Önleme ise daha kökten bir yol izler: dört koşul birlikte gerekli olduğuna göre, birini sistemden tümüyle kaldırmak kilitlenmeyi imkânsız kılar. Karar çalışma anında değil, tasarımda verilir.

7.7.1 Koşullara saldırmak

Tablo 7.11 her koşul için kullanılan yöntemi ve bedelini özetler.

Koşul	Saldırı	Bedeli
Karşılıklı dışlama	Kaynağı paylaşılabilir yapmak: biriktirme, salt okunur paylaşım, kilitsiz yapılar	Çoğu kaynak doğası gereği paylaşılamaz (mutex!)
Tut ve bekle	Bütün kaynakları başta tek seferde istemek ya da yeni istekten önce elindekileri bırakmak	Düşük kullanım, açlık; ihtiyaç önceden bilinmeli
Kesme yok	Bekleyecek sürecin kaynaklarını geri almak; tryLock ile geri çekilmek; sanallaştırmak	Yalnız durumu kaydedilebilen kaynaklarda; canlı kilit riski
Döngüsel bekleme	Kaynaklara toplam bir sıra vermek ve yalnız artan sırada istemek (kilit sıralaması)	Sıraya uymak programcı disiplini ister

Tablo 7.11: Dört koşula saldırı yöntemleri.

İki yanlış anlama sık görülür. Karşılıklı dışlamayı kaldırmak "her seferinde yalnız bir süreç kaynak isteyebilir" demek değildir; bu, dışlamanın kendisidir. "Kesme yok" koşulunu kaldırmak da "süreç bitene kadar kaynakları tutsun" demek değildir; tersine, kaynakların sahibinden geri alınabilmesi demektir.

7.7.2 Karşılıklı dışlamaya saldırı

Bir kaynak aynı anda birden çok sürece verilebiliyorsa kimse onu beklemez ve kilitlenmeye katılamaz. Yazıcı bunun klasik örneğidir.

Tanım 7.8

Biriktirme (*spooling*), yavaş bir aygıta gidecek işleri diskte sıraya alıp aygıta sırayla göndermektir. Süreçler çıktılarını bir biriktirme dizinine yazar; aygıtı yalnız tek bir arka plan süreci (*daemon*) kullanır.



Şekil 7.34: Yazıcı biriktirme. P1, P2 ve P3 çıktılarını biriktirme dizinine yazar; yazıcıyı yalnız `lpd` arka plan sü-

reci tutar. Kullanıcı süreçleri yazıcıyı hiç beklemez.

Süreçler açısından yazıcı artık paylaşılabilir: herkes aynı anda iş bırakabilir, gerçek aygıtı tek bir süreç kullanır. Yazıcı için çekişme kalkar. Başka örnekler de vardır. Salt okunur dosyalar ve programların kod sayfaları aynı anda paylaşılır. Okur kilitleri birbirini dışlamaz; yalnız yazar kilidi dışlar. Kilitsiz (*lock-free*) veri yapıları karşılaştır ve değiştir (*compare-and-swap*, CAS) komutuyla ilerler; kilit olmadığı için kilitlenme de olmaz, ama açlık hâlâ mümkündür.

Yöntemin bir sınırı vardır: biriktirme diski de bir kaynaktır. Tanenbaum'un verdiği örnekte birçok süreç çıktısının yarısını diske yazmış ve disk dolmuştur. Hiçbir çıktı tamamlanmadığı için arka plan süreci basmaya başlamaz, süreçler de yer bekler: yine kilitlenme. Genel ilke şudur: bir kaynak gerçekten gerekmedikçe atanmaz ve onu isteyebilecek süreçlerin sayısı en aza indirilir.

7.7.3 Tut ve beklemeye saldırı

Hiçbir süreç bir kaynağı tutarken başka bir kaynak beklemezse halka kurulamaz. Bunu sağlamanın iki protokolü vardır:

1. **Protokol 1.** Süreç ihtiyaç duyacağı bütün kaynakları çalışmaya başlamadan önce tek bir istekte alır: hepsi ya da hiçbiri.
2. **Protokol 2.** Süreç yalnız elinde hiçbir kaynak yokken istek yapabilir. Yeni bir kaynak istemeden önce tuttuklarının hepsini bırakır.

Silberschatz'ın örneğinde bir süreç DVD'deki veriyi diske kopyalar, diskteki dosyayı sıralar ve sonucu yazıcıya basar. Protokol 1'de süreç DVD sürücüsünü, disk dosyasını ve yazıcıyı başta alır. Yazıcı yalnız en sonda gerektiği hâlde baştan sona tutulur ve kimse kullanamaz. Protokol 2'de süreç önce DVD ile diski alır, kopyalar ve ikisini bırakır; sonra diski ve yazıcıyı alır. Kullanım artar ama başka sorunlar çıkar: süreç ikinci kez diski istediğinde başka biri dosyayı değiştirmiş olabilir.



Şekil 7.35: İki protokolün zaman çizelgesi. Protokol 1'de yazıcı en baştan tutulur ve büyük kısmı boşa geçer.

Protokol 2'de kaynaklar yalnız kullanıldıkları sırada tutulur (Silberschatz §8.5.2).

İki protokolün ortak sorunları vardır. Kaynak kullanımı düşer, çünkü kaynaklar gerekmediği zamanlarda da tutulur. Çok sayıda popüler kaynak isteyen bir süreç, hepsi aynı anda boş olmadığı için hiç başlayamayabilir; bu da açlıktır. Ayrıca ihtiyaçlar çoğu zaman önceden bilinmez. Pratikte bu fikir iki yerde yaşar: `pthread_mutex_trylock` ile "ya hepsi ya hiçbiri" edinimi ve veritabanlarında bütün kilitleri baştan alan muhafazakâr iki aşamalı kilitleme.

7.7.4 Kesme yok koşuluna saldırı

Kaynaklar zorla geri alınabiliyorsa bekleme halkası kırılabilir. Silberschatz şu geri alma protokolünü verir: kaynak tutan bir süreç hemen karşılanamayan bir istek yaparsa, tuttuğu bütün kaynaklar örtük olarak bırakılır. Süreç, eski kaynaklarını ve yeni istediğini birlikte alabildiğinde yeniden başlar.

Bu protokol yalnız durumu kaydedilip geri yüklenebilen kaynaklara uygundur: işlemci yazmaçları, bellek, veritabanı işlemleri. Yarıda kesilen bir yazıcı işi ya da bir mutex ile korunan veri yapısı bu yolla geri alınamaz. Bazı kaynaklar sanallaştırılarak kesilebilir hâle getirilir: yazıcı biriktirilir, bellek sayfalanır.

Mutex için uygulanabilir bir yol **zaman aşımlı denemedir**. İkinci kilidi belirli bir süre içinde alamayan iş parçacığı elindeki kilidi kendisi bırakır ve rastgele bir süre sonra yeniden dener. Kaynak zorla alınmaz, ama sahibi onu gönüllü olarak bırakır; sonuç aynıdır, halka kırılır.

7.7.5 Kod izleme: tryLock ile geri alma

Java'da `ReentrantLock.tryLock(süre, birim)` kilidi en çok verilen süre kadar bekler; alırsa `true`, alamazsa `false` döndürür. Aşağıdaki yöntem bir hesaptan ötekine para aktarır. Her hesabın kendi kilidi vardır.

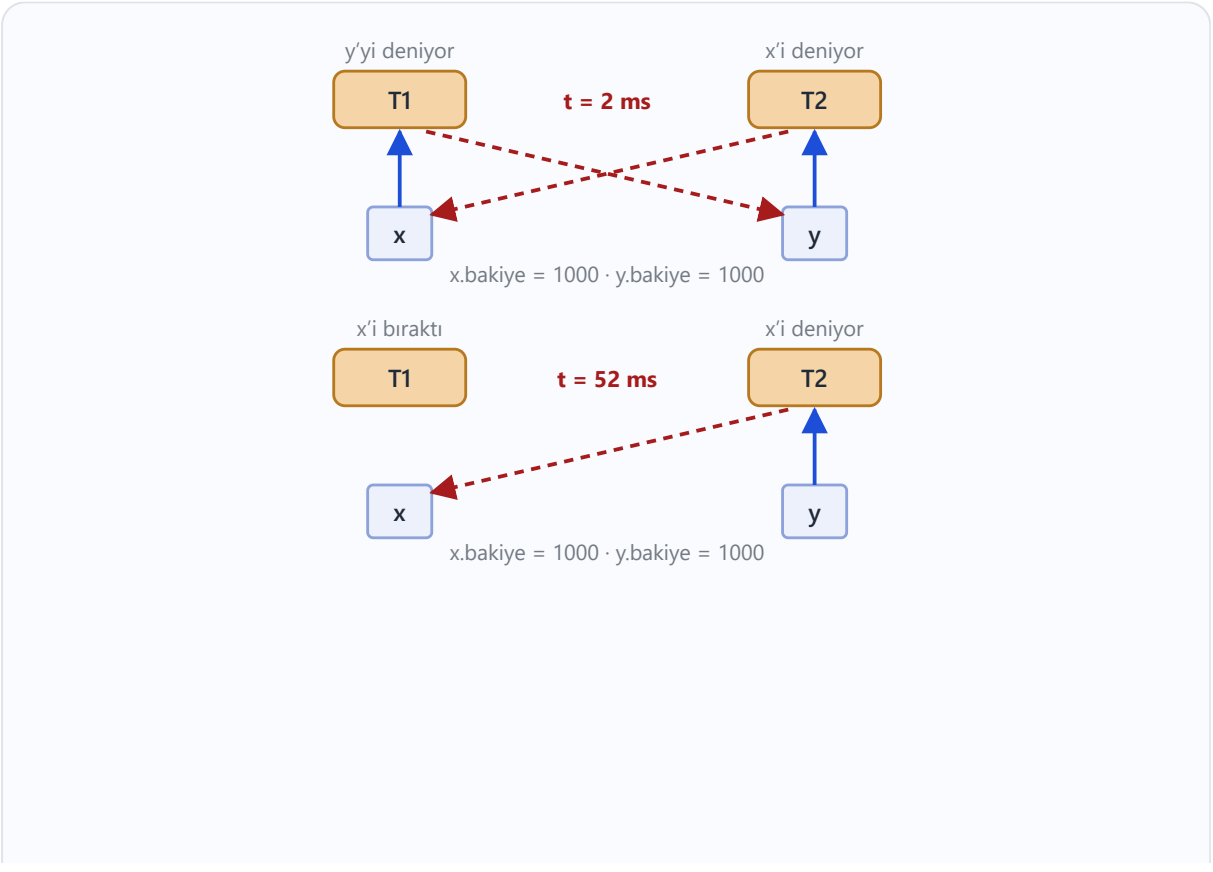
```
boolean dene(Hesap k, Hesap h, int m)
        throws InterruptedException {
  if (!k.kilit.tryLock(50, MILLISECONDS))
    return false;
  try {
    if (!h.kilit.tryLock(50, MILLISECONDS))
      return false;  /* k'yi de bırak */
    try { k.bakiye -= m; h.bakiye += m;
          return true; }
    finally { h.kilit.unlock(); }
  } finally { k.kilit.unlock(); }
}
/* çağıran: while (!dene(x, y, 1))
              Thread.sleep(rastgele); */
```

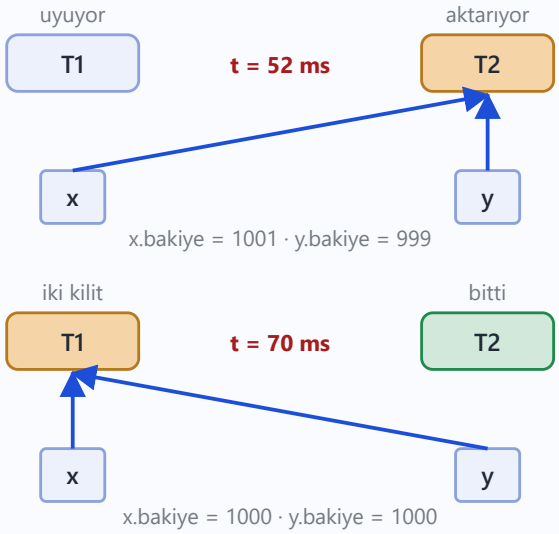
Yöntem önce kaynak hesabın (`k`) kilidini en çok 50 ms bekler. Alamazsa hemen `false` döner. Alırsa hedef hesabın (`h`) kilidini de 50 ms bekler. İkinci kilit alınamazsa `return false` çalışır; ama dıştaki `finally` bloğu dönüşten önce `k` 'nin kilidini bırakır. Yani "k'yi de bırak" yorumunun gerçekleşmesini `finally` sağlar. İki kilit de alınırsa aktarım yapılır ve kilitler ters sırada bırakılır. Çağıran taraf yöntem başarılı olana kadar rastgele sürelerle uyuyup yeniden dener.

İki iş parçacığı ters yönde aktarım yapsın: T1 `x` 'ten `y` 'ye, T2 `y` 'den `x` 'e 1 birim. İki hesapta da başlangıçta 1000 birim vardır. Tablo 7.12 bir kötü zamanlamayı izler.

t (ms)	T1	T2	x kilidi	y kilidi	x / y bakiye
0	hazır	hazır	boş	boş	1000 / 1000
0	x'i hemen alır	hazır	T1	boş	1000 / 1000
1	x'i tutar	y'yi alır	T1	T2	1000 / 1000
2	y'yi dener (≤ 50 ms)	x'i dener (≤ 50 ms)	T1	T2	1000 / 1000
52	süre doldu: `false`	x'i dener	T1	T2	1000 / 1000
52	`finally` : x'i bırakır	x'i dener	boş	T2	1000 / 1000
52	rastgele uyur	x'i alır	T2	T2	1000 / 1000
52	uyuyor	aktarır	T2	T2	1001 / 999
53	uyuyor	önce x'i, sonra y'yi bırakır; `true`	boş	boş	1001 / 999
70	yeniden dener, iki kilidi alır ve aktarır	bitti	T1	T1	1000 / 1000

Tablo 7.12: tryLock ile geri almanın izlemesi.





Şekil 7.36: tryLock ile geri alma. (a) İkisi de ikinci kilidi en çok 50 ms bekler: döngüsel bekleme vardır ama süreli. (b) T1'in süresi dolar ve `finally` x'i bırakır. (c) T2 x'i alır ve aktarır. (d) T1 rastgele uyuduktan sonra yeniden dener; toplam yine 2000'dir.

Adım 2'de döngüsel bekleme gerçekten oluşur, ama süresiz değildir. 52. milisaniyede T1 vazgeçip `x`'i bırakır; kesme yok koşulu gönüllü olarak kırılmıştır. Düz `lock()` kullanılsaydı kilitlenme t = 2 ms anında kesinleşirdi. Program JDK 21 ile 2 iş parçacığı ve 1000'er ters yönlü aktarımla çalıştırıldı: sonuç `x=1000 y=1000 toplam=2000`'dir ve program her seferinde bitmiştir.

Rastgele bekleme önemlidir. İki iş parçacığı hep aynı süre bekleyip aynı anda yeniden denerse aynı çakışma tekrar tekrar yaşanabilir. Süreçler bloke olmaz ama hiçbiri ilerlemez; bu duruma canlı kilit denir.

7.7.6 Döngüsel beklemeye saldırı

Pratikte en çok kullanılan önleme yöntemi döngüsel beklemeyi kırmaktır. Her kaynak türüne bir numara verilir ve süreçlerin kaynakları yalnız artan numara sırasıyla istemesine izin verilir.

#	Kaynak
1	Görüntüleyici
2	Yazıcı
3	Çizici
4	Teyp sürücüsü
5	Blu-ray sürücüsü

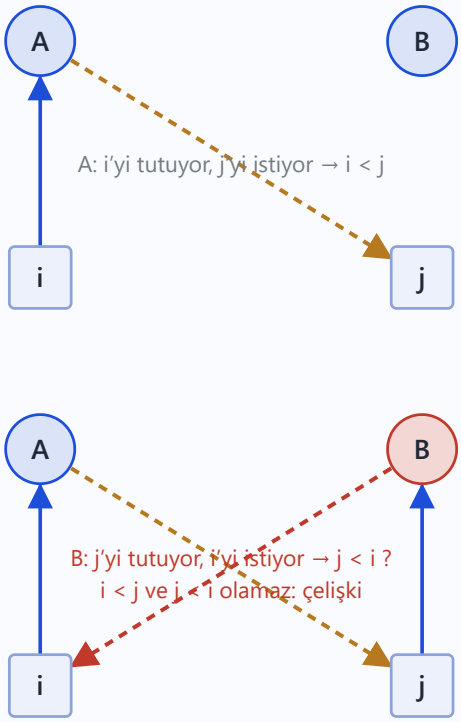
Tablo 7.13: Kaynak numaralandırma örneği (Tanenbaum).

Yazıcı ve teyp gereken bir süreç önce 2'yi, sonra 4'ü ister. Teybi tuttuktan sonra yazıcıyı isteyemez.

Teorem 7.3 (Kaynak sıralaması)

Her süreç kaynakları yalnız elindekilerden büyük numaralı olacak biçimde istiyorsa döngüsel bekleme oluşamaz.

İspat. Önce iki süreçli durumu düşünelim. A, i numaralı kaynağı tutup j'yi istiyorsa kurala göre $i < j$'dir. B, j'yi tutup i'yi isterse $j < i$ olmalıdır. İkisi birden doğru olamaz; bu kenar hiç oluşmaz. Genel durumda bir döngü olduğunu varsayalım ve döngüdeki en büyük numaralı kaynağa R_k diyelim. R_k'yi tutan süreç döngüdeki bir sonraki kaynağı bekler. O kaynağın numarası k'den küçüktür, çünkü k en büyüktür. Ama kurala göre süreç elindekinden küçük numaralı bir kaynak isteyemez. Çelişki; döngü yoktur.



Şekil 7.37: Numaralandırma döngüyü kırar (Tanenbaum Şekil 6-13). (a) A, i'yi tutup j'yi istiyorsa $i < j$. (b) B'nin j'yi tutup i'yi istemesi $j < i$ gerektirir: çelişki, bu kenar oluşamaz.

Kuralın bir çeşidi de vardır: süreç K_j'yi isteyecekse elindeki bütün K_i ($i \geq j$) kaynaklarını önce bırakır. Bu yöntemin en zor kısmı, bütün programların uyacağı uygun bir sıralamayı bulmaktır. Bir süreç işinin doğal akışında teybi yazıcıdan önce kullanıyorsa, kurala uymak için yazıcıyı gereğinden önce almak zorunda kalır.

Örnek 7.11

Kaynaklar Tablo 7.13'teki gibi numaralanmıştır. Süreçler hiçbir kaynağı bırakmadan şu istek dizilerini yapıyor. Hangisi kurala uyar? (a) Teyp → Yazıcı → Blu-ray, (b) Çizici → Görüntüleyici → Teyp, (c) Blu-ray → Çizici → Teyp, (d) Yazıcı → Teyp → Blu-ray.

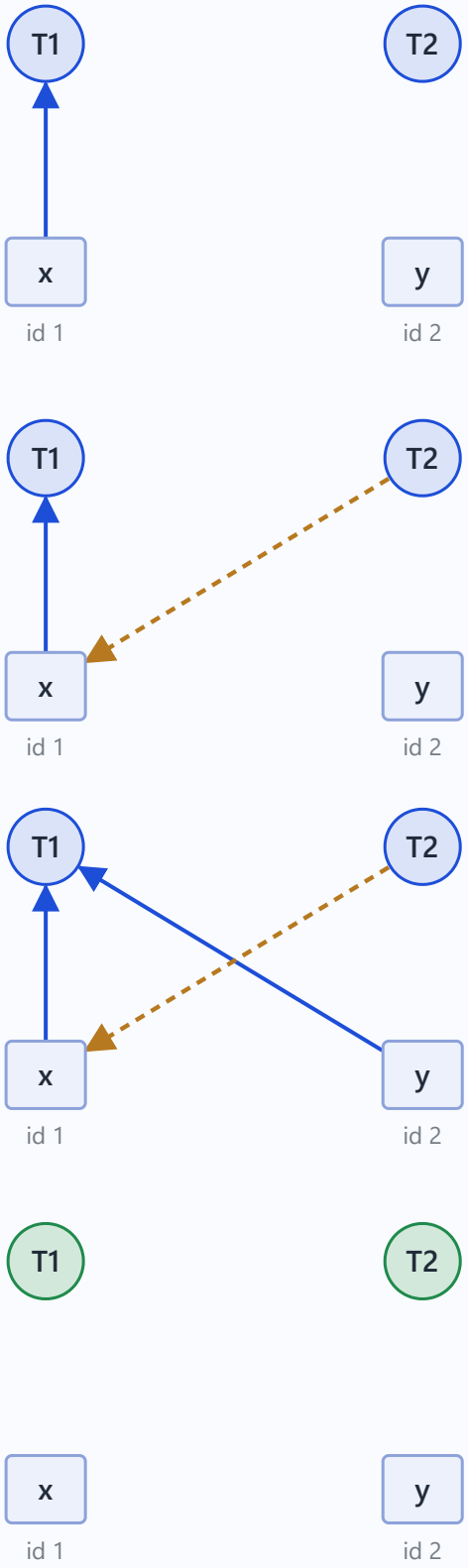
1. **(a)** 4 → 2 → 5: teyp (4) tutulurken yazıcı (2) istenir; kurala aykırıdır.
2. **(b)** 3 → 1 → 4: çizici (3) tutulurken görüntüleyici (1) istenir; aykırıdır.
3. **(c)** 5 → 3 → 4: Blu-ray (5) tutulurken çizici (3) istenir; aykırıdır.
4. **(d)** 2 → 4 → 5: kesin artan sıradadır; kurala uyan tek dizidir.

7.7.7 Kilit sıralaması pratikte

Kaynak numaralandırmanın yazılımdaki karşılığı **kilit sıralamasıdır**. Ters yönlü aktarım örneğine dönelim. T1 `x`'ten `y`'ye, T2 `y`'den `x`'e aktarıyor. "Doğal" sıra, yani önce kaynak hesabı, sonra hedef hesabı kilitlemek, iki iş parçacığında terstir ve kilitlenmeye açıktır. Çözüm, kilitleri hesapların kimlik numarasına göre almaktır; bu sıra bütün iş parçacıkları için ortaktır.

```
void transfer(Hesap k, Hesap h, int m) {
  Hesap ilk = k.id < h.id ? k : h;
  Hesap ikinci = k.id < h.id ? h : k;
  ilk.kilit.lock();
  try {
    ikinci.kilit.lock();
    try { k.bakiye -= m; h.bakiye += m; }
    finally { ikinci.kilit.unlock(); }
  } finally { ilk.kilit.unlock(); }
}
/* T1: transfer(x, y, 1)   x.id = 1
   T2: transfer(y, x, 1)   y.id = 2 */
```

`ilk` her zaman numarası küçük olan hesaptır, `ikinci` numarası büyük olan. Hangi yönde aktarım yapılırsa yapılsın kilitler aynı sırayla alınır. Aktarım satırı ise `k` ve `h` üzerinden yazıldığı için yön korunur. Şekil 7.38 iki iş parçacığının izini gösterir.



Şekil 7.38: Kilit sıralaması. (a) T1 için `ilk` = x (1 < 2); x'i alır. (b) T2 için `k` = y ama `ilk` yine x'tir; T2 x'i bekler ve elinde hiçbir kilit yoktur. (c) T1 y'yi de alır ve aktarır. (d) T1 kilitleri bırakır; T2 önce x'i, sonra y'yi alır. Döngü oluşamaz.

1. **T1**, `transfer(x, y, 1)` çağırır. x.id = 1 < y.id = 2 olduğu için `ilk` = x'tir. T1 x'i alır.
2. **T2**, `transfer(y, x, 1)` çağırır. `k` = y'dir ama `ilk` yine x'tir (id 1). T2 x'i bekler. Önemli olan, T2'nin beklerken elinde hiçbir kilit tutmamasıdır.
3. **T1**, y'yi de alır; y'yi kimse tutmuyordur. Aktarımı yapar.
4. **T1** kilitleri bırakır. T2 x'i, sonra y'yi alır ve kendi aktarımını yapar.

Program JDK 21 ile her iş parçacığı 100.000 ters yönlü aktarım yaparak çalıştırıldı; sonuç `x=1000 y=1000`'dir ve kilitlenme olmamıştır. Linux çekirdeği de aynı yöntemi kullanır: aynı türden iki kilidi bellekteki adres sırasıyla alır. Çizelgeleyicinin iki çalışma kuyruğunu birlikte kilitleyen `double_rq_lock` fonksiyonu buna örnektir. Çekirdekteki kilit sırası belgelenir ve `lockdep` ile denetlenir.

Sık yapılan hata

Kilit sırasını yalnız bir fonksiyonda uygulamak. Sıralama ancak kilitlerin alındığı *her* yerde uygulanırsa işe yarar. Bir başka fonksiyon aynı iki kilidi ters sırada alıyorsa döngü yine kurulabilir. Bu yüzden büyük projelerde kilit sırası belgelenir ve araçlarla denetlenir.

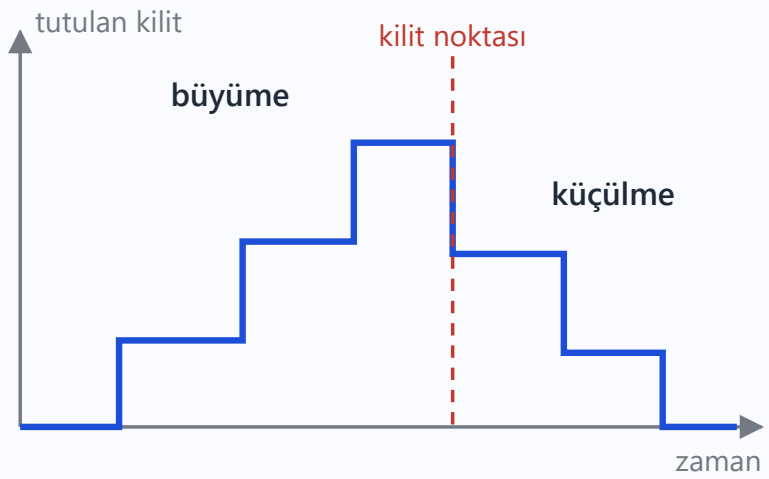
7.8 Diğer sorunlar

Kilitlenme, eş zamanlı programların tek sorunu değildir. Veritabanlarının kilitleme protokolü, iletilerle haberleşen süreçler, sürekli deneyen ama ilerlemeyen süreçler ve hiç sırası gelmeyen süreçler benzer ama farklı sorunlar doğurur. Bu kesim onları kilitlenmeden ayırır.

7.8.1 İki aşamalı kilitleme

Tanım 7.9

İki aşamalı kilitleme (*two-phase locking*, 2PL), veritabanı işlemleri için bir protokoldür. **Büyüme** evresinde işlem yalnız kilit alır, **küçülme** evresinde yalnız bırakır. İlk bırakıştan sonra yeni kilit alınamaz.



Şekil 7.39: İki aşamalı kilitleme. Tutulan kilit sayısı büyüme evresinde artar, kilit noktasından sonra küçülme evresinde azalır.

2PL'nin amacı kilitlenmeyi önlemek değil, **serileştirilebilirliktir**: eş zamanlı çalışan işlemlerin sonucu, işlemler sanki sırayla çalışmış gibi olur. Düz 2PL kilitlenebilir. T1 *a* kaydını, T2 *b* kaydını kilitler; sonra T1 *b*'yi, T2 *a*'yı ister. İki işlem de büyüme evresindedir ve birbirini bekler.

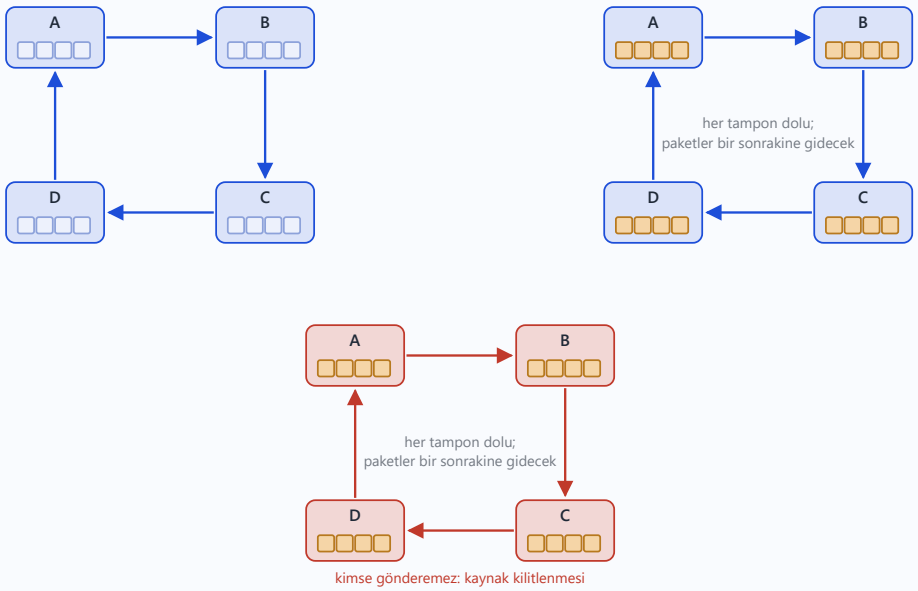
Tanenbaum bir çeşidini anlatır: birinci evrede işlem bütün kayıtları kilitlemeyi dener. Kayıtlardan birini başka bir işlem kilitlemişse elindekilerin hepsini bırakır ve baştan başlar. Bu çeşit tut ve beklemeyi kaldırır, çünkü işlem hiçbir zaman kilit tutarken beklemez. Günümüz veritabanlarında yaygın olan **katı** (*strict*) 2PL'de ise kilitler işlem sonunda, onaylama ya da iptal anında bırakılır. Bu çeşit de kilitlenebilir; veritabanı yönetim sistemi kilitlenmeyi bekleme çizgesiyle tespit eder ve işlemlerden birini geri sarar.

7.8.2 İletişim kilitlenmeleri

Şimdiye kadarki kilitlenmelerin hepsi kaynak kilitlenmesiydi. İletilerle haberleşen süreçlerde kaynak olmadan da kilitlenme olabilir. A, B'ye bir istek gönderir ve yanıt bekler. İstek ağda kaybolursa A yanıtı, B de isteği sonsuza dek bekler. Her süreç yalnız ötekinin tetikleyebileceği bir olayı beklediği için bu bir kilitlenmedir; ama ortada tutulan bir kaynak yoktur.

Bu tür **iletişim kilitlenmesinde** kaynak sıralaması ya da kaçınma işe yaramaz, çünkü sıralanacak ya da hesaplanacak kaynak yoktur. Çözüm zaman aşımıdır: gönderen belirli bir süre yanıt alamazsa iletiyi yeniden gönderir. Bu da yeni bir sorun doğurur: asıl ileti kaybolmamış, yalnız gecikmişse alıcı aynı isteği iki kez alır. Yinelenen iletileri ayırt etmek için protokolde sıra numarası gerekir.

Ağ donanımında ise tamponlar gerçek kaynaktır. Şekil 7.40 dört yönlendiricinin halka biçiminde bağlandığı bir ağı gösterir.



Şekil 7.40: Yönlendiricilerde kaynak kilitlenmesi (Tanenbaum §6.7.2). (a) Dört yönlendirici halka biçimindedir; her birinin tampon belleği sınırlıdır. (b) Yoğun trafikte bütün tamponlar saat yönünde bir sonraki yönlendiriciye gidecek paketlerle dolar. (c) Kimse gönderemez: A, B'nin boş tamponunu; B, C'ninkini; C, D'ninkini; D de A'nınkini bekler.

Bu kez kilitlenme bir kaynak kilitlenmesidir ve döngüsel bekleme açıkça görülür. Çözümler de önleme fikirlerine dayanır: zaman aşımıyla paket düşürmek (kesme), sanal kanallar kullanmak ve kredi tabanlı akış denetimiyle tamponların dolmasını baştan engellemek.

7.8.3 Canlı kilit

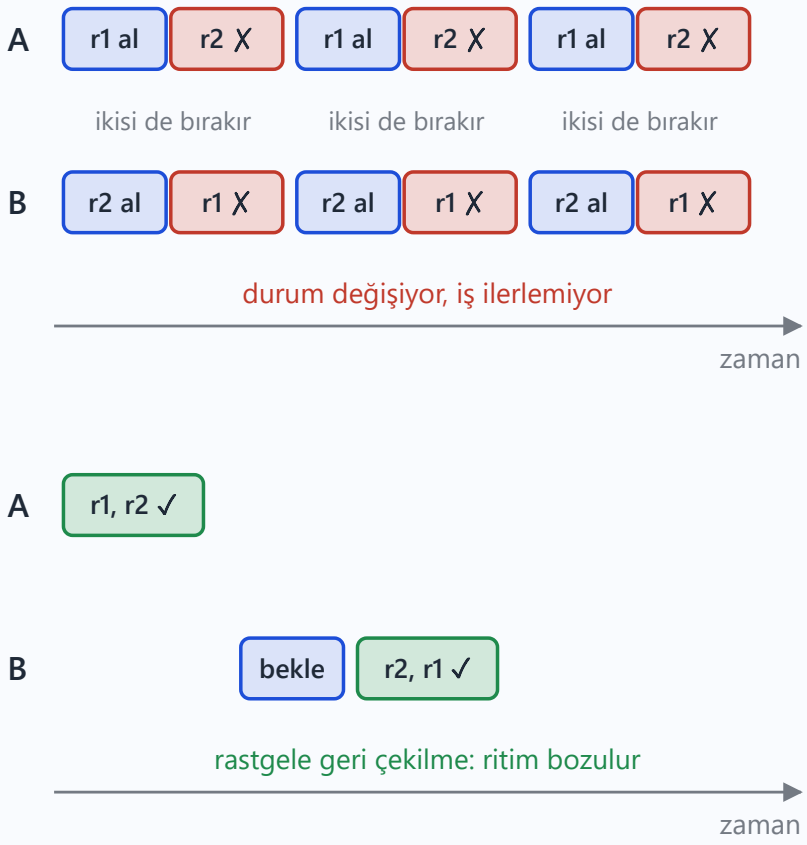
Dar bir koridorda karşılaşan iki kişi düşünelim. İkisi de kibar davranıp aynı anda sağa kayar, sonra aynı anda sola kayar. Kimse duvara çarpmaz, ikisi de sürekli hareket eder, ama kimse geçemez.

> **Tanım 7.10**
>
> **Canlı kilit** (*livelock*), süreçlerin bloke olmadığı ve durumlarını sürekli değiştirdiği, ama hiçbirinin ilerleyemediği durumdur. Kilitlenmede süreçler uyur; canlı kilitte çalışır ama faydalı iş yapmaz.

Tanenbaum'un örneğinde süreçler kaynakları meşgul beklemeyle alır. `enter_region` kilidi alamazsa döngüde dönmeye devam eder:

```
void process_A(void) {
  enter_region(&resource_1);
  enter_region(&resource_2);
  use_both_resources();
  leave_region(&resource_2);
  leave_region(&resource_1);
}  /* B: önce resource_2, sonra 1 */
```

B, kaynakları ters sırada alırsa A `resource_1`'i, B `resource_2`'yi alır. Sonra ikisi de ötekini alamadığı için sonsuza dek döner. Süreçler bloke değildir ve işlemci harcarlar; yine de ilerlemezler. Daha tipik bir canlı kilit, tryLock ile geri çekilen iki iş parçacığında görülür:



Şekil 7.41: Canlı kilit. Üstte: A r1'i, B r2'yi alır; ikisi de ötekini dener, başaramaz ve aynı anda bırakır. Aynı ritim tekrarlanır: durum değişir, iş ilerlemez. Altta: rastgele geri çekilme ritmi bozar ve iki süreç de kaynaklarını alır.

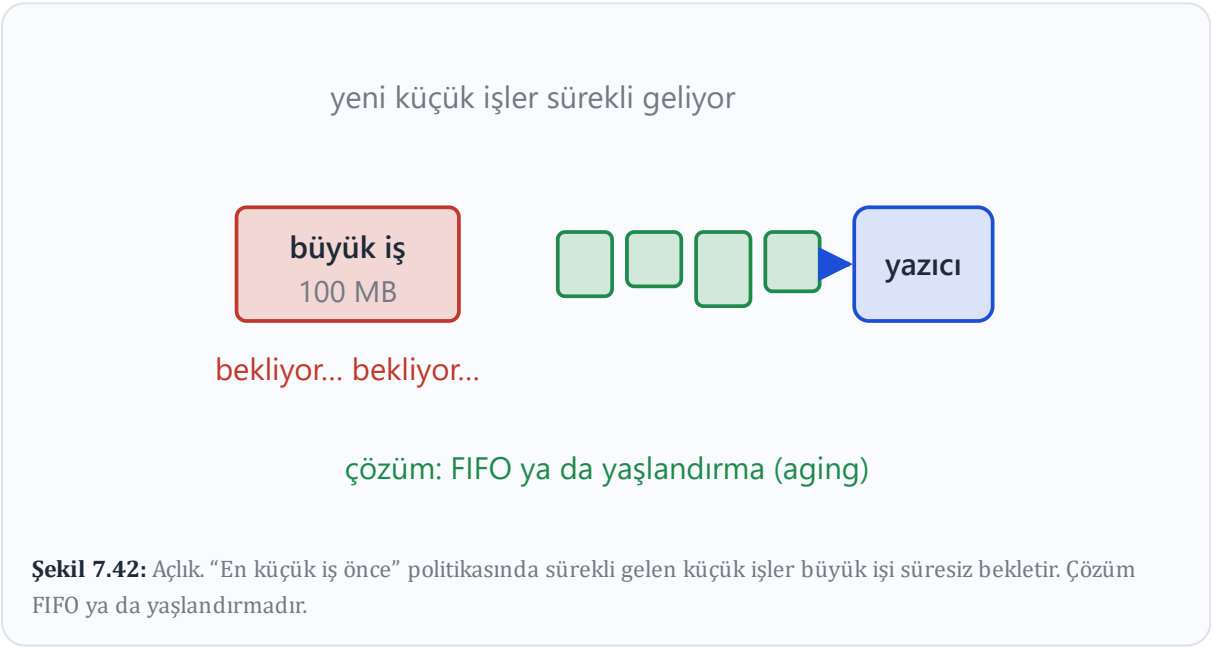
1. A r1'i, B r2'yi alır. A r2'yi dener, B r1'i dener; iki deneme de başarısız olur.
2. İkisi de kibar davranır: elindekini bırakır ve aynı süre sonra yeniden dener.
3. Aynı şey tekrarlanır. Süreçler bloke değildir, işlemci harcarlar ve durumları sürekli değişir; ama hiçbiri ilerlemez.

Çözüm ritmi bozmaktır. Rastgele ve her başarısızlıkta artan (*exponential backoff*) bir bekleme süresi, iki sürecin aynı anda yeniden denemesini olasılıksız kılar. Öncelik ya da sıra numarası vermek ve kilit sıralaması kullanmak da canlı kilidi önler. Ethernet'in CSMA/CD protokolündeki üstel geri çekilme aynı fikre dayanır: çarpışan iki istasyon rastgele sürelerle bekler.

7.8.4 Açlık

> **Tanım 7.11**
>
> **Açlık** (*starvation*), kaynak kullanılabilir olduğu hâlde bir sürece hiç sıra gelmediği için sürecin süresiz beklemesidir. Öteki süreçler ilerler; sistem kilitlenmemiştir.

Açlık, kaynak dağıtım politikasının sonucudur. Örneğin yazıcıyı her seferinde en küçük dosyaya vermek ortalama bekleme süresini azaltır. Ama sürekli yeni küçük işler geliyorsa 100 MB'lık büyük dosyaya hiç sıra gelmez.



Şekil 7.42: Açlık. "En küçük iş önce" politikasında sürekli gelen küçük işler büyük işi süresiz bekletir. Çözüm FIFO ya da yaşlandırmadır.

Başka açlık örnekleri de vardır: katı öncelikli çizelgelemede düşük öncelikli bir iş, okur öncelikli okur-yazar kilidinde bir yazar, kurtarmada her seferinde kurban seçilen bir süreç. Çözüm ya ilk gelen ilk hizmet alır (FIFO) politikasıdır ya da bekledikçe sürecin önceliğini artıran **yaşlandırmadır** (*aging*). Tablo 7.14 üç sorunu karşılaştırır.

	Kilitlenme	Canlı kilit	Açlık
Süreç durumu	bloke	çalışıyor	hazır ya da bloke
Kim etkilenir?	döngüdeki herkes	etkileşen grup	bir ya da birkaç süreç
Sistem ilerliyor mu?	döngü dışındakiler ilerler	grup ilerlemez	evet, yalnız aç kalan ilerlemez

	Kilitlenme	Canlı kilit	Açlık
Tipik çare	sıralama, tespit, kaçınma	rastgele geri çekilme	FIFO, yaşlandırma

Tablo 7.14: Kilitlenme, canlı kilit ve açlığın karşılaştırması.

7.8.5 Kilitlenme dışı hatalar: atomiklik ve sıra

Eş zamanlılık hatası denince akla ilk kilitlenme gelir, ama en yaygın hata kilitlenme değildir. Lu ve arkadaşları 2008'de MySQL, Apache, Mozilla ve OpenOffice'te 105 eş zamanlılık hatası incelemiştir. Bunların 74'ü kilitlenme dışıdır ve neredeyse hepsi iki kalıba uyar.

Atomiklik ihlali. Birlikte atomik olması gereken iki erişimin arasına başka bir iş parçacığı girer.

```
/* T1 */
if (oturum->ad != NULL)
    puts(oturum->ad);  /* artık NULL? */
/* T2 */
oturum->ad = NULL;
```

T1 önce `oturum->ad`'ın `NULL` olmadığını sınar, sonra onu yazdırır. İki satır arasında T2 alanı `NULL` yaparsa `puts` boş bir işaretçiyle çağrılır ve program çöker. Çare, sınama ile kullanımı ve T2'nin yazmasını aynı mutex altına almaktır.

Sıra ihlali. "A, B'den önce olur" diye varsayılır, ama bu sıra hiçbir şeyle zorlanmaz.

```
/* T1: baslat() */
isci = thread_olustur(govde);
/* T2: govde(), yeni iş parçacığı */
d = isci->durum;  /* isci atandı mı? */
```

T1, yeni iş parçacığını oluşturur ve tanıtıcısını `isci` değişkenine atar. Yeni iş parçacığı (T2) ise ilk satırında `isci->durum`'a erişir. Programcı atamanın önce yapılacağını varsaymıştır; ama yeni iş parçacığı, `thread_olustur` dönüp atama yapılmadan önce çalışmaya başlayabilir. O zaman `isci` henüz atanmamıştır. Çare, bir `hazir` bayrağı ve bir koşul değişkenidir: T2 `while (!hazir) wait` ile bekler, T1 atamadan sonra `hazir = 1` yapıp `signal` gönderir. Sıfırdan başlayan bir semafor da aynı işi görür.

> **Sık yapılan hata**
>
> "Her erişim kilitliyse kod doğrudur" diye düşünmek. Sınama ve kullanım ayrı ayrı kilitli olsa da aralarında kilit bırakılıyorsa atomiklik ihlali sürer. ThreadSanitizer gibi veri yarışı araçları bu durumu göremeyebilir, çünkü her erişim tek başına korunmuştur. Atomik olması gereken birim, tek tek erişimler değil, sınama ile kullanımın bütünüdür.

7.9 Pratikte kilitlenme

Genel amaçlı işletim sistemleri kullanıcı programlarının kilitlenmesiyle ilgilenmez; sorumluluk programcıdadır. Bu kesim kilitlenmenin günlük kodda nasıl ortaya çıktığını, kapsüllemenin onu nasıl gizlediğini ve hangi araçlarla bulunduğunu Java ve Linux örnekleriyle anlatır. Bu kesimdeki Java programları JDK 21 ile derlenip çalıştırılmıştır.

7.9.1 Java'da kilitlenme

Java'da her nesnenin bir **monitörü** vardır. `synchronized (a) { … }` bloğu `a`'nın monitörünü alır ve blok bitince bırakır. Monitör aynı anda tek bir iş parçacığına ait olabilir; bu, karşılıklı dışlamadır. Aşağıdaki iki iş parçacığı iki monitörü ters sırada alır:

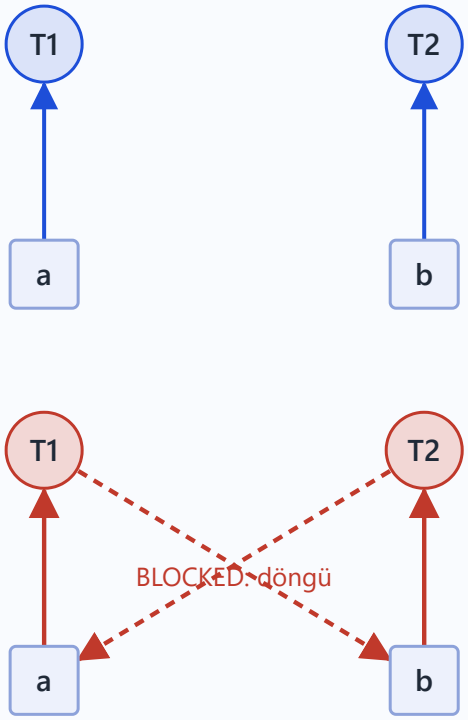
```
Object a = new Object(), b = new Object();

Thread t1 = new Thread(() -> {
  synchronized (a) {
    uyu(10);   /* pencereyi büyüt */
    synchronized (b) { isle(); }
  }
}, "T1");
Thread t2 = new Thread(() -> {
  synchronized (b) {
    uyu(10);
    synchronized (a) { isle(); }
  }
}, "T2");
t1.start(); t2.start();
```

`uyu(10)`, `Thread.sleep(10)`'u çağıran yardımcı bir yöntemdir; ilk kilit alındıktan sonra kilitlenme penceresini büyütür. `isle()` ise iki kilit altında yapılan işi temsil eder. Java iş parçacıklarının durumları da kilitlenmeyi izlemeye yardım eder: `RUNNABLE` çalışmaya hazır ya da çalışıyor, `TIMED_WAITING` süreli uykuda, `BLOCKED` ise bir monitörü bekliyor demektir.

Adım	Olay	a'nın sahibi	b'nin sahibi	T1	T2
1	`t1.start(); t2.start();`	—	—	RUNNABLE	RUNNABLE
2	T1 a'nın monitörüne girer ve uyur	T1	—	TIMED_WAITING	RUNNABLE
3	T2 b'nin monitörüne girer ve uyur	T1	T2	TIMED_WAITING	TIMED_WAITING
4	T1 uyanır, b'yi ister: b T2'de	T1	T2	BLOCKED	TIMED_WAITING
5	T2 uyanır, a'yı ister: a T1'de	T1	T2	BLOCKED	BLOCKED

Tablo 7.15: Ters sıralı `synchronized` bloklarının izlemesi.



Şekil 7.43: Java'da kilitlenme. (a) T1 a'yı, T2 b'yi tutar. (b) İkisi de ötekinin monitörünü ister ve `BLOCKED` olur: döngü.

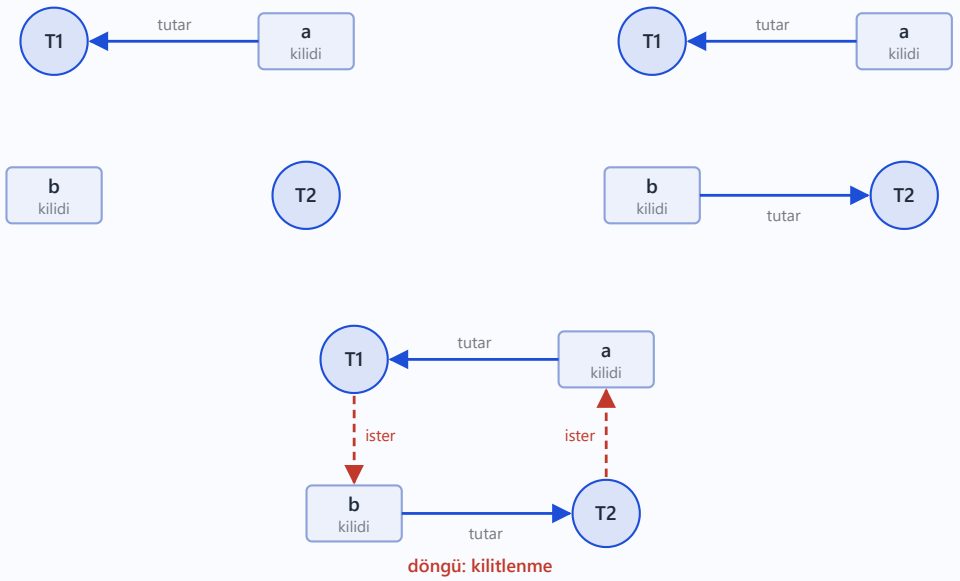
JVM monitörleri kesilemez; bekleyen bir `synchronized` bloğunu kesmenin, zaman aşımı koymanın ya da kilidi geri almanın yolu yoktur. Program asla bitmez. `uyu(10)` kilitlenmeyi çok olası kılar, ama garanti etmez: bir iş parçacığı ikinci kilidi öteki daha başlamadan alırsa program normal biter. Bu notun doğrulamasında program JDK 21 ile çalıştırıldı ve beş çalıştırmanın beşinde de iki iş parçacığı `BLOCKED` kaldı. 10 ms yerine 50 ms uyuyan sürüm de sekiz saniyelik zaman aşımıyla durdurulana kadar hiçbir şey yazmadı. İki ilk kilit alınana kadar iş parçacıklarını bir `CountDownLatch` ile bekleten sürüm ise her çalıştırmada kilitlenir, çünkü zamanlamaya hiç yer bırakmaz.

7.9.2 Kapsülleme kilit sırasını gizler

Ters sıralı kilitler her zaman kodda açıkça görünmez. `Collections.synchronizedList` bir listeyi sarar ve her yöntemini listenin kendi kilidiyle korur. `a.addAll(b)` çağrısı içeride önce `a` 'nın kilidini alır. Sonra argümanın elemanlarını kopyalamak için `b.toArray()` 'i çağırır; o da `b` 'nin kilidini alır.

```
var a = Collections.synchronizedList(
          new ArrayList<Integer>());
var b = Collections.synchronizedList(
          new ArrayList<Integer>());
/* a.addAll(b) içeride:              */
/* synchronized (a) { b.toArray() }  */
/*   → synchronized (b) { ... }      */
new Thread(() -> a.addAll(b)).start();
new Thread(() -> b.addAll(a)).start();

/* Düzeltme: kopyayı kilit dışında al */
Integer[] k = b.toArray(new Integer[0]);
a.addAll(Arrays.asList(k));
```



Şekil 7.44: Gizli kilit sırası. (a) T1: `a.addAll(b)` önce a'nın kilidini alır. (b) Aynı anda T2: `b.addAll(a)` önce b'nin kilidini alır. (c) İçeride `b.toArray()` ve `a.toArray()` çağrılır; her biri ötekinin kilidini ister.

İki iş parçacığı aynı anda `a.addAll(b)` ve `b.addAll(a)` çağırırsa ters sıralı edinim kendiliğinden oluşur: biri a → b, öteki b → a sırasıyla kilitler. Kodu yazan kişi hiçbir kilit görmez. Düzeltme, kopyayı kilit dışında almaktır: `b.toArray()` önce ayrı bir çağrıyla yapılır ve kendi kilidini bırakır; ardından `a.addAll` yalnız `a` 'nın kilidini alır. Hiçbir an iki kilit birlikte tutulmaz.

Çapraz çağrı	JDK 8	JDK 21
`Vector` : `a.addAll(b)` ‖ `b.addAll(a)`	10/10	0/10
`synchronizedList` : aynı	10/10	10/10
Düzeltilmiş sürüm	0/10	0/10

Tablo 7.16: Çapraz `addAll` denemeleri (2 iş parçacığı × 200.000 tur; 10 denemede kilitlenen; sonuç çalıştırmaya

göre değişebilir, ör. 9/10).

Tablodaki ilginç satır `Vector`'dür. JDK 8'de `Vector.addAll` kendi kilidini tutarken argümanın `toArray()`'ini çağırır ve kilitlenir. JDK 21'de ise argümanın `toArray()`'ini kendi kilidini almadan önce çağırır; bu yüzden kilitlenmez. Aynı kaynak kod, kütüphanenin iç ayrıntısı değiştiği için farklı davranır. Kural şudur: kilit tutulurken yabancı bir yöntem, yani iç kilitlemesi bilinmeyen bir yöntem çağrılmaz. Bu notun doğrulamasında aynı deney yeniden yapıldı: JDK 8'de `Vector` ve `synchronizedList` 10 denemenin 10'unda kilitlendi. JDK 21'de `synchronizedList` 10 denemenin 9'unda kilitlendi, `Vector` hiç kilitlenmedi. Düzeltilmiş kod iki sürümde de hiç kilitlenmedi. Kilitlenme olasılığa bağlı olduğu için 9 ile 10 arasındaki fark beklenen bir oynamadır.

7.9.3 Kilitlenmeyi bulmak: jstack

Kilitlenen bir Java programı hata iletisi vermez; yalnız durur. JDK ile gelen `jstack` aracı çalışan bir JVM'deki bütün iş parçacıklarının yığınlarını yazdırır ve kilitlenme varsa bunu raporun sonunda ayrıca belirtir:

```
$ jstack <pid>
Found one Java-level deadlock:
=============================
"T1":
  waiting to lock monitor 0x… (object 0x…, a java.lang.Object),
  which is held by "T2"
"T2":
  waiting to lock monitor 0x… (object 0x…, a java.lang.Object),
  which is held by "T1"

Java stack information for the threads listed above:
"T1":
        at Asili.calis(Asili.java:7)
        - waiting to lock <0x…d0> (a java.lang.Object)
        - locked <0x…c0> (a java.lang.Object)
…
Found 1 deadlock.
```

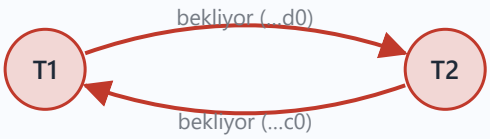
Rapor kimin hangi monitörü beklediğini ve o monitörün kimde olduğunu söyler: T1, T2'nin tuttuğu monitörü; T2 de T1'in tuttuğu monitörü bekler. Yığın bilgisi de kilitlenmenin kodun hangi satırında oluştuğunu gösterir. Bu notun doğrulamasında kilitlenen program JDK 21'de çalışırken `jstack` çalıştırıldı ve rapor aynı biçimde geldi: “Found one Java-level deadlock”, iki iş parçacığı için “waiting to lock” ve “which is held by” satırları, sonda da “Found 1 deadlock.”. Aynı rapor `jcmd <pid> Thread.print` ile de alınır; JConsole ve VisualVM gibi görsel araçlarda “Detect Deadlock” düğmesi aynı işi yapar.

Program kendi kilitlenmesini de bulabilir. `ThreadMXBean` arayüzü JVM'in iş parçacığı bilgisini verir:

```
ThreadMXBean mx =
    ManagementFactory.getThreadMXBean();
long[] ids = mx.findDeadlockedThreads();
if (ids != null)
  for (ThreadInfo ti : mx.getThreadInfo(ids))
    System.out.println(ti.getThreadName()
      + " bekliyor, kilit sahibi "
      + ti.getLockOwnerName());
```

`findDeadlockedThreads()`, kilitlenmiş iş parçacıklarının kimliklerini döndürür; kilitlenme yoksa `null` döndürür. Kilitlenen programda bu kod şu çıktıyı verir:

```
T1 bekliyor, kilit sahibi T2
T2 bekliyor, kilit sahibi T1
```



Şekil 7.45: JVM'in kurduğu bekleme çizgesi: T1, T2'nin tuttuğu nesneyi (...d0), T2 de T1'in tuttuğu nesneyi (...c0) bekler.

JVM, monitör ve `ReentrantLock` sahipliğinden bir bekleme çizgesi kurar ve döngü arar; yani tek örnekli kaynaklar için tespit algoritmasını çalıştırır. Ama yalnız tespit eder, kurtarmaz. JVM bir iş parçacığını öldürüp monitörünü başkasına veremez, çünkü korunan veri tutarsız kalabilir. Kilitlenen süreci yeniden başlatmak gerekir.

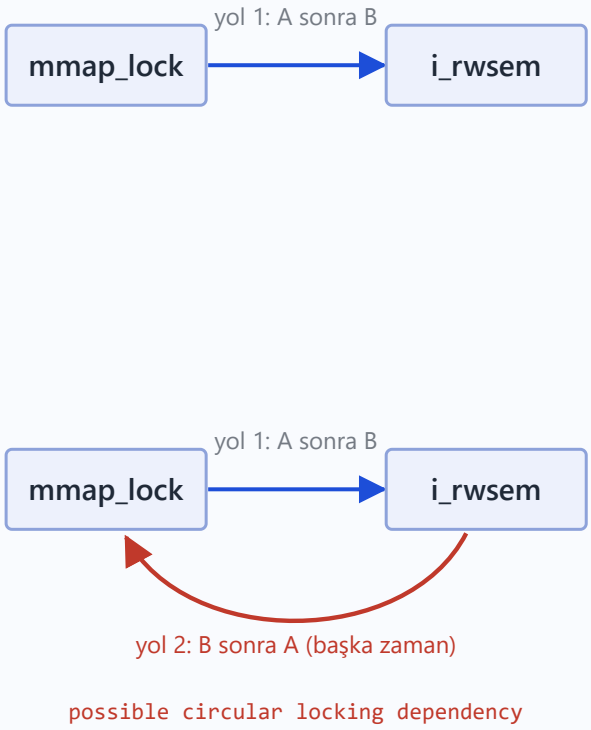
7.9.4 lockdep ve öteki araçlar

Her ortamın kilitlenme için kendi araçları vardır. Tablo 7.17 başlıcalarını listeler.

Ortam	Araç ya da davranış
Linux çekirdeği	`lockdep` (`CONFIG_PROVE_LOCKING`): kilit sırası doğrulayıcı; askıda görev (*hung task*) dedektörü
C/C++	ThreadSanitizer: `lock-order-inversion` uyarısı; Valgrind Helgrind
Java	`jstack`, `jcmd`, `ThreadMXBean`
Go	çalışma zamanı: `fatal error: all goroutines are asleep - deadlock!`
Veritabanı	PostgreSQL, InnoDB: bekleme çizgesi ve kurban işlemi geri sarma

Tablo 7.17: Kilitlenme bulma araçları.

Linux çekirdeğindeki `lockdep` farklı bir fikre dayanır. Her kilidi bir **sınıfa** bağlar; aynı türdeki bütün kilit örnekleri aynı sınıftır. Bir görev A sınıfından bir kilidi tutarken B sınıfından bir kilit alınca çizgeye "A → B" kenarı eklenir. Başka bir kod yolunda B tutulurken A alınırsa çizgede döngü oluşur ve `lockdep` "possible circular locking dependency" uyarısı verir.



Şekil 7.46: `lockdep`. (a) Bir yolda `mmap_lock` tutulurken `i_rwsem` alınır: A → B kenarı kaydedilir. (b) Başka bir zamanda başka bir yolda ters sıra görülür: döngü, kilitlenme hiç yaşanmadan rapor edilir.

İki kod yolu hiç aynı anda çalışmamış olabilir; `lockdep` yine de uyarır. Araçlar bu bakımdan ikiye ayrılır. `lockdep` ve ThreadSanitizer *olası* kilitlenmeyi, yani kilit sırası ihlalini bulur; kilitlenme hiç gerçekleşmemiş olsa bile. `jstack` ve Go çalışma zamanı ise *gerçekleşmiş* kilitlenmeyi raporlar. Birinci grup önleme, ikinci grup tespit stratejisinin araçlarıdır.

7.10 Özet

- **Kilitlenme**, bir süreç kümesindeki her sürecin yalnız kümedeki başka bir sürecin tetikleyebileceği bir olayı beklemesidir. Kilitlenmeler **kesilemez** kaynaklarla ilgilidir; işlemci ve bellek gibi **kesilebilir** kaynaklar yeniden dağıtılabilir.
- Süreç kaynağı **ister, kullanır, bırakır**. İki kaynağı ters sırada almak, hiç kilitlenmeyen bir programı ara sıra kilitlenen bir programa dönüştürür.

- **Dört koşul** birlikte gereklidir: karşılıklı dışlama, tut ve bekle, kesme yok, döngüsel bekleme. İlk üçü politikadır, dördüncüsü o anki durumdur.
- **Kaynak tahsis çizgesinde** döngü yoksa kilitlenme yoktur. Tek örnekli kaynaklarda döngü ⇔ kilitlenme; çok örnekli kaynaklarda döngü gerekli ama yeterli değildir. Tek örnekli durumda **bekleme çizgesi** kullanılır.
- Dört strateji vardır: görmezden gelme (devekuşu), tespit ve kurtarma, kaçınma, önleme. Genel amaçlı işletim sistemleri kullanıcı süreçleri için devekuşu yaklaşımını seçer.
- Tek örnekli kaynaklarda tespit, döngü aramadır; renkli DFS $O(V + E)$'de çalışır. Çok örnekli kaynaklarda E, A, C, R yapılarıyla "isteği boştakilerle karşılanan süreç biter ve tuttuklarını bırakır" döngüsü çalıştırılır; maliyeti $O(m \cdot n^2)$'dir.
- **Kurtarma** kaynak kesme, geri sarma ya da sonlandırmayla yapılır ve her zaman bir iş kaybı getirir. Kurban seçiminde aynı süreç hep seçilirse açlık oluşur.
- **Güvenli durumda** herkesin azami isteğini yapması hâlinde bile bütün süreçleri bitiren bir dizi vardır. Güvensiz durum kilitlenme değildir, ama bitiş garantisi yoktur.
- **Bankacı algoritması** her isteği verilmiş gibi yapar ve yeni durum güvenliyse verir. Süreç iki nedenle bekletilir: kaynak yoktur ya da vermek durumu güvensiz yapar. Azami ihtiyaçların önceden bilinmesini gerektirdiği için pratikte nadiren kullanılır.
- **Önleme** koşullardan birini kırar. En etkili yöntem **kilit sıralamasıdır**: kilitler herkes için ortak bir sırayla alınırsa döngü oluşamaz. tryLock ile geri çekilme kesme yok koşulunu kırar, ama rastgele bekleme olmazsa **canlı kilide** yol açabilir.
- **2PL** serileştirilebilirlik içindir ve kilitlenebilir. **Açlık** politikanın sonucudur ve yaşlandırmayla çözülür. En yaygın eş zamanlılık hataları ise kilitlenme değil, atomiklik ve sıra ihlalleridir.
- Java'da kilitlenme `jstack` ya da `ThreadMXBean` ile bulunur ama çözülmez. `lockdep` ve ThreadSanitizer olası kilitlenmeyi, kilitlenme yaşanmadan bildirir.

Yaklaşım	Ne zaman?	Bilgi gereksinimi	Bedel	Kullanım
Görmezden gelme	—	yok	ara sıra yeniden başlatma	genel amaçlı işletim sistemleri
Tespit ve kurtarma	sonra	şu anki tahsisler ve istekler	periyodik $O(m \cdot n^2)$, kaybolan iş	veritabanları, JVM araçları
Kaçınma	her istekte	azami ihtiyaçlar (Max)	ertelemeler, istek başına $O(m \cdot n^2)$	nadiren, özel sistemler
Önleme	tasarımda	kaynak sırası	esneklik ve kullanım kaybı	çekirdek, uygulama kodu

Tablo 7.18: Dört stratejinin karşılaştırması.

7.11 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

kilitlenme *deadlock* · kesilebilir / kesilemez kaynak *preemptable / nonpreemptable resource* · kilitlenmenin dört koşulu *Coffman conditions* · karşılıklı dışlama *mutual exclusion* · tut ve bekle *hold and wait* · kesme yok *no preemption* · döngüsel bekleme *circular wait* · kaynak tahsis çizgesi *resource allocation graph* · bekleme çizgesi *wait-for graph* · kilitlenme tespiti *deadlock detection* · kilitlenmeden kurtarma *deadlock recovery* · denetim noktası *checkpoint* · kilitlenmeden kaçınma *deadlock avoidance* · güvenli / güvensiz durum *safe / unsafe state* · bankacı algoritması *banker's algorithm* · kilitlenmeyi önleme *deadlock prevention* · biriktirme *spooling* · mutex *mutex* · semafor *semaphore* · monitör *monitor* · meşgul bekleme *busy waiting* · iki aşamalı kilitleme *two-phase locking (2PL)* · canlı kilit *livelock* · açlık *starvation* · yaşlandırma *aging* · atomik komut *atomic instruction*

7.12 Kendini Dene

1. Bir sistemde süreçler kilitleri `trylock` ile alıyor. İkinci kilidi alamayan süreç elindeki kilidi bırakıyor ve hep tam 5 ms bekleyip yeniden deniyor. Hangi kilitlenme koşulu kırılmıştır? Bu tasarımda hangi yeni sorun ortaya çıkabilir ve nasıl giderilir?

   Cevap

   Süreç beklemek yerine elindeki kaynağı kendisi bırakır; kaynak sahibinden geri alınmış olur. Bu, kesme yok koşulunun gönüllü olarak kırılmasıdır. İki süreç aynı ritimde alıp bırakırsa hiçbiri ilerlemez: canlı kilit. Süreçler bloke değildir, işlemci harcarlar. Çare, bekleme süresini rastgele yapmak ve her başarısızlıkta artırmak (üstel geri çekilme) ya da kilitleri ortak bir sırayla almaktır.

2. Bekleme çizgesinde kenarlar P1 → P2, P2 → P3, P2 → P4, P4 → P5 ve P5 → P3'tür. Renkli DFS P1'den başlayıp komşuları numara sırasıyla deniyor. Döngü bulunur mu? Düğümler hangi sırayla siyaha boyanır?

   Cevap

   P1, P2, P3 sırayla griye boyanır. P3'ün komşusu yoktur; ilk siyah P3'tür. P2'nin sıradaki komşusu P4 beyazdır; P4 ve ardından P5 gri olur. P5'in tek komşusu P3 siyahtır; siyah düğüme giden kenar döngü değildir. P5 siyah olur, ardından P4, P2 ve P1. Siyaha boyanma sırası P3, P5, P4, P2, P1'dir ve `dongu` `false` döner: döngü yoktur. P5 → P3 kenarı yalnız iki yolun P3'te birleştiğini gösterir.

3. Üç kaynak türü vardır ve E = (3 2 2)'dir. Tahsisler P1 (1 0 1), P2 (1 1 0), P3 (0 1 1); istekler P1 (1 1 0), P2 (0 0 1), P3 (1 0 0)'dır. Tespit algoritması her işaretlemeden sonra baştan taranarak çalıştırılırsa sistem kilitlenmiş çıkar mı? P3'ün isteği (1 0 1) olsaydı ne olurdu?

   Cevap

   Tahsislerin toplamı (2 2 2)'dir; A = (1 0 0). P1 (1 1 0) ikinci türden ister, boşta yok. P2 (0 0 1) üçüncü türden ister, boşta yok. P3 (1 0 0) ≤ (1 0 0) ✓; P3 biter ve A = (1 1 1) olur. Baştan: P1 (1 1 0) ≤ (1 1 1) ✓, A = (2 1 2). P2 (0 0 1) ✓, A = (3 2 2) = E. Sıra P3, P1, P2'dir; kilitlenme yoktur. P3'ün isteği (1 0 1) olsaydı A = (1 0 0) iken üç satırın hiçbiri karşılanamazdı ve üç süreç de kilitlenmiş olurdu.

4. Tek türden 12 birim kaynak var. A 5 birim tutuyor (en çok 10), B 2 (en çok 4), C 2 (en çok 9). Durum güvenli mi? Güvenli diziler hangileridir? Bu durumda C bir birim daha isterse istek verilmeli mi?

   Cevap

   Boşta 12 − 9 = 3 birim vardır. Kalan ihtiyaçlar A 5, B 2, C 7'dir. Yalnız B karşılanabilir (2 ≤ 3); B biter, boş 5 olur. Sonra yalnız A karşılanabilir (5 ≤ 5); A biter, boş 10 olur. Son olarak C (7 ≤ 10) biter. Tek güvenli dizi ⟨B, A, C⟩'dir; durum güvenlidir. C'ye 1 birim verilirse boş 2, ihtiyaçlar A 5, B 2, C 6 olur. B biter ve boş 4'e çıkar; ama A'nın 5'i de C'nin 6'sı da karşılanamaz. Durum güvensizdir; istek reddedilir ve C bekletilir.

5. Beş süreç aynı türden kaynakları teker teker istiyor; her biri aynı anda en çok 3 örnek kullanıyor ve işi bitince hepsini bırakıyor. Kilitlenmenin hiç oluşmaması için en az kaç örnek gerekir? Bir eksiği kilitlenmeye nasıl yol açar?

   Cevap

   En kötü durumda her süreç 2 örnek tutup üçüncüyü bekler: 5 · 2 = 10 örnek. Bir örnek daha varsa süreçlerden biri üçüncüsünü alır, biter ve 3 örneği bırakır; ötekiler de sırayla biter. En az 5 · 2 + 1 = 11 örnek gerekir. 10 örnekte beş sürecin her biri ikişer örnek tutabilir; boşta örnek kalmaz ve herkes üçüncüyü bekler: kilitlenme.

6. Silberschatz örneğinin başlangıç durumunda (Available = (3 3 2)) T4, (3 3 0) istiyor. Bankacı algo-

ritması ne karar verir? Aynı başlangıç durumunda T0, (0 2 0) isteseydi karar ne olurdu?

Cevap

T4: (3 3 0) ≤ Need_4 = (4 3 1) ve ≤ (3 3 2); iki denetim de geçer. Varmış gibi Available = (0 0 2), Need_4 = (1 0 1) olur. T0 (7 4 3), T1 (1 2 2), T2 (6 0 0), T3 (0 1 1) ve T4 (1 0 1) satırlarının hiçbiri (0 0 2) ile karşılanamaz: durum güvensizdir, istek reddedilir. T0: varmış gibi Available = (3 1 2), Need_0 = (7 2 3) olur. T3 (0 1 1) biter → (5 2 3); T1 (1 2 2) → (7 2 3); T2 (6 0 0) → (10 2 5); T0 (7 2 3) → (10 5 5); T4 (4 3 1) → (10 5 7). Güvenli dizi ⟨T3, T1, T2, T0, T4⟩ vardır; istek verilir. Aynı istek T1'in (1 0 2) isteğinden sonra reddediliyordu: karar isteğin kendisine değil, durumun tamamına bağlıdır.

7. Bir programda `transfer(k, h)` fonksiyonu kilitleri hesap numarası sırasıyla alıyor. Aynı programdaki `rapor(h1, h2)` fonksiyonu ise iki hesabın kilidini her zaman önce `h1`, sonra `h2` sırasıyla alıyor. Program kilitlenebilir mi? Neden?

Cevap

Kilitlenebilir. Kilit sıralaması yalnız kilitlerin alındığı her yerde uygulanırsa döngüyü önler. `rapor(y, x)` çağrısı y'yi tutup x'i isteyebilir. Aynı anda `transfer` numara sırasıyla x'i tutup y'yi isterse döngü oluşur. Çare, `rapor`'u da aynı sıraya uydurmaktır. `lockdep` ya da ThreadSanitizer gibi araçlar bu ters sırayı kilitlenme hiç yaşanmadan bildirir.

8. Bir Java sunucusunda iki iş parçacığı canlı kilide girmiş, başka bir sunucuda iki iş parçacığı kilitlenmiştir. `jstack` hangisini "Found one Java-level deadlock" olarak raporlar? İşlemci kullanımı iki durumda nasıl görünür?

Cevap

`jstack` yalnız kilitlenmeyi raporlar: kilitlenen iş parçacıkları `BLOCKED` durumdadır ve monitör sahipliğinden kurulan bekleme çizgesinde döngü vardır. Canlı kilitteki iş parçacıkları `RUNNABLE` ya da kısa uykularda `TIMED_WAITING` görünür. Kilitleri sürekli alıp bıraktıkları için çizgede kalıcı bir döngü yoktur ve rapor çıkmaz. Kilitlenmede işlemci kullanımı düşer, çünkü iş parçacıkları uyur. Canlı kilitte ise işlemci meşgul görünür ama iş ilerlemez.

7.13 İleri okuma

- A. S. Tanenbaum ve H. Bos, *Modern Operating Systems* (4. baskı), Bölüm 6: kaynak tahsis çizgesini, döngü arama ve matris tabanlı tespit algoritmalarını, kaynak yörüngelerini ve bankacı algoritmasını bu nottaki örneklerle ayrıntılı işler.
- A. Silberschatz, P. B. Galvin ve G. Gagne, *Operating System Concepts* (10. baskı, 2018), Bölüm 8: dört koşulu, önleme protokollerini ve bankacı algoritmasının güvenlik ve istek parçalarını sözde kodla verir; Linux `lockdep` aracını da tanıtır.

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, "Common Concurrency Problems" bölümü: atomiklik ve sıra ihlallerini, kilitlenmenin dört koşulunu ve kilit sıralamasını gerçek sistemlerden alınmış hatalarla anlatır. Kitap çevrim içi ve ücretsizdir.
- S. Lu, S. Park, E. Seo ve Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics", ASPLOS 2008: dört büyük açık kaynak projedeki 105 eş zamanlılık hatasını sınıflandıran ve kilitlenme dışı hataların ağırlığını gösteren çalışmadır.

7.14 Kaynaklar

1. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books, ostep.org — "Common Concurrency Problems".
2. E. G. Coffman, M. J. Elphick, A. Shoshani, "System Deadlocks", *ACM Computing Surveys* 3(2), 1971.
3. E. W. Dijkstra, "Cooperating Sequential Processes", EWD-123, 1965.
4. R. C. Holt, "Some Deadlock Properties of Computer Systems", *ACM Computing Surveys* 4(3), 1972.
5. Linux çekirdek belgeleri: Documentation/locking/lockdep-design.rst.
6. S. Lu, S. Park, E. Seo, Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics", ASPLOS 2008.
7. Java SE 21 API belgeleri: `java.lang.management.ThreadMXBean`, `java.util.concurrent.locks.ReentrantLock`; `jstack` kılavuzu.
8. PostgreSQL belgeleri: "Explicit Locking — Deadlocks", `deadlock_timeout`.
9. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 8: Deadlocks.
10. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı, Pearson, 2014 (5. baskı 2023) — Bölüm 6: Deadlocks.

KISIM III

Bellek ve Depolama

Bölüm 8–11

KISIM III · BÖLÜM 8

Bellek Yönetimi

Bir program ancak ana belleğe yüklendikten sonra çalışabilir. İşlemci komutları ve verileri yalnız yazmaçlardan ve ana bellekten okur; disk ancak giriş/çıkış işlemiyle erişilen yavaş bir depodur. Aynı anda birçok süreç çalıştıran bir sistemde ana bellek bu süreçler arasında paylaştırılmak zorundadır. Paylaşım üç soruyu beraberinde getirir. Bir süreç ötekinin belleğine nasıl erişemez? Bir program belleğin herhangi bir yerine yüklendiğinde adresleri nasıl doğru kalır? Boş alan nasıl izlenir ve parçalara bölünmesi nasıl önlenir? Not, önce bellek hiyerarşisini ve önbelleği ele alır. Ardından taban ve limit yazmaçlarıyla başlayan adres çevirisini anlatır. Bağlama ve yükleme, takas, bitişik tahsis ve boş alan yönetimi bunu izler. Ağırlık bugünkü sistemlerin temeli olan sayfalamadadır: sayfa tablosu, TLB, etkin erişim süresi, çok düzeyli, hash ve ters tablolar. Son kesimler kesimlemeyi ve x86-64, ARMv8 ile RISC-V'nin gerçek çözümlerini karşılaştırır.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum08` · **Laboratuvar:** `github.com/sercankulcu/operating-systems-labs/lab08`

ÖĞRENME ÇIKTILARI

Soyutlama
Bellek hiyerarşisini, önbellek satırlarını ve ortalama erişim süresini; adres alanını, koruma ve yer değiştirme problemlerini açıklamak.

Adresler
Bağlama zamanlarını, mantıksal ve fiziksel adres ayrımını ve MMU'nun taban-limit denetimini açıklamak.

Bağlama
Dinamik yüklemeyi, statik ve dinamik bağlamayı, PLT/GOT ile paylaşımlı kütüphaneleri ve takası tartışmak.

Bitişik tahsis
Uyan algoritmalarını bir istek dizisine uygulamak; iç ve dış parçalanmayı, biteşlemi ve bağlı listeyi çözümlemek.

Sayfalama
Adresi (p, d) olarak bölmek, sayfa tablosu ve TLB ile çeviri yapmak, etkin erişim süresini hesaplamak.

Gerçek sistemler
Hiyerarşik, hash ve ters tabloları; x86-64'ün dört ve beş düzeyli sayfalamasını, büyük sayfaları, kesimlemeyi ve ARMv8'i karşılaştırmak.

8.1 Bellek donanımı

Bellek yönetimini anlamak için önce belleğin kendisine bakmak gerekir. İşletim sistemi belleği soyut bir bayt dizisi olarak görür, ama bu dizinin arkasında hızları ve boyutları çok farklı birkaç katman vardır. Hangi katmanı donanımın, hangisini işletim sisteminin yönettiğini bilmek, sonraki bütün tasarım kararlarını açıklar.

8.1.1 Bellek hiyerarşisi ve bellek yöneticisi

İşlemcinin doğrudan erişebildiği depolama yalnız iki yerdedir: yazmaçlar ve ana bellek. Bir komut, işlenenini yazmaçtan ya da bellekten alır; diskteki bir veriye ise doğrudan erişemez. Diskteki veri önce bir giriş/çıkış işlemiyle belleğe getirilmelidir. Bu yüzden çalışan bir programın kodu ve kullandığı veriler ana bellekte bulunmak zorundadır.

Bellek birimi de çok basit bir arayüz sunar. Ona yalnız bir adres akışı gelir: ya “şu adresi oku” ya da “şu adrese şu veriyi yaz”. Bellek birimi adresin nereden geldiğini bilmez. Adres bir komut sayacından, bir dizi indisinden ya da bir işaretçiden üretilmiş olabilir; bellek için hepsi aynıdır. Bu sadelik, adresleri belleğe varmadan önce değiştiren bir donanım katmanının neden mümkün olduğunu da gösterir.

Hız farkı ise çok büyüktür. Yazmaç bir saat turunda okunur. Ana bellekteki bir DRAM erişimi ise yüzlerce tur sürer ve bu sırada işlemci durur (*stall*). Aradaki boşluğu işlemcinin içindeki küçük ve hızlı bellekler, yani **önbellekler** (*cache*) kapatır. Şekil 8.1 bugünkü tipik değerleri gösterir: yazmaçlar bir nanosaniyenin altında, L1 ve L2 önbellek 1–5 ns, L3 10–20 ns, DRAM 80–120 ns, NVMe SSD 10–100 µs, sabit disk 5–10 ms. Yukarıdan aşağı inildikçe gecikme ve kapasite artar, bayt başına maliyet düşer.



Şekil 8.1: Bellek hiyerarşisi. Yazmaçlardan diske doğru gecikme ve kapasite artar. Değerler 2026 itibarıyla tipik mertebelerdir; çekirdek ve modele göre değişir.

Programcının bellekten beklentisi aslında gerçekleşmesi imkânsız bir listedir: belleğin sınırsız, çok hızlı, kalıcı ve korumalı olması. Tek bir teknoloji bu özelliklerin hepsini birden veremez. Gerçek çözüm hiyerarşidir. Küçük ve hızlı katmanlar sık kullanılan veriyi tutar, büyük ve yavaş katmanlar geri kalanını. İyi kurulmuş bir hiyerarşi programcıya "neredeyse en üst katman kadar hızlı, neredeyse en alt katman kadar büyük" bir bellek izlenimi verir.

Hiyerarşinin katmanlarını farklı yöneticiler denetler. Önbellekleri tamamen donanım yönetir; işletim sistemi onların içeriğine karışmaz. Ana belleği ise işletim sistemi yönetir. İşletim sisteminin bu işle uğraşan kısmına **bellek yöneticisi** denir.

Tanım 8.1

Bellek yönetimi (*memory management*), ana belleğin süreçler arasında paylaştırılmasıdır. Bellek yöneticisi belleğin hangi bölgelerinin boş, hangilerinin dolu olduğunu izler; süreçlere bellek ayırır, işleri bitince geri alır ve süreçleri birbirinden korur. Bellek yetmediğinde süreçlerin ya da sayfaların bir kısmını diske taşımak da onun işidir.

Bellek yöneticisinin işi ile işlemci çizelgeleyicisinin işi sık karıştırılır. Hazır kuyruğundan sıradaki süreci seçmek çizelgeleyicinin görevidir. Bellek yöneticisi ise "hangi süreç nerede, ne kadar yer kaplıyor, kime ne kadar daha yer verilebilir" sorularıyla ilgilenir.

8.1.2 Ana bellek türleri

“Bellek” sözcüğü gündelik dilde birkaç farklı donanım için kullanılır. Tablo 8.1'de dört temel tür karşılaştırılmıştır. Ana bellek olarak kullanılan tür **RAM**'dir (*random access memory*, rastgele erişimli bellek). RAM okunur ve yazılır, hızlıdır, ama uçucudur: güç kesilince içeriği kaybolur. DRAM hücreleri küçük kondansatörlerdir ve yük sızdığı için düzenli aralıklarla yenilenmeleri gerekir. SRAM ise yenileme istemez, daha hızlıdır ama daha pahalıdır ve daha çok yer kaplar. Bu yüzden ana bellek DRAM (bugün DDR5, LPDDR5X), önbellekler SRAM'dir.

Tür	Özellik	Kullanım
RAM (rastgele erişimli)	Okunur ve yazılır, hızlı, uçucu. DRAM yenileme ister; SRAM daha hızlı ve pahalı.	Ana bellek (DRAM: DDR5, LPDDR5X), önbellekler (SRAM)
ROM (salt okunur)	Üretimde yazılır, değiştirilemez, kalıcı, ucuz.	Basit aygıtlar, mikrodenetleyicilerde sabit kod
EEPROM / Flash	Elektrikle silinip yeniden yazılabilir, kalıcı; yazma yavaş, yazma ömrü sınırlı.	UEFI/BIOS ürün yazılımı (SPI flash), SSD, USB bellek, telefon depolaması (NAND flash)
CMOS / NVRAM	Pil ya da flash ile korunan küçük alan.	Saat, önyükleme ayarları

Tablo 8.1: Ana bellek türleri.

ROM (*read-only memory*) üretim sırasında yazılır ve sonradan değiştirilemez. Kalıcıdır ve ucuzdur; basit aygıtlarda ve mikrodenetleyicilerde değişmeyecek kodu tutar. **EEPROM** ve onun bir türü olan **flash bellek** elektrikle silinip yeniden yazılabilir. Kalıcıdırlar, ama yazma işlemi okumadan çok yavaştır ve bir hücre ancak sınırlı sayıda yazılabilir. Bugünkü bilgisayarlarda BIOS/UEFI kodu ROM'da değil, anakarttaki bir SPI flash yongasında durur; ürün yazılımının güncellenebilmesi bundandır. SSD'ler, USB bellekler ve telefonların depolaması da NAND flash kullanır. Son olarak **CMOS** ya da **NVRAM**, bir pil ya da flash ile korunan küçük bir alandır; saat ve önyükleme ayarları burada tutulur.

Bellek yönetiminin konusu bu türlerden yalnız biridir: süreçlerin çalıştığı ana bellek, yani DRAM. Notun geri kalanında “bellek” sözcüğü bu anlamda kullanılır.

8.1.3 Önbellek satırları ve yerellik

Önbellek ile ana bellek arasında veri tek tek baytlar hâlinde değil, sabit boyutlu bloklar hâlinde taşınır. Bu bloğa **önbellek satırı** (*cache line*) denir. Bugünkü işlemcilerin çoğunda satır 64 bayttır. Bellek bu boyutta ardışık satırlara bölünmüş gibi düşünülür: satır 0, 0–63 adreslerini; satır 1, 64–127 adreslerini kapsar ve böyle sürer.

Bir a adresinin hangi satırda olduğu basit bir bölmeyle bulunur. Satır numarası $\lfloor a / 64 \rfloor$, satır içindeki konum a mod 64'tür. $64 = 2^6$ olduğu için bu bölme gerçekte bir bit ayırma işlemidir: adresin alt 6 biti satır içi konumu, geri kalan üst bitleri satır numarasını verir. Şekil 8.2 200 adresini örnek alır. 200 ikili tabanda 11001000'dır. Alt altı bit 001000, yani 8; üst iki bit 11, yani 3'tür. Gerçekten de $3 \times 64 + 8 = 200$.



Şekil 8.2: 64 baytlık satırlarda adres 200'ün bölünmesi: üst bitler satır numarası 3, alt 6 bit satır içi konum 8.

İşlemci bir sözcük okurken donanım önce önbelleğe bakar. Satır önbellekte varsa buna **isabet** (*hit*) denir ve veri L1'den 4–5 turda gelir. Yoksa **ıska** (*miss*) olur; istek bir alt düzeye, en kötü durumda DRAM'e iner ve yüzlerce tur sürer. Önbellek pahalı olduğu için küçüktür. Tek bir büyük önbellek yerine L1, L2, L3 gibi giderek büyüyen ve yavaşlayan düzeyler kurulur.

Küçük bir önbelleğin büyük bir belleğin önünde işe yaramasının nedeni programların davranışıdır. Programlar belleğe rastgele erişmez; erişimlerde iki düzen vardır.

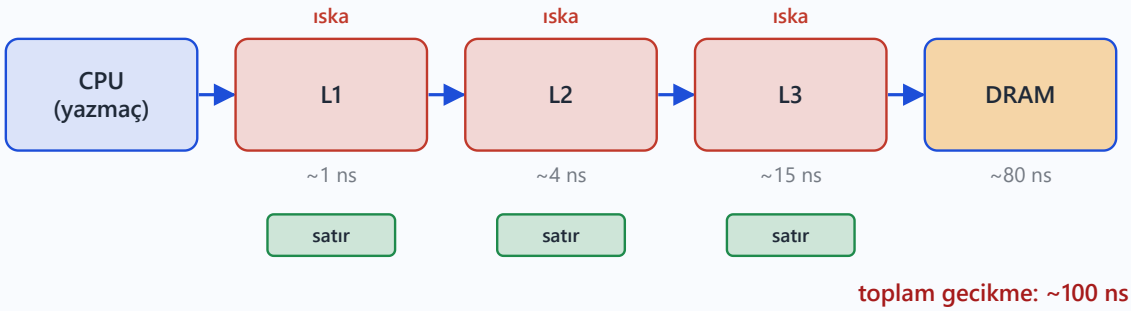
Zamansal yerellik (*temporal locality*)
: Az önce kullanılan bir veri kısa süre içinde yine kullanılır. Döngü sayacı, döngü gövdesinin komutları ve sık çağrılan bir fonksiyon böyledir.

Uzamsal yerellik (*spatial locality*)
: Bir adrese erişen program kısa süre içinde komşu adreslere de erişir. Bir diziyi baştan sona gezmek, ardışık komutları yürütmek böyledir.

Önbellek satırı uzamsal yerelliği kullanır: bir baytlık ıskada 64 baytın tamamı getirildiği için komşu baytlara sonraki erişimler isabet olur. Zamansal yerelliği ise satırın önbellekte kalması kullanır.

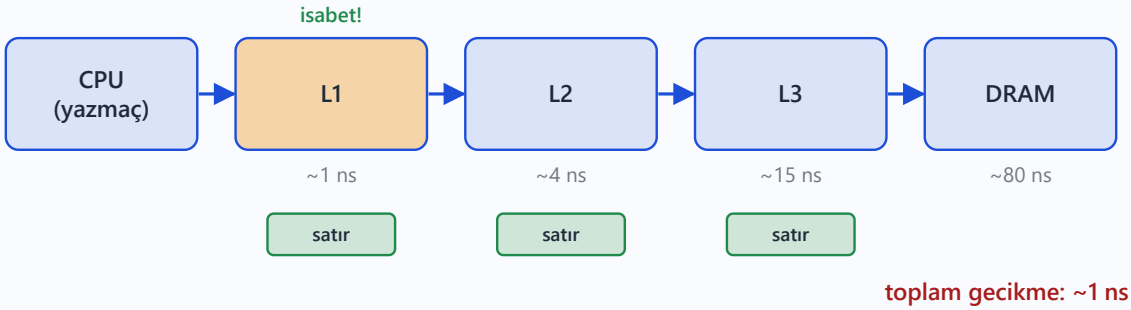
8.1.4 Bir okumanın hiyerarşideki yolu

İşlemci `LOAD R1, [x]` komutuyla x değişkenini okumak istesin. Donanım önce en yakın ve en hızlı düzeye, L1'e bakar. Satır L1'de yoksa yaklaşık 1 ns kaybedilmiştir ve arama L2'de sürer. L2'de de, L3'te de ıska olursa, düzeylerin gecikmeleri toplanır: 1 + 4 + 15 = 20 ns. Sonunda veri DRAM'den gelir; bu da yaklaşık 80 ns ekler ve toplam 100 ns'yi bulur (Şekil 8.3).



Şekil 8.3: Üç düzeyde de ıska olan bir okuma. Veri DRAM'den gelir; tek bir sözcük değil, 64 baytlık satırın tamamı L3, L2 ve L1'e yerleştirilir. Toplam gecikme yaklaşık 100 ns'dir.

DRAM'den gelen tek bir sözcük değil, sözcüğün bulunduğu 64 baytlık satırın tamamıdır. Satır yolda L3'e, L2'ye ve L1'e de yerleştirilir. Aynı satırdaki başka bir veriye ikinci kez erişildiğinde artık L1 isabeti olur ve yaklaşık 1 ns yeter (Şekil 8.4). İlk erişimin 100 ns'lik bedeli, satırın geri kalanına yapılan erişimlerle paylaşılmış olur. Önbelleği işe yarar kılan yerellik budur.



Şekil 8.4: Aynı satıra ikinci erişim L1 isabetidir; gecikme yaklaşık 1 ns'ye iner.

8.1.5 Kod izleme: doğrudan eşlemeli önbellek

Bir satır önbelleğin hangi yerine konabilir? En basit cevap **doğrudan eşlemedir** (*direct mapping*): önbellek birkaç *kümeye* bölünür ve her blok yalnız tek bir kümeye gidebilir. Blok numarası b, küme sayısı K ise blok b mod K kümesine yerleşir. Kümede o anda başka bir blok varsa yeni gelen onu dışarı atar.

Aşağıdaki C parçası dört kümeli, 64 baytlık satırlı bir doğrudan eşlemeli önbelleği taklit eder. `etiket` dizisi her kümede o anda hangi bloğun bulunduğunu tutar; −1 boş demektir.

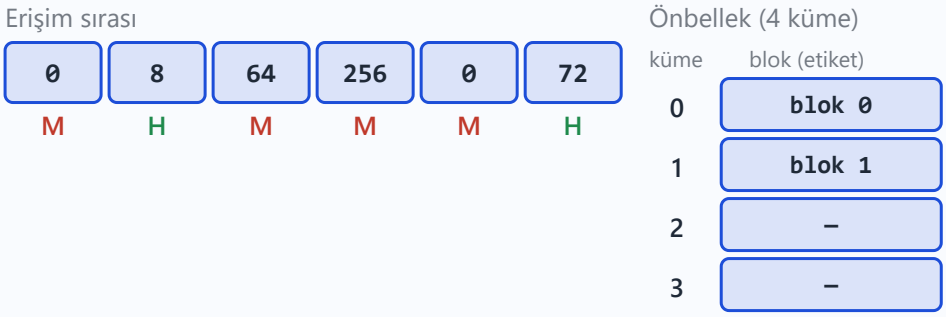
```
int adres[6] = {0, 8, 64, 256, 0, 72};
int etiket[4] = {-1, -1, -1, -1};
int isabet = 0, iska = 0;
for (int i = 0; i < 6; i++) {
  int blok = adres[i] / 64;
  int kume = blok % 4;
  if (etiket[kume] == blok) isabet++;
  else { etiket[kume] = blok; iska++; }
}
```

Döngü her adres için önce blok numarasını (`adres / 64`), sonra kümeyi (`blok % 4`) hesaplar. Kümedeki etiket bu blokla aynıysa isabet sayacı artar. Değilse blok kümeye yazılır ve ıska sayacı artar. Bu basitleştirilmiş modelde etiket olarak blok numarasının tamamı saklanır. Gerçek donanım küme numarasını adresten zaten bildiği için yalnız üst bitleri, yani gerçek etiketi saklar.

i	adres	blok	küme	Sonuç	etiket[0..3] sonra	Açıklama
0	0	0	0	ıska	0, −1, −1, −1	küme 0 boş
1	8	0	0	isabet	0, −1, −1, −1	8, 0 ile aynı satırda: uzamsal yerellik
2	64	1	1	ıska	0, 1, −1, −1	küme 1 boş
3	256	4	0	ıska	4, 1, −1, −1	blok 4, blok 0'ı kovar (çakışma ıskası)
4	0	0	0	ıska	0, 1, −1, −1	blok 0, blok 4'ü kovar (çakışma ıskası)
5	72	1	1	isabet	0, 1, −1, −1	72, 64 ile aynı satırda

Tablo 8.2: Doğrudan eşlemeli önbellek kodunun izlemesi (4 küme, 64 B satır). Değerler Python ile yeniden hesaplanarak doğrulandı.

Döngü bittiğinde `isabet = 2`, `iska = 4` olur; isabet oranı %33'tür. İzlemede iki farklı ıska türü görülür. İlk üç ıska kaçınılmazdır: bloklar önbelleğe ilk kez gelir. Bunlara *zorunlu ıska* denir. 3. ve 4. adımlardaki ıskalar ise başka nedenlidir. 0 ve 256 adresleri farklı bloklardadır (0 ve 4), ama ikisi de 0. kümeye düşer. Önbelleğin 2. ve 3. kümeleri boş durduğu hâlde bu iki blok birbirini kovar. Bu tür ıskaya **çakışma ıskası** (*conflict miss*) denir.



Şekil 8.5: Doğrudan eşlemeli önbellek izlemesinin son durumu. M ıskayı, H isabeti gösterir. 0 ile 256 aynı kümeye düştüğü için birbirini kovdu.

Çakışma ıskalarını azaltmanın yolu, bir bloğa birden çok yer tanımaktır. **k yollu küme ilişkili** (*k-way set associative*) önbellekte her küme k satır tutar. Blok yine tek bir kümeye gider, ama o kümenin k yerinden herhangi birine yerleşebilir.

Örnek 8.1

Aynı dört satırlık önbellek 2 kümeli ve 2 yollu kurulsun; kümede yer gerekince en uzun süredir kullanılmayan satır çıksın. Aynı 0, 8, 64, 256, 0, 72 dizisinde kaç isabet olur?

1. **Kümeler.** Bloklar 0, 0, 1, 4, 0, 1'dir. Küme numarası artık blok mod 2'dir: 0, 0, 1, 0, 0, 1.
2. **İzleme.** 0: küme 0 boş, ıska. 8: blok 0 küme 0'da, isabet. 64: küme 1 boş, ıska. 256: blok 4 küme 0'a gelir; kümede bir yer daha boş olduğu için blok 0 yerinde kalır, ıska. 0: blok 0 hâlâ küme 0'da, isabet. 72: blok 1 küme 1'de, isabet.
3. **Sonuç.** 3 isabet, 3 ıska. Doğrudan eşlemede 2 isabet vardı. Kazanç tamamen çakışma ıskasının ortadan kalkmasından gelir.

Sık yapılan hata

Küme numarasını adresin kendisinden hesaplamak (ör. 256 mod 4 = 0, 72 mod 4 = 0). Kümeyi belirleyen adres değil, blok numarasıdır. Önce adres satır boyuna bölünür, sonra blok numarasının küme sayısına göre kalanı alınır: 72 / 64 = 1, 1 mod 4 = 1.

8.1.6 Kod izleme: ortalama erişim süresi

Bir hiyerarşinin ne kadar iyi çalıştığını tek bir sayıyla özetlemek mümkündür: **ortalama bellek erişim süresi**. Her erişim önce en üst düzeye uğrar ve o düzeyin süresini harcar. Iska olursa istek bir alt düzeye iner ve onun ortalama süresi eklenir. Bir düzeyin süresi s, ıska oranı m, altındaki düzeyin ortalama süresi T_{alt} ise:

$$T = s + m \cdot T_{alt}$$

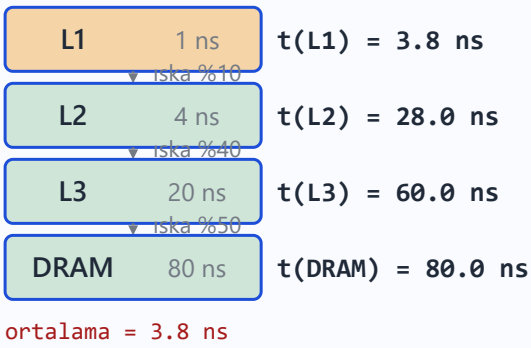
Formül kendini tekrar eder: alttaki düzeyin süresi de aynı biçimde hesaplanır. En alttaki DRAM her isteği karşıladığı için orada T yalnız DRAM'in süresidir. Aşağıdaki kod hesabı alttan yukarı doğru yapar.

```
/* L1, L2, L3, DRAM erişim süresi (ns) */
double sure[4] = {1, 4, 20, 80};
double iska[3] = {0.10, 0.40, 0.50};
double t = sure[3];         /* DRAM */
for (int k = 2; k >= 0; k--)
  t = sure[k] + iska[k] * t;
printf("ortalama = %.1f ns\n", t);
```

`t` önce DRAM'in 80 ns'siyle başlar. Döngü `k = 2`'den (L3) `k = 0`'a (L1) iner ve her turda bir üst düzeyin süresini hesaplar. Tablo 8.3 her turu gösterir.

Adım	k	Düzey	Hesap	t (ns)
başlangıç	—	DRAM	`t = sure[3]`	80
1	2	L3	20 + 0,50 × 80	60
2	1	L2	4 + 0,40 × 60	28
3	0	L1	1 + 0,10 × 28	3,8
son	−1	—	döngü biter, `ortalama = 3.8 ns` yazılır	3,8

Tablo 8.3: Ortalama erişim süresi kodunun izlemesi. Değerler Python ile yeniden hesaplanarak doğrulandı.



Şekil 8.6: Dört düzeyin erişim süreleri, ıska oranları ve alttan yukarı hesaplanan ortalama süreler.

Sonuç dikkat çekicidir. Her erişimin %10'u L1'i ıskaladığı hâlde ortalama 3,8 ns'dir. Bu, DRAM'in 80 ns'sinden yaklaşık 21 kat hızlıdır (80 / 3,8 ≈ 21,05). Hiyerarşiyi işe yarar kılan, üst düzeylerdeki yüksek isabet oranıdır.

Formülün biçimi ıska oranlarının ne kadar etkili olduğunu da gösterir. L1'in ıska oranı %10'dan %5'e inerse, alttaki 28 ns aynı kalır ve ortalama 1 + 0,05 × 28 = 2,4 ns olur. L1'in kendi süresini 1 ns'den 0,5 ns'ye indirmek ise yalnız 0,5 ns kazandırır. Ortalamaya ıskalar hâkimdir.

Örnek 8.2

L1 2 ns'de yanıt veriyor ve isteklerin %5'inde ıskalıyor. L2 10 ns sürüyor ve kendisine gelen isteklerin %20'sini ıskalıyor. DRAM 100 ns'dir. Ortalama erişim süresi nedir? L2 olmasaydı ne olurdu?

1. **L2 düzeyi.** $T_{L2} = 10 + 0{,}20 \times 100 = 30$ ns.
2. **L1 düzeyi.** $T = 2 + 0{,}05 \times 30 = 2 + 1{,}5 = 3{,}5$ ns.
3. **L2 olmadan.** L1 ıskaları doğrudan DRAM'e gider: $2 + 0{,}05 \times 100 = 7$ ns. İkinci düzey ortalamayı yarıya indirmiştir.

Sık yapılan hata

İkinci düzeyin ıska oranını bütün erişimlere oranlamak. L2'nin %20'lik ıska oranı L2'ye *gelen* istekler içindir; bütün erişimlerin yalnız %5'i L2'ye gelir. Bu yüzden DRAM'e inen erişim oranı %20 değil, 0,05 × 0,20 = %1'dir. Formül bu çarpımı kendiliğinden yapar.

8.1.7 Önbellek tasarımının dört sorusu

Her önbellek tasarımı dört soruya cevap vermek zorundadır.

1. **Veri ne zaman önbelleğe konur?** Genellikle ıska anında. Donanım ayrıca, erişim düzenini tahmin ederek sıradaki satırları önceden de getirir (*prefetch*).
2. **Hangi satıra konur?** Bu eşleme sorusudur: doğrudan eşlemeli, k yollu küme ilişkili ya da tam ilişkili.
3. **Yer gerekince hangisi çıkar?** Bu değiştirme politikasıdır: LRU'ya yakın yaklaşımlar, rastgele seçim ya da FIFO.
4. **Çıkan satır nereye gider?** Satır değiştirilmişse (“kirli” ise) çıkarken alt düzeye yazılır (*write-back*). Diğer seçenek her yazmayı hemen alt düzeye iletmektir (*write-through*).

Bu dört sorunun yanında üç ödünleşim vardır. Büyük önbellek daha çok isabet verir, ama daha yavaş ve daha pahalıdır. Daha akıllı bir değiştirme politikası isabeti artırır, ama donanımı karmaşıklaştırır. Çok çekirdekli işlemcilerde aynı satırın farklı çekirdeklerdeki kopyalarının güncel tutulması gerekir; buna tutarlılık denir ve MESI gibi protokollerle sağlanır.

Aynı dört soru yalnız donanım önbelleklerine özgü değildir. İşletim sisteminin yönettiği “önbelleklerde” de karşımıza çıkar: dosyaların tutulduğu sayfa önbelleğinde, sanal bellekte ve TLB'de. Bu notta anlatılacak TLB doğrudan bir önbellektir; sanal bellek ise ana belleği diskin önbelleği gibi kullanır.

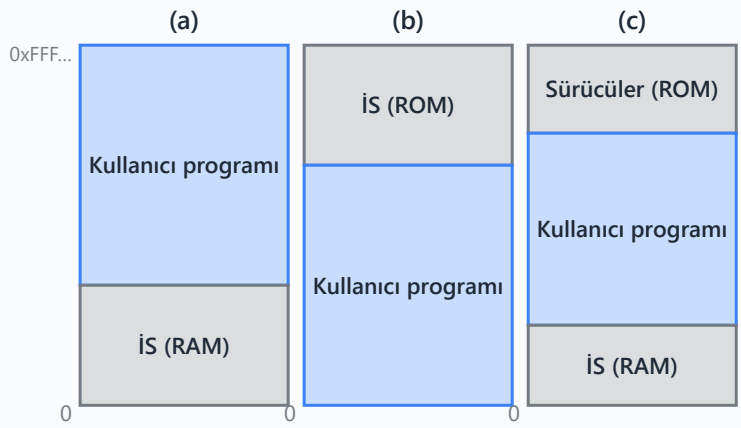
8.2 Adres alanı

Birden çok programın aynı bellekte güvenle çalışabilmesi için iki problemin çözülmesi gerekir. Birincisi korumadır: bir program başka bir programın belleğine erişememelidir. İkincisi yer değiştirmedir: bir program belleğin neresine yüklenirse yüklensin doğru çalışmalıdır. Bu kesim, iki problemin nereden çıktığını ve en basit donanım çözümünü anlatır.

8.2.1 Soyutlama olmadan bellek

İlk bilgisayarlarda bellek için hiçbir soyutlama yoktu. 1960'lardan önceki ana bilgisayarlarda ve 1980'lerden önceki kişisel bilgisayarlarda her program fiziksel belleği olduğu gibi görürdü. `MOV REGISTER1, 1000` komutu, fiziksel 1000 adresindeki sözcüğü okurdu. Programın gördüğü adres ile bellek yongasının gördüğü adres aynıydı.

Bu düzende bellekte aynı anda iki program bulunamaz. İkisi de aynı fiziksel adresleri kullanır ve biri ötekinin verisini ezer. Bellek yine de işletim sistemi ile tek bir kullanıcı programı arasında paylaşılabilir. Şekil 8.7 üç yaygın düzeni gösterir. (a)'da işletim sistemi RAM'in alt ucundadır. (b)'de işletim sistemi üst uçta, ROM'dadır. (c)'de aygıt sürücüleri üstte ROM'da, işletim sisteminin geri kalanı altta RAM'dedir. (a) ve (c) düzenlerinde kullanıcı programındaki bir hata işletim sisteminin bulunduğu bölgeye yazıp onu çökertebilir. (b) bazı gömülü sistemlerde, (c) ise MS-DOS çalıştıran ilk PC'lerde kullanıldı; orada BIOS sürücüleri ROM'daydı.



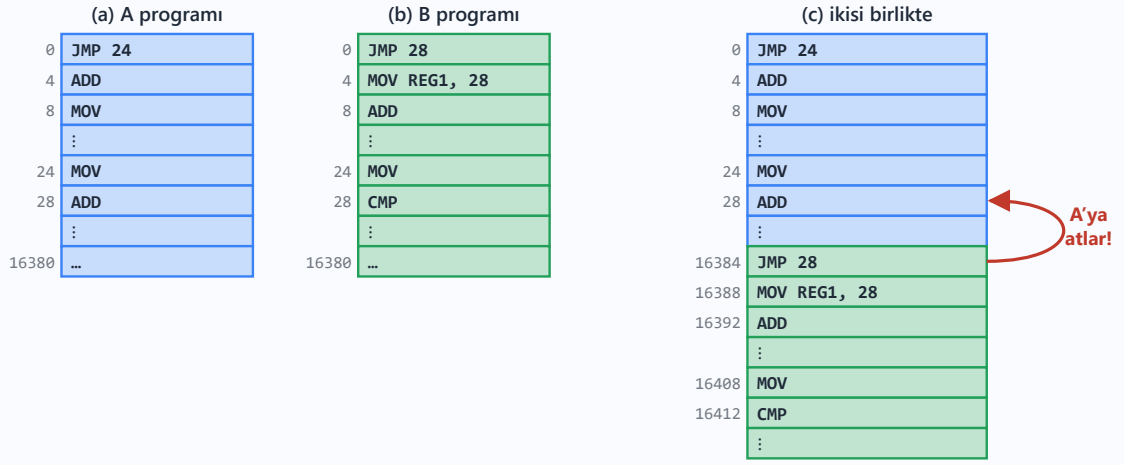
Şekil 8.7: Soyutlama olmadan belleği işletim sistemi ve tek bir kullanıcı programı arasında paylaştırmanın üç yolu.

Bu yaklaşım tarihe karışmış değildir. Bellek yönetim birimi olmayan mikrodenetleyiciler bugün de böyle çalışır. Bunların çoğunda yine de bir **bellek koruma birimi** (*memory protection unit*, MPU) bulunur. MPU adres çevirisi yapmaz, yalnız belirli bölgelere erişim izinlerini denetler.

8.2.2 Yer değiştirme problemi

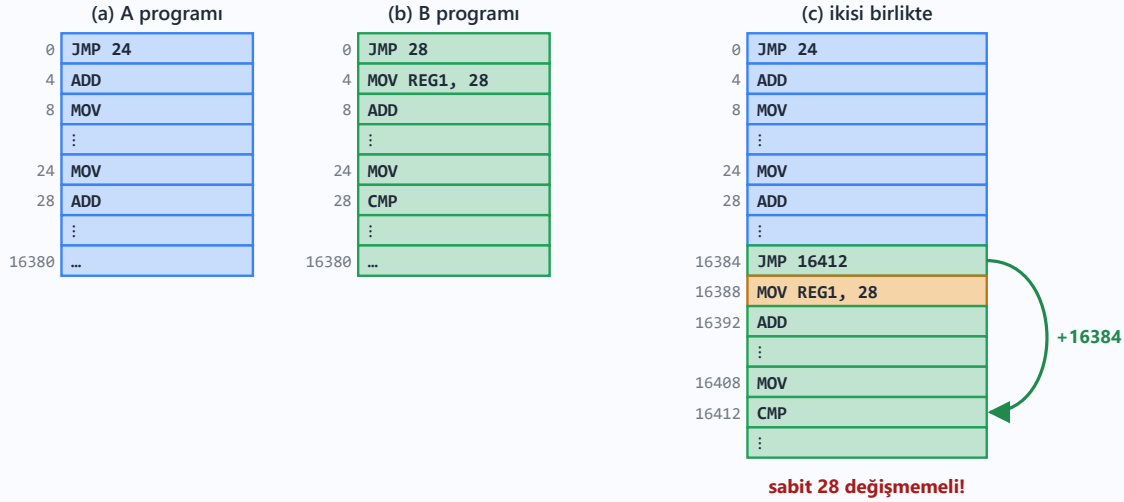
İki programı aynı anda belleğe koymaya çalışınca ortaya çıkan sorunu Tanenbaum'un örneği açıkça gösterir. A ve B, 16 KB'lık iki programdır ve ikisi de 0 adresinden başlayacağı varsayılarak derlenmiştir. A'nın ilk komutu `JMP 24`, B'nin ilk komutu `JMP 28`'dir. İşletim sistemi A'yı 0 adresine, B'yi hemen üstüne, 16384 adresine yükler.

A sorunsuz çalışır. B ise ilk komutunda çöker. `JMP 28`, B'nin 28. baytındaki `CMP` komutuna atlamak ister. Ama 28 mutlak bir adrestir ve fiziksel 28 adresinde A'nın `ADD` komutu vardır. B, kendi kodu yerine A'nın ortasına atlar (Şekil 8.8).



Şekil 8.8: Yer değiştirme problemi. (a) ve (b) 0'dan başlayacak biçimde derlenmiş iki program, (c) ikisi birlikte bellekte. B'nin `JMP 28` komutu A'nın içine atlar.

Bu probleme **yer değiştirme** (*relocation*) problemi denir. Programdaki adresler, programın belleğe yüklendiği yere göre düzeltilmelidir. IBM 360'ın bulduğu ilk çözüm *statik yer değiştirmedir*: yükleyici programı belleğe koyarken her adres sözcüğüne programın başlangıç adresini ekler. B 16384'e yüklendiği için `JMP 28`, `JMP 16412` olur (Şekil 8.9).



Şekil 8.9: Statik yer değiştirme. Yükleyici atlama adresine 16384 ekler; ama `MOV REG1, 28` komutundaki 28 bir sabittir ve değişmemelidir.

Statik yer değiştirmenin iki zayıf yanı vardır. Birincisi yükleme yavaşlar, çünkü programın her sözcüğüne bakılması gerekir. İkincisi daha ince bir sorundur: yükleyici hangi sözcüğün adres, hangisinin sıradan bir sayı olduğunu bilemez. B'nin ikinci komutu `MOV REG1, 28`, yazmaca 28 sayısını yükler. Bu

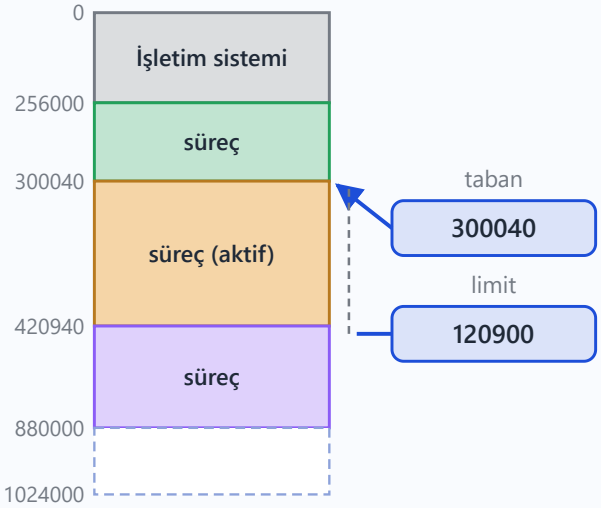
28 bir adres değildir; 16384 eklenirse program yanlış çalışır. Yükleyicinin doğru karar verebilmesi için derleyicinin, programın hangi sözcüklerinin adres olduğunu gösteren ek bir yer değiştirme bilgisi üretmesi gerekir.

8.2.3 Taban ve limit yazmaçları

Koruma ve yer değiştirme problemlerini birlikte çözen soyutlama **adres alanıdır**.

> **Tanım 8.2**
>
> **Adres alanı** (*address space*), bir sürecin bellekte kullanabileceği adreslerin kümesidir. Her sürecin kendine ait, diğer süreçlerinkinden bağımsız bir adres alanı vardır.

Adres alanını gerçeklemenin en basit donanımı iki yazmaçtır. **Taban yazmacı** (*base register*) sürecin bellekteki başlangıç adresini, **limit yazmacı** (*limit register*) bölgenin uzunluğunu tutar. Program bellekte bitişik bir alana olduğu gibi yüklenir. Şekil 8.10 örneğinde taban 300040, limit 120900'dür. Süreç 300040 ile 300040 + 120900 − 1 = 420939 arasındaki adreslere erişebilir.



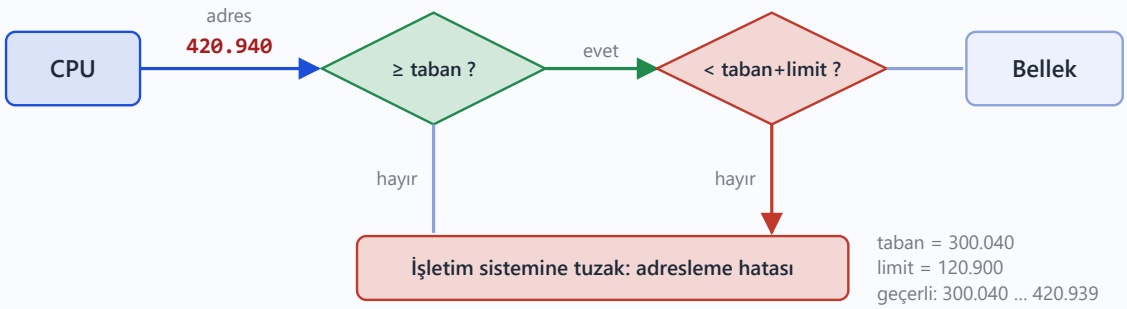
Şekil 8.10: Taban ve limit yazmaçları. Etkin süreç 300040 ile 420939 arasındaki adreslere erişebilir.

Bu iki yazmaç ayrıcalıklı komutlarla, yalnız çekirdek modunda yüklenebilir. Kullanıcı programı kendi tabanını ya da limitini değiştiremez; değiştirebilseydi koruma anlamsız olurdu. İşletim sistemi bağlam değiştirirken yeni sürecin taban ve limit değerlerini bu yazmaçlara koyar.

Tarihte bu düzenin iki uç örneği vardır. CDC 6600 süper bilgisayarı tam olarak böyle çalışıyordu. Intel 8088'in kesim yazmaçları ise yalnız taban işlevi görüyordu; limit yazmacı olmadığı için yer değiştirme vardı ama koruma yoktu.

8.2.4 Donanımsal adres koruması

Taban ve limit yazmaçları korumayı her bellek erişiminde iki karşılaştırmayla sağlar. İşlemcinin ürettiği adres önce tabanla karşılaştırılır: adres tabandan küçükse tuzak oluşur. Sonra taban + limit ile karşılaştırılır: adres bu değerden küçük değilse yine tuzak oluşur. İki koşul da sağlanırsa erişim belleğe gider (Şekil 8.11). Karşılaştırmaları paralel çalışan özel donanım yapar ve erişimi yavaşlatmaz.



Şekil 8.11: Donanımsal adres koruması. 420940 = taban + limit olduğu için ikinci karşılaştırma başarısız olur ve işletim sistemine tuzak üretilir.

Adres	≥ taban?	< taban + limit?	Sonuç
300040	evet	evet	erişim (ilk geçerli adres)
356000	evet	evet	erişim
420939	evet	evet	erişim (son geçerli adres)
420940	evet	hayır	tuzak
299999	hayır	—	tuzak; başka sürecin alanı korunur

Tablo 8.4: Taban = 300040, limit = 120900 iken beş adresin denetimi (geçerli aralık 300040 … 420939).

Tuzak oluşunca denetim çekirdeğe geçer. Çekirdek bunu adresleme hatası sayar ve süreci genellikle sonlandırır; Unix sistemlerinde süreç `SIGSEGV` sinyali alır. Denetim yalnız kullanıcı modundaki erişimler için yapılır. Çekirdek modunda denetim yoktur, çünkü işletim sisteminin bütün belleğe erişebilmesi gerekir: bir süreci belleğe yüklemek, bir sistem çağrısının argümanını kullanıcı tamponundan okumak bunu gerektirir.

Sık yapılan hata

Taban + limit adresini geçerli saymak. Limit bölgenin uzunluğudur; geçerli adresler taban, taban + 1, …, taban + limit − 1'dir. 120900 baytlık bir bölgenin son baytı 300040 + 120899 = 420939'dadır. 420940 artık bölgenin dışındadır.

8.2.5 Adres bağlama zamanları

Bir program kaynak koddan bellekteki sürece giden yolda birkaç aşamadan geçer (Şekil 8.12). Derleyici `main.c`'yi nesne dosyasına (`main.o`) çevirir. Bağlayıcı bu dosyayı diğer nesne dosyaları ve statik kütüphanelerle birleştirip çalıştırılabilir dosyayı (`a.out`) üretir. Yükleyici bu dosyayı belleğe alır, dinamik kütüphaneleri ekler ve süreç çalışmaya başlar.



Şekil 8.12: Programın derleyici, bağlayıcı ve yükleyiciden geçerek belleğe yerleşmesi. Adres bağlama bu üç aşamadan birinde yapılabilir.

Adresler bu yolculuk boyunca biçim değiştirir. Kaynak kodda adresler semboliktir: programcı `count` adlı bir değişken yazar. Derleyici bu sembolü göreli bir adrese bağlar: "bu modülün başından 14 bayt sonra". Bağlayıcı ya da yükleyici de göreli adresi mutlak bir adrese çevirir, örneğin 74014. Bir sembolün mutlak adrese ne zaman bağlandığı, programın bellekte taşınıp taşınamayacağını belirler. Tablo 8.5 üç seçeneği karşılaştırır.

Bağlama zamanı	Üretilen kod	Konum değişirse	Donanım
Derleme	Mutlak kod (ör. MS-DOS .COM)	yeniden derlenir	gerekmez
Yükleme	Yeri değiştirilebilir kod; yükleyici adresleri düzeltir	yeniden yüklenir	gerekmez
Yürütme	Mantıksal adresler; süreç çalışırken taşınabilir	hiçbir şey	MMU (taban-limit, sayfalama)

Tablo 8.5: Adres bağlama zamanları.

Derleme zamanında bağlamada derleyici programın belleğe nereye yükleneceğini önceden bilir ve mutlak kod üretir. MS-DOS'un `.COM` dosyaları böyleydi. Başlangıç adresi değişirse program yeniden derlenmelidir. **Yükleme zamanında** bağlamada derleyici yeri değiştirilebilir kod üretir; mutlak adresler yükleme anında, statik yer değiştirmeyle belirlenir. Başlangıç adresi değişirse program yeniden yüklenir. **Yürütme zamanında** bağlamada ise adres çevirisi her erişimde, program çalışırken yapılır. Süreç çalışırken bir bellek bölgesinden ötekine taşınabilir. Bunun için özel donanım gerekir. Bugünkü genel amaçlı işletim sistemlerinin hepsi yürütme zamanında bağlama kullanır.

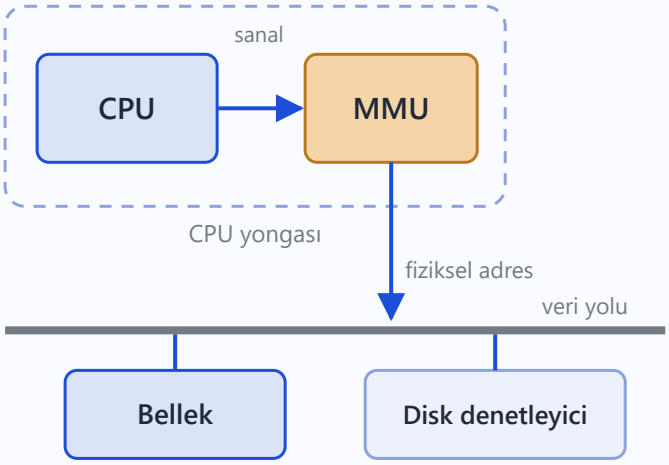
8.2.6 Mantıksal ve fiziksel adres, MMU

Yürütme zamanında bağlama, iki tür adresi birbirinden ayırır.

Tanım 8.3

Mantıksal adres (*logical address*), işlemcinin ürettiği adrestir; **sanal adres** de denir. **Fiziksel adres** (*physical address*), bellek biriminin gördüğü, bellek adres yazmacına (MAR) yüklenen adrestir. Bir programın ürettiği bütün mantıksal adreslerin kümesi **mantıksal adres alanı**, bunlara karşılık gelen fiziksel adreslerin kümesi **fiziksel adres alanıdır**.

Derleme ve yükleme zamanında bağlamada iki adres aynıdır. Yürütme zamanında bağlamada ise farklıdır ve aralarındaki dönüşümü çalışma anında bir donanım yapar: **bellek yönetim birimi** (*memory management unit*, MMU). Program yalnız mantıksal adresleri görür ve onlarla çalışır; fiziksel adresleri hiç görmez. Bir işaretçinin değerini ekrana yazdıran bir C programı, bu yüzden her zaman bir mantıksal adres yazdırır.

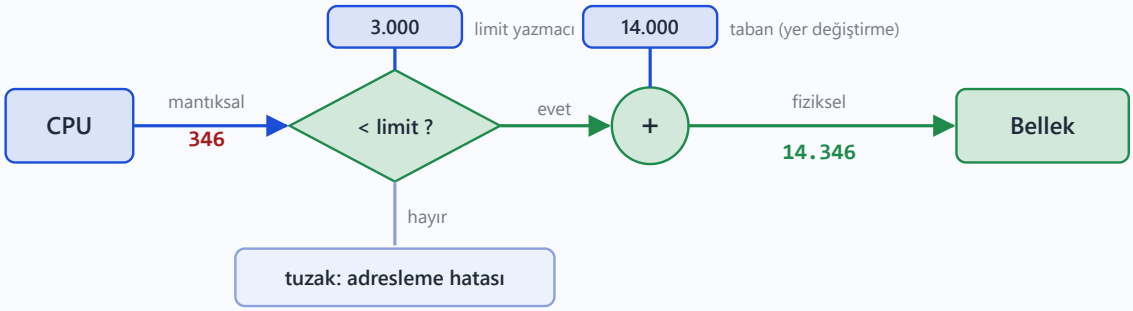


Şekil 8.13: MMU işlemci yongasının içindedir. İşlemci sanal adresi MMU'ya verir; MMU fiziksel adresi veri yoluna koyar.

Şekil 8.13 MMU'nun yerini gösterir. Eski sistemlerde MMU ayrı bir yongaydı. Bugünkü işlemcilerde MMU, TLB'leri ve sayfa tablosu yürüyücüsüyle birlikte her çekirdeğin içindedir. Bellek veri yoluna çıkan her adres zaten fiziksel adrestir.

8.2.7 MMU ile dinamik yer değiştirme

Taban-limit düzeninin yürütme zamanındaki biçiminde MMU iki iş yapar. Önce mantıksal adresi limitle karşılaştırır; adres limitten küçük değilse tuzak üretir. Sonra adrese taban değerini ekler ve fiziksel adresi bulur. Silberschatz bu yazmaca **yer değiştirme yazmacı** (*relocation register*) der; Tanenbaum'un taban yazmacıyla aynı şeydir.



Şekil 8.14: MMU ile dinamik yer değiştirme. Mantıksal 346, limit 3000'den küçüktür; tabanla toplanır ve fiziksel 14346 bulunur.

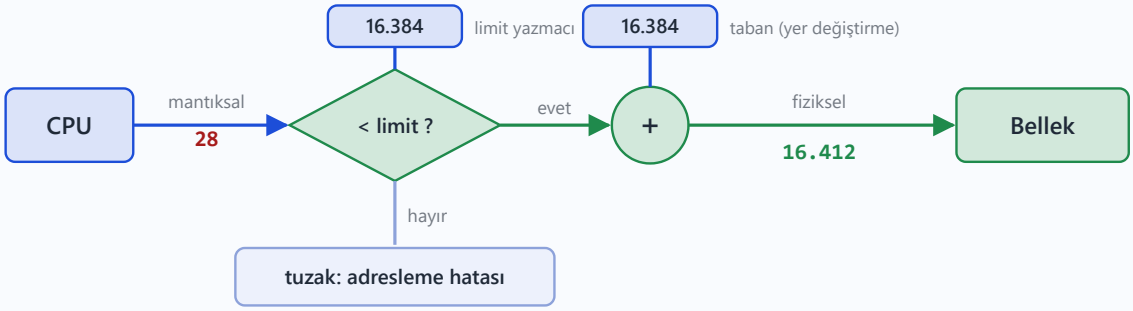
Örnek 8.3

Limit 3000, taban 14000 olsun. Mantıksal 346, 2999 ve 3500 adresleri nereye gider?

1. **346.** 346 < 3000, geçerli. Fiziksel adres 346 + 14000 = 14346.
2. **2999.** 2999 < 3000, son geçerli mantıksal adres. Fiziksel 2999 + 14000 = 16999.
3. **3500.** 3500 ≥ 3000: toplama yapılmadan tuzak oluşur. Karşılaştırma mantıksal adres üzerinden yapıldığı için sonuç, taban değeri ne olursa olsun aynıdır.

Program kendini 0 ile 2999 arasında görür. İşletim sistemi süreci başka bir yere taşırsa yalnız taban yazmacını değiştirir; programın tek bir adresi değişmez.

Bu düzen, yer değiştirme probleminin örneğindeki B programını da kurtarır. B için taban = 16384, limit = 16384 olsun. `JMP 28` komutu bellekte olduğu gibi kalır. Komut çalışınca MMU 28'i limitle karşılaştırır, geçerli bulur ve 28 + 16384 = 16412 fiziksel adresini üretir (Şekil 8.15). Statik yer değiştirmeye de, ek yer değiştirme bilgisine de gerek kalmaz; `MOV REG1, 28` içindeki sabit hiç değişmez, çünkü yalnız belleğe giden adresler çevrilir.



Şekil 8.15: B programı taban = limit = 16384 ile çalışır. `JMP 28` değişmeden kalır; MMU 16412 üretir.

Bunun bir bedeli vardır: her bellek erişiminde bir toplama ve bir karşılaştırma gerekir. Karşılaştırma hızlıdır. Toplama ise, elde yayılımı yüzünden, özel devreler kullanılmadıkça bir erişime göre yavaş kalır. Bu bedel, ileride sayfalamayla birlikte daha da büyüyecek bir sorunun ilk işaretidir: adres çevirisi her erişimin yolu üzerindedir ve çok hızlı olmak zorundadır.

8.3 Bağlama ve yükleme

Bir programın belleğe yüklenmesi "dosyanın tamamını belleğe kopyala" kadar basit değildir. Program kütüphanelere başvurur; bu kütüphaneler programla birlikte mi, ayrı mı yüklenecek? Kullanılmayabilecek büyük bir kod parçası baştan mı yüklenmeli? Bu kesim bu soruların bugünkü cevaplarını anlatır.

8.3.1 Dinamik yükleme

Dinamik yükleme (*dynamic loading*), bir rutinin çağrılana kadar belleğe alınmamasıdır. Programın ana kısmı yüklenir ve çalışmaya başlar. Bir rutine ihtiyaç duyulduğunda program onu kendisi yükletir. Hiç çağrılmayan kod hiç yüklenmez. Bu, hata işleme kodu gibi nadiren çalışan ama büyük kısımlar için yararlıdır.

Dinamik yükleme için özel donanım gerekmez; program buna göre tasarlanır. İşletim sistemi yalnız gerekli kütüphane işlevlerini sağlar. POSIX sistemlerinde bunlar `dlopen`, `dlsym` ve `dlclose`'dur; Windows'taki karşılıkları `LoadLibrary` ve `GetProcAddress`'tir. Aşağıdaki program matematik kütüphanesini çalışırken yükler ve içindeki `sqrt` işlevini çağırır.

```
#include <dlfcn.h>
#include <stdio.h>

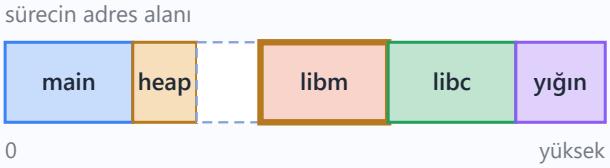
int main(void) {
  void *h = dlopen("libm.so.6", RTLD_LAZY);
  if (!h) { puts(dlerror()); return 1; }
  double (*kok)(double) =
      (double (*)(double)) dlsym(h, "sqrt");
  printf("%.3f\n", kok(2.0));
  dlclose(h);
  return 0;
}
```

Program satır satır şöyle çalışır.

1. Program başladığında adres alanında yalnız `main`'in kodu, C kütüphanesi (libc), heap ve yığın vardır. Matematik kütüphanesi (libm) yüklü değildir.
2. `dlopen("libm.so.6", RTLD_LAZY)` çağrısı dinamik bağlayıcıyı (`ld.so`) çalıştırır. Dinamik bağlayıcı dosyayı bulur ve `mmap` ile sürecin adres alanına eşler (Şekil 8.16). Yükleme programın isteğiyle, şimdi olur. Dönen `h` kütüphaneyi temsil eden bir tanıtıcıdır; dosya bulunamazsa `NULL` döner

ve `dlerror()` nedenini bildirir.

3. `dlsym(h, "sqrt")` kütüphanenin sembol tablosunda `sqrt` adını arar ve bu sürecin adres alanındaki adresini döndürür. Dönen `void *` değeri, `double` alıp `double` döndüren bir fonksiyon işaretçisine dönüştürülür.
4. `kok(2.0)` çağrısı fonksiyon işaretçisi üzerinden yapılır ve ekrana `1.414` yazılır (%.3f biçimiyle √2).
5. `dlclose(h)` kütüphanenin referans sayısını bir azaltır. Sayı 0'a inince kütüphane adres alanından çıkarılır.



Şekil 8.16: `dlopen` sonrası adres alanı: libm, main ile libc'nin arasına eşlenmiştir.

Eski glibc sürümlerinde bu programı derlerken `-ldl` seçeneği gerekirdi. glibc 2.34'ten beri bu işlevler libc'nin içindedir ve ek seçenek gerekmez.

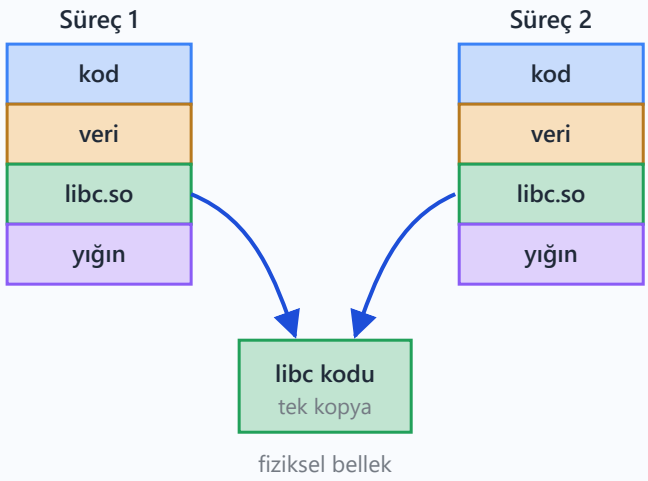
8.3.2 Statik ve dinamik bağlama

Program bir kütüphane işlevini çağırdığında, o işlevin kodunun bir yerde bulunması gerekir. Bunun iki yolu vardır. **Statik bağlamada** bağlayıcı (`ld`), derlemeden sonra kütüphane rutinlerini çalıştırılabilir dosyanın içine kopyalar; örneğin `ld *.o -lc -lm`. **Dinamik bağlamada** ise dosyada yalnız küçük bir *taslak* (*stub*) kalır. Rutinin gerçek adresi yürütme zamanında çözülür. Tablo 8.6 iki yaklaşımı karşılaştırır.

	Statik bağlama	**Dinamik bağlama**
Kim, ne zaman?	Bağlayıcı (ld), derlemeden sonra	Dinamik bağlayıcı (ld.so), yüklemede ya da çalışırken
Dosya	.a, .lib	.so, .dll, .dylib
Bellek	Her programda ayrı kopya	Kod sayfaları bir kez bellekte, paylaşılır
Güncelleme	Yeniden bağlamak gerekir	Kütüphaneyi değiştirmek yeter (sürüm: soname)
Risk	Büyük dosya; güvenlik yaması gecikir	Sürüm uyuşmazlığı (“DLL cehennemi”)

Tablo 8.6: Statik ve dinamik bağlama.

Dinamik bağlamanın en büyük kazancı bellektedir. Yüzlerce süreç aynı C kütüphanesini kullanır. Statik bağlamada her sürecin içinde ayrı bir kopya bulunurdu. Dinamik bağlamada kütüphanenin kod sayfaları fiziksel bellekte bir kez bulunur ve her sürecin adres alanına eşlenir (Şekil 8.17). Bu tür kütüphaneye **paylaşımlı kütüphane** (*shared library*) denir.



Şekil 8.17: İki süreç aynı libc kod sayfalarını kendi adres alanlarına eşler; fiziksel bellekte tek kopya bulunur.

Güncelleme de kolaylaşır. C kütüphanesinde bir güvenlik açığı kapatıldığında yeni kütüphaneyi kurmak yeter; onu kullanan bütün programlar bir sonraki çalıştırmada yamalı sürümü kullanır. Statik bağlanmış programlar ise yeniden bağlanana kadar açığı taşır. Uyumsuz sürümler karışmasın diye kütüphanelerin bir sürüm adı (soname) vardır; örneğin `libm.so.6`'daki 6 böyle bir sürümdür. Bu önlem alınmadığında yaşanan karmaşa Windows dünyasında "DLL cehennemi" adıyla anılır.

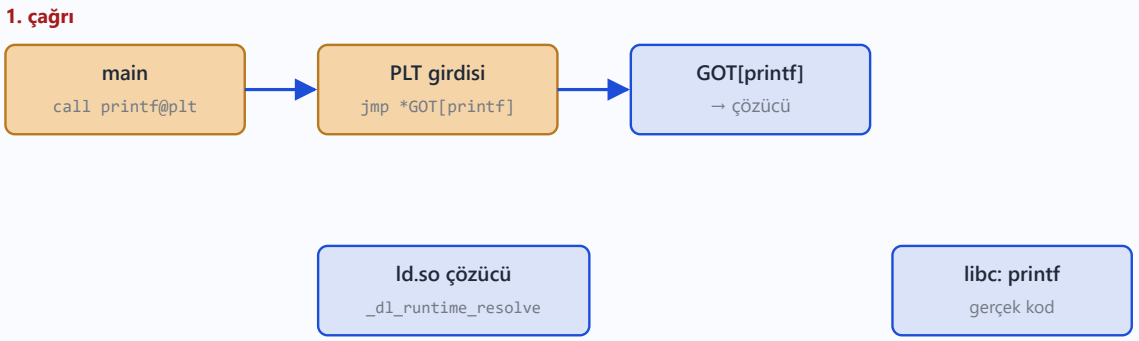
8.3.3 Tembel bağlama: PLT ve GOT

Ders kitaplarında dinamik bağlama genellikle şöyle anlatılır: taslak ilk çağrıldığında rutinin adresini bulur ve kendini bu adresle değiştirir. Bu anlatım kavramsal olarak doğrudur, ama Linux gibi ELF biçimli dosya kullanan sistemlerde değişen şey kod değildir. Gerçek düzenek iki tabloya dayanır.

PLT (*procedure linkage table*)
: Çalıştırılabilir dosyadaki küçük atlama taslaklarının tablosudur. Program `printf`'i doğrudan değil, `printf@plt` taslağını çağırır. Kitaplardaki "taslak" budur.

GOT (*global offset table*)
: Sürece özel, yazılabilir bir adres tablosudur. PLT taslağı, GOT'taki ilgili girdinin gösterdiği adrese dolaylı olarak atlar.

İlk çağrıda GOT girdisi gerçek `printf`'i değil, dinamik bağlayıcının çözücüsünü gösterir. Çözücü sembolü yüklü kütüphanelerde arar, bulduğu adresi GOT girdisine yazar ve `printf`'e atlar. Bundan sonraki çağrılar PLT → GOT → `printf` yolunu izler; araya yalnız bir dolaylı atlama girer. Bu düzene **tembel bağlama** (*lazy binding*) denir, çünkü bir sembol ancak ilk kullanıldığında çözülür. Tablo 8.7 adımları sıralar.



Şekil 8.18: Tembel bağlamada ilk çağrı: PLT, GOT'taki adres üzerinden çözücüye gider.

Adım	Ne olur?	GOT[printf]
1	`main`, `call printf@plt` ile PLT taslağına gider.	çözücü
2	PLT, `jmp *GOT[printf]` ile GOT'taki adrese atlar; bu adres çözücüdür.	çözücü
3	`ld.so` çözücüsü sembolü arar, bulduğu adresi GOT girdisine yazar.	printf
4	Çözücü `printf`'e atlar. Kod sayfaları değişmedi; yalnız sürece özel GOT sayfası değişti.	printf
5	Sonraki çağrılar: PLT → GOT → `printf`, tek dolaylı atlama.	printf

Tablo 8.7: `printf`'in ilk ve sonraki çağrıları.

Değişenin kod değil GOT olması, paylaşım için kritiktir. Kod sayfaları hiç yazılmadığı için bütün süreçler arasında paylaşılabilir kalır. Her sürecin yalnız kendi GOT sayfası farklıdır. Bu işi işletim sistemi değil, kullanıcı alanında çalışan dinamik bağlayıcı yapar.

Tembel bağlamanın güvenlik açısından bir zayıflığı vardır: GOT yazılabilir olduğu için bir saldırgan bellek taşmasıyla oradaki bir adresi değiştirip denetimi ele geçirebilir. Bu yüzden birçok dağıtım, programları `-z now` seçeneğiyle derler. Bu "tam RELRO" düzeninde bütün semboller yükleme anında çözülür ve GOT ardından salt okunur yapılır.

Sık yapılan hata

İlk çağrıdan sonra `call printf@plt` komutunun hedefinin değiştiğini ya da libc'nin `printf` gövdesinin yeniden yazıldığını düşünmek. İkisi de olmaz: kod sayfaları salt okunurdur ve süreçler arasında paylaşılır. Değişen tek yer sürece özel GOT girdisidir.

8.3.4 Statik ve paylaşımlı kütüphane oluşturma

Bir C kaynak dosyasından (`func.c`) iki tür kütüphane üretilebilir. Statik kütüphane şu komutlarla oluşturulur ve kullanılır:

```
$ gcc -c func.c -o func.o
$ ar rcs libfunc.a func.o
$ gcc main.c -o main -static -L. -lfunc
$ ./main
```

`ar` aracı nesne dosyalarını tek bir arşivde (`libfunc.a`) toplar. Bağlayıcı bu arşivden yalnız programın gerçekten kullandığı nesneleri kopyalar. `-static` seçeneği, C kütüphanesinin de statik sürümünün (`libc.a`) kullanılmasını ister; sonuçta hiçbir paylaşımlı kütüphaneye bağımlı olmayan bir dosya çıkar.

Paylaşımlı kütüphane için adımlar şöyledir:

```
$ gcc -fPIC -c func.c -o func.o
$ gcc -shared -o libfunc.so func.o
$ gcc main.c -o main -L. -lfunc
$ export LD_LIBRARY_PATH=$(pwd)
$ ./main
$ ldd ./main   (libfunc.so => ./libfunc.so)
```

Buradaki kilit seçenek `-fPIC` 'tir: **konumdan bağımsız kod** (*position-independent code*). Böyle derlenen kod mutlak adres kullanmaz. Atlamalar program sayacına göreli yapılır, global veriye ise GOT üzerinden erişilir. Bu sayede aynı fiziksel sayfa her süreçte farklı bir sanal adrese eşlenebilir. Adres alanı yerleşiminin rastgeleleştirilmesi (*address space layout randomization*, ASLR) de buna dayanır. `ldd` komutu, programın hangi paylaşımlı kütüphaneleri hangi dosyalardan alacağını gösterir.



Şekil 8.19: Statik bağlamada kütüphane kodu her programın içine kopyalanır; paylaşımlı kütüphanede tek kopya iki sürecin adres alanına eşlenir.

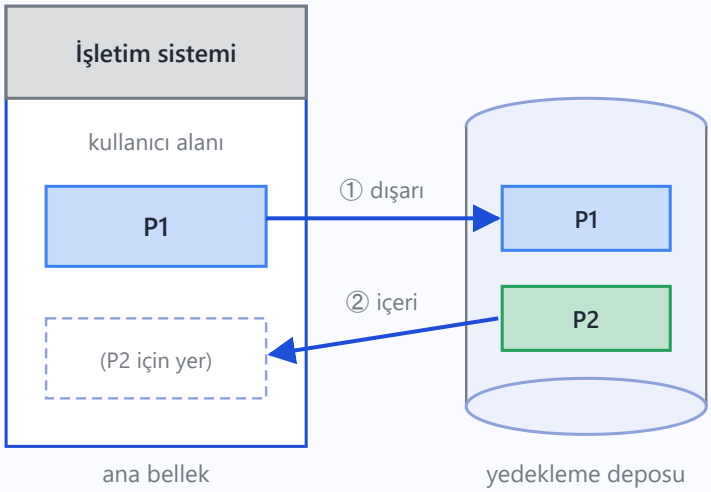
`LD_LIBRARY_PATH` değişkeni deneme için kullanışlıdır ama kalıcı bir çözüm değildir. Kalıcı çözüm ya kütüphanenin arama yolunu çalıştırılabilir dosyaya gömmek (`-Wl,-rpath,'$ORIGIN'`) ya da kütüphaneyi sistem dizinine kurup `ldconfig` çalıştırmaktır. Bugünkü dağıtımlar programların kendisini de konumdan bağımsız (*position-independent executable*, PIE) derler. Böylece programın kendi kodu da ASLR ile her çalıştırmada farklı bir adrese yüklenir.

8.4 Takas

Bellekteki süreçlerin toplam ihtiyacı fiziksel belleği aşarsa ne olur? Bir çözüm, o an çalışmayan süreçleri geçici olarak diske taşımaktır. Bu fikre **takas** (*swapping*) denir.

8.4.1 Standart takas

Standart takasta bir sürecin tamamı geçici olarak bir **yedekleme deposuna** (*backing store*), genellikle hızlı bir diske yazılır. Süreç yeniden çalışacağı zaman belleğe geri getirilir (Şekil 8.20). Takas sayesinde süreçlerin toplam bellek ihtiyacı fiziksel belleği aşabilir ve aynı anda daha çok süreç yürütülebilir; yani çoklu programlama derecesi artar.



Şekil 8.20: Standart takas: P1 sürecinin tamamı yedekleme deposuna yazılır, P2 belleğe okunur.

Takastan dönen süreç, belleğin daha önceki yerine değil, o an boş olan başka bir yere yerleşebilir. Bu yüzden takas, yürütme zamanında bağlama gerektirir. Derleme ya da yükleme zamanında bağlanmış bir süreç ancak tam olarak eski yerine dönebilir.

Standart takasın bedeli aktarım süresidir. Silberschatz'ın örneğinde 100 MB'lık bir süreç, 50 MB/s aktarım hızlı bir diske yazılır. Dışarı yazmak 100 / 50 = 2 s, geri okumak da 2 s sürer; gidiş-dönüş 4 saniyedir. Bu süre boyunca takas edilen süreç çalışamaz.

Takas edilecek süreç seçilirken bir kısıt daha vardır: bekleyen giriş/çıkış işlemi olan bir süreç takas edilmemelidir. Süreç bir aygıttan veri bekliyorsa, aygıt veriyi doğrudan bellek erişimiyle (DMA) sürecin tamponuna yazacaktır. Süreç takas edilip yerine başka bir süreç yüklenirse, DMA aktarımı eski fiziksel adrese, yani artık başka sürecin belleğine yazar. Çare ya böyle bir süreci takas etmemek ya da giriş/çıkışı her zaman çekirdek tamponları üzerinden yapmaktır.

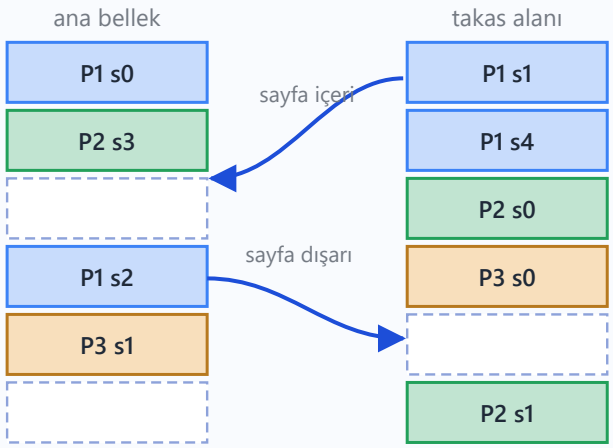
Örnek 8.4

200 MB'lık bir süreç 50 MB/s aktarım hızlı bir diske takas ediliyor; gecikmeler ihmal ediliyor. Gidiş-dönüş ne kadar sürer?

1. **Dışarı.** 200 / 50 = 4 s.
2. **İçeri.** Yine 4 s.
3. **Toplam.** 8 s. Bir zaman diliminin onlarca milisaniye olduğu düşünülürse, bütün süreci taşımanın neden bugün kabul edilmediği açıktır.

8.4.2 Sayfa takası ve bugünkü sistemler

Bugün süreçlerin boyutları gigabaytlarla ölçülür. 1 GiB'lık bir süreç, 3 GB/s okuma-yazma hızlı bir NVMe SSD'de bile gidiş-dönüşte yaklaşık 2 × 1,07 / 3 ≈ 0,72 s tutar. Bu yüzden Linux, Windows ve macOS süreçleri bütün olarak değil, sayfa sayfa takas eder (Şekil 8.21). Yalnız uzun süredir kullanılmayan, yani "soğuk" sayfalar diske gider. 4 KiB'lık bir sayfa onlarca mikrosaniyede geri gelir.



Şekil 8.21: Sayfalamalı takas: süreçlerin tamamı değil, tek tek sayfaları bellek ile takas alanı arasında taşınır.

Bugünkü sistemler diske gitmeden önce bir ara adım daha kullanır: sıkıştırma. Soğuk sayfalar önce bellekte sıkıştırılıp küçültülür. Linux'ta zswap ve zram, Windows'ta bellek sıkıştırma, macOS'ta sıkıştırıcı bunu yapar. Sıkıştırılmış bir sayfayı açmak, onu diskten okumaktan çok daha hızlıdır.

Mobil sistemlerde durum farklıdır. Flash belleğin yazma ömrü sınırlıdır ve her yazma enerji harcar. Bu yüzden telefonlarda geleneksel takas ya hiç yoktur ya da çok sınırlıdır. Android zram ile sıkıştırma yapar; bellek yine yetmezse `lmkd` arka plandaki uygulamaları sonlandırır. iOS da sıkıştırma kullanır ve gerekince uygulamaları sonlandırır (jetsam). Kullanıcı açısından bu, arka plandaki bir uygulamaya dönüldüğünde onun baştan başlaması olarak görülür.

8.5 Bitişik tahsis

Taban ve limit yazmaçlarıyla çalışan bir sistemde her süreç bellekte tek parça, kesintisiz bir bölge kaplar. Bu kesim, böyle bir sistemde boş bölgelerin nasıl seçildiğini ve zamanla neden küçük, işe yaramaz parçalara bölündüğünü anlatır.

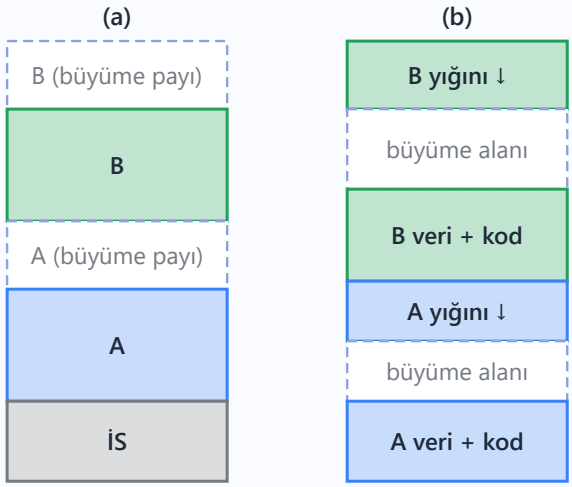
8.5.1 Bitişik bellek tahsisi

Bellek önce iki bölgeye ayrılır: biri işletim sistemine, öteki kullanıcı süreçlerine. İşletim sistemi belleğin alt ya da üst ucuna konabilir. Linux ve Windows çekirdeği yüksek adreslere koyar. Kullanıcı bölgesinde ise her süreç tek bir bitişik bölgededir ve koruma taban-limit yazmaçlarıyla sağlanır.

> **Tanım 8.4**
>
> **Bitişik tahsis** (*contiguous allocation*), her sürece tek parça, ardışık adreslerden oluşan bir bellek bölgesi verilmesidir.

Kullanıcı bölgesini süreçlere bölmenin iki yolu vardır. **Sabit bölümlemede** bellek baştan belirli sayıda ve boyda bölüme ayrılır; IBM OS/360'ın MFT sürümü böyle çalışırdı. Her bölüme bir süreç girer, dolayısıyla bölüm sayısı aynı anda bellekte bulunabilecek süreç sayısını, yani çoklu programlama derecesini belirler. Küçük bir süreç büyük bir bölüme girerse bölümün geri kalanı boşa gider. **Değişken bölümlemede** ise (OS/360 MVT) her sürece tam ihtiyacı kadar yer verilir. Süreçlerin arasında kalan boş bloklara **delik** (*hole*) denir.

Bir sorun daha vardır: süreçler büyür. Bir program `malloc` ile heap'ten yer ister, fonksiyon çağrıları yığını derinleştirir. Bitişik bir bölgenin hemen üstünde başka bir süreç varsa büyüme imkânsızdır. Bu yüzden süreçlere baştan bir büyüme payı bırakılır (Şekil 8.22). (a)'da yalnız veri kesimi büyür. (b)'de yığın yukarıdan aşağı, veri aşağıdan yukarı büyür ve pay ikisinin arasında kalır. Süreç takas edilip geri getirildiğinde yalnız kullandığı kısım taşınır, boş pay taşınmaz.



Şekil 8.22: Büyüyen süreçler için pay bırakmak: (a) veri kesimi büyür, (b) yığın aşağı, veri yukarı doğru büyür.

8.5.2 Değişken bölümleme

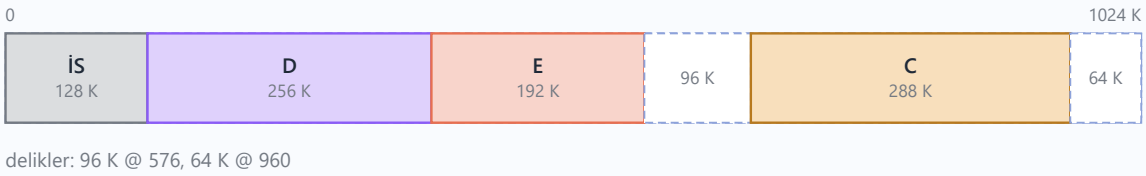
Değişken bölümlemede işletim sistemi bir tablo ya da liste tutar: hangi bölgeler hangi süreçlere ait, hangi bölgeler delik. Bir süreç geldiğinde onu alabilecek kadar büyük bir delik bulunur. Süreç deliğin bir ucuna yerleşir, artan kısım daha küçük bir delik olarak kalır. Süreç bittiğinde alanı deliğe döner ve komşu deliklerle hemen birleştirilir.

1024 K'lık bir bellekte ilk 128 K işletim sistemine ait olsun ve süreçler ilk uygun deliğe yerleşsin. Tablo 8.8 olayları sırayla izler.

Olay	Yerleşim	Delikler sonra
başlangıç	—	896 K @ 128
A (320 K) gelir	128–448	576 K @ 448
B (224 K) gelir	448–672	352 K @ 672
C (288 K) gelir	672–960	64 K @ 960
A sonlanır	—	320 K @ 128, 64 K @ 960
D (256 K) gelir	128–384	64 K @ 384, 64 K @ 960
B sonlanır	—	288 K @ 384, 64 K @ 960
E (192 K) gelir	384–576	96 K @ 576, 64 K @ 960

Tablo 8.8: Değişken bölümlemede bir olay dizisi (1024 K bellek, İS 0–128 K). Delikler "boy @ başlangıç" biçiminde; Python ile yeniden izlendi.

D, A'nın bıraktığı 320 K'lık deliğe girer ve geriye 64 K'lık küçük bir delik kalır. B sonlanınca onun 224 K'lık alanı solundaki 64 K'lık delikle birleşir ve 288 K'lık tek bir delik oluşur. Bitişik delikler birleştirilmeseydi, 288 K'lık bir isteğe yer bulunamazdı. E bu deliğe girdikten sonra toplam boş alan 96 + 64 = 160 K'dır, ama iki parçaya dağılmıştır. 100 K'lık bir süreç gelse, toplam yer yetmesine karşın sığmaz.



Şekil 8.23: Olay dizisinin sonu: D, E ve C bellekte; 96 K ve 64 K'lık iki delik birbirinden ayrı.

8.5.3 Yerleştirme algoritmaları

Gelen bir isteği alabilecek birden çok delik varsa hangisi seçilmeli? Bu sorunun klasik cevapları şunlardır.

İlk uyan (*first fit*)
: Liste baştan taranır ve yeterince büyük ilk delikte durulur. Hızlıdır. Zamanla listenin başında küçük delikler birikir.

Sonraki uyan (*next fit*)
: İlk uyanın bir türüdür; arama baştan değil, bir önceki aramanın kaldığı yerden sürer. Bays'in 1977'deki benzetimlerine göre ilk uyandan biraz kötü sonuç verir.

En iyi uyan (*best fit*)
: Bütün liste taranır ve isteğe en yakın boydaki delik seçilir. Liste boyuna sıralı tutulmuyorsa yavaştır. Geriye işe yaramayacak kadar küçük delikler bırakır.

En kötü uyan (*worst fit*)
: En büyük delik seçilir; amaç, artan parçanın başka bir istek için kullanılabilecek kadar büyük kalmasıdır. Benzetimlerde iyi sonuç vermez.

Hızlı uyan (*quick fit*)
: Sık istenen boyutlar (4 K, 8 K, 12 K …) için ayrı listeler tutulur. Uygun deliği bulmak çok hızlıdır; ama serbest kalan bir bloğun komşularını bulup birleştirmek pahalıdır.

Bütün algoritmalar süreçler ve delikler için ayrı listeler tutularak hızlandırılabilir: arama yalnız delik listesinde yapılır. Bugünkü `malloc` gerçeklemeleri (jemalloc, tcmalloc) hızlı uyanın genelleştirilmiş bir biçimini, boyut sınıflarını kullanır. Silberschatz ve Knuth'un aktardığı benzetimlerin sonucu açıktır. Hem zaman hem bellek kullanımı açısından ilk uyan ve en iyi uyan, en kötü uyandan iyidir. İlk uyan genellikle daha hızlıdır.

8.5.4 Kod izleme: en iyi uyan

Aşağıdaki fonksiyon, delik boylarını tutan bir dizide en iyi uyan aramasını yapar. Bulduğu deliği istek kadar küçültür ve deliğin indisini döndürür; yer yoksa −1 döndürür.

```
int en_iyi(int delik[], int n, int istek) {
  int en = -1;
  for (int i = 0; i < n; i++)
    if (delik[i] >= istek &&
        (en == -1 || delik[i] < delik[en]))
      en = i;
  if (en != -1) delik[en] -= istek;
  return en;
}
/* delik = 120, 450, 230, 300, 180 (K) */
/* en_iyi(delik, 5, 210); en_iyi(delik, 5, 160); */
```

`en` değişkeni o ana kadar bulunan en iyi adayın indisidir; −1 "henüz aday yok" demektir. Döngü her delik için iki koşula bakar. Birincisi deliğin isteğe yetmesidir (`delik[i] >= istek`). İkincisi ya henüz aday olmaması ya da bu deliğin mevcut adaydan küçük olmasıdır. İki koşul birlikte sağlanırsa aday değişir. Döngü bitince seçilen delik istek kadar küçültülür. Fonksiyon her çağrıda bütün listeyi gezer, yani n delik için O(n) zaman harcar.

Çağrı	i	delik[i]	Karar	en
istek = 210	0	120	120 < 210, sığmaz	−1
	1	450	sığar, ilk aday	1
	2	230	sığar, 450'den küçük	2
	3	300	sığar ama 230'dan büyük	2
	4	180	sığmaz	2
Dönüş 2; delik[2] = 230 − 210 = 20. Delikler: 120, 450, 20, 300, 180.				
istek = 160	0	120	sığmaz	−1
	1	450	sığar, ilk aday	1
	2	20	sığmaz	1
	3	300	sığar, 450'den küçük	3
	4	180	sığar, 300'den küçük	4
Dönüş 4; delik[4] = 180 − 160 = 20. Delikler: 120, 450, 20, 300, 20.				

Tablo 8.9: `en_iyi` fonksiyonunun iki çağrısı (delikler 120, 450, 230, 300, 180 K). Değerler Python ile doğrulandı.



Şekil 8.24: İki çağrıdan sonra delik listesi. En iyi uyan iki kez 20 K'lık işe yaramaz delik bıraktı.

İzleme en iyi uyanın tipik davranışını gösterir: her seferinde isteğe en yakın deliği seçtiği için geriye çok küçük artıklar kalır. Aynı iki istek ilk uyanla karşılansaydı ikisi de 450'lik deliğe girerdi: önce 240, sonra 80 K kalırdı. En kötü uyanla 210 K yine 450'ye girer (240 kalır), 160 K ise o an en büyük delik olan 300'e girerdi (140 kalır). Hangi artıkların gelecekte işe yarayacağı, sonraki isteklere bağlıdır.

8.5.5 Dört algoritma aynı istek dizisinde

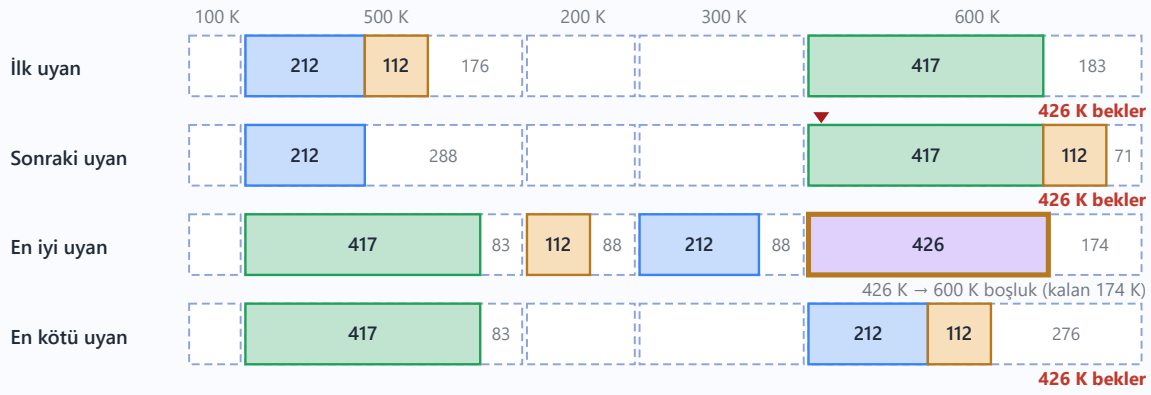
Silberschatz'ın klasik alıştırması, algoritmaların farkını tek bir örnekte gösterir.

Örnek 8.5

Delikler adres sırasıyla 100, 500, 200, 300 ve 600 K'dır. İstekler sırayla 212, 417, 112 ve 426 K'dır. İlk uyan, sonraki uyan, en iyi uyan ve en kötü uyan her isteği nereye koyar?

1. **212 K.** İlk uyan ve sonraki uyan ilk yeterli delik olan 500'e koyar (288 kalır). En iyi uyan yetenlerin (500, 300, 600) en küçüğü 300'e koyar (88 kalır). En kötü uyan 600'e koyar (388 kalır).
2. **417 K.** İlk uyanda 288 yetmez, 600'e girer (183 kalır). Sonraki uyan 500'den sonra aramayı sürdürür; 200 ve 300 yetmez, 600'e girer (183 kalır). En iyi uyanda yetenler 500 ve 600'dür; 500'e girer (83 kalır). En kötü uyanda o anki en büyük delik 500'dür (600 artık 388); 500'e girer (83 kalır).
3. **112 K.** İlk uyan baştan arar: 100 yetmez, 288 yeter (176 kalır). Sonraki uyan en son kaldığı 600'lük deliğin artığı 183'te durur (71 kalır). En iyi uyan yetenlerin en küçüğü olan 200'e koyar (88 kalır). En kötü uyan en büyük delik 388'e koyar (276 kalır).
4. **426 K.** İlk uyanda en büyük delik 300, sonraki uyanda 300, en kötü uyanda 300'dür: yer yoktur, istek bekler. Yalnız en iyi uyanda 600'lük delik hâlâ bütündür; istek oraya girer (174 kalır).

Bu dizide dört isteğin hepsini yalnız en iyi uyan yerleştirebilir. Diğerlerinde toplam boş alan 426 K'dan fazladır, ama bu alan tek parça değildir.



Şekil 8.25: Dört algoritmanın aynı istek dizisinden sonraki durumu. 426 K'yı yalnız en iyi uyan yerleştirebildi.

Bu sonuç en iyi uyanın her zaman üstün olduğu anlamına gelmez. Örnek, büyük deliği bütün tutmanın işe yaradığı bir dizidir. Başka dizilerde sonuç değişir. Çalışma sorularındaki diziyi ele alalım: delikler 150, 400, 80, 250, 600 K; istekler 230, 120, 380, 90 K. Bu kez en kötü uyan 380 K'yı yerleştiremez. İlk uyan ve en iyi uyan ise dört isteği de karşılar. Aynı karşılaştırmayı farklı dizilerle denemek için Bellek Tahsisi Simülatörü kullanılabilir.

Sık yapılan hata

En kötü uyanda ilk isteğin seçtiği deliği sonraki isteklerde de en büyük delik saymak. Delikler her yerleştirmeden sonra küçülür. 212 K'dan sonra 600'lük delik 388'e iner; 417 K için o anki en büyük delik artık 500'dür.

8.5.6 İç ve dış parçalanma

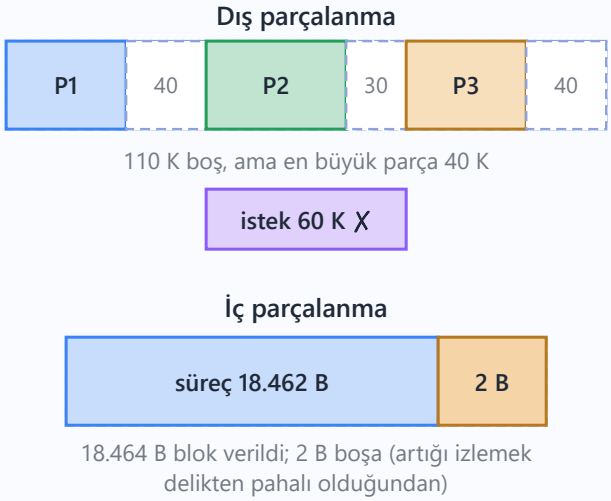
Değişken bölümlemede ortaya çıkan ve önceki örneklerde gördüğümüz sorunun adı **dış parçalanmadır**.

Tanım 8.5

Dış parçalanma (*external fragmentation*), bir isteği karşılayacak kadar toplam boş bellek bulunduğu hâlde bu belleğin bitişik olmamasıdır. **İç parçalanma** (*internal fragmentation*) ise sürece verilen bloğun istenenden büyük olması ve aradaki farkın bloğun içinde boşa gitmesidir.

Dış parçalanmanın büyüklüğünü Knuth'un **%50 kuralı** özetler. İlk uyanla çalışan bir sistemde N tahsisli blok varken, ortalamada yaklaşık 0,5 N blok parçalanma yüzünden boşa gider. Boşa giden blokların payı 0,5 N / (N + 0,5 N) = 1/3'tür; yani belleğin üçte biri kullanılamaz hâle gelebilir.

İç parçalanma ise sabit boyutlu bloklarla çalışan her düzende görülür: sabit bölümlemede, sayfalamada ve aşağıda anlatılan buddy sisteminde. Tanenbaum'un örneği ilginç bir nedeni de gösterir (Şekil 8.26). 18.464 baytlık bir delik varken 18.462 baytlık bir süreç gelirse, geriye 2 baytlık bir delik kalacaktır. Bu 2 baytı ayrı bir delik olarak listede izlemenin maliyeti, deliğin kendisinden büyüktür. Bu yüzden süreç 18.464 baytın tamamını alır ve 2 bayt bloğun içinde boşa gider.



Şekil 8.26: Üstte dış parçalanma: 110 K boş, ama en büyük parça 40 K olduğu için 60 K'lık istek sığmaz. Altta iç parçalanma: verilen bloğun 2 baytı kullanılmaz.

Dış parçalanmanın iki çözümü vardır. Birincisi *sıkıştırmadır*: süreçler kaydırılıp delikler tek bir blokta toplanır. İkincisi daha köklüdür: bir sürecin adres alanının fiziksel bellekte bitişik olma zorunluluğunu kaldırmak. Sayfalama ve kesimleme bu ikinci yolu izler.

İç parçalanmanın iyi bilinen bir kaynağı, fiziksel çerçeveleri ayırmak için Linux'ta da kullanılan **buddy (kardeş) sistemidir** (*buddy system*). Bu düzende bloklar hep ikinin kuvveti boyundadır. Bir istek, ona yeten en küçük ikinin kuvvetine yuvarlanır. Uygun boyda boş blok yoksa daha büyük bir blok ikiye bölünür; oluşan iki yarıya birbirinin "kardeşi" denir. Bir blok serbest kalınca kardeşi de boşsa ikisi yeniden birleşir. Kardeşin adresi tek bir XOR işlemiyle bulunur: adres $\oplus$ boy.

Örnek 8.6

1024 K'lık boş bir blokla başlayan bir buddy sisteminde sırayla A = 100 K, B = 240 K, C = 64 K ve D = 256 K istensin. Bloklar nereye yerleşir ve ne kadar iç parçalanma oluşur?

1. **A = 100 K.** 128 K'ya yuvarlanır. 1024 bölünür (512 + 512), soldaki 512 bölünür (256 + 256), soldaki 256 bölünür (128 + 128). A, 0 adresindeki 128 K'lık bloğu alır. İç parçalanma 28 K. Boş: 128 K @ 128, 256 K @ 256, 512 K @ 512.
2. **B = 240 K.** 256 K'ya yuvarlanır ve 256'daki boş bloğa girer. İç parçalanma 16 K.
3. **C = 64 K.** Tam 64 K'dır. 128'deki boş blok bölünür (64 + 64); C, 128'deki yarıyı alır. Boş: 64 K @ 192, 512 K @ 512.
4. **D = 256 K.** 256 K boş blok yoktur; 512'deki blok bölünür ve D, 512'deki yarıyı alır. Boş: 64 K @ 192, 256 K @ 768.
5. **Serbest bırakma.** A bırakılınca kardeşi (0 $\oplus$ 128 = 128) C'nin içinde olduğundan birleşme olmaz. C bırakılınca kardeşi 192 boştur: 128 K @ 128 oluşur; onun kardeşi 0'daki A bloğu da boştur: 256 K @ 0 oluşur. B ve D de bırakılınca bloklar 512'ye, sonra 1024'e

kadar birleşir.

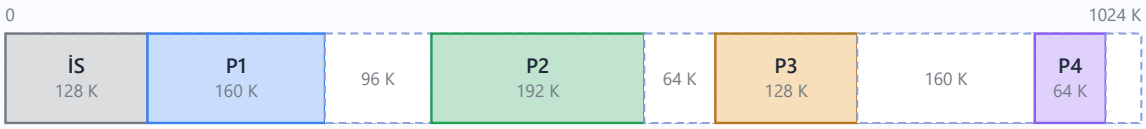
Toplam iç parçalanma 28 + 16 = 44 K'dır. (Python ile adım adım izlendi.)

8.5.7 Sıkıştırma

Tanım 8.6

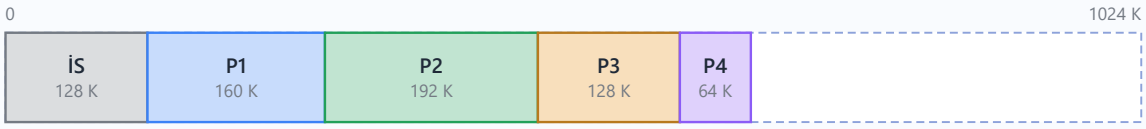
Sıkıştırma (*compaction*), bellekteki süreçleri bir uca doğru kaydırıp bütün delikleri tek bir büyük blokta toplamaktır.

Şekil 8.27 başlangıç durumunu gösterir. İşletim sistemi 0–128 K'dadır. P1 128'de (160 K), P2 384'te (192 K), P3 640'ta (128 K), P4 928'de (64 K) başlar. Aralarda 96, 64, 160 ve 32 K'lık dört delik vardır; toplam 352 K boş alan bulunduğu hâlde en büyük delik 160 K'dır.



Şekil 8.27: Sıkıştırmadan önce: dört süreç arasında dört delik, toplam 352 K.

Sıkıştırma süreçleri sırayla aşağı kaydırır. P1 zaten işletim sisteminin hemen üstündedir. P2, 384'ten 288'e (128 + 160) taşınır. P3, 640'tan 480'e (288 + 192), P4 de 928'den 608'e (480 + 128) taşınır. Sonunda 672 ile 1024 arasında tek bir 352 K'lık delik kalır (Şekil 8.28). Bu iş için 192 + 128 + 64 = 384 K kopyalanmıştır. Her taşınan sürecin taban yazmacı yeni adresle güncellenir.



Şekil 8.28: Sıkıştırmadan sonra: süreçler bitişik, tek bir 352 K'lık delik.

Sıkıştırma yalnız yürütme zamanında bağlama kullanan sistemlerde mümkündür. Süreçteki adresler yükleme anında sabitlenmişse, süreç taşındıktan sonra yanlış yerleri gösterir. Taban yazmacı kullanan sistemde ise yalnız yazmacın değeri değişir.

Sıkıştırma pahalıdır. Tanenbaum'un hesabına göre 8 baytı 8 ns'de kopyalayan bir makine, yani saniyede 1 GB, 16 GB'lık belleği yaklaşık 16 s'de sıkıştırır. Bugünkü yaklaşık 20 GB/s'lik kopyalama hızıyla bile iş yaklaşık bir saniye sürer (16 / 20 = 0,8 s; 16 GiB için ≈ 0,86 s). Bu sırada süreçler durur.

Sıkıştırma (*compaction*), veri sıkıştırma (*compression*) ile karıştırılmamalıdır. Amaç veriyi küçültmek değil, blokları bitiştirmektir.

Sıkıştırma tamamen tarihe karışmış da değildir. Linux, büyük sayfalar için bitişik fiziksel çerçeve üretmek amacıyla arka planda sayfa göçü yapar; bu işi `kcompactd` çekirdek iş parçacığı yürütür.

8.6 Boş alan yönetimi

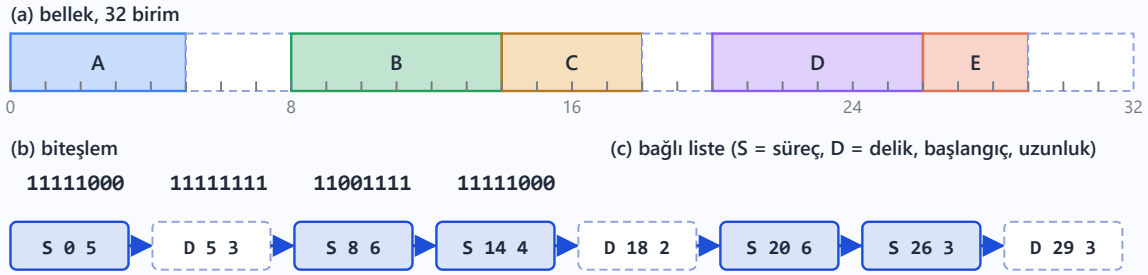
Yerleştirme algoritmalarının hepsi aynı bilgiye dayanır: hangi bölgeler boş? Bu bilginin nasıl tutulduğu, aramanın ve serbest bırakmanın ne kadar hızlı olacağını belirler. İki temel yapı vardır: biteşlem ve bağlı liste.

8.6.1 Biteşlem ve bağlı liste

Bellek önce sabit boyutlu *tahsis birimlerine* bölünür; birim birkaç sözcük de olabilir, birkaç kilobayt da.

> **Tanım 8.7**
>
> **Biteşlem** (*bitmap*), her tahsis birimi için bir bit tutan dizidir: 0 birimin boş, 1 dolu olduğunu gösterir. **Bağlı liste** yaklaşımında ise her süreç ve her delik için bir düğüm tutulur; düğümde türü (S: süreç, D: delik), başlangıç birimi ve uzunluk bulunur.

Şekil 8.29 aynı belleğin iki gösterimini verir. 32 birimlik bellekte A, B, C, D ve E süreçleri ile üç delik vardır. Biteşlemin ilk baytı 11111000'dır: 0–4 arası birimler A'ya ait, 5–7 boştur. Bağlı listede aynı bilgi “S 0 5, D 5 3, S 8 6, …” düğümleriyle tutulur.



Şekil 8.29: (a) Beş süreç ve üç delikli 32 birimlik bellek, (b) aynı durumun biteşlemi, (c) bağlı listesi.

	Biteşlem	Bağlı liste
Yapı	Her tahsis birimine 1 bit	Her süreç ya da delik için bir düğüm
Boyut	Sabit: birim 4 B ise belleğin 1/33'ü; 16 GiB bellek ve 4 KiB birimle 512 KiB	Süreç ve delik sayısıyla orantılı

	Biteşlem	Bağlı liste
Arama	k ardışık 0 bulmak yavaş	Delik listesi gezilir; adrese göre sıralıysa birleştirme kolay

Tablo 8.10: Biteşlem ile bağlı listenin karşılaştırılması.

Biteşlemin boyutu birim büyüklüğüne bağlıdır ve sabittir. Birim 4 bayt, yani 32 bit ise her 32 bitlik bellek için 1 bit harcanır; biteşlem belleğin 1/33'ünü kaplar. 16 GiB bellek 4 KiB'lık birimlere bölünürse 2^{34} / 2^{12} = 2^{22} birim olur. Biteşlem de 2^{22} bit, yani 512 KiB tutar. Birim küçüldükçe biteşlem büyür; birim büyüdükçe son birimdeki iç parçalanma artar.

Örnek 8.7

512 MiB'lık bellek 4 KiB'lık birimlerle bir biteşlemde izleniyor. Biteşlem kaç bayttır?

1. **Birim sayısı.** 2^{29} / 2^{12} = 2^{17} birim.
2. **Bayt.** Her birime 1 bit: 2^{17} bit / 8 = 2^{14} bayt = 16 KiB.

Sık yapılan yanlış, bit sayısını (128 Ki) bayt sanıp 128 KiB bulmaktır.

8.6.2 Kod izleme: biteşlemde boş alan arama

Biteşlemin zayıf yanı aramadır. k birimlik bir süreç için yer bulmak, dizide art arda k tane 0 bulmak demektir. Aşağıdaki fonksiyon bunu doğrusal bir taramayla yapar.

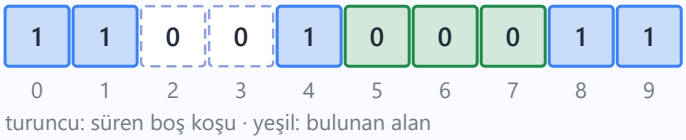
```
/* k ardışık boş (0) birim ara */
int bul(const char bit[], int n, int k) {
  int bas = 0, uz = 0;
  for (int i = 0; i < n; i++) {
    if (bit[i] == 0) {
      if (uz == 0) bas = i;  /* yeni koşu */
      if (++uz == k) return bas;
    } else
      uz = 0;                /* koşu bozuldu */
  }
  return -1;
}
```

`uz` o anda sürmekte olan sıfır koşusunun uzunluğunu, `bas` da koşunun başladığı indisi tutar. Bir 0 görülünce, koşu yeni başlıyorsa (`uz == 0`) başlangıç kaydedilir ve uzunluk bir artar. Uzunluk k'ye ulaşınca başlangıç döndürülür. Bir 1 görülünce koşu bozulur ve `uz` sıfırlanır. Tablo 8.11, 10 birimlik 1 1 0 0 1 0 0 0 1 1 biteşleminde k = 3 aramasını izler.

i	bit[i]	bas	uz	Açıklama
0	1	0	0	dolu birim

i	bit[i]	bas	uz	Açıklama
1	1	0	0	dolu birim
2	0	2	1	yeni koşu
3	0	2	2	koşu uzar, henüz k'ye ulaşmadı
4	1	2	0	2 birimlik koşu bozuldu; bu delik süreç için küçük
5	0	5	1	yeni koşu
6	0	5	2	koşu uzar
7	0	5	3	uz = k: 5 döndürülür

Tablo 8.11: `bul(bit, 10, 3)` izlemesi. Python ile doğrulandı.



turuncu: süren boş koşu · yeşil: bulunan alan

Şekil 8.30: Arama sonucu: 5, 6 ve 7 numaralı birimler süreç için ayrılır.

Fonksiyon 5 döndürür. Ayırıcı ardından 5, 6 ve 7 numaralı bitleri 1 yapar. Arama her seferinde dizinin başından başlar ve en kötü durumda bütün biteşlemi gezer: n birim için O(n). Büyük bir biteşlemde bu yavaştır. Gerçek gerçeklemeler biteşlemi sözcük sözcük tarayarak ve donanımın "ilk sıfır biti bul" komutlarını kullanarak hızlandırır, ama doğrusal yapı değişmez.

Örnek 8.8

18 birimlik belleğin biteşlemi 1 1 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1'dir (birim 0'dan başlar). İlk uyanla sırayla 3, 4 ve 5 birimlik istekler geliyor. Her biri nereye yerleşir?

1. **Sıfır koşuları.** [2] (1 birim), [4–5] (2 birim), [8–10] (3 birim), [13–16] (4 birim).
2. **3 birim.** İlk iki koşu yetmez; 8'den başlayan koşu tam yeter. 8–10 dolar.
3. **4 birim.** Kalan koşular [2], [4–5], [13–16]; yalnız 13 yeter. 13–16 dolar.
4. **5 birim.** Geriye 3 boş birim kalmıştır ve en uzun koşu 2'dir: istek karşılanamaz. Başlangıçta da 4'ten uzun koşu yoktu.

8.6.3 Bağlı listede serbest bırakma

Bağlı listenin güçlü yanı serbest bırakmadır. Liste adrese göre sıralı tutulursa, sonlanan bir X sürecinin iki komşusu listede hemen önünde ve arkasındadır. Komşuların türüne göre dört durum vardır (Tablo 8.12).

Sol komşu	Sağ komşu	Ne olur?	Düğüm sayısı
süreç	süreç	X'in düğümü D'ye çevrilir	değişmez
süreç	delik	X sağdaki delikle birleşir	1 azalır
delik	süreç	soldaki delik X kadar uzar	1 azalır
delik	delik	üç düğüm tek deliğe iner	2 azalır

Tablo 8.12: X sürecinin serbest bırakılmasında dört durum.



Şekil 8.31: İki yanında delik bulunan X sonlanınca üç düğüm tek bir deliğe iner.

Birleştirme yalnız komşulara bakarak, sabit zamanda yapılır. Son durumda önceki düğümü değiştirmek gerektiği için çift bağlı liste işi kolaylaştırır. Delikler ayrı bir listede, boyuta göre sıralı tutulursa en iyi uyan hızlanır. Liste baştan tarandığında ilk yeten delik zaten en küçük yetendir; ilk uyan en iyi uyana dönüşür. Ama bu kez serbest bırakma yavaşlar, çünkü komşu delikler artık listede yan yana değildir.

Bir ayrıntı daha önemlidir: delik listesini tutmak için ayrıca bellek gerekmez. Liste deliklerin kendi içinde saklanabilir. Her deliğin ilk sözcükleri o deliğin boyunu ve sonraki deliğin adresini tutar. Delik zaten boş olduğu için bu yer bedavadır.

8.6.4 `malloc`'un içi: başlık ve gömülü boş liste

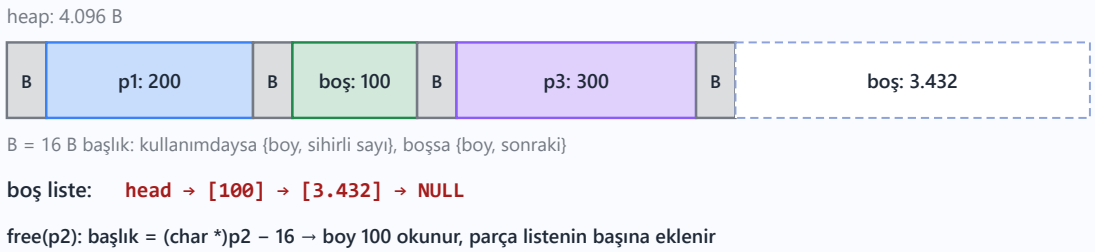
Aynı fikirler kullanıcı alanında da geçerlidir. C kütüphanesinin `malloc` ve `free` işlevleri, heap denilen bölgeyi bir boş liste ile yönetir. Bunun için özel bir sorun çözülmelidir: boş listenin düğümleri için ayrı bir veri yapısı kurulamaz, çünkü o yapıya yer ayırmak için yine `malloc` çağırmak gerekir. Çözüm, düğümü boş parçanın kendi içine gömmektir.

Örnek ayırıcıda her parçanın başında 16 baytlık bir başlık vardır. Parça kullanımdaysa başlıkta boyu ve sabit bir sihirli sayı, boşsa boyu ve sonraki boş parçanın adresi bulunur. Tablo 8.13 4096 baytlık bir heap üzerindeki işlemleri izler.

İşlem	Hesap	Boş liste sonra
başlangıç	4096 − 16 = 4080	[4080]
`p1 = malloc(200)`	200 + 16 = 216 B kullanılır; 4080 − 216 = 3864	[3864]
`p2 = malloc(100)`, `p3 = malloc(300)`	3864 − 116 − 316 = 3432	[3432]
`free(p2)`	başlık `(char *)p2 - 16`'dan boy 100 okunur	[100] → [3432]

İşlem	Hesap	Boş liste sonra
`free(p1)`	komşu boş parçayla birleşir: 216 + 116 = 332, boy 332 − 16 = 316	[316] → [3432]

Tablo 8.13: 4096 baytlık heap'te 16 baytlık başlıklı ayırıcı. Python ile doğrulandı.



Şekil 8.32: `free(p2)` sonrası heap: 100 baytlık boş parça listenin başına eklenmiştir. 150 baytlık bir istek artık oraya sığmaz.

`free` işlevi boyut almaz; boyu, kendisine verilen adresin 16 bayt gerisindeki başlıktan okur. `free(p2)` 'den sonra 100 baytlık bir boş parça vardır, ama 150 baytlık bir istek ona sığmaz: kullanıcı alanında da dış parçalanma oluşur. `free(p1)` ile p1'in parçası adres sırasındaki komşusu olan boş parçayla birleşir ve liste kısalır.

Sihirli sayı
: Başlıktaki sabit desen bir tampon taşması, aynı işaretçinin iki kez serbest bırakılması ya da yanlış bir işaretçi yüzünden bozulabilir. Kütüphane bunu fark edip programı durdurabilir.

Gerçek istek
: `malloc(n)` en az n + başlık kadar bayt tüketir. Ayrıca adresler hizalanmak zorunda olduğu için boy yukarı yuvarlanır; glibc'de hizalama 16 bayttır.

Sistem çağrısı mı?
: Hayır, `malloc` bir kütüphane işlevidir. Heap'te yer bitince `brk` / `sbrk` sistem çağrılarıyla heap'i büyütür. Büyük istekleri (glibc'de 128 KiB ve üstü) doğrudan `mmap` ile alır.

Örnek 8.9

Aynı 16 bayt başlıklı ayırıcıda 4096 baytlık heap'ten üç kez `malloc(1000)` yapılıyor. Boş parçanın boyu kaçtır? Dördüncü bir `malloc(1040)` karşılanabilir mi?

1. **Her çağrı.** 1000 + 16 = 1016 bayt tüketir.
2. **Kalan.** 4080 − 3 × 1016 = 4080 − 3048 = 1032 bayt.
3. **Dördüncü istek.** 1040 > 1032 olduğu için bu heap'ten karşılanamaz. Ayırıcı heap'i `sbrk` ya da `mmap` ile büyütür.

Sık yapılan hata

`free`'ye `malloc`'un döndürmediği bir adres vermek, örneğin `free(p + 4)`. Ayırıcı başlığı p + 4 − 16 adresinde arar; orada geçerli bir başlık yoktur. Sonuç tanımsız davranıştır. Sihirli sayı denetimi olan bir ayırıcı programı durdurur; olmayan bir ayırıcı boş listeyi bozar ve hata çok sonra, ilgisiz bir yerde ortaya çıkar.

8.7 Sayfalama

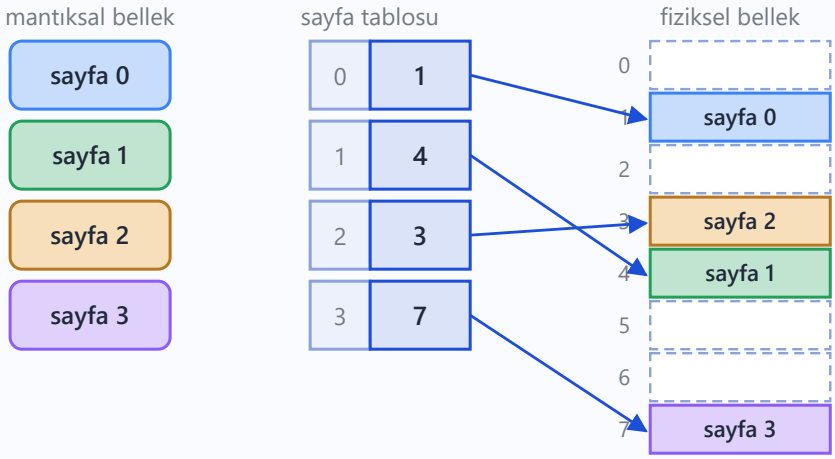
Bitişik tahsisin bütün sorunları tek bir varsayımdan doğar: sürecin fiziksel bellekte tek parça durması. Sayfalama bu varsayımı kaldırır. Süreç küçük, eşit parçalara bölünür ve her parça belleğin boş olan herhangi bir yerine konabilir. Bugünkü bütün genel amaçlı işletim sistemleri sayfalama kullanır.

8.7.1 Temel fikir

Tanım 8.8

Sayfalamada (*paging*) fiziksel bellek sabit boyutlu bloklara, **çerçevelere** (*frame*) bölünür. Mantıksal bellek de aynı boyda bloklara, **sayfalara** (*page*) bölünür. Her sürecin bir **sayfa tablosu** (*page table*) vardır; tablo, sürecin her sayfasının hangi çerçevede bulunduğunu tutar.

Bu düzenin sonuçları hemen görülür. Sürecin fiziksel bellekteki yeri bitişik olmak zorunda değildir; ardışık sayfalar birbirinden uzak çerçevelerde durabilir (Şekil 8.33). Bu yüzden dış parçalanma yoktur: her boş çerçeve, her sürecin her sayfası için uygundur. N sayfalık bir süreç için herhangi N boş çerçeve yeter. İç parçalanma ise yalnız sürecin son sayfasında kalır, çünkü süreç boyu sayfa boyunun tam katı olmayabilir.

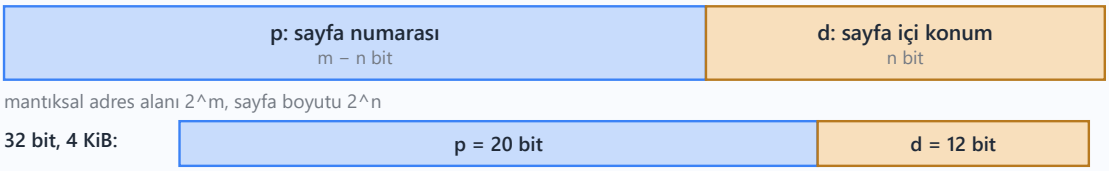


Şekil 8.33: Sayfalama. Mantıksal sayfalar, sayfa tablosu üzerinden bitişik olmayan fiziksel çerçevelere eşlenir: 0 → 1, 1 → 4, 2 → 3, 3 → 7.

İşletim sistemi hangi çerçevelerin boş olduğunu izler ve her sürecin sayfa tablosunu kurar. Takas için kullanılan yedekleme deposu da aynı boyda bloklara bölünür; böylece bir sayfa diskteki herhangi bir bloğa yazılıp herhangi bir çerçeveye geri okunabilir.

8.7.2 Adresin (p, d) olarak bölünmesi

Sayfalamada işlemcinin ürettiği her mantıksal adres iki parçaya bölünür: **sayfa numarası** p ve **sayfa içi konum** d (*offset*). Mantıksal adres alanı 2^m bayt, sayfa boyutu 2^n bayt ise adresin alt n biti konumu, üst m − n biti sayfa numarasını verir (Şekil 8.34).



Şekil 8.34: Mantıksal adres sayfa numarası p ve sayfa içi konum d olarak ikiye bölünür. 32 bit adres ve 4 KiB sayfada p 20 bit, d 12 bittir.

Sayfa tablosu p'yi bir çerçeve numarası f'ye çevirir. Konum hiç değişmez. Fiziksel adres çerçevenin başlangıcına konumun eklenmesiyle bulunur:

> $p = \lfloor a \,/\, \text{sayfa boyu} \rfloor$, $d = a \bmod \text{sayfa boyu}$, $\text{fiziksel} = f \times \text{sayfa boyu} + d$

Sayfa boyu ikinin bir kuvveti olduğu için bu bölme ve kalan, birer bit kaydırma ve maskeleme işlemidir. 4 KiB'lık sayfada p = a >> 12, d = a & 0xFFF'tir. Donanım bu işlemleri bedavaya yapar; ikinin kuvveti olmayan bir sayfa boyu gerçek bir bölücü devre gerektirirdi.

Örnek	Hesap	Sonuç
32 bit adres, 4 KiB sayfa	$4096 = 2^{12}$ → d 12 bit, p = 32 − 12 = 20 bit	2^{20} = 1.048.576 sayfa
0x12345678	p = 0x12345678 >> 12, d = 0x12345678 & 0xFFF	p = 0x12345 (74.565), d = 0x678 (1.656)
16 bit adres, 4 KiB (Tanenbaum)	8196 = 2 · 4096 + 4	p = 2, d = 4
48 bit (x86-64), 4 KiB	p = 48 − 12	36 bit sayfa numarası

Tablo 8.14: Adres bölme örnekleri (Python ile doğrulandı).

Onaltılık yazımda bölme özellikle kolaydır: her onaltılık hane 4 bittir, dolayısıyla 12 bitlik konum son üç hanedir. 0x12345678'in son üç hanesi 0x678 konumu, geri kalanı 0x12345 sayfa numarasını verir.

Sık yapılan hata

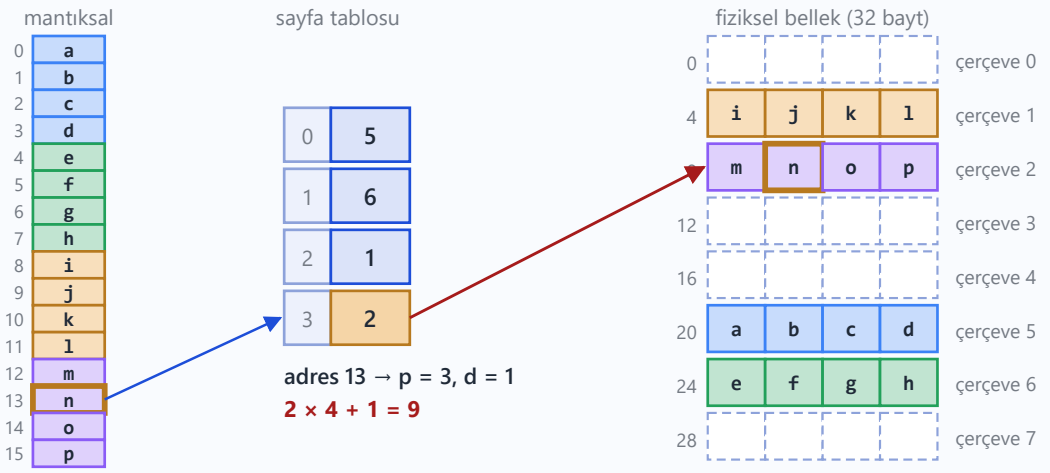
Fiziksel adresi bulurken konumu unutmak ya da mantıksal adresin tamamını çerçeve başına eklemek. 2 KiB'lık sayfada mantıksal 9000 ve 4 → 7 eşlemesi için doğru hesap şudur: p = ⌊9000 / 2048⌋ = 4, d = 9000 − 8192 = 808, fiziksel = 7 × 2048 + 808 = 15144. Yalnız 7 × 2048 = 14336 yazmak konumu unutmaktır; 14336 + 9000 = 23336 ise bütün adresi eklemektir.

8.7.3 Sayfa tablosuyla çeviri

Silberschatz'ın küçük örneği çeviriyi elle izlemeye uygundur. Sayfa boyu 4 bayt, fiziksel bellek 32 bayttır; yani 8 çerçeve vardır. Sürecin 16 baytlık mantıksal belleği a'dan p'ye kadar harflerle doludur ve dört sayfa kaplar. Sayfa tablosu 0 → 5, 1 → 6, 2 → 1, 3 → 2'dir.

Mantıksal	Harf	p	d	Çerçeve	Fiziksel
0	a	0	0	5	5 × 4 + 0 = 20
3	d	0	3	5	5 × 4 + 3 = 23
4	e	1	0	6	6 × 4 + 0 = 24
13	n	3	1	2	2 × 4 + 1 = 9

Tablo 8.15: 4 baytlık sayfalarla çeviri (Silberschatz Şekil 9.10).



Şekil 8.35: 4 baytlık sayfalarla çeviri: mantıksal 13 (n), sayfa 3, konum 1'dir; çerçeve 2'de fiziksel 9'a karşılık gelir.

Örnek iki önemli noktayı gösterir. Birincisi, sürecin harfleri fiziksel bellekte sırasını korumaz: i…l çerçeve 1'de, m…p çerçeve 2'de, a…d çerçeve 5'tedir. Program yine de belleğini a'dan p'ye ardışık görür. İkincisi, sayfa tablosu girdisinin indisi sayfa numarasıdır; sayfa numarası tabloda ayrıca saklanmaz. Tablonun 3. girdisine bakmak “sayfa 3 nerede?” sorusunun cevabıdır.

8.7.4 MMU içinde çeviri

Tanenbaum'un örneğinde sanal adres 16 bittir (64 KiB, 16 sayfa), fiziksel bellek 32 KiB'tır (8 çerçeve). Sayfa boyu 4 KiB olduğu için sanal adresin üst 4 biti sayfa numarası, alt 12 biti konumdur. Çerçeve numarası 3 bittir. Aşağıdaki komutlar beş farklı adrese erişir; sayfa tablosu da yorumlarda verilmiştir.

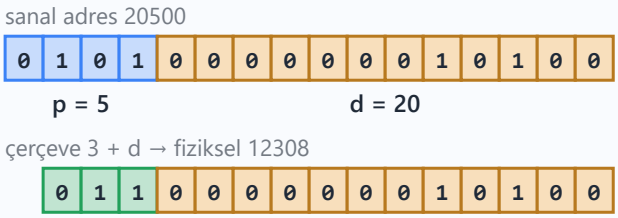
```
/* 16 bit sanal adres, 4 KiB sayfa */
MOV REG, 0
MOV REG, 8192
MOV REG, 20500
MOV REG, 8196
MOV REG, 32780

/* sayfa tablosu (sayfa → çerçeve) */
/* 0→2 1→1 2→6 3→0 4→4 5→3 */
/* 9→5 11→7, diğerleri: yok */
```

MMU her adreste aynı işi yapar. Üst 4 biti tablo indisi olarak kullanır ve girdideki 3 bitlik çerçeve numarasını alır. Alt 12 biti hiç değiştirmeden bu numaraya ekler. Tablo 8.16 beş adresin çevirisini gösterir.

Sanal	İkili (p | d)	p	d	Çerçeve	Fiziksel
0	`0000 000000000000`	0	0	2	8192
8192	`0010 000000000000`	2	0	6	24576
20500	`0101 000000010100`	5	20	3	12288 + 20 = 12308
8196	`0010 000000000100`	2	4	6	24576 + 4 = 24580
32780	`1000 000000001100`	8	12	yok	sayfa hatası

Tablo 8.16: MMU çevirisi (16 bit sanal adres, 4 KiB sayfa). Python ile doğrulandı.



Şekil 8.36: 20500 adresinin çevirisi: üst 4 bit (0101 = 5) çerçeve numarasıyla (011 = 3) değiştirilir, alt 12 bit aynen kopyalanır.

Son adres ilginçtir. 32780 = 8 × 4096 + 12'dir ve sayfa 8 eşlenmemiştir; tablodaki girdisinin mevcut/yok biti 0'dır. MMU fiziksel adres üretemez ve bir **sayfa hatası** (*page fault*) tuzağı oluşturur. Denetim işletim sistemine geçer. Sayfa diskteyse işletim sistemi onu bir çerçeveye getirip tabloyu günceller; sayfa hiç geçerli değilse süreci sonlandırır.

8.7.5 Sayfa boyutu ve iç parçalanma

Sayfalamada iç parçalanma yalnız son sayfada oluşur, ama sayfa boyu seçimini etkileyecek kadar önemlidir.

Örnek 8.10

Sayfa boyu 2048 bayt, süreç 72.766 bayttır. Süreç kaç çerçeve kaplar ve ne kadar iç parçalanma oluşur?

1. **Bölme.** 72.766 = 35 × 2048 + 1086. Süreç 35 tam sayfa ve 1086 baytlık bir artık kaplar.
2. **Çerçeve.** Artık için de bir çerçeve gerekir: 36 çerçeve.
3. **İç parçalanma.** Son çerçevede 2048 − 1086 = 962 bayt boşa gider.

En kötü durumda süreç bir sayfadan yalnız bir bayt taşar ve iç parçalanma bir sayfa eksi bir bayt olur. Süreç boyları rastgele dağılıyorsa ortalama iç parçalanma yarım sayfadır. Bu, küçük sayfa lehine bir gerekçedir. Buna karşılık küçük sayfa daha çok sayfa ve dolayısıyla daha büyük bir sayfa tablosu, daha

çok TLB ıskası demektir. Büyük sayfa ise disk ile bellek arasında daha verimli aktarım ve daha az tablo girdisi sağlar.

Tanenbaum bu ödünleşimi basit bir modelle ölçer. Ortalama süreç boyu s bayt, bir tablo girdisinin boyu e bayt, sayfa boyu p olsun. Süreç başına yaklaşık s / p sayfa ve s·e / p bayt tablo gerekir; iç parçalanma da ortalama p / 2'dir. Toplam ek yük:

> ek yük = s·e / p + p / 2

p küçülünce birinci terim, büyüyünce ikinci terim artar. Türev alınıp sıfıra eşitlenirse $-s \cdot e / p^2 + 1/2 = 0$, yani $p = \sqrt{(2 \cdot s \cdot e)}$ bulunur. s = 1 MiB, e = 8 bayt için $p = \sqrt{(2 \times 2^{20} \times 8)} = \sqrt{(2^{24})} = 2^{12} = 4096$ bayt çıkar (Şekil 8.37). Bu değerde ek yük 2048 + 2048 = 4096 bayttır; 2048 ve 8192 baytlık sayfalarda ise 5120 bayta çıkar.

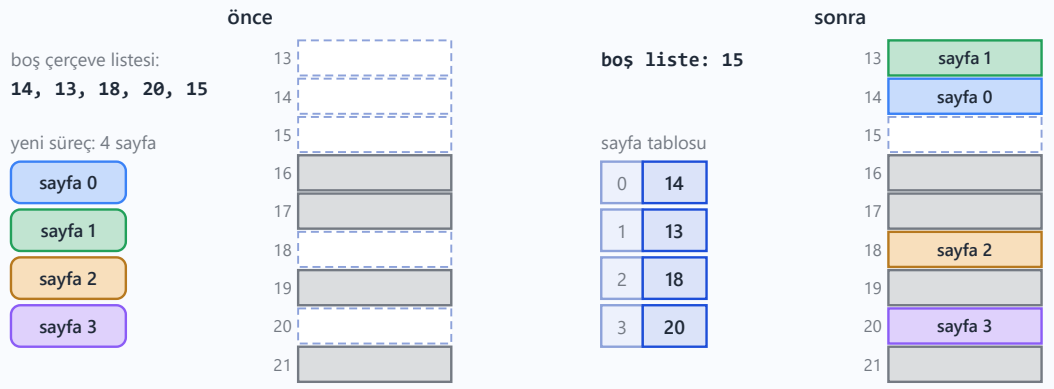


Şekil 8.37: Ek yük s·e / p + p / 2 eğrisi; s = 1 MiB, e = 8 bayt için en küçük değer 4 KiB'ta.

Gerçek sistemler de bu civarda kalır. x86-64'te temel sayfa 4 KiB'tır; ayrıca 2 MiB ve 1 GiB'lık büyük sayfalar vardır. ARM64 4, 16 ya da 64 KiB'lık sayfa kullanabilir; Apple Silicon 16 KiB kullanır.

8.7.6 Boş çerçeveler

Yeni bir süreç geldiğinde işletim sistemi boş çerçeve listesinden sürecin sayfa sayısı kadar çerçeve alır ve sayfa tablosunu doldurur. Şekil 8.38 örneğinde boş liste 14, 13, 18, 20, 15'tir. Dört sayfalık süreç listenin ilk dört çerçevesini alır: sayfa 0 → 14, 1 → 13, 2 → 18, 3 → 20. Listede yalnız 15 kalır.



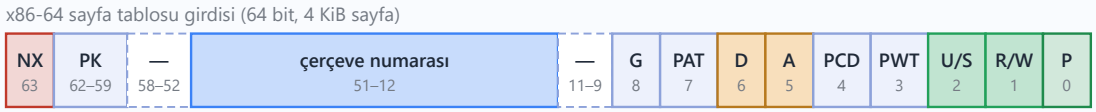
Şekil 8.38: Yeni sürecin dört sayfası boş çerçeve listesindeki 14, 13, 18 ve 20 numaralı çerçevelere yerleşir.

İşletim sistemi bunun için bir **çerçeve tablosu** tutar: her fiziksel çerçeve için çerçevenin boş olup olmadığı, doluysa hangi sürecin hangi sayfasını tuttuğu. Bu tablo yalnız tahsiste değil, başka işlerde de gerekir. Örneğin bir süreç `read` sistem çağrısıyla bir tampon adresi verdiğinde bu adres mantıksaldır. Çekirdek veriyi oraya yazabilmek için adresi sürecin sayfa tablosuyla fiziksel adrese çevirir.

Her sürecin ayrı bir sayfa tablosu olduğu için bağlam değiştirmede donanıma yeni sürecin tablosunu göstermek gerekir. Dağıtıcı bunu bir yazmacı güncelleyerek yapar; x86'da bu yazmaç CR3'tür. Bu ek iş bağlam değiştirme süresini uzatır.

8.7.7 Sayfa tablosu girdisi

Sayfa tablosu girdisi (*page table entry*, PTE) yalnız çerçeve numarasını tutmaz. Yanında, donanımın ve işletim sisteminin kullandığı bir dizi bit bulunur. Şekil 8.39 x86-64'ün 64 bitlik girdisini gösterir.



Şekil 8.39: x86-64 sayfa tablosu girdisi (64 bit, 4 KiB sayfa): NX, koruma anahtarı, çerçeve numarası, G, PAT, D, A, önbellek bitleri, U/S, R/W ve P.

Bit	Anlamı (Tanenbaum adı)	Kim yazar?
P	Mevcut/yok (*present/absent*): 0 ise erişim sayfa hatası üretir	İS
R/W, U/S, NX	Koruma: yazılabilir mi, kullanıcı modu erişebilir mi, çalıştırılabilir mi	İS
A	Erişildi (*referenced*): okuma ya da yazmada 1 olur; sayfa yer değiştirmede kullanılır	donanım (İS sıfırlar)
D	Değiştirildi (*modified*, kirli): yazmada 1 olur; kirli sayfa çıkarılırken diske yazılır	donanım
PCD, PWT, PAT	Önbelleği devre dışı bırakma ve yazma politikası: bellek eşlemeli G/Ç için	İS

Bit	Anlamı (Tanenbaum adı)	Kim yazar?
G	Global: bağlam değiştirmede TLB'den silinmez (çekirdek sayfaları)	İS

Tablo 8.17: Sayfa tablosu girdisinin bitleri.

Erişildi ve değiştirildi bitleri işletim sistemine çok değerli bilgi verir. Donanım bir sayfaya her erişimde A'yı, her yazmada D'yi 1 yapar. İşletim sistemi A bitlerini periyodik olarak sıfırlayıp hangi sayfaların son zamanlarda kullanıldığını öğrenir. D biti ise bir sayfa bellekten çıkarılırken onu diske yazmanın gerekip gerekmediğini söyler: değişmemiş bir sayfanın diskteki kopyası zaten günceldir.

Önbellek bitleri, bellek eşlemeli giriş/çıkış için gereklidir. Bir aygıtın yazmacı bellek adresine eşlenmişse, o adresten okunan değer önbellekten değil aygıttan gelmelidir. Aksi hâlde program aygıtın eski durumunu okur.

Bir sayfa bellekte değilse diskteki yeri nerede tutulur? Disk adresi PTE'de tutulmaz; donanım onu kullanmaz. P biti 0 olan bir girdinin geri kalan bitleri tamamen işletim sistemine aittir. Linux takas alanındaki yeri bu bitlere kodlar.

8.7.8 Geçerli / geçersiz biti

Sayfa tablosu, sürecin adres alanının tamamı için girdi içerir; ama süreç bu alanın hepsini kullanmaz. Kullanılmayan sayfaların girdileri **geçersiz** olarak işaretlenir. Silberschatz'ın örneğinde adres 14 bittir (0 … 16383), sayfa 2 KiB'tır, yani 8 sayfa vardır. Süreç yalnız 0 ile 10468 arasını kullanır: sayfa 0–5 geçerlidir ve 2, 3, 4, 7, 8, 9 numaralı çerçevelerdedir; sayfa 6 ve 7 geçersizdir.

Adres	p, d	Girdi	Sonuç
3.000	1, 952	geçerli, çerçeve 3	3 × 2048 + 952 = 7.096
10.468	5, 228	geçerli, çerçeve 9	9 × 2048 + 228 = 18.660 (sürecin son adresi)
12.000	5, 1760	geçerli, çerçeve 9	20.192 (program dışı, ama yakalanmaz)
12.500	6, 212	geçersiz	tuzak

Tablo 8.18: Geçerli/geçersiz biti ile çeviri (2 KiB sayfa). Python ile doğrulandı.



Şekil 8.40: 12.000 adresi programın dışındadır ama sayfa 5'in içinde kaldığı için donanım onu yakalamaz.

Üçüncü satır sayfalamanın bir sınırını gösterir. Sürecin son adresi 10.468'dir; 12.000 programa ait değildir. Ama sayfa 5, 10.240 ile 12.287 arasını kapsar ve geçerli işaretlidir. Donanım koruması sayfa ayrıntı düzeyindedir: bir sayfanın içindeki her adres ya geçerlidir ya da hiçbiri geçerli değildir. Sürecin son sayfasındaki kullanılmayan kısım, iç parçalanmanın bir başka yüzüdür. 12.500 ise sayfa 6'dadır ve geçersiz girdi bir tuzak üretir. İşletim sistemi bu tuzağın bir programlama hatası mı (`SIGSEGV`), yoksa henüz belleğe getirilmemiş bir sayfaya erişim mi olduğuna karar verir.

Bütün tabloyu tutmak yerine bir **sayfa tablosu uzunluk yazmacı** (*page-table length register*, PTLR) da kullanılabilir. Bu yazmaç tablonun kaç girdi içerdiğini tutar; daha büyük bir sayfa numarası doğrudan tuzak üretir. Böylece kullanılmayan sayfaların girdileri için yer harcanmaz.

8.7.9 Koruma bitleri

Sayfalamada koruma da sayfa düzeyinde yapılır. İzin bitleri her PTE'de ayrı tutulur ve her erişimde, TLB girdisiyle birlikte denetlenir.

Okuma / yazma (R/W)
: Kod sayfaları salt okunur eşlenir; koda yazma girişimi koruma hatası üretir.

Kullanıcı / denetçi (U/S)
: Çekirdek sayfalarına kullanıcı modunda erişilemez. Böylece aynı tabloda hem çekirdeğin hem sürecin eşlemeleri bulunabilir.

Çalıştırma yasağı (NX / XD)
: Veri, yığın ve heap sayfalarında kod çalıştırılamaz. İşletim sistemleri W^X kuralını uygular: bir sayfa ya yazılabilir ya çalıştırılabilirdir, ikisi birden olamaz. Böylece yığına yazılan saldırı kodu çalıştırılamaz.

SMEP / SMAP
: Çekirdek modundayken kullanıcı sayfalarındaki kodun çalıştırılmasını (SMEP) ve kullanıcı verisine izinsiz erişimi (SMAP) engeller.

Koruma anahtarları (PKU)

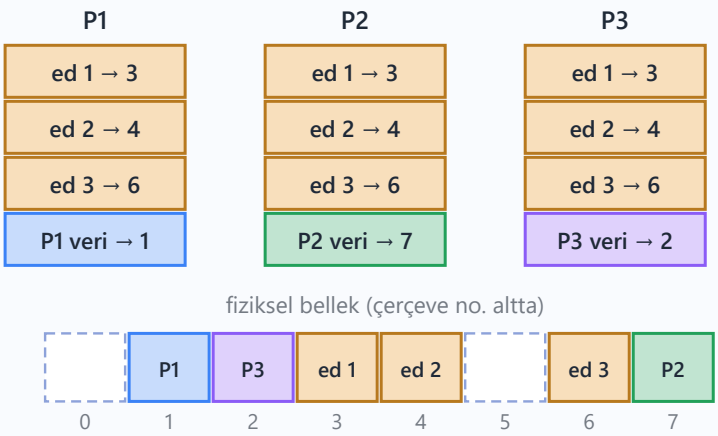
PTE'deki 4 bitlik anahtar ve bir kullanıcı yazmacı (PKRU) ile, sistem çağrısı yapmadan bir sayfa grubunun iznini değiştirmeyi sağlar.

Geçerli / geçersiz

Erişim izninin en kaba biçimidir: sayfa adres alanında hiç yoksa her erişim tuzaktır.

8.7.10 Paylaşımlı sayfalar

Sayfalamanın önemli bir kazancı, ortak kodun kolayca paylaşılabilmesidir. Yeniden girilir (*reentrant*), yani çalışırken kendini değiştirmeyen kod birçok süreç arasında paylaşılabilir. Üç kullanıcı aynı metin düzenleyicisini çalıştırırsa düzenleyicinin kod sayfaları bellekte yalnız bir kez bulunur. Her sürecin sayfa tablosu bu çerçeveleri gösterir; yalnız veri sayfaları her süreç için ayrıdır (Şekil 8.41).



Şekil 8.41: Üç süreç düzenleyici kodunun üç sayfasını aynı fiziksel çerçevelere (3, 4, 6) eşler; her sürecin verisi ayrı çerçevededir.

Paylaşımlı kütüphaneler de bu yolla paylaşılır. Paylaşımın bir sonucu vardır: sayfanın "sahibi" olan süreç bittiğinde çerçeve hemen boşaltılamaz, çünkü başka süreçler onu kullanıyor olabilir. Bu yüzden her paylaşılan çerçeve için bir referans sayısı tutulur.

Bugünkü sistemlerde paylaşımın başka biçimleri de vardır. `fork()` çağrısında ebeveyn ve çocuk bütün sayfaları **yazınca kopyala** (*copy-on-write*, COW) yöntemiyle paylaşır. Sayfalar iki süreçte de salt okunur işaretlenir; süreçlerden biri bir sayfaya yazmaya çalışınca koruma hatası olur ve çekirdek yalnız o sayfanın kopyasını çıkarır. Bellek eşlemeli dosyalar (`mmap`) ve paylaşımlı bellek (`shm_open`) ise iki sürecin aynı çerçeveleri görmesini sağlar; süreçler veri kopyalamadan iletişim kurar.

8.8 TLB

Sayfalama dış parçalanmayı ortadan kaldırır, ama bir bedeli vardır: her bellek erişimi önce bir tablo erişimi gerektirir. Bu kesim, bu bedelin nasıl neredeyse sıfıra indirildiğini anlatır.

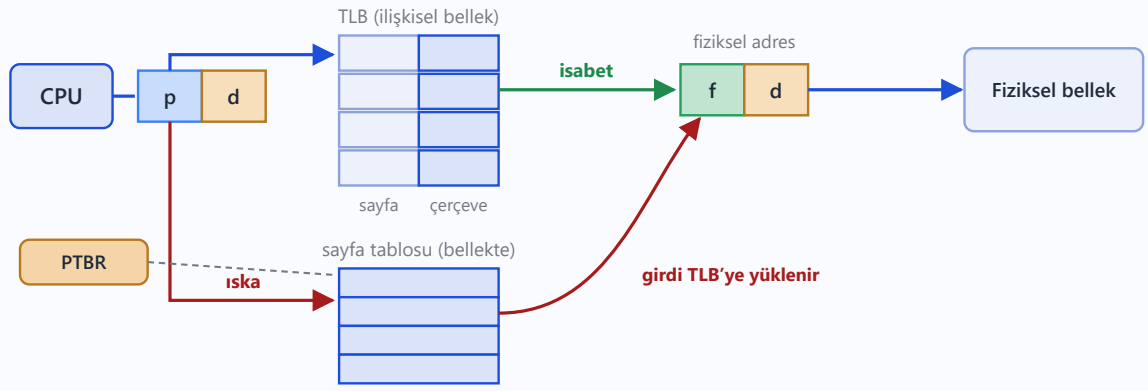
8.8.1 Sayfa tablosu nerede durur?

Sayfa tablosu küçük olsaydı işlemcinin içinde, özel yazmaçlarda tutulabilirdi. Ama 32 bit adres ve 4 KiB sayfada bir tablonun 2^{20}, yani bir milyondan fazla girdisi vardır. Bu kadar girdi yazmaçlara sığmaz. Tablo bu yüzden ana bellekte durur. Başlangıç adresini **sayfa tablosu taban yazmacı** (*page-table base register*, PTBR) tutar; bu yazmaç x86'da CR3, ARM'da TTBR0/TTBR1'dir. Bağlam değiştirmede yalnız bu yazmaç değişir.

Bu çözümün sorunu açıktır: her veri erişimi artık iki bellek erişimidir. Önce PTBR + p adresindeki girdi okunur, sonra veriye erişilir. Bellek erişim süresi iki katına çıkar. Çözüm, son kullanılan çevirileri tutan küçük bir donanım önbelleğidir.

> **Tanım 8.9**
>
> **TLB** (*translation look-aside buffer*), MMU içindeki küçük, hızlı ve ilişkisel bir önbellektir. Son kullanılan (sayfa numarası, çerçeve numarası) çiftlerini tutar. İlişkisel olması, bir sayfa numarasının bütün girdilerle aynı anda, paralel olarak karşılaştırılması demektir.

Çeviri şöyle yürür (Şekil 8.42). İşlemci bir adres ürettiğinde sayfa numarası önce TLB'de aranır. Bulunursa (*TLB isabeti*) çerçeve numarası hemen alınır ve tablo erişimi yapılmaz. Bulunmazsa (*TLB ıskası*) PTBR üzerinden bellekteki tabloya gidilir, girdi okunur ve bir sonraki erişim için TLB'ye yüklenir. TLB doluysa bir girdi çıkarılır.



Şekil 8.42: TLB ile sayfalama donanımı. Sayfa numarası önce TLB'de aranır; ıskada PTBR ile bellekteki tabloya gidilir ve girdi TLB'ye yüklenir.

Bugünkü işlemcilerde tipik olarak her çekirdekte 64 girdilik bir L1 TLB ve 1–3 bin girdilik bir L2 TLB bulunur. TLB girdileri bir adres alanı kimliğiyle (*address-space identifier*, ASID; Intel'de PCID) etiketlenebilir. Böylece her girdi hangi sürece ait olduğunu bilir ve bağlam değiştirmede TLB'nin boşaltılması gerekmez.

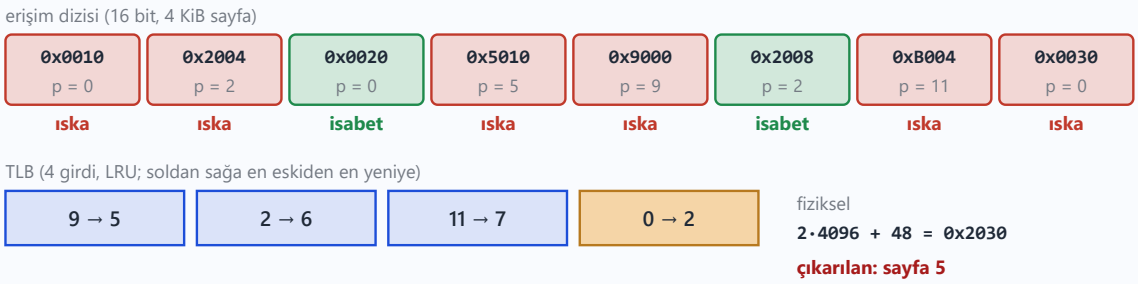
8.8.2 TLB isabet ve ıska izlemesi

Tanenbaum'un MMU örneğindeki sayfa tablosu (0 → 2, 2 → 6, 5 → 3, 9 → 5, 11 → 7) bu kez bir TLB ile kullanılsın. TLB 4 girdiliktir ve yer gerekince en uzun süredir kullanılmayan (LRU) girdiyi çıkarır. TLB başlangıçta boştur. Aşağıdaki sözde kod her erişimde yapılanları özetler.

```
p = adres / 4096;  d = adres % 4096;
if (p TLB'de) {           /* isabet */
  f = TLB[p];  p'yi en yeni yap;
} else {                  /* ıska */
  f = sayfa_tablosu[p];
  if (TLB dolu) en eski girdiyi çıkar;
  TLB'ye (p, f) ekle;
}
fiziksel = f * 4096 + d;
```

Erişim	p	Sonuç	Fiziksel	Çıkan	TLB sonra
0x0010	0	ıska	0x2010	—	0
0x2004	2	ıska	0x6004	—	0, 2
0x0020	0	isabet	0x2020	—	2, 0
0x5010	5	ıska	0x3010	—	2, 0, 5
0x9000	9	ıska	0x5000	—	2, 0, 5, 9
0x2008	2	isabet	0x6008	—	0, 5, 9, 2
0xB004	11	ıska	0x7004	0	5, 9, 2, 11
0x0030	0	ıska	0x2030	5	9, 2, 11, 0

Tablo 8.19: 4 girdilik LRU TLB izlemesi. TLB sütununda girdiler en eskiden en yeniye sıralıdır. Python ile doğrulandı.



Şekil 8.43: İzlemenin son adımı: 0x0030 için sayfa 0 TLB'de yoktur, en uzun süredir kullanılmayan sayfa 5 çıkarılır.

Sekiz erişimde yalnız 2 isabet vardır. İsabetlerde sayfa tablosuna hiç gidilmez. 0x0020 isabetinden sonra sayfa 0 “en yeni” konuma geçer. Ama 0xB004 geldiğinde TLB doludur ve en uzun süredir kullanılmayan girdi yine sayfa 0'dır, çünkü 0x0020'den sonra 5, 9 ve 2 kullanılmıştır. Hemen ardından

gelen 0x0030 bu yüzden ıskalar. Kısa örneklerde oran düşük çıkar; gerçek programlarda ise yerellik sayesinde TLB isabet oranı %99'u aşar.

Farklı adreslerle TLB isabet ve ıskalarını, sayfa hatalarını denemek için Sayfalama ve Adres Çevirisi Simülatörü kullanılabilir.

8.8.3 Etkin erişim süresi

TLB'nin kazancı tek bir sayıyla ölçülür.

Tanım 8.10

Etkin erişim süresi (*effective access time*, EES), TLB isabet ve ıskalarının olasılıkları hesaba katılarak bulunan ortalama bellek erişim süresidir. TLB arama süresi ε, bellek erişim süresi t, TLB isabet oranı α ise tek düzeyli tabloda:

> $\text{EES} = (t + \varepsilon) \cdot \alpha + (2t + \varepsilon) \cdot (1 - \alpha) = 2t + \varepsilon - t \cdot \alpha$

Formülün mantığı basittir. İsabette TLB'ye bakılır ve tek bir bellek erişimi yapılır: $t + \varepsilon$. Iskada TLB'ye bakılır, sonra tablo girdisi için bir, veri için bir bellek erişimi yapılır: $2t + \varepsilon$. Ortalama, bu iki sürenin olasılıklarla ağırlıklı toplamıdır.

Örnek 8.11

$\varepsilon = 20$ ns, $t = 100$ ns olsun. $\alpha = \%80$ ve $\alpha = \%98$ için etkin erişim süresi nedir?

1. **İki durum.** İsabet: 20 + 100 = 120 ns. Iska: 20 + 100 + 100 = 220 ns.
2. **α = %80.** 120 × 0,80 + 220 × 0,20 = 96 + 44 = 140 ns. Kapalı biçimle de 2 × 100 + 20 − 100 × 0,80 = 140 ns.
3. **Yorum.** TLB olmasaydı her erişim 200 ns sürerdi; 140 ns bundan iyidir, ama yalın bellek süresinden hâlâ %40 yavaştır.
4. **α = %98.** 120 × 0,98 + 220 × 0,02 = 117,6 + 4,4 = 122 ns. Yavaşlama %22'ye iner. Gerçek iş yüklerinde oran genellikle %99'un üzerindedir; α = %99'da EES 121 ns'dir.

Çok düzeyli tablolarda ıska çok daha pahalıdır. x86-64'ün dört düzeyli tablosunda bir ıska dört tablo erişimi ve bir veri erişimi gerektirir: 20 + 4 × 100 + 100 = 520 ns. Bu durumda α = %98 için EES = 120 × 0,98 + 520 × 0,02 = 117,6 + 10,4 = 128 ns, α = %99 için 118,8 + 5,2 = 124 ns olur. Iska ne kadar pahalıysa isabet oranı o kadar önemlidir. Bu yüzden donanım tablo yürüyüşünün ara düzeylerini de ayrı bir önbellekte tutar (*page walk cache*). Gerçek bir ıska böylece çoğu zaman dört değil, bir ya da iki bellek erişimine iner.

Örnek 8.12

Bellek erişimi 80 ns, TLB araması 10 ns, tablo tek düzeyli, isabet oranı %95'tir. (a) EES nedir? (b) EES'nin 92 ns'ye inmesi için isabet oranı en az kaç olmalıdır?

1. **İki durum.** İsabet 10 + 80 = 90 ns, ıska 10 + 80 + 80 = 170 ns.
2. **(a)** 0,95 × 90 + 0,05 × 170 = 85,5 + 8,5 = 94 ns.
3. **(b)** $90\alpha + 170(1-\alpha) \le 92 \rightarrow 170 - 80\alpha \le 92 \rightarrow \alpha \ge 78 / 80 = 0{,}975$. İsabet oranı en az %97,5 olmalıdır.

TLB'nin etkisini bir döngü üzerinde saymak da öğreticidir.

Örnek 8.13

`for (i = 0; i < 1000; i++) a[i] = 0;` döngüsü aşağıdaki dört komuta derlenir. Dizi a sanal 40.000 adresinde başlar, `int` 4 bayttır, sayfa 4 KiB'tır ve kod tek bir sayfadadır. Doğrusal sayfa tablosu var. TLB yokken ve varken toplam kaç bellek erişimi olur?

```
dongu: movl $0,(%edi,%eax,4)  /* a[i] = 0 */
       incl %eax              /* i++      */
       cmpl $1000,%eax
       jne  dongu
```

1. **Bir tur.** Dört komut getirilir ve `movl` belleğe bir kez yazar: 5 erişim.
2. **TLB yok.** Her erişimden önce bir PTE okunur: tur başına 10, toplam 10 × 1000 = 10.000 erişim.
3. **TLB var.** 5 × 1000 = 5000 erişime yalnız TLB ıskalarındaki PTE okumaları eklenir. Dizi 40.000 ile 43.999 arasındadır ve $\lfloor 40.000 / 4096 \rfloor = 9$ ile $\lfloor 43.999 / 4096 \rfloor = 10$ numaralı sayfalara yayılır. Kod sayfasıyla birlikte üç ıska olur: 5000 + 3 = 5003 erişim.

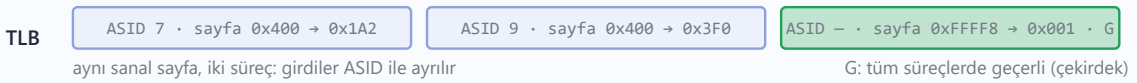
8.8.4 TLB yönetimi

TLB'nin nasıl doldurulduğu ve temizlendiği mimariden mimariye değişir. Tablo 8.20 başlıca konuları özetler.

Konu	Açıklama
Donanım yönetimli	x86-64, ARMv8, RISC-V: ıskada MMU sayfa tablosunu kendisi yürür (*page walk*); işletim sistemi yalnız tabloyu kurar.
Yazılım yönetimli	MIPS, SPARC, Alpha gibi eski RISC işlemcilerde ıska işletim sistemine tuzak olur; çekirdek girdiyi bulup TLB'ye yazar. Bu bütün RISC'ler için geçerli değildir: ARMv8 ve RISC-V donanım yönetimlidir.

Konu	Açıklama
Yumuşak / sert ıska	Yumuşak ıska: sayfa bellekte, yalnız TLB'de yok (nanosaniyeler). Sert ıska: sayfa diskte, G/Ç gerekir (mikro- ya da milisaniyeler).
ASID / PCID	Girdi süreç kimliğiyle etiketlenir; bağlam değiştirmede TLB temizlenmez. Meltdown (2018) sonrası getirilen KPTI'nin maliyetini PCID azaltır.
Global girdiler	Çekirdek eşlemeleri G biti ile işaretlenir ve bütün süreçlerde geçerli kalır.
TLB vurusu	Çok çekirdekte bir eşleme değişince diğer çekirdeklerin TLB'leri işlemciler arası kesmelerle (IPI) geçersizlenir; yeni işlemcilerde yayın komutları vardır (AMD INVLPGB).

Tablo 8.20: TLB yönetimi.



Şekil 8.44: ASID etiketli TLB: iki sürecin aynı sanal sayfası (0x400) farklı çerçevelere eşlenir; G işaretli çekirdek girdisi bütün süreçlerde geçerlidir.

ASID'nin değeri bağlam değiştirmede ortaya çıkar. Etiket yoksa, yeni süreç eski sürecin çevirilerini kullanmasın diye bütün TLB boşaltılmalıdır; yeni süreç de ilk erişimlerinde art arda ıskalar. ASID ile iki sürecin aynı sanal sayfası TLB'de yan yana durabilir (Şekil 8.44). *TLB vurusu* (*TLB shootdown*) ise çok çekirdekli sistemlerin sorunudur: bir çekirdek bir eşlemeyi değiştirdiğinde, aynı eşlemeyi TLB'sinde tutan öteki çekirdeklere haber vermek zorundadır. Bu iletişim pahalıdır ve büyük sistemlerde ölçeklenmeyi sınırlar.

8.9 Sayfa tablosu yapıları

Tek düzeyli, doğrusal bir sayfa tablosu küçük adres alanlarında işe yarar. Adres alanı büyüdükçe tablonun kendisi bir bellek sorununa dönüşür. Bu kesim, tabloyu küçültmenin üç yolunu anlatır: hiyerarşik, hash ve ters tablolar.

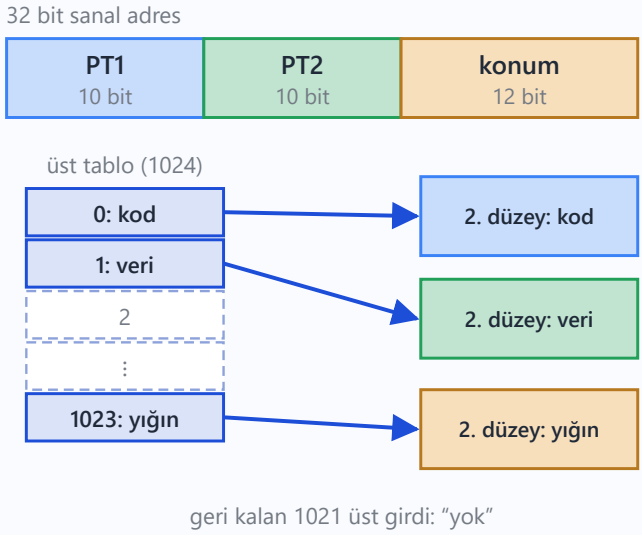
8.9.1 Hiyerarşik sayfa tablosu

32 bit adres ve 4 KiB sayfada tek düzeyli tablonun 2^{20} girdisi vardır; girdi 4 bayt ise tablo 4 MiB tutar. Bu tablo her süreç için ayrı ve bellekte bitişik olmalıdır. 100 süreçlik bir sistemde yalnız sayfa tabloları 400 MiB eder. 64 bitte durum çok daha kötüdür. 4 KiB sayfa ve 8 baytlık girdiyle tek düzeyli tablo $2^{52} \times 8$ B = 32 PiB, yani yaklaşık 33,6 milyon GiB tutar. Adresin yalnız 48 biti kullanılsa bile $2^{36} \times 8$ B = 512 GiB gerekir.

Çözüm, tablonun kendisini sayfalamaktır. Sanal adres iki indise ve bir konuma bölünür. 32 bit x86'da 10 bit üst tabloyu (sayfa dizini), 10 bit ikinci düzey tabloyu indisler, 12 bit konumdur (Şekil 8.45). Burada bölünen şey tablo girdisi değil, sanal adrestir.

Tanım 8.11

Hiyerarşik sayfa tablosu (*hierarchical page table*), sayfa tablosunun çok düzeyli bir ağaç olarak tutulmasıdır. Üst tablonun her girdisi bir alt düzey tabloyu gösterir; yalnız kullanılan bölgelerin alt tabloları vardır.



Şekil 8.45: İki düzeyli sayfa tablosu. Üst tablonun yalnız kod, veri ve yığın girdileri ikinci düzey tablolara işaret eder; kalan 1021 üst girdi "yok"tur.

Kazanç seyrek adres alanlarından gelir. Bir sürecin 4 MiB kod, 4 MiB veri ve 4 MiB yığın kullandığını düşünelim. Her ikinci düzey tablo 1024 sayfa × 4 KiB = 4 MiB'lık bir bölgeyi kapsar. Bu süreç için bir üst tablo ve üç alt tablo yeter: 4 × 4 KiB = 16 KiB. Tek düzeyli tablo 4 MiB tutardı. Kullanılmayan bölgelerin alt tabloları hiç oluşturulmaz; üst tablodaki girdileri yalnız "yok" olarak işaretlenir.

8.9.2 İki düzeyli çeviri örneği

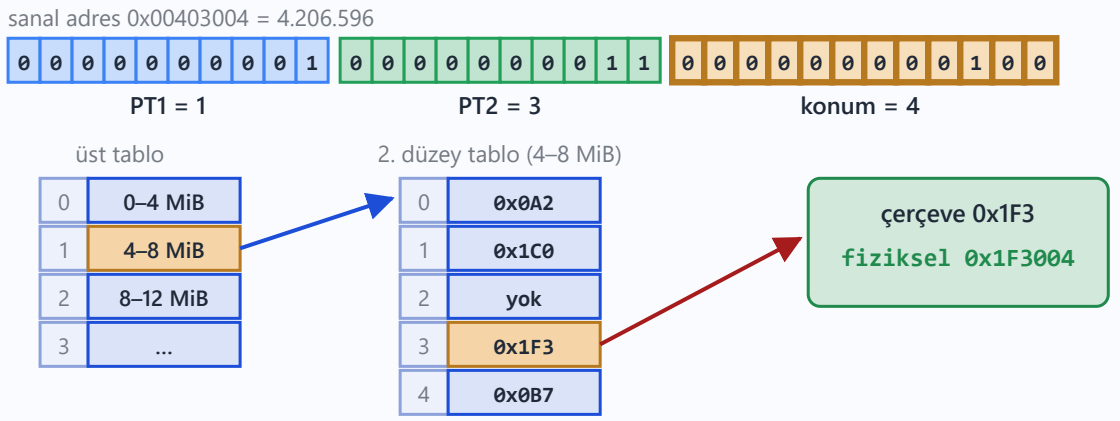
Örnek 8.14

Tanenbaum'un örneğinde sanal adres 0x00403004'tür (4.206.596). Bu adres 4 MiB ile 8 MiB arasında, veri bölgesindedir. İki düzeyli tabloda nasıl çevrilir?

1. **Bölme.** 0x00403004 = 0000000001 0000000011 000000000100. Üst 10 bit PT1 = 1, orta 10 bit PT2 = 3, konum = 4.
2. **Üst tablo.** Üst tablonun 1. girdisi, 4–8 MiB bölgesinin ikinci düzey tablosunu gösterir. Her üst girdi 4 MiB'lık bir bölgeyi kapsar.

3. **İkinci düzey.** O tablonun 3. girdisine bakılır. Bu girdi bölge içinde 12.288–16.383 bayt aralığındaki sayfadır; mutlak olarak 4.206.592–4.210.687. Girdinin mevcut biti 0 olsaydı sayfa hatası oluşurdu.
4. **Fiziksel adres.** Girdi çerçeve numarasını verir (örnekte 0x1F3). Konum eklenir: 0x1F3004.

TLB ıskasında bu çeviri iki tablo erişimi ve bir veri erişimi gerektirir. Örnekteki çerçeve numaraları açıklama amaçlıdır.



Şekil 8.46: 0x00403004 adresinin çevrilmesi: PT1 = 1, PT2 = 3, konum = 4; çerçeve 0x1F3, fiziksel 0x1F3004.

32 bit x86'da üst tablonun adresi CR3'tedir. İki düzeydeki tabloların her biri tam bir 4 KiB'lık sayfayı doldurur: 1024 girdi × 4 bayt. Bu seçim tesadüf değildir. Tablonun bir sayfaya tam sığması, alt tabloların da sıradan sayfalar gibi herhangi bir çerçeveye konmasını, gerekirse diske takas edilmesini sağlar.

8.9.3 Kod izleme: iki düzeyli çeviri

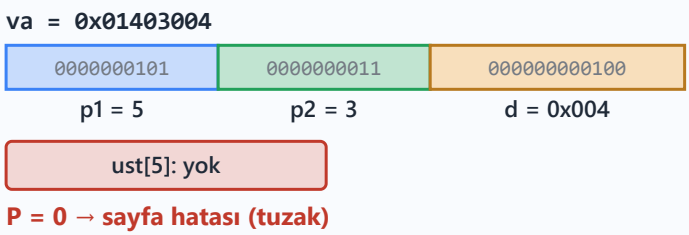
MMU'nun yaptığı iş C ile şöyle yazılabilir:

```
/* 32 bit: 10 + 10 + 12 bit */
uint32_t cevir(uint32_t va) {
  uint32_t p1 = va >> 22;
  uint32_t p2 = (va >> 12) & 0x3FF;
  uint32_t d  = va & 0xFFF;
  pte_t e1 = ust[p1];
  if (!e1.p) sayfa_hatasi(va);
  pte_t e2 = e1.tablo[p2];
  if (!e2.p) sayfa_hatasi(va);
  return (e2.cerceve << 12) | d;
}
```

`va >> 22` üst 10 biti alır. `(va >> 12) & 0x3FF` önce konumu atar, sonra alt 10 biti, yani orta indisi bırakır. `va & 0xFFF` konumdur. Fonksiyon üst tablonun girdisine, oradan ikinci düzey tablonun girdisine gider; iki adımda da mevcut biti 0 ise sayfa hatası üretir. Sonunda çerçeve numarasını 12 bit sola kaydırıp konumla birleştirir.

Değişken	va = 0x00801A3C	va = 0x01403004
`p1 = va >> 22`	2	5
`p2 = (va >> 12) & 0x3FF`	0x801 & 0x3FF = 1	0x1403 & 0x3FF = 3
`d = va & 0xFFF`	0xA3C	0x004
`e1 = ust[p1]`	P = 1 (1. bellek erişimi)	P = 0 → sayfa hatası
`e2 = e1.tablo[p2]`	çerçeve 0x2F, P = 1 (2. erişim)	—
dönüş	(0x2F << 12) | 0xA3C = 0x0002FA3C	—

Tablo 8.21: `cevir` fonksiyonunun iki adresle izlenmesi. Üst tabloda yalnız girdi 2 geçerli; onun tablosunda girdi 1 → çerçeve 0x2F. Python ile doğrulandı.



Şekil 8.47: 0x01403004 için üst tablonun 5. girdisi geçersizdir: 20–24 MiB bölgesi için ikinci düzey tablo hiç yoktur.

İlk adreste üçüncü bellek erişimi veriye yapılır. İkinci adres 20–24 MiB bölgesindedir. Bu bölge için ikinci düzey tablo hiç oluşturulmamıştır; MMU üst tabloda tuzak üretir. Tablo için bellek harcanmamış olması, hiyerarşik tablonun bütün kazancıdır.

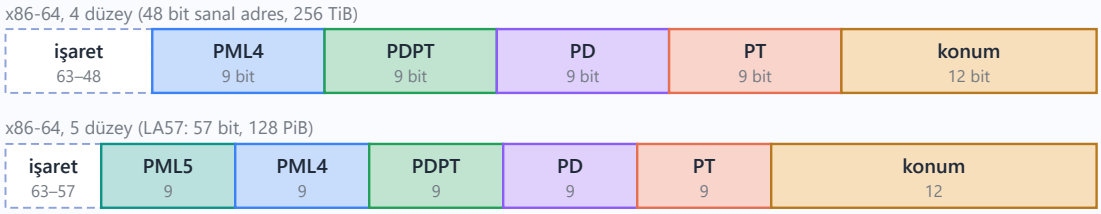
Örnek 8.15

Aynı 10 + 10 + 12 bölünmesinde bir süreç adres alanının en altındaki 12 MiB'ı ve en üstteki 4 MiB'ı kullanıyor. Kaç sayfa tablosu gerekir?

1. **Alt tablolar.** Her alt tablo 4 MiB kapsar: alttaki 12 MiB için 3, üstteki 4 MiB için 1 tablo.
2. **Toplam.** 4 alt tablo + 1 üst tablo = 5 tablo = 20 KiB. Doğrusal tablo 4 MiB olurdu.

8.9.4 x86-64: dört ve beş düzeyli sayfalama

x86-64'te her sayfa tablosu 512 girdi × 8 bayt = 4 KiB'tır; yani yine tam bir sayfadır ve her düzey 9 bitlik bir indis kullanır. Dört düzey ve 12 bitlik konumla 4 × 9 + 12 = 48 bitlik sanal adres elde edilir; bu 256 TiB'lık bir adres alanıdır (Şekil 8.48). Düzeylerin adları üstten alta PML4, PDPT, PD ve PT'dir.



Şekil 8.48: x86-64 sanal adresinin dört ve beş düzeyli sayfalamada bölünmesi: her düzey 9 bit, konum 12 bit.

64 bitlik adresin üst 16 biti kullanılmaz ama rastgele de olamaz: hepsi bit 47'nin kopyası olmalıdır. Böyle adreslere *kanonik adres* denir. Sonuçta adres alanı ikiye ayrılır: alt yarı kullanıcıya, üst yarı çekirdeğe verilir. Beş düzeyli sayfalama (LA57) bir PML5 düzeyi ekler ve 57 bitlik, 128 PiB'lık sanal adres verir. Intel Ice Lake sunucu işlemcileri ve AMD Zen 4 ile sonrası bunu destekler; Linux 4.14'ten beri kullanabilir.

Hiyerarşik tablonun 64 bitte de işe yaramasının nedeni budur: 64 bitin hepsi kullanılmaz ve kullanılan adres alanı da seyrektir. Bugünkü x86-64 ve ARMv8 tam olarak hiyerarşik tablo kullanır. Bir orta düzey girdisinde PS biti 1 ise yürüyüş orada biter: PD düzeyinde 2 MiB'lık, PDPT düzeyinde 1 GiB'lık büyük bir sayfa bulunmuş olur.

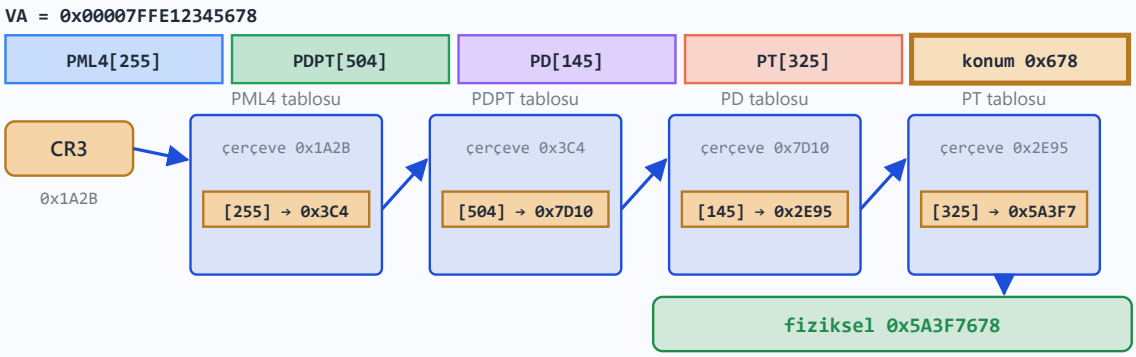
8.9.5 Dört düzeyli yürüyüş

Örnek 8.16

Kullanıcı yığınına yakın bir adres olan VA = 0x00007FFE12345678 dört düzeyde nasıl çevrilir?

1. **Bölme (9 + 9 + 9 + 9 + 12).** PML4 = (VA >> 39) & 0x1FF = 255, PDPT = (VA >> 30) & 0x1FF = 504, PD = (VA >> 21) & 0x1FF = 145, PT = (VA >> 12) & 0x1FF = 325, konum = 0x678.
2. **PML4.** CR3, PML4 tablosunun çerçevesini verir (0x1A2B). Girdi 255 okunur; PDPT tablosunun çerçevesi 0x3C4'tür.
3. **PDPT.** PDPT[504] sayfa dizininin (PD) çerçevesini verir: 0x7D10. Burada PS = 1 olsaydı 1 GiB'lık bir sayfa bulunmuş olurdu.
4. **PD.** PD[145] son düzey tablonun çerçevesini verir: 0x2E95. PS = 1 olsaydı 2 MiB'lık bir sayfa bulunurdu.
5. **PT.** PT[325] çerçeve 0x5A3F7'yi verir; konum eklenir: fiziksel 0x5A3F7678.

TLB ıskasında 4 tablo erişimi ve 1 veri erişimi yapılır. Çerçeve değerleri örnektir; indisler Python ile hesaplandı.



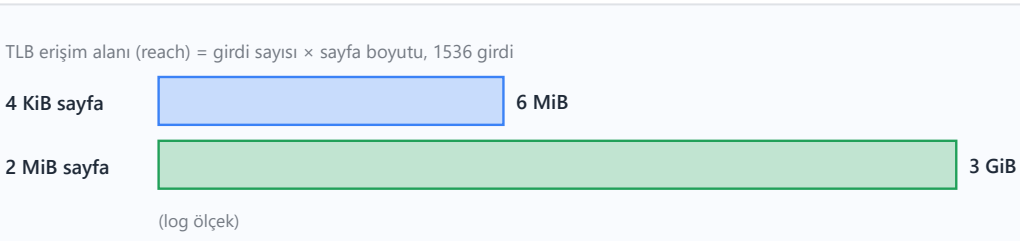
Şekil 8.49: x86-64 dört düzeyli sayfa tablosu yürüyüşünün sonu: PT[325] çerçeve 0x5A3F7'yi verir.

Sık yapılan hata

x86-64'te de 32 bit x86'daki gibi 10 bitlik indis kullanmak. Girdi 8 bayta çıktığı için 4 KiB'lık bir tabloya 512 girdi sığar; indis 9 bittir. İndisler 0x1FF ile maskelenir, 0x3FF ile değil.

8.9.6 Büyük sayfalar

Büyük çalışma kümesiyle çalışan programlar, örneğin veritabanları, sanal makineler ve yapay zekâ iş yükleri, 4 KiB'lık sayfalarla sürekli TLB ıskası yaşar. Sorunun ölçüsü **TLB erişim alanıdır** (*TLB reach*): girdi sayısı × sayfa boyu. 1536 girdilik bir TLB, 4 KiB'lık sayfalarla yalnız 6 MiB'ı kapsar; 2 MiB'lık sayfalarla aynı TLB 3 GiB'ı kapsar (Şekil 8.50).



Şekil 8.50: TLB erişim alanı: 1536 girdi ile 4 KiB sayfada 6 MiB, 2 MiB sayfada 3 GiB.

Tablo boyutu da küçülür. 1 GiB'ı 4 KiB'lık sayfalarla eşlemek $2^{30} / 2^{12} = 262.144$ PTE, yani 2 MiB'lık tablo ister. 2 MiB'lık sayfalarla yalnız 512 PD girdisi, yani 4 KiB yeter.

Linux büyük sayfaları üç biçimde sunar. `hugetlbfs`, önceden ayrılmış 2 MiB ya da 1 GiB'lık sayfaları açıkça isteyen programlar içindir. Şeffaf büyük sayfalarda (*transparent huge pages*, THP) çekirdek uygun bölgeleri kendiliğinden büyük sayfalarla eşler. Son sürümlerdeki mTHP ise 16–64 KiB'lık ara

boyutlar kullanır. Windows'ta aynı işi "large pages" görür.

Büyük sayfaların bedeli iki tanedir: iç parçalanma artar ve 2 MiB'lık bitişik bir fiziksel bölge bulmak gerekir. Bitişik bölge bulma sorunu, buddy ayırıcının ve sıkıştırmanın bugün hâlâ neden önemli olduğunu açıklar. ARMv8 bu soruna ara bir çözüm getirir: *contiguous* biti işaretli 16 komşu 4 KiB'lık girdi tek bir TLB girdisiyle, 64 KiB olarak tutulabilir.

8.9.7 Hash sayfa tablosu

32 bitten büyük adres alanları için bir başka yaklaşım hash tablosudur.

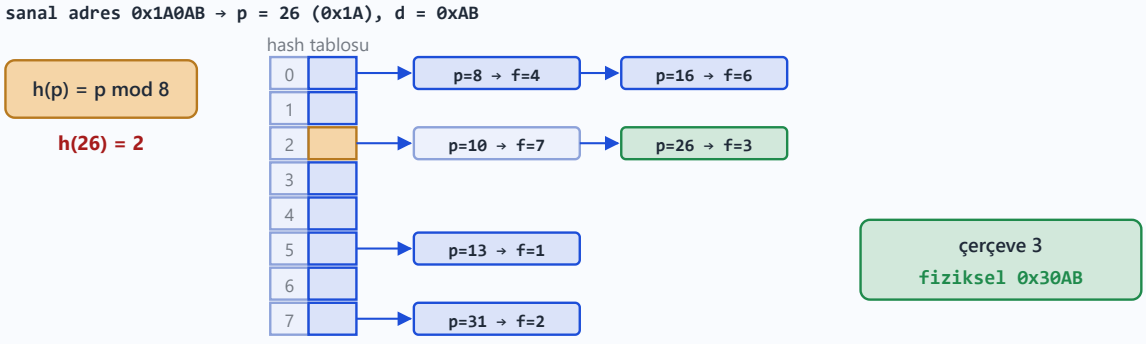
Tanım 8.12

Hash sayfa tablosunda (*hashed page table*) sanal sayfa numarası bir hash fonksiyonundan geçirilir. Tablonun her kovası, aynı hash değerine düşen (sanal sayfa, çerçeve, sonraki) girdilerinin bağlı listesidir.

Örnek 8.17

8 kovalı bir tabloda h(p) = p mod 8'dir. Kova 2'nin zinciri (p = 10 → f = 7), (p = 26 → f = 3)'tür. Sanal adres 0x1A0AB nereye çevrilir? (4 KiB sayfa)

1. **Bölme.** p = 0x1A = 26, d = 0x0AB.
2. **Hash.** h(26) = 26 mod 8 = 2.
3. **Zincir.** İlk girdi p = 10: eşleşmez, sonrakine geçilir. İkinci girdi p = 26: eşleşir, çerçeve 3.
4. **Fiziksel adres.** 3 × 4096 + 0xAB = 0x30AB.



Şekil 8.51: Hash sayfa tablosunda 26 numaralı sayfanın aranması: kova 2'nin zincirinde ikinci girdi eşleşir.

Hash tablosunun boyutu kullanılan sayfa sayısıyla orantılıdır, adres alanının büyüklüğüyle değil. Bu, 64 bitlik seyrek adres alanları için çekici bir özelliktir. *Kümelenmiş* türünde her girdi tek bir sayfa yerine birden çok (örneğin 16) ardışık sayfanın çerçevelerini tutar. Bu tür, seyrek ama yer yer yoğun

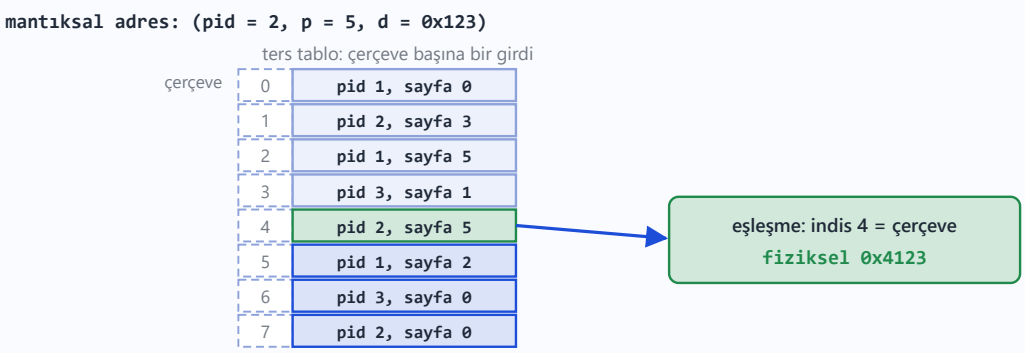
adres alanlarına uygundur.

8.9.8 Ters sayfa tablosu

Şimdiye kadarki bütün tablolarda süreç başına bir tablo vardı ve tablonun boyu sanal adres alanına bağlıydı. Ters tablo bakış açısını tersine çevirir.

> **Tanım 8.13**
>
> **Ters sayfa tablosu** (*inverted page table*), fiziksel çerçeve başına bir girdi tutan, sistemde tek olan tablodur. Her girdi o çerçevede hangi sürecin hangi sayfasının bulunduğunu, yani (pid, sayfa) çiftini tutar. Girdinin indisi çerçeve numarasıdır.

Çeviri bir arama işidir. (pid = 2, p = 5, d = 0x123) adresi için tablo baştan taranır. Girdi 0 (1, 0), girdi 1 (2, 3), girdi 2 (1, 5), girdi 3 (3, 1) eşleşmez. Girdi 4 (2, 5) eşleşir. İndisin kendisi çerçeve numarası olduğu için fiziksel adres 4 × 4096 + 0x123 = 0x4123'tür (Şekil 8.52).



Şekil 8.52: Ters sayfa tablosunda (pid 2, sayfa 5) aranması: indis 4'te eşleşme bulunur ve fiziksel adres 0x4123'tür.

Tablonun boyu fiziksel bellekle orantılıdır ve süreç sayısından bağımsızdır. 16 GiB RAM ve 4 KiB'lık çerçevelerle 4.194.304 girdi vardır; girdi başına 8 bayt ile tablo 32 MiB tutar. Bu bütün sistem için tek bir tablodur.

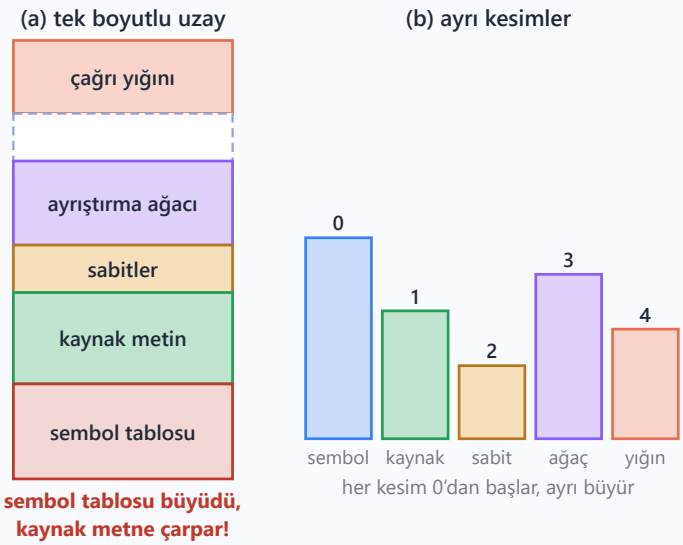
Bedeli üç tanedir. Birincisi aramadır: doğrusal tarama her erişimde kabul edilemeyecek kadar yavaştır. Gerçek sistemler bu yüzden (pid, p) çiftini hash'leyip doğrudan ilgili girdiye gider ve önüne bir TLB koyar. İkincisi paylaşımdır: bir çerçeve tek bir (pid, sayfa) çifti tutabildiği için paylaşımlı bellek zorlaşır. Üçüncüsü sayfa hatalarıdır: bellekte olmayan bir sayfanın diskteki yeri ters tabloda bulunmaz; bunun için süreç başına ayrı, dış bir tablo gerekir. PowerPC ve IBM POWER mimarileri hash'li ters tablo kullanır; IA-64 de bu yaklaşımı benimsemişti.

8.10 Kesimleme

Sayfalama, programcıdan tamamen gizli bir mekanizmadır: adres alanı tek boyutlu bir bayt dizisidir ve sayfa sınırları programın mantıksal yapısıyla hiç ilgili değildir. Kesimleme ise adres alanını programın kendi parçalarına göre böler.

8.10.1 Neden kesimleme?

Tanenbaum'un örneği bir derleyicidir. Derleyici çalışırken birçok tablo kurar: kaynak metin, sembol tablosu, sabitler tablosu, ayrıştırma ağacı ve çağrı yığını. Bu tabloların hepsi çalışma boyunca büyüyüp küçülür. Tek boyutlu bir adres alanında her tabloya ardışık bir bölge ayrılır. Bir tablo dolunca, komşusunda boş yer olsa bile ona çarpar (Şekil 8.53). Sembol tablosu kaynak metnin bölgesine taşar; derleyici ya durur ya da tabloları elle kaydırmak zorunda kalır.



Şekil 8.53: (a) Tek boyutlu adres alanında derleyici tabloları birbirine çarpar. (b) Ayrı kesimlerde her tablo 0'dan başlar ve bağımsız büyür.

Tanım 8.14

Kesimleme (*segmentation*), sürece birbirinden bağımsız, değişken uzunluklu birden çok adres alanı vermektir. Her adres alanına **kesim** (*segment*) denir. Bir adres iki parçalıdır: (kesim numarası, kesim içi konum).

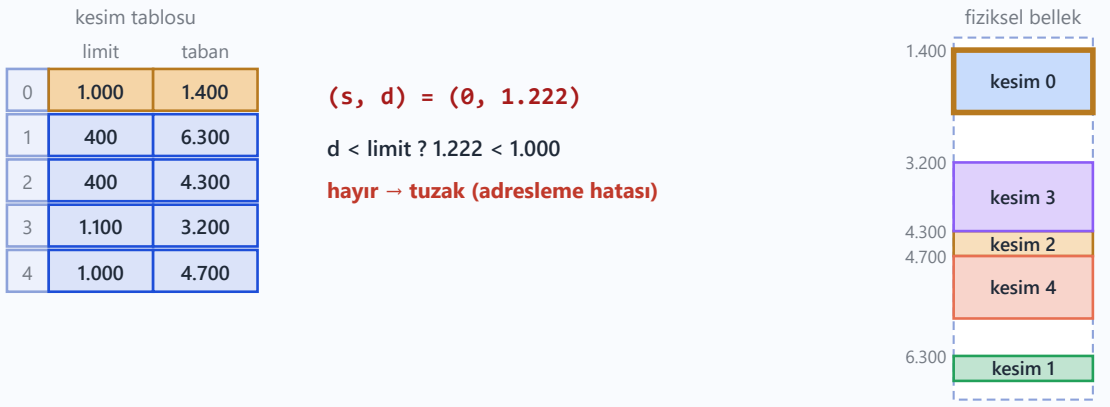
Kesimleme programcının bakışına uyar. Programcı programını tek bir bayt dizisi olarak değil; ana program, fonksiyonlar, yığın, diziler ve sembol tablosu gibi ayrı birimler olarak düşünür. Kesimleme bu birimlerin her birine kendi adres alanını verir. Her kesim 0'dan başlar ve öteki kesimleri etkilemeden büyüyüp küçülebilir.

8.10.2 Kesim tablosuyla çeviri

Çeviri bir **kesim tablosu** ile yapılır. Tablonun her girdisi bir kesimin tabanını (fiziksel başlangıç adresi) ve limitini (uzunluğunu) tutar. Tablonun bellekteki yerini kesim tablosu taban yazmacı (STBR), girdi sayısını kesim tablosu uzunluk yazmacı (STLR) tutar. (s, d) adresi için donanım önce d < limit[s] koşuluna bakar. Koşul sağlanırsa fiziksel adres taban[s] + d'dir; sağlanmazsa tuzak oluşur. Kesimleme aslında her kesim için ayrı bir taban-limit çiftidir.

(s, d)	d < limit?	Sonuç
(2, 53)	53 < 400	4.300 + 53 = 4.353
(3, 852)	852 < 1.100	3.200 + 852 = 4.052
(1, 399)	399 < 400	6.300 + 399 = 6.699
(0, 1.222)	1.222 ≥ 1.000	tuzak

Tablo 8.22: Silberschatz'ın beş kesimli örneği. Kesim tablosu (limit, taban): 0 → (1000, 1400), 1 → (400, 6300), 2 → (400, 4300), 3 → (1100, 3200), 4 → (1000, 4700). Python ile doğrulandı.



Şekil 8.54: (0, 1.222) adresi: kesim 0'ın boyu yalnız 1.000 bayttır, donanım tuzak üretir.

Son satırda denetim olmasaydı adres 1.400 + 1.222 = 2.622 olurdu. Bu adres kesim 0'ın dışındadır; sürecin başka bir kesimine ya da başka bir sürecin belleğine ait olabilir. Limit denetimi tam olarak bunu engeller.

Sık yapılan hata

Konum limite eşitken erişimi geçerli saymak. Geçerli konumlar 0 … limit − 1'dir. Limiti 400 olan bir kesimde (1, 399) son geçerli baytı gösterir; (1, 400) artık tuzaktır.

8.10.3 Kesimi seçen bitler, aşağı büyüyen yığın

Kesim numarası adresin ayrı bir parçası olmak zorunda değildir; sanal adresin üst bitleri de kesimi seçebilir. OSTEP'tekine benzer bir örnekte sanal adres 14 bittir (16 KiB). Üst iki bit kesimi seçer: 00 kod, 01 heap, 11 yığın. Kalan 12 bit konumdur, dolayısıyla bir kesim en çok 4 KiB olabilir. Kesim yazmaçları şöyledir:

Kesim	Taban	Boy	Yön	Koruma
00 kod	24.576	3.072	+	`r-x`
01 heap	40.960	2.048	+	`rw-`
11 yığın	32.768	2.048	−	`rw-`

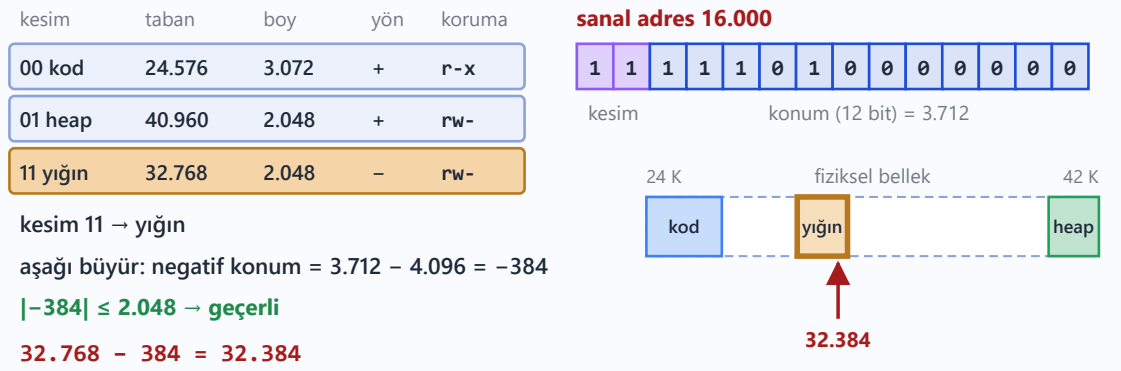
Tablo 8.23: Kesim yazmaçları (14 bit sanal adres).

Yığın kesiminin "yön" biti eksidir: yığın aşağı doğru büyür. Böyle bir kesimde konum negatif yorumlanır. Konumdan en büyük kesim boyu (4.096) çıkarılır; elde edilen negatif sayının mutlak değeri boydan büyük olmamalıdır. Fiziksel adres taban + negatif konumdur.

Örnek 8.18

Sanal 1.000, 4.500, 16.000 ve 7.000 adresleri nereye gider?

1. **1.000.** İkili 00 001111101000: kesim 00 (kod), konum 1.000. 1.000 < 3.072, geçerli. Fiziksel 24.576 + 1.000 = 25.576.
2. **4.500.** 01 000110010100: kesim 01 (heap), konum 4.500 − 4.096 = 404. 404 < 2.048, geçerli. Fiziksel 40.960 + 404 = 41.364. Sanal adresin tamamını tabana eklemek (40.960 + 4.500 = 45.460) yanlış olurdu.
3. **16.000.** 11 111010000000: kesim 11 (yığın), konum 3.712. Yığın aşağı büyüdüğü için 3.712 − 4.096 = −384. |−384| ≤ 2.048, geçerli. Fiziksel 32.768 − 384 = 32.384.
4. **7.000.** 01 101101011000: kesim 01 (heap), konum 2.904. 2.904 ≥ 2.048: donanım tuzak üretir, işletim sistemi süreci `SIGSEGV` ile sonlandırır.



Şekil 8.55: 16.000 adresi: üst bitler 11 yığını seçer; yığın aşağı büyüdüğü için konum −384 olarak yorumlanır ve fiziksel adres 32.384'tür.

Örnek 8.19

Aynı yazmaçlarla sanal 13.000 ve 14.500 adresleri nereye gider?

1. **13.000.** 13.000 − 12.288 = 712; kesim 11 (yığın), konum 712. 712 − 4.096 = −3.384; |−3.384| > 2.048 olduğu için tuzak.
2. **14.500.** Kesim 11, konum 2.212. 2.212 − 4.096 = −1.884; |−1.884| ≤ 2.048, geçerli. Fiziksel 32.768 − 1.884 = 30.884.

Koruma bitleri paylaşımı da mümkün kılar. `r-x` izinli bir kod kesimi birçok süreçte aynı fiziksel bölgeye eşlenebilir. Kesim salt okunur olduğu için süreçlerden biri ona yazmaya çalışırsa koruma hatası oluşur ve ötekiler etkilenmez.

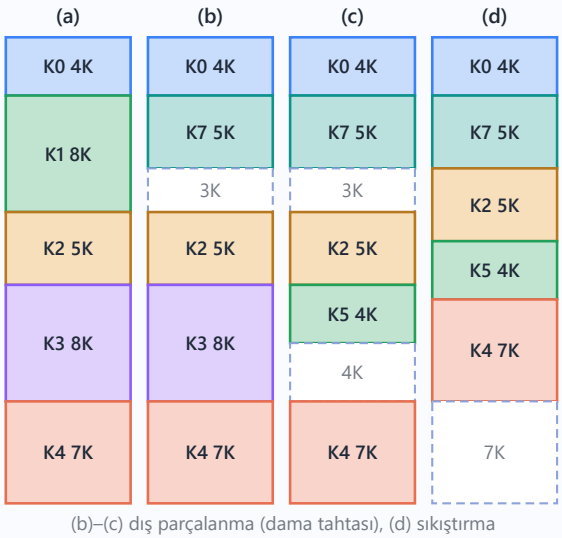
8.10.4 Kesimlemenin artıları ve eksileri

Kesimlemenin avantajları programın mantıksal yapısından gelir:

- Büyüyen ya da küçülen tablolar birbirini etkilemez.
- Kesimler ayrı ayrı derlenebilir. Bir fonksiyon kendi kesimindeyse, fonksiyon değişince öteki kesimlerin adresleri değişmez.
- Bir kütüphane ayrı bir kesime konup süreçler arasında paylaşılabilir.
- Koruma kesim başına ayarlanır: kod yalnız çalıştırılabilir, sabitler yalnız okunabilir, veri okunup yazılabilir.

Dezavantajları ise bitişik tahsisten tanıdıktır. Kesimler değişken uzunlukludur; bellekte yer değiştirdikçe aralarında delikler oluşur. Bu dış parçalanmaya, deseni yüzünden *dama tahtası* (*checkerboarding*) da denir (Şekil 8.56). Çaresi yine sıkıştırmadır. Ayrıca bir kesimin bütünüyle bel-

lekte bulunması gerekir; çok büyük bir kesim belleğe sığmayabilir. Bu son sorunun çözümü kesimleri de sayfalamaktır; MULTICS ve Intel 386 bu yolu izledi.



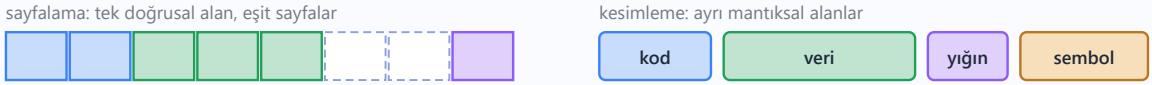
Şekil 8.56: Kesimlerin yer değiştirmesiyle oluşan dama tahtası: (b)–(c) dış parçalanma, (d) sıkıştırma.

8.10.5 Sayfalama ile karşılaştırma

Tablo 8.24, Tanenbaum'un karşılaştırma tablosunu bugünkü donanıma uyarlayarak verir.

Durum	Sayfalama	Kesimleme
Programcı tekniğin farkında olmalı mı?	Hayır	Evet
Kaç doğrusal adres alanı var?	1	Çok
Toplam adres alanı fiziksel belleği aşabilir mi?	Evet	Evet
Kod ve veri ayrılıp ayrı korunabilir mi?	Sayfa bitleriyle evet*	Evet
Boyu değişen tablolar kolayca yerleşir mi?	Hayır	Evet
Kullanıcılar arasında işlev paylaşımı kolay mı?	Paylaşımlı sayfalarla*	Evet
Parçalanma	İç (son sayfa)	Dış
Neden icat edildi?	Fiziksel bellekten büyük doğrusal adres alanı	Program ve veriyi bağımsız mantıksal uzaylara ayırmak; paylaşım, koruma

Tablo 8.24: Sayfalama ve kesimleme (Tanenbaum Şekil 3-32'den uyarlandı). * Tanenbaum burada sayfalama için “hayır” der; bugünkü R/W/NX bitleri ve paylaşımlı sayfalar aynı işi sayfa düzeyinde yapar.



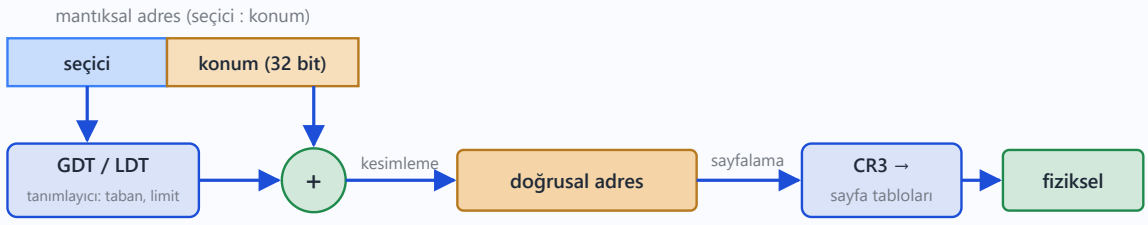
Şekil 8.57: Sayfalamada tek doğrusal adres alanı eşit sayfalara bölünür; kesimlemede kod, veri ve yığın ayrı boyutlu kesimlerdir.

İki teknik farklı problemleri çözmek için doğdu. Sayfalama fiziksel bellekten büyük, tek bir doğrusal adres alanı sağlamak; kesimleme ise programı mantıksal parçalara ayırmak, bu parçaları ayrı korumak ve paylaşmak için icat edildi. Tarih sayfalamadan yana sonuçlandı: kesimlemenin koruma ve paylaşım avantajları sayfa bitleriyle elde edilebiliyor, sayfalamanın dış parçalanmadan kurtulma avantajı ise başka bir yolla elde edilemiyor.

8.11 Gerçek mimariler ve uygulama

8.11.1 x86: kesimleme, sayfalama ve düz model

x86 ailesi kesimlemenin bütün tarihini taşır. 8086'nın *gerçek modunda* fiziksel adres kesim × 16 + konum formülüyle hesaplanır. Sonuç 20 bittir, yani 1 MiB'lık bir adres alanı verir; koruma yoktur. 80386'nın *korumalı modunda* ise mantıksal adres bir seçici ile bir konumdan oluşur. 16 bitlik seçicinin 13 biti indis, bir biti tablo seçimi (GDT ya da LDT), iki biti istenen ayrıcalık düzeyidir (RPL). Seçici, tablodan 8 baytlık bir tanımlayıcı seçer; tanımlayıcıda kesimin tabanı ve limiti bulunur. Taban + konum *doğrusal adresi* verir. Doğrusal adres ardından iki düzeyli sayfalamadan geçerek fiziksel adrese çevrilir (Şekil 8.58). Bu, MULTICS'in “sayfalanmış kesimler” fikrinin donanımdaki karşılığıdır.



Şekil 8.58: Intel 32 bit mimaride kesimleme mantıksal adresi doğrusal adrese, sayfalama doğrusal adresi fiziksel adrese çevirir.

Donanım bu kadar zengin olmasına karşın Linux ve Windows, 32 bitte bile kesimlemeyi fiilen kapattı. Bütün kesimlerin tabanı 0, limiti 4 GiB yapıldı. Böylece doğrusal adres mantıksal adresle aynı olur; buna **düz model** denir. Koruma tamamen sayfa bitleriyle sağlandı. Bunun bir nedeni taşınabilirliktir: kesimleme yalnız x86'da vardı, sayfalama ise her mimaride.

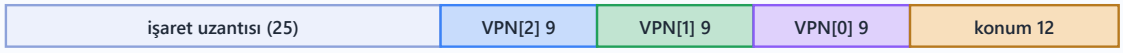
x86-64'ün uzun modu bu durumu donanıma yansıttı. CS, DS, ES ve SS kesimlerinin tabanı donanımca 0 sayılır ve limit denetlenmez. Yalnız FS ve GS kesimlerinin tabanı hâlâ kullanılır: kullanıcı alanında iş parçacığına yerel depolamayı (TLS), çekirdekte çekirdek başına veriyi göstermek için. CS ise ayrıca o anki ayrıcalık düzeyini taşır.

8.11.2 ARMv8-A ve RISC-V

Bugünkü öteki iki büyük mimari aynı fikirleri farklı adlarla uygular (Tablo 8.25).

	ARMv8-A (AArch64)	RISC-V
Sayfa (granül)	4, 16 ya da 64 KiB; Apple Silicon ve bazı Android 15+ aygıtlar 16 KiB	4 KiB (+ 2 MiB, 1 GiB, 512 GiB büyük sayfa)
Sanal adres	48 bit (4 KiB ile 4 düzey, L0–L3); LVA/LPA2 ile 52 bit	Sv39 (3 düzey), Sv48 (4), Sv57 (5)
Tablo tabanı	TTBR0_EL1 alt yarı (kullanıcı), TTBR1_EL1 üst yarı (çekirdek)	satp (mod + ASID + kök çerçeve)
Süreç etiketi	ASID 8/16 bit	ASID (16 bite kadar)
Koruma	AP bitleri, UXN/PXN (çalıştırma yasağı), PAN (≈ SMAP)	R/W/X/U bitleri, SUM
TLB	Donanım yürüyüşü; A/D bitleri isteğe bağlı olarak donanımda (v8.1); contiguous biti	Donanım yürüyüşü (çoğu gerçeklemede); `sfence.vma`

Tablo 8.25: ARMv8-A ve RISC-V'de bellek yönetimi.



Sv39: 3 düzey × 512 girdi; satp kök tablonun çerçevesini ve ASID'i tutar

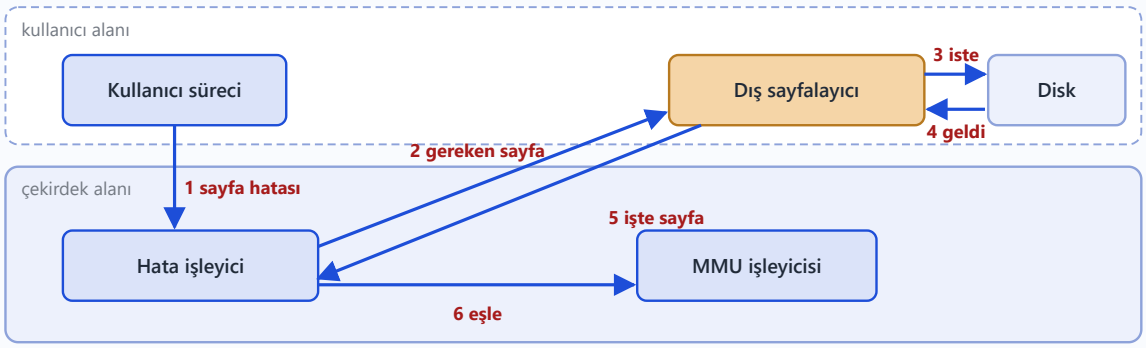
Şekil 8.59: RISC-V Sv39 sanal adresi: üç 9 bitlik indis (VPN[2], VPN[1], VPN[0]) ve 12 bit konum; üst 25 bit bit 38'in kopyasıdır.

İki mimaride de kesimleme yoktur. Sanal adres düzdür; çeviri çok düzeyli sayfa tablosu, TLB ve ASID ile yapılır. ARMv8'in iki tablo taban yazmacı, x86-64'ün kanonik adres bölünmesinin karşılığıdır: kullanıcı ve çekirdek adresleri ayrı tablolardan çevrilir. RISC-V'nin Sv39'u 3 × 9 + 12 = 39 bitlik, 512 GiB'lık bir adres alanı verir; daha büyük alanlar için Sv48 ve Sv57 vardır. Kısacası x86-64 ile aynı fikirler, farklı adlarla uygulanır.

8.11.3 Politika ve mekanizma ayrımı

İşletim sistemi sayfalamaya dört anda katılır. Süreç oluşturulurken sayfa tablosu kurulur. Bağlam değiştirmede yeni sürecin tablosu etkinleştirilir ve TLB ayarlanır. Sayfa hatasında eksik sayfa getirilir. Süreç sonlanınca çerçeveler ve takas alanı bırakılır.

Bu işlerin bir kısmı makineye bağlıdır (MMU'nun tablo biçimi), bir kısmı değildir (hangi sayfanın ne zaman nereye konacağı). Mach çekirdeği bu ayrımı açıkça yaptı. *Mekanizma* çekirdekte kaldı: makineye bağlı bir MMU işleyicisi ve makineden bağımsız bir sayfa hatası işleyicisi. *Politika* ise kullanıcı alanında çalışan bir dış sayfalayıcıya verildi (Şekil 8.60). Sayfa hatası olunca çekirdekteki işleyici, gereken sayfayı dış sayfalayıcıdan ister. Sayfalayıcı sayfayı diskten okur ve geri verir. MMU işleyicisi de sayfayı sürecin adres alanına eşler.



Şekil 8.60: Dış sayfalayıcı ile sayfa hatası: çekirdekteki hata işleyici, kullanıcı alanındaki sayfalayıcıdan sayfayı ister; MMU işleyicisi sayfayı eşler.

Bu tasarım esneklik ve modülerlik kazandırır: farklı uygulamalar farklı sayfalama politikaları kullanabilir. Bedeli, her sayfa hatasında kullanıcı ve çekirdek sınırını birkaç kez geçmektir. Fikrin bugünkü karşılıkları da vardır. Linux'un `userfaultfd` düzeneği sayfa hatalarını kullanıcı alanındaki bir programa iletir; sanal makinelerin canlı göçü ve süreçlerin dondurulup başka makinede sürdürülmesi (CRIU) buna dayanır. seL4 ve Fuchsia gibi mikro çekirdeklerde ise sayfalayıcı zaten kullanıcı alanındadır.

8.12 Özet

- İşlemci yalnız yazmaçlara ve ana belleğe doğrudan erişir. **Bellek hiyerarşisinde** önbellekleri donanım, ana belleği işletim sistemi yönetir. Önbellek 64 baytlık **satırlarla** çalışır ve zamansal ile uzamsal **yerellik** sayesinde işe yarar.
- **Doğrudan eşlemeli** önbellekte blok b, b mod K kümesine gider; aynı kümeye düşen bloklar birbirini kovar (çakışma ıskası). **Ortalama erişim süresi** $T = s + m \cdot T_{alt}$ formülüyle alttan yukarı hesaplanır; örnekte 3,8 ns, DRAM'den yaklaşık 21 kat hızlı.
- **Adres alanı** koruma ve yer değiştirme problemlerini çözer. **Taban ve limit** yazmaçlarıyla geçerli adresler taban … taban + limit − 1'dir; dışarısı tuzak üretir.
- Adres bağlama derleme, yükleme ya da yürütme zamanında yapılabilir. Bugünkü sistemler **yürütme zamanında** bağlar: işlemci **mantıksal** adres üretir, **MMU** onu **fiziksel** adrese çevirir.

- **Dinamik yükleme** (`dlopen`) kodu çağrılınca yükler. **Dinamik bağlamada** kütüphane kodu bir kez bellekte bulunur ve paylaşılır; ELF'te ilk çağrıda değişen kod değil, sürece özel **GOT** girdisidir. `-fPIC` konumdan bağımsız kod üretir.
- Standart **takas** bütün süreci taşır ve yavaştır; bugünkü sistemler soğuk sayfaları takas eder ve önce sıkıştırır.
- **Bitişik tahsiste** delik seçimi ilk uyan, sonraki uyan, en iyi uyan ya da en kötü uyanla yapılır. **Dış parçalanmada** toplam boş alan yeter ama bitişik değildir; %50 kuralına göre belleğin üçte biri kaybolabilir. **İç parçalanma** bloğun içindeki kullanılmayan kısımdır. **Sıkıştırma** delikleri birleştirir ama pahalıdır.
- Boş alan **biteşlem** (birim başına bir bit; k ardışık sıfır aramak doğrusal) ya da **bağlı liste** (adrese göre sıralıysa birleştirme sabit zamanda) ile izlenir. `malloc` boş listeyi boş parçaların içine gömer.
- **Sayfalamada** mantıksal bellek sayfalara, fiziksel bellek **çerçevelere** bölünür; adres (p, d) olur ve fiziksel = f × sayfa boyu + d. Dış parçalanma yoktur; iç parçalanma ortalama yarım sayfadır. Optimum sayfa boyu yaklaşık √(2se)'dir.
- **Sayfa tablosu girdisi** çerçeve numarasıyla birlikte P, R/W, U/S, NX, A, D ve G bitlerini tutar. **Geçerli/geçersiz** biti adres alanı dışını yakalar, ama koruma sayfa ayrıntı düzeyindedir.
- **TLB** son çevirileri tutar. **Etkin erişim süresi** EES = (t + ε) α + (2t + ε)(1 − α); ε = 20, t = 100 ns için α = %80'de 140 ns, %98'de 122 ns. Çok düzeyli tabloda ıska çok daha pahalıdır.
- **Hiyerarşik tablo** yalnız kullanılan bölgelerin alt tablolarını tutar; x86-64 dört (48 bit) ya da beş (57 bit) düzey, her düzeyde 9 bit kullanır. **Büyük sayfalar** TLB erişim alanını büyütür. **Hash** tablonun boyu kullanılan sayfa sayısıyla, **ters** tablonunki fiziksel bellekle orantılıdır.
- **Kesimleme** adres alanını (kesim, konum) biçiminde mantıksal parçalara böler; her kesim bir taban-limit çiftidir ve dış parçalanmaya yol açar. x86-64'te kesimlemeden yalnız FS/GS tabanları kalmıştır; ARMv8 ve RISC-V'de kesimleme yoktur.

8.13 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

bellek yönetimi *memory management* · önbellek, önbellek satırı *cache, cache line* · isabet / ıska *hit / miss* · taban ve limit yazmaçları *base and limit registers* · yer değiştirme *relocation* · mantıksal / fiziksel adres *logical / physical address* · bellek yönetim birimi (MMU) *memory management unit* · paylaşımlı kütüphane *shared library* · takas *swapping* · bitişik tahsis *contiguous allocation* · iç / dış parçalanma *internal / external fragmentation* · sıkıştırma *compaction* · biteşlem *bitmap* · buddy (kardeş) sistemi *buddy system* · sayfalama, sayfa, çerçeve *paging, page, frame* · sayfa tablosu, sayfa tablosu girdisi *page table, page table entry* · geçerli / geçersiz biti *valid–invalid bit* · TLB *translation lookaside buffer* · etkin erişim süresi *effective access time*

hiyerarşik sayfa tablosu *hierarchical page table* | hash sayfa tablosu *hashed page table* | ters sayfa tablosu *inverted page table* | kesimleme *segmentation*

8.14 Kendini Dene

1. 4 kümeli, 64 bayt satırlı, başlangıçta boş doğrudan eşlemeli bir önbellekte sırayla 16, 80, 40, 272 ve 20 adreslerine erişiliyor. Hangi erişimler isabettir?

   Cevap

   Bloklar 0, 1, 0, 4, 0; kümeler 0, 1, 0, 0, 0'dır. 16: ıska. 80: ıska. 40: blok 0 küme 0'da, isabet. 272: blok 4 blok 0'ı kovar, ıska. 20: blok 0 artık yok, ıska. Toplam 1 isabet, 4 ıska.

2. Bir sürecin taban yazmacı 5000, limit yazmacı 2000'dir. MMU önce limit denetimi yapıp sonra tabanı ekliyor. Mantıksal 1999 ve 2000 adresleri ne olur?

   Cevap

   1999 < 2000 olduğu için geçerlidir ve fiziksel 6999'a gider. 2000 limite eşittir; geçerli konumlar 0 … 1999 olduğu için tuzak oluşur.

3. Delikler adres sırasıyla 300, 120, 500 ve 200 K'dır. İstekler sırayla 180, 100 ve 450 K'dır. İlk uyan, en iyi uyan ve en kötü uyan hangisinde bir istek karşılanamaz?

   Cevap

   İlk uyan: 180 → 300 (120 kalır), 100 → aynı deliğin 120'lik artığı (20 kalır), 450 → 500 (50 kalır); hepsi yerleşir. En iyi uyan: 180 → 200 (20), 100 → 120 (20), 450 → 500 (50); hepsi yerleşir. En kötü uyan: 180 → 500 (320), 100 → 320 (220); 450 için en büyük delik 300'dür, istek karşılanamaz.

4. Sayfa boyu 1 KiB'tır ve sayfa 4, çerçeve 9'a eşlenmiştir. Mantıksal 5000 adresinin fiziksel karşılığı nedir?

   Cevap

   $p = \lfloor 5000 / 1024 \rfloor = 4$, $d = 5000 - 4096 = 904$. Fiziksel $= 9 \times 1024 + 904 = 9216 + 904 = 10120$.

5. TLB araması 5 ns, bellek erişimi 100 ns, tablo tek düzeyli ve isabet oranı %90'dır. Etkin erişim sü-

resi nedir? EES'nin 125 ns olması için isabet oranı kaç olmalıdır?

Cevap

İsabet 105 ns, ıska 205 ns. EES = 0,9 × 105 + 0,1 × 205 = 94,5 + 20,5 = 115 ns. 205 − 100α = 125 için α = 0,8, yani %80.

6. 32 bit x86 iki düzeyli sayfalamasında (10 + 10 + 12) sanal 0x00C05123 adresi için üst indis, alt indis ve konum nedir?

Cevap

Üst indis 0x00C05123 >> 22 = 3, alt indis (0x00C05123 >> 12) & 0x3FF = 0xC05 & 0x3FF = 5, konum 0x123'tür. Adres 12–16 MiB bölgesindedir.

7. 8 GiB RAM, 4 KiB çerçeve ve 8 baytlık girdiyle bir ters sayfa tablosu ne kadar yer kaplar? Sistemde 200 süreç olsaydı bu değer değişir miydi?

Cevap

$2^{33} / 2^{12} = 2^{21}$ = 2.097.152 girdi; × 8 bayt = 16 MiB. Değişmezdi: ters tablo fiziksel bellekle orantılıdır ve sistemde tektir, süreç sayısına bağlı değildir.

8. Silberschatz'ın kesim tablosunda (kesim 2: limit 400, taban 4300; kesim 4: limit 1000, taban 4700) (4, 999) ve (2, 400) adresleri nereye gider?

Cevap

(4, 999): 999 < 1000, fiziksel 4700 + 999 = 5699. (2, 400): konum limite eşittir; geçerli konumlar 0 … 399 olduğu için tuzak.

9. 1024 K'lık bir buddy sisteminde 70 K'lık bir istek kaç K'lık blok alır ve ne kadar iç parçalanma oluşur?

Cevap

70 K'ya yeten en küçük ikinin kuvveti 128 K'dır. Süreç 128 K'lık bloğu alır; iç parçalanma 128 − 70 = 58 K'dır.

10. Dinamik bağlanmış bir programda `puts` ilk kez çağrıldıktan sonra hangi sayfa değişir ve bu

neden önemlidir?

Cevap

Yalnız sürece özel GOT sayfasındaki `puts` girdisi değişir; çözücü oraya gerçek adresi yazar. Programın ve libc'nin kod sayfaları değişmediği için salt okunur kalır ve bütün süreçler arasında paylaşılmaya devam eder.

8.15 İleri okuma

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, "Address Translation", "Segmentation", "Free-Space Management", "Paging: Faster Translations (TLBs)" ve "Paging: Smaller Tables" bölümleri: kesim yazmaçları ve TLB konularını kısa, çalıştırılabilir benzetimlerle anlatır. Kitap ostep.org'da ücretsizdir.
- A. S. Tanenbaum ve H. Bos, *Modern Operating Systems*, Bölüm 3: yer değiştirme problemini, biteşlem ve bağlı listeyi, MMU ve çok düzeyli tablo örneklerini bu nottaki şekillerin özgün biçimleriyle anlatır.
- U. Drepper, "What Every Programmer Should Know About Memory", 2007: önbellek, TLB ve NUMA'nın program performansına etkisini ölçümlerle gösteren uzun bir makaledir.
- Linux çekirdek belgeleri, "Transparent Hugepage Support": şeffaf büyük sayfaların nasıl çalıştığını, ne zaman kazanç sağladığını ve nasıl ayarlandığını anlatır.

8.16 Kaynaklar

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 9: Main Memory.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 3: Memory Management (§3.1–3.3, 3.5–3.7).
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Address Spaces", "Address Translation", "Segmentation", "Free-Space Management", "Paging", "TLBs", "Smaller Tables".
4. D. E. Knuth, *The Art of Computer Programming*, Cilt 1, §2.5 Dynamic Storage Allocation (%50 kuralı).
5. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Cilt 3A, Bölüm 4–5 (Paging, Protection); *Arm Architecture Reference Manual for A-profile* (VMSAv8-64); *RISC-V Privileged Specification* (Sv39/48/57).
6. U. Drepper, "How To Write Shared Libraries", 2011; U. Drepper, "What Every Programmer Should Know About Memory", 2007; Linux çekirdek belgeleri: mm/, "Transparent Hugepage Support", userfaultfd.

KISIM III · BÖLÜM 9

Sanal Bellek

Bir program çalışırken kodunun ve verisinin hepsine aynı anda ihtiyaç duymaz. Hata işleme kodu belki hiç çalışmaz, büyük bir dizinin yalnız ilk satırları kullanılır, bir menünün çoğu seçeneğine hiç dokunulmaz. Sanal bellek bu gözlemden yararlanır: sürecin yalnız gerçekten kullanılan sayfaları fiziksel bellekte durur, geri kalanı diskte bekler ya da hiç yüklenmez. Böylece bir süreç fiziksel bellekten büyük olabilir, aynı bellekte daha çok süreç çalışabilir ve programlar daha hızlı başlar. Not, bu düzeni ayakta tutan donanım ve çekirdek mekanizmasıyla başlar: geçerli biti, sayfa hatası, komutun yeniden başlatılması. Ardından sayfa hatasının bedeli etkin erişim süresiyle ölçülür; yazınca kopyalama ve bellek eşlemeli dosyalar aynı mekanizmanın iki kullanımı olarak anlatılır. Notun en uzun kısmı, bellek dolunca hangi sayfanın çıkacağına karar veren yer değiştirme algoritmalarıdır. FIFO, OPT, LRU, ikinci şans, saat, NRU, yaşlandırma, çalışma kümesi ve WSClock aynı referans dizisi üzerinde adım adım çalıştırılır. Son kesimler çerçeve tahsisini, thrashing'i ve sayfa boyutu gibi tasarım kararlarını ele alır. Çekirdeğin kendi belleği için kullandığı buddy ve slab ayırıcıları ile Linux ve Windows'un bugünkü çözümleri notu tamamlar. Bütün izlemeler ve sayılar algoritmalar gerçekten çalıştırılarak denetlenmiştir.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum09` · **Laboratuvar:** `github.com/sercankulcu/operating-systems-labs/lab09`

ÖĞRENME ÇIKTILARI

Talebe bağlı sayfalama
Sanal belleğin amacını açıklamak; geçerli–geçersiz bitini, sayfa hatasının işlenme adımlarını ve komutun yeniden başlatılmasını anlatmak.

Performans
Sayfa hatası olasılığından etkin erişim süresini hesaplamak; HDD, SSD ve sıkıştırılmış takas arasındaki farkı yorumlamak.

Yazınca kopyalama ve mmap
`fork()` sonrasında yazınca kopyalamayı ve bellek eşlemeli dosyaların sayfa önbelleğiyle ilişkisini adım adım izlemek.

Yer değiştirme
FIFO, OPT, LRU, ikinci şans, saat, NRU, yaşlandırma, çalışma kümesi ve WSClock algoritmalarını bir referans dizisi üzerinde elle çalıştırmak; Belady anomalisini açıklamak.

Tahsis ve thrashing
Eşit ve orantılı, yerel ve genel tahsisi karşılaştırmak; thrashing'i çalışma kümesi ve sayfa hatası sıklığıyla açıklamak.

Tasarım ve gerçek sistemler
Sayfa boyutunu, TLB erimini, program yapısını, sayfa sabitlemeyi, buddy ve slab ayırıcılarını; Linux (MGLRU, zswap) ve Windows yaklaşımlarını tanımak.

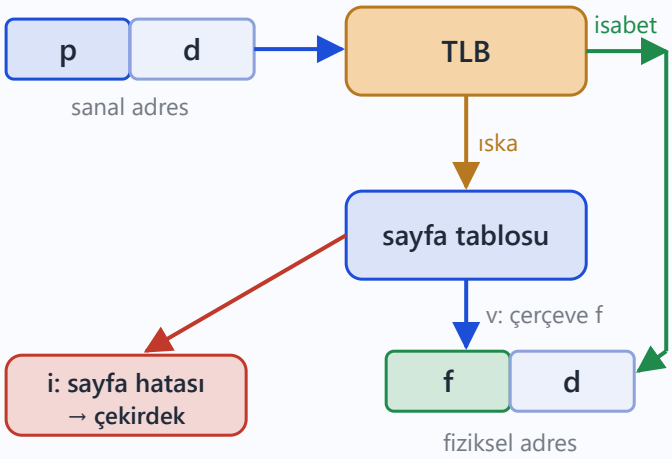
9.1 Sanal belleğe giriş

Sayfalamalı bir sistemde süreç, adres alanını sabit boyutlu sayfalar hâlinde görür ve her sayfa fiziksel bellekte bir çerçeveye yerleşir. Klasik sayfalama sürecin bütün sayfalarının bellekte olmasını bekler. Sanal bellek bu koşulu kaldırır: sayfaların bir kısmı bellekte, bir kısmı diskte durabilir ve süreç yine çalışır. Bu kesim önce sayfalamanın gerekli parçalarını kısaca hatırlatır, sonra koşulu kaldırmanın neden değerli olduğunu anlatır.

9.1.1 Ön bilgi: sayfalama ve TLB

Sürecin ürettiği her adres bir **sanal adrestir** (*virtual address*). Donanım bu adresi iki parçaya ayırır: üst bitler sayfa numarası p, alt bitler sayfa içindeki konum, yani bağıl konum d'dir. 4 KB'lık sayfada bağıl konum 12 bittir. Sayfa tablosu p'yi bir çerçeve numarası f'ye çevirir; bağıl konum hiç değişmeden fiziksel adrese taşınır. Fiziksel adres böylece f ile d'nin yan yana yazılmasıdır.

Sayfa tablosu bellekte durduğu için her erişimde tabloya gitmek bellek trafiğini ikiye katlardı. Bunu **TLB** (*translation lookaside buffer*) önler. TLB, son kullanılan sayfa–çerçeve eşlemelerini tutan küçük ve çok hızlı bir önbellektir. Programlar kısa sürede az sayıda sayfaya dokunduğu için isabet oranı %99'un üstündedir. Iskada donanım sayfa tablosunu yürür; x86-64'te bu yürüyüş 4 ya da 5 düzeylidir (Şekil 9.1).



Şekil 9.1: Adres çevirisi. Sanal adres (p, d) parçalarına ayrılır. Önce TLB'ye bakılır; ıskada sayfa tablosu yürünür. Girdi geçerliyse çerçeve f ile bağıl konum d birleşir, geçersizse sayfa hatası tuzağı çekirdeğe geçer.

Sayfa tablosunun her girdisi çerçeve numarasının yanında birkaç bit taşır. **Geçerli** (*present*) biti sayfanın bellekte olup olmadığını söyler. **R** (referans, x86'da *accessed*) biti sayfaya erişildiğinde donanımca 1 yapılır. **M** (değişti, x86'da *dirty*) biti sayfaya yazıldığında 1 olur. Koruma bitleri de yazma iznini (R/W), kullanıcı erişimini (U/S) ve kod çalıştırma yasağını (NX) belirler. R ve M bitleri yer değiştirme algoritmalarının hammaddesidir. Geçerli biti 0 olan bir girdiye erişim ise bir **sayfa hatası** doğurur. Sayfa bellekte değilse yedekleme deposundan getirilir; gerekirse seçilen bir kurban sayfanın yerine yüklenir.

Farklı adreslerle çeviriyi, TLB isabet ve ıskalarını, sayfa hatalarını denemek için Sayfalama ve Adres Çevirisi Simülatörü kullanılabilir.

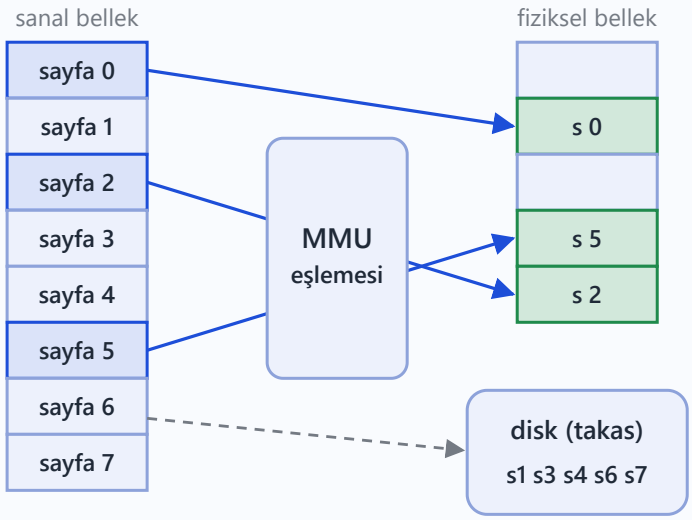
9.1.2 Neden sanal bellek?

Gerçek programlara bakıldığında bir programın tamamının nadiren kullanıldığı görülür. Hata durumlarını işleyen kod çoğu çalıştırmada hiç yürümez. Seyrek kullanılan seçenekler ve özellikler bellekte boşuna yer kaplar. Programcılar dizileri en kötü duruma göre büyük ayırır: 100×100'lük bir tablo çoğu zaman 10×10'u kadar dolar. Programın tamamı gerekli olduğunda bile hepsi aynı anda gerekmez.

Eski çözüm **takastı** (*swapping*): sürecin tamamı belleğe getirilir ya da bütünüyle diske götürülür. Büyük bir süreçte bu çok pahalıdır, çünkü süreç birkaç sayfasını kullanacak olsa bile bütün görüntüsü taşınır.

Tanım 9.1

Sanal bellek (*virtual memory*), sürecin mantıksal belleğini fiziksel bellekten ayıran tekniktir. Süreç büyük ve kesintisiz bir adres alanı görür; bu alanın yalnız bir kısmı fiziksel bellektedir. Süreç kısmen bellekteyken de çalışır (Şekil 9.2).



Şekil 9.2: Sekiz sayfalık sanal belleğin yalnız 0, 2 ve 5 numaralı sayfaları fiziksel bellektedir. Diğerleri diskte, takas alanında durur. Eşlemeyi MMU (bellek yönetim birimi) yapar.

Bu ayrımın kazançları hem programcıya hem sisteme yansır:

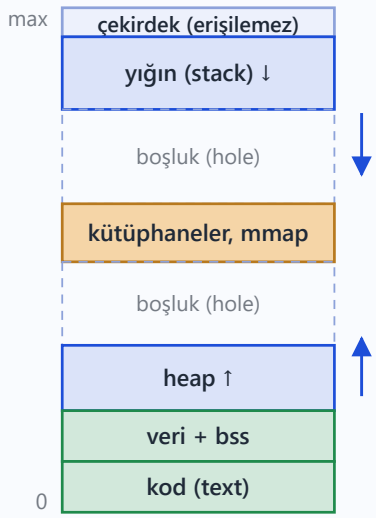
- Program fiziksel bellekle sınırlı değildir. 16 GB'lık bir makinede sanal boyutu 40 GB'ı aşan süreçler çalışabilir.
- Her süreç daha az çerçeve kullanır, dolayısıyla aynı bellekte daha çok süreç tutulur. Çok programlama derecesi ve CPU kullanımı artar.
- Yükleme ve takas için daha az G/Ç gerekir: yalnız kullanılan sayfalar okunur. Programlar daha hızlı başlar.
- Sayfa paylaşımı kolaylaşır; aynı çerçeve birden çok sürecin tablosunda görünebilir.

Sanal bellekten önce fiziksel bellekten büyük programlar elle parçalara bölünürdü. Bu eski **katman** (*overlay*) yönteminde programcı, programı birbirinin yerine yüklenecek parçalara ayırmak ve yüklemeyi kendisi yönetmek zorundaydı. Sanal bellekle aynı iş işletim sistemince sayfa düzeyinde ve otomatik olarak yapılır.

9.1.3 Sanal adres alanı

Sürecin bellekteki mantıksal görünümüne **sanal adres alanı** denir. Süreç kendini 0'dan başlayan bitişik adreslerde görür. Fiziksel bellek ise dağınık çerçevelerden oluşur ve iki görünüm arasındaki eşlemeyi MMU yapar.

Alanın düzeni bütün Unix benzeri sistemlerde aynı mantığı izler (Şekil 9.3). En altta kod (*text*), onun üstünde ilk değerli veri ve sıfırla başlayan bss bölümü vardır. Heap bunların üstünden yukarı doğru büyür. Yığın (*stack*) en yüksek kullanıcı adresinden aşağı doğru büyür. Arada paylaşımlı kütüphanelerin ve `mmap` bölgelerinin eşlendiği alan durur. Geri kalan kullanılmayan aralıklar **boşluktur** (*hole*).



Şekil 9.3: Sanal adres alanı. Yığın yukarıdan aşağı, heap aşağıdan yukarı büyür. Aradaki boşluklarda kütüphaneler ve mmap bölgeleri yer bulur; boşluklar sayfa ya da çerçeve harcamaz.

Boşluğun çerçeve harcamaması önemlidir. Yığın ya da heap yeni bir sayfaya taşınana kadar o sayfa için fiziksel bellek ayrılmaz. Böyle, büyük kısmı kullanılmayan bir adres alanına **seyrek adres alanı** (*sparse address space*) denir. Sayfa tablosu da boşluklar için girdi tutmak zorunda değildir; çok düzeyli tabloda kullanılmayan alt tablolar hiç ayrılmaz.

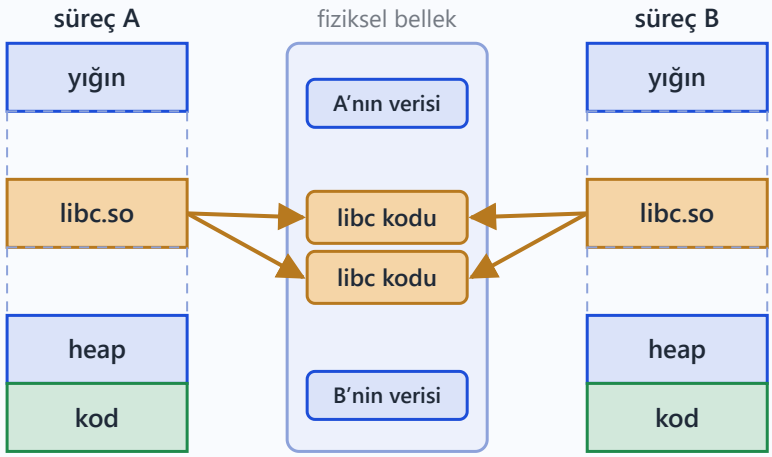
Bugünkü x86-64 işlemciler 48 bitlik sanal adres kullanır; bunun yarısı, 128 TiB, kullanıcıya ayrılır. 5 düzeyli sayfalamayla sanal adres 57 bite çıkar. Adres alanı düzeni rastgeleleştirme (ASLR, *address space layout randomization*) bölgelerin yerini her çalıştırmada rastgele kaydırır. Böylece saldırgan bir kütüphanenin ya da yığının adresini önceden bilemez.

Sanal bellek iki yolla kurulabilir. Bugün evrensel olan yol **talebe bağlı sayfalamadır**. Değişken boyutlu kesimlerle çalışan talebe bağlı kesimleme ise tarihsel bir yaklaşımdır; Burroughs bilgisayarları ve OS/2 bunu kullanmıştı. Linux'ta bir sürecin bölgeleri `cat /proc/self/maps` ile görülebilir. Çıktının her satırı bir **sanal bellek alanıdır** (VMA, *virtual memory area*): başlangıç ve bitiş adresi, izinler ve varsa eşlenen dosya.

9.1.4 Paylaşımlı sayfalar ve kütüphaneler

Sanal belleğin bir yan kazancı da paylaşımdır. Aynı fiziksel çerçeve birden çok sürecin sayfa tablosunda görünebilir; her süreç onu kendi adres alanının istediği yerinde görür (Şekil 9.4).

- **Paylaşımlı kütüphane** (libc, libstdc++): salt okunur kod sayfaları her süreçte kendi adresinde eşlenir, ama fiziksel kopya tektir. Yüz süreç libc kullansa da libc kodu bellekte bir kez durur.
- **Paylaşımlı bellekle süreçler arası iletişim**: iki süreç aynı bölgeyi eşler; birinin yazdığını öteki hemen görür.
- `fork()`: çocuk süreç yaratılırken sayfalar kopyalanmaz, paylaşılır. Kopya ancak biri yazınca çıkarılır.



Şekil 9.4: İki süreç libc.so bölgesini kendi adres alanlarında farklı yerlere eşler, ama ikisi de aynı fiziksel çerçevelere bakar. Verileri ise her süreçte ayrı çerçevelerdedir.

Yazılabilir veri sayfaları her süreçte özeldir. Paylaşımın güvenli olmasının nedeni kod sayfalarının salt okunur olmasıdır: hiçbir süreç ortak kopyayı değiştiremez.

9.2 Talebe bağlı sayfalama

Bir program diskten yüklenirken iki seçenek vardır. Birincisi bütün programı baştan belleğe okumaktır; ama kullanıcı programın küçük bir kısmını kullanacaksa bu boşa harcanmış zaman ve bellektir. İkincisi sayfaları ancak gerektiğinde getirmektir. Sanal bellek sistemleri ikinci yolu izler.

9.2.1 Tembel yükleme

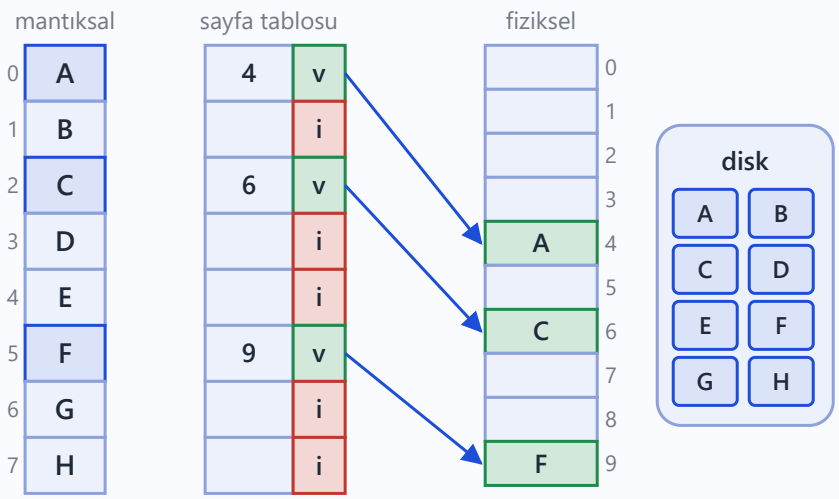
Tanım 9.2

Talebe bağlı sayfalama (*demand paging*) yönteminde süreç başlarken hiçbir sayfa yüklenmez (ya da çok az sayfa yüklenir). Bir sayfa yalnız erişildiği anda diskten belleğe getirilir. Hiç erişilmeyen sayfa hiç yüklenmez.

Bu tembel yaklaşımın getirisi açıktır: daha az G/Ç, daha az bellek, daha hızlı başlatma ve aynı bellekte daha çok kullanıcı. Bedeli ise her ilk erişimin bir gecikme yaratmasıdır. Bir erişimde üç durum ayrılır:

1. Adres sürecin hiçbir bölgesinde değilse erişim **geçersizdir**; süreç sonlandırılır.
2. Adres geçerli bir bölgededir ve sayfa bellektedir; erişim normal yürür.
3. Adres geçerlidir ama sayfa bellekte değildir; sayfa getirilir ve erişim yinelenir.

Bu işi yapan birime **sayfalayıcı** (*pager*) denir. Eski kitaplarda "tembel takasçı" (*lazy swapper*) adı da geçer, ama yanıltıcıdır: takasçı bütün süreci taşır, sayfalayıcı ise tek tek sayfalarla uğraşır (Şekil 9.5).



Şekil 9.5: A'dan H'ye sekiz sayfalık mantıksal bellek. Sayfa tablosunda A (çerçeve 4), C (çerçeve 6) ve F (çerçeve 9) geçerlidir. Diğer girdiler geçersizdir; bu sayfalar yalnız diskte bulunur.

En uç biçimine **saf talebe bağlı sayfalama** denir. Süreç hiç sayfası bellekte olmadan başlatılır; ilk komut bile bir sayfa hatasıyla gelir. İlk birkaç yüz mikrosaniyede hatalar art arda yağar. Ama süreç gereken sayfaları topladıkça hata oranı hızla düşer. Bunu sağlayan, programların **yerellik** (*locality*) özelliğidir: bir program kısa bir süre içinde adres alanının küçük bir bölgesinde çalışır. Döngüler aynı kodu, diziler komşu adresleri, yığın en üstteki birkaç sayfayı kullanır. Yerellik olmasaydı her komut birkaç sayfa hatası doğurur ve talebe bağlı sayfalama işe yaramazdı.

9.2.2 Geçerli–geçersiz biti

Talebe bağlı sayfalama, sayfa tablosu girdisindeki tek bir bitle çalışır.

Tanım 9.3

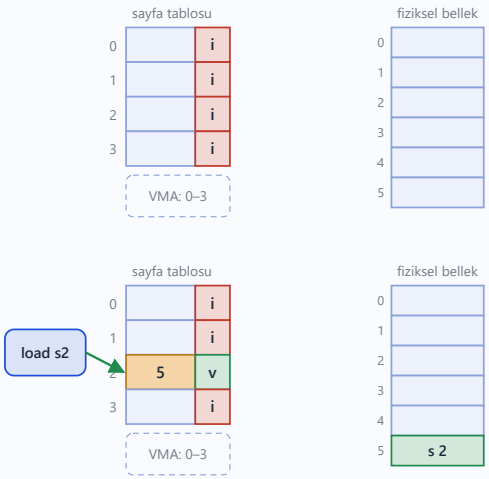
Geçerli–geçersiz biti (*valid–invalid bit*) her sayfa tablosu girdisinde bulunur. *v* (geçerli), sayfanın hem adres alanına ait hem de bellekte olduğunu gösterir. *i* (geçersiz), sayfanın ya bellekte olmadığını ya da adres alanına hiç ait olmadığını gösterir.

i biti iki farklı durumu kapsar. Donanım ikisini ayırmaz; ikisinde de tuzak üretir. Ayrımı işletim sistemi kendi kayıtlarına, yani VMA listesine bakarak yapar. Aşağıdaki örnek dört sayfalık bir bölgede bu ayrımı adım adım izler.

Örnek 9.1

Sürecin VMA'sı 0–3 numaralı sayfaları kapsıyor. Fiziksel bellekte 0–5 numaralı altı çerçeve var. Süreç önce sayfa 2'ye, sonra yine sayfa 2'ye, en sonda sayfa 6'ya erişiyor. Her erişimde ne olur?

1. **Başlangıç.** Tablonun dört girdisi de *i* yapılmıştır; hiçbir çerçeve ayrılmamıştır (Şekil 9.6a).
2. **Sayfa 2'ye ilk erişim.** MMU girdi 2'de *i* görür ve sayfa hatası tuzağı üretir. Çekirdek adresin geçerli bir bölgede (VMA 0–3) olduğunu doğrular. Sayfayı boş çerçeve 5'e okur, girdiyi (5, v) yapar ve komutu yeniden çalıştırır (Şekil 9.6b).
3. **Sayfa 2'ye ikinci erişim.** Girdi artık geçerlidir. Çeviri doğrudan yapılır ve eşleme TLB'ye de yüklenir. Tuzak oluşmaz.
4. **Sayfa 6'ya erişim.** MMU yine tuzak üretir. Çekirdek bu kez VMA listesinde sayfa 6'yı kapsayan bir bölge bulamaz. Bu bir **geçersiz referanstır**; çekirdek süreci `SIGSEGV` sinyaliyle sonlandırır.



Şekil 9.6: (a) Başlangıçta dört girdi de geçersizdir. (b) Sayfa 2'ye ilk erişimden sonra: sayfa çerçeve 5'e okunmuş, girdi (5, v) olmuştur. VMA 0–3 dışındaki sayfa 6'ya erişim ise `SIGSEGV` ile biter.

Sık yapılan hata

"Geçersiz bit = hatalı erişim" diye düşünmek. Talebe bağlı sayfalamada *i* biti çoğu zaman yalnız "bu sayfa henüz yüklenmedi" anlamına gelir ve tuzak tamamen normal bir olaydır. Süreç ancak çekirdek adresin hiçbir bölgeye ait olmadığını ya da izinlerin ihlal edildiğini anlarsa sonlandırılır. `int *p = NULL; *p = 5;` satırı bu ikinci duruma örnektir: 0. sayfa bilerek hiçbir bölgeye ait sayılmaz.

9.2.3 Sayfa hatasının işlenmesi

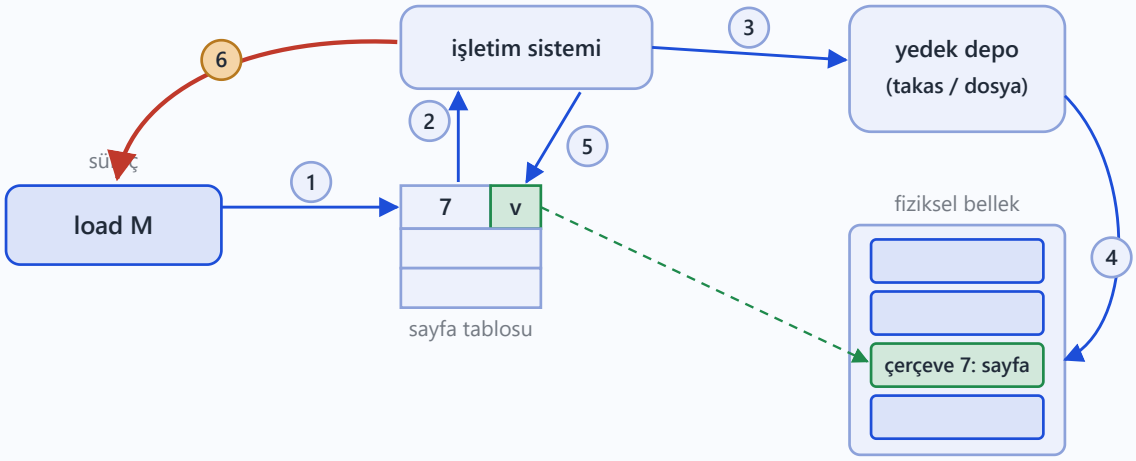
Tanım 9.4

Sayfa hatası (*page fault*), sayfa tablosunda geçersiz işaretli bir sayfaya erişilince MMU'nun ürettiği tuzaktır. Tuzak, denetimi çekirdekteki sayfa hatası işleyicisine verir.

Silberschatz'ın klasik şeması bir sayfa hatasının işlenmesini altı adımda anlatır (Şekil 9.7). Süreç M adresinden bir değer yüklemek istiyor ve M'nin sayfasına ait girdinin biti *i*'dir.

1. **Referans.** MMU girdiyi okur ve sayfanın bellekte olmadığını görür.
2. **Tuzak.** Donanım çekirdeğe geçer. Çekirdek referansın geçerli olup olmadığına bakar; geçersizse süreci sonlandırır.
3. **Sayfa yedek depoda.** Sayfanın diskteki yeri bulunur; bu yer takas alanı ya da bir dosya olabilir. Boş bir çerçeve seçilir.
4. **Getirme.** Disk okuma işi çizelgelenir ve sayfa çerçeve 7'ye okunur. Okuma milisaniyeler sürebileceği için CPU bu arada başka bir sürece verilir.

5. **Tabloyu güncelleme.** Girdiye çerçeve 7 yazılır, bit *v* yapılır.
6. **Yeniden başlatma.** Hataya yol açan komut baştan çalışır. Süreç, sayfa hiç eksik değilmiş gibi devam eder.



Şekil 9.7: Sayfa hatasının işlenmesi (Silberschatz Şekil 10.5'in yeniden çizimi). 1 referans, 2 tuzak, 3 sayfanın yedek depoda bulunması, 4 çerçeve 7'ye okunması, 5 tablonun güncellenmesi, 6 komutun yeniden başlatılması.

Sayfa hatası, tuzakların *hata* (*fault*) türündendir. Bu türde kaydedilen program sayacı hatayı yapan komutu gösterir, bir sonrakini değil. Komutun yeniden çalıştırılabilmesi bu sayede mümkündür. x86 işlemciler hatalı sanal adresi ayrıca CR2 yazmacına yazar; işleyici hangi sayfanın istendiğini oradan okur.

Bir ayrıntı da boş çerçeveyle ilgilidir. Üçüncü adımda boş çerçeve yoksa bir kurban seçilmelidir. Kurban değiştirilmişse (kirliyse) önce diske yazılır. Yazma sürerken çerçeve “meşgul” işaretlenir ki başka bir hata onu da kurban seçmesin. Kurbanı seçmek, sayfa yer değiştirme algoritmalarının işidir.

9.2.4 Kod izleme: adres çevirisi ve sayfa hatası

Adres çevirisini donanım yapar, ama mantığını birkaç satır C ile yazmak mümkündür. Aşağıdaki `ceviri()` fonksiyonu 4 KB'lık sayfalarla bir sanal adresi fiziksel adrese çevirir; TLB ıskasında sayfa tablosuna gider, gerekirse sayfa hatası işleyicisini çağırır.

```
int ceviri(int va) {
  int p = va >> 12;      /* sayfa no */
  int d = va & 0xFFF;    /* bağıl konum */
  int f = tlb_ara(p);    /* -1: ıska */
  if (f < 0) {
    if (!pt[p].gecerli)  /* bellekte yok */
      sayfa_hatasi(p);   /* diskten al */
    f = pt[p].cerceve;
    tlb_ekle(p, f);
  }
  pt[p].R = 1;
  return (f << 12) | d;
}
```

İlk iki satır adresi parçalar. `va >> 12` adresi 12 bit sağa kaydırarak sayfa numarasını bulur; `va & 0xFFF` alt 12 biti, yani bağıl konumu ayırır. `tlb_ara(p)` TLB'de sayfayı arar ve bulamazsa −1 döndürür. Iskada `if` bloğu çalışır: girdinin geçerli biti 0 ise `sayfa_hatasi(p)` sayfayı diskten getirir ve girdiyi günceller. Ardından çerçeve tablodan okunur ve eşleme TLB'ye eklenir. Her durumda sayfanın R biti 1 yapılır; yer değiştirme algoritmaları bu biti kullanır. Son satır çerçeve numarasını 12 bit sola kaydırıp bağıl konumla birleştirir.

Başlangıç durumunda TLB'de sayfa 0 (çerçeve 1) ve sayfa 1 (çerçeve 4) vardır. Sayfa 2 bellektedir (çerçeve 5) ama TLB'de değildir. Sayfa 3 ise diskte durur. Fonksiyon sırayla üç adresle çağrılıyor (Tablo 9.1).

Çağrı	p	d	TLB	Tablo	f	Sonuç
`ceviri(0x2A10)`	2	0xA10	ıska	v, sayfa hatası yok	5	(5 << 12) | 0xA10 = 0x5A10
`ceviri(0x2B04)`	2	0xB04	isabet	bakılmaz	5	0x5B04
`ceviri(0x3008)`	3	0x008	ıska	i → sayfa hatası, (v, 7)	7	0x7008

Tablo 9.1: `ceviri()` izlemesi. TLB başta {0→1, 1→4}; sayfa 2 çerçeve 5'te, sayfa 3 diskte; boş çerçeve 7.

- **0x2A10.** Üst bitler 0x2, alt 12 bit 0xA10'dur: p = 2, d = 0xA10. TLB'de p = 2 yoktur, `f = -1`. Girdi 2'nin biti *v* olduğu için sayfa hatası çağrılmaz. f = 5 tablodan okunur ve (2, 5) TLB'ye eklenir. R biti 1 olur; sonuç (5 << 12) | 0xA10 = 0x5A10.
- **0x2B04.** Aynı sayfanın başka bir baytı: p = 2, d = 0xB04. Bu kez TLB'de isabet vardır, `if` bloğu atlanır ve sayfa tablosuna hiç gidilmez. Sonuç 0x5B04'tür. Yerellik sayesinde erişimlerin büyük çoğunluğu bu kısa yolu izler.
- **0x3008.** p = 3, d = 0x008. TLB'de yoktur. Girdi 3'ün biti *i*'dir, yani sayfa hatası oluşur. Çekirdek sayfayı diskten boş çerçeve 7'ye okur; süreç bu sırada bekler. İşleyici girdiyi (v, 7) yapar. Komut yeniden çalışınca f = 7 okunur ve TLB'ye eklenir; sonuç 0x7008'dir (Şekil 9.8).



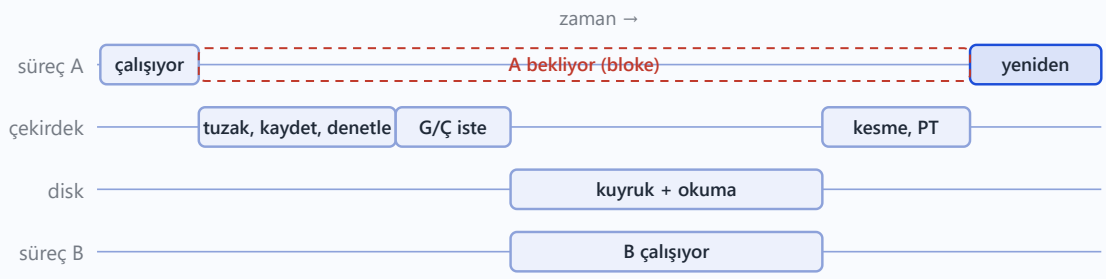
Şekil 9.8: Üç çağrıdan sonra: TLB'de dört eşleme var, sayfa 3 artık çerçeve 7'de ve geçerli. Sayfa 2 ve 3'ün R bitleri 1'dir.

Üç çeviride 1 TLB isabeti ve 2 ıska vardır; ıskalardan biri sayfa hatasına, yani disk G/Ç'sine dönüşmüştür. Fonksiyonun kendisi O(1)'dir. Asıl maliyet, ıskanın tablo yürüyüşünde ve özellikle sayfa hatasının disk okumasındadır. İkisinin farkı birkaç büyüklük mertebesidir.

9.2.5 Sayfa hatasının zaman çizelgesi

Ders kitapları sayfa hatasını on beş ayrıntılı adımda anlatır. Bu adımlar üç büyük maliyete toplanır: tuzağı işlemek (mikrosaniyeler), sayfayı okumak (HDD'de milisaniyeler, NVMe SSD'de 10–100 µs) ve süreci yeniden başlatmak (mikrosaniyeler). Zaman çizelgesi bunları A süreci, çekirdek, disk ve B süreci satırlarında gösterir (Şekil 9.9).

1. **Adım 1–5.** Çekirdeğe tuzak; kullanıcı yazmaçları ve süreç durumu kaydedilir. Kesmenin sayfa hatası olduğu ve referansın geçerli olduğu anlaşılır. Sayfanın diskteki yeri bulunur.
2. **Adım 6–9.** Okuma isteği disk ya da SSD kuyruğuna konur. Aygıt beklenir; HDD'de arama ve dönme gecikmesi eklenir. Bu arada CPU başka bir sürece, B'ye verilir.
3. **Adım 10–13.** "G/Ç tamamlandı" kesmesi gelir. B'nin durumu kaydedilir, kesmenin diskten geldiği anlaşılır. Sayfa tablosu güncellenir ve A hazır kuyruğuna alınır.
4. **Adım 14–15.** A yeniden CPU'yu aldığında yazmaçları ve sayfa tablosu yüklenir; yarıda kalan komut baştan çalışır.



Şekil 9.9: Sayfa hatasının zaman çizelgesi. A tuzağa girer, çekirdek G/Ç ister; disk okurken B çalışır. Kesme gelince tablo güncellenir ve A yeniden başlar.

Çizelge bir şeyi açıkça gösterir: sayfa hatası A için uzun bir bekleme, sistem için ise yalnız bir bağlam değiştirmedir. CPU boşa gitmez; okuma sürerken B ilerler. Ama A'nın gözünden tek bir bellek erişimi, normal süresinin binlerce katına çıkmıştır. Bu farkın sayısal karşılığı etkin erişim süresidir.

9.2.6 Komutu yeniden başlatma ve hata türleri

Sayfa hatasından sonra komutun baştan çalışabilmesi, talebe bağlı sayfalamanın donanımdan istediği tek gerçek özelliktir. Hata, komut getirilirken ya da bir işlenen okunurken oluşabilir. Çoğu durumda komutu baştan çalıştırmak yeter: komut henüz hiçbir şeyi değiştirmemiştir.

Zor durum, komutun bir kısmını değiştirdikten sonra hata almasıdır. IBM 360'ın MVC komutu 256 bayta kadar bir bloğu taşır. Kaynak ya da hedef blok sayfa sınırını aşıyorsa taşıma yarıda hata alabilir. Bloklar örtüşüyorsa kaynak çoktan bozulmuştur ve komut baştan çalıştırılamaz. Otomatik artırmalı adresleme de benzer bir sorun yaratır: komut yazmacı artırmıştır, yeniden çalışınca yanlış adresten başlar.

Motorola 680x0 ailesinde `MOVE.L 6(A1),2(A0)` komutu üç sözcük uzunluğundadır. İkinci işlenen okunurken hata olursa program sayacı komutun içinde bir yeri gösterir. İşletim sistemi komutun nerede başladığını bilemez. Bu yüzden bazı işlemciler her komuttan önce program sayacını gizli bir yazmaca kopyalar.

Çözümler iki türlüdür. Birincisi, komut başlamadan önce dokunacağı bölgelerin iki ucunu da yoklamaktır; eksik sayfa varsa hata komut başlamadan alınır. İkincisi, değiştirilen değerleri geçici yazmaçlarda tutup hata olursa eski hâline döndürmektir. Bugünkü işlemcilerde **kesin istisnalar** (*precise exceptions*) bunu donanımda güvenceye alır: hata anında hatalı komuttan öncekiler tamamlanmış, sonrakiler hiç başlamamış görünür.

Linux, sayfa hatalarını sonuçlarına göre üç türe ayırır (Tablo 9.2). Ayrım performans açısından önemlidir, çünkü yalnız büyük hatalar disk G/Ç'si gerektirir.

Tür	Ne olur?
Küçük (*minor*)	Sayfa zaten bellektedir: sayfa önbelleğinde, paylaşımlı kütüphanede, sıfır sayfası ya da yazınca kopyalama sayfası olarak. Yalnız tablo güncellenir, G/Ç yoktur.
Büyük (*major*)	Sayfa diskten ya da takastan okunmalıdır. G/Ç ve bekleme vardır.
Geçersiz	Bölge yoktur ya da izin ihlal edilmiştir. Süreç `SIGSEGV` alır.

Tablo 9.2: Linux'ta sayfa hatası türleri.

Bir programın kaç küçük ve büyük hata yaptığı Linux'ta kolayca ölçülür:

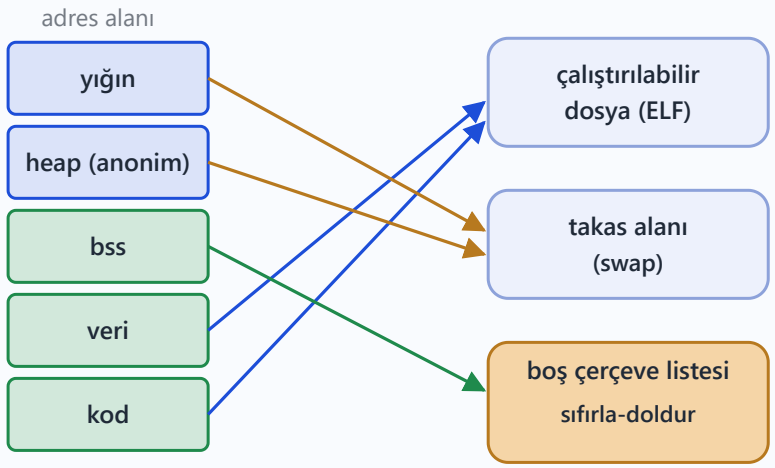
```
$ /usr/bin/time -v ./prog | grep -i fault
$ ps -o min_flt,maj_flt -p PID
$ perf stat -e page-faults,major-faults ./prog
```

`/usr/bin/time -v` programın sonunda "Minor" ve "Major page faults" satırlarını basar. `ps` çalışan bir sürecin sayaçlarını gösterir. `perf stat` ise çekirdeğin olay sayaçlarını okur. Aynı program ikinci kez çalıştırıldığında büyük hata sayısı çoğu zaman sıfıra yakın çıkar: ilk çalıştırmada okunan dosya sayfaları sayfa önbelleğinde kalmıştır ve küçük hatayla eşlenir.

9.2.7 Sayfalar nereden gelir?

Sayfa hatası işleyicisi eksik sayfayı getirmek için önce sayfanın kaynağını bilmelidir. Sayfalar kaynaklarına göre üç gruba ayrılır (Şekil 9.10).

- **Dosya destekli sayfalar**: kod, salt okunur veri ve `mmap` ile eşlenmiş dosyalar. Bellekteki kopya temizse atılır; gerektiğinde dosyadan yeniden okunur.
- **Anonim sayfalar**: heap, yığın ve değiştirilmiş veri. Bunların arkasında bir dosya yoktur. Bellekten atılmadan önce takas alanına yazılmaları gerekir, yoksa içerik kaybolur.
- **Sıfırla doldurulan sayfalar** (*zero-fill-on-demand*): bss bölümünün ve yeni heap sayfalarının ilk erişimi sıfırlanmış bir çerçeveyle karşılanır. Sıfırlama bir güvenlik gereğidir: çerçeve daha önce başka bir sürecin verisini taşıyor olabilir.



Şekil 9.10: Sayfaların kaynağı. Kod ve veri çalıştırılabilir dosyadan (ELF), heap ve yığın takas alanından gelir. bss ve yeni heap sayfaları boş çerçeve listesinden sıfırlanarak verilir.

Hata anında boş çerçeve aramak zaman kaybıdır. Bu yüzden işletim sistemi bir **boş çerçeve havuzu** tutar ve hata işleyicisi oradan bir çerçeve alır. Havuz belirli bir eşiğin altına inince geri kazanım başlar ve sayfalar çıkarılarak havuz doldurulur. Takas alanı ayrı bir disk bölümü ya da bir dosya olabilir; Linux'ta etkin takas alanları `swapon --show` ile listelenir.

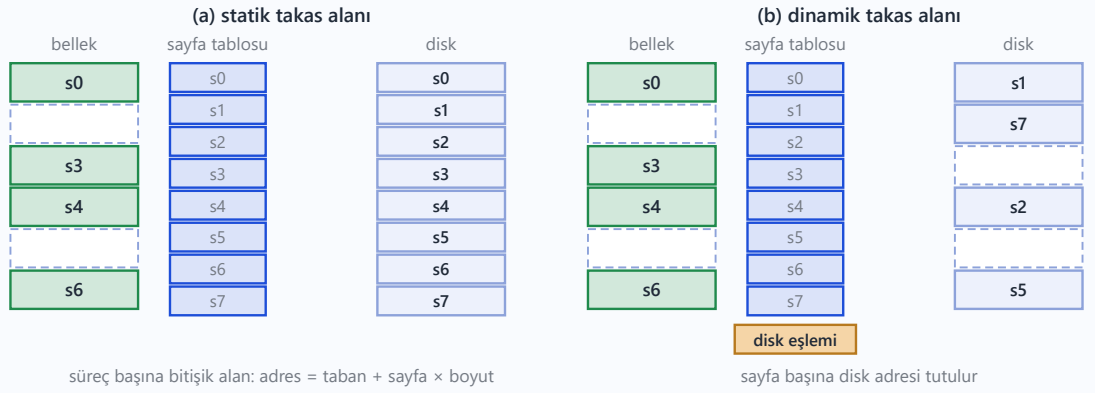
9.2.8 Yedekleme deposu

Tanım 9.5

Yedekleme deposu (*backing store*) ya da **takas alanı** (*swap space*), bellekte olmayan sayfaların diskte tutulduğu alandır.

Takas alanındaki yerin nasıl ayrılacağı konusunda iki yaklaşım vardır (Şekil 9.11).

- **Statik ayırma.** Süreç başlarken bütün boyu kadar bitişik takas alanı ayrılır ve başlangıç adresi süreç tablosuna yazılır. Sayfa k'nın diskteki adresi "taban + k × sayfa boyu" ile hesaplanır. Kod, veri ve yığın için ayrı alanlar ayrılabilir; büyüyebilen yığın bu durumda ayrıca ele alınır.
- **Dinamik ayırma.** Önceden yer ayrılmaz. Sayfa diske çıkarken boş bir yuva bulunur ve adresi sayfa başına tutulan bir **disk eşleminde** saklanır. Değişmemiş kod sayfaları hiç takasa yazılmaz; gerektiğinde çalıştırılabilir dosyadan yeniden okunur.



Şekil 9.11: (a) Statik takas alanında her sayfanın diskte sabit bir kopyası vardır. (b) Dinamik yaklaşımda yer, sayfa diske çıkarken ayrılır ve disk eşleminde tutulur.

Takas alanının yeri de değişebilir. Linux geleneksel olarak ayrı bir takas bölümü kullanır, ama dosya sistemindeki bir takas dosyası da (`swapfile`) olur. Windows sayfaları `pagefile.sys` dosyasına yazar. Ayrı bölüm biraz daha hızlıdır, çünkü dosya sistemi katmanını atlar; dosya ise boyutu kolayca değiştirilebildiği için esnektir.

9.3 Performans: etkin erişim süresi

Talebe bağlı sayfalama bellek erişimini ortalamada yavaşlatır. Erişimlerin çoğu normal hızda yürür, ama arada bir gelen sayfa hatası diske gitmeyi gerektirir. İkisinin ağırlıklı ortalaması sistemin gerçekte ne kadar hızlı olduğunu söyler.

Tanım 9.6

Etkin erişim süresi (EAT, *effective access time*), sayfa hatası olasılığı hesaba katılarak bulunan ortalama bellek erişim süresidir. Bir erişimin sayfa hatası doğurma olasılığı p ($0 \le p \le 1$), bellek erişim süresi ma ise:

$$\text{EAT} = (1 - p) \times ma + p \times \text{sayfa hatası süresi}$$

Sayfa hatası süresi üç parçanın toplamıdır: tuzağı işlemek, sayfayı okumak ve süreci yeniden başlatmak. İlk ve son parça mikrosaniyeler sürer. Okuma ise depolamanın türüne göre değişir ve süreyi belirleyen odur. $p = 0$ ise hiç hata yoktur ve EAT = ma olur; $p = 1$ ise her erişim hatadır.

Örnek 9.2

Bellek erişimi 100 ns, sabit diskten (HDD) sayfa hatası hizmet süresi 25 ms. Bin erişimde bir hata olursa ($p = 0{,}001$) EAT nedir? Bellek kaç kat yavaşlar?

1. **Birimler.** 25 ms = 25.000.000 ns.
2. **Sadeleştirme.** $\text{EAT} = 100(1 - p) + 25.000.000\,p = 100 + 24.999.900\,p$.
3. **Yerine koyma.** $\text{EAT} = 100 + 24.999.900 \times 0{,}001 = 100 + 24.999{,}9 = 25.099{,}9$ ns ≈ 25 µs.
4. **Oran.** 25.099,9 / 100 ≈ 251. Bin erişimde tek bir hata, belleği yaklaşık 251 kat yavaşlatır.

Sonuç çarpıcıdır. Erişimlerin %99,9'u 100 ns'de biter, ama kalan %0,1'in her biri 25 ms sürdüğü için ortalamayı o belirler. Bu yüzden asıl soru tersinden sorulur: belli bir yavaşlamaya razı olursak hata oranı en çok ne olabilir?

Örnek 9.3

Aynı sistemde yavaşlamanın %10'u geçmemesi isteniyor. p en çok ne olabilir?

1. **Hedef.** %10 yavaşlama, EAT < 110 ns demektir.
2. **Eşitsizlik.** $110 > 100 + 24.999.900\,p \rightarrow 10 > 24.999.900\,p$.
3. **Çözüm.** $p < 10 / 24.999.900 \approx 4{,}0 \times 10^{-7}$.
4. **Yorum.** $1 / (4 \times 10^{-7}) = 2.500.000$: yaklaşık 2,5 milyon erişimde birden az hata olmalıdır.

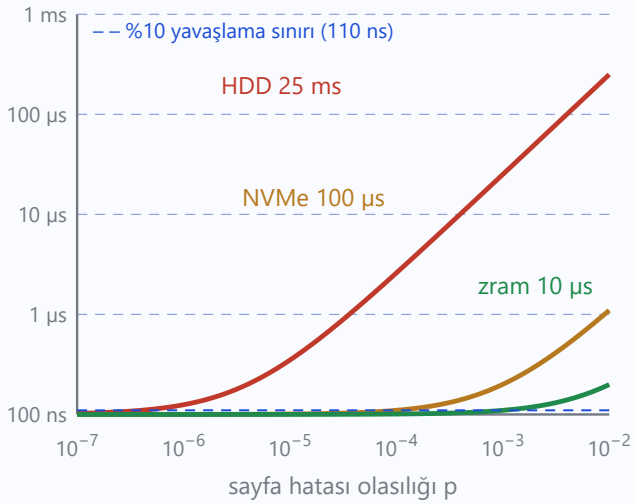
Hata oranı bu kadar düşük tutulabiliyorsa bunun nedeni yine yerelliktir. Ama sonuç, sistemin yer değiştirme kararlarında hata yapmaya hiç tahammülü olmadığını da gösterir.

Depolama teknolojisi değiştikçe hesap da değişir. 2026 itibarıyla NVMe SSD'den bir sayfa okumak 10–100 µs sürer. Linux'un zram mekanizmasında sayfa RAM'de sıkıştırılmış durur; "okuma" bir sıkıştırma çözme işlemidir ve yaklaşık 10 µs sürer (Tablo 9.3, Şekil 9.12).

Depolama	Hata süresi	EAT (p = 0,001)	Yavaşlama	%10 için p sınırı
HDD	25 ms	25.099,9 ns	≈ 251 kat	$4{,}0 \times 10^{-7}$
NVMe SSD	100 µs	199,9 ns	≈ 2 kat	$1{,}0 \times 10^{-4}$
zram	10 µs	109,9 ns	≈ %10	$1{,}0 \times 10^{-3}$

Tablo 9.3: Bellek erişimi 100 ns iken üç depolama türünde EAT.

NVMe satırının hesabı şöyledir: 0,999 × 100 + 0,001 × 100.000 = 99,9 + 100 = 199,9 ns. %10 sınırı ise 10 / 99.900 ≈ $1{,}0 \times 10^{-4}$ çıkar. zram için 99,9 + 0,001 × 10.000 = 109,9 ns'dir.



Şekil 9.12: Sayfa hatası olasılığına göre etkin erişim süresi (iki eksen de logaritmik). HDD, NVMe ve zram eğrileri; kesikli çizgi %10 yavaşlama sınırıdır (110 ns).

Hızlı depolama hata cezasını iki üç büyüklük mertebesi küçültür: HDD'nin 25 ms'si NVMe'nin 100 µs'sinin 250, zram'ın 10 µs'sinin 2.500 katıdır. Yine de bellek hâlâ yüzlerce kat hızlıdır ve hata oranı EAT'yi belirlemeyi sürdürür. Yer değiştirme algoritmalarının bugün de önemli olmasının nedeni budur.

Silberschatz'ın 10. baskısı aynı hesabı 200 ns bellek ve 8 ms hata süresiyle yapar. p = 0,001 için EAT = 200 × 0,999 + 8.000.000 × 0,001 = 199,8 + 8.000 = 8.199,8 ns ≈ 8,2 µs bulunur. Bu, 8.199,8 / 200 ≈ 41 katlık bir yavaşlamadır. %10 sınırı ise 220 > 200 + 7.999.800 p eşitsizliğinden $p < 2{,}5 \times 10^{-6}$ çıkar.

Sık yapılan hata

Birimleri eşitlemeden hesaplamak. 100 ns ile 25 ms aynı formülde toplanırken ms değeri ns'ye çevrilmezse sonuç anlamsız çıkar. p = 0,001 için "%0,1 hata, demek ki %0,1 yavaşlama" sonucu da yanlıştır. Yavaşlamayı hata oranı tek başına değil, hata oranı ile hata cezasının çarpımı belirler.

9.4 Yazınca kopyalama

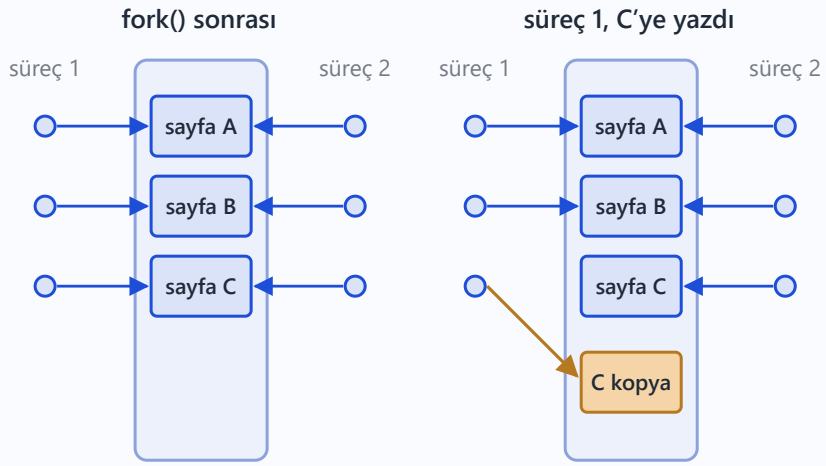
`fork()` ebeveynin bir kopyasını yaratır. Saf hâliyle ebeveynin bütün sayfaları kopyalanmalıdır. Oysa çocukların çoğu yaratıldıktan hemen sonra `exec()` çağırır ve bu kopyaların hepsini çöpe atar. Sanal bellek bu israfı önlemenin yolunu verir.

9.4.1 fork() ve paylaşılan sayfalar

Tanım 9.7

Yazınca kopyalama (COW, *copy-on-write*) yönteminde `fork()` ebeveynin sayfalarını kopyalamaz. Ebeveyn ve çocuk aynı fiziksel sayfaları paylaşır; yazılabilir sayfalar iki tarafta da salt okunur ve COW olarak işaretlenir. Taraflardan biri böyle bir sayfaya yazınca yalnız o sayfanın kopyası çıkarılır.

Mekanizma koruma bitleri üzerine kuruludur. Yazma denemesi salt okunur sayfada bir **koruma hatası** doğurur. Çekirdek girdideki COW işaretini görünce bunun gerçek bir ihlal olmadığını anlar. Boş bir çerçeve alır, sayfanın içeriğini oraya kopyalar ve kopyayı yazana yazılabilir olarak eşler. Ardından yazma komutu yeniden çalışır (Şekil 9.13).



Şekil 9.13: Yazınca kopyalama. Solda `fork()` sonrası: iki süreç A, B ve C sayfalarını paylaşır. Sağda süreç 1 C'ye yazdıktan sonra: yalnız C'nin kopyası çıkarılmıştır, A ve B ortak kalır.

Hiç değiştirilmeyen sayfalar, örneğin kod ve salt okunur veri, iki süreç yaşadıkça paylaşılır. Bu yüzden `fork()` ardından `exec()` çağırmak neredeyse bedavadır: yalnız sayfa tabloları kopyalanır ve `exec()` onları da atar. Paylaşılan birim dosya ya da bölge değil, tek tek sayfalardır; yalnız yazılan sayfa kopyalanır.

Kopya için gereken çerçeve sıfırla doldurulan sayfaların havuzundan alınır. Bir de `vfork()` vardır. Bu çağrı sayfa tablosunu bile kopyalamaz: çocuk ebeveynin adres alanını doğrudan kullanır ve ebeveyn, çocuk `exec()` ya da `_exit()` çağırana kadar bekler. Çocuk bu arada bir değişkeni değiştirirse ebeveyn de değişikliği görür; bu yüzden `vfork()` tehlikelidir. Bugün yeni program başlatmak için `posix_spawn()` tercih edilir.

Paylaşımın doğru yönetilmesi için çekirdek her çerçeve için bir **referans sayısı** tutar: çerçeveyi kaç sayfa tablosu eşliyor. Bir koruma hatasında referans sayısı 1'den büyükse kopya gerekir. Sayı 1 ise sayfa artık yalnız bu sürecindir; kopyalamaya gerek kalmaz, sayfa yazılabilir yapılır.

9.4.2 Kod izleme: fork() ve COW

Aşağıdaki program bir global değişken (veri sayfası D) ve bir heap tamponu (heap sayfası H) kullanır. Çocuk değişkeni okur ve değiştirir, ebeveyn tampona yazar.

```
int x = 1;                /* veri: D */
int main(void) {
  char *b = malloc(4096);  /* heap: H */
  b[0] = 'a';
  pid_t pid = fork();
  if (pid == 0) {          /* çocuk */
    printf("%d\n", x);    /* okuma */
    x = 2;                 /* yazma */
    _exit(0);
  }
  wait(NULL);
  b[0] = 'b';              /* ebeveyn */
  return 0;
}
```

İzleme, iki sürecin sayfa tablolarını ve çerçevelerin referans sayılarını adım adım takip eder (Tablo 9.4).

Adım	Ebeveyn D, H	Çocuk D, H	ref(f1), ref(f2)	Olay
`b[0] = 'a'`	f1 RW, f2 RW	—	1, 1	yalnız ebeveyn var
`fork()`	f1 RO, f2 RO	f1 RO, f2 RO	2, 2	yalnız tablolar kopyalanır
çocuk `printf`	aynı	aynı	2, 2	okuma serbest, hata yok
çocuk `x = 2`	f1 RO, f2 RO	f3 RW, f2 RO	1, 2	koruma hatası; D, f3'e kopyalanır
çocuk `_exit(0)`	f1 RO, f2 RO	—	1, 1	f3 boşa düşer
ebeveyn `b[0] = 'b'`	f1 RO, f2 RW	—	1, 1	hata; ref = 1, kopya yok

Tablo 9.4: `fork()` ve COW izlemesi. f1 D'yi, f2 H'yi tutuyor; f3 boş.

- **Başlangıç.** Ebeveyn D ve H sayfalarını yazılabilir (RW) olarak kullanıyor; `b[0] = 'a'` H'yi çoktan yüklemiştir.
- `fork()`**.** Çekirdek yalnız sayfa tablolarını kopyalar. İki tarafta da D ve H salt okunur (RO) işaretlenir, f1 ve f2'nin referans sayıları 2 olur.
- **Çocuk** `x`**'i okur.** Okuma izinli olduğu için hata da kopya da yoktur. Ekrana 1 basılır.
- **Çocuk** `x = 2` **yazar.** D salt okunur olduğundan koruma hatası oluşur. Çekirdek COW'u tanır: f3'ü ayırır, D'nin içeriğini oraya kopyalar ve çocuğa RW olarak eşler. f1'in referans sayısı 1'e iner. Ebeveynin `x`'i hâlâ 1'dir.
- **Çocuk çıkar.** Eşlemeleri kaldırılır, f3 boş listeye döner, f2'nin referans sayısı da 1'e iner.
- **Ebeveyn H'ye yazar.** H hâlâ salt okunur işaretli olduğu için hata oluşur. Ama f2'nin referans sayısı 1'dir; kopyaya gerek yoktur, sayfa yalnız yazılabilir yapılır.

Program boyunca tek bir sayfa kopyalanmıştır. Ebeveynin D sayfası ise hâlâ RO işaretlidir. Ebeveyn D'ye yazarsa bir hata daha oluşur, ama o da referans sayısı 1 olduğu için kopyasız çözülür. Yöntemin bedeli, yazılan her paylaşılan sayfa için bir tuzak ve bir sayfa kopyalamadır. Kazancı ise hiç yazılma-

yan bütün sayfaların kopyalanmamasıdır.

Örnek 9.4

`fork()` sonrasında ebeveyn ve çocuk 100 sayfayı paylaşıyor. Çocuk bunlardan 3'üne yazıp `exec()` çağırıyor. Kaç sayfa kopyalanır?

1. **Yazılan sayfalar.** Çocuğun her yazması, referans sayısı 2 olan bir sayfaya gider; üç koruma hatası, üç kopya.
2. **Okunan ve hiç dokunulmayan sayfalar.** Kopya üretmez.
3. `exec()`**.** Çocuğun bütün eşlemelerini atar, kopyalar dahil. Ebeveynin sayfalarının referans sayıları 1'e iner.
4. **Sonuç.** 3 sayfa kopyalanır. Saf `fork()` 100 sayfa kopyalardı.

Sık yapılan hata

Her yazmanın kopya ürettiğini sanmak. Kopyayı belirleyen, yazılan çerçevenin o anki referans sayısıdır. Çocuk A sayfasını kopyaladıktan sonra ebeveyn A'ya yazarsa ebeveynin çerçevesi artık yalnız ona aittir ve kopya gerekmez.

9.5 Bellek eşlemeli dosyalar

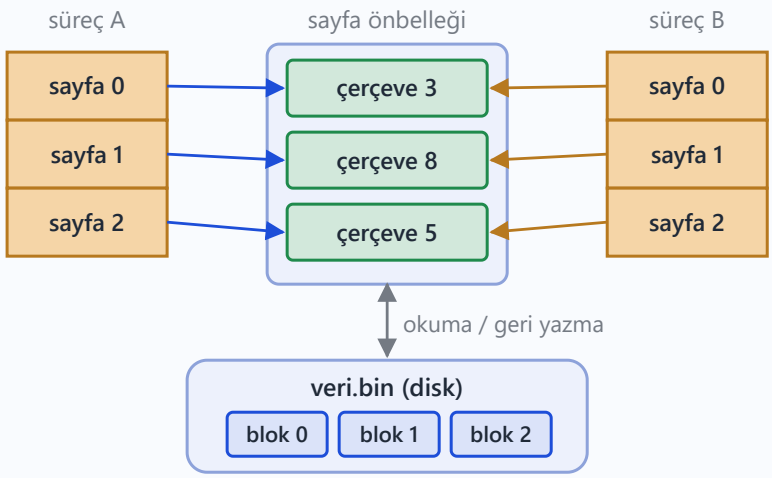
Bir dosyayı okumanın alışılmış yolu `read()` ve `write()` çağrılarıdır. Her çağrı bir sistem çağrısıdır ve veriyi çekirdek tamponu ile kullanıcı tamponu arasında kopyalar. Sanal bellek ikinci bir yol açar: dosyayı adres alanına eşlemek ve bellek gibi kullanmak.

9.5.1 mmap() ve eşleme türleri

Tanım 9.8

Bellek eşlemeli dosya (*memory-mapped file*), sayfaları sürecin adres alanının bir bölgesine eşlenmiş dosyadır. POSIX'te eşlemeyi `mmap()` kurar. Dosya bundan sonra okuma ve yazma çağrılarıyla değil, sıradan bellek erişimiyle okunup yazılır.

Eşlemenin kendisi hiçbir şey okumaz; yalnız bir VMA oluşturur. Sayfalar talebe bağlı getirilir: her sayfaya ilk erişim bir sayfa hatası üretir. Sayfa sayfa önbelleğinde zaten varsa hata küçüktür, yoksa büyüktür. Yazılan sayfalar kirli olur ve arka planda ya da `msync()` veya `munmap()` çağrısıyla dosyaya yazılır (Şekil 9.14).



Şekil 9.14: İki süreç aynı dosyayı eşliyor. Sanal sayfalar sayfa önbelleğindeki aynı çerçevelere bakar; çerçeveler gerektiğinde diskteki dosya bloklarından okunur ve geri yazılır.

`mmap()` 'in bayrakları eşlemenin nasıl davranacağını belirler:

- `MAP_SHARED`: değişiklikler dosyaya ve aynı dosyayı eşleyen herkese görünür. İki süreç bu yolla paylaşımlı bellek kurup haberleşebilir.
- `MAP_PRIVATE`: yazmalar yazınca kopyalamayla özel bir kopyaya gider, dosya değişmez. Çalıştırılabilir dosyaların veri bölümü böyle yüklenir: program ilk değerleri dosyadan alır, değiştirdiklerini kendine saklar.
- `MAP_ANONYMOUS`: arkasında dosya olmayan, sıfırla doldurulan bir bölge. `malloc()` büyük blokları bu yolla ayırır.

Windows'ta aynı işi `CreateFileMapping` ile oluşturulan eşleme nesnesi ve `MapViewOfFile` çağrısı yapar.

Bellek eşlemenin üstünlüğü, kopyanın ortadan kalkmasıdır. Sayfa önbelleğindeki çerçeve doğrudan sürecin adres alanına eşlenir; çekirdek tamponundan kullanıcı tamponuna ikinci bir kopya yapılmaz. Ayrıca sık okunan küçük parçalar için sistem çağrısı maliyeti de ortadan kalkar. Bedeli, her yeni sayfanın bir sayfa hatası üretmesi ve G/Ç hatalarının sinyal olarak (`SIGBUS`) gelmesidir.

9.5.2 Kod izleme: mmap

Aşağıdaki program 12 KiB'lık bir dosyayı paylaşımlı ve yazılabilir olarak eşler, bir bayt okur, bir bayt yazar, değişikliği diske aktarır ve eşlemeyi kaldırır.

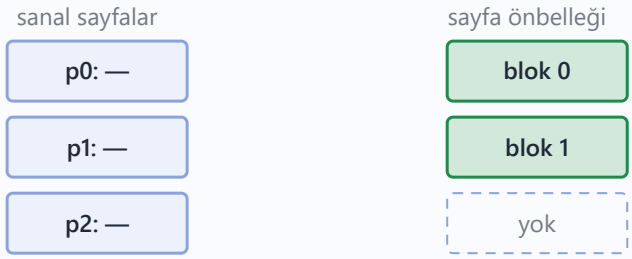
```c
int fd = open("veri.bin", O_RDWR);
struct stat st;
fstat(fd, &st);          /* 12 KiB = 3 sayfa */
char *p = mmap(NULL, st.st_size,
     PROT_READ | PROT_WRITE,
     MAP_SHARED, fd, 0);
close(fd);               /* eşleme sürer */
char c = p[0];           /* sayfa 0 */
p[5000] = 'x';           /* sayfa 1 */
msync(p, st.st_size, MS_SYNC);
munmap(p, st.st_size);
```

İzleme sayfa tablosu girdilerini ve sayfa önbelleğini birlikte takip eder (Tablo 9.5).

Satır	p0	p1	p2	Sayfa önbelleği	Olay
`open`, `fstat`	—	—	—	boş	boyut 12 KiB = 3 sayfa
`mmap`	boş	boş	boş	boş	yalnız VMA kurulur
`close(fd)`	boş	boş	boş	boş	eşleme sürer
`c = p[0]`	v	boş	boş	blok 0	hata, blok 0 okunur
`p[5000] = 'x'`	v	v, M = 1	boş	blok 0, blok 1	hata, okuma, yazma
`msync`	v	v, M = 0	boş	blok 0, blok 1	blok 1 dosyaya yazılır
`munmap`	—	—	—	blok 0, blok 1	VMA kaldırılır

Tablo 9.5: mmap izlemesi; dosya 3 sayfa (4 KiB'lık p0, p1, p2).

- **Açma.** Dosya açılır, `fstat` boyutun 12 KiB, yani 3 sayfa olduğunu söyler.
- `mmap`**.** Yalnız bir VMA oluşturulur; tablo girdileri boştur ve hiçbir sayfa okunmamıştır.
- `close(fd)`**.** Eşlemeyi bozmaz, çünkü VMA dosyaya kendi referansını tutar.
- `p[0]`**.** Sayfa 0'a ilk erişim bir sayfa hatası doğurur. Blok 0 sayfa önbelleğine okunur (büyük hata) ve girdi geçerli olur.
- `p[5000]`**.** 5000 / 4096 = 1 (tam bölme), yani sayfa 1; bağıl konum 5000 − 4096 = 904'tür. Yine hata ve okuma olur, ardından yazma sayfayı kirli yapar (M = 1).
- `msync`**.** `MS_SYNC` bayrağıyla kirli sayfa dosyaya yazılır ve çağrı yazma bitene kadar dönmez; M temizlenir. Sayfa 2'ye hiç dokunulmadığı için hiç okunmamıştır.
- `munmap`**.** VMA kaldırılır. Sayfalar sayfa önbelleğinde kalabilir; başka bir süreç dosyayı açarsa onları küçük hatayla kullanır.



Şekil 9.15: `munmap` sonrası: sanal sayfaların eşlemesi kalkmıştır, ama blok 0 ve blok 1 sayfa önbelleğinde durur. Blok 2 hiç okunmamıştır.

Sık yapılan hata

`MAP_PRIVATE` ile yazılan verinin `munmap` anında dosyaya gideceğini sanmak. Özel eşlemede ilk yazma bir koruma hatası üretir ve çekirdek sayfanın özel bir kopyasını çıkarır. Değişiklik yalnız o sürecin adres alanındadır ve `munmap` ile kaybolur. Aynı dosyayı `MAP_SHARED` ile eşlemiş başka bir süreç de bu yazmayı hiç görmez.

9.6 Sayfa yer değiştirme

Talebe bağlı sayfalamanın anlatımında her sayfa hatasında boş bir çerçeve bulunduğu varsayıldı. Gerçek sistemlerde bellek çoğu zaman doludur. Bunun nedeni israf değil, tam tersidir: boş duran RAM boşa harcanmış RAM'dir. İşletim sistemi boş belleği sayfa önbelleği olarak kullanır ve ancak gerektiğinde geri alır. Bu yüzden hata anında çoğunlukla bir sayfa çıkarılmalıdır.

9.6.1 Boş çerçeve yoksa

Boş çerçeve yokken işletim sisteminin üç seçeneği vardır. Süreci sonlandırmak kabul edilemez: sanal bellek kullanıcıya görünmemelidir. Bütün bir süreci takas alanına atmak mümkündür, ama ağır bir önlemdir ve aşırı yüklenmede kullanılır. Olağan yol tek bir sayfanın yerini değiştirmektir.

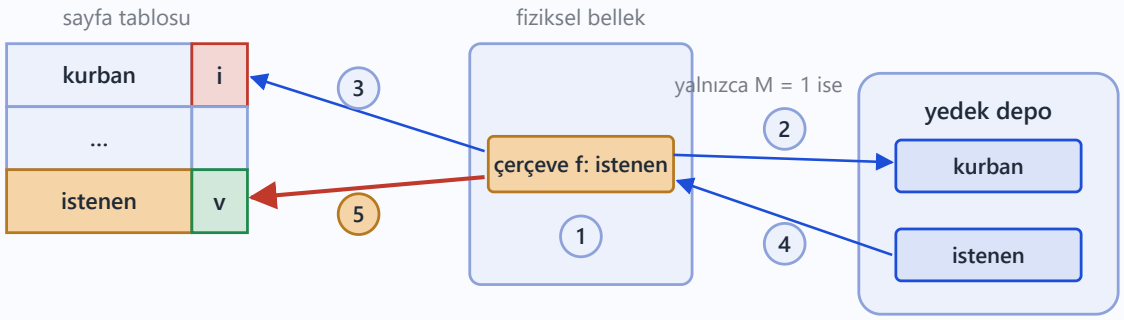
Tanım 9.9

Sayfa yer değiştirme (*page replacement*), boş çerçeve yokken bellekteki bir sayfayı (**kurban**, *victim*) seçip çıkarmak ve onun çerçevesine istenen sayfayı yüklemektir. Kurbanı seçen kurala **yer değiştirme algoritması** denir.

Yer değiştirme, sayfa hatası işleyicisine beş adım ekler (Şekil 9.16):

1. Yer değiştirme algoritması bir kurban çerçeve seçer.

2. Kurban kirliyse (M = 1) takas alanına ya da dosyasına yazılır. Temizse yazmaya gerek yoktur; diskteki kopya zaten günceldir. Böylece G/Ç yarıya iner.
3. Kurbanın girdisi geçersiz yapılır ve eşlemesi TLB'den silinir. Çok çekirdekli sistemde diğer çekirdeklerin TLB'leri de temizlenmelidir; buna **TLB vurusu** (*TLB shootdown*) denir.
4. İstenen sayfa boşalan çerçeveye okunur.
5. Yeni sayfanın girdisi geçerli yapılır ve tuzağa yol açan komut yeniden başlatılır.



Şekil 9.16: Sayfa yer değiştirmenin adımları: kurban seçilir, kirliyse diske yazılır, girdisi geçersiz yapılır, istenen sayfa boşalan çerçeveye okunur ve yeni girdi geçerli yapılır.

M biti burada önemli bir tasarruf sağlar. Kirli kurbanın bedeli iki disk işlemidir: bir yazma ve bir okuma. Temiz kurbanınki yalnız okumadır. Bu yüzden birçok algoritma, aynı derecede eski iki sayfadan temiz olanı tercih eder.

Sanal bellek sistemi iki ayrı soruyu yanıtlamalıdır. **Çerçeve tahsisi**: bir sürece kaç çerçeve verilecek? **Yer değiştirme**: hangi sayfa çıkacak? Bu kesim ikinci soruyla ilgilenir. İki sorunun amacı aynıdır: sayfa hatası sayısını en aza indirmek ve aynı sayfanın bellek ile disk arasında tekrar tekrar gidip gelmesini önlemek.

9.6.2 Algoritmaları değerlendirmek

Bir algoritmayı değerlendirmek için onu bir dizi bellek erişimi üzerinde çalıştırır ve sayfa hatalarını sayarız. Gerçek adres dizileri milyonlarca öğe içerir, ama algoritmanın kararı yalnız sayfa numarasına bağlıdır. Bu yüzden diziyi kısaltmak mümkündür.

Tanım 9.10

Referans dizisi (*reference string*), bir sürecin eriştiği sayfa numaralarının sırasıdır. Dizi bir adres izinden elde edilir: her adres sayfa numarasına çevrilir ve art arda gelen aynı sayfalar tek sayılır. Bir sayfaya hemen ardından yapılan erişim hiçbir zaman hata doğurmadığı için bu kısaltma sonucu değiştirmez.

Örnek 9.5

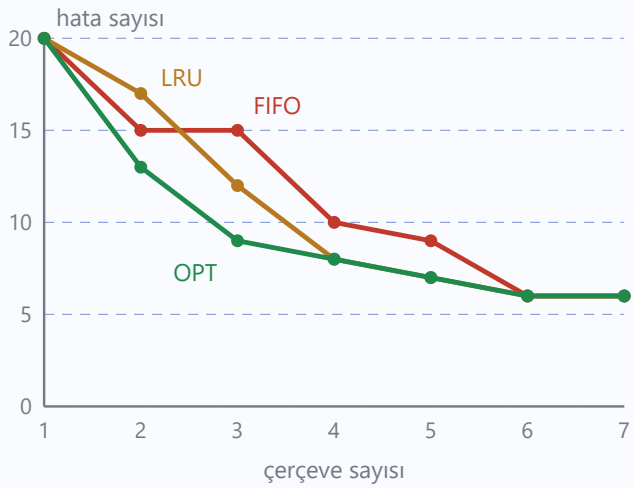
Sayfa boyutu 100 bayt. Adres izi 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, ... biçiminde ilerliyor. Referans dizisi nedir?

1. **Sayfa numaraları.** Adresi 100'e bölmek yeter: 1, 4, 1, 6, 1, 1, 1, 1, 6, 1, ...
2. **Art arda tekrarları atmak.** 0102, 0103, 0104, 0101 hepsi sayfa 1'dedir ve tek erişim sayılır.
3. **Sonuç.** Silberschatz'ın izinin tamamı 1 4 1 6 1 6 1 6 1 6 1 dizisine iner.

Bu notta algoritmalar Silberschatz'ın ortak dizisi üzerinde, 3 çerçeveyle çalıştırılır:

> 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Dizide 20 referans ve 6 farklı sayfa (7, 0, 1, 2, 3, 4) vardır. Her farklı sayfanın ilk yüklenmesi kaçınılmaz bir hatadır; dolayısıyla hiçbir algoritma 6'nın altına inemez. Çerçeve sayısı arttıkça hata sayısının azalması beklenir (Şekil 9.17). Çerçeve sayısı 6'ya ulaşınca bütün sayfalar sığar ve yalnız ilk yüklemeler kalır.



Şekil 9.17: Ortak dizide çerçeve sayısına göre sayfa hatası. 1'den 7'ye kadar çerçeve için FIFO 20, 15, 15, 10, 9, 6, 6; LRU 20, 17, 12, 8, 7, 6, 6; OPT 20, 13, 9, 8, 7, 6, 6 hata yapar.

Eğrideki değerler algoritmalar çalıştırılarak denetlenmiştir. FIFO'nun 2 ve 3 çerçevede aynı (15) kalması dikkat çekicidir: bir çerçeve eklemek bu dizide hiçbir şey kazandırmaz. Bölümde çalıştırılacak algoritmalar şunlardır: OPT, FIFO, LRU, ikinci şans, saat, NRU, NFU ve yaşlandırma, çalışma kümesi ve WSClock. Farklı dizilerle denemek için Sayfa Yer Değiştirme Simülatörü FIFO, OPT, LRU, LFU ve ikinci şansı adım adım çalıştırır.

9.6.3 FIFO

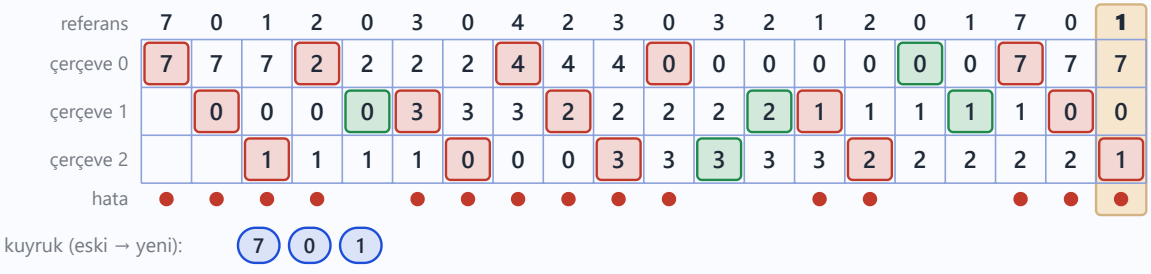
Tanım 9.11

FIFO (*first-in, first-out*, ilk giren ilk çıkar) algoritmasında sayfalar yüklenme sırasına göre bir kuyrukta tutulur. Yeni yüklenen sayfa kuyruğun sonuna eklenir. Yer gerekince kuyruğun başındaki, yani en önce yüklenmiş sayfa çıkar.

FIFO'nun gerçeklemesi çok kolaydır ve hiçbir donanım biti gerektirmez. Sorunu, sayfanın ne zaman *kullanıldığını* hiç hesaba katmamasıdır. En eski sayfa, en yoğun kullanılan sayfa olabilir: başlangıçta yüklenen bir değişken bütün çalışma boyunca kullanılıyorsa FIFO onu yine de ilk fırsatta atar.

Ortak dizideki izleme şöyledir (Şekil 9.18). İlk üç referans (7, 0, 1) boş çerçevelere yüklenir: 3 hata. Kuyruk sırası 7, 0, 1'dir.

- **Referans 4 (sayfa 2).** Hata. Kuyruğun başındaki 7 çıkar, 2 sona girer. Kuyruk: 0, 1, 2. Hata sayısı 4.
- **Referans 5 (0).** Bellekte, isabet. FIFO'da isabet kuyruğu değiştirmez.
- **Referans 6 (3).** Hata; 0 çıkar. Kuyruk: 1, 2, 3. Hata 5.
- **Referans 7 (0).** Hata; 1 çıkar. 0 bir adım önce atılmıştı ve hemen geri istendi. Kuyruk: 2, 3, 0. Hata 6.
- **Referans 8–11 (4, 2, 3, 0).** Dördü de hata. Sırasıyla 2, 3, 0 ve 4 çıkar. Her biri bir sonraki adımlarda yeniden gerekecektir. Hata 10, kuyruk: 2, 3, 0.
- **Referans 12, 13 (3, 2).** İsabet.
- **Referans 14, 15 (1, 2).** Hata; 2 ve 3 çıkar. Hata 12, kuyruk: 0, 1, 2.
- **Referans 16, 17 (0, 1).** İsabet.
- **Referans 18–20 (7, 0, 1).** Üçü de hata; 0, 1 ve 2 çıkar. Toplam 15 hata.



Şekil 9.18: FIFO, ortak dizi, 3 çerçeve. Kırmızı hücre hatayla yüklenen sayfayı, yeşil hücre isabeti, alt satırdaki nokta sayfa hatasını gösterir. Toplam 15 hata: 3 ilk yükleme ve 12 yer değiştirme.

Kurbanların sırası 7, 0, 1, 2, 3, 0, 4, 2, 3, 0, 1, 2'dir. Bu sıra tam olarak yüklenme sırasıdır; FIFO erişimi değil yüklenmeyi izler. Algoritmanın her adımı O(1)'dir: bir sayfanın bellekte olup olmadığı bir hash tablosuyla, kurban da kuyruğun başından bulunur.

9.6.4 Belady anomalisi

Çerçeve sayısı arttıkça sayfa hatasının azalması doğal görünür. 1969'da Belady, Nelson ve Shedler bunun FIFO için her zaman doğru olmadığını gösterdi.

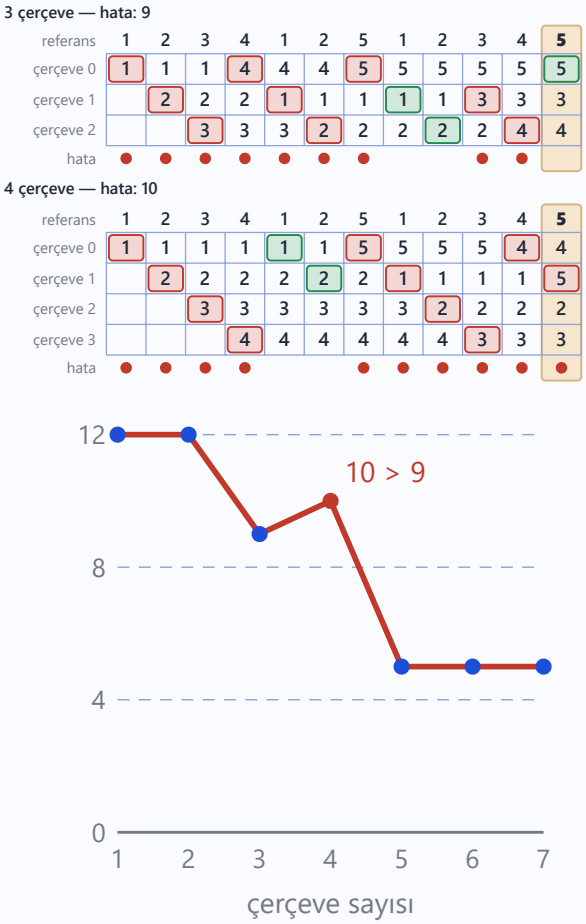
Tanım 9.12

Belady anomalisi (*Belady's anomaly*), bazı yer değiştirme algoritmalarında çerçeve sayısı artınca sayfa hatası sayısının da artabilmesidir.

Örnek 9.6

Referans dizisi 1 2 3 4 1 2 5 1 2 3 4 5. FIFO 3 ve 4 çerçeveyle kaçar hata yapar?

1. **3 çerçeve, referans 1–7.** 1, 2, 3 boşlara yüklenir. 4 gelince 1 çıkar, 1 gelince 2 çıkar, 2 gelince 3 çıkar, 5 gelince 4 çıkar. Yedi referansın hepsi hata.
2. **3 çerçeve, referans 8–12.** Bellekte {5, 1, 2} vardır. 1 ve 2 isabettir. 3 gelince 1, 4 gelince 2 çıkar. Son 5 isabettir. Toplam 9 hata.
3. **4 çerçeve, referans 1–6.** 1, 2, 3, 4 boşlara yüklenir; 1 ve 2 isabettir. 4 hata.
4. **4 çerçeve, referans 7–12.** 5 gelince en eski 1 çıkar. Ardından gelen 1, 2'yi; 2, 3'ü; 3, 4'ü; 4, 5'i; 5 de 1'i atar. Altısı da hata. Toplam 10 hata.
5. **Sonuç.** 4 çerçeve 10, 3 çerçeve 9 hata yapar: daha çok bellek, daha çok hata.



Şekil 9.19: Belady anomalisi. Solda aynı dizinin 3 ve 4 çerçeveli FIFO tabloları; sağda çerçeve sayısına göre hata sayısı. 3 çerçevede 9, 4 çerçevede 10 hata.

Anomalinin nedeni, FIFO'da iki çerçeve sayısındaki bellek içeriklerinin birbirinden kopmasıdır. 7. referanstan sonra 3 çerçevede {1, 2, 5}, 4 çerçevede {2, 3, 4, 5} bulunur. Küçük bellekte olan 1, büyük bellekte yoktur. Sıradaki referanslar tam da 1 ve 2 olduğu için küçük bellek kazanır.

Bazı algoritmalarda bu kopma hiç yaşanmaz.

Teorem 9.1 (Yığın algoritmaları)

Bir algoritmada her an n çerçevedeki sayfa kümesi $n + 1$ çerçevedekinin alt kümesi olsun. Böyle bir algoritmaya **yığın algoritması** (*stack algorithm*) denir. Yığın algoritmaları Belady anomalisi göstermez: $n + 1$ çerçevedeki her hata n çerçevede de hatadır.

LRU bir yığın algoritmasıdır. n çerçeveli LRU belleğinde son kullanılan n farklı sayfa durur. $n + 1$ çerçevelide son kullanılan $n + 1$ farklı sayfa durur ve ilki her zaman ikincinin içindedir. OPT de aynı özelliği taşır. Aynı dizide LRU 3 çerçeveyle 10, 4 çerçeveyle 8 hata yapar. FIFO ise yığın algoritması değildir; kümelerin bozulduğu adımlar bu örnekte 7., 8. ve 11. referanslardır.

Sık yapılan hata

Belady anomalisini "FIFO her zaman çerçeve eklendikçe kötüleşir" diye anlamak. Anomali yalnız belirli dizilerde ve belirli çerçeve sayıları arasında görülür. Aynı dizide FIFO 1, 2, 3, 4, 5 çerçeveyle 12, 12, 9, 10, 5 hata yapar; genel eğilim yine düşüştür. Anomali, FIFO'nun bu düşüşü *garanti edemediğini* söyler.

9.6.5 OPT

Tanım 9.13

Optimal algoritma (OPT ya da MIN, *optimal page replacement*), gelecekte en geç kullanılacak ya da hiç kullanılmayacak sayfayı çıkarır. Belady 1966'da bu algoritmanın verilen bir dizi ve çerçeve sayısı için en az hatayı verdiğini gösterdi.

OPT'nin tanımı kolaydır, ama gerçeklenmesi imkânsızdır: işletim sistemi bir sürecin gelecekte hangi sayfaya erişeceğini bilemez. Bu yüzden OPT'ye "kristal küre" algoritması da denir. Yine de çok değerlidir, çünkü bir **alt sınır** verir. Yeni bir algoritmanın OPT'ye ne kadar yaklaştığı onun ne kadar iyi olduğunu söyler. Gerçek bir sistemde OPT ancak bir iz kaydedilip ikinci kez çalıştırılarak hesaplanabilir; ilk çalıştırma geleceği öğrenmek içindir.

Ortak dizideki izleme kurban seçimlerine odaklanır. Her hatada bellekteki sayfaların bir sonraki kullanım anı (referans numarası) karşılaştırılır (Tablo 9.6).

Referans no	İstenen	Sonraki kullanımlar	Kurban
4	2	7 → 18, 0 → 5, 1 → 14	7
6	3	2 → 9, 0 → 7, 1 → 14	1
8	4	2 → 9, 0 → 11, 3 → 10	0
11	0	2 → 13, 4 → yok, 3 → 12	4
14	1	2 → 15, 0 → 16, 3 → yok	3
18	7	2 → yok, 0 → 19, 1 → 20	2

Tablo 9.6: OPT'nin yer değiştirme anları. Sağdaki sütun bellekteki her sayfanın bir sonraki kullanıldığı referans numarasıdır.

Referans 4'te bellekte 7, 0 ve 1 vardır. 0 hemen 5. referansta, 1 ise 14.'de gerekir; 7 ancak 18.'de gerekecektir. En uzağa kalan 7 çıkar. Referans 11'de sayfa 4 bir daha hiç kullanılmayacaktır; böyle bir sayfa varsa hemen kurban olur. Altı yer değiştirme ile 3 ilk yükleme toplamı 9 hata verir (Şekil 9.20). Bu sayı, bu dizi ve 3 çerçeve için ulaşılabilecek en iyi değerdir.



Şekil 9.20: OPT, ortak dizi, 3 çerçeve. Toplam 9 hata (3 ilk yükleme, 6 yer değiştirme): bu dizi için alt sınır.

9.6.6 LRU

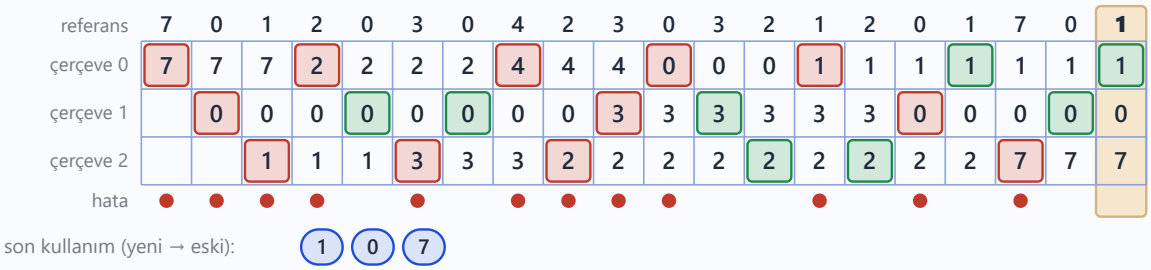
OPT geleceğe bakar. Gelecek bilinemiyorsa en iyi tahmin yakın geçmiştir: yerellik ilkesine göre yakın geçmişte kullanılmayan bir sayfa yakın gelecekte de büyük olasılıkla kullanılmayacaktır. LRU bu fikri kurala dönüştürür; bir anlamda geriye bakan OPT'dir.

Tanım 9.14

LRU (*least recently used*, en uzun süredir kullanılmayan) algoritması, en uzun süredir erişilmemiş sayfayı çıkarır. Her sayfa için son kullanım anı tutulur ve kurban, son kullanım anı en eski olan sayfadır.

Ortak dizideki izleme FIFO'yla aynı başlar, ama isabetler artık sırayı değiştirir (Şekil 9.21).

- **Referans 1–3.** 7, 0, 1 boşlara yüklenir. 3 hata.
- **Referans 4 (2).** En uzun süredir kullanılmayan 7 çıkar. Hata 4.
- **Referans 5 (0).** İsabet; 0 artık en yeni sayfadır.
- **Referans 6 (3).** Bellekte 1, 2, 0 vardır; son kullanımları 3, 4 ve 5'tir. 1 çıkar. Hata 5. FIFO burada 0'ı atmıştı; LRU, 0'ın az önce kullanıldığını bilir.
- **Referans 7 (0).** İsabet. FIFO'nun hata yaptığı yerde LRU kazanır.
- **Referans 8–11 (4, 2, 3, 0).** Dördü de hata; sırayla 2, 3, 0 ve 4 çıkar. Hata 9. Bu dört adımda dizi her seferinde bir önce atılan sayfayı geri ister ve LRU da kötü sonuç verir.
- **Referans 12, 13 (3, 2).** İsabet.
- **Referans 14 (1).** 0 çıkar. Hata 10. **Referans 15 (2)** isabet.
- **Referans 16 (0).** 3 çıkar. Hata 11. **Referans 17 (1)** isabet.
- **Referans 18 (7).** 2 çıkar. Hata 12. **Referans 19, 20 (0, 1)** isabet.



Şekil 9.21: LRU, ortak dizi, 3 çerçeve. Toplam 12 hata (9 yer değiştirme): FIFO'dan (15) iyi, OPT'den (9) kötü.

LRU uygulamada çok iyi sonuç verir ve birçok algoritmanın yaklaşmaya çalıştığı hedeftir. Ama her bellek erişiminde bir kaydın güncellenmesini ister. Saniyede milyarlarca erişim yapan bir işlemcide bu iş donanım desteği olmadan yapılamaz. Bir uyarı da adlarla ilgilidir: en *son* kullanılanı atan algoritmanın adı MRU'dur (*most recently used*). LRU ise en *eski* kullanılanı atar.

9.6.7 LRU'yu gerçeklemek: sayaç ve yığın

Tam LRU için iki klasik gerçekleme vardır.

Sayaç (zaman damgası). CPU'ya her bellek erişiminde artan mantıksal bir saat eklenir. Bir sayfaya her erişimde, saatin değeri o sayfanın tablo girdisine yazılır. Kurban, damgası en küçük olan sayfadır. Bu yöntemde güncelleme basit ama sıktır: her bellek erişimi sayfa tablosuna bir yazma demektir. Seçim ise pahalıdır: her hatada bütün tablo taranmalıdır. Saat taşmasına da çözüm gerekir.

Yığın. Sayfa numaraları bir yığında tutulur. Bir sayfaya erişilince sayfa yığının ortasından çıkarılır ve tepeye konur. Tepede en son kullanılan, en altta en uzun süredir kullanılmayan sayfa durur; kurban en alttakidir. Yığın, başı ve sonu işaretçili bir çift bağlı listeyle gerçeklenir. Ortadan bir düğümü çıkarıp başa eklemek en fazla altı işaretçi değişikliği ister. Kurban kuyruktan O(1)'de alınır ve arama gerekmez.

Örnek 9.7

Referans dizisi 4 7 0 7 1 0 1 2 1 2 7 1 2. LRU yığını her adımda nasıl değişir? Dizinin sonunda bir yer değiştirme gerekse hangi sayfa çıkar?

1. **4, 7, 0.** Üçü de yığında yoktur; sırayla tepeye itilir. Yığın (tepeden alta): 0, 7, 4.
2. **7.** Yığında vardır; ortadan çıkarılıp tepeye konur: 7, 0, 4.
3. **1.** Yeni; tepeye: 1, 7, 0, 4.
4. **0, 1.** İkisi de tepeye taşınır: önce 0, 1, 7, 4; sonra 1, 0, 7, 4.
5. **2.** Yeni; tepeye: 2, 1, 0, 7, 4.
6. **1, 2, 7, 1, 2.** Hepsi yığında vardır ve tepeye taşınır. Son durum: 2, 1, 7, 0, 4.

7. **Sonuç.** En altta 4 vardır: 4 ilk referanstan beri hiç kullanılmamıştır. Bir yer değiştirmede o çıkar.



Şekil 9.22: LRU yığınının dizinin sonundaki durumu: tepede 2, en altta 4. Çift bağlı listede erişim, düğümü çıkarıp başa eklemektir; kurban kuyruktan O(1)'de alınır.

Yığın yönteminde güncelleme pahalı, seçim ucuzdur; sayaç yönteminde tersi. İkisi de her bellek erişiminde iş ister ve bugün hiçbir genel amaçlı işlemcinin donanımında bulunmaz. Pratikte LRU'ya yaklaşan, R bitine dayalı algoritmalar kullanılır.

9.6.8 LRU'yu gerçeklemek: matris yöntemi

Tanenbaum küçük çerçeve sayıları için zarif bir donanım çözümü anlatır. n çerçeveli bir makinede donanım $n \times n$ bitlik bir matris tutar. Başlangıçta bütün bitler 0'dır.

Tanım 9.15

LRU matris yöntemi: k çerçevesine erişilince önce k. satırın bütün bitleri 1, ardından k. sütunun bütün bitleri 0 yapılır. Her an satırların ikili değeri ne kadar küçükse o çerçeve o kadar uzun süredir kullanılmamıştır. Kurban, ikili değeri en küçük satırdır.

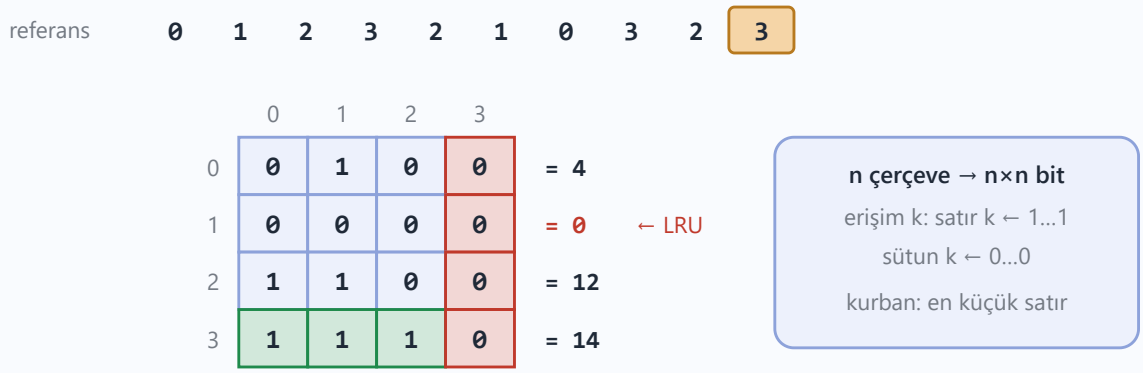
Yöntemin neden çalıştığını görmek kolaydır. k'ya erişim, k. satırı "herkesten daha yeni" yapar: k. satırda kendi sütunu dışında her bit 1'dir. Aynı anda diğer her satırın k. biti 0 olur, yani "k'dan daha yeni değilim" bilgisi yazılır. Satır i'nin j. biti 1 ise i, j'den daha yakın zamanda kullanılmıştır.

Örnek 9.8

4 çerçeve; erişim sırası 0 1 2 3 2 1 0 3 2 3. Her erişimden sonra satırların ikili değerleri ve LRU çerçeve nedir?

1. **0.** Satır 0 = 1111, sonra sütun 0 temizlenir: satır 0 = 0111 = 7. Diğer satırlar 0'dır. Değerler 7, 0, 0, 0; en küçükler 1, 2, 3 eşittir ve en küçük numaralı 1 seçilir.

2. **1.** Satır 1 = 1011 = 11; satır 0'ın 1. biti silinir: 0011 = 3. Değerler 3, 11, 0, 0 → LRU 2.
3. **2.** Satır 2 = 1101 = 13; satır 0 = 0001 = 1, satır 1 = 1001 = 9. Değerler 1, 9, 13, 0 → LRU 3.
4. **3.** Satır 3 = 1110 = 14; diğerlerinin son biti silinir: 0, 8, 12. Değerler 0, 8, 12, 14 → LRU 0.
5. **Kalan erişimler.** 2 → 0, 8, 13, 12 (LRU 0); 1 → 0, 11, 9, 8 (LRU 0); 0 → 7, 3, 1, 0 (LRU 3); 3 → 6, 2, 0, 14 (LRU 2); 2 → 4, 0, 13, 12 (LRU 1); 3 → 4, 0, 12, 14 (LRU 1).



Şekil 9.23: On erişimden sonra matris. Satır değerleri 4, 0, 12, 14; en küçük satır 1 olduğu için LRU çerçeve 1'dir. Gerçekten de 1 en son altıncı erişimde kullanılmıştı.

Son durumu doğrulamak için kullanım sırasına bakmak yeter. Son erişimler sırasıyla 3 (onuncu), 2 (dokuzuncu), 0 (yedinci) ve 1'dir (altıncı). Satır değerlerinin büyükten küçüğe sırası 14, 12, 4, 0, yani 3, 2, 0, 1'dir. Matris tam LRU sırasını kodlar. Bedeli n^2 bittir: 4 çerçevede 16 bit sorun değildir, ama milyonlarca çerçeveli bir makinede yöntem kullanılamaz.

9.6.9 İkinci şans

Tam LRU donanımı yoktur, ama hemen bütün işlemciler R bitini tutar. R biti, sayfanın son temizlendiğinden beri kullanılıp kullanılmadığını söyler. Kullanım sırasını vermez, ama "yakın zamanda kullanıldı mı?" sorusunu yanıtlar. FIFO'nun en büyük kusuru da tam bu soruyu sormamasıdır.

Tanım 9.16

İkinci şans (*second chance*) algoritması FIFO kuyruğunun başındaki sayfanın R bitine bakar. R = 0 ise sayfa hem eski hem kullanılmamıştır ve hemen atılır. R = 1 ise bit temizlenir, sayfa yeni yüklenmiş gibi kuyruğun sonuna taşınır ve arama sürer.

Örnek 9.9

Tanenbaum'un örneğinde A'dan H'ye sekiz sayfa FIFO sırasındadır; yüklenme zamanları 0, 3, 7, 8, 12, 14, 15, 18'dir. R bitleri A = 1, B = 0, C = 1, D = 0, E = 1, F = 0, G = 1, H = 1. Zaman 20'de bir sayfa hatası oluşuyor. Hangi sayfa çıkar?

1. **Baştaki A.** R = 1: yakın zamanda kullanılmıştır. Atılmaz; R'si 0 yapılır ve yükleme zamanı 20'ymiş gibi kuyruğun sonuna taşınır.
2. **Yeni baş B.** R = 0: hem eski hem kullanılmamış. Kurban budur.
3. **Sonuç.** B atılır, yeni sayfa I kuyruğun sonuna R = 1 ile eklenir. Kuyruk artık C, D, E, F, G, H, A, I'dir (Şekil 9.24).



Şekil 9.24: İkinci şans (Tanenbaum Şekil 3-15'in yeniden çizimi). Üstte zaman 20'deki hatadan önceki FIFO listesi, altta hatadan sonra: A ikinci şansla sona taşınmış, B atılmış, I eklenmiştir.

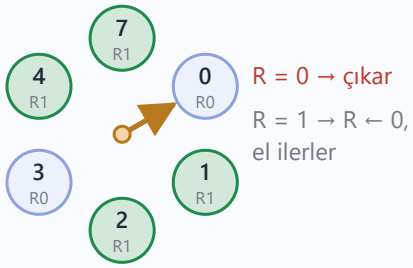
Bütün sayfaların R biti 1 ise ne olur? Algoritma bir tur döner, her sayfanın R'sini temizleyip onu sona taşır. Tur bitince baştaki sayfa yine ilk sayfadır ve R'si artık 0'dır; o atılır. Bu durumda ikinci şans saf FIFO gibi davranır. Algoritma her zaman sonlanır, çünkü en kötü durumda bir tur sonra her R biti 0'dır.

İkinci şansın zayıf yanı gerçeklemededir: sayfalar listede sürekli taşınır. Bu taşımalar gereksiz iştir ve aynı karar daha ucuza verilebilir.

9.6.10 Saat algoritması

Tanım 9.17

Saat algoritması (*clock algorithm*) ikinci şansla aynı kararı verir, ama sayfaları taşımaz. Çerçeveler dairesel bir listede durur ve bir **el** (işaretçi) en eski sayfayı gösterir. Hata anında elin gösterdiği sayfanın R'si 0 ise sayfa çıkar ve el bir ilerler. R = 1 ise R temizlenir, el ilerler ve arama sürer (Şekil 9.25).



Şekil 9.25: Saat algoritması: çerçeveler dairesel listede, el en eski sayfayı gösterir. R = 0 → çıkar; R = 1 → R ← 0, el ilerler.

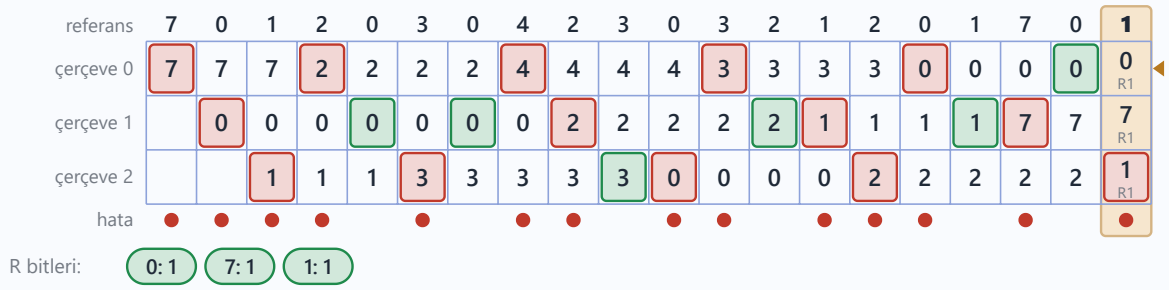
Yüklenen ya da erişilen sayfanın R'si donanımca 1 yapılır. El yalnız hata anında döner; isabetlerde kıpırdamaz. El sayfa tablosunda değil, *çerçeve listesinde* döner: fiziksel çerçeveler üzerinde sırayla gezinir.

Ortak dizide saat algoritması şöyle çalışır. İlk üç referans 7, 0, 1'i çerçeve 0, 1, 2'ye yükler; üçünün de R'si 1'dir ve el çerçeve 0'ı gösterir (Tablo 9.7).

Referans no	İstenen	R temizlenenler	Kurban	Hata
4	2	7, 0, 1	7	4
6	3	0	1	5
8	4	2, 0, 3	2	6
9	2	—	0	7
11	0	3, 4, 2	3	8
12	3	—	4	9
14	1	2, 0, 3	2	10
15	2	—	0	11
16	0	—	3	12
18	7	1, 2, 0	1	13
20	1	—	2	14

Tablo 9.7: Saat algoritmasının yer değiştirme anları, ortak dizi, 3 çerçeve. “R temizlenenler”, elin geçerken R'sini sıfırladığı sayfalardır.

- **Referans 4 (2).** 7, 0 ve 1'in R'si 1'dir. El üçünün de R'sini temizleyip bir tur atar ve yine çerçeve 0'a, 7'ye gelir. 7'nin R'si artık 0'dır; 7 çıkar. Tüm R'ler 1 olduğu için saat burada FIFO gibi davranmıştır.
- **Referans 5 (0).** İsabet; donanım 0'ın R'sini 1 yapar, el kıpırdamaz.
- **Referans 6 (3).** El çerçeve 1'de, 0'ın üstündedir. R = 1 olduğu için 0 ikinci şans alır. El sonraki çerçevedeki sayfa 1'e geçer; onun R'si 0'dır ve 1 çıkar. FIFO burada 0'ı atmıştı.
- **Referans 9 (2).** Önceki taramada R'si temizlenen ve o zamandan beri kullanılmayan 0 hemen kurban olur. Saatin zayıflığı burada görünür: 0 az sonra, 11. referansta yeniden istenecektir.
- **Sonuç.** Toplam 14 hata. FIFO'dan (15) bir az, LRU'dan (12) iki fazla.



Şekil 9.26: Saat algoritması, ortak dizi, 3 çerçeve. Hücrelerde R biti, turuncu üçgen saatin elidir. Toplam 14 hata.

Saat algoritması LRU'ya kaba bir yaklaşımdır: yalnız “son taramadan beri kullanıldı mı?” sorusunu sorar. Buna karşılık her isabette hiçbir iş yapmaz, hatada da çoğu zaman birkaç adımda kurban bulur. Bu ucuzluk onu gerçek sistemlerin temeli yapmıştır.

9.6.11 Kod izleme: saat algoritması

Aşağıdaki kod saat algoritmasını M bitiyle birlikte gerçekler. `cer` çerçeve dizisi, `el` saatin elidir; `N` çerçeve sayısıdır.

```
int kurban_sec(void) {
  while (cer[el].R == 1) { /* 2. şans */
    cer[el].R = 0;
    el = (el + 1) % N;
  }
  int k = el;              /* kurban */
  el = (el + 1) % N;
  return k;
}
void hata(int p) {
  int k = kurban_sec();
  if (cer[k].M) diske_yaz(k); /* kirli */
  cer[k].sayfa = p; cer[k].R = 1;
  cer[k].M = 0;
}
```

`kurban_sec()` içindeki `while` döngüsü, elin altındaki çerçevenin R'si 1 olduğu sürece döner: R'yi 0 yapar ve eli bir ilerletir (`% N` dairesel ilerlemeyi sağlar). R = 0 olan ilk çerçevede döngü biter. O çerçeve kurbandır ve el bir sonrakine geçer. `hata()` kurban kirliyse önce diske yazar; sonra yeni sayfayı R = 1, M = 0 ile yerleştirir.

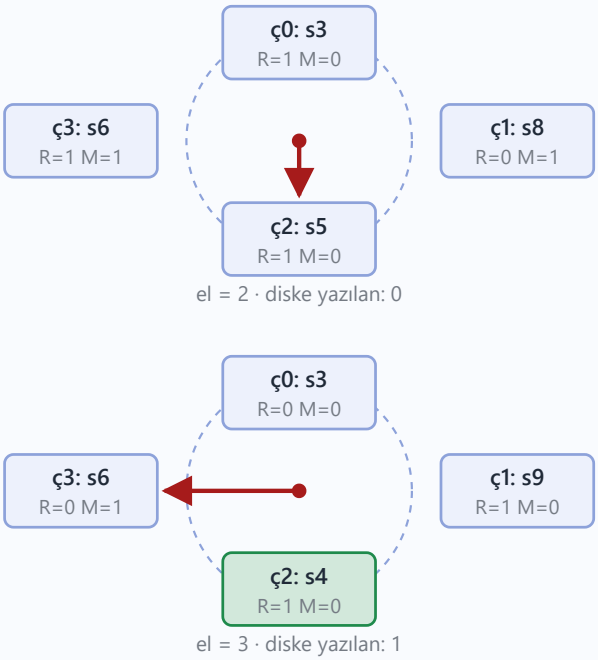
Dört çerçevenin başlangıç durumu şöyledir: çerçeve 0'da s3 (R = 1, M = 0), çerçeve 1'de s8 (R = 0, M = 1), çerçeve 2'de s5 (R = 1, M = 0), çerçeve 3'te s6 (R = 1, M = 1). El çerçeve 2'dedir. Önce sayfa 9, sonra sayfa 4 için hata oluşuyor (Tablo 9.8).

Adım	`el`	Çerçevedeki sayfa	R	Karar
hata(9)	2	s5	1	R ← 0, el 3'e
	3	s6	1	R ← 0, el 0'a
	0	s3	1	R ← 0, el 1'e
	1	s8	0	kurban k = 1, el 2'ye; M = 1 → diske yaz; s9 yüklenir
hata(4)	2	s5	0	kurban k = 2, el 3'e; M = 0 → yazma yok; s4 yüklenir

Tablo 9.8: `hata(9)` ve `hata(4)` izlemesi. Her satır döngünün bir adımıdır.

İlk hatada el tam bir tur döner. s5, s6 ve s3 yakın zamanda kullanılmıştır ve ikinci şans alır. s8'in R'si 0'dır, ama M biti 1'dir: bellekteki kopya diskte yoktur. Atılmadan önce diske yazılır ve bu bir ek G/Ç'dir. s9 çerçeve 1'e R = 1, M = 0 ile yüklenir.

İkinci hatada el çerçeve 2'dedir. s5'in R'si önceki turda sıfırlanmıştı ve o zamandan beri kullanılmadı. Kurban hemen bulunur. s5 temizdir, diskteki kopya günceldir; yazmaya gerek yoktur ve temiz kurban daha ucuzdur. Son durumda çerçeveler (s3, 0, 0), (s9, 1, 0), (s4, 1, 0), (s6, 0, 1)'dir; el çerçeve 3'te bekler (Şekil 9.27). İki hatada 3 R biti sıfırlanmış ve 1 disk yazması yapılmıştır.



Şekil 9.27: Saat kodunun başlangıç ve son durumu. Başta el çerçeve 2'de; iki hatadan sonra s8 ve s5 çıkmış, s9 ve s4 girmiş, el çerçeve 3'tedir. Diske yazılan sayfa sayısı 1.

`kurban_sec()` en kötü durumda $N + 1$ adım sürer: bütün R'ler 1 ise el bir tur atar ve başladığı yerde biter. Ortalama durumda çok daha kısadır. Bellek çok büyükse tek bir el bile turunu çok geç tamamlar; bu sorunun çözümü iki elli saattir.

9.6.12 NRU

Saat algoritması yalnız R bitine bakar. Oysa aynı derecede eski iki sayfadan temiz olanı atmak bir disk yazmasını kurtarır. NRU iki biti birlikte kullanır.

Tanım 9.18

NRU (*not recently used*, yakın zamanda kullanılmayan) algoritmasında her sayfanın R ve M bitleri vardır. Saat kesmesi R bitlerini periyodik olarak temizler. Sayfalar dört sınıfa ayrılır: sınıf 0 = (R, M) = (0, 0), sınıf 1 = (0, 1), sınıf 2 = (1, 0), sınıf 3 = (1, 1). Hata anında boş olmayan en düşük sınıftan bir sayfa atılır.

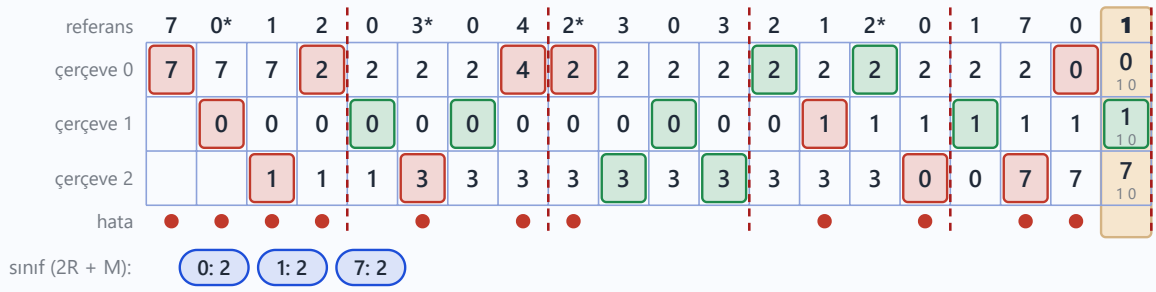
Sınıf numarası 2R + M'dir; R biti ağır basar. Bu tercih bilinçlidir: yakın zamanda kullanılmış temiz bir sayfayı atmak, kullanılmamış kirli bir sayfayı yazmaktan daha kötüdür. İlk bakışta sınıf 1 imkânsız görünebilir: bir sayfa kullanılmadan nasıl değişir? Değişmiştir, ama sonra bir saat tıkı R'sini silmiştir. M ise tıkla silinmez, çünkü sayfanın diske yazılması gerekip gerekmediğini söyler.

NRU'nun LRU'dan farkı ayrıntı düzeyindedir. LRU tam kullanım sırasını izler. NRU yalnız iki soruyu sorar: son saat aralığında kullanıldı mı, kirli mi? Sonuç kaba ama ucuzdur. Bir sınıfın içinde hangi sayfanın seçileceği belirsizdir; bu notta eşitlikte en eski yüklenen seçilir.

Örnek 9.10

Ortak dizi, 3 çerçeve. 2., 6., 9. ve 15. referanslar yazmadır (0*, 3*, 2*, 2*). Saat tıkı her 4 referansta bir gelir ve bütün R'leri temizler. NRU kaç hata ve kaç kirli yazma yapar?

1. **Referans 1–3.** 7, 0*, 1 yüklenir. 0 yazmayla yüklendiği için M = 1'dir.
2. **Referans 4 (2).** Sınıflar: 7 → 2, 0 → 3, 1 → 2. En düşük sınıf 2'de 7 ve 1 vardır; en eski yüklenen 7 çıkar. Ardından tık: bütün R'ler 0.
3. **Referans 6 (3*).** 0 az önce okunduğu için R = 1. Sınıflar: 2 → 0, 0 → 3, 1 → 0. Sınıf 0'daki en eski 1 çıkar.
4. **Referans 8 (4).** Sınıflar: 2 → 0, 0 → 3, 3 → 3. 2 çıkar; ardından tık.
5. **Referans 9 (2*).** Tık R'leri silmiştir: 4 → 0, 0 → 1, 3 → 1. 4 çıkar. 0 ve 3 kirli ama kullanılmamıştır.
6. **Referans 14 (1).** Sınıflar 2 → 3, 0 → 1, 3 → 1. Sınıf 1'deki en eski 0 çıkar; kirli olduğu için önce diske yazılır (yazma 1).
7. **Referans 16 (0).** 2 → 3, 1 → 2, 3 → 1. 3 çıkar ve yazılır (yazma 2); ardından tık.
8. **Referans 18 (7).** 2 → 1, 1 → 2, 0 → 0. 0 çıkar; bu kez okuma yoluyla geri geldiği için temizdir.
9. **Referans 19 (0).** 2 → 1, 1 → 2, 7 → 2. Kirli 2 çıkar ve yazılır (yazma 3).
10. **Sonuç.** 11 hata, 3 kirli sayfa yazması.



Şekil 9.28: NRU, ortak dizi, 3 çerçeve. Yıldız yazma erişimini, hücrelerin altındaki küçük sayılar R ve M bitlerini, kesikli çizgiler saat tıklarını gösterir. Toplam 11 hata, 3 kirli yazma.

NRU'nun 11 hatası LRU'nun 12'sinden azdır. Bu, NRU'nun LRU'dan iyi olduğu anlamına gelmez. R ve M bitine dayalı algoritmaların sonucu tıkların nereye düştüğüne ve eşitlik kuralına bağlıdır. Kısa tek bir dizide şans payı büyüktür.

9.6.13 Geliştirilmiş saat

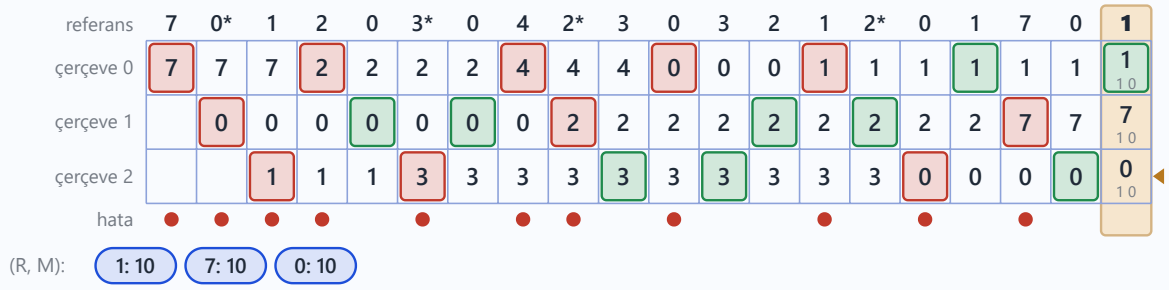
NRU'nun sınıf fikri saat algoritmasının dairesel listesiyle birleştirilebilir. Silberschatz bu algoritmaya "geliştirilmiş ikinci şans" (*enhanced second-chance*) adını verir; eski macOS sürümleri de bunu kullanmıştı.

Tanım 9.19

Geliştirilmiş saat algoritması hata anında iki tür tur atar. *Tur 1*: elden başlayarak (0, 0) sınıfından bir sayfa aranır; bitlere dokunulmaz. *Tur 2*: (0, 1) sınıfından bir sayfa aranır; el geçtiği her sayfanın R'sini 0 yapar. Tur 2 de sonuçsuz kalırsa tur 1'e dönülür. Artık bütün R'ler 0 olduğu için aramanın sonunda bir kurban mutlaka bulunur.

Algoritmada saat tıkı yoktur; R bitleri yalnız el geçerken temizlenir. Ortak dizi ve aynı yazmalarla izleme şöyledir:

- **Referans 4 (2).** Bellekte 7 (1, 0), 0 (1, 1), 1 (1, 0) vardır ve el çerçeve 0'dadır. Tur 1'de (0, 0) yoktur. Tur 2'de (0, 1) de yoktur, ama el geçerken bütün R'leri temizler. Yeniden tur 1'de el 7'nin üstündedir ve 7 artık (0, 0)'dır: 7 çıkar.
- **Referans 6 (3*).** 2 (1, 0), 0 (1, 1), 1 (0, 0); el çerçeve 1'de. Tur 1'de 0'ı geçer, 1'i (0, 0) bulur: 1 çıkar.
- **Referans 8 (4).** Tur 1 ve tur 2 sonuçsuz kalır; tur 2 bütün R'leri temizler. Yeniden tur 1'de 2 (0, 0) bulunur: 2 çıkar.
- **Referans 9 (2*).** Bellekte 4 (1, 0), 0 (0, 1), 3 (0, 1). Tur 1'de (0, 0) yoktur. Tur 2 elin altındaki 0'ı (0, 1) bulur: 0 çıkar ve önce diske yazılır (yazma 1).
- **Sonraki hatalar.** 11. referansta 4, 14.'de 0, 16.'da kirli 3 (yazma 2), 18.'de kirli 2 (yazma 3) çıkar.
- **Sonuç.** 11 hata, 3 kirli yazma (Şekil 9.29).



Şekil 9.29: Geliştirilmiş saat, ortak dizi, 3 çerçeve. Hücrelerde R ve M bitleri, üçgen saatin elidir. Toplam 11 hata, 3 kirli yazma.

Sonuç NRU'yla aynıdır, ama kurbanlar farklıdır: NRU 7, 1, 2, 4, 0, 3, 0, 2'yi, geliştirilmiş saat 7, 1, 2, 0, 4, 0, 3, 2'yi atar. Fark 9. referansta doğar. NRU'da bir tık tüm R'leri silmiştir ve 4 sınıf 0'dadır. Geliştirilmiş saatte ise 4'ün R'si hâlâ 1'dir ve kirli ama kullanılmamış 0 seçilir. Bedel, bir hatada belleği

birkaç kez taramaktır.

9.6.14 NFU ve yaşlandırma

R biti yalnız son aralığı hatırlar. Daha uzun bir geçmişi yazılımla tutmak mümkündür.

Tanım 9.20

NFU (*not frequently used*, sık kullanılmayan) algoritmasında her sayfanın bir yazılım sayacı vardır. Her saat tıkında sayaca o anki R biti eklenir (sayaç += R) ve R temizlenir. Hata anında sayacı en küçük sayfa atılır.

NFU'nun kusuru hiç unutmamasıdır. Programın ilk evresinde çok kullanılan bir sayfa yüksek bir sayaç biriktirir. Program başka bir evreye geçip o sayfaya hiç dokunmasa bile sayacı yüksek kalır ve sayfa bellekte kalmayı sürdürür. Sayaç kullanım *sayısını* tutar, kullanım *zamanını* tutmaz. Küçük bir değişiklik bu kusuru giderir.

Tanım 9.21

Yaşlandırma (*aging*) algoritmasında her tıkta sayaç önce bir bit sağa kaydırılır, sonra R biti en sola, yani en anlamlı bite eklenir. Böylece yakın geçmişteki kullanım sayaca daha ağır basar. Hata anında sayacı en küçük sayfa atılır.

Örnek 9.11

Tanenbaum'un örneğinde altı sayfa (0–5) ve 8 bitlik sayaçlar var; sayaçlar başta 0. Beş tıkta görülen R bitleri (sayfa 0'dan 5'e): tık 0: 1 0 1 0 1 1, tık 1: 1 1 0 0 1 0, tık 2: 1 1 0 1 0 1, tık 3: 1 0 0 0 1 0, tık 4: 0 1 1 0 0 0. Son sayaçlar nedir, hangi sayfa atılır?

1. **Tık 0.** R'si 1 olan sayfaların sayacı 10000000 olur: 0, 2, 4, 5. Diğerleri 00000000'dır.
2. **Tık 1.** Önce kaydırma, sonra R: sayfa 0 → 11000000, 1 → 10000000, 2 → 01000000, 3 → 00000000, 4 → 11000000, 5 → 01000000.
3. **Tık 2–3.** Aynı işlem: tık 3 sonunda 11110000, 01100000, 00010000, 01000000, 10110000, 01010000.
4. **Tık 4.** 01111000 (120), 10110000 (176), 10001000 (136), 00100000 (32), 01011000 (88), 00101000 (40).
5. **Karar.** En küçük değer 32'dir: sayfa 3 atılır.



Şekil 9.30: Yaşlandırma (Tanenbaum Şekil 3-17; sayaç değerleri R bitlerinden yeniden hesaplandı). Beşinci tıktan sonra sayacı en küçük olan sayfa 3'tür.

Sayfa 3 ve 5'in karşılaştırması yaşlandırmanın mantığını gösterir. İkisi de son iki tıkta kullanılmamıştır. Sayfa 3 en son tık 2'de kullanılmıştır (00100000), sayfa 5 de tık 2'de, ama ayrıca tık 0'da (00101000). Eşit yeni geçmişte daha uzun geçmiş belirleyici olur.

Yaşlandırma LRU'ya iyi yaklaşır, ama iki farkı vardır. Birincisi, aynı tık aralığında kullanılan iki sayfanın hangisinin daha sonra kullanıldığı bilinmez; ikisinin de en anlamlı biti 1'dir. İkincisi, 8 bitlik sayaç yalnız son 8 tıkı hatırlar. 9 tık önce kullanılan bir sayfa ile 1.000 tık önce kullanılan bir sayfanın sayacı aynıdır: 0. Tıklar yaklaşık 20 ms aralıkla gelirse 8 bit 160 ms'lik bir geçmiş demektir ve bu çoğu zaman yeterlidir.

Sık yapılan hata

Sayacın bitlerini ters sırada okumak. En son tık en anlamlı bittir. R bitleri tık 0'dan tık 4'e 1, 1, 1, 0, 0 sırasıyla gelen bir sayfanın sayacı 11100000 değil 00111000'dır (56): en yeni iki tıkta kullanılmamıştır. NFU aynı sayfaya 3 puan verirdi ve yalnız 2 kez kullanılmış ama en son iki tıkta kullanılan bir sayfayı atardı.

9.6.15 Yaşlandırma ortak dizide

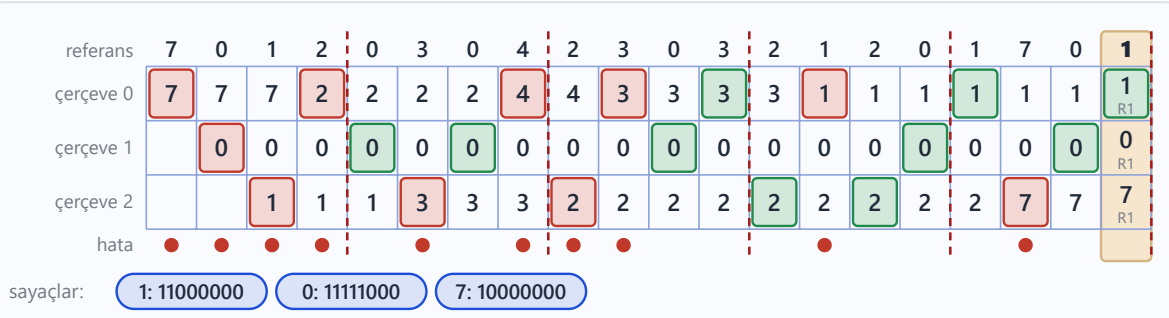
Yaşlandırmayı ortak dizide çalıştırmak bir ayrıntıyı ortaya çıkarır. Tıklar her 4 referansta bir gelir. Yeni yüklenen sayfanın sayacı 0, R'si 1'dir. Karşılaştırmada henüz sayaca eklenmemiş R biti en anlamlı bit sayılır, yani 9 bitlik bir değer (R, sayaç) kullanılır. Bunu yapmasaydık yeni yüklenmiş sayfanın sayacı 0 olurdu ve sayfa hemen ilk hatada atılırdı. Eşitlikte en eski yüklenen sayfa seçilir.

Referans no	İstenen	Anahtarlar	Kurban
4	2	7: 100000000, 0: 100000000, 1: 100000000	7
6	3	2: 010000000, 0: 110000000, 1: 010000000	1
8	4	2: 010000000, 0: 110000000, 3: 100000000	2
9	2	4: 010000000, 0: 011000000, 3: 010000000	3

Referans no	İstenen	Anahtarlar	Kurban
10	3	4: 010000000, 0: 011000000, 2: 100000000	4
14	1	3: 010000000, 0: 011100000, 2: 110000000	3
18	7	1: 110000000, 0: 011110000, 2: 011000000	2

Tablo 9.9: Yaşlandırma, ortak dizi, 3 çerçeve, tık her 4 referansta. Değerler 9 bitlik (R, sayaç) anahtarlardır.

Referans 4'te üç sayfanın anahtarı eşittir ve en eski yüklenen 7 çıkar. Ardından ilk tık gelir ve R'ler sayaçlara kaydırılır. Referans 6'da 2 ve 1'in anahtarı eşittir (yalnız sayaçlarında bir tık önceki kullanım var); 1 daha önce yüklendiği için çıkar. 0 ise 5. referansta yeniden kullanıldığı için R'si 1'dir ve korunur. Referans 9 ve 10'da kısa aralıkla iki hata olur: 2 için 3, 3 için 4 atılır. Toplam 10 hata yapılır (Şekil 9.31).



Şekil 9.31: Yaşlandırma, ortak dizi, 3 çerçeve, her 4 referansta bir tık (kesikli çizgiler). Toplam 10 hata; son sayaçlar 1: 11000000, 0: 11111000, 7: 10000000.

10 hatayla yaşlandırma bu dizide LRU'dan (12) iyidir, OPT'ye (9) çok yakındır. Bu sonuç da genellenmemelidir. Tek bir kısa dizi algoritmaları sıralamaya yetmez; tık sıklığı ya da eşitlik kuralı değişince sonuç değişir.

9.7 Çalışma kümesi

Buraya kadarki algoritmalar tek bir soruyu yanıtladı: hangi sayfa çıkmalı? Denning 1968'de soruyu değiştirdi: bir sürecin verimli çalışması için bellekte *hangi sayfaları* tutması gerekir? Yanıt yine yerellikten gelir ve hem yer değiştirmeye hem de çerçeve tahsisine yön verir.

9.7.1 Çalışma kümesi modeli

Bir süreç bir süre küçük bir sayfa grubunu kullanır, sonra başka bir gruba geçer. Örneğin bir fonksiyon çağrıldığında fonksiyonun kodu, yerel değişkenleri ve kullandığı global veriler bir **yerellik** oluşturur. Fonksiyon dönünce süreç başka bir yerelliğe geçer. Yerellikler birbiriyle örtüşebilir.

Tanım 9.22

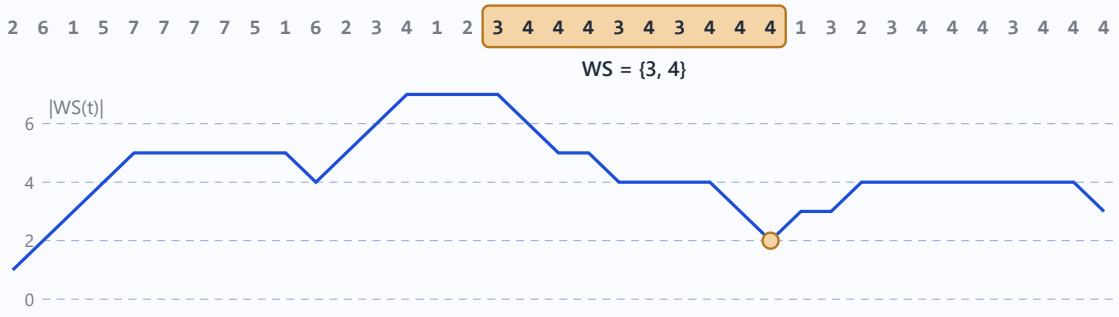
Çalışma kümesi (*working set*) WS(t, Δ), t anında biten son Δ bellek referansında kullanılan farklı sayfaların kümesidir. Δ'ya **çalışma kümesi penceresi** denir.

Δ'nın seçimi önemlidir. Δ çok küçükse pencere yerelliğin tamamını kapsamaz ve gerekli sayfalar dışarıda kalır. Çok büyükse birkaç yerellik üst üste biner ve küme gereğinden büyür. Δ = ∞ ise çalışma kümesi programın hayatı boyunca dokunduğu bütün sayfalardır. Modelin özü şudur: sürecin çalışma kümesi bellekteyse az hata olur; değilse süreç thrashing'e girer.

Örnek 9.12

Silberschatz'ın dizisi (37 referans): 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4. Pencere Δ = 10 referans. Farklı anlarda çalışma kümesi nedir?

1. ***t* = 10.** Pencere 1.–10. referanslar: 2 6 1 5 7 7 7 7 5 1 → {1, 2, 5, 6, 7}, boyut 5. Silberschatz'taki t_1 anı budur.
2. ***t* = 14.** Pencere 5.–14.: 7 7 7 7 5 1 6 2 3 4 → {1, 2, 3, 4, 5, 6, 7}, boyut 7. Eski ve yeni yerellik üst üste biner.
3. ***t* = 18.** Pencere 9.–18. → {1, 2, 3, 4, 5, 6}, boyut 6.
4. ***t* = 22.** Pencere 13.–22. → {1, 2, 3, 4}, boyut 4.
5. ***t* = 26.** Pencere 17.–26.: 3 4 4 4 3 4 3 4 4 4 → {3, 4}, boyut 2. Silberschatz'ın t_2 anı; süreç yeni bir yerelliğe yerleşmiştir.
6. ***t* = 30 ve 37.** {1, 2, 3, 4} (boyut 4) ve {2, 3, 4} (boyut 3).



Şekil 9.32: Çalışma kümesi penceresi (Δ = 10) t = 26'da: pencere yalnız 3 ve 4'ü içerir. Alttaki çizgi |WS(t)|'nin zamanla değişimidir.

Çalışma kümesinin boyutu zamanla dalgalanır: t = 10'dan 37'ye boyutlar 5, 4, 5, 6, 7, 7, 7, 7, 6, 5, 5, 4, 4, 4, 4, 3, 2, 3, 3, 4, … biçiminde gider. Bir yerellikten ötekine geçişte küme geçici olarak büyür, çünkü pencere iki yerelliğin sayfalarını birlikte görür. Yeni yerellik oturunca küme küçülür.

9.7.2 Çalışma kümesi algoritması

Kesin tanım, yani son Δ referansın kümesi, her bellek erişiminde kaydırılan bir yazmaç ister. Bu çok pahalıdır. Uygulanabilir yaklaşım referans sayısı yerine zamanı kullanır: çalışma kümesi, son τ sanal zaman içinde kullanılan sayfalardır. **Sanal zaman** sürecin kendi CPU süresidir; süreç beklerken akmaz ve saat tıklarıyla ölçülür.

Tanım 9.23

Çalışma kümesi algoritmasında her sayfa tablosu girdisinde son kullanım zamanı ve R biti tutulur. Hata anında tablo taranır. R = 1 olan sayfanın zamanı şimdiki sanal zaman yapılır ve R temizlenir. R = 0 olan sayfanın **yaşı** (şimdi − son kullanım) τ'dan büyükse sayfa çalışma kümesinde değildir ve kurban olabilir.

Örnek 9.13

Tanenbaum'un örneği: şimdiki sanal zaman 2204, τ = 500. Sekiz sayfanın (son kullanım, R) değerleri: (2084, 1), (2003, 1), (1980, 1), (1213, 0), (2014, 1), (2020, 1), (2032, 1), (1620, 0). Hangi sayfa çıkar?

1. **R = 1 olanlar.** Altı sayfa bu tıkta kullanılmıştır; zamanları 2204 yapılır, R'leri temizlenir.
2. **R = 0 olanlar.** Yaşlar 2204 − 1213 = 991 ve 2204 − 1620 = 584'tür. İkisi de τ = 500'den büyüktür, ikisi de çalışma kümesinde değildir.
3. **Seçim.** Bulunan ilk uygun sayfa, yani zamanı 1213 olan sayfa çıkarılır. Tarama diğer girdileri güncelleyerek sürer.



Şekil 9.33: Çalışma kümesi algoritması (Tanenbaum Şekil 3-20'nin yeniden çizimi). Tarama sonunda R = 1 olanların zamanı 2204 olmuştur; yaşı 991 olan sayfa çıkar.

Bütün sayfalar çalışma kümesindeyse, yani R = 0 olanların hepsinin yaşı τ'dan küçükse, R = 0 olanlar arasında en eskisi seçilir. Hiç R = 0 sayfa yoksa rastgele bir sayfa seçilir; tercihen temiz bir sayfa. Algoritmanın dezavantajı her hatada bütün tabloyu taramaktır.

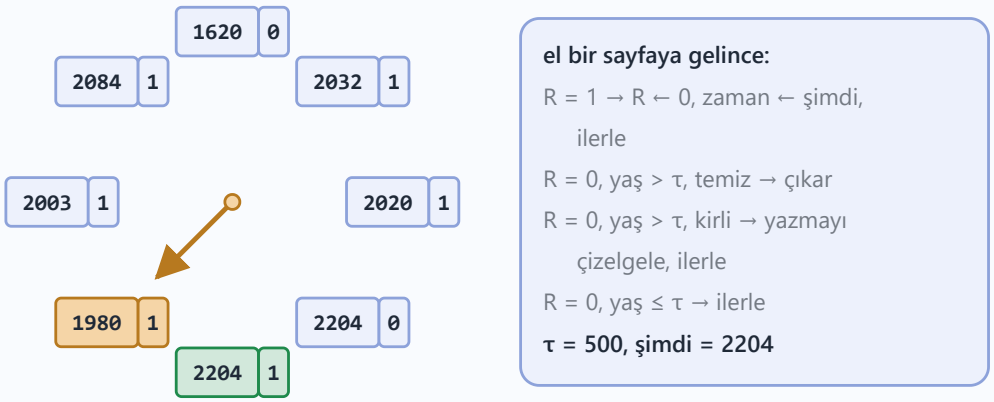
9.7.3 WSClock

Çalışma kümesi fikri ile saatin dairesel listesini birleştiren algoritma Carr ve Hennessy'nin 1981 tarihli WSClock'udur. Tam tablo taraması yerine el, çerçeveler üzerinde sırayla gezinir ve ilk uygun kurbanda durur.

> **Tanım 9.24**
>
> **WSClock** algoritmasında her çerçevenin son kullanım zamanı, R ve M bitleri vardır. El bir çerçeveye gelince: R = 1 ise R ← 0, zaman ← şimdi yapılır ve el ilerler. R = 0, yaş > τ ve sayfa temizse sayfa kurbandır. R = 0, yaş > τ ve sayfa kirliyse sayfanın diske yazılması çizelgelenir ve el ilerler. R = 0 ve yaş ≤ τ ise sayfa çalışma kümesindedir; el ilerler.

Kirli eski sayfanın hemen atılmaması önemlidir. Yazma işi diske verilir, el ise temiz bir sayfa aramayı sürdürür. Yazma bitince sayfa temizlenmiş olur ve sonradan ucuza atılır. El tam bir tur atıp başladığı yere dönerse iki durum vardır:

- En az bir yazma çizelgelendiyse el, yazması biten ilk sayfaya kadar döner ve onu atar.
- Hiç yazma çizelgelenmediyse bütün sayfalar çalışma kümesindedir. Herhangi bir temiz sayfa atılır; o da yoksa elin altındaki sayfa yazılıp atılır.



Şekil 9.34: WSClock (Tanenbaum Şekil 3-21'in yeniden çizimi), τ = 500, şimdi = 2204. El zamanı 2014 olan sayfada R = 1 bulur, onu güncelleyip ilerler; zamanı 1213 olan sayfanın yaşı 991 > τ olduğu için o çerçeveye yeni sayfa yüklenir.

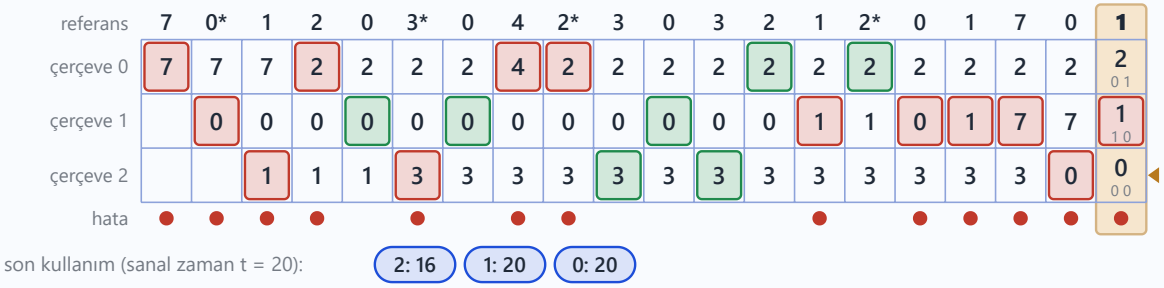
Tanenbaum'un örneğinde el önce zamanı 2014 olan sayfadadır. R = 1, yani sayfa bu tıkta kullanılmıştır ve iyi bir kurban değildir. R temizlenir, zaman 2204 yapılır, el ilerler. Sıradaki sayfanın zamanı 1213, R'si 0'dır. Yaşı 2204 − 1213 = 991 > τ olduğu için çalışma kümesinde değildir. Temizse kurban

olur; kirli olsaydı yazması çizelgelenir ve el ilerlerdi. Yeni sayfa bu çerçeveye zaman 2204 ve R = 1 ile yüklenir. WSClock basit, verimli ve uygulamada yaygın bir tasarımdır.

9.7.4 WSClock ortak dizide

WSClock'u ortak dizide çalıştırmak için sanal zaman olarak referans numarası alınır ve $\tau = 4$ seçilir. Yazmalar NRU'dakiyle aynıdır (2., 6., 9., 15. referanslar). Yazması çizelgelenen sayfa, tur sonunda temizlenmiş kabul edilir.

- **Referans 4 (2).** El 7, 0 ve 1'i sırayla gezer; üçünün de R'si 1'dir. R'ler temizlenir, zamanları 4 yapılır. Tur biter, yazma yoktur. İlk temiz sayfa 7'dir (0 kirlidir) ve 7 çıkar.
- **Referans 6 (3*).** 0: R = 1 → R = 0, zaman 6. 1: R = 0, yaş $6 - 4 = 2 \le \tau$, çalışma kümesinde. 2: R = 1 → R = 0, zaman 6. Tur biter, yazma yoktur. Elden başlayınca ilk temiz sayfa 1'dir; 1 çıkar.
- **Referans 8 ve 9.** Aynı biçimde önce 2, sonra 4 ilk temiz sayfa olarak çıkar.
- **Referans 14 (1).** Bellekte 0*, 3*, 2* vardır; üçü de kirlidir ve R'leri 1'dir. Tur biter, temiz sayfa yoktur: elin altındaki 0 yazılıp atılır (yazma 1).
- **Referans 16–18.** Her seferinde çalışma kümesinde olmayan temiz bir sayfa bulunamaz; tur sonunda ilk temiz sayfa çıkar: sırasıyla 1, 0, 1.
- **Referans 19 (0).** 3: R = 0, yaş $19 - 14 = 5 > \tau$ ve kirli: yazması çizelgelenir. 2: yaş $3 \le \tau$. 7: R = 1 → R = 0. Tur biter; yazılan 3 artık temizdir ve kurban olur (yazma 2).
- **Referans 20 (1).** 7 ilk temiz sayfa olarak çıkar.
- **Sonuç.** 13 hata, 2 kirli yazma (Şekil 9.35).



Şekil 9.35: WSClock, ortak dizi, 3 çerçeve, $\tau = 4$. Hücrelerde R ve M bitleri, durum satırında son kullanım zamanları. Toplam 13 hata, 2 kirli yazma.

13 hata, LRU'dan bile kötü bir sonuçtur. Bunun nedeni örneğin küçüklüğüdür: 3 çerçeveyle her sayfa çalışma kümesindedir ve τ ölçütü hiçbir sayfayı ayırt edemez. Algoritma çoğu hatada tur sonundaki yedek kurala, "ilk temiz sayfa"ya düşer. Çalışma kümesi fikrinin getirisi, binlerce çerçeveli bir bellekte ve uzun bir çalışmada ortaya çıkar.

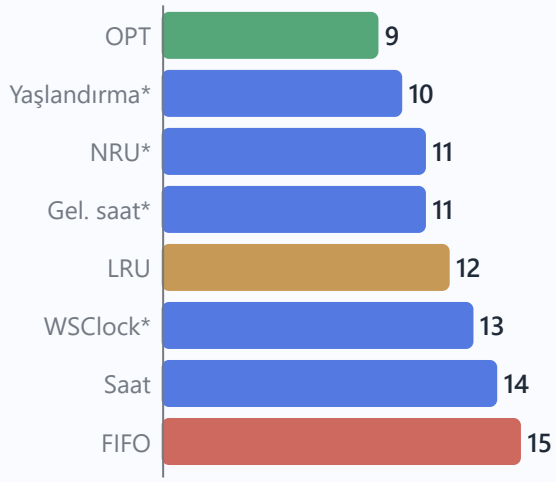
9.8 Algoritmaların karşılaştırması

9.8.1 Ortak dizideki sonuçlar

Ortak dizide (3 çerçeve) bütün algoritmaların sonucu yan yana konabilir (Tablo 9.10, Şekil 9.36).

Algoritma	Hata	Yorum
OPT	9	Uygulanamaz; kıyaslama için alt sınır.
Yaşlandırma*	10	LRU'ya iyi yaklaşan verimli algoritma.
NRU*	11	LRU'ya çok kaba yaklaşım; ucuz.
Geliştirilmiş saat*	11	NRU'nun sınıflarını saatle birleştirir.
LRU	12	Çok iyi, ama tam hâli donanım ister.
WSClock*	13	İyi ve verimli; küçük örnekte avantajı görünmez.
Saat	14	İkinci şansın gerçekçi gerçeklemesi.
FIFO	15	Önemli sayfaları atabilir; Belady anomalisi.
İkinci şans	—	FIFO'ya göre büyük iyileşme (kararları saatle aynıdır).
NFU	—	LRU'ya oldukça kaba yaklaşım; hiç unutmaz.
Çalışma kümesi	—	İyi, ama her hatada tablo taraması pahalı.

Tablo 9.10: Ortak dizide hata sayıları ve Tanenbaum'un genel değerlendirmesi. Yıldızlı algoritmalar yazma ve tık bilgisini de kullanır.



Şekil 9.36: Ortak dizide hata sayıları: OPT 9, yaşlandırma 10, NRU ve geliştirilmiş saat 11, LRU 12, WSClock 13, saat 14, FIFO 15.

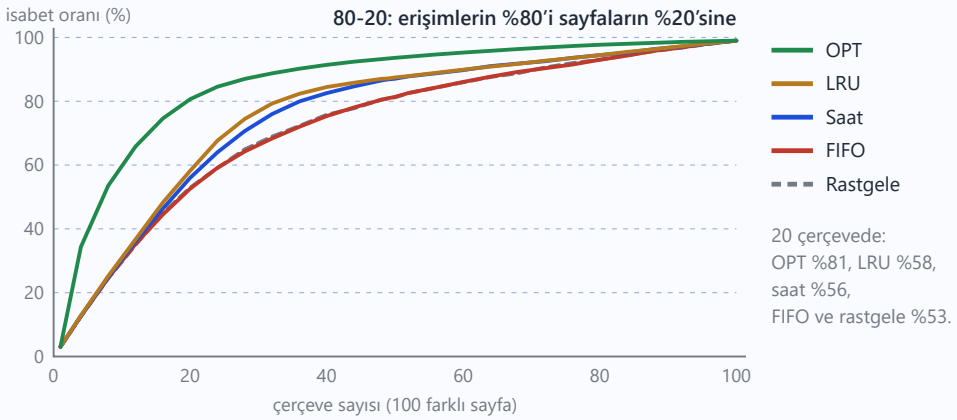
Tablodaki sıralama yanıltıcı olabilir. R ve M bitli algoritmalar, LRU'nun hiç kullanmadığı yazma ve tık bilgisinden yararlanır; kısa tek bir dizi genel bir sıralama vermez. Tanenbaum'un değerlendirmesi uzun ölçümlere dayanır: pratikte en iyi iki algoritma yaşlandırma ve WSClock'tur. NFU ise hiç unut-

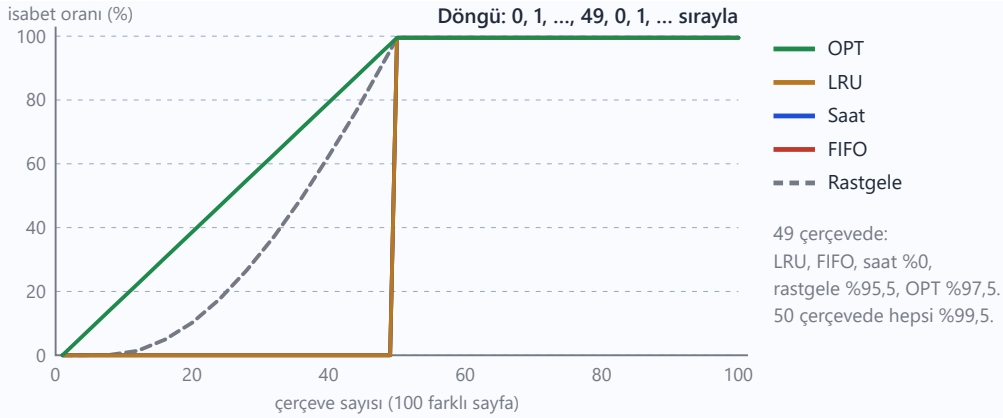
madığı için LRU'ya iyi yaklaşmaz; iyi yaklaşım yaşlandırmadır.

9.8.2 İş yükü politikayı belirler

Bir algoritmanın iyiliği erişim örüntüsüne bağlıdır. Bunu görmek için OSTEP'teki deney tekrarlanabilir. 100 farklı sayfaya (döngüde 50 sayfaya) 10.000 erişim yapan üç yapay iş yükü vardır. Her biri, çerçeve sayısı 0'dan 100'e kadar değiştirilerek beş politikayla çalıştırılır (Şekil 9.37). Rastgele politika için 5 denemenin ortalaması alınır.

- **Yerelliksiz iş yükü.** Her erişim 100 sayfadan rastgele seçilir. Geçmiş geleceğe dair hiçbir şey söylemez; LRU, FIFO, saat ve rastgele aynı doğruyu izler ve isabet oranı yaklaşık çerçeve / 100'dür. Yalnız geleceği bilen OPT öndedir. 100 çerçevede bütün sayfalar sığar ve hepsi %99 isabete ulaşır (kalan %1 ilk yüklemelerdir).
- **80-20 iş yükü.** Erişimlerin %80'i sayfaların %20'sine gider. Sıcak sayfalar varsa geçmişi kullanan politikalar kazanır. 20 çerçevede OPT %81, LRU %58, saat %56, FIFO ve rastgele yaklaşık %53 isabet verir. Saat LRU'ya çok yakındır; FIFO rastgeleden iyi değildir.
- **Döngü iş yükü.** 0, 1, ..., 49, 0, 1, ... sırayla tekrarlanır. Bellekten biraz büyük bir döngü LRU'nun en kötü durumudur. 49 çerçevede atılan sayfa hep en yakında gerekecek olandır ve LRU, FIFO ve saat her erişimde ıskalar: %0. Rastgele %95,5, OPT %97,5 isabet verir. 50 çerçevede döngü sığar ve hepsi %99,5'e çıkar.





Şekil 9.37: Çerçeve sayısına göre isabet oranı. Üstte 80-20 iş yükü: geçmişi kullanan politikalar öndedir. Altta döngü: 49 çerçevede LRU, FIFO ve saat %0, rastgele %95,5 isabet verir.

Bu deney bu not için Python'da yeniden çalıştırılmıştır. Rastgele sayı üretecine bağlı küçük farklarla aynı sonuçlar çıkar: 80-20'de 20 çerçeve için OPT %80,7, LRU %58,1, saat %55,2, FIFO %52,7, rastgele %52,4; döngüde 49 çerçeve için OPT %97,5, rastgele %95,5, diğerleri %0.

Döngü örüntüsü yalnız yapay bir deney değildir. Büyük bir dosyayı baştan sona bir kez okumak da bir taramadır ve LRU'yu gerçek sistemlerde vurur: bir daha kullanılmayacak dosya sayfaları sıcak sayfaları bellekten kovar. Linux bu yüzden yeni sayfayı önce etkin olmayan listeye koyar ve ancak ikinci erişimde etkin listeye taşır.

> **Sık yapılan hata**
>
> LRU'nun her durumda FIFO'dan ve rastgeleden iyi olduğunu sanmak. LRU yerelliğe dayanır. Erişimler bellekten biraz büyük bir döngü izlediğinde LRU hep yanlış sayfayı atar ve kör rastgele seçim onu açık farkla geçer.

9.9 Çerçeve tahsisi

Birden çok süreç belleği paylaştığında ikinci soru ortaya çıkar: her sürece kaç çerçeve verilmeli?

9.9.1 Eşit ve orantılı tahsis

Tahsisin iki sınırı vardır. **Alt sınırı** mimari belirler: bir komutun dokunabileceği bütün sayfalar aynı anda bellekte olmalıdır, yoksa komut hiç tamamlanamaz. Komutun kendisi, işlenenleri ve dolaylı adresler sayfa sınırlarını aşabilir. Örneğin IBM 370'in bellekten belleğe taşıma komutu MVC altı çerçeve isteyebilir: komutun kendisi, kaynak blok ve hedef blok ayrı ayrı iki sayfaya taşabilir. **Üst sınır** ise boş fiziksel bellektir. Bu iki sınır arasında üç yaklaşım vardır.

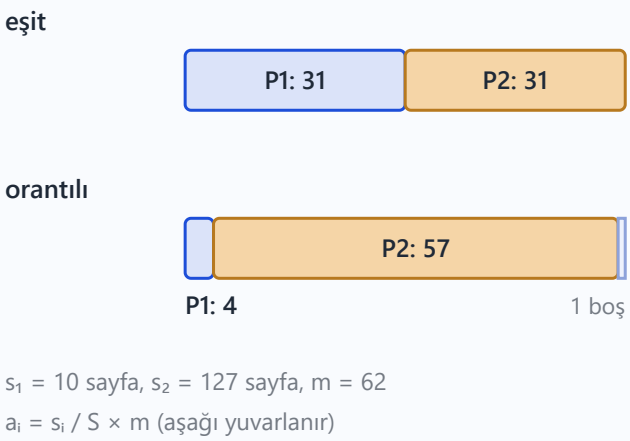
Tanım 9.25

Eşit tahsis: m çerçeve n sürece $\lfloor m / n \rfloor$ çerçeve olarak bölünür. **Orantılı tahsis**: i. sürecin sanal boyutu s_i, toplam $S = \Sigma\, s_i$ ise sürece $a_i = s_i / S \times m$ çerçeve (aşağı yuvarlanarak) verilir. **Öncelikli tahsis**: oran boyut yerine ya da boyutla birlikte önceliğe göre kurulur.

Örnek 9.14

62 çerçeve, 10 sayfalık P1 ve 127 sayfalık P2. Eşit ve orantılı tahsis ne verir?

1. **Eşit.** $\lfloor 62 / 2 \rfloor = 31$: her birine 31 çerçeve. 10 sayfalık P1, 21 çerçeveyi hiç kullanamaz.
2. **Orantılı.** $S = 137$. P1: $10 / 137 \times 62 \approx 4{,}53 \to 4$. P2: $127 / 137 \times 62 \approx 57{,}47 \to 57$.
3. **Kalan.** $4 + 57 = 61$; aşağı yuvarlama yüzünden 1 çerçeve boşta kalır (Şekil 9.38).



Şekil 9.38: 62 çerçevenin iki sürece dağıtılması: eşitte 31 ve 31, orantılıda 4 ve 57 (1 çerçeve boş).

Öncelikli tahsiste yüksek öncelikli süreç, sayfa hatası yaptığında düşük öncelikli bir sürecin çerçevesini alabilir. Çok programlama derecesi değiştikçe tahsis de değişir: yeni bir süreç gelince herkesin payı biraz azalır. NUMA (bellek erişimi eşit olmayan) sistemlerde bir çerçevenin *nerede* olduğu da önemlidir. Süreç, çalıştığı işlemci düğümüne yakın bellekten beslenmelidir; uzak düğümün belleği daha yavaştır.

9.9.2 Yerel ve genel yer değiştirme

Kurbanın hangi çerçeveler arasından seçileceği de bir karardır.

Tanım 9.26

Genel yer değiştirmede (*global replacement*) kurban bütün çerçeveler arasından seçilir; bir süreç başka bir sürecin çerçevesini alabilir. **Yerel yer değiştirmede** (*local replacement*) süreç kurbanı yalnız kendi çerçeveleri arasından seçer.

Örnek 9.15

Üç süreç (A, B, C) 16 çerçeveyi paylaşıyor. Sayılar yaşlandırma sayacıdır: A0 10, A1 7, A2 5, A3 4, A4 6, A5 3; B0 9, B1 4, B2 6, B3 2, B4 5, B5 6, B6 12; C1 3, C2 5, C3 6. A, A6 sayfasında hata yapıyor. Hangi sayfa çıkar?

1. **Yerel.** Adaylar yalnız A'nın çerçeveleridir. En küçük değer A5'in 3'üdür; A5 çıkar. A'nın çerçeve sayısı 6'da kalır.
2. **Genel.** Bütün çerçeveler adaydır. En küçük değer B3'ün 2'sidir; B3 çıkar. A 7 çerçeveye büyür, B 6'ya küçülür (Şekil 9.39).



Şekil 9.39: Yerel ve genel yer değiştirme (Tanenbaum Şekil 3-22'nin yeniden çizimi). Üstte başlangıç; altta genel politikanın sonucu: A6, B3'ün çerçevesine yerleşmiştir. Yerel politikada A5 çıkardı.

Genel politika belleği daha verimli kullanır: boşta duran çerçeveler ihtiyacı olana geçer ve toplam verim artar. Bedeli öngörülemezliktir. Bir sürecin çalışma süresi, başka süreçlerin bellek davranışına göre çok değişebilir; aynı program bir gün 10, ertesi gün 30 saniye sürebilir. Yerel politika bu tutarlılığı sağlar, ama bir süreçte boşta duran çerçeveler sıkışan başka bir sürece geçemez. Ayrıca çalışma kümesi büyüyüp küçülürken çerçeve sayısını ayarlayan ayrı bir mekanizma ister; sayfa hatası sıklığı bu mekanizmadır. Linux ve Windows genel geri kazanım kullanır; sınırlar cgroup'lar ve iş nesneleriyle konur.

9.10 Thrashing

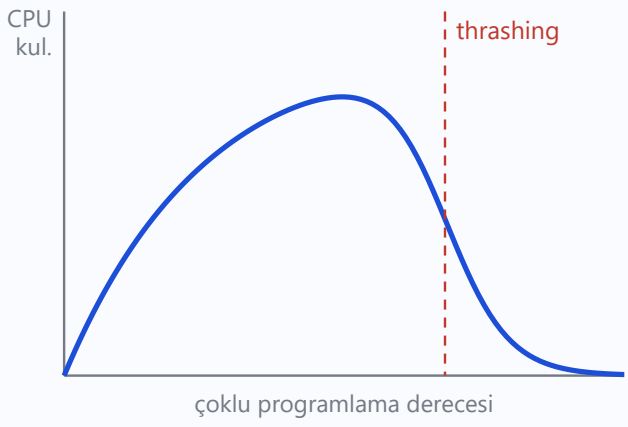
9.10.1 Boşa çabalama

Tanım 9.27

Thrashing (boşa çabalama), sistemin zamanının çoğunu sayfaları bellek ile disk arasında taşımakla geçirmesidir. Süreçlerin etkin sayfaları, yani çalışma kümeleri belleğe sığmayınca ortaya çıkar.

Mekanizma bir kısır döngüdür. Bir sürecin çalışma kümesi belleğe sığmazsa süreç sürekli hata yapar. Attığı sayfa hemen yeniden gerekir; o sayfayı getirmek için başka bir gerekli sayfayı atar. Süreçler zamanlarının çoğunu sayfa beklemekle geçirir, disk kuyruğu dolar, hazır kuyruğu boşalır ve CPU kullanımı düşer.

Durumu kötüleştiren bir geri besleme de vardır. Eski sistemlerin uzun vadeli çizelgeleyicileri CPU kullanımını izler ve düşük kullanımı "CPU boşta, daha çok iş alabiliriz" diye yorumlardı. Yeni süreç sisteme girince herkesin çerçeve payı daha da küçülür, hatalar artar, CPU kullanımı daha da düşer ve çizelgeleyici bir süreç daha ekler. Çok programlama derecesi arttıkça CPU kullanımı önce yükselir, bir noktadan sonra hızla çöker (Şekil 9.40).



Şekil 9.40: CPU kullanımı çok programlama derecesi arttıkça önce yükselir; thrashing başlayınca hızla düşer.

Nedenin formülü kısadır: $\Sigma\ |WS_i| > m$, yani çalışma kümelerinin toplamı bellekten büyüktür. Yerel yer değiştirme thrashing'i bir süreçle sınırlar: sıkışan süreç başkalarının çerçevesini çalamaz. Ama sorunu çözmez; o süreç yine thrashing'dedir ve disk kuyruğunu doldurarak başkalarının sayfa hatalarını da yavaşlatır.

Çareler bellekle talep arasındaki dengeyi kurmaya dayanır. Çok programlama derecesi düşürülür: bazı süreçler askıya alınır ya da takasa atılır. RAM eklenir. Son çare, sistemin bellek tükenmesi öldürücüsüdür (OOM, *out of memory*). Linux'ta belirti `vmstat 1` çıktısında görülür: `si` ve `so` sütunlarında yüksek takas trafiği, `wa` sütununda yüksek G/Ç beklemesi. `/proc/pressure/memory` dosyası (PSI, *pressure stall information*) bellek baskısını, süreçlerin bellek yüzünden beklediği zamanın yüzdesi olarak verir.

Sık yapılan hata

Düşük CPU kullanımını her zaman boşta kapasite sanmak. Bir izleme ekranında CPU %7, takas diski %100 ve sayfa hatası oranı çok yüksekse sistem boşta değil, thrashing'dedir. Bu durumda yeni iş almak doğru değildir; tersine, bazı süreçleri askıya almak gerekir.

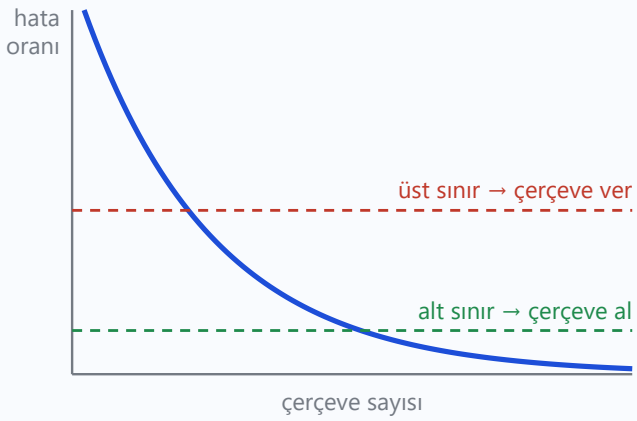
9.10.2 Çalışma kümesi ve PFF ile önlemek

Çalışma kümesi stratejisi. İşletim sistemi her sürecin çalışma kümesini izler ve toplam talebi hesaplar: $D = \Sigma\ |WS_i|$. $D > m$ ise thrashing yaklaşıyor demektir ve bir süreç askıya alınır; sayfaları yazılır, çerçeveleri diğerlerine dağıtılır. D yeterince küçükse yeni bir süreç başlatılabilir. Askıdan dönen süreç için önce çalışma kümesini yüklemek, yani ön sayfalama yapmak, dönüşteki hata fırtınasını önler.

Sayfa hatası sıklığı. Çalışma kümesini izlemek pahalıdır. Daha doğrudan bir geri besleme, sürecin hata oranına bakmaktır.

Tanım 9.28

Sayfa hatası sıklığı (PFF, *page-fault frequency*) yönteminde her sürecin hata oranı için bir üst ve bir alt sınır belirlenir. Oran üst sınırı aşarsa sürece çerçeve verilir; alt sınırın altına inerse süreçten çerçeve geri alınır (Şekil 9.41).



Şekil 9.41: Sayfa hatası oranı çerçeve sayısı arttıkça azalır. Üst sınırın üstünde sürece çerçeve verilir, alt sınırın altında çerçeve geri alınır.

Hata oranı yüksek ama verilecek boş çerçeve yoksa bir süreç askıya alınır ve çerçeveleri dağıtılır. Sayfa hataları ile yerellikler arasında da bir bağ vardır. Süreç yeni bir yerelliğe geçerken hata oranında bir tepe oluşur. Çalışma kümesi yüklenince oran düşer ve bir sonraki geçişe kadar düşük kalır. PFF bu tepeleri “daha çok çerçeve gerek”, sakin dönemleri “fazla çerçeve var” diye okur.

9.10.3 Kod izleme: PFF

Aşağıdaki fonksiyon PFF'in basit bir biçimidir. Her sayfa hatasında çağrılır. `t` şimdiki referans numarası, `s->son` son hatanın zamanı, `C` eşiktir. Son hatadan bu yana geçen süre `C`'den büyükse hatalar seyrektir ve son hatadan beri kullanılmayan sayfalar bırakılır. Küçükse hatalar sıktır ve hiçbir sayfa atılmadan küme büyür.

```
void pff(Surec *s, int p, int t) {
  if (t - s->son > C) {   /* seyrek hata */
    for (int i = s->n - 1; i >= 0; i--)
      if (s->kume[i].R == 0)
        cikar(s, i);    /* kullanılmadı */
  }
  for (int i = 0; i < s->n; i++)
    s->kume[i].R = 0;     /* sıfırla */
  ekle(s, p);             /* R = 1 ile */
  s->son = t;
}
```

Seyrek hata dalında döngü kümede sondan başa doğru gezinir. Bu yön bilinçlidir: `cikar()` bir öğeyi silince arkadaki öğeler bir sola kayar. Baştan sona gezilseydi bir öğe atlanırdı. Ardından bütün R bitleri sıfırlanır, böylece bir sonraki hatada “bu aralıkta kullanıldı mı?” sorusu yeniden sorulabilir. Yeni sayfa R = 1 ile eklenir ve hata zamanı kaydedilir.

Süreç belleksiz başlıyor; `C` = 2 ve referans dizisi (zaman 1'den 14'e) 1 2 1 3 1 3 1 3 4 4 5 4 5 2'dir (Tablo 9.11).

`t`	Sayfa	`t - son`	Dal	Çıkarılan	Küme sonra
1	s1	1 − 0 = 1	sık	—	{s1}
2	s2	2 − 1 = 1	sık	—	{s1, s2}
4	s3	4 − 2 = 2	sık	—	{s1, s2, s3}
9	s4	9 − 4 = 5	seyrek	s2	{s1, s3, s4}
11	s5	11 − 9 = 2	sık	—	{s1, s3, s4, s5}
14	s2	14 − 11 = 3	seyrek	s3, s1	{s4, s5, s2}

Tablo 9.11: `pff()` izlemesi, C = 2. Yalnız hata anları gösterilmiştir.

- ***t* = 1, 2, 4.** Aralıklar 1, 1 ve 2'dir; hiçbiri `C` 'yi aşmaz. Hatalar sıktır, hiçbir sayfa atılmaz; küme 3 sayfaya büyür. Aradaki 3. referans (s1) isabettir.
- ***t* = 9.** 5–8. referanslar s1 ve s3'e isabet eder ve R bitlerini 1 yapar. s4 hatasında aralık 5 > 2'dir: hatalar seyrektir. Son hatadan beri kullanılmayan (R = 0) tek sayfa s2'dir ve çıkarılır.
- ***t* = 11.** Aralık 2 ≤ C. Hiçbir sayfa atılmaz ve küme 4 sayfaya büyür. Süreç yeni bir yerelliğe (s4, s5) geçerken eski sayfalar (s1, s3) hâlâ bellektedir.
- ***t* = 14.** 12. ve 13. referanslar s4 ve s5'e isabet eder. s2 hatasında aralık 3 > 2'dir. R = 0 olan s3 ve s1 çıkarılır. Son küme s4 (R = 0), s5 (R = 0), s2 (R = 1)'dir.



Şekil 9.42: PFF izlemesinin sonu: 14 referansta 6 hata. Kırmızı hücreler sayfa hatası, yeşiller isabettir; son küme s4, s5, s2.

14 referansta 6 hata olmuştur. Yerellik 1–3 sayfalarından 4–5 sayfalarına kayınca önce hatalar sıklaşmış ve küme büyümüştür. Sonra seyrek bir hata, eski yerelliğin sayfalarını bırakmanın fırsatı olmuştur. Fonksiyonun maliyeti küme boyutuyla doğrusaldır, $O(n)$; ama yalnız hata anlarında çalışır.

9.11 Tasarım konuları

Yer değiştirme algoritması ve tahsis politikası sanal bellek sisteminin yalnız bir kısmıdır. Sistemin gerçekte ne kadar iyi çalışacağını birçok küçük karar belirler: kurban ne zaman aranır, sayfalar önceden getirilir mi, sayfa ne kadar büyük olmalı, programcı ne yapabilir, hangi sayfalar asla atılmamalı?

9.11.1 Temizleme politikası ve sayfalama daemonu

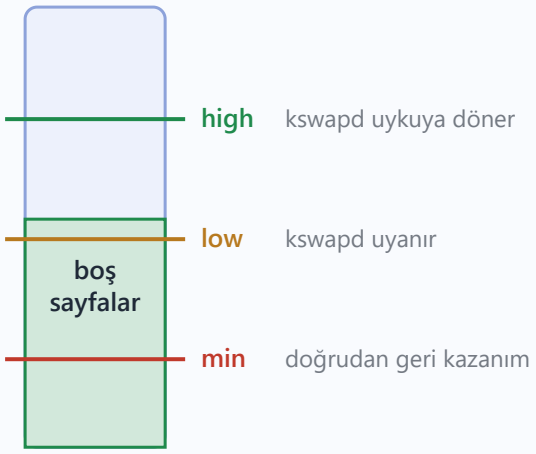
Kurbanı ancak sayfa hatası anında aramak, hatayı yapan süreci gereksiz yere bekletir. Kurban kirliyse süreç hem yazmayı hem okumayı bekler. Daha iyi bir yol, boş çerçeveleri önceden hazır tutmaktır.

> **Tanım 9.29**
>
> **Sayfalama daemonu** (*paging daemon*), periyodik olarak uyanan bir çekirdek iş parçacığıdır. Boş çerçeve sayısı azsa yer değiştirme algoritmasıyla sayfaları önceden serbest bırakır ve kirli sayfaları önceden diske yazar.

Önceden yazmanın getirisi şudur: tahliye anı geldiğinde sayfa çoğu zaman temizdir ve hata yalnız bir okuma bekler. Serbest bırakılan çerçeve de hemen silinmez. İçeriği yerinde kalır ve sayfa yeniden gerekirse havuzdan geri alınabilir; bu durumda disk okuması yerine küçük bir hata yeter.

Bu işi saat algoritmasıyla yapmanın bir yolu **iki elli saattir**. Öndeki el kirli sayfaları yazdırır ve R bitlerini temizler. Arkadaki el belli bir mesafeden onu izler ve hâlâ R = 0 olan, yani iki el arasındaki sürede kullanılmayan sayfaları serbest bırakır. Elleri arasındaki mesafe, bir sayfaya “kullanılmak için” verilen süreyi belirler.

Linux'ta bu daemonun adı `kswapd`'dir ve bellek bölgesi başına üç su düzeyiyle çalışır: min, low ve high (Şekil 9.43). Boş sayfa sayısı low'un altına inince kswapd uyanır ve sayfaları arka planda geri kazanır. Sayı high'a çıkınca yeniden uykuya döner. Sayı min'in de altına inerse ayırma yapan süreç beklemeden işe girişir ve geri kazanımı kendisi yapar; buna **doğrudan geri kazanım** denir ve gecikmeyi artırır. Kirli dosya sayfalarını ayrıca writeback iş parçacıkları diske yazar.



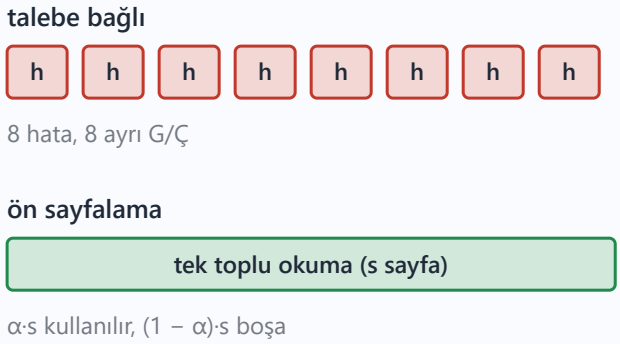
Şekil 9.43: Boş sayfa su düzeyleri: high'ın üstünde kswapd uyur, low'un altına inince uyanır; min'in altında ayırma yapan süreç doğrudan geri kazanım yapar.

9.11.2 Ön sayfalama

Talebe bağlı sayfalama, süreç başlarken ve askıdan dönerken bir hata fırtınası yaşar: her sayfa ayrı bir hatayla, ayrı bir G/Ç'yle gelir.

> **Tanım 9.30**
>
> **Ön sayfalama** (*prepaging*), gerekeceği tahmin edilen sayfaları hata beklemeden, tek bir toplu okumayla önceden getirmektir.

Ön sayfalamanın kârlı olup olmadığı basit bir hesapla görülür. s sayfa önceden getirilsin ve bunların α oranı gerçekten kullanılsın ($0 \le \alpha \le 1$). Kazanç $\alpha \cdot s$ sayfa hatasının önlenmesidir. Bedel $(1 - \alpha) \cdot s$ gereksiz sayfanın okunması ve bellekte yer tutmasıdır. α 1'e yakınsa ön sayfalama kazandırır, 0'a yakınsa kaybettirir (Şekil 9.44).



Şekil 9.44: Talebe bağlı sayfalamada her sayfa ayrı bir hatayla gelir (8 hata, 8 ayrı G/Ç). Ön sayfalamada çalışma kümesi tek toplu okumayla gelir; α·s sayfa kullanılır, (1 − α)·s boşa gider.

Örnek 9.16

Askıdan dönen bir sürecin çalışma kümesindeki $s = 40$ sayfa önceden getiriliyor ve bunların $\alpha = 0{,}6$'sı kullanılıyor. Bir sayfayı tek tek hatayla getirmek 5 ms, 40 sayfayı tek seferde okumak 30 ms sürüyor. Ön sayfalama kârlı mı? α hangi değerin altında zarar ettirir?

1. **Kullanılan sayfa.** $\alpha \times s = 0{,}6 \times 40 = 24$.
2. **Ön sayfalamasız.** 24 hata × 5 ms = 120 ms.
3. **Ön sayfalamalı.** Tek toplu okuma, 30 ms. Kazanç 90 ms'dir.
4. **Eşik.** Kazanç $40\alpha \times 5 = 200\alpha$ ms, bedel 30 ms: $200\alpha > 30 \rightarrow \alpha > 0{,}15$.
5. **Gizli bedel.** $(1 - \alpha) \times s = 16$ gereksiz sayfa çerçeve tutar ve başka sayfaların atılmasına yol açabilir.

Bu örnekte eşik düşüktür, çünkü toplu okuma çok ucuzdur. Toplu okumanın maliyeti tek tek hatalarınkine yaklaştıkça eşik yükselir. Klasik uygulama, askıya alınan sürecin çalışma kümesini hatırlayıp geri dönüşte yüklemektir. Linux dosyalar için önden okuma (*readahead*), takas için `page-cluster` ayarını kullanır. Uygulama da ipucu verebilir: `madvise(MADV_WILLNEED)` ya da `mmap(MAP_POPULATE)`. Windows'ta Prefetcher ve SysMain (eski adı SuperFetch) sık kullanılan sayfaları sistem boştayken yükler.

9.11.3 Sayfa boyutu

Sayfa boyutunu genellikle donanım belirler, ama yeni bir mimari tasarlanırken bu bir seçimdir ve birbirine karşı çeken etkenler vardır (Tablo 9.12).

Küçük sayfa	Büyük sayfa
az iç parçalanma	küçük sayfa tablosu

Küçük sayfa	Büyük sayfa
yerelliği ince izler, az gereksiz veri getirir	az sayfa hatası, verimli G/Ç (arama ve kurma maliyeti bir kez ödenir)
çok TLB ıskası	büyük TLB erimi

Tablo 9.12: Küçük ve büyük sayfanın üstünlükleri.

İki etkeni basit bir modelle tartmak mümkündür. Süreç boyutu s bayt, sayfa tablosu girdisi e bayt, sayfa boyutu p bayt olsun. Sayfa tablosu s/p girdi, yani se/p bayt tutar. Sürecin son sayfası ortalama yarı boş kalır ve $p/2$ bayt iç parçalanma yaratır.

Teorem 9.2 (En uygun sayfa boyutu)

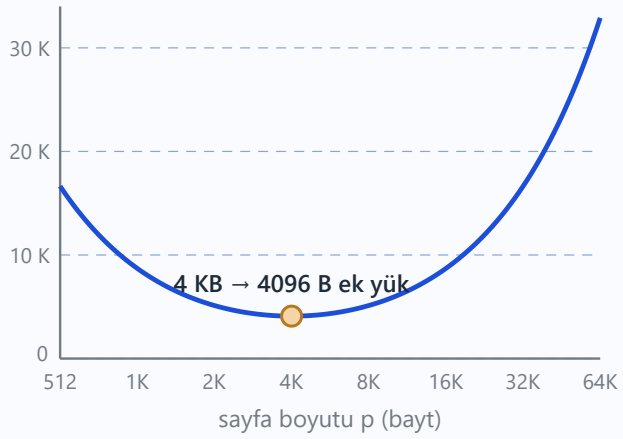
Ek maliyet $f(p) = se/p + p/2$ ise en küçük değer $p = \sqrt{(2se)}$ noktasındadır.

Kanıt türev almaktır: $f'(p) = -se/p^2 + 1/2 = 0$ denkleminden $p^2 = 2se$ çıkar. İkinci türev pozitif olduğu için bu nokta bir en küçük değerdir.

Örnek 9.17

s = 1 MB, e = 8 B için en uygun sayfa boyutu nedir? s = 4 MB olursa ne değişir?

1. **İkinin kuvvetleri.** 1 MB = 2^{20}, 8 = 2^3. $2se = 2 \times 2^{20} \times 2^3 = 2^{24}$.
2. **Karekök.** $p = \sqrt{(2^{24})} = 2^{12} = 4.096$ B = 4 KB.
3. **Ek yük.** $f(4.096) = 2^{23} / 2^{12} + 2.048 = 2.048 + 2.048 = 4.096$ B. 2 KB'da 5.120, 8 KB'da yine 5.120 B olur.
4. ***s* = 4 MB.** $2 \times 2^{22} \times 2^3 = 2^{26}$, $\sqrt{(2^{26})} = 2^{13}$ = 8 KB.



Şekil 9.45: s = 1 MB ve e = 8 B için ek maliyet se/p + p/2. En küçük değer 4 KB'de, 4.096 B'dir.

Model kabadır; TLB'yi ve G/Ç'yi hiç hesaba katmaz. Yine de 4 KB'ın neden onlarca yıl standart kaldığını açıklar. 2026 itibarıyla x86-64 4 KB sayfa kullanır ve 2 MB ile 1 GB'lık büyük sayfaları destekler. Apple Silicon 16 KB sayfa kullanır. Android 15 ve sonrası 16 KB sayfalı çekirdekleri destekler. Bazı ARM64 sunucular 64 KB sayfayla çalışır. 1 KB'lık sayfalar yalnız eski sistemlerde görülür. Eğilim, bellek büyüdükçe sayfanın da büyümesidir.

9.11.4 TLB erimi ve büyük sayfalar

Tanım 9.31

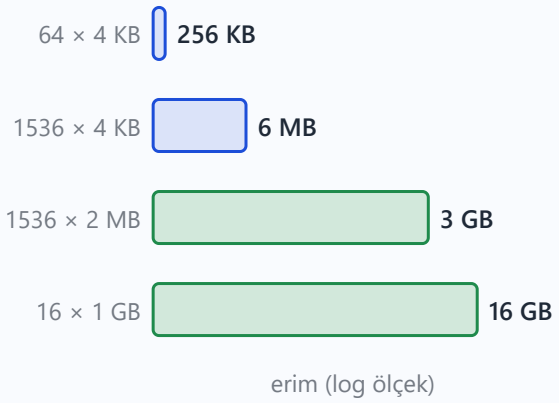
TLB erimi (*TLB reach*), TLB'den ıskasız erişilebilen bellek miktarıdır: TLB girdi sayısı × sayfa boyutu.

Sürecin çalışma kümesi erime sığmıyorsa erişimlerin önemli bir kısmı TLB ıskasıyla sonuçlanır ve zaman sayfa tablosu yürüyüşüne gider. Erimi artırmanın iki yolu vardır. Girdi sayısını artırmak pahalıdır, çünkü TLB ilişkisel bellektir ve her girdi paralel karşılaştırma devresi ister. Sayfayı büyütmek çok daha etkilidir (Şekil 9.46).

Örnek 9.18

(a) 64 girdili bir TLB'nin 4 KB ve 2 MB sayfalarla erimi nedir? (b) Bir veritabanının sıcak verisi 48 MB; ikinci düzey TLB 1.024 girdilidir. 4 KB ve 2 MB sayfalarla veri erime sığar mı?

1. **(a)** 64 × 4 KB = 256 KB; 64 × 2 MB = 128 MB.
2. **(b) 4 KB.** 1.024 × 4 KB = 4 MB. 48 MB'lık veri sığmaz; erişimlerin çoğu TLB ıskası olur.
3. **(b) 2 MB.** 1.024 × 2 MB = 2 GB. Veri rahatça sığar.



Şekil 9.46: TLB erimi, logaritmik ölçekte: 64 × 4 KB = 256 KB, 1536 × 4 KB = 6 MB, 1536 × 2 MB = 3 GB, 16 × 1 GB = 16 GB.

x86-64'te büyük sayfalar 2 MB ve 1 GB'tır. Linux iki yol sunar: uygulamanın açıkça istediği `hugetlbfs` ve çekirdeğin kendiliğinden kullandığı **şeffaf büyük sayfalar** (THP, *transparent huge pages*). THP'de `khugepaged` iş parçacığı ardışık küçük sayfaları arka planda büyük sayfalarda birleştirir. 6.x çekirdekleri ayrıca çok boyutlu folyo ve mTHP desteğiyle 4 KB ile 2 MB arasındaki boyutları da kullanabilir. Windows'ta büyük sayfa kullanmak için özel bir ayrıcalık gerekir.

Büyük sayfanın bedelleri de vardır: iç parçalanma, büyük bitişik fiziksel bölge bulma zorunluluğu (bunun için bellek sıkıştırma gerekebilir), sayfa başına daha uzun kopyalama ve sıfırlama süresi. Büyük sayfadan en çok yararlanan yazılımlar veritabanları ve sanal makinelerdir; ikisi de büyük ve sık erişilen bellek bölgeleri kullanır.

9.11.5 Program yapısı ve yerellik

Talebe bağlı sayfalama programcıya görünmez olmalıdır, ama programcının seçimleri hata sayısını büyük ölçüde değiştirebilir. En bilinen örnek iki boyutlu dizinin gezilme sırasıdır. C dizileri **satır öncelikli** (*row-major*) saklar: `data[i][0]`, `data[i][1]`, … bellekte ardışıktır.

```
int data[128][128];   /* satır = 1 sayfa */
/* A: sütun sütun */
for (int j = 0; j < 128; j++)
  for (int i = 0; i < 128; i++)
    data[i][j] = 0;
/* B: satır satır */
for (int i = 0; i < 128; i++)
  for (int j = 0; j < 128; j++)
    data[i][j] = 0;
```

Dizi 128 × 128 `int`'tir; sayfa 128 sözcüktür, yani her satır tam bir sayfadır. Sürece 64 çerçeve ayrılmıştır, başta hiçbiri dolu değildir ve yer değiştirme LRU'dur.

Örnek 9.19

A ve B döngüleri kaçar sayfa hatası üretir?

1. **A, sütun 0.** İç döngü `i` üzerindedir: `data[0][0]`, `data[1][0]`, … Her `i` farklı bir satır, yani farklı bir sayfadır. Sütun 0 boyunca 128 hata olur. 128 sayfa 64 çerçeveye sığmaz; sütun bittiğinde sayfa 0–63 çoktan atılmıştır.
2. **A, sütun 1.** `data[0][1]` yine sayfa 0'dadır, ama sayfa 0 atılmıştır: yeniden hata. LRU her seferinde en yakında gerekecek sayfayı atar.
3. **A, toplam.** Her erişim hatadır: 128 × 128 = 16.384 hata.
4. **B, satır 0.** `data[0][0]` sayfa 0'da hata yapar; ardından `data[0][1…127]` aynı sayfadadır ve isabettir.
5. **B, toplam.** Her satıra tek hata: 128 hata. Aynı iş, 128 kat daha az hata.



Şekil 9.47: 128 × 128 dizinin ölçekli gösterimi: her satır bir sayfadır. Satır satır gezen B döngüsü her satırda tek hata yapar, toplam 128.

Aynı ilke başka boyutlarda da geçerlidir. 64 × 1024'lük bir `int` dizisinde (her satır 4 KB = bir sayfa) 32 çerçeveyle satır satır doldurma 64, sütun sütun doldurma 64 × 1.024 = 65.536 hata yapar: 1.024 kat fark. Fortran ise dizileri sütun öncelikli saklar ve orada doğru sıra tersidir.

Veri yapısı seçimi de yerelliği etkiler. Yığın iyi yerellik gösterir: erişimler hep tepeye yakındır. Hash tablosu ise erişimleri bilerek dağıtır ve kötü yerellik gösterir. İşaretçi kovalayan bağlı yapılar da düğümleri belleğe dağıttığı için kötüdür. Derleyiciler bazen **döngü değiştirme** (*loop interchange*) dönüşümüyle iç ve dış döngünün yerini kendiliğinden değiştirir. Derleyici ve yükleyici de kodu sayfa sınırlarını gözeterek yerleştirebilir: birbirini çağıran fonksiyonları aynı sayfaya koymak gibi.

9.11.6 G/Ç kilitleme ve sayfa sabitleme

Bazı sayfaların, en azından bir süre için, kesinlikle atılmaması gerekir. En önemli örnek G/Ç tamponlarıdır.

Örnek 9.20

Süreç A diskten okuma için bir tampon veriyor. Genel yer değiştirme politikası kullanılıyor. Ne ters gidebilir?

1. **İstek.** Çekirdek doğrudan bellek erişimini (DMA) tamponun bulunduğu çerçeve 42'nin *fiziksel* adresiyle programlar.
2. **Bekleme.** A bloke olur; aygıt kuyrukta bekler ya da çalışır. Çerçeve 42 bu sırada sıradan bir sayfa gibi görünür.
3. **Yer değiştirme.** B bir sayfa hatası yapar. Genel politika çerçeve 42'yi kurban seçer ve B'nin sayfasını oraya yükler.
4. **Bozulma.** DMA tamamlanınca aygıt veriyi çerçeve 42'ye yazar, yani B'nin sayfasının üstüne. Veri yanlış sürecin belleğine gitmiştir.



Şekil 9.48: G/Ç kilitleme. A'nın tamponu olan çerçeve 42 DMA sürerken kilitlidir; kilit biti 1 olduğu için kurban seçilemez.

Tanım 9.32

Sayfa sabitleme ya da **kilitleme** (*pinning, page locking*), bir çerçeveye kilit biti koyarak onu yer değiştirme adaylarının dışında tutmaktır.

Sorunun iki çözümü vardır. Birincisi, G/Ç süresince tampon çerçevesini kilitlemektir. İkincisi, G/Ç'yi her zaman çekirdek tamponlarına yapıp veriyi sonra kullanıcı tamponuna kopyalamaktır; bu güvenlidir ama fazladan bir kopya ister.

Sabitlenen başka sayfalar da vardır. Çekirdek kodunun ve verisinin bir kısmı hiç atılmaz; sayfa hatası işleyicisinin kendisi atılsaydı onu kim getirirdi? Gerçek zamanlı süreçler gecikmeye tahammül edemediği için belleklerini `mlock()` ya da `mlockall()` ile kilitler. Linux sürücüleri kullanıcı sayfalarını `pin_user_pages()` ile sabitler; RDMA ağ kartları, GPU'lar ve io_uring'in sabit tamponları bunu kullanır. IOMMU ise aygıtların da çevrilmiş adres kullanmasını sağlar ve yanlış adrese DMA yapılmasını donanımda engeller.

Bir öncelik inceliği de vardır. Düşük öncelikli bir süreç, uzun bir beklemeden sonra sayfasını yeni getirmiş ama henüz hiç çalışmamış olabilir. Bu sayfanın R biti 0'dır ve yüksek öncelikli bir sürecin hatası onu hemen kurban seçebilir. Getirme emeği boşa gider. Bu yüzden yeni yüklenmiş sayfa bir kez kullanılana kadar kilitli tutulabilir.

Kilitlemenin riski de açıktır: kilitli bir sayfa unutulursa çerçeve sonsuza dek kullanılamaz. Çok sayıda kilit belleği daraltır ve thrashing'i kolaylaştırır. Bu yüzden kullanıcının kilitleyebileceği bellek sınırlıdır; Linux'ta sınır `ulimit -l` ile görülür.

9.12 Çekirdek belleği

Kullanıcı süreçlerinin belleği sayfalanır ve boş çerçeve havuzundan gelir. Çekirdeğin kendi belleği ise farklı kurallara uyar. Çekirdek belleğinin çoğu sayfalanmaz, çünkü sayfa hatası işleyicisi gibi kodlar hata almamalıdır. Çekirdek sayfa boyutundan çok daha küçük yapılar ayırır (süreç tanımlayıcıları,

dosya nesneleri, semaforlar) ve sayfa düzeyindeki ayırma bunlar için israftır. Bazı aygıtlar DMA için fiziksel olarak bitişik bellek ister, çünkü sanal adresleri bilmezler. Bu ihtiyaçlar için iki ayırıcı kullanılır.

9.12.1 Buddy sistemi

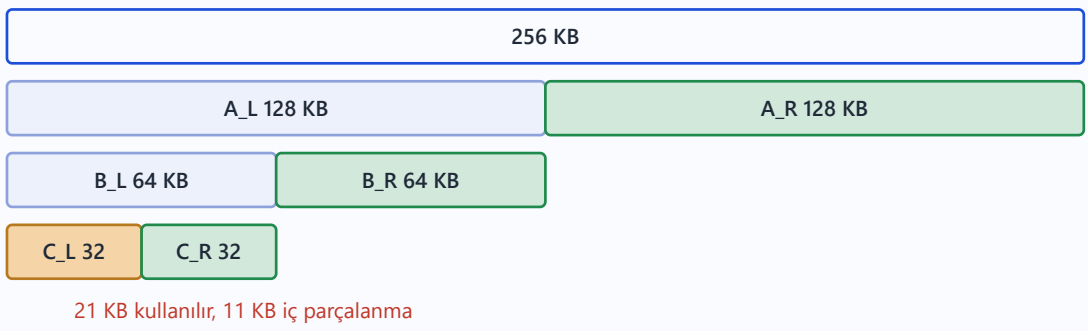
Tanım 9.33

Buddy (kardeş) **sistemi**, fiziksel olarak bitişik bir bölgeyi yalnız 2'nin kuvveti boyutlarında bloklara bölen ayırıcıdır. İstek, onu karşılayan en küçük 2'nin kuvvetine yukarı yuvarlanır. Uygun boyutta boş blok yoksa daha büyük bir blok iki eş yarıya, iki **kardeşe** bölünür. Bir blok bırakılınca kardeşi de boşsa ikisi birleşir.

Örnek 9.21

Çekirdeğin elinde fiziksel olarak bitişik 256 KB var. 21 KB'lık bir istek geliyor. Bölmeler nasıl olur, iç parçalanma ne kadardır?

1. **Yuvarlama.** 21 KB'ı karşılayan en küçük 2'nin kuvveti 32 KB'tır.
2. **İlk bölme.** 256 KB, 128'er KB'lık A_L ve A_R kardeşlerine bölünür.
3. **İkinci bölme.** A_L, 64'er KB'lık B_L ve B_R'ye bölünür.
4. **Üçüncü bölme.** B_L, 32'şer KB'lık C_L ve C_R'ye bölünür. C_L isteğe verilir.
5. **Bedel.** İstenen 21, verilen 32 KB: 11 KB iç parçalanma.



Şekil 9.49: Buddy sistemi: 256 KB'lık blok 21 KB'lık istek için 128, 64 ve 32 KB'lık kardeşlere bölünür; C_L isteğe verilir.

Buddy sisteminin asıl güçlü yanı birleştirmedir (*coalescing*). C_L bırakılınca kardeşi C_R boşsa ikisi B_L'yi yeniden oluşturur; B_R de boşsa A_L oluşur ve bu böyle sürer. Kardeşin adresini bulmak tek bir işlemdir: k boyutlu bir bloğun kardeşinin adresi, blok adresinin k'ya karşılık gelen bitinin çevrilmesiyle bu-

lunur (adres XOR boyut). Linux'un sayfa ayırıcısı bir buddy ayırıcısıdır. 0'dan 10'a kadar mertebe için ayrı boş listeler tutar: 4 KB'tan 4 MB'a kadar. Listelerin durumu `cat /proc/buddyinfo` ile görülür.

9.12.2 Kod izleme: buddy ayırıcı

Aşağıdaki kod buddy ayırıcının özünü gösterir. Boyutlar KB cinsinden 2'nin kuvveti olarak tutulur: `ayir(k)` 2^k KB'lık bir blok ister. `bos[j]`, 2^j KB'lık boş blokların listesidir. `cek(j)` listeden bir blok alır, `ekle(j, a)` listeye ekler, `sil(k, b)` listede varsa çıkarıp doğru döndürür.

```
int ayir(int k) {          /* 2^k KB */
  int j = k;
  while (bos[j] == NULL) j++;
  int a = cek(j);
  while (j > k) {   /* böl, sağ yarı boş */
    j--; ekle(j, a + (1 << j));
  }
  return a;
}
void birak(int a, int k) {
  int b = a ^ (1 << k);    /* kardeş */
  while (k < MAX && sil(k, b)) {
    a = a < b ? a : b;     /* birleş */
    k++; b = a ^ (1 << k);
  }
  ekle(k, a);
}
```

`ayir()` önce boş blok bulunan en küçük `j ≥ k` listesini arar ve oradan bir blok çeker. Sonra blok istenen boya inene kadar ikiye bölünür: her bölmede sağ yarı (`a + (1 << j)`) boş listeye eklenir, sol yarı bölünmeye devam eder. `birak()` kardeşin adresini `a ^ (1 << k)` ile bulur. Kardeş aynı boydaki boş listedeyse onu listeden çıkarır, iki bloğun küçük adreslisini yeni blok yapar ve bir üst boyda yeniden dener. Kardeş boş değilse ya da en büyük boya (`MAX`) ulaşıldıysa blok listeye eklenir.

256 KB'lık bölgeyle (`MAX` = 8) başlanır ve sırayla `A = ayir(5)`, `B = ayir(6)`, `birak(0, 5)`, `birak(64, 6)` çağrılır (Tablo 9.13).

Çağrı	Olay	bos[5]	bos[6]	bos[7]	bos[8]
başlangıç	tek blok	—	—	—	@0
`ayir(5)`	j = 8, @0 çekilir; bölmeler @128, @64, @32; A = 0	@32	@64	@128	—
`ayir(6)`	bos[6]'da @64 hazır; B = 64	@32	—	@128	—
`birak(0, 5)`	kardeş 0 ⊕ 32 = 32 boş → @0 (64); kardeş 0 ⊕ 64 = 64 dolu	—	@0	@128	—
`birak(64, 6)`	kardeş 64 ⊕ 64 = 0 boş → @0 (128); kardeş @128 boş → @0 (256)	—	—	—	@0

Tablo 9.13: Buddy ayırıcı izlemesi. Adresler KB, "@" blok başlangıcıdır.

- `A = ayir(5)`. 21 KB'lık istek 32 KB'a (2^5) yuvarlanmıştır. bos[5], bos[6] ve bos[7] boştur; ilk dolu liste bos[8]'dir: j = 8. @0'daki 256 KB'lık blok çekilir. j = 7'de sağ yarı @128, j = 6'da @64, j = 5'te @32 boş listelere girer. j = k olunca döngü biter ve sol yarı, @0'daki 32 KB, A'ya verilir.
- `B = ayir(6)`. bos[6]'da @64 hazırdır; bölme gerekmez. A'nın iç parçalanması 32 − 21 = 11 KB'tır.
- `birak(0, 5)`. Kardeşin adresi tek bit çevrilerek bulunur: 0 ⊕ 32 = 32. @32 bos[5]'tedir; birleşip @0'da 64 KB olurlar. Yeni kardeş 0 ⊕ 64 = 64'tür, ama @64'ü B kullanıyor. Birleşme durur, @0 bos[6]'ya eklenir.
- `birak(64, 6)`. Kardeş 64 ⊕ 64 = 0 boştur; 128 KB'lık @0 oluşur. Yeni kardeş @128 de boştur ve birleşip 256 KB olurlar. k = MAX olduğundan döngü durur; bölge başlangıçtaki gibi tek parçadır.



Şekil 9.50: `ayir(5)` sonunda bölge: A @0'da 32 KB; @32 (32 KB), @64 (64 KB) ve @128 (128 KB) boş listelerdedir.

Ayırma ve bırakma, bölme ve birleşme sayısı kadar adım sürer; bu da en çok mertebe sayısı kadardır, yani O(log *n*). Boyut başına ayrı boş liste tutulduğu için arama da hızlıdır. Kardeşin adresinin XOR ile bulunması, blokların hep kendi boyutunun katı olan adreslerde başlamasına dayanır.

9.12.3 Buddy: istek dizisi örneği

Örnek 9.22

1 MiB (1.024 K) tek boş blokla başlanıyor. Sırasıyla A = 70 K, B = 35 K, C = 80 K isteniyor; A bırakılıyor; D = 60 K isteniyor; B, D ve C bırakılıyor. Her adımda ne olur?

1. **A = 70 K.** 128 K'ya yuvarlanır. 1.024 → 512 + 512 → 256 + 256 → 128 + 128 bölünür; A, 0–128'i alır. İç parçalanma 58 K.
2. **B = 35 K.** 64 K'ya yuvarlanır. Boş 128–256 bloğu 64 + 64'e bölünür; B 128–192'yi alır. İç parçalanma 29 K.
3. **C = 80 K.** 128 K'ya yuvarlanır. Boş 256–512 bloğu 128 + 128'e bölünür; C 256–384'ü alır. İç parçalanma 48 K.
4. **A bırakılır.** Kardeşi 128–256 bölünmüş durumdadır ve içinde B vardır: birleşme olmaz.

5. **D = 60 K.** 64 K'ya yuvarlanır. B'nin boş kardeşi 192–256 verilir, bölme gerekmez. İç parçalanma 4 K.
6. **B bırakılır.** Kardeşi D doludur: birleşme yok.
7. **D bırakılır.** 128–192 ile 192–256 birleşip 128–256 olur; A'nın boş 0–128'iyle birleşip 0–256 olur. Kardeşi 256–512'de C vardır: dur.
8. **C bırakılır.** 256–384 + 384–512 → 256–512; 0–256 ile → 0–512; 512–1024 ile → 0–1024. Bellek yine tek bloktur.

A, B ve C için toplam iç parçalanma 58 + 29 + 48 = 135 K'dır. Buddy ayırma ve birleştirmede hızlıdır, ama en kötü durumda bloğun yaklaşık yarısı boşa gider: $2^k + 1$ baytlık bir istek 2^{k+1} baytlık blok alır. Bu yüzden küçük nesneler için buddy'nin üstüne ikinci bir ayırıcı kurulur.

Sık yapılan hata

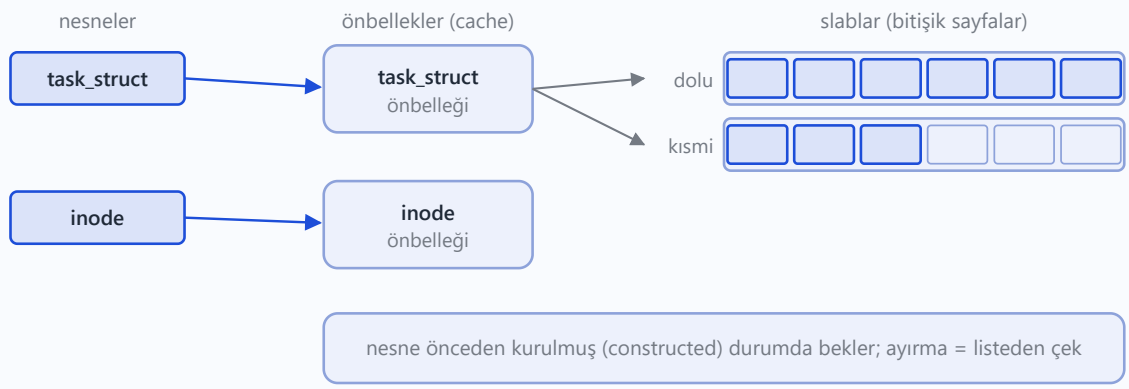
Komşu iki boş bloğun her zaman birleştirilebileceğini sanmak. Yalnız *kardeşler* birleşir: aynı bloğun bölünmesinden doğan, aynı boydaki iki yarı. Örneğin 64–128 ile 128–192 komşu ve aynı boydadır, ama kardeş değildir; 64'ün kardeşi 0, 128'in kardeşi 192'dir.

9.12.4 Slab ayırıcı

Tanım 9.34

Slab ayırıcı (*slab allocator*), sık kullanılan her çekirdek nesnesi türü için ayrı bir **önbellek** tutar. Önbellek bir ya da daha çok **slab**'dan, her slab fiziksel olarak bitişik bir ya da birkaç sayfadan oluşur. Slab, o türden nesnelerle tam boyutlu olarak paketlenir.

Bir slab üç durumdan birindedir: dolu (bütün nesneleri kullanımda), kısmi (bir kısmı kullanımda) ve boş. Nesneler önbellekte önceden kurulmuş durumda bekler; ayırma, listeden bir nesne çekmektir (Şekil 9.51).



Şekil 9.51: Slab ayırıcı: task_struct ve inode nesneleri için ayrı önbellekler; her önbellekte dolu ve kısmi slablar, slablar bitişik sayfalardan oluşur.

Bir isteğin yolu şöyledir. `fork()` yeni bir `task_struct` ister. Ayırıcı önce kısmi slablardaki boş bir nesneye bakar; bulursa onu verir. Kısmi slab dolarsa boş slablara geçilir. Kısmi ya da boş slab kalmadıysa buddy ayırıcıdan yeni sayfalar alınır ve yeni bir slab kurulur. Bellek baskısı olduğunda boşalan slablar buddy ayırıcıya geri verilir.

Slab ayırıcıyı Bonwick 1994'te Solaris için tasarladı. Yararları şunlardır:

- Parçalanma yoktur: nesneler tam boyutludur ve sayfa içinde yan yana paketlenir.
- Ayırma ve bırakma O(1)'dir; bırakılan nesne yalnız “boş” işaretlenir.
- Önbellekteki nesneler sıcak ve hazırdır; her ayırmada yeniden kurulmaları gerekmez.
- CPU başına listeler kilitsiz bir hızlı yol sağlar.

Linux'ta slab ayırıcının üç gerçeklemesi uzun süre birlikte yaşadı: SLAB, SLUB ve SLOB. SLOB 6.4 sürümünde, klasik SLAB 6.8 sürümünde (2024) kaldırıldı; bugün tek ayırıcı SLUB'dır. Genel amaçlı `kmalloc()` de boyut sınıflarına ayrılmış slab önbelleklerinden beslenir. Önbelleklerin durumu `slabtop` ve `/proc/slabinfo` ile izlenir.

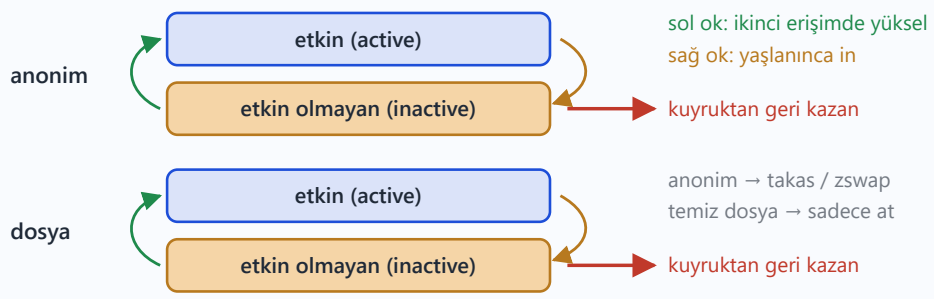
9.13 Gerçek sistemler

9.13.1 Linux: sayfa geri kazanımı

Linux'ta geri kazanım su düzeyleriyle tetiklenir. Bir bölgenin boş sayfa sayısı low'un altına inince kswapd uyanır; min'in altına inerse ayırma yapan süreç doğrudan geri kazanım yapar ve gecikme artar.

Klasik Linux geri kazanımı saate benzeyen iki liste çifti kullanır: **etkin** (*active*) ve **etkin olmayan** (*inactive*) listeler. Anonim ve dosya sayfaları için ayrı birer çift vardır (Şekil 9.52). Yeni sayfa etkin olmayan listeye girer; yeniden erişilirse etkin listeye yükselir. Etkin listenin kuyruğundaki sayfalar yaş-

landıkça etkin olmayan listeye iner. Kurbanlar etkin olmayan listenin kuyruğundan seçilir. Anonim kurban takasa ya da zswap'a gider, temiz dosya sayfası yalnız atılır. Kilitli sayfalar için ayrıca atılamaz (*unevictable*) bir liste vardır.



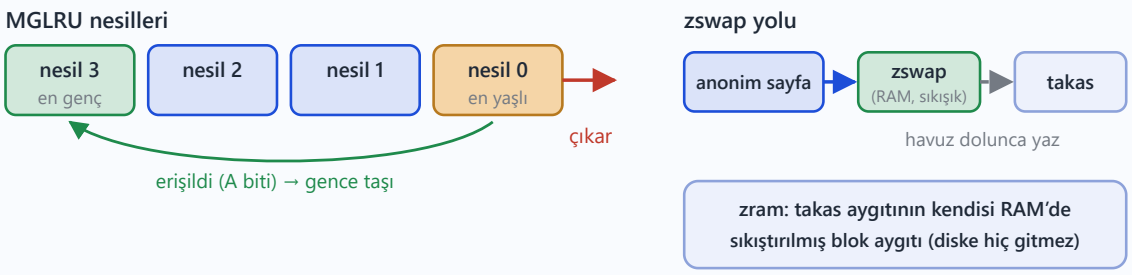
Şekil 9.52: Linux geri kazanım listeleri: yeni sayfa etkin olmayan listeye girer, ikinci erişimde etkin listeye yükselir; yaşlanan sayfa geri iner ve kurban etkin olmayanın kuyruğundan seçilir.

Bu iki liste düzeni, döngü örüntüsünün LRU'yu vurmasına karşı bir önlemdir: bir kez okunan dosya sayfaları etkin olmayan listede kalır ve sıcak sayfaları kovamaz. `vm.swappiness` ayarı (0–200) anonim sayfaları takasa atmakla dosya sayfalarını bırakmak arasındaki tercihi belirler. Bellek gerçekten tükenirse OOM öldürücü bir süreci seçip sonlandırır.

9.13.2 Linux: MGLRU, zswap ve zram

MGLRU (*multi-generational LRU*, çok nesilli LRU) 2022'de Linux 6.1 ile geldi. Sayfalar iki liste yerine dört nesle ayrılır. Erişim bitleri sayfa tabloları yürünerek toplu hâlde toplanır; erişilen sayfa en genç nesle taşınır, kurban en yaşlı nesilden seçilir (Şekil 9.53). Sonuç daha az CPU harcaması ve daha iyi kurban seçimidir. MGLRU ChromeOS ve Android'de varsayılandır; birçok dağıtım çekirdeği derleme seçeneğiyle açar. Durumu `/sys/kernel/mm/lru_gen/enabled` dosyasından okunur.

zswap, takasın önünde duran sıkıştırılmış bir önbellektir. Anonim sayfa diske gitmeden önce RAM'de sıkıştırılır ve bir havuzda tutulur. Havuz dolunca en eski girdiler gerçek takas aygıtına yazılır. **zram** ise RAM'de sıkıştırılmış bir blok aygıtıdır ve doğrudan takas alanı olarak kullanılır; sayfalar diske hiç gitmez. Fedora (33. sürümden beri), Android ve ChromeOS zram'ı varsayılan olarak kullanır.



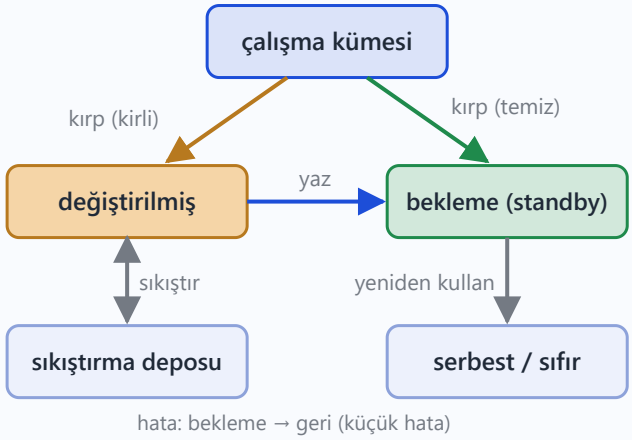
Şekil 9.53: MGLRU: sayfalar nesillere ayrılır, en yaşlı nesilden çıkarılır, erişilen sayfa en genç nesle taşınır. zswap anonim sayfayı sıkıştırıp RAM'de tutar, havuz dolunca takasa yazar; zram takas aygıtının kendisidir.

Sıkıştırma CPU harcar, ama diske gitmekten yüzlerce kat ucuzdur. EAT hesabında zram'ın 10 µs'lik hata süresi bunu sayılarla gösterir: p = 0,001'de HDD 251 kat, zram yalnız %10 yavaşlama getirir.

9.13.3 Windows: çalışma kümeleri ve sıkıştırma

Windows, Denning'in modelini adıyla kullanır. Her sürecin bir **çalışma kümesi** vardır ve kümenin en az ve en çok sınırları belirlenmiştir. Sayfa hatası kümede yer açar ya da kümeyi büyütür. Bellek azalınca **çalışma kümesi yöneticisi** (*balance set manager*) kümeleri kırpar.

Kırpılan sayfa hemen silinmez (Şekil 9.54). Temizse **bekleme** (*standby*) listesine, kirliyse **değiştirilmiş** (*modified*) listeye gider. Değiştirilmiş sayfalar yazılınca bekleme listesine geçer. Bekleme listesindeki bir sayfa yeniden istenirse küçük bir hatayla geri alınır. Serbest ve sıfırlanmış listeler yeni ayırmaları besler.



Şekil 9.54: Windows sayfa listeleri: çalışma kümesinden kırpılan temiz sayfa bekleme listesine, kirli sayfa değiştirilmiş listeye gider. Sıkıştırma deposu ve serbest/sıfır listeleri de akışın parçasıdır.

Windows 10 (2015) ile **bellek sıkıştırma** geldi: kırpılan sayfalar sistem sürecindeki bir sıkıştırma deposunda tutulur; `pagefile.sys` dosyasına yazmak son çaredir. Önceden getirme işini Prefetcher ve SysMain yapar. Büyük sayfalar `VirtualAlloc(MEM_LARGE_PAGES)` ile istenir.

macOS ve iOS da benzer bir yol izler: sıkıştırılmış bellek kullanılır. iOS'ta takas hiç yoktur; bellek baskısında sistem, Jetsam mekanizmasıyla arka plandaki uygulamaları sonlandırır. Üç sistemin ortak noktası, diske gitmeden önce bellekte kalan her imkânı denemektir.

9.14 Özet

- Sanal bellek mantıksal belleği fiziksel bellekten ayırır. Süreç kısmen bellekteyken çalışır; program fiziksel bellekle sınırlı kalmaz, aynı bellekte daha çok süreç tutulur ve paylaşım kolaylaşır.
- Talebe bağlı sayfalamada sayfa yalnız erişildiğinde getirilir. Geçersiz girdiye erişim sayfa hatası tuzağı üretir. Çekirdek referansı denetler, sayfayı yedekleme deposundan boş bir çerçeveye okur, tabloyu günceller ve komutu yeniden başlatır.
- Linux sayfa hatalarını küçük (G/Ç yok), büyük (diskten okuma) ve geçersiz (`SIGSEGV`) diye ayırır. Sayfalar dosya destekli, anonim ya da sıfırla doldurulan sayfalardır.
- Etkin erişim süresi EAT = $(1 - p) \times ma + p \times$ hata süresi'dir. HDD'de (25 ms) $p = 0{,}001$ belleği 251 kat yavaşlatır; %10 yavaşlama için $p < 4 \times 10^{-7}$ gerekir. NVMe ve zram cezayı küçültür, ama yok etmez.
- Yazınca kopyalama, `fork()` sonrasında sayfaları paylaştırır ve yalnız yazılan sayfayı kopyalar; kopyanın gerekip gerekmediğini çerçevenin referans sayısı belirler. `mmap()` dosyayı adres alanına eşler; `MAP_SHARED` değişiklikleri dosyaya, `MAP_PRIVATE` özel bir kopyaya yazar.
- Boş çerçeve yoksa bir kurban seçilir; kirli kurban önce diske yazılır. Algoritmalar referans dizisi üzerindeki hata sayısıyla karşılaştırılır.
- Ortak dizide (3 çerçeve) OPT 9, yaşlandırma 10, NRU ve geliştirilmiş saat 11, LRU 12, WSClock 13, saat 14, FIFO 15 hata yapar. OPT gerçeklenemez ama alt sınırdır.
- FIFO Belady anomalisi gösterebilir: 1 2 3 4 1 2 5 1 2 3 4 5 dizisinde 3 çerçeve 9, 4 çerçeve 10 hata yapar. LRU ve OPT yığın algoritmalarıdır ve anomali göstermez.
- Tam LRU donanım ister (sayaç, yığın, matris). Pratikte R ve M bitlerine dayalı yaklaşımlar kullanılır: ikinci şans, saat, NRU, geliştirilmiş saat, NFU ve yaşlandırma. Yaşlandırma LRU'ya iyi yaklaşır; NFU hiç unutmadığı için yaklaşamaz.
- Çalışma kümesi WS(t, Δ), son Δ referansta kullanılan sayfalardır. Çalışma kümesi algoritması ve WSClock bu fikri sanal zaman ve τ eşiğiyle gerçekler.
- İş yükü politikayı belirler: yerelliksiz erişimde politikalar eşittir, sıcak sayfalarda LRU kazanır, bellekten biraz büyük döngüde LRU sıfır isabet verir.
- Çerçeveler eşit, orantılı ya da önceliğe göre dağıtılır; kurban yerel ya da genel seçilir. $\Sigma\ |WS_i| > m$ olunca thrashing başlar; çalışma kümesi stratejisi ve PFF çok programlama derecesini ayarlayarak onu önler.

- Tasarım kararları: sayfalama daemonu ve su düzeyleri, ön sayfalama (α eşiği), en uygun sayfa boyutu $\sqrt{(2se)}$, TLB erimi ve büyük sayfalar, satır öncelikli gezme, G/Ç tamponlarının sabitlenmesi.
- Çekirdek belleği buddy ayırıcıyla 2'nin kuvveti bloklar hâlinde ayrılır ve kardeşler XOR ile bulunup birleştirilir; küçük nesneler için slab (Linux'ta SLUB) kullanılır.
- Linux etkin/etkin olmayan listeler, MGLRU, zswap ve zram; Windows çalışma kümeleri, bekleme listeleri ve bellek sıkıştırma kullanır.

9.15 Kavram listesi

sanal bellek *virtual memory* · sanal adres *virtual address* · talebe bağlı sayfalama *demand paging* · sayfa hatası *page fault* · yedekleme deposu (takas alanı) *backing store (swap space)* · etkin erişim süresi *effective access time* · yazınca kopyalama *copy-on-write* · bellek eşlemeli dosya *memory-mapped file* · sayfa önbelleği *page cache* · sayfa yer değiştirme *page replacement* · referans dizisi *reference string* · referans biti / kirli bit *reference bit / dirty bit* · Belady anomalisi *Belady's anomaly* · optimal algoritma (OPT) *optimal page replacement* · en uzun süredir kullanılmayan (LRU) *least recently used* · ikinci şans *second chance* · saat algoritması *clock algorithm* · yaşlandırma *aging* · yerellik *locality* · çalışma kümesi *working set* · thrashing (boşa çabalama) *thrashing* · TLB *translation lookaside buffer* · iç parçalanma *internal fragmentation* · buddy (kardeş) sistemi *buddy system* · slab ayırıcı *slab allocator* · sıkıştırma *compression*

9.16 Kendini Dene

1. Referans dizisi 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6, çerçeve sayısı 4. FIFO, LRU ve OPT kaçar sayfa hatası yapar?

   Cevap

   FIFO 14, LRU 10, OPT 8. LRU'nun kurbanları sırasıyla 3, 4, 5, 6, 1, 7; OPT'ninkiler 4, 5, 1, 7'dir. 7 farklı sayfa olduğu için en az 7 hata kaçınılmazdır; OPT buna çok yaklaşır. 3 çerçeveyle sonuçlar FIFO 16, LRU 15, OPT 11'dir.

2. Bellek erişimi 50 ns, sayfa hatası hizmet süresi 10 ms. Yavaşlamanın %20'yi geçmemesi için p en

çok ne olabilir?

Cevap

%20 yavaşlama EAT < 60 ns demektir. 10 ms = 10.000.000 ns. 60 > 50 + (10.000.000 − 50) p → p < 10 / 9.999.950 ≈ 1,0 × 10^{-6}: yaklaşık bir milyon erişimde birden az hata.

3. Dört boş çerçeve ve 1 2 3 4 1 5 2 6 1 3 dizisiyle saat algoritması (el başta çerçeve 0'da, el yalnız hatada döner) kaç hata yapar? Hangi sayfalar çıkar?

Cevap

8 hata. İlk dört referans boşlara yüklenir; 1 isabettir. 5 gelince dört sayfanın da R'si 1'dir; el hepsini temizleyip bir tur atar ve 1'i çıkarır. 2 isabettir (R ← 1). 6 gelince el 2'nin R'sini temizler ve R'si 0 olan 3'ü çıkarır. 1 gelince elin altındaki 4 (R = 0) çıkar. 3 gelince el 5'in R'sini temizler, 2'yi çıkarır. Kurbanlar 1, 3, 4, 2'dir.

4. Referans dizisi (zaman 1'den) 1 2 1 3 2 4 4 5 4 5 6 1, Δ = 4. W(4), W(8) ve W(12) nedir?

Cevap

W(4) = {1, 2, 3} (1 2 1 3), W(8) = {2, 4, 5} (2 4 4 5), W(12) = {1, 4, 5, 6} (4 5 6 1). Kümeler süreç yeni sayfalara geçtikçe kayar; W(12) bir yerellik geçişi sırasında büyümüştür.

5. 2.048 girdili bir TLB'nin 16 KB'lık sayfalarla erimi nedir? Çalışma kümesi 48 MB olan bir süreç için yeterli midir?

Cevap

2.048 × 16 KB = 32.768 KB = 32 MB. 48 MB'lık çalışma kümesi sığmaz; erişimlerin bir kısmı TLB ıskasıyla sayfa tablosu yürüyüşüne gider. 2 MB'lık büyük sayfalar erimi 4 GB'a çıkarırdı.

6. Süreç boyutu 2 MB, sayfa tablosu girdisi 4 bayt. Ek maliyet $se/p + p/2$ modeline göre en uygun sayfa boyutu nedir?

Cevap

$p = \sqrt{(2se)} = \sqrt{(2 \times 2^{21} \times 2^{2})} = \sqrt{(2^{24})} = 2^{12}$ = 4 KB.

7. 128 K'lık bir buddy bölgesinde sırayla A = 10 K, B = 30 K, C = 20 K isteniyor; sonra A ve C bırakılı-

yor. Her istek nereye yerleşir, toplam iç parçalanma nedir, en sonda boş bloklar nelerdir?

Cevap

A 16 K'ya yuvarlanır: 128 → 64 + 64 → 32 + 32 → 16 + 16; A 0–16'yı alır (6 K). B 32 K: boş 32–64 hazırdır (2 K). C 32 K: boş 64–128 bölünür, C 64–96'yı alır (12 K). Toplam 20 K. A bırakılınca kardeşi 16–32 boştur, 0–32 oluşur; onun kardeşinde B vardır, dur. C bırakılınca kardeşi 96–128 boştur, 64–128 oluşur; kardeşi 0–64 bölünmüştür, dur. Boş bloklar 0–32 ve 64–128; B 32–64'tedir.

8. `fork()` sonrasında 50 sayfa yazınca kopyalamayla paylaşılıyor. Çocuk 0–4 numaralı sayfalara yazıyor; ardından ebeveyn 0, 1 ve 10 numaralı sayfalara yazıyor. Kaç sayfa kopyalanır?

Cevap

6. Çocuğun beş yazması beş kopya üretir. Ebeveyn 0 ve 1'e yazdığında bu çerçevelerin referans sayısı artık 1'dir; kopya gerekmez, sayfa yalnız yazılabilir yapılır. Sayfa 10 hâlâ paylaşılıyordur (referans sayısı 2) ve altıncı kopyayı üretir.

9. Üç sayfanın (s0, s1, s2) 8 bitlik yaşlandırma sayaçları başta 0. Dört tıkta görülen R bitleri: tık 0: 1 1 0, tık 1: 1 0 1, tık 2: 0 1 1, tık 3: 0 1 0. Yaşlandırma ve NFU hangi sayfayı atar?

Cevap

Sayaçlar s0 = 00110000 (48), s1 = 11010000 (208), s2 = 01100000 (96). Yaşlandırma s0'ı atar: son iki tıkta hiç kullanılmamıştır. NFU sayaçları 2, 3, 2'dir; s0 ile s2 eşittir ve NFU ikisini ayırt edemez. Oysa s2 bir tık önce kullanılmıştı ve daha yenidir.

10. Bir sunucuda CPU kullanımı %10'a düştü, takas diski sürekli meşgul. Bir yönetici “CPU boşta” diyerek kuyruğa yeni toplu işler eklemek istiyor. Ne olur? Doğru müdahale nedir?

Cevap

Sistem thrashing'dedir: $\Sigma\ |WS_i|$ belleği aşmıştır ve süreçler sayfa bekliyor. Yeni iş her sürecin çerçeve payını daha da küçültür, hatalar artar ve CPU kullanımı daha da düşer. Doğrusu çok programlama derecesini düşürmektir: bazı süreçleri askıya almak ya da takasa atmak, kalıcı çözüm olarak RAM eklemek. PFF bu kararı süreç başına hata oranıyla otomatik verir.

9.17 İleri okuma

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, “Swapping: Mechanisms”, “Swapping: Policies” ve “Complete VM Systems” bölümleri: yer değiştirme politikalarını yerelliksiz, 80-20 ve döngü iş yükleriyle karşılaştırır, VAX/VMS ve Linux'un sanal bellek tasa-

rımını anlatır; ostep.org'da ücretsizdir.

- P. J. Denning, “The Working Set Model for Program Behavior”, *Communications of the ACM*, 11(5), 1968: çalışma kümesi kavramını ve thrashing'e karşı kullanımını ortaya koyan temel makaledir.
- J. Bonwick, “The Slab Allocator: An Object-Caching Kernel Memory Allocator”, USENIX Summer 1994: nesne önbelleği fikrini ve Solaris'teki ölçümleri kısa ve okunaklı bir biçimde verir.
- Linux çekirdek belgeleri, `admin-guide/mm/multigen_lru.rst` ve `admin-guide/mm/zswap.rst`: MGLRU'nun nesillerini ve zswap'ın ayarlarını çekirdeğin kendi belgesinde anlatır.

9.18 Kaynaklar

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 10: Virtual Memory.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 3: Memory Management.
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — “Swapping: Mechanisms”, “Swapping: Policies”, “Complete VM Systems”.
4. L. A. Belady, “A Study of Replacement Algorithms for a Virtual-Storage Computer”, *IBM Systems Journal* 5(2), 1966; P. J. Denning, “The Working Set Model for Program Behavior”, *CACM* 11(5), 1968.
5. R. W. Carr, J. L. Hennessy, “WSCLOCK — A Simple and Effective Algorithm for Virtual Memory Management”, SOSP 1981; J. Bonwick, “The Slab Allocator”, USENIX Summer 1994.
6. Linux çekirdek belgeleri: `admin-guide/mm/multigen_lru.rst`, `admin-guide/mm/zswap.rst`, `admin-guide/mm/transhuge.rst`, `admin-guide/blockdev/zram.rst`.
7. P. Yosifovich, A. Ionescu, M. Russinovich, D. Solomon, *Windows Internals*, 7. baskı, Kısım 1, Bölüm 5: Memory management.

KISIM III · BÖLÜM 10

Dosya Sistemleri

Bir süreç bittiğinde belleği geri alınır; içindeki veriler de kaybolur. Kalıcı olması gereken her şey diske yazılır. Disk ise yalnızca numaralı bloklar tanır: "şu bloğu oku", "şu bloğa yaz". Dosya sistemi bu ham blokların üzerine adı, boyutu ve sahibi olan dosyaları, iç içe dizinleri ve erişim haklarını kurar. Not önce kullanıcının gördüğü yüzü anlatır: dosya, öznitelikler, sistem çağrıları, açık dosya tabloları, dizinler, bağlar ve izinler. Ardından gerçeklemeye geçer: diskin bölümlenmesi, blokların dosyalara bitişik, bağlı liste, FAT ve i-node yöntemleriyle dağıtılması, boş alanın izlenmesi ve önbellek. Son kesimler çökmeden sonra tutarlılığı ele alır: fsck, günlükleme, yazınca kopyalama ve günlük yapılı dosya sistemleri. VFS ve NFS ile kapanır.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum10` · **Laboratuvar:** `github.com/sercankulcu/operating-systems-labs/lab10`

ÖĞRENME ÇIKTILARI

Dosya soyutlaması
Dosyayı adı, türü, öznitelikleri ve işlemleriyle tanımlamak; açık dosya tablolarını ve dosya tanımlayıcılarını izlemek.

Dizinler
Dizin yapılarını karşılaştırmak, bir yolu adım adım çözmek, katı ve sembolik bağı, bağlamayı açıklamak.

Koruma
rwx izin bitlerini ve özel bitleri okumak; r, w ve x'in dosyada ve dizinde ne anlama geldiğini ayırt etmek.

Tahsis
Bitişik, bağlı liste, FAT ve i-node tahsisini aynı dosya üzerinde uygulamak; en büyük dosya boyutunu ve bir bağıl konumun disk bloğunu hesaplamak.

Boş alan ve performans
Biteşlem, liste, gruplama ve sayma yöntemlerinin boyutunu hesaplamak; tampon ve sayfa önbelleğini, önden okumayı açıklamak.

Güvenilirlik ve gerçek sistemler
fsck, günlükleme, yazınca kopyalama ve LFS'i karşılaştırmak; NTFS, ext4, FAT32/exFAT, VFS, NFS ve SSD'de TRIM'i tanımak.

10.1 Neden dosya sistemi?

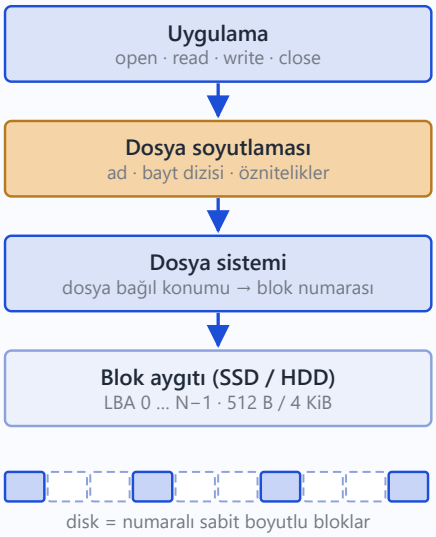
Bir programın verisi çalışırken adres alanında durur. Bu yer üç gereksinim için yetersizdir. Birincisi boyuttur: bir veritabanı ya da video arşivi adres alanına, hatta ana belleğe sığmayacak kadar büyük olabilir. İkincisi **kalıcılıktır**: süreç bitince, hatta makine kapanınca veri yerinde kalmalıdır. Üçüncüsü **eş zamanlı erişimdir**: birçok süreç aynı veriyi okuyup değiştirebilmelidir. Bu üç gereksinimi diske yazılan dosyalar karşılar.

Disk ise işletim sistemine çok sade bir arayüz sunar. Bir sabit disk de bir SSD de kendini numaralı, sabit boyutlu bloklardan oluşan uzun bir dizi olarak gösterir. Blokların numarasına **mantıksal blok adresi** (*logical block address*, LBA) denir. Disk iki işlem bilir: k. bloğu oku ve k. bloğa yaz. Hangi bloğun hangi dosyaya ait olduğu, dosyanın adı, kimin okuyabileceği gibi sorular diskin umurunda değildir. Bu soruları dosya sistemi yanıtlar.

Tanım 10.1

Dosya (*file*), adı olan, kalıcı bir bayt dizisi ve ona eşlik eden özniteliklerdir. **Dosya sistemi** (*file system*), diskteki blokları dosyalara ve dizinlere dönüştüren; onları adlandıran, yerleştiren ve koruyan işletim sistemi bileşenidir.

Dosya, bellekteki adres alanına benzeyen mantıksal bir birimdir. Süreçler dosya oluşturur, okur, yazar ve siler. İşletim sistemi ise adlandırmayı, iç yapıyı, erişimi, korumayı ve diskteki yerleşimi yönetir. Şekil 10.1 bu katmanları gösterir. Uygulama `open`, `read`, `write` ve `close` çağrılarıyla dosyalarla konuşur. Dosya sistemi “dosyanın 5000. baytı” gibi bir isteği “disk bloğu 1729” gibi bir adrese çevirir. Blok aygıtı da yalnızca o bloğu okur.



Şekil 10.1: Uygulamadan blok aygıtına katmanlar. Dosya soyutlaması blokları gizler; disk yalnızca numaralı, sabit boyutlu bloklar tanır.

Konuya iki yönden bakılabilir. **Kullanıcı** neyin göründüğüyle ilgilenir: dosyanın adı, türü, hangi dizinde olduğu, izinleri, hangi işlemlerin yapılabildiği. **Gerçekleyici** ise işin içini düşünür: bloklar dosyalara nasıl dağıtılır, boş alan nasıl izlenir, elektrik kesilirse yapılar nasıl tutarlı kalır? Not önce kullanıcının bakışını, sonra gerçekleyicinin bakışını izler.

10.2 Dosya kavramı

10.2.1 Dosya adları ve uzantılar

Dosyayı oluşturan süreç ona bir ad verir; süreç bittikten sonra başka süreçler dosyayı bu adla bulur. Adlandırma kuralları dosya sisteminden dosya sistemine değişir. Tablo 10.1 yaygın dört sistemi karşılaştırır. Eski MS-DOS'un FAT sistemi “8.3” biçimini kullanırdı: en çok sekiz karakterlik ad, nokta, üç karakterlik uzantı. Uzun adlar Windows 95 ile gelen VFAT'ta ek dizin girdilerinde saklandı. Kısa ad ise geriye uyumluluk için `RAPOR~1.TXT` gibi kırpılmış biçimde durmaya devam etti.

Dosya sistemi	Ad uzunluğu	Büyük/küçük harf	Not
FAT (MS-DOS)	8.3 (ör. `RAPOR~1.TXT`)	duyarsız	VFAT uzun adı ek girdilerde: 255 UTF-16
NTFS	255 UTF-16 birimi	korur, duyarsız	`\ / : * ? " < > |` yasak
ext4, XFS	255 bayt	duyarlı	yalnız `/` ve NUL yasak
APFS	255 UTF-8 karakter	macOS'ta varsayılan duyarsız	Unicode normalizasyonu

Tablo 10.1: Dosya adlandırma kuralları.

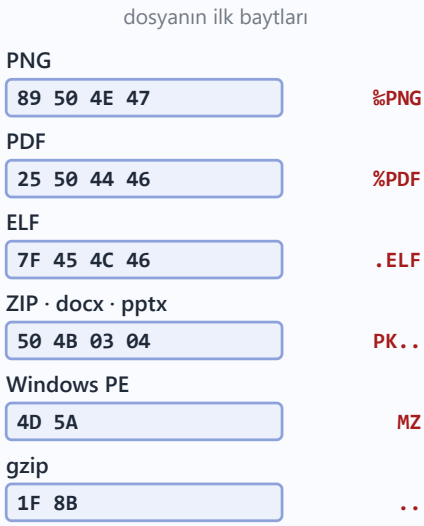
"Büyük/küçük harfe duyarlı" bir sistemde `Rapor.txt` ile `rapor.txt` iki ayrı dosyadır. ext4 böyledir. NTFS ise adı yazıldığı gibi saklar, ama ararken harf farkını önemsemez: iki ad aynı dosyayı bulur. Bugünkü sistemlerin hepsi adları Unicode olarak saklar, bu yüzden Türkçe karakterli adlar sorun çıkarmaz. ext4'ün sınırının karakter değil bayt olduğuna da dikkat etmek gerekir. UTF-8'de "ş" iki bayt tuttuğundan, Türkçe karakter ağırlıklı bir ad 255 karakterden önce sınıra ulaşır.

Adın son noktadan sonraki kısmına **uzantı** (*extension*) denir: `tez.pdf` dosyasının uzantısı `pdf`'tir. UNIX'te uzantı yalnızca bir gelenektir; çekirdek ona hiçbir anlam yüklemez. Programlar ise yükleyebilir. Örneğin `gcc`, `.c` uzantılı kaynak dosyayı derler, `.o` uzantılı nesne dosyasını yalnızca bağlar. Windows'ta uzantı kayıt defterinde (*registry*) bir programla ilişkilendirilir. Dosyaya çift tıklanınca o program açılır. Bu ilişkilendirme bir güvenlik tuzağını da doğurur. Explorer bilinen uzantıları varsayılan olarak gizlediği için `fatura.pdf.exe` adlı bir program ekranda `fatura.pdf` diye görünür.

Dosya sistemleri zamanla değişti. Eski Windows sürümleri FAT16 ve FAT32 kullanırdı; güncel Windows sistem diski için NTFS kullanır. FAT32 ve exFAT bugün USB bellek ve SD kart gibi taşınabilir ortamlarda yaşar, çünkü hemen her aygıt onları okuyabilir.

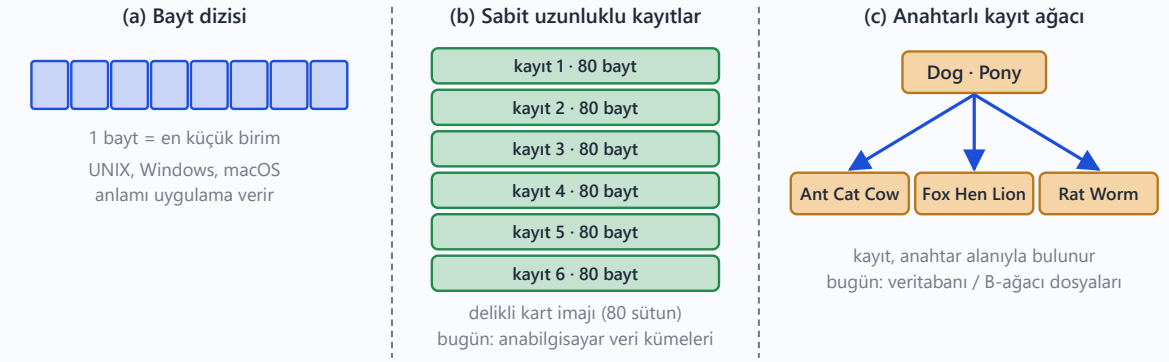
10.2.2 Sihirli sayılar ve dosya yapısı

Uzantı, kullanıcının değiştirebileceği bir addır; dosyanın gerçek türünü kanıtlamaz. Daha güvenilir ipucu dosyanın ilk baytlarıdır. Pek çok biçim belirli bir bayt dizisiyle başlar. Bu diziye **sihirli sayı** (*magic number*) denir. Şekil 10.2 birkaç örnek verir: PNG görüntüsü `89 50 4E 47` ile, PDF `%PDF` ile, Linux'un çalıştırılabilir biçimi ELF `7F 45 4C 46` ile, ZIP arşivi `PK` ile başlar. UNIX'teki `file rapor` komutu türü uzantıya değil bu baytlara bakarak söyler. İlginç bir sonuç: `docx`, `xlsx` ve `pptx` dosyaları da `PK` ile başlar, çünkü bunlar içinde XML dosyaları bulunan ZIP arşivleridir.



Şekil 10.2: Dosya türünü tanıtan sihirli sayılar: PNG, PDF, ELF, ZIP, Windows PE ve gzip başlık baytları.

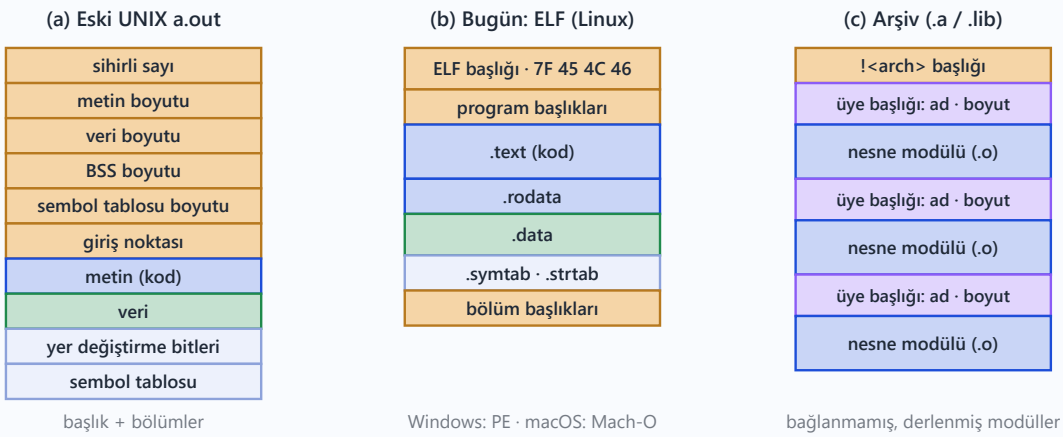
Dosyanın içinin nasıl düzenlendiğine **dosya yapısı** denir. Şekil 10.3 tarihte görülen üç yapıyı gösterir. (a)'daki **bayt dizisi** yapısında işletim sistemi dosyayı yalnızca baytların sırası olarak görür. İçine her şey konabilir; yapıyı uygulama belirler. Bu, en esnek seçenektir ve UNIX, Windows ve macOS'un tümü bunu kullanır. (b)'deki **sabit uzunluklu kayıt** yapısında okuma ve yazma birimi kayıttır; 80 baytlık kayıtlar delikli kartların 80 sütununun mirasıdır. Bu yapı bugün anabilgisayarların veri kümelerinde yaşar. (c)'deki **kayıt ağacında** her kaydın bir anahtar alanı vardır. "Anahtarı Pony olan kaydı getir" gibi bir istek doğrudan desteklenir.



Şekil 10.3: Üç dosya yapısı: (a) bayt dizisi, (b) sabit uzunluklu kayıt dizisi, (c) anahtarla aranan kayıt ağacı.

Bugün anahtarla erişim işletim sisteminde değil, kütüphanelerde ve veritabanlarında yapılır. SQLite, bir B+ ağacını sıradan bir bayt dizisi dosyasının içine yerleştirir. Böylece çekirdek sade kalır, karmaşık yapılar kullanıcı alanında kurulur.

İkili dosyaların da kendi iç düzeni vardır. Şekil 10.4 üç örnek verir. (a) Eski UNIX'in `a.out` biçiminde dosya bir başlıkla açılır. Başlıkta sihirli sayı, kod, veri ve BSS kesimlerinin boyları, sembol tablosunun boyu ve giriş noktası bulunur. Başlığı kod, veri, yer değiştirme bitleri ve sembol tablosu izler. (b) Linux'un bugünkü biçimi ELF de aynı fikri daha esnek kurar: program başlıkları yükleyiciye, bölüm başlıkları bağlayıcıya yöneliktir. Windows'ta karşılığı PE, macOS'ta Mach-O biçimidir. (c) Arşiv dosyası (`.a` ya da `.lib`) derlenmiş ama bağlanmamış nesne modüllerini (`.o`) art arda, her birinin önüne ad ve boy taşıyan bir üye başlığı koyarak saklar. Çekirdek bir dosyayı yalnızca doğru sihirli sayıyla başlıyorsa program olarak çalıştırır.



Şekil 10.4: Çalıştırılabilir ve arşiv biçimleri: (a) eski a.out, (b) ELF, (c) nesne modülü arşivi.

10.2.3 Dosya türleri

UNIX dosyaları birkaç türe ayırır. **Normal dosyalar** kullanıcı verisini tutar. Bunlar ya metindir (yazdırılabilir karakterlerden oluşan satırlar; satır sonu UNIX'te LF, Windows'ta CRLF) ya da ikilidir (yapısını yalnızca onu kullanan program bilir). **Dizinler**, dosya sisteminin yapısını tutan sistem dosyalarıdır. **Özel dosyalar** giriş/çıkış aygıtlarını modeller. Karakter aygıtları (terminal, seri port) baytları sırayla verir. Blok aygıtları (disk) ise bloklar hâlinde rastgele erişime açıktır.

UNIX'in ünlü ilkesi “her şey bir dosyadır” şunu söyler: aygıtlar, borular ve soketler de aynı `open` / `read` / `write` arayüzüyle kullanılır. Örneğin `/dev/null`'a yazılan her şey kaybolur, ondan okumak hemen dosya sonu döndürür. Tablo 10.2 türleri `ls -l` çıktısının ilk karakteriyle birlikte verir.

Harf	Tür	Örnek
`-`	normal dosya (metin ya da ikili)	`notlar.txt`, `/bin/ls`
`d`	dizin	`/home`
`l`	sembolik bağ	`/bin → usr/bin`
`c`	karakter aygıtı (seri, bayt bayt)	`/dev/tty`, `/dev/null`
`b`	blok aygıtı (blok blok, rastgele)	`/dev/nvme0n1`, `/dev/sda`
`p`	adlandırılmış boru (FIFO)	`mkfifo kanal`
`s`	UNIX soketi	`/run/systemd/notify`

Tablo 10.2: UNIX dosya türleri ve `ls -l` çıktısındaki harfleri.

10.2.4 Öznitelikler, stat ve ls -l

Adın ve verinin dışında işletim sistemi her dosya için bir dizi bilgi tutar. Bunlara **öznitelik** (*attribute*) ya da **üst veri** (*metadata*) denir. Tablo 10.3 yaygın öznitelikleri listeler.

Öznitelik	Anlam	Nerede / nasıl
Ad, tür, konum	ad dizin girdisinde; tür ve blok adresleri i-node ya da MFT kaydında	tüm sistemler
Boyut	mantıksal boyut (`st_size`) ve ayrılmış alan (`st_blocks`)	seyrek dosyada ikisi farklıdır
Sahip, grup	UID / GID; Windows'ta SID	`chown` , `icacls`
Koruma	rwx bitleri, ACL	`chmod` , `setfacl`
Zaman damgaları	erişim (atime), içerik değişimi (mtime), i-node değişimi (ctime), oluşturma (btime)	`stat` ; Linux'ta relatime
Bağ sayısı	dosyaya işaret eden ad sayısı	0 olunca bloklar serbest
Read-only, Hidden, System	salt okunur, gizli, sistem dosyası	Windows `attrib` ; UNIX'te gizli = ad nokta ile başlar
Archive	son yedekten sonra değişti	yedekleme aracı sıfırlar
Compressed, Encrypted	sıkıştırılmış / şifrelenmiş	NTFS, Btrfs, APFS; fscrypt, BitLocker
Sparse, Temporary	sıfır blokları diskte yok; geçici (önbellek ipucu)	NTFS bayrakları
Reparse point	özel işleme: sembolik bağ, junction, bulut yer tutucu	NTFS
Genişletilmiş öznitelik	ad = değer çiftleri	xattr (SELinux etiketi), NTFS ADS

Tablo 10.3: Dosya öznitelikleri.

Bazı öznitelikler açıklama ister. Bir **seyrek dosyada** (*sparse file*) hiç yazılmamış bölgeler diskte yer kaplamaz; okunduklarında sıfır döner. Bu yüzden 1 GiB boyutlu bir seyrek dosya diskte yalnızca birkaç kilobayt tutabilir: `st_size` 1 GiB'tır, `st_blocks` ise küçüktür. Üç zaman damgası sık karıştırılır. **mtime** içeriğin, **ctime** ise i-node'un (izin, sahip, bağ sayısı) son değiştiği andır; ctime "oluşturma zamanı" değildir. Oluşturma zamanı (btime) ayrı bir alandır. Her okumada atime'ı güncellemek her okumayı bir yazmaya çevirir. Linux bu yüzden varsayılan olarak *relatime* kullanır: atime yalnızca mtime'dan eskiyse ya da bir günden eskiyse güncellenir.

Windows'un bazı öznitelikleri de yanlış anlaşılır. Temporary özniteliği dosyayı kendiliğinden silmez; yalnızca önbellek yöneticisine "bunu diske yazmakta acele etme" der. Symlink, Device ve Junction öznitelik değil dosya türüdür. Creator ve Max size gibi alanlar eski sistemlere aittir.

Bir programın öznitelikleri okuması `stat` sistem çağrısıyla olur. Çağrı şu yapıyı doldurur:

```
struct stat {
  dev_t     st_dev;      /* aygıt (dosya sistemi)  */
  ino_t     st_ino;      /* i-node numarası        */
  mode_t    st_mode;     /* tür + izin bitleri      */
  nlink_t   st_nlink;    /* katı bağ sayısı         */
  uid_t     st_uid;      /* sahip kullanıcı         */
  gid_t     st_gid;      /* sahip grup              */
  dev_t     st_rdev;     /* aygıt dosyası ise no.   */
  off_t     st_size;     /* boyut (bayt)            */
  blksize_t st_blksize;  /* tercih edilen G/Ç boyu  */
  blkcnt_t  st_blocks;   /* ayrılmış 512 B birim    */
  struct timespec st_atim; /* son erişim   */
  struct timespec st_mtim; /* son içerik   */
  struct timespec st_ctim; /* son i-node   */
};
```

Alanların çoğu tablodaki özniteliklerin karşılığıdır. `st_ino` dosyanın i-node numarasıdır; aynı dosya sisteminde bir dosyayı tek başına tanımlar. `st_mode` tür ve izin bitlerini birlikte taşır. `st_nlink` katı bağ sayısıdır. `st_blocks`, dosyaya ayrılmış alanı 512 baytlık birimlerle verir; blok boyu 4 KiB olsa bile birim 512 B'tır. POSIX.1-2008'den beri zamanlar nanosaniye çözünürlüklü `struct timespec` alanlarıdır. Eski `st_atime` adı geriye uyumluluk için bir makro olarak kalır. Linux'ta oluşturma zamanı ayrı bir çağrıyla, `statx()` ile okunur.



Şekil 10.5: `ls -l` çıktısının alanları: tür, izinler, bağ sayısı, sahip, grup, boyut, değişiklik zamanı ve ad.

`ls -l` bu yapıyı insanın okuyacağı biçime çevirir (Şekil 10.5). İlk karakter tür, sonraki dokuz karakter sahip, grup ve diğerlerinin izinleridir. Ardından bağ sayısı, sahip ve grup adı, bayt cinsinden boyut, mtime ve ad gelir. Ad, `stat` yapısında yoktur: ad i-node'da değil, dizin girdisinde durur. Aynı i-node'a birden çok ad verilebilmesi de buradan gelir.

10.2.5 Dosya işlemleri ve sistem çağrıları

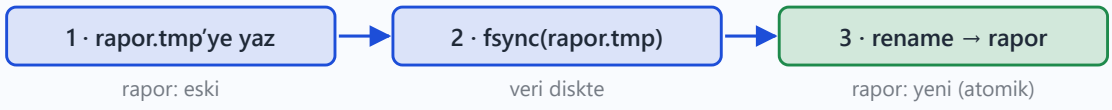
Dosya bir soyut veri tipidir: onu tanımlayan, üzerinde yapılabilen işlemlerdir. Tablo 10.4 temel işlemleri ve POSIX karşılıklarını verir.

İşlem	POSIX çağrısı	Ne yapar?
Oluşturma	`open(O_CREAT)`, `creat`	boş dosya ve dizin girdisi oluşturur, öznitelikleri atar
Silme	`unlink`	adı siler; bağ sayısı 0 olup son tanımlayıcı da kapanınca alan boşalır
Açma	`open`	yolu çözer, i-node'u belleğe alır, tanımlayıcı (fd) döndürür
Kapatma	`close`	fd'yi ve gerekirse tablo girdilerini serbest bırakır
Okuma / yazma	`read`, `write`	dosya konumundan itibaren okur / yazar, konumu ilerletir
Sona ekleme	`open(O_APPEND)`	her yazma atomik olarak dosya sonuna gider
Konumlama	`lseek`	konumu değiştirir (rastgele erişim); G/Ç yapmaz
Öznitelik okuma / yazma	`stat`, `chmod`, `chown`, `utimensat`	öznitelikleri okur / değiştirir
Yeniden adlandırma	`rename`	adı değiştirir; aynı dosya sisteminde atomiktir
Kısaltma, eşleme	`ftruncate`, `mmap`	boyutu kısaltır ya da uzatır; dosyayı adres alanına eşler

Tablo 10.4: Dosya işlemleri ve POSIX sistem çağrıları.

Tablodaki birkaç ayrıntı önemlidir. `unlink` dosyayı değil bir adı siler. Dosyanın blokları ancak hiçbir ad kalmadığında ve onu açık tutan son tanımlayıcı da kapandığında serbest kalır. Bu yüzden açık bir dosya silinse bile onu kullanan süreç okumaya devam edebilir. `lseek` yalnızca çekirdekteki konum sayısını değiştirir; diske dokunmaz. `O_APPEND` ile açılmış bir dosyada "sona git ve yaz" işlemi bölünmez. İki süreç aynı günlük dosyasına aynı anda yazsa bile kayıtları birbirinin üstüne binmez.

`rename`'in atomik olması, uygulamalarda yaygın bir kalıbın temelidir (Şekil 10.6). Bir belgeyi güvenle kaydetmek için yeni içerik önce geçici bir dosyaya yazılır. Ardından `fsync` ile verinin diske ulaşması beklenir. En sonda `rename(gecici, asil)` çağrılır. Çökme hangi anda olursa olsun, diskte ya eski dosyanın tamamı ya da yeni dosyanın tamamı görünür; yarım yazılmış bir dosya görünmez.



Şekil 10.6: Güvenli kaydetme: geçici dosyaya yazma, fsync ve rename. Her anda ya eski ya yeni dosya görünür.

Sık yapılan hata

Belgeyi yerinde açıp `O_TRUNC` ile kesmek ve yeni içeriği yazmak. Kesme ile yazmanın arasında bir çökme olursa ortada ne eski ne yeni içerik kalır. `fsync` adımını atlamak da tehlikelidir:

`rename` diske verinin kendisinden önce ulaşabilir ve sonuç sıfır baytlık bir "yeni" dosya olur.

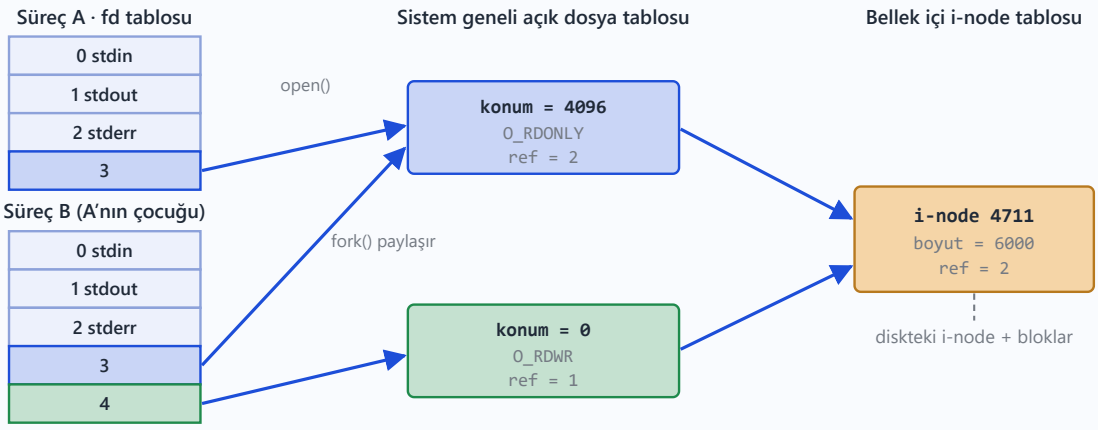
10.2.6 Açık dosya tabloları

Her `read` çağrısında dosyanın yolunu baştan çözmek çok pahalı olurdu. Bu yüzden dosya önce açılır. `open` yolu bir kez çözer, dosyanın i-node'unu belleğe alır ve sürece küçük bir tam sayı döndürür. Sonraki çağrılar dosyayı bu sayıyla anar.

Tanım 10.2

Dosya tanımlayıcısı (*file descriptor*, fd), çekirdeğin süreç başına tuttuğu tanımlayıcı tablosunda bir indistir. 0, 1 ve 2 sırasıyla standart girdi, standart çıktı ve standart hatadır. `open` her zaman en küçük boş numarayı verir.

Çekirdek açık dosyaları üç düzeyde izler (Şekil 10.7). En üstte her sürecin kendi **tanımlayıcı tablosu** vardır; girdileri ortadaki tabloyu gösterir. Ortada sistem genelindeki **açık dosya tablosu** (*open-file table*) bulunur. Her girdisi bir açılışı temsil eder ve iki önemli bilgi taşır: dosyadaki geçerli konum (bağıl konum) ve erişim modu (salt okuma, okuma-yazma). En altta **bellek içi i-node tablosu** durur. Burada her dosya, kaç kez açılmış olursa olsun, tek bir kez bulunur.



Şekil 10.7: Süreç başına fd tabloları, sistem geneli açık dosya tablosu ve bellek içi i-node tablosu. A'nın fd 3'ü ile çocuğu B'nin fd 3'ü aynı girdiyi paylaşır; B'nin fd 4'ü aynı dosyanın ikinci bir açılışıdır.

Bu ayrımın sonucu şöyle özetlenebilir. `fork` ve `dup` tanımlayıcı tablosundaki girdiyi kopyalar; kopyalar aynı açık dosya girdisini gösterir. Dolayısıyla konumu paylaşırlar: biri 10 bayt okursa öbürünün konumu da 10 bayt ilerler. Aynı dosyayı ikinci kez `open` ile açmak ise yeni bir girdi kurar. Bu girdinin konumu ayrıdır ve 0'dan başlar. Her iki durumda da i-node tektir. Açık dosya girdisindeki referans sayısı onu kaç tanımlayıcının gösterdiğini, i-node'daki referans sayısı da kaç girdinin gösterdiğini tutar. Sayı 0'a inince ilgili yapı serbest bırakılır. Windows'ta aynı rolü her sürecin **tanıtıcı** (*handle*) tablosu oynar.

Örnek 10.1

`harf.txt` dosyasının içeriği `ABCDEFGHIJ` (10 bayt). Bir süreç dosyayı açar ve 4 bayt okur. Sonra `fork` çağırır; çocuk aynı tanımlayıcıdan 3 bayt okuyup sonlanır. Ebeveyn çocuğu bekler, dosyayı ikinci kez açar ve yeni tanımlayıcıdan 2 bayt okur. Son olarak ilk tanımlayıcıdan 2 bayt okur. Hangi baytlar okunur?

1. **İlk okuma.** `open` fd 3'ü verir; girdi E1 kurulur, konum 0. Okunan `ABCD`, E1'in konumu 4.
2. **fork.** Çocuğun fd 3'ü de E1'i gösterir; E1'in referans sayısı 2.
3. **Çocuk.** Ortak konum 4'ten 3 bayt okur: `EFG`. E1'in konumu 7. Çocuk sonlanınca referans sayısı 1'e iner, konum 7 kalır.
4. **İkinci open.** Yeni girdi E2 kurulur, fd 4, konum 0. Okunan `AB`; E1 etkilenmez.
5. **Ebeveynin fd 3'ü.** E1'in konumu 7'dir: okunan `HI`, konum 9.

Ebeveynin son okuması `EF` değil `HI` 'dir, çünkü çocuk ortak konumu ilerletmiştir.

10.2.7 Kod izleme: open, read, close

Aşağıdaki program 6000 baytlık `notlar.txt` dosyasını 4096 baytlık bir tamponla üç kez okur ve kapatır.

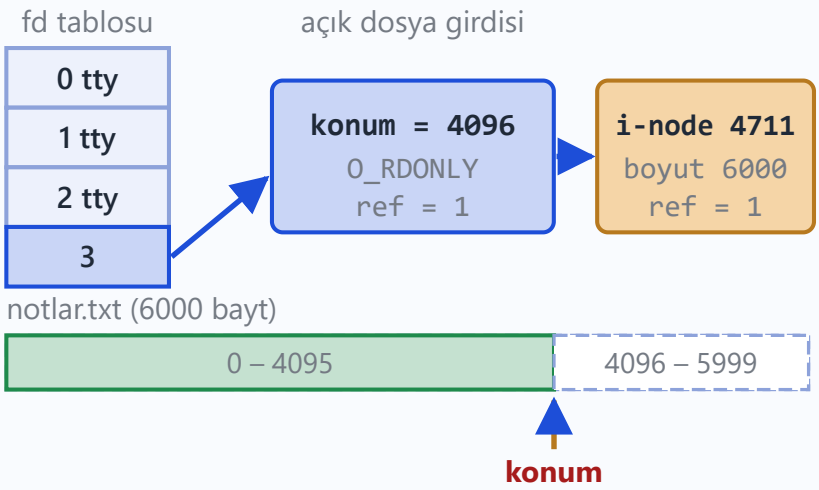
```
#include <fcntl.h>
#include <unistd.h>

int main(void) {
  char buf[4096];
  int fd = open("notlar.txt", O_RDONLY);
  ssize_t n = read(fd, buf, sizeof buf);
  n = read(fd, buf, sizeof buf);
  n = read(fd, buf, sizeof buf);
  close(fd);
  return 0;
}
```

`open` yolu çözer ve dosyanın i-node'unu (numarası 4711 olsun) bellek içi tabloya alır. Konumu 0 olan bir açık dosya girdisi kurar ve en küçük boş numarayı, yani 3'ü döndürür; 0, 1 ve 2 kabuktan miras kalan terminal tanımlayıcılarıdır. İlk `read` dosyanın baştaki bloğunu önbelleğe alır (gerekirse diskten okur) ve 4096 baytı `buf` 'a kopyalar. Konum 4096 olur. İkinci `read` de 4096 bayt ister, ama dosyada yalnızca 6000 − 4096 = 1904 bayt kalmıştır. Çağrı 1904 döndürür; konum 6000 olur. Bu bir hata değildir: istenenden az bayt döndüren bir okuma, yani **kısa okuma**, olağan bir durumdur. Üçüncü `read` 0 döndürür. Sıfır dönüşü dosya sonu (EOF) demektir; hata olsaydı −1 döner ve `errno` ayarlanırdı. `close` fd 3'ü boşaltır. Girdinin referans sayısı 0'a iner ve girdi silinir. i-node'un sayısı da 0 olur; artık bellekten atılabilir. Tablo 10.5 bu adımları özetler.

Adım	Dönüş	Konum	Girdi ref.	i-node ref.
`open`	3	0	1	1
1. `read`	4096	4096	1	1
2. `read`	1904	6000	1	1
3. `read`	0 (EOF)	6000	1	1
`close`	0	—	0 (silinir)	0

Tablo 10.5: `open`, `read`, `close` izlemesi (6000 baytlık dosya, 4096 baytlık tampon).



Şekil 10.8: İlk `read`'den sonra tablolar: fd 3, konumu 4096 olan açık dosya girdisini; girdi de i-node 4711'i gösterir.

10.2.8 Kod izleme: dosya kopyalama

Kopyalama, okuma ve yazmanın birlikte kullanıldığı en sade programdır:

```
#define BUF 4096

int kopyala(const char *src, const char *dst) {
  char buf[BUF];
  int in  = open(src, O_RDONLY);
  int out = open(dst, O_WRONLY | O_CREAT
                      | O_TRUNC, 0644);
  ssize_t n;
  while ((n = read(in, buf, BUF)) > 0)
    write(out, buf, n);
  close(in);
  close(out);
  return 0;
}
```

Kaynak salt okuma için açılır ve 3 numarayı alır. Hedef üç bayrakla açılır ve 4 numarayı alır. `O_WRONLY` yalnız yazma ister. `O_CREAT` dosya yoksa onu oluşturur. `O_TRUNC` ise dosya varsa içeriğini siler. `O_CREAT` kullanıldığında üçüncü argüman yeni dosyanın izinleridir: `0644`, yani sahip için okuma-yazma, diğerleri için yalnız okuma. Döngü her turda en çok `BUF` bayt okur ve okuduğu kadarını yazar. Koşuldaki atama dikkat ister: `read`'in dönüşü önce `n`'ye yazılır, sonra 0'dan büyük olup olmadığına bakılır.

`abc` 10.000 bayt olsun. İzleme Tablo 10.6'da verilir. İlk iki tur 4096'şar bayt kopyalar; toplam 8192. Üçüncü `read` kalan 10.000 − 8192 = 1808 baytı döndürür ve `write` tam 1808 bayt yazar. Dördüncü `read` 0 döndürür ve döngü biter. Toplam 4 `read` ve 3 `write` çağrısı yapılır.

Tur	`read` dönüşü (n)	`write`	Kopyalanan toplam
1	4096	4096 bayt	4096
2	4096	4096 bayt	8192
3	1808	1808 bayt	10.000
4	0 (EOF)	—	10.000; döngü biter

Tablo 10.6: `kopyala("abc", "xyz")` izlemesi (abc 10.000 bayt, BUF = 4096).

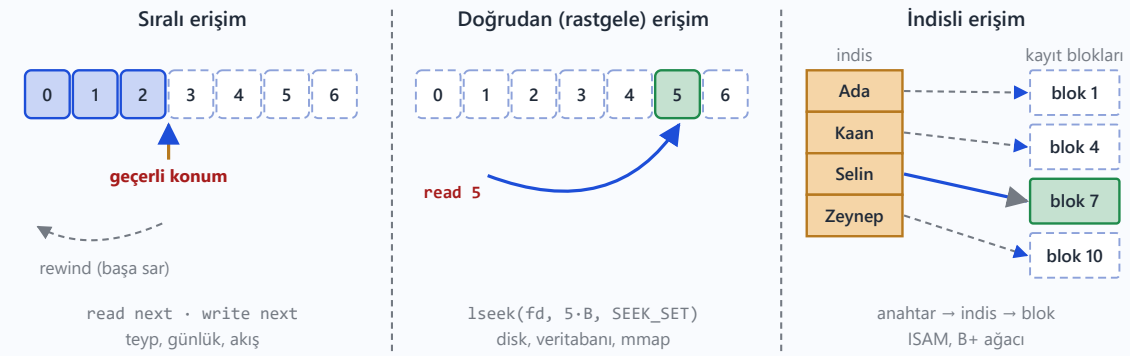
Sık yapılan hata

Döngüde `write(out, buf, BUF)` yazmak. Son turda tamponda yalnız 1808 geçerli bayt vardır; geri kalan 2288 bayt önceki turdan kalan çöptür. Hedef dosya 12.288 bayt olur ve sonu bozuk çıkar. Yazılacak miktar her zaman `read`'in döndürdüğü `n`'dir.

Yukarıdaki kod kısa tutulmak için hataları denetlemez. Gerçek bir programda her çağrının −1 dönüşüne bakılmalıdır. `write` da istenenden az bayt yazabilir (ör. disk dolarken ya da bir boruya yazarken); bu yüzden kalan kısım bir iç döngüyle tamamlanmalıdır. Tampon boyu da önemlidir. 6 MiB'lık (6.291.456 bayt) bir dosya bayt bayt kopyalanırsa 6.291.457 `read` ve 6.291.456 `write` gerekir. Çekirdeğe her geçiş 0,3 µs sürerse yalnızca bu ek yük yaklaşık 3,77 saniye tutar. 4096 baytlık tamponla çağrı sayısı 1537 + 1536 = 3073'e, ek yük de yaklaşık 0,92 ms'ye iner. 4096 bayt sayfa ve blok boyuna da denk gelir.

10.2.9 Erişim yöntemleri

Bir dosyanın içine hangi sırayla erişildiğine **erişim yöntemi** denir (Şekil 10.9). **Sıralı erişimde** dosya baştan sona okunur ya da yazılır. Geçerli konum her işlemde ilerler; geri gitmek için başa sarmak (*rewind*) gerekir. Teyp ve kaset böyle çalışırdı. Bugün de günlük dosyaları, video akışları ve derleyicinin kaynak dosyayı okuması en yaygın kalıptır.



Şekil 10.9: Sıralı, doğrudan ve indisli erişim.

Doğrudan (rastgele) erişimde dosya numaralı blokların ya da sabit uzunluklu kayıtların dizisidir. İstenen kayıttan başlanabilir: B baytlık kayıtlarda 5. kayıt için `lseek(fd, 5·B, SEEK_SET)` yeterlidir. `pread` konumlamayı ve okumayı tek çağrıda yapar, ortak konumu da değiştirmez. Diskler ve SSD'ler doğrudan erişimi destekler; veritabanları bunu kullanır. **İndisli erişim** doğrudan erişimin üzerine kurulur. Küçük bir indis, anahtarı kaydın bulunduğu bloğa eşler. İndis büyürse indisin de indisi tutulur. IBM'in ISAM yöntemi ve bugünkü B+ ağaçları bu fikrin örnekleridir. Son olarak **bellek eşlemeli erişimde** (`mmap`) dosyanın sayfaları adres alanına eşlenir; okuma ve yazma sıradan bellek erişimine dönüşür.

10.3 Dizinler

10.3.1 Dizin işlemleri

> **Tanım 10.3**
>
> **Dizin** (*directory*), dosya adlarını dosyalara eşleyen özel bir dosyadır. Her eşlemeye **dizin girdisi** (*directory entry*) denir; UNIX'te bir girdi bir adı bir i-node numarasıyla eşler.

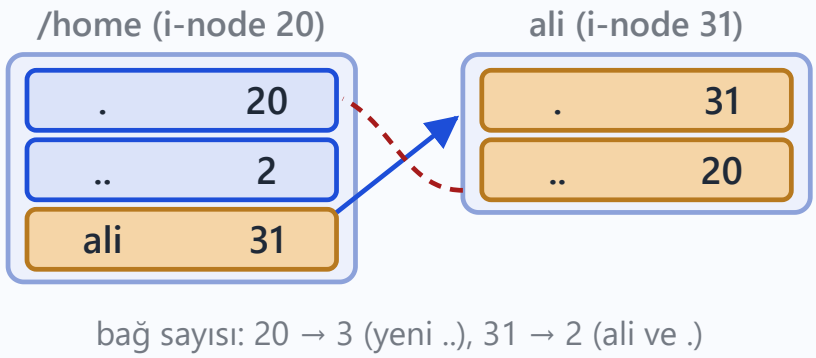
Dizin de bir dosyadır, ama kullanıcı onu `write` ile değiştiremez. Bozuk bir dizin bütün alt ağacı ulaşılmaz kılabileceği için çekirdek dizinlerin içeriğini yalnızca belirli çağrılarla değiştirir. Tablo 10.7 bu çağrıları listeler.

Çağrı	İşlev
`mkdir`	boş dizin oluşturur (içinde yalnız `.` ve `..`)
`rmdir`	dizini siler; dizin boş olmalıdır
`opendir` / `closedir`	dizin akışını açar / kapatır
`readdir`	açık dizindeki bir sonraki girdiyi (ad, i-node no., tür) döndürür
`rename`	dosya ya da dizini yeniden adlandırır veya taşır (aynı dosya sisteminde atomik)

Çağrı	İşlev
`link`	var olan bir dosyaya yeni bir ad (katı bağ) ekler; i-node'un bağ sayısı artar
`symlink`	hedef yolunu içeren sembolik bağ oluşturur
`unlink`	bir adı kaldırır; bağ sayısı 0 olursa dosya silinir

Tablo 10.7: Dizin işlemleri.

Yeni bir dizinin iki girdisi baştan vardır: `.` dizinin kendisini, `..` üst dizini gösterir. Bu iki girdi bağ sayılarını da etkiler (Şekil 10.10). `/home`'un i-node'u 20 olsun ve içinde başka alt dizin bulunmasın. Bağ sayısı 2'dir: kökteki `home` girdisi ve kendi `.` girdisi. `mkdir /home/ali` yeni dizine i-node 31'i verir. `/home`'a `ali → 31` girdisi eklenir; yeni dizinde `. → 31` ve `.. → 20` bulunur. Sonuçta i-node 31'in bağ sayısı 2 olur (`ali` ve `.`). i-node 20'ninki 3'e çıkar, çünkü yeni dizinin `..` girdisi de onu gösterir. Genel kural şudur: bir dizinin bağ sayısı 2 artı alt dizin sayısıdır.

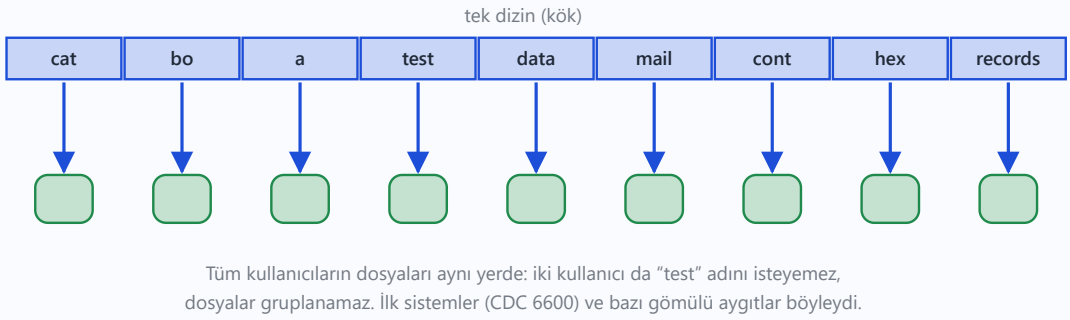


Şekil 10.10: `mkdir /home/ali` : üst dizine `ali` girdisi eklenir; yeni dizinde `.` kendisini, `..` üst dizini gösterir.

`link`'in adı yanıltabilir. Bu çağrı bir dizini başka bir dizine bağlamaz; var olan bir dosyaya ikinci bir ad verir. Dizinlere katı bağ vermek sıradan kullanıcılara yasaktır, çünkü dizin ağacında döngü oluşturabilir.

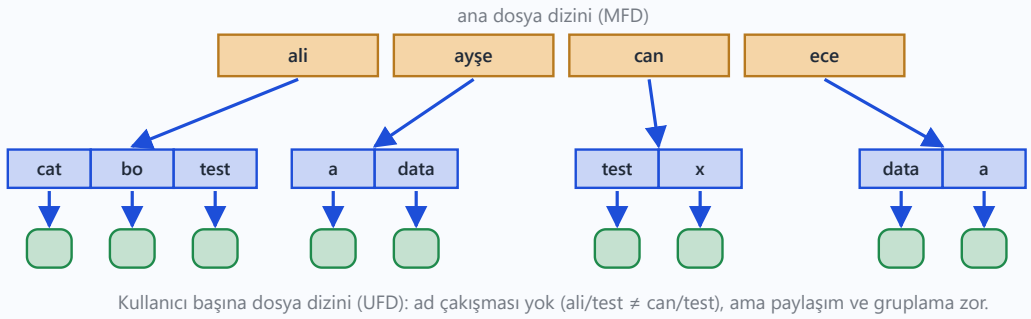
10.3.2 Dizin yapıları

Dizinlerin birbirine göre nasıl düzenleneceği zaman içinde değişti. En basit yapı **tek düzeyli dizindir** (Şekil 10.11): bütün dosyalar tek bir kök dizinde durur. CDC 6600 gibi ilk süper bilgisayarlar ve bazı gömülü aygıtlar böyleydi. Yapı basittir ama iki sorun doğurur. İki kullanıcı aynı adı, örneğin `test`'i, kullanamaz. Dosyalar da konularına göre gruplanamaz.



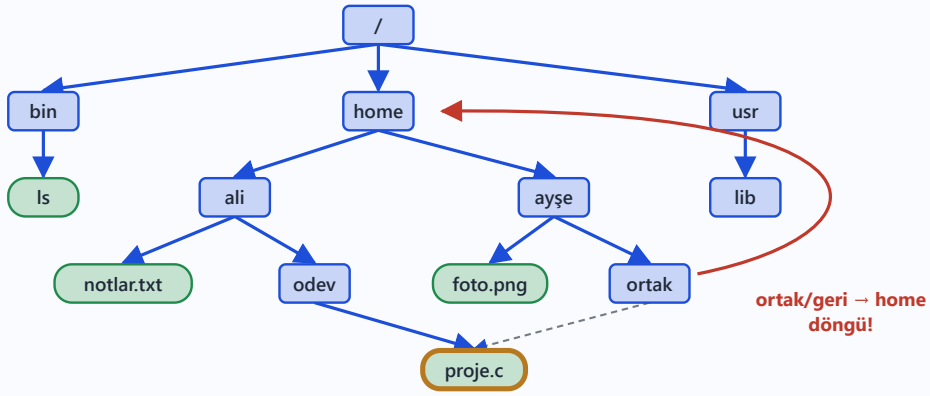
Şekil 10.11: Tek düzeyli dizin: tüm kullanıcıların dosyaları aynı yerde.

İki düzeyli dizinde bir ana dosya dizini (*master file directory*, MFD) her kullanıcıya ayrı bir kullanıcı dosya dizini (UFD) verir (Şekil 10.12). Ad çakışması ortadan kalkar: `ali/test` ile `can/test` farklı dosyalardır. Bir dosyanın tam adı artık "kullanıcı/dosya" biçimindeki bir yoldur. Sistem programları için ayrı bir "kullanıcı" dizini gerekir. Bir komut yazıldığında hangi dizinlere bakılacağını söyleyen **arama yolu** (*search path*) fikri buradan doğar. Paylaşım ve gruplama ise yine zordur.



Şekil 10.12: İki düzeyli dizin: ana dizin altında kullanıcı başına bir dizin.

Bugünkü genel amaçlı sistemlerin hepsi **ağaç** (hiyerarşik) yapı kullanır. Kullanıcı istediği kadar alt dizin açabilir ve her dosyanın kökten başlayan tek bir yolu vardır. Ağaca paylaşım eklenince yapı **döngüsüz çizgeye** (*directed acyclic graph*, DAG) dönüşür (Şekil 10.13). Şekilde `proje.c` hem `ali/odev` hem `ayşe/ortak` altında görünür: aynı dosyanın iki adı vardır. Paylaşım iki yeni soru getirir. Ağaçta gezinen bir yedekleme ya da sayma programı aynı dosyayı iki kez sayabilir. Paylaşılan dosya silinirken de hangi adın silindiği, verinin ne zaman serbest kalacağı belirsizleşir. Bağ sayısı ikinci soruyu yanıtlar: veri, son ad da silinince serbest kalır.



Şekil 10.13: Paylaşılan `proje.c` yapıyı döngüsüz çizgeye çevirir. `ortak/geri` adlı bir bağ `home`'a geri dönerse genel çizgede döngü oluşur.

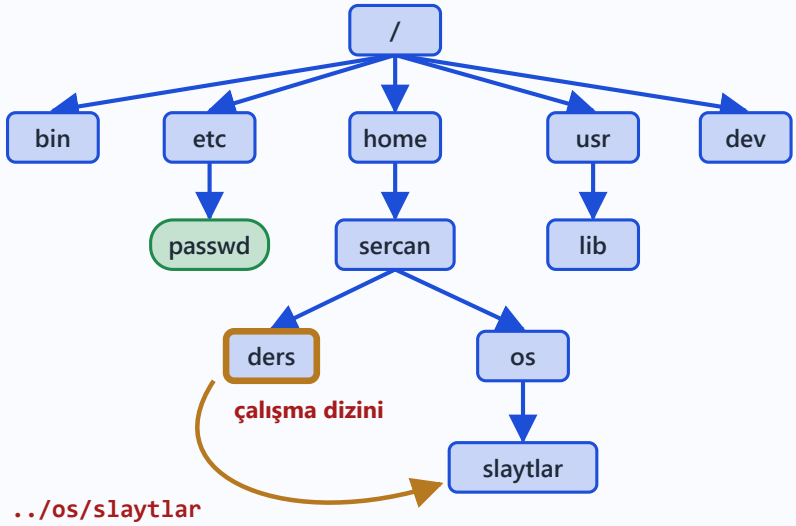
Bir dizine bağ verilmesine izin verilirse **genel çizge** ortaya çıkar. Şekildeki `ortak/geri → home` bağı bir döngü kurar. Ağaçta gezinen bir arama sonsuza gidebilir. Hiçbir yerden ulaşılamayan bir döngünün bağ sayıları da sıfıra inmez; bu dizinleri bulmak için pahalı bir çöp toplama gerekir. UNIX bu yüzden dizinlere katı bağı yasaklar.

10.3.3 Yol adları

Tanım 10.4

Mutlak yol kök dizinden başlar: `/home/sercan/os/slaytlar`. **Göreli yol** sürecin çalışma dizininden başlar. Çalışma dizini `pwd` ile görülür, `cd` ile değiştirilir.

Her dizinde bulunan iki özel ad göreli yolları güçlü kılar: `.` bulunulan dizin, `..` üst dizindir. Kökte `..` yine kökü gösterir. Şekil 10.14'te çalışma dizini `/home/sercan/ders`'tir. `../os/slaytlar` önce üst dizine, yani `/home/sercan`'a çıkar, oradan `os`'a ve `slaytlar`'a iner. Ayırıcı UNIX'te `/`, Windows'ta `\`'dir. Windows API'si `/`'yi de kabul eder. Windows'ta mutlak yol bir sürücü harfiyle (`C:\Users\sercan\os`), ağ paylaşımı ise iki ters bölüyle (`\\sunucu\pay`) başlar.



Şekil 10.14: UNIX dizin ağacı; çalışma dizini `/home/sercan/ders`, göreli yol `../os/slaytlar`.

Dizin adlarının da tarihi vardır. İlk UNIX sistemlerinde kullanıcı dizinleri `/usr` altındaydı; Tanenbaum'un örneklerindeki `/usr/ast` buradan gelir. Bugün kullanıcı dizinleri `/home` (macOS'ta `/Users`) altındadır. `/usr` ise salt okunur sistem yazılımına ayrılmıştır.

Örnek 10.2

Çalışma dizini `/home/elif/odev/os`. `../../notlar/./bolum10.txt` hangi dosyayı gösterir?

1. İlk `..`: `/home/elif/odev`.
2. İkinci `..`: `/home/elif`.
3. `notlar`: `/home/elif/notlar`.
4. `.` bulunulan dizindir; yol değişmez.
5. `bolum10.txt`: sonuç `/home/elif/notlar/bolum10.txt`.

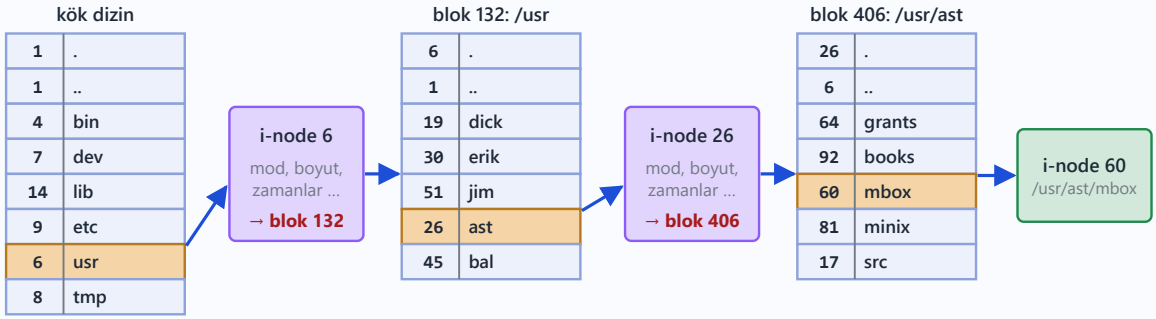
10.3.4 Yol çözümleme

Bir yolun i-node numarasına çevrilmesine **yol çözümleme** denir. Çekirdek yolu bileşen bileşen yürür. Her dizin için önce dizinin i-node'unu okur; i-node dizin içeriğinin hangi blokta olduğunu söyler. Sonra o bloğu okur ve bir sonraki bileşenin adını arar. Bulduğu girdi bir sonraki i-node numarasını verir.

Tanenbaum'un klasik örneği `/usr/ast/mbox` yolunu çözer (Şekil 10.15). Yol mutlak olduğu için arama kökten başlar. Kök dizinin i-node numarası bilinir: bu örnekte 1, V7 UNIX'te ve ext4'te 2'dir. Kök i-node'u bağlama sırasında belleğe alınmıştır.

1. **Kök dizin bloğu okunur.** `usr` aranır ve i-node 6 bulunur.

2. **i-node 6 okunur.** `/usr` bir dizindir; içeriği blok 132'dedir. Bu adımda izin denetimi de yapılır: dizinin içinden geçmek için x hakkı gerekir.
3. **Blok 132 okunur.** `ast` aranır ve i-node 26 bulunur.
4. **i-node 26 okunur.** `/usr/ast`'in içeriği blok 406'dadır.
5. **Blok 406 okunur.** `mbox` aranır ve i-node 60 bulunur.



Şekil 10.15: `/usr/ast/mbox` yolunun çözülmesi: kök dizin → i-node 6 → blok 132 → i-node 26 → blok 406 → i-node 60.

Önbellek yoksa bu çözümleme 5 disk okuması tutar. Dosyanın kendi i-node'unu (60) ve ilk veri bloğunu okumak için iki okuma daha gerekir. Genel kural kolay çıkarılır. Kök i-node'u bellekte ve her dizin tek blokta olsun. k bileşenli bir yolun son i-node'una ulaşmak 2k okuma, ilk veri bloğunu da okumak 2k + 1 okuma ister. Göreli bir yolda arama kök yerine çalışma dizininin i-node'undan başlar; çalışma dizini zaten bellekte olduğu için daha ucuzdur. Linux, yeni çözülmüş ad → i-node eşlemelerini **dentry önbelleğinde** tutar. Aynı yol ikinci kez açıldığında disk okumalarının çoğu ortadan kalkar.

Örnek 10.3

`/var/log/sistem/gunluk.txt` dosyasının ilk veri bloğu okunacak. Önbellekte yalnız kök i-node'u var; her dizin tek bloğa sığıyor. En az kaç disk okuması gerekir?

1. Yolda dört bileşen var: `var`, `log`, `sistem`, `gunluk.txt`.
2. Her bileşen için iki okuma yapılır: içinde arandığı dizinin bloğu ve bulunan i-node. Kök dizinin bloğu ilk okumadır; dört bileşen $4 \times 2 = 8$ okuma eder.
3. Son olarak dosyanın ilk veri bloğu okunur: toplam $2 \cdot 4 + 1 = 9$ okuma.

10.3.5 Katı ve sembolik bağlar

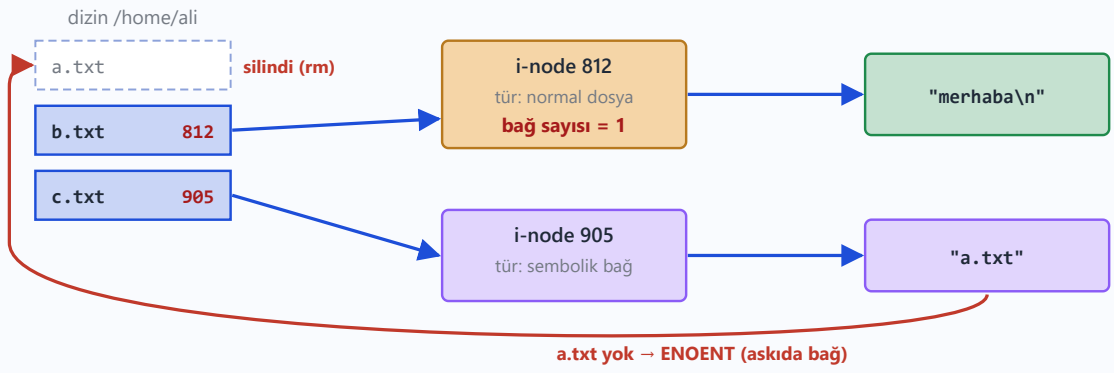
Tanım 10.5

Katı bağ (*hard link*), var olan bir i-node'u gösteren ikinci bir dizin girdisidir. **Sembolik bağ** (*symbolic link*), içeriği başka bir dosyanın yolu olan, kendi i-node'u olan küçük bir özel dosyadır.

İki bağ türünün farkı en iyi bir izlemeyle görülür (Tablo 10.8). `echo merhaba > a.txt` yeni bir dosya oluşturur. Dizinde `a.txt → 812` girdisi vardır ve i-node 812'nin bağ sayısı 1'dir. `ln a.txt b.txt` katı bağ kurar. Yeni girdi `b.txt → 812` aynı i-node'u gösterir ve bağ sayısı 2 olur. İki ad tamamen eşittir; hangisinin "asıl" olduğu sorusunun yanıtı yoktur. `ln -s a.txt c.txt` ise sembolik bağ kurar: i-node 905 olan yeni bir dosya oluşur ve içeriği `"a.txt"` dizgisidir. i-node 812'nin bağ sayısı değişmez.

Komut	Dizin girdileri	i-node 812 bağ sayısı	Sonuç
`echo merhaba > a.txt`	a.txt → 812	1	yeni dosya
`ln a.txt b.txt`	a.txt → 812, b.txt → 812	2	iki eşit ad
`ln -s a.txt c.txt`	+ c.txt → 905 (içerik "a.txt")	2	sembolik bağın kendi i-node'u var
`rm a.txt`	b.txt → 812, c.txt → 905	1	`cat b.txt` : merhaba; `cat c.txt` : ENOENT

Tablo 10.8: Katı ve sembolik bağ izlemesi (Python ile benzetildi).



Şekil 10.16: `rm a.txt` sonrası: `b.txt` aynı i-node'u göstermeye devam eder; `c.txt` artık var olmayan bir adı gösterir ve askıda kalır.

Son satır iki türün özünü gösterir (Şekil 10.16). `rm a.txt` yalnızca bir adı siler ve bağ sayısını 1'e indirir. Veri yerinde durur; `b.txt` çalışır. Sembolik bağ ise i-node'u değil adı saklar. `c.txt` açılırken çekirdek `"a.txt"` yolunu yeniden çözer, adı bulamaz ve `ENOENT` (böyle bir dosya yok) hatası döndü-

rür. Bu duruma **askıda bağ** (*dangling link*) denir. Aynı adla yeni bir dosya oluşturulursa sembolik bağ sessizce o yeni dosyayı açmaya başlar; katı bağ ise eski veriyi göstermeye devam eder.

Tanenbaum katı bağların ince bir yan etkisine dikkat çeker. C kullanıcısının dosyasına B bir katı bağ verir; sonra C dosyayı siler. B'nin adı ve veri kalır, ama i-node'un sahibi hâlâ C'dir. Dosyanın kapladığı alan da C'nin kotasından düşülmeye devam eder.

İki türü karşılaştıralım. Katı bağın ek maliyeti yoktur; ama yalnızca aynı dosya sistemi içinde kurulabilir (i-node numarası başka bir dosya sisteminde anlamsızdır) ve dizinlere verilemez. Sembolik bağ dizinleri, başka dosya sistemlerindeki dosyaları, hatta ağ yollarını gösterebilir. Bedeli her erişimde ek bir yol çözümüdür. Sembolik bağlar döngü de kurabilir. Çekirdek bu yüzden bir yol çözülürken izlenen bağ sayısını sınırlar; sınır aşılırsa `ELOOP` hatası döner. Windows'ta katı bağ `mklink /H`, sembolik bağ `mklink` ile kurulur. Dizinler için bir de *junction* türü vardır.

Örnek 10.4

Boş bir dizinde (ext4) `mkdir d`, `echo x > d/a`, `ln d/a d/b`, `ln -s a d/c` ve `mkdir d/e` çalıştırılıyor. `d`, `d/a`, `d/b`, `d/c` ve `d/e` 'nin bağ sayıları nedir?

1. **d: 3.** Üst dizindeki `d` girdisi, kendi `.` girdisi ve alt dizin `e` 'nin `..` girdisi.
2. **d/a ve d/b: 2.** Aynı i-node'un iki adıdır.
3. **d/c: 1.** Sembolik bağın kendi i-node'udur; `stat` bağı izlemediği için hedefin değil bağın sayısı gelir.
4. **d/e: 2.** `d` 'deki girdi ve kendi `.` girdisi.

Ardından `rm d/a` yapılırsa `d/b` 'nin bağ sayısı 1'e iner ve veri durur. `d/c` ise `a` adını aradığı için askıda kalır.

10.3.6 Bağlama

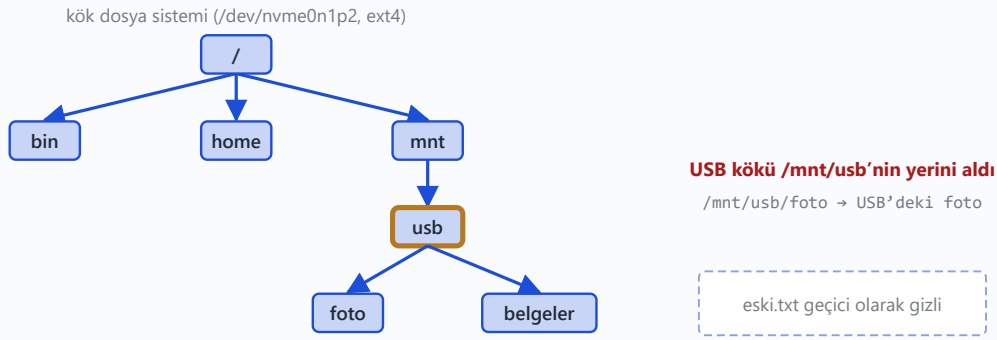
Bir makinede birden çok dosya sistemi bulunur: sistem diskinin bölümleri, takılan bir USB bellek, ağdan gelen bir paylaşım. UNIX bunları tek bir ağaçta birleştirir.

Tanım 10.6

Bağlama (*mounting*), bir dosya sisteminin kökünü var olan dizin ağacındaki bir dizine, yani **bağlama noktasına**, eklemektir.

Şekil 10.17 işlemi üç adımda gösterir. Başlangıçta iki ayrı dosya sistemi vardır: ext4 biçimli kök sistem ve exFAT biçimli bir USB bellek. `/mnt/usb` sıradan bir dizindir ve içinde `eski.txt` bulunur. `mount /dev/sdb1 /mnt/usb` çekirdeğin bağlama tablosuna bir kayıt ekler. Bundan sonra yol çözümü

`/mnt/usb`'ye geldiğinde USB'nin kök i-node'una atlar. `/mnt/usb/foto` artık USB'deki `foto` dizinidir. `eski.txt` silinmez, yalnızca gizlenir. `umount /mnt/usb`, o dosya sisteminde açık dosya yoksa bağlamayı kaldırır; açık dosya varsa `EBUSY` hatası verir. Bağlama kalkınca `eski.txt` yeniden görünür.



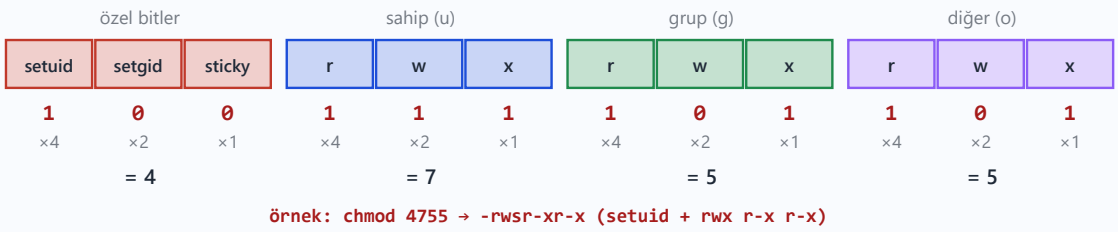
Şekil 10.17: `mount /dev/sdb1 /mnt/usb` sonrası: USB'nin kökü `/mnt/usb`'nin yerini alır; eski içerik geçici olarak gizlidir.

Açılışta önce kök dosya sistemi bağlanır. Diğerleri `/etc/fstab` dosyasındaki ya da systemd birimlerindeki tanımlara göre bağlanır. `findmnt` komutu bağlama ağacını gösterir. Bağlarken seçenek de verilebilir: `ro` salt okunur bağlar, `noexec` o sistemdeki programların çalıştırılmasını, `nosuid` de setuid bitinin etkisini kapatır. Takılan bir USB belleğe bu seçeneklerle güvenli bir sınır çizilir. Windows her birime bir sürücü harfi (`C:`, `D:`) verir, ama NTFS bir birimi boş bir klasöre de bağlayabilir.

10.4 Koruma

10.4.1 rwx izin bitleri

Paylaşılan bir sistemde her dosyaya kimin ne yapabileceği belirlenmelidir. UNIX bunun için kaba ama hızlı bir yöntem kullanır. Kullanıcılar üç sınıfa ayrılır: dosyanın **sahibi** (u), dosyanın **grubu** (g) ve **diğerleri** (o). Her sınıfın üç hakkı vardır: okuma (r), yazma (w) ve çalıştırma (x). Dokuz bit böylece sekizli üç basamakla yazılır: r = 4, w = 2, x = 1. `chmod 754 betik.sh`, sahibe 7 = rwx, gruba 5 = r-x, diğerlerine 4 = r-- verir ve `ls -l`'de `rwxr-xr--` görünür.



Şekil 10.18: 12 izin biti: özel bitler, sahip, grup ve diğerleri. Örnek 4755 = setuid + rwx r-x r-x.

Hakların anlamı dosyada ve dizinde farklıdır (Tablo 10.9). Dosyada r içeriği okumak, w değiştirmek, x program olarak çalıştırmaktır. Dizinin içeriği ise ad → i-node eşlemeleridir. Dizinde r girdileri listelemek (`ls`), w girdi eklemek, silmek ya da yeniden adlandırmak, x ise dizinin içinden geçmektir: yol çözümü ve `cd` bunu ister.

Hak	Dosyada	Dizinde
r	içeriği oku	girdileri listele (`ls`)
w	içeriği değiştir	girdi ekle, sil, yeniden adlandır (dosyanın kendi izni önemsiz)
x	program olarak çalıştır	içinden geç (yol çözümü, `cd`)

Tablo 10.9: r, w ve x'in dosyada ve dizinde anlamı.

Bu tablonun iki şaşırtıcı sonucu vardır. Birincisi: bir dosyayı silmek dosyanın değil, bulunduğu dizinin w hakkını ister. Silme bir dizin girdisini kaldırmaktır; dosyanın içeriğine dokunmaz. Salt okunur bir dosya, w hakkı olan bir dizinde silinebilir. İkincisi: dizinde x olup r olmaması anlamlı bir birleşimdir. `/home/ali` dizini `drwx--x--x` ise başka bir kullanıcı dizini listeleyemez. Ama içindeki bir dosyanın adını biliyorsa yol çözülür ve dosyanın kendi r biti izin veriyorsa okuyabilir.

Denetim sırası da önemlidir. Çekirdek önce isteyenin dosyanın sahibi olup olmadığına bakar; sahipse yalnızca sahip haklarını uygular. Sahip değilse ve dosyanın grubundaysa grup haklarını, hiçbiri değilse diğerlerinin haklarını uygular. İlk eşleşen sınıf kullanılır, sonrakilere bakılmaz.

Örnek 10.5

`-r--rw-rw- 1 ali ogr 120 rapor.txt` . ali, ogr grubunun da üyesidir. ali dosyaya yazabilir mi?

1. İsteyen ali, dosyanın sahibi de ali: sahip sınıfı eşleşir.
2. Sahip hakları `r--` 'dir; w yoktur.
3. Grup ve diğerleri sınıflarına hiç bakılmaz. Yazma `EACCES` ile reddedilir.

Sonuç ilk bakışta tuhaftır: grubun ve hatta herkesin yazabildiği bir dosyaya sahibi yazamaz. Sahip isterse `chmod u+w` ile kendine w verebilir; çünkü izinleri değiştirme hakkı sahibe aittir.

10.4.2 Özel bitler, umask ve ACL

Dokuz bitin önünde üç **özel bit** daha vardır (Şekil 10.18): setuid (4), setgid (2) ve sticky (1). **setuid** bitli bir program, onu çalıştıranın değil dosya sahibinin kimliğiyle çalışır. `passwd` komutu root'a ait ve setuid bitlidir; sıradan bir kullanıcı parolasını değiştirirken yalnız root'un yazabildiği parola dosyasını bu sayede günceller. `ls -l` , setuid'i sahibin x yerinde `s` olarak gösterir: 4755 → `-rwsr-xr-x` . **sticky** bit dizinlerde kullanılır. Herkesin yazabildiği bir dizinde bir girdiyi yalnızca dosyanın sahibi, dizinin sahibi ya da root silebilir veya yeniden adlandırabilir. `/tmp` böyledir: `drwxrwxrwt` (1777). Sticky bit açık ama diğerlerinin x hakkı kapalıysa son karakter büyük `T` olur: 1770 → `drwxrwx--T` .

Yeni bir dosyanın izinlerini **umask** belirler. Programlar dosyaları genellikle 666, dizinleri 777 izniyle oluşturmak ister. Çekirdek bu istekten umask'teki bitleri kapatır: izin = istek & ~umask. umask 027 iken yeni dosya 666 & ~027 = 640 (`rw-r-----`), yeni dizin 777 & ~027 = 750 (`rwxr-x---`) olur. Bu ayarda grup yalnızca okuyabilir, diğerleri hiçbir şey yapamaz.

> **Sık yapılan hata**
>
> umask'i istekten çıkarmak. Sekizli çıkarma 666 − 027 = 637 verir: grup yazabilir ve çalıştırabilir, diğerleri her şeyi yapabilir. umask bit bit maskedir: 666 & ~027 = 640'tır. İki işlem ancak umask'in bütün bitleri istekte de açıksa aynı sonucu verir.

Sahip, grup ve diğerleri ayrımı kaba tanelidir. “Bu dosyayı ali okuyabilsin, can okuyamasın, ikisi de aynı gruptaysa” gibi bir isteği karşılamaz. Bunun için **erişim denetim listesi** (*access control list*, ACL) kullanılır. ACL adlı kullanıcılar ve gruplar için ayrı girdiler tutar; Linux'ta `setfacl` ile kurulur ve `ls -l` çıktısında izinlerin sonunda bir `+` görünür. NTFS'in bütün koruması ACL tabanlıdır.

10.5 Disk düzeni

10.5.1 Bölümleme: MBR ve GPT

Bir disk bir ya da daha çok **bölüme** (*partition*) ayrılır. Her bölümde ayrı bir dosya sistemi bulunabilir; bir bölüm takas alanı, LVM fiziksel birimi ya da şifreli bir kap da olabilir. Bölümlerin listesi diskin başındaki bir tabloda durur. Bu tablonun iki biçimi vardır (Şekil 10.19).



Şekil 10.19: MBR ve GPT disk düzenleri.

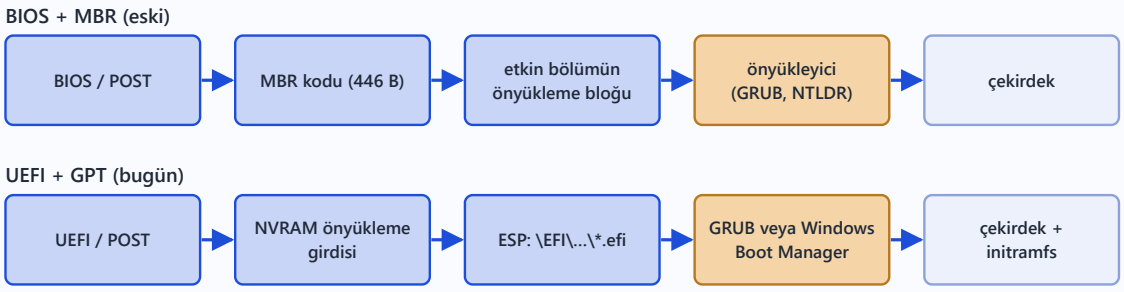
MBR (*master boot record*) 1983'ten beri kullanılan eski biçimdir. Diskin ilk sektörü olan LBA 0, 512 bayttır. İlk 446 baytta önyükleyici kodu, ardından dört adet 16 baytlık bölüm girdisi, sonunda da `55 AA` imzası bulunur. Her girdi bölümün başlangıcını ve uzunluğunu 32 bitlik sektör numarasıyla tutar. Bundan iki sınır doğar. En çok dört birincil bölüm açılabilir; daha fazlası için bir “genişletilmiş” bölümün içine mantıksal bölümler zincirlenir. İkinci ve daha ciddi sınır, adreslenebilen en büyük alandır: $2^{32} \times 512$ B = 2 TiB. 3 TiB'lık bir disk MBR ile bölümlenirse son 1 TiB'a hiçbir bölüm tanımlanamaz.

GPT (*GUID partition table*) UEFI standardının parçasıdır ve 64 bitlik LBA kullanır. LBA 0'da bir **koruyucu MBR** durur; bu sahte tablo, GPT'yi tanımayan eski araçların diski "boş" sanıp üzerine yazmasını önler. LBA 1'de GPT başlığı, LBA 2–33'te 128 baytlık 128 bölüm girdisi bulunur. Her bölümün kendine özgü bir GUID'i ve türünü gösteren bir tür GUID'i vardır. Başlık ve girdiler CRC32 sağlama toplamıyla korunur. Diskin sonunda yedekleri tutulur. 2 TiB'tan büyük diskler ve Windows 11 GPT gerektirir.

10.5.2 Açılış

Bölüm tablosunun biçimi makinenin nasıl açıldığını da belirler (Şekil 10.20). Güç verilince donanım kendini sınar (POST). Sonraki adımlar iki düzende farklıdır.

Eski **BIOS** 16 bitlik gerçek modda çalışır ve dosya sistemi tanımaz. Diskin ilk sektörünü, yani MBR'yi belleğe alıp çalıştırır. MBR kodu bölüm tablosunda "etkin" işaretli bölümü bulur ve onun ilk bloğunu, **önyükleme bloğunu**, çalıştırır. Bu blok asıl önyükleyiciyi (GRUB, eski Windows'ta NTLDR) yükler. **UEFI** ise 32 ya da 64 bit çalışır, kendi sürücüleri ve FAT dosya sistemi desteği vardır. Sektör değil dosya arar. Kalıcı bellekteki (NVRAM) önyükleme girdilerini okur ve EFI sistem bölümündeki (ESP, FAT32 biçimli) `.efi` programını yükler. Etkin bayrağı kullanılmaz. Her iki düzende de bundan sonra önyükleyici çekirdeği yükler; fark yalnızca önyükleyiciye nasıl ulaşıldığındadır.



Şekil 10.20: BIOS/MBR ve UEFI/GPT açılış zincirleri.

PC'ler 2012 civarından beri UEFI ile gelir. BIOS uyumluluk modu (CSM) 2020'den sonra çoğu yeni sistemden kaldırıldı.

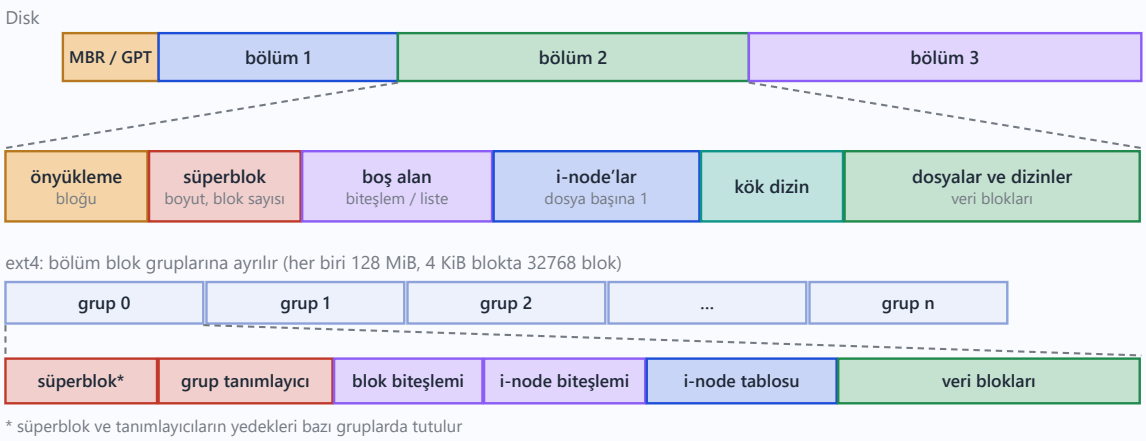
10.5.3 Bir dosya sisteminin düzeni

Bir bölümün içi, dosya sistemine göre değişse de benzer parçalardan oluşur (Şekil 10.21). Bölümün ilk bloğu **önyükleme bloğudur**. Bölüm açılış için kullanılmasa bile bu yer ayrılır. Diskin ilk bloğu ise MBR'dir; ikisini karıştırmamak gerekir.

Tanım 10.7

Süperblok (*superblock*), dosya sisteminin kimliğini ve temel parametrelerini tutan bloktur: sihirli sayı, blok boyutu, blok ve i-node sayıları, boş sayaçlar, bağlama durumu. Dosya sistemi bağlanırken ilk okunan yapıdır.

Süperbloğu boş alanı izleyen yapı (biteşlem ya da liste), dosya başına birer i-node'dan oluşan i-node tablosu, kök dizin ve geri kalan veri blokları izler. Bu düzende her dosya erişimi önce diskin başındaki i-node'a, sonra uzaktaki veri bloklarına gider. Sabit diskte bu, uzun kol hareketleri demektir. ext4 bu yüzden bölümü **blok gruplarına** ayırır. 4 KiB blokla her grup 32.768 blok, yani 128 MiB'tır. Her grubun kendi biteşlemleri ve i-node tablosu vardır. Bir dosyanın i-node'u ve verisi aynı gruba konarak baş hareketi azaltılır. Fikir BSD'nin hızlı dosya sistemindeki (FFS) silindir gruplarından gelir. Süperblok ve grup tanımlayıcılarının yedekleri bazı gruplarda tutulur; asıl süperblok bozulursa yedekten kurtarılır.



Şekil 10.21: Bir bölümdeki dosya sistemi düzeni ve ext4 blok grupları.

10.5.4 i-node numarasından disk adresine

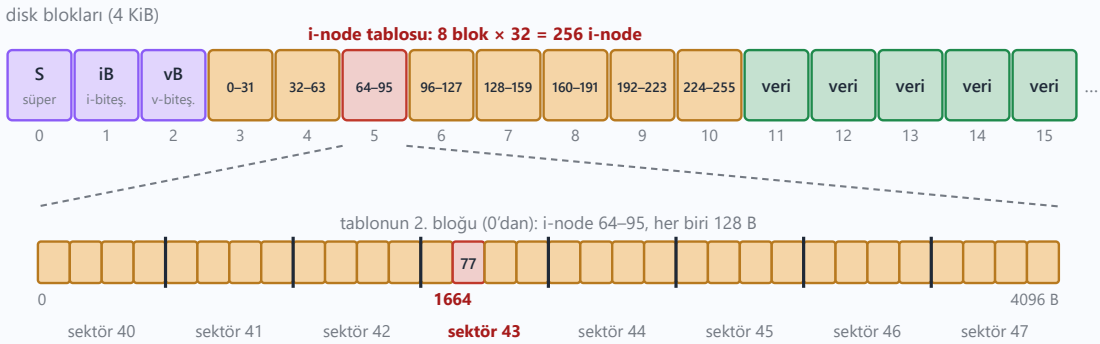
i-node tablosu ardışık bloklarda durduğu için bir i-node'un yeri numarasından hesaplanır. Bunun için süperbloktan üç bilgi gerekir: tablonun başladığı blok, blok boyu ve i-node boyu. Hesap üç adımlıdır:

> bağıl konum = n × i-node boyu, (tablo bloğu, blok içi) = (bağıl konum div blok, bağıl konum mod blok), disk bloğu = tablonun başı + tablo bloğu

Örnek 10.6

Blok 4 KiB, i-node 128 B. Blok 0 süperblok, blok 1–2 biteşlemler, blok 3–10 i-node tablosudur. i-node numaraları 0'dan başlar; disk 512 baytlık sektörlerle adreslenir. i-node 77 diskte nerededir?

1. **Blok başına i-node.** 4096 / 128 = 32. Tablo 8 blokta 256 i-node tutar.
2. **Tablo başından bağıl konum.** 77 × 128 = 9856 = 2 × 4096 + 1664. Tablonun 2. bloğu (0'dan sayarak), blok içinde 1664. bayt; 1664 / 128 = 13, yani o bloğun 13. i-node'u.
3. **Disk bloğu.** 3 + 2 = blok 5.
4. **Sektör.** Bayt adresi 5 × 4096 + 1664 = 22.144; 22.144 / 512 = 43,25 → sektör 43, içinde 0,25 × 512 = 128. bayt. Bir sektörde 512 / 128 = 4 i-node vardır.



Şekil 10.22: i-node 77'nin yeri: tablonun 2. bloğu (disk bloğu 5), blok içinde 1664. bayt, sektör 43.

Süperbloktaki "i-node tablosu nerede" bilgisi bu hesap için vazgeçilmezdir. ext4'te iki ayrıntı eklenir. Numaralar 1'den başlar ve tablo blok gruplarına bölünür. Grup başına g i-node varsa grup = (n − 1) div g, grup içi indis = (n − 1) mod g'dir. g = 8192 ve i-node boyu 256 B iken i-node 20.000 için: 19.999 div 8192 = 2, 19.999 mod 8192 = 3615. Grubun tablosundaki bağıl konum 3615 × 256 = 925.440 = 225 × 4096 + 3840'tır: grubun tablosunda blok 225, bağıl konum 3840.

Sık yapılan hata

Tablo bloğunu disk bloğu sanmak. 9856 div 4096 = 2, tablonun 2. bloğudur; disk bloğu tablonun başlangıcı eklenerek bulunur (3 + 2 = 5). Numaraların 0'dan mı 1'den mi başladığı da hesabı bir i-node kaydırır.

10.6 Tahsis yöntemleri

Dosya sistemini gerçeklemenin merkezindeki soru şudur: bir dosyanın bloklarının diskte nerede olduğu nasıl kaydedilir? Kayıt yöntemi üç şeyi belirler: boş alanın ne kadar verimli kullanıldığını, dosyanın büyüyüp büyüyemeyeceğini ve dosyanın ortasındaki bir bloğa ne kadar hızlı ulaşıldığını. Dört temel yöntem vardır: bitişik, bağlı liste, FAT ve indisli tahsis.

10.6.1 Aynı dosya, dört yöntem

Silberschatz'ın örneğindeki `jeep` dosyası beş bloktur. Aşağıda onu 32 bloklu bir diske dört yöntemle yerleştiriyoruz; Şekil 10.23 indisli yerleşimi gösterir.

- **Bitişik:** beş blok yan yana, 9–13'te durur. Dizin girdisinde yalnız başlangıç (9) ve uzunluk (5) vardır. i. blok = başlangıç + i.
- **Bağlı liste:** bloklar diske dağılmıştır: 9, 16, 1, 10, 25. Her blok bir sonrakinin numarasını taşır; dizin girdisi ilk ve son bloğu bilir.
- **FAT:** aynı zincir, ama işaretçiler bloklarda değil, bellekteki bir tabloda: FAT[9] = 16, FAT[16] = 1, FAT[1] = 10, FAT[10] = 25, FAT[25] = EOF. Blokların tamamı veriye kalır.
- **İndisli:** beş adresin tamamı tek bir indis bloğunda (19) sırayla durur: 9, 16, 1, 10, 25. i. bloğa doğrudan erişilir.



Şekil 10.23: `jeep` dosyasının indisli tahsisi: dizin girdisi indis bloğunu (19), indis bloğu da beş veri bloğunu sırayla gösterir.

10.6.2 Bitişik tahsis

> **Tanım 10.8**
>
> **Bitişik tahsis** (*contiguous allocation*), her dosyayı diskte ardışık bloklara yerleştirmektir. Dizin girdisi (başlangıç, uzunluk) çiftini tutar.

Yöntemin artıları belirgindir. Gerçeklemesi basittir: yalnızca iki sayı saklanır. Rastgele erişim kolaydır: dosyanın i. bloğu başlangıç + i'dir. Bütün dosya tek bir arama ve ardışık okumalarla okunur; sabit diskte bundan daha hızlı bir düzen yoktur.

Eksileri ise zaman geçtikçe ortaya çıkar. Tablo 10.10 Tanenbaum'un örneğini izler. 40 bloklu disk yedi dosyayla doludur: A 4, B 3, C 6, D 5, E 12, F 6, G 4 blok. D ve F silinir. Geride iki delik kalır: 13–17'de 5 blok, 30–35'te 6 blok. Toplam 11 boş blok vardır. Şimdi 8 blokluk H dosyası gelir. Toplam boş alan yeter, ama hiçbir delik 8 blok değildir: H sığmaz.

Durum	Delikler (başlangıç, boy)	Toplam boş	En büyük delik
7 dosya, disk dolu	—	0	—
D ve F silindi	(13, 5), (30, 6)	11	6
H (8 blok) geliyor	(13, 5), (30, 6)	11	6 < 8 → sığmaz
sıkıştırma sonrası	H 29–36'ya yerleşir, (37, 3)	3	3

Tablo 10.10: Bitişik tahsiste dış parçalanma (40 blok; Python ile yeniden hesaplandı).

Bu, bellek yönetimindeki **dış parçalanmanın** aynısıdır. Çözümlerden biri sıkıştırmadır (birleştirme): E ve G sola kaydırılır, tek bir 11 blokluk delik oluşur ve H yerleşir (Şekil 10.24). Bedeli yüksektir: diskte 12 + 4 = 16 bloğun okunup yeniden yazılması gerekir. Bitişik tahsisin ikinci sorunu, dosyanın son boyunun baştan bilinmesi gerekmesidir. Bir dosya büyümek isterse arkasındaki blok doluysa ya büyüyemez ya da bütünüyle başka yere taşınmalıdır.



Şekil 10.24: Sıkıştırma: G ve E kaydırıldı, tek bir 11 blokluk delik oluştu ve H yerleşti. Bedeli 16 bloğun yeniden yazılmasıdır.

Bu eksiklere karşın bitişik tahsis, içeriği bir kez yazılıp değişmeyen ortamlarda bugün de kullanılır: CD-ROM ve DVD dosya sistemleri (ISO 9660, UDF) böyledir. Yöntemin bir de modern torunu vardır. Dosya tek bir parça yerine birkaç bitişik parçaya bölünür. Bu parçalara **extent** denir; ext4 ve NTFS'in temeli budur.

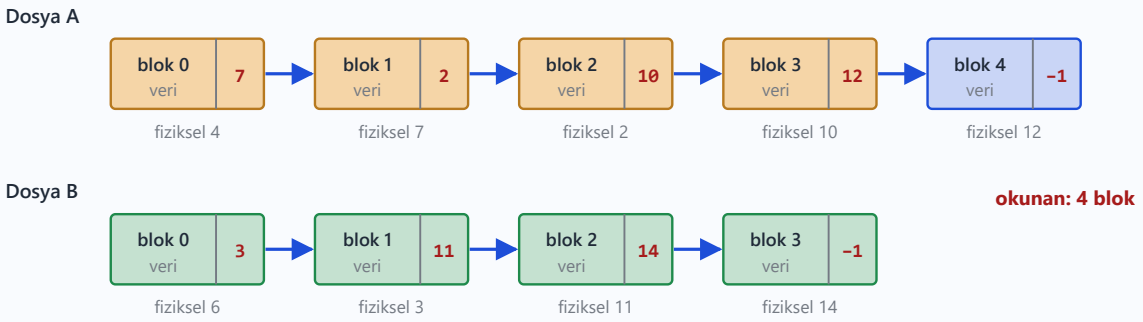
10.6.3 Bağlı liste ile tahsis

Tanım 10.9

Bağlı liste ile tahsiste (*linked allocation*) her blok, dosyanın bir sonraki bloğunun numarasını taşır. Dizin girdisi ilk bloğu (Silberschatz'ın düzeninde son bloğu da) bilir.

Bu yöntemde dış parçalanma yoktur: herhangi bir boş blok dosyanın sonuna eklenebilir. Dosya kolayca büyür ve boyutunun baştan bilinmesi gerekmez. İç parçalanma ise her yöntemde olduğu gibi burada da vardır: son bloğun doldurulmayan kısmı boşa gider.

Sorun rastgele erişimdedir. Şekil 10.25'teki A dosyasının blokları 4 → 7 → 2 → 10 → 12 zincirindedir. Mantıksal 3. bloğu (0'dan sayarak) okumak isteyelim. Dizin girdisi yalnızca fiziksel 4'ü bilir. Fiziksel 4 okunur: sonraki 7'dir. 7 okunur: sonraki 2. 2 okunur: sonraki 10. Ancak şimdi fiziksel 10'un mantıksal 3. blok olduğu anlaşılır ve okunur. Tek bir blok için 4 disk okuması yapılmıştır. Mantıksal k. blok k + 1 okuma ister; rastgele erişim O(n)'dir. Sabit diskte bu okumaların her biri ayrı bir arama da olabilir.



Şekil 10.25: Bağlı liste tahsisi: A'nın mantıksal 3. bloğuna (fiziksel 10) ulaşmak için 4 blok okunur.

İki küçük sorun daha vardır. Her bloğun birkaç baytı işaretçiye gittiği için bloktaki veri miktarı 4096 − 4 = 4092 bayt olur. Bu sayı ikinin kuvveti değildir; 4096 baytlık sayfalarla çalışan programlar için her okuma iki bloğa yayılır. İkincisi güvenilirliktir: tek bir işaretçi bozulursa zincirin geri kalanı, yani dosyanın sonu, kaybolur.

10.6.4 FAT: tablo ile tahsis

Bağlı listenin iki sorununun da çaresi işaretçileri bloklardan çıkarmaktır. İşaretçiler diskin başındaki tek bir tabloda toplanır: **dosya tahsis tablosu** (*file allocation table*, FAT). Tablonun i. girdisi, i. bloktan sonra gelen bloğun numarasıdır. Boş bloklar 0, zincirin sonu özel bir EOF değeriyle işaretlenir. Bloklar artık tamamen veridir.



Şekil 10.26: FAT tablosunda A dosyasının 4 → 7 → 2 → 10 → 12 zinciri; B dosyası 6 → 3 → 11 → 14.

Şekil 10.26'daki tabloda iki dosya vardır. A için: girdi 4 = 7, girdi 7 = 2, girdi 2 = 10, girdi 10 = 12, girdi 12 = EOF. B için: 6 → 3 → 11 → 14 → EOF. Rastgele erişim hâlâ zinciri izlemeyi gerektirir, ama tablo bellekte durduğu için zincir bellekte izlenir: disk okuması yoktur. Tablo çok önemli olduğu için diskte iki kopya hâlinde tutulur.

FAT'ın asıl sorunu boyutudur. Tablonun diskteki her blok için bir girdisi vardır; tablo disk boyutuyla orantılı büyür ve hızlı olması için tamamı bellekte durmalıdır.

Örnek 10.7

200 GiB'lık bir disk 1 KiB'lık bloklara bölünmüş; her FAT girdisi 4 bayt. Tablo ne kadar bellek ister?

1. Blok sayısı: $200 \times 2^{30} / 2^{10} = 200 \times 2^{20} = 209.715.200$.
2. Tablo: $209.715.200 \times 4$ B = 838.860.800 B = 800 MiB. Girdi 3 bayta sıkıştırılsa bile 600 MiB.

Gerçek FAT32'de blok yerine birkaç bloktan oluşan **kümeler** (*cluster*) kullanılır. 2 TiB'lık bir birim 32 KiB'lık kümelerle $2^{41} / 2^{15} = 67.108.864$ küme eder; 4 baytlık girdilerle tablo 256 MiB tutar.

10.6.5 Kod izleme: FAT zinciri

Aşağıdaki fonksiyon, ilk bloğu `ilk` olan bir dosyada `ofset`. baytı içeren disk bloğunu bulur. Blok boyu `BLOK` = 1024 bayttır; `fat` bellekteki tablodur, `SON` zincirin sonunu gösterir.

```
int blok_bul(int ilk, long ofset) {
  int b = ilk;
  long n = ofset / BLOK; /* blok sırası */
  while (n > 0 && b != SON) {
    b = fat[b];          /* zinciri izle */
    n--;
  }
  return b;          /* SON: dosya dışı */
}
```

`n = ofset / BLOK` tam sayı bölmesidir ve aranan baytın dosyanın kaçıncı bloğunda (0'dan sayarak) olduğunu verir. Döngü zincirde n adım ilerler; her adımda `b` bir sonraki bloğa geçer. Döngünün ikinci koşulu `b != SON` zincirin erken bitmesine karşı bir korumadır. İstenen blok dosyada yoksa fonksiyon `SON` döndürür. Çalışma süresi O(n)'dir, ama her adım yalnızca bir bellek erişimidir.

İki çağrı izleyelim (Tablo 10.11). `blok_bul(4, 3500)`'de A dosyası blok 4'ten başlar. 3500 / 1024 = 3: aranan bayt dosyanın 3. bloğundadır ve zincirde 3 adım ilerlemek gerekir. fat[4] = 7 (n = 2), fat[7] = 2 (n = 1), fat[2] = 10 (n = 0). Döngü biter; disk bloğu 10'dur. Bayt 3500 bu bloğun $3500 - 3 \cdot 1024 = 428$. baytıdır.

Çağrı	Başta n	İzlenen zincir (b, n)	Dönüş
`blok_bul(4, 3500)`	3	(4, 3) → (7, 2) → (2, 1) → (10, 0)	10; blok içi 428. bayt

Çağrı	Başta n	İzlenen zincir (b, n)	Dönüş
`blok_bul(6, 5000)`	4	(6, 4) → (3, 3) → (11, 2) → (14, 1) → (SON, 0)	SON: dosya dışı

Tablo 10.11: `blok_bul` izlemesi (Python'a birebir çevrilerek çalıştırıldı).



Şekil 10.27: `blok_bul(4, 3500)` 'ün son adımı: izlenen zincir 4 → 7 → 2 → 10.

`blok_bul(6, 5000)` daha öğreticidir. B dosyası 4 bloktur, yani en çok 4096 bayt tutar. 5000 / 1024 = 4: dosyanın 4. bloğu (beşinci blok) istenir. Zincir 6 → 3 → 11 → 14 diye ilerler; dördüncü adımda fat[14] = SON gelir ve n = 0 olur. Dosyanın 4. bloğu yoktur, bağıl konum dosyanın dışındadır. Bu bağıl konumdan okuyan `read` 0 bayt döndürür (dosya sonu). Aynı bağıl konuma `write` yapılırsa dosya sistemi yeni bir blok ayırıp 14'ün girdisine bağlar.

Örnek 10.8

Küme boyu 4 KiB; FAT'ta FAT[5] = 9, FAT[9] = 2, FAT[2] = 11, FAT[11] = EOF. `sunum.pptx` 'in ilk kümesi 5. Dosya hangi kümelerde durur, 9000. bayt hangi kümededir?

1. 9000 / 4096 = 2 (tam sayı bölmesi): bayt, dosyanın 2. kümesinde (0'dan sayarak).
2. Zincir: 5 (0. küme) → 9 (1.) → 2 (2.) → 11 (3.) → EOF. Dosya 4 kümededir; en çok 16 KiB tutar ve diskte dağınıktır.
3. 2. küme fiziksel küme 2'dir. Küme içindeki konum 9000 − 2 × 4096 = 808. bayttır.

10.6.6 FAT ailesi

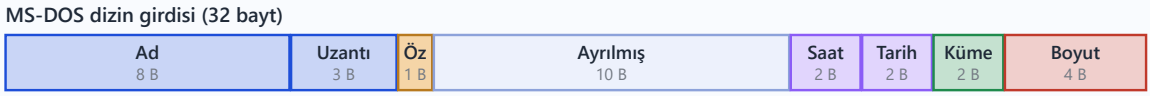
FAT adı, MS-DOS'tan bugüne bir aileyi anlatır. Aile üyeleri FAT girdisinin bit sayısıyla adlandırılır (Tablo 10.12). Girdi ne kadar genişse o kadar çok küme adreslenir.

	FAT12	FAT16	FAT32	exFAT
FAT girdisi	12 bit	16 bit	28 bit kullanılır	32 bit
En büyük birim	≈ 16 MiB	2 GiB (32 KiB küme)	2 TiB (512 B sektör)	128 PiB
En büyük dosya	birim kadar	2 GiB	4 GiB − 1 bayt	16 EiB (kuramsal)
Boş alan	FAT'ta 0	FAT'ta 0	FAT'ta 0 + FSInfo ipucu	ayrı biteşlem

	FAT12	FAT16	FAT32	exFAT
Kullanım	disket	eski diskler	USB, UEFI ESP	SDXC/SDUC, büyük USB

Tablo 10.12: FAT12, FAT16, FAT32 ve exFAT.

FAT'ta dizin girdisi her şeyi taşır: ad, uzantı, öznitelikler, zaman, tarih, ilk küme ve boyut (Şekil 10.28). Boyut alanı 32 bit olduğundan FAT32'de bir dosya en çok 4 GiB − 1 bayt olabilir. 4 GiB'lık bir film dosyası FAT32 biçimli USB belleğe kopyalanamaz. exFAT bu sınırı kaldırır. Ayrıca boş alanı ayrı bir biteşlemde tutar ve bitişik olduğu bilinen dosyalarda FAT zincirini hiç kullanmayabilir. Windows'un grafik biçimlendirme aracı FAT32'yi tarihsel olarak 32 GB ile sınırlar; bu sınır dosya sisteminin değil aracındır.



Öz: öznitelik. FAT32: "ayrılmış" alana oluşturma/erişim zamanı ve ilk kümenin üst 16 biti yerleşti; uzun adlar ek girdilerde.

Şekil 10.28: MS-DOS'un 32 baytlık dizin girdisi. FAT32'de "ayrılmış" alana oluşturma ve erişim zamanı ile ilk kümenin üst 16 biti yerleşti.

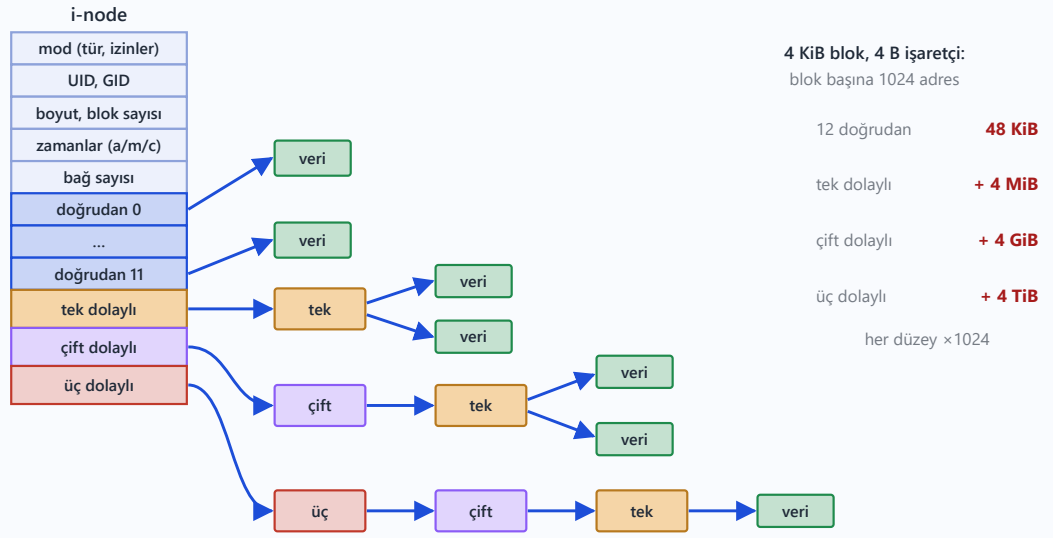
10.6.7 İndisli tahsis ve i-node

> **Tanım 10.10**
>
> **İndisli tahsiste** (*indexed allocation*) bir dosyanın bütün blok adresleri bir indis bloğunda toplanır; i. girdi i. bloğun adresidir. UNIX'te bu yapıya **i-node** (indis düğümü) denir. i-node, dosyanın blok adreslerini ve özniteliklerini birlikte tutar.

İndisli tahsis bağlı listenin iyi yanlarını korur: dış parçalanma yoktur, dosya boş blok olan her yere büyür. Üstüne rastgele erişimi de hızlandırır: i. bloğun adresi indis bloğunun i. girdisindedir, zincir izlemek gerekmez.

i-node'un FAT'a göre büyük bir üstünlüğü bellek kullanımındadır. Bir i-node yalnızca dosya açıkken bellekte bulunur. K dosya açıksa gereken bellek K × i-node boyutudur. FAT gibi disk boyutuyla değil, açık dosya sayısıyla orantılıdır. 1000 açık dosya ve 256 baytlık i-node'larla 256.000 bayt (yaklaşık 250 KiB) yeter; disk ister 1 TB ister 100 TB olsun.

Bedeli iki yönlüdür. Küçük bir dosyada bile indis için yer harcanır. i-node sabit boyutlu olduğu için büyük bir dosyanın bütün adresleri ona sığmaz. Çözüm, i-node'un son birkaç girdisini veri bloğu yerine başka indis bloklarını gösterecek biçimde kullanmaktır: **dolaylı bloklar**.



Şekil 10.29: UNIX i-node'u: öznitelikler, 12 doğrudan blok adresi, tek, çift ve üç dolaylı bloklar. 4 KiB blokla her düzey kapasiteyi 1024 kat büyütür.

Şekil 10.29 ext2/ext3'ün ve klasik UNIX'in yapısını gösterir. i-node'da 12 **doğrudan** işaretçi vardır; dosyanın ilk 12 bloğunun adresleri doğrudan buradadır. 13. işaretçi **tek dolaylıdır**: veri değil, adreslerle dolu bir indis bloğunu gösterir. 14. işaretçi **çift dolaylıdır**: indis bloklarını gösteren bir indis bloğunu gösterir. 15. işaretçi **üç dolaylıdır**: bir düzey daha. Küçük dosyalar yalnızca doğrudan işaretçileri kullanır ve dolaylı blok hiç ayrılmaz. Dosya büyüdükçe ağaç derinleşir. V7 UNIX'te doğrudan işaretçi sayısı 10'du.

10.6.8 En büyük dosya boyutu

Bu yapıyla bir dosyanın alabileceği en büyük boyut hesaplanabilir. Varsayımlar: blok 4 KiB = 4096 B, blok adresi 4 B. Bir indis bloğu 4096 / 4 = 1024 adres tutar; P = 1024 diyelim.

Örnek 10.9

i-node'da 12 doğrudan, 1 tek, 1 çift ve 1 üç dolaylı işaretçi var. En büyük dosya kaç bayttır?

1. **Doğrudan:** 12 blok × 4 KiB = 48 KiB (49.152 B).
2. **Tek dolaylı:** bir indis bloğu 1024 veri bloğu gösterir: 1024 × 4 KiB = 4 MiB.
3. **Çift dolaylı:** 1024 indis bloğunun her biri 1024 veri bloğu gösterir: 1024^2 = 1.048.576 blok = 4 GiB.
4. **Üç dolaylı:** bir düzey daha: 1024^3 = 1.073.741.824 blok = 4 TiB.
5. **Toplam:** $12 + 1024 + 1024^2 + 1024^3$ = 1.074.791.436 blok; × 4096 = 4.402.345.721.856 B ≈ 4,004 TiB.

Boyutun neredeyse tamamı üç dolaylı bloktan gelir; doğrudan işaretçilerin katkısı ihmal edilecek kadar küçüktür.

Genel formül şudur: D doğrudan işaretçi, blok boyu B ve adres boyu a ise P = B / a ve en büyük boyut $(D + P + P^2 + P^3) \times B$'dir. Tanenbaum'un örneğinde blok 1 KiB, adres 4 B (P = 256) ve 10 doğrudan işaretçi vardır: 10 + 256 + 65.536 + 16.777.216 = 16.843.018 blok, yaklaşık 16 GiB.

Bu hesap yalnızca işaretçi yapısının sınırıdır. Gerçek sistemlerde başka sınırlar daha önce devreye girer. ext2/ext3'te i-node'daki blok sayacı 32 bittir ve 512 baytlık birimleri sayar. 4 KiB blokla gerçek sınır $2^{32} \times 512$ B = 2 TiB'tır. ext4 extent'lerle bu sınırı 16 TiB'a çıkarır.

Sık yapılan hata

Çift dolaylı kapasiteyi 2 × 1024 blok sanmak. Çift dolaylı blok 1024 indis bloğunu, her biri de 1024 veri bloğunu gösterir: çarpım 1024^2'dir. Her dolaylılık düzeyi kapasiteyi P ile çarpar, P kadar artırmaz.

10.6.9 Bağıl konumdan disk bloğuna

Bir `read` çağrısında dosya sistemi bir bayt bağıl konumunu disk bloğuna çevirir. Önce mantıksal blok numarası bulunur: lb = bağıl konum div 4096. Sonra bu numaranın hangi bölgeye düştüğüne bakılır. Aynı yapıda (12 doğrudan, P = 1024) bölgeler şöyledir:

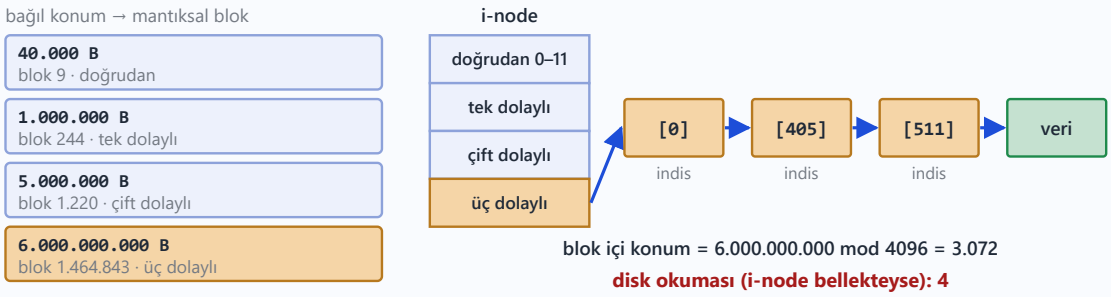
- lb < 12: doğrudan işaretçi lb. i-node bellekteyse 1 okuma (veri).
- 12 ≤ lb < 12 + 1024: tek dolaylı bloğun (lb − 12). girdisi. 2 okuma.
- sonraki 1024^2 blok: çift dolaylı. r = lb − 1036; dış indis r div 1024, iç indis r mod 1024. 3 okuma.
- sonraki 1024^3 blok: üç dolaylı. r = lb − 1.049.612; indisler r div 1024^2, (r div 1024) mod 1024 ve r mod 1024. 4 okuma.

Bağıl konum (B)	Mantıksal blok	Düzey ve indisler	Blok içi konum	Okuma
40.000	9	doğrudan 9	3136	1
1.000.000	244	tek dolaylı [232]	576	2
5.000.000	1220	çift dolaylı [0][184]	2880	3
6.000.000.000	1.464.843	üç dolaylı [0][405][511]	3072	4

Tablo 10.13: Dört bağıl konumun çevrilmesi (4 KiB blok, 12 doğrudan; Python ile doğrulandı).

Tablonun satırları tek tek izlenebilir. 40.000 / 4096 = 9 ve 9 < 12'dir: doğrudan işaretçi 9 bloğun adresini verir. 1.000.000 / 4096 = 244; 244 − 12 = 232 < 1024 olduğundan tek dolaylı bloğun 232. girdisine bakılır. 5.000.000 / 4096 = 1220; 1220 − 12 − 1024 = 184 çift dolaylı bölgeye düşer: dış indis 184

div 1024 = 0, iç indis 184. 6 GB için 6.000.000.000 / 4096 = 1.464.843 bulunur. $12 + 1024 + 1024^2 = 1.049.612$ çıkarılınca 415.231 kalır. Üç indis 415.231 div 1024^2 = 0, (415.231 div 1024) mod 1024 = 405 ve 415.231 mod 1024 = 511'dir.



Şekil 10.30: 6.000.000.000. baytın yolu: i-node → üç dolaylı blok [0] → çift dolaylı blok [405] → tek dolaylı blok [511] → veri. i-node bellekteyse 4 okuma.

Okuma sayıları en kötü durumdur. Sıralı okuyan bir program aynı dolaylı bloklara tekrar tekrar başvurur; önbellek ilk düzeyleri bellekte tutar ve ek okumaların çoğu kaybolur.

Örnek 10.10

1000 bloklu bir dosyanın mantıksal 699. bloğu okunacak. Dizin girdisi ve i-node bellekte; başka hiçbir şey önbellekte değil. Blok 4 KiB, adres 4 B, i-node'da 12 doğrudan işaretçi var. Dört yöntemde kaç disk okuması gerekir?

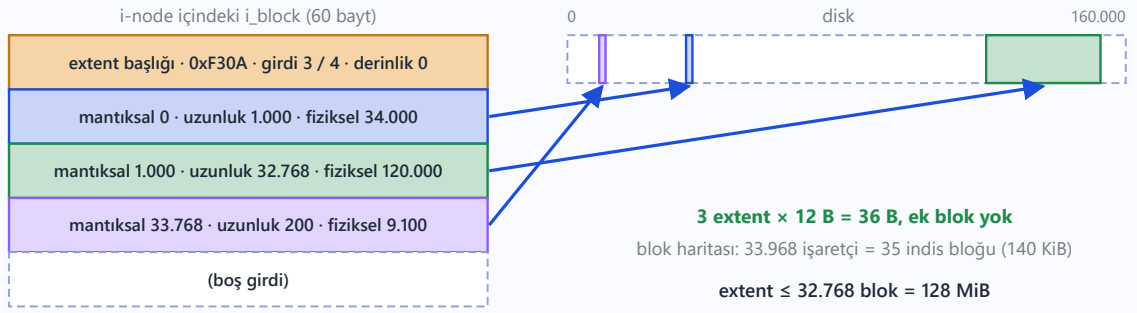
1. **Bitişik: 1.** Adres = başlangıç + 699.
2. **Bağlı liste: 700.** 0 … 698 numaralı bloklar okunmadan 699'un adresi öğrenilemez; 699 da okunur.
3. **FAT: 1.** Zincirde 699 adım bellekte izlenir, sonra tek okuma yapılır.
4. **i-node: 2.** 699 − 12 = 687 < 1024: tek dolaylı blok ve veri bloğu.

10.6.10 Extent'ler ve NTFS

Klasik i-node her blok için ayrı bir adres tutar. Büyük ve çoğunlukla bitişik bir dosyada bu çok israflıdır: art arda 32.768 bloğu anlatmak için 32.768 adres yazılır. ext4 bunun yerine **extent** kullanır.

Tanım 10.11

Extent, bitişik bir blok dizisini tek kayıtla anlatan (mantıksal başlangıç, uzunluk, fiziksel başlangıç) üçlüsüdür.

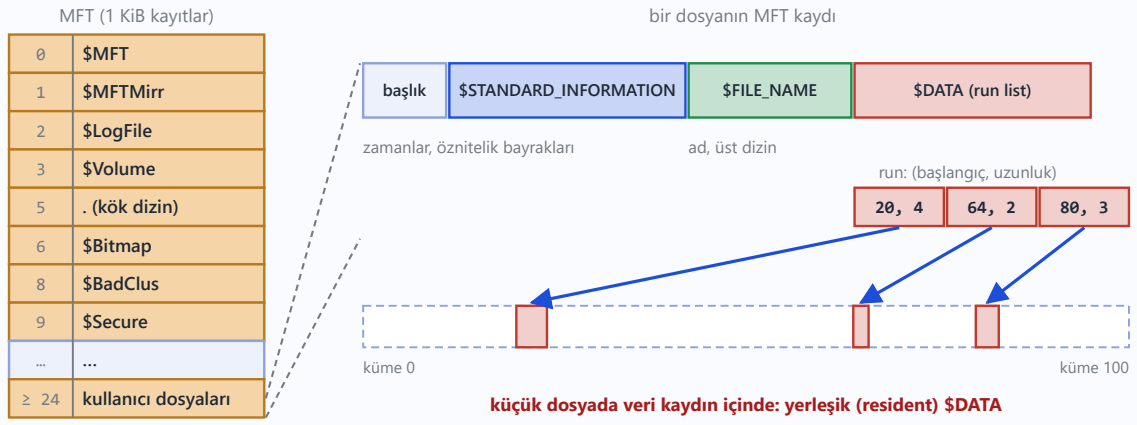


Şekil 10.31: ext4 i-node'unda extent başlığı ve üç extent; her extent diskte bitişik bir bölgeyi tanımlar.

Şekil 10.31'deki dosya üç extent'tir. Mantıksal 0'dan başlayan 1000 blok fiziksel 34.000'de, mantıksal 1000'den başlayan 32.768 blok fiziksel 120.000'de, mantıksal 33.768'den başlayan 200 blok fiziksel 9100'dedir. Dosya toplam 33.968 bloktur. Her extent 12 bayt tuttuğu için üç extent 36 bayta sığar; i-node'un 60 baytlık `i_block` alanında yer vardır, ek blok gerekmez. Aynı dosya klasik blok haritasıyla 33.968 işaretçi ister. Bunlar 12 doğrudan işaretçi, 1 tek dolaylı blok, 1 çift dolaylı blok ve onun gösterdiği 33 indis bloğuna yayılır: 35 indis bloğu, yani 140 KiB.

ext4'te fiziksel adres 48 bittir: 4 KiB blokla birim 1 EiB'a kadar büyüyebilir. Bir dosya en çok 16 TiB olabilir. Bir extent en çok 32.768 blok (128 MiB) anlatır. i-node'a 4 extent sığar; daha fazlası gerekirse girdiler bir **extent ağacının** indis düğümlerini gösterir. Extent'ler ancak dosya gerçekten bitişik yazılırsa işe yarar. ext4 bunun için **gecikmeli tahsis** (*delayed allocation*) kullanır: veri önce bellekte bekler, bloklar ancak diske yazılırken seçilir. O anda dosyanın ne kadar büyüdüğü bilindiği için çoklu blok ayırıcı büyük bitişik alanlar bulabilir. XFS, Btrfs, APFS ve NTFS de extent kullanır.

NTFS'in merkezinde **ana dosya tablosu** (*master file table*, MFT) durur (Şekil 10.32). NTFS'te her şey bir dosyadır; MFT'nin kendisi de MFT'nin 0. kaydıdır. İlk kayıtlar sistemin kendi meta dosyalarına ayrılmıştır: `$MFTMirr` (yedek), `$LogFile` (günlük), `$Volume`, kök dizin (kayıt 5), `$Bitmap` (boş küme biteşlemi) gibi. Kullanıcı dosyaları 24. kayıttan başlar.



Şekil 10.32: NTFS ana dosya tablosu: meta dosyalar ve bir dosya kaydının öznitelikleri, veri koşuları.

Her dosya en az bir 1 KiB'lık MFT kaydıdır ve kaydın içeriği bir öznitelikler listesidir. `$STANDARD_INFORMATION` zamanları ve bayrakları, `$FILE_NAME` adı ve üst dizini tutar. Verinin kendisi de bir özniteliktir: `$DATA`. Küçük bir dosyanın verisi kaydın içine sığar; buna **yerleşik** (*resident*) veri denir ve dosyayı okumak için başka bir blok okunmaz. Büyük veri ise koşu listesi (*run list*) olarak tutulur. Örnekteki (20, 4), (64, 2), (80, 3) üç extent'tir. Dizinler B+ ağacıdır (`$INDEX_ROOT`, `$INDEX_ALLOCATION`). NTFS katı bağ, reparse point, alternatif veri akışı (ADS), sıkıştırma, EFS şifreleme ve kota da destekler. Windows sunucularında bunlara ek olarak yazınca kopyalama kullanan ve sağlama toplamı tutan ReFS vardır.

10.6.11 Dosya sistemlerinin karşılaştırması

Tablo 10.14 bugün yaygın dosya sistemlerini sınırları ve çökme tutarlılığı yöntemleriyle karşılaştırır.

Dosya sistemi	En büyük dosya	En büyük birim	Çökme tutarlılığı	Tipik kullanım
FAT32	4 GiB − 1 B	2 TiB	yok (chkdsk/fsck)	USB, UEFI ESP
exFAT	16 EiB*	128 PiB	yok	SD kart, harici disk
NTFS	16 EiB* (Windows: 256 TiB–8 PiB)	256 TiB–8 PiB	meta veri günlüğü	Windows
ext4	16 TiB (4 KiB blok)	1 EiB	günlük (jbd2)	Linux, Android
XFS	8 EiB	8 EiB	meta veri günlüğü	RHEL, büyük sunucular
Btrfs	16 EiB	16 EiB	yazınca kopyalama + sağlama toplamı	Fedora, openSUSE, NAS
ZFS	16 EiB	2^{128} B (kuramsal)	yazınca kopyalama + sağlama toplamı	FreeBSD, TrueNAS, sunucu
APFS	8 EiB	8 EiB	yazınca kopyalama, anlık görüntü	macOS, iOS

Tablo 10.14: Dosya sistemlerinin karşılaştırması (* kuramsal sınır; işletim sistemi ve küme boyu daha düşük sınır koyar).

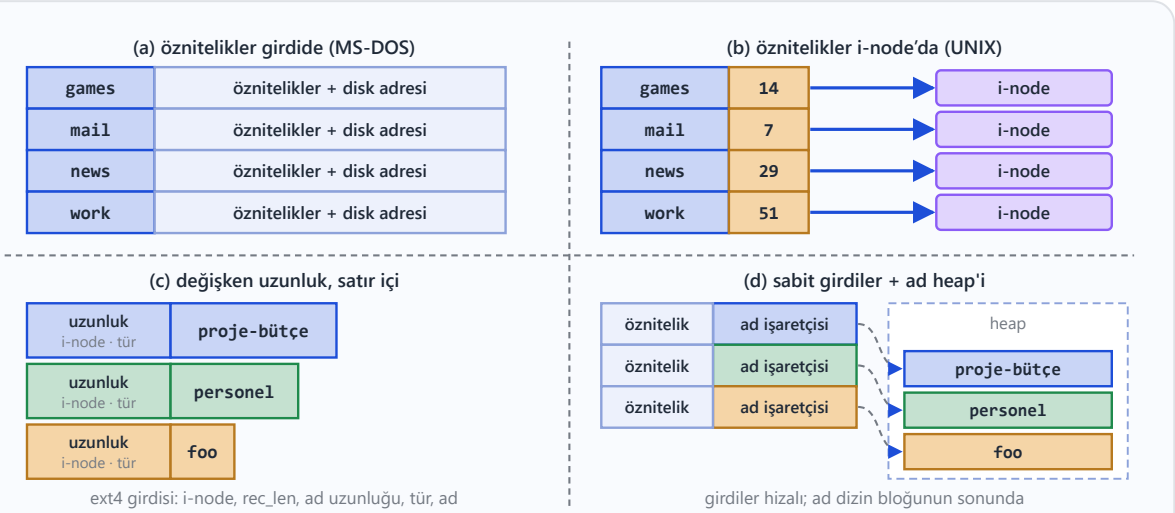
Windows 10'un 1709 sürümünden sonra NTFS 2 MiB'lık kümelerle 8 PiB'lık birimlere ulaşır. Apple'ın eski HFS+ sistemi 2017'de yerini APFS'e, Linux'ta ext3 de ext4'e bıraktı.

10.7 Dizinlerin gerçeklenmesi

10.7.1 Dizin girdileri ve uzun adlar

Bir dosya açılırken dizin girdisi, dosyanın verisini bulmaya yetecek bilgiyi vermelidir. Bu bilgi tahsis yöntemine göre değişir. Bitişik tahsiste ilk bloğun adresi, bağlı liste ve FAT'ta ilk bloğun numarası, UNIX'te ise i-node numarası yeterlidir. Öznitelikler nerede durduğu da iki yolla çözülür (Şekil

10.33). (a) MS-DOS'ta öznitelikler ve disk adresi doğrudan dizin girdisinin içindedir. (b) UNIX'te dizin girdisi yalnızca ad ve i-node numarasıdır; öznitelikler i-node'dadır. İkinci düzen katı bağları mümkün kılar: aynı i-node'u gösteren iki girdi aynı öznitelikleri paylaşır.



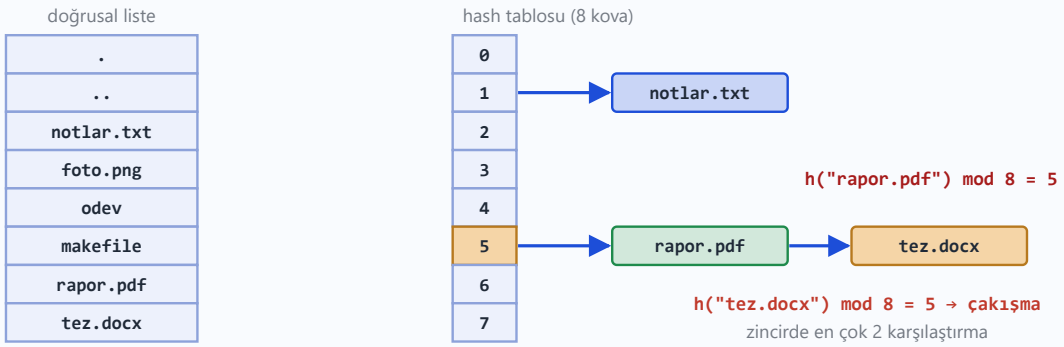
Şekil 10.33: Dizin girdisi biçimleri: (a) öznitelikler girdide, (b) öznitelikler i-node'da, (c) değişken uzunluklu girdiler, (d) sabit girdiler ve ad heap'i.

Uzun adlar ayrı bir sorundur. Her girdiye 255 karakterlik sabit bir ad alanı ayırmak çok yer harcar, çünkü adların çoğu kısadır. İki çözüm vardır. (c) **Değişken uzunluklu girdilerde** her girdi kendi uzunluğunu taşır, ad da girdinin içinde durur. ext4'ün girdisi böyledir: i-node, girdi uzunluğu (`rec_len`), ad uzunluğu, tür ve ad. Bir dosya silinince girdinin yerinde bir delik kalır. Yeni bir girdi bu deliğe sığmayabilir. Bir girdi sayfa sınırına denk gelirse okurken bir sayfa hatası daha yaşanabilir. (d) İkinci çözümde girdiler sabit uzunluklu ve hizalıdır. Her girdi adın kendisi yerine bir ad işaretçisi taşır; adlar dizin bloğunun sonundaki bir **heap'te** durur. Girdileri taramak kolaylaşır, ama heap ayrıca yönetilmelidir.

10.7.2 Dizinde arama

Yol çözümlemenin her adımında bir dizinde bir ad aranır. Aramanın hızı dizinin nasıl tutulduğuna bağlıdır (Şekil 10.34).

Doğrusal liste en basit yapıdır: girdiler art arda durur ve arama baştan sona tarar. Şekil 10.34'teki listede `rapor.pdf`'i bulmak için `.`, `..`, `notlar.txt`, `foto.png`, `odev`, `makefile` ve `rapor.pdf` ile 7 karşılaştırma yapılır. Arama O(n)'dir. Yeni bir ad eklenmeden önce aynı adın olmadığı denetlenmelidir, bu da bir tam tarama demektir. Silmede girdi boş işaretlenir ya da son girdi silinenin yerine taşınır.



Şekil 10.34: Doğrusal liste ile 8 kovalı hash tablosu. `rapor.pdf` ve `tez.docx` aynı kovaya (5) düşer; çakışma zincirle çözülür.

Hash tablosu adın özetinden bir kova seçer: h("rapor.pdf") mod 8 = 5. Arama ortalama O(1)'dir. İki ad aynı kovaya düşebilir; şekilde `tez.docx` da 5'e düşer. Bu **çakışma** kovadan çıkan bir zincirle çözülür ve zincirde en çok 2 karşılaştırma yapılır. Sabit kova sayısı ise büyüyen dizinde sorun olur: 100.000 girdili bir dizinde 8 kovanın her zinciri ortalama 12.500 girdi uzunluğundadır ve arama yeniden doğrusala döner. Hash ayrıca sırayı korumaz; `ls` alfabetik çıktı için girdileri ayrıca sıralamak zorundadır.

Modern sistemler bu yüzden ağaç kullanır. ext4 büyük dizinlerde ad özetine göre sıralı bir **htree** kurar. NTFS, XFS ve Btrfs **B+ ağacı** kullanır. B+ ağacında arama O(log n)'dir ve girdiler sıralı listelenebilir; iki üstünlük bir arada elde edilir. Linux en son kullanılan ad → i-node eşlemelerini ayrıca bellekteki dentry önbelleğinde tutar; sık açılan yollar diske hiç inmeden çözülür.

10.8 Boş alan yönetimi

10.8.1 Blok boyutu seçimi

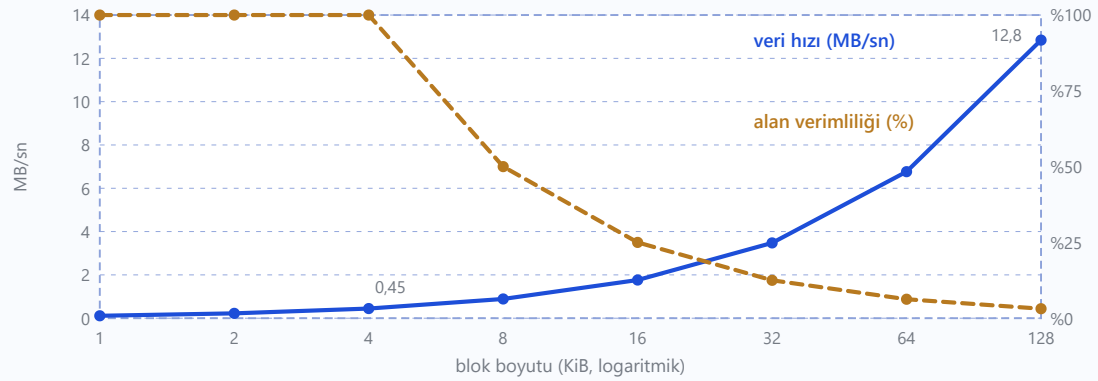
Disk sabit boyutlu bloklara bölünür; blok boyutunun seçimi iki hedef arasında bir ödünleşimdir. Büyük blok bir aramada çok veri getirir ve veri hızını artırır. Küçük dosyalarda ise bloğun büyük kısmı boş kalır ve alan boşa gider. Küçük blok tersini yapar.

Tanenbaum'un modeli bu ödünleşimi sayılarla gösterir (Şekil 10.35). Diskin ortalama arama süresi 5 ms, dönüş süresi 8,33 ms'dir; bir izde 1 MB vardır. Bir bloğu okuma süresi, arama, yarım dönüşlük ortalama bekleme ve aktarma süresinin toplamıdır:

> süre = 5 + 4,165 + (b / 1 MB) × 8,33 ms

Bütün dosyalar 4 KiB olsun. 4 KiB'lık bloklarla bir okuma 9,198 ms sürer: veri hızı 4096 B / 9,198 ms ≈ 0,45 MB/sn. 128 KiB'lık bloklarla okuma 10,206 ms sürer, ama 131.072 bayt getirir: ≈ 12,8 MB/sn. Veri hızı yaklaşık 29 kat artmıştır. Öte taraftan 4 KiB'lık bir dosya 128 KiB'lık bir bloğun yalnızca

4/128 = %3,1'ini doldurur. Blok ≤ 4 KiB iken alan verimliliği %100'dür, 8 KiB'ta %50'ye, 128 KiB'ta %3,1'e düşer.



Şekil 10.35: Blok boyutu büyüdükçe veri hızı artar, 4 KiB'lık dosyalar için alan verimliliği düşer (Tanenbaum modeli; Python ile hesaplandı).

Blok	Okuma süresi (ms)	Veri hızı (MB/sn)	Alan verimliliği
1 KiB	9,173	0,11	%100
4 KiB	9,198	0,45	%100
16 KiB	9,295	1,76	%25
64 KiB	9,686	6,77	%6,25
128 KiB	10,206	12,84	%3,1

Tablo 10.15: Blok boyutu modeli (arama 5 ms, dönüş 8,33 ms, iz 1 MB = 2^{20} B, dosyalar 4 KiB).

Gerçek dosya boyutları üzerine yapılan ölçümler ilginç bir tablo çizer. Dosyaların çoğu küçüktür, ama disk alanının çoğunu az sayıdaki büyük dosya kaplar. Bugün varsayılan blok boyu sayfa boyuyla aynı olan 4 KiB'tır. Büyük dosyalar ise extent'ler ve küme seçenekleriyle desteklenir: blok küçük kalır, ama büyük dosyanın blokları bitişik ayrılır.

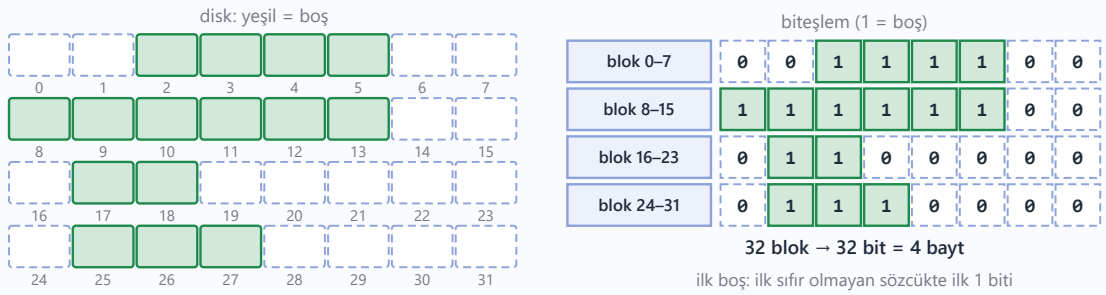
10.8.2 Dört yöntem

Dosya sistemi hangi blokların boş olduğunu bilmelidir. Silberschatz'ın örneğinde 32 bloklu diskin 15 bloğu boştur: 2–5, 8–13, 17–18 ve 25–27. Aynı boş blok kümesi dört yöntemle tutulabilir.

Tanım 10.12

Biteşlem (*bitmap*), her blok için bir bit tutan dizidir. Bu notta ve sunumdaki örnekte 1 = boş, 0 = dolu kabul edilir; bazı sistemler tersini kullanır.

Biteşlemde 32 blok 32 bit, yani 4 bayt tutar (Şekil 10.36). Örneğin biteşlemi `00111100 11111100 01100000 01110000`'dır. Biteşlemin boyutu sabittir ve küçüktür. Bitişik boş blok aramak kolaydır: art arda 1'ler aranır. İşlemciler bir sözcükteki ilk 1 bitini tek komutla bulabildiği için ilk boş bloğu bulmak hızlıdır: ilk sıfır olmayan sözcük bulunur, içindeki ilk 1 biti alınır. Biteşlemin hızlı olması için bellekte tutulması gerekir.

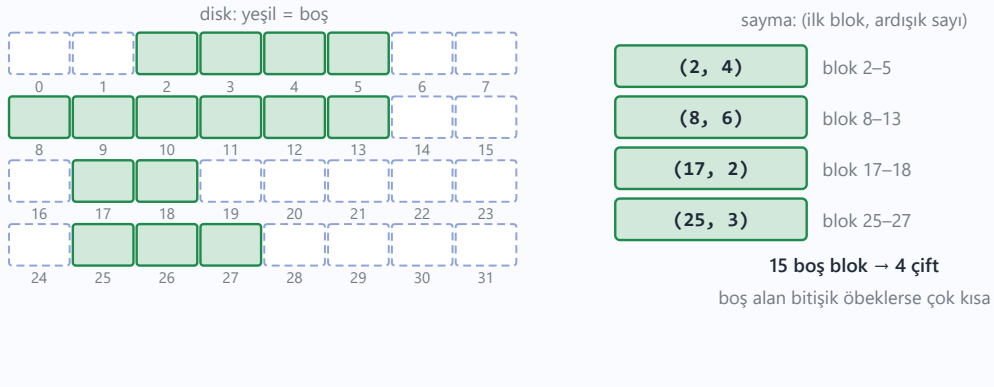


Şekil 10.36: 32 bloklu diskin biteşlemi (1 = boş): 15 boş blok, 32 bit = 4 bayt.

Bağlı listede boş bloklar birbirine bağlanır. Bir baş işaretçisi ilk boş bloğu (2) gösterir; her boş blok bir sonrakinin numarasını taşır: 2 → 3 → 4 → 5 → 8 → … → 13 → 17 → 18 → 25 → 26 → 27 → −1. Ek alan gerekmez, çünkü bilgi boş blokların içindedir. Tek bir boş blok almak ucuzdur: listenin başı verilir. Ama çok sayıda ya da bitişik boş blok bulmak pahalıdır; listede gezinmek her blok için bir G/Ç demektir.

Gruplama bağlı listeyi iyileştirir. İlk boş blok, sonraki n − 1 boş bloğun adresini ve bir sonraki grubun bloğunu tutar. n = 4 ile örnekte gruplar şöyledir: blok 2 → {3, 4, 5}, sonraki grup 8; blok 8 → {9, 10, 11}, sonraki 12; blok 12 → {13, 17, 18}, sonraki 25; blok 25 → {26, 27}, son. Tek bir okumayla n − 1 boş blok bulunur. Klasik UNIX'in boş liste blokları böyle çalışırdı.

Sayma yöntemi boş blokların çoğu zaman bitişik öbekler hâlinde bulunmasından yararlanır. Her öbek (ilk blok, ardışık blok sayısı) çiftiyle tutulur (Şekil 10.37). Örnekteki 15 boş blok yalnızca 4 çifttir: (2, 4), (8, 6), (17, 2), (25, 3). Bu, extent fikrinin boş alana uygulanmasıdır. XFS sayma çiftlerini biri başlangıca, biri uzunluğa göre sıralı iki B+ ağacında tutar; böylece hem "şu bloğun yakınında boş yer" hem de "en az şu büyüklükte boş öbek" soruları hızla yanıtlanır.



Şekil 10.37: Sayma: 15 boş blok 4 (başlangıç, uzunluk) çiftiyle anlatılır.

Örnek 10.11

32 bloklu bir diskte boş bloklar 3–6, 10–11, 20–24 ve 30'dur. Sayma yöntemiyle ve biteşlemle (1 = boş, blok 0 en solda) nasıl gösterilir?

1. **Sayma:** her öbeğin başlangıcı ve uzunluğu: (3, 4), (10, 2), (20, 5), (30, 1).
2. **Biteşlem:** blok 0–7 için 00011110, 8–15 için 00110000, 16–23 için 00001111, 24–31 için 10000010.

Biteşlem disk ne kadar dolu olursa olsun 32 bittir. Sayma listesinin uzunluğu ise öbek sayısına bağlıdır; disk doluyken ya da boş alan büyük öbeklerdeyken çok kısadır.

10.8.3 Kod izleme: biteşlemle blok ayırma

Aşağıdaki fonksiyon N = 16 bloklu bir diskin biteşleminde boş blok arar. Bu kodda biteşlemin anlamı terstir: `bitesl[i] == 1` blok doludur, 0 boştur. Arama `ipucu` numaralı bloktan başlar ve dizinin sonuna gelince başa sarar.

```
int blok_ayir(int ipucu) {
  for (int k = 0; k < N; k++) {
    int i = (ipucu + k) % N;
    if (bitesl[i] == 0) {  /* boş mu? */
      bitesl[i] = 1;
      return i;
    }
  }
  return -1;               /* disk dolu */
}
```

`i = (ipucu + k) % N`, `ipucu`'dan başlayıp k adım ilerideki konumdur; mod işlemi aramanın 15'ten sonra 0'a dönmesini sağlar. İlk boş blok bulunduğunda biti 1 yapılır ve numarası döndürülür. Bütün bloklar doluysa döngü N turdan sonra biter ve −1 döner. En kötü durumda süre O(N)'dir. `ipucu` parametresinin amacı yerelliktir: dosyanın yeni bloğu, son bloğunun hemen arkasından aranırsa bloklar birbirine yakın düşer ve sıralı okumada kafa hareketi azalır.

Başlangıçta biteşlem 1 1 0 1 1 1 0 0 1 1 1 0 1 0 1 1'dir (blok 0'dan 15'e). Boş bloklar 2, 6, 7, 11 ve 13'tür. Bir dosya blok 3'te bitiyor ve 3 blok daha istiyor. Her çağrının ipucu, dosyanın o anki son bloğunun bir fazlasıdır. Tablo 10.16 izlemeyi verir.

Çağrı	Bakılan konumlar (i)	Dönüş	Dosyanın blokları
`blok_ayir(4)`	4 (dolu), 5 (dolu), 6 (boş)	6	…, 3, 6

Çağrı	Bakılan konumlar (i)	Dönüş	Dosyanın blokları
`blok_ayir(7)`	7 (boş)	7	..., 3, 6, 7
`blok_ayir(8)`	8, 9, 10 (dolu), 11 (boş)	11	..., 3, 6, 7, 11

Tablo 10.16: `blok_ayir` izlemesi (Python'a birebir çevrilerek çalıştırıldı).



Şekil 10.38: Üç çağrıdan sonra biteşlem: dosyaya 6, 7 ve 11 ayrıldı.

Dosya 6, 7 ve 11 bloklarını aldı. Bloklar bitişik değildir ama birbirine yakındır. Arama ipucusuz, hep 0'dan başlasaydı ilk çağrı blok 2'yi verirdi; dosya diskin başına geri dönerdi. İpucu sona yakınsa başa sarma devreye girer: aynı başlangıç biteşleminde `blok_ayir(14)` 14 ve 15'i dolu bulur, 0 ve 1'i de geçer ve blok 2'yi döndürür.

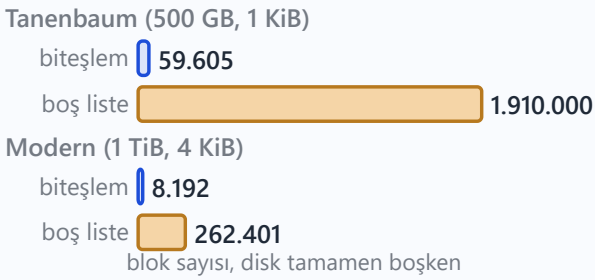
10.8.4 Sayısal karşılaştırma

Biteşlem ile boş blok listesinin ne kadar yer tuttuğu, büyük bir disk üzerinde hesaplanabilir (Tablo 10.17). Listede her blok numarası 32 bittir. 1 KiB'lık bir liste bloğu 256 numara tutar: 255 boş blok adresi ve sonraki liste bloğuna 1 bağ.

	Tanenbaum örneği	Modern örnek
Disk / blok	500 GB, 1 KiB blok	1 TiB, 4 KiB blok
Blok sayısı	$500 \times 10^9 / 1024 \approx 488{,}3$ milyon	$2^{40} / 2^{12} = 268.435.456$
Liste bloğu	1 KiB / 4 B = 256 → 255 adres + 1 bağ	4096 / 4 = 1024 → 1023 adres + 1 bağ
Liste, disk boşken	488,3 M / 255 ≈ 1,91 milyon blok	268,4 M / 1023 ≈ 262.401 blok (≈ 1 GiB)
Biteşlem	488,3 M bit / 8192 ≈ 59.605 blok	2^{28} bit = 32 MiB = 8192 blok

Tablo 10.17: Boş alan kaydının boyutu (disk tamamen boşken; Python ile hesaplandı).

Tanenbaum örneğinde blok sayısı $500 \times 10^9 / 1024 = 488.281.250$'dir. Disk tamamen boşken liste $488.281.250 / 255 \approx 1.914.828$ blok tutar. Biteşlem ise blok başına bir bit demektir: 1 KiB'lık blok 8192 bit taşır ve $488.281.250 / 8192 \approx 59.605$ blok yeter. Biteşlem listeden yaklaşık 32 kat küçüktür. Liste yalnızca disk neredeyse doluyken küçüktür, çünkü yalnız boş blokları listeler. Biteşlemin boyutu ise disk doluluğundan bağımsızdır.



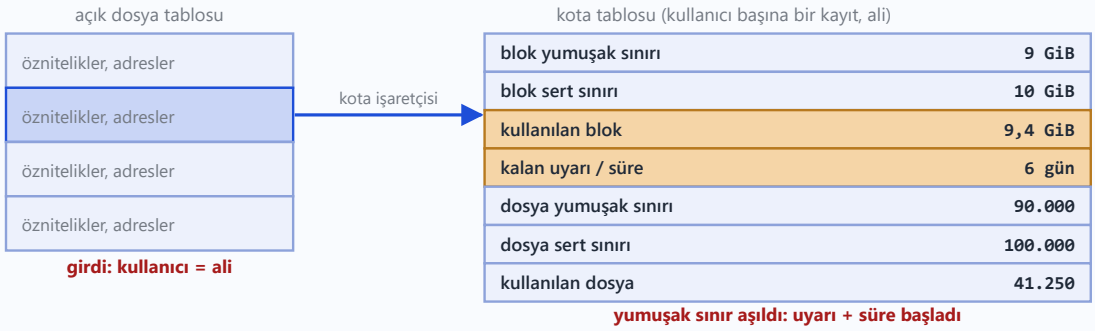
Şekil 10.39: Disk boşken biteşlem ve boş listenin blok sayısı.

Listenin bir üstünlüğü de vardır: bellekte yalnızca tek bir liste bloğu tutmak yeterlidir. Blok dolunca diske yazılır, boşalınca yenisi okunur. Bu basit politikanın bir tuzağı vardır. Bellekteki blok tam doluyken üç bloklu geçici bir dosya silinirse blok taşar ve diske yazılır. Hemen ardından aynı dosya yeniden oluşturulursa blok boşalır ve yenisi diskten okunur. Geçici dosyalar böylece sürekli disk G/Ç'sine yol açar. Tanenbaum'un önerdiği çözüm bellekteki işaretçi bloğunu yarı dolu tutmaktır; hem ayırma hem serbest bırakma disk G/Ç'si olmadan bir süre sürer.

Gerçek sistemler bu yöntemlerin birleşimlerini kullanır. ext4 her blok grubu için ayrı bir biteşlem tutar. XFS sayma yöntemini iki B+ ağacıyla uygular. ZFS ayırma ve serbest bırakma kayıtlarını bir günlük gibi biriktiren **alan haritası** (*space map*) kullanır. Btrfs boş ve dolu alanı extent ağacında izler.

10.8.5 Disk kotaları

Çok kullanıcılı bir sistemde tek bir kullanıcı bütün diski doldurmamalıdır. **Kota** (*quota*) her kullanıcının kullanabileceği blok ve dosya sayısını sınırlar (Şekil 10.40). Bir dosya açılınca açık dosya tablosundaki girdisi, sahibinin kota kaydını gösterir. Her yeni blok ya da i-node tahsisinde kullanıcının sayaçları artar ve sınırlarla karşılaştırılır.



Şekil 10.40: Açık dosya girdisi kullanıcının kota kaydını gösterir. ali'nin 9,4 GiB'lık kullanımı 9 GiB'lık yumuşak sınırı aşmıştır; uyarı verildi ve süre başladı.

Her sınırın iki düzeyi vardır. **Yumuşak sınır** aşılabilir; kullanıcı uyarılır ve bir süre (ör. 7 gün) başlar. Kullanıcı bu süre içinde dosya silerek sınırın altına inmelidir. **Sert sınır** hiçbir koşulda aşılamaz. Süre biterse ya da sert sınıra ulaşılırsa yazma `EDQUOT` hatasıyla reddedilir. Şekildeki ali 9,4 GiB kullanmış, 9 GiB'lık yumuşak sınırı aşmıştır; 6 günü kalmıştır ve 10 GiB'lık sert sınıra kadar yazabilir. Dosya sayısında ise 41.250 dosyayla 90.000'lik yumuşak sınırın çok altındadır. Bugün Linux'ta quota araçları (ext4 ve XFS proje kotaları), NTFS'te birim kotaları, ZFS'te veri kümesi kotaları ve Btrfs'te alt birim (qgroup) kotaları vardır.

10.9 Performans

10.9.1 Bir dosyayı oluşturmak kaç G/Ç'dir?

Dosya sistemi çağrılarının diske ne kadar iş yüklediğini görmek için önbelleği olmayan basit bir dosya sistemi düşünelim. i-node ve biteşlem kullanılsın, her dizin tek bir bloğa sığsın. Kök dizinin i-node'unun yeri süperbloktan biliniyor olsun. Arpaci-Dusseau'ların OSTEP kitabındaki bu tür sayım, çağrıların arkasındaki gizli maliyeti ortaya çıkarır. Tablo 10.18 dört işlemin G/Ç'lerini sıralar.

Çağrı	G/Ç sırası	Toplam
`open("/ders/not.txt")`, dosya var	oku kök i-node, kök veri, ders i-node, ders veri, not i-node	5 = 5 okuma
`read(fd, buf, 4096)`	oku not i-node, not veri[0]; yaz not i-node (erişim zamanı)	3 = 2 okuma + 1 yazma
`creat("/ders/not.txt")`, dosya yok	oku kök i-node, kök veri, ders i-node, ders veri, i-biteşlem; yaz i-biteşlem, ders veri; oku not i-node; yaz not i-node, ders i-node	10 = 6 okuma + 4 yazma
`write(fd, buf, 4096)`, yeni blok	oku not i-node, v-biteşlem; yaz v-biteşlem, not veri, not i-node	5 = 2 okuma + 3 yazma

Tablo 10.18: Önbelleksiz bir dosya sisteminde çağrıların disk G/Ç'leri.

Açmak yol uzunluğuyla orantılıdır: her dizin için bir i-node ve bir veri bloğu, sonda dosyanın i-node'u. Okuma i-node'dan blok adresini bulur, bloğu okur ve i-node'daki erişim zamanını günceller. Son yazma, `noatime` ya da `relatime` bağlama seçenekleriyle çoğu zaman ortadan kalkar.

Oluşturma çok daha pahalıdır. Önce yol çözülür (4 okuma). Sonra i-node biteşleminden boş bir i-node seçilir ve işaretlenir (1 okuma, 1 yazma). Dizine yeni girdi eklenir (1 yazma). Yeni i-node'u içeren blok okunur, i-node ilklendirilir ve yazılır (1 okuma, 1 yazma). Son olarak dizinin i-node'unda boyut ve zaman güncellenir (1 yazma). Toplam 10 G/Ç'dir. Yeni blok isteyen her `write` da 5 G/Ç tutar: i-node okunur, veri biteşleminden blok seçilip işaretlenir, veri yazılır ve i-node'a yeni adres ile boyut yazılır. Bu 5 G/Ç'nin yalnızca biri asıl veridir.



Şekil 10.41: `creat` ve üç 4 KiB'lık `write`: 10 + 3 × 5 = 25 G/Ç. Kalın çerçeveli üç kutu kullanıcı verisidir; kalan 22 G/Ç üst veridir.

Önbelleksiz bir dosyayı oluşturup 12 KiB yazmak böylece 10 + 3 × 5 = 25 G/Ç eder (Şekil 10.41). Bunların yalnızca üçü kullanıcının verisidir. Önbellek bu sayıyı dramatik biçimde azaltır. Okumalar bellekten karşılanır. Gecikmeli yazma da aynı bloğa yapılan yazmaları birleştirir: i-node biteşlemi, veri biteşlemi, dizin verisi, dizin i-node'u, dosyanın i-node'u ve üç veri bloğu birer kez yazılır. 25 G/Ç'den 8 yazma kalır. Bu sayım, tampon önbelleğinin ve gecikmeli yazmanın neden vazgeçilmez olduğunu gösterir. Yazmaların hangi sırayla yapıldığı ise çökme tutarlılığını belirler.

10.9.2 Tampon önbelleği

Tanım 10.13

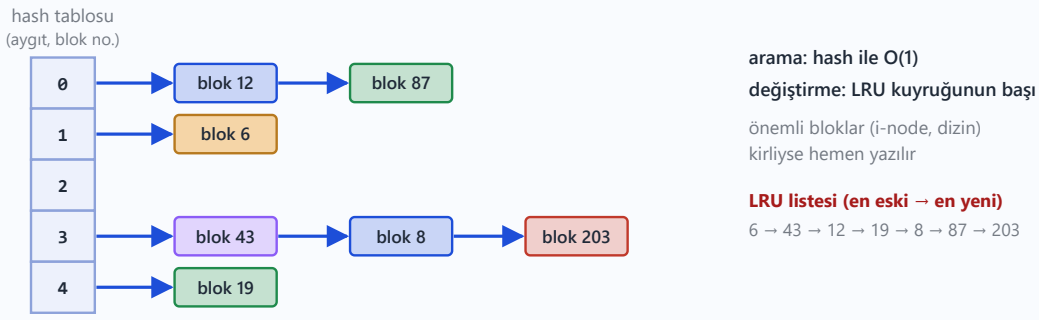
Tampon önbelleği (*buffer cache*), disk bloklarının kopyalarını bellekte tutan çekirdek önbelleğidir. İstenen blok önbellekteyse **isabet**, değilse **ıska** olur.

Önbelleğin değeri, bellek ile disk arasındaki büyük hız farkından gelir (Tablo 10.19). Bir NVMe SSD'den rastgele 4 KiB okumak DRAM erişiminin yaklaşık bin katı sürer; sabit diskte arama ve dönüş bir o kadar daha yavaştır.

Erişim	Gecikme
DRAM'den 64 bayt	≈ 80–120 ns
NVMe SSD 4 KiB rastgele okuma	≈ 50–100 µs
HDD arama + dönüş	≈ 5–10 ms (100–280 MB/sn sıralı)

Tablo 10.19: Tipik erişim gecikmeleri (2026).

Klasik tampon önbelleği iki yapıdan oluşur (Şekil 10.42). Bir **hash tablosu**, (aygıt, blok numarası) çiftinden tamponu O(1) sürede bulur; aynı kovaya düşen tamponlar zincirlenir. Bir **LRU listesi** de tamponları son kullanım sırasına dizer. Yer gerektiğinde listenin başındaki, en uzun süre kullanılmamış tampon seçilir. i-node ve dizin blokları gibi önemli bloklar değiştirildiğinde beklemeden yazılabilir; bunlar bozulursa bütün dosya sistemi zarar görür.



Şekil 10.42: Tampon önbelleği: hash tablosu zincirleri ve LRU listesi (en eski → en yeni: 6, 43, 12, 19, 8, 87, 203).

Yazmalar için iki politika vardır. **Anında yazma** (*write-through*) her değişikliği hemen diske gönderir. Güvenlidir ama yavaştır; eski MS-DOS ve çıkarılabilir ortamlar böyle çalışır, çünkü kullanıcı USB belleği her an çekebilir. **Geri yazma** (*write-back*) değişen, yani **kirli** blokları bellekte biriktirir ve sonra yazar. Linux 30 saniyeden eski kirli sayfaları arka planda diske yazar. Uygulama verinin diske ulaşmasını beklemek isterse `fsync` çağırır.

> **Sık yapılan hata**
>
> `write` başarıyla döndü diye verinin diskte olduğunu sanmak. Geri yazmalı önbellekte `write` veriyi yalnızca sayfa önbelleğine kopyalar. Birkaç saniye içinde elektrik kesilirse veri kaybolur. `close` da diske yazmayı garanti etmez. Kalıcılık için `fsync(fd)` ya da `O_SYNC` gerekir.

10.9.3 Kod izleme: bread

Aşağıdaki `bread` fonksiyonu bir bloğu önbellek üzerinden okur:

```
Tampon *bread(int blok) {
  Tampon *t = hash_ara(blok);
  if (t != NULL) {          /* isabet */
    lru_sona(t);
    return t;
  }
  t = lru_bas();            /* en eski */
  if (t->kirli)
    diske_yaz(t);         /* geri yaz */
  t->blok = blok; t->kirli = 0;
  diskten_oku(t);           /* ıska: G/Ç */
  lru_sona(t);
  return t;
}
```

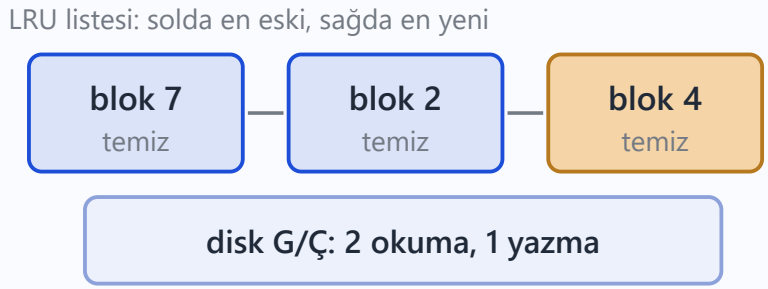
Fonksiyon önce bloğu hash tablosunda arar. Bulursa (isabet) tamponu LRU listesinin sonuna, en yeni konuma taşır ve döndürür; disk G/Ç'si yoktur. Bulamazsa (ıska) listenin başındaki en eski tamponu alır. Tampon kirliyse içeriği kaybolmasın diye önce diske yazılır. Sonra tampon yeni bloğa atanır, kirli biti temizlenir, içerik diskten okunur ve tampon listenin sonuna eklenir.

Önbellekte 3 tampon olsun: LRU sırasıyla bloklar 5, 9 ve 2. Blok 9 kirlidir, yani değiştirilmiş ama henüz diske yazılmamıştır. İstekler sırasıyla 9, 7, 2 ve 4'tür. Tablo 10.20 izlemeyi verir.

İstek	Sonuç	Atılan	LRU sonra (eski → yeni)	Okuma / yazma
9	isabet	—	5, 2, 9*	0 / 0
7	ıska	5 (temiz)	2, 9*, 7	1 / 0
2	isabet	—	9*, 7, 2	1 / 0
4	ıska	9 (kirli → yazılır)	7, 2, 4	2 / 1

Tablo 10.20: `bread` izlemesi (başlangıç LRU: 5, 9*, 2; * kirli; Python ile benzetildi).

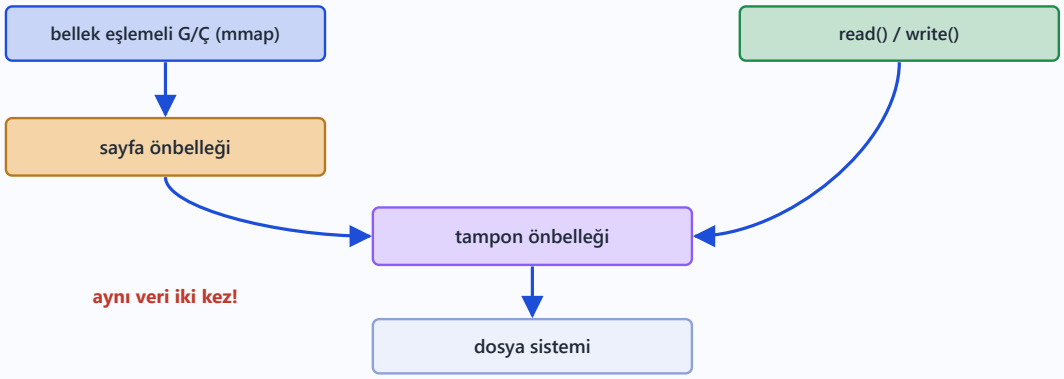
İlk istek bir isabettir; blok 9 en yeni konuma taşınır ve en eski blok artık 5'tir. `bread(7)` ıskadır. Blok 5 temizdir: diskteki kopyası güncel olduğu için geri yazmaya gerek yoktur. Tampon blok 7 için yeniden kullanılır ve 1 okuma yapılır. `bread(2)` yine isabettir. Bu adımdan sonra kirli blok 9 listenin başına gelmiştir. `bread(4)` ıskadır ve en eski tampon 9'u tutar. Değişiklik kaybolmasın diye blok 9 önce diske yazılır, sonra blok 4 okunur. Kirli bir tamponu atmak ıskanın maliyetini ikiye katlar. Sonuçta 4 istekte 2 isabet, toplam 2 okuma ve 1 yazma yapılmıştır. Önbellek olmasaydı 4 okuma ve blok 9'un değişikliği için ayrı bir yazma gerekirdi.



Şekil 10.43: Dört istekten sonra önbellek: 7, 2, 4; toplam 2 okuma ve 1 yazma.

10.9.4 Sayfa önbelleği ve birleşik önbellek

Dosyalara iki yoldan erişilir: `read` / `write` çağrılarıyla ve `mmap` ile bellek eşlemeli G/Ç'yle. Eski sistemlerde bu iki yol ayrı önbellekler kullanırdı (Şekil 10.44). `mmap`'in verisi sanal bellek sisteminin **sayfa önbelleğinde**, `read` / `write`'ın verisi tampon önbelleğinde dururdu. Aynı dosya iki yoldan birden kullanılınca aynı veri bellekte iki kez bulunurdu. Bu **çift önbellekleme** hem belleği israf ederdi hem de iki kopyanın tutarlı kalmasını zorlaştırırdı.



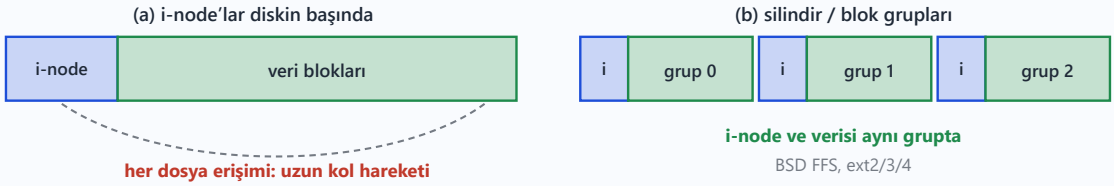
Şekil 10.44: Birleşik olmayan önbellek: `mmap` ve `read` / `write` verisi iki ayrı önbellekte, aynı veri iki kez.

Birleşik tampon önbelleğinde dosya sayfaları tek bir yerde durur ve iki yol da aynı sayfaları görür. Linux 2.4'ten beri böyledir; Windows'un önbellek yöneticisi ve Solaris de aynı yolu izler. Sayfa önbelleği sanal bellekle aynı sayfa yer değiştirme algoritmalarını paylaşır: bir dosya sayfası ile bir sürecin anonim sayfası aynı havuzda yarışır. İstisnalar da vardır. ZFS kendi ARC önbelleğini tutar. Kendi önbelleğini yöneten veritabanları ise `O_DIRECT` bayrağıyla işletim sisteminin önbelleğini tamamen atlar.

10.9.5 Önden okuma, yerleşim ve birleştirme

Önden okuma (*read-ahead*), sıralı erişimi önceden sezmektir. k. blok istendiğinde k + 1. blok da (önbellekte değilse) okunur; program onu istediğinde hazırdır. Önden okuma yalnızca sıralı erişimde işe yarar. Rastgele erişimde gereksiz okumalar diski meşgul eder. Bu yüzden dosya başına bir "bu dosya sıralı mı okunuyor?" izi tutulur. Linux pencereyi erişim kalıbına göre büyütür ya da küçültür; varsayılan pencere 128 KiB'tır. Uygulama da `posix_fadvise` ile ipucu verebilir.

Sabit diskte en pahalı iş kol hareketidir. Ardışık okunacak blokları diskte birbirine yakın koymak bu hareketi azaltır. Biteşlemle bitişik boş blok bulmak kolaydır. Eski bir öneri blokları ikişer ikişer ayırmaktı. Bugün extent'ler, gecikmeli tahsis ve çoklu blok ayırıcı bu işi yapar. i-node'ların yeri de önemlidir (Şekil 10.45). (a) i-node'lar diskin başında toplanırsa her dosya erişimi i-node ile veri arasında uzun bir kol hareketi ister. (b) Disk silindir ya da blok gruplarına bölünür ve her grubun kendi i-node'ları olursa i-node ve verisi aynı grupta durur. BSD FFS ve ext2/3/4 böyledir.



Şekil 10.45: i-node yerleşimi: (a) diskin başında, (b) her silindir ya da blok grubunda.

Zamanla dosyalar parçalanır: blokları diske dağılır. **Birleştirme** (*defragmentation*) dosyaları yeniden bitişik hâle getirir. Windows'ta bu aracın adı "Sürücüleri birleştir ve iyileştir"dir. Linux'ta da benzer araçlar vardır: `e4defrag`, `xfs_fsr`, `btrfs filesystem defragment`. ext4'ün tahsis politikası bu ihtiyacı büyük ölçüde azaltır. SSD'de arama süresi olmadığı için birleştirmenin kazancı yoktur; tersine gereksiz yazma, yani aşınma demektir. Windows SSD'de birleştirme yerine TRIM gönderir.

10.9.6 SSD: FTL ve TRIM

Flaş belleğin bir sayfası yerinde güncellenemez; yazılmadan önce ait olduğu büyük silme bloğunun bütünüyle silinmesi gerekir. SSD'nin denetleyicisindeki **flaş çeviri katmanı** (*flash translation layer*, FTL) bu yüzden her yazmayı yeni, boş bir sayfaya yönlendirir ve mantıksal blok adresini o sayfaya eşler. Eski sayfa geçersiz olur. SSD'nin içi böylece günlük yapılı bir sisteme benzer. Boş sayfa azalınca denetleyici **çöp toplama** yapar: bir silme bloğundaki geçerli sayfaları başka yere kopyalar ve bloğu siler.

Burada bir bilgi eksikliği doğar. Dosya sistemi bir dosyayı silince blokları yalnızca kendi biteşleminde boş işaretler. SSD bundan habersizdir; o sayfaları hâlâ geçerli sanar ve çöp toplamada boşuna kopyalar. Gereksiz kopyalar **yazma büyütmesini** artırır, hem hızı düşürür hem aşınmayı hızlandırır. Çözüm, dosya sisteminin boşalan blokları SSD'ye **TRIM** komutuyla (NVMe'de Deallocate) bildirmesidir. Linux'ta bildirim ya haftalık `fstrim.timer` ile toplu olarak ya da `discard` bağlama seçeneğiyle her silmede yapılır.

10.10 Tutarlılık ve kurtarma

10.10.1 Yedekleme

Yedeklemenin iki amacı vardır. Birincisi **felaketten kurtarmadır**: disk arızası, yangın, fidye yazılımı. İkincisi **dikkatsizlikten kurtarmadır**: yanlışlıkla silinen ya da üzerine yazılan bir dosyayı geri getirmek. Pratikte ikincisi çok daha sık gerekir.

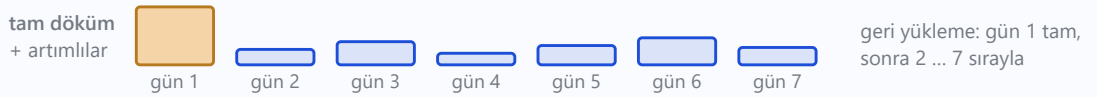
Yedeklemenin iki temel biçimi vardır (Tablo 10.21). **Fiziksel döküm** diski blok 0'dan sona kadar kopyalar. Çok hızlı ve basittir, ama dizin atlayamaz, artımlı yedek alamaz, tek bir dosyayı geri yüklemeyi de zorlaştırır. **Mantıksal döküm** seçilen dizinleri ve dosyaları yollarıyla birlikte kopyalar. Daha yavaş ve karmaşıktır; buna karşılık yalnızca değişen dosyaları alabilir ve tek bir dosyayı başka bir makineye bile kolayca geri yükler.

	Fiziksel döküm	Mantıksal döküm
Ne kopyalanır?	blok 0'dan sona tüm (kullanılan) bloklar	seçilen dizinler ve dosyalar, yollarıyla
Hız, basitlik	çok hızlı, basit	daha yavaş, karmaşık
Dizin atlama, artımlı yedek	yapamaz	yapabilir (değişenler)

	Fiziksel döküm	Mantıksal döküm
Tek dosyayı geri yükleme	zor	kolay; başka makinede de
Bozuk bloklar	denetleyici yeniden eşlemeli ya da İS listeyi bilmeli	sorun değil

Tablo 10.21: Fiziksel ve mantıksal döküm.

Yaygın bir strateji tam ve artımlı dökümü birleştirir (Şekil 10.46). Örneğin ayda bir tam döküm, her gün de yalnızca son dökümden beri değişenler alınır. Geri yüklemede boş dosya sistemine önce son tam döküm yazılır: önce dizinler, sonra dosyalar. Ardından artımlı dökümler sırayla uygulanır. Geçici dosyalar ve özel dosyalar (`/dev` , `/proc` , `/tmp`) yedeklenmez. Sistem kullanılırken alınan bir yedek tutarsız olabilir: dökümün başında kopyalanan bir dizin, sonunda değişmiş olabilir. Bu yüzden önce bir **anlık görüntü** (LVM, ZFS/Btrfs, Windows VSS) alınır ve yedek o donmuş görüntüden çıkarılır.



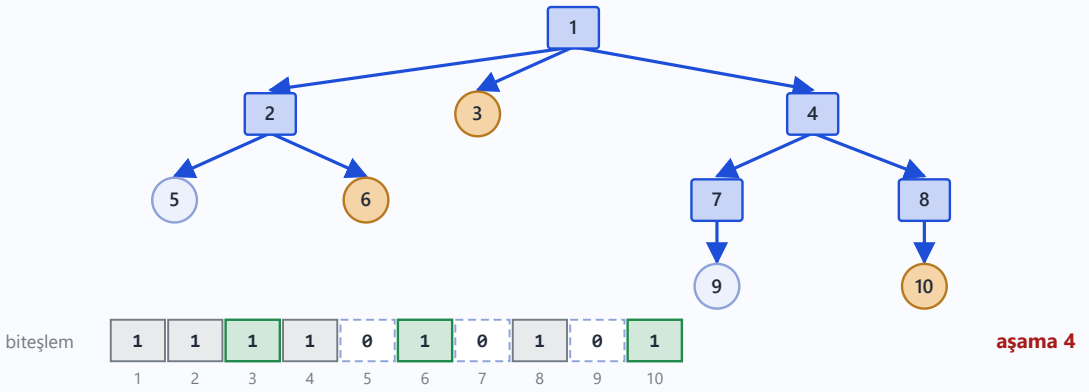
Şekil 10.46: Tam ve artımlı döküm: ilk gün tam döküm, sonraki günler yalnızca değişenler.

Yedeklerin nerede durduğu da önemlidir. **3-2-1 kuralı** şunu söyler: 3 kopya, 2 farklı ortam, 1 kopya başka bir yerde (çevrim dışı ya da değiştirilemez). Fidye yazılımı ağdaki her diski şifreleyebileceği için çevrim dışı kopya son güvencedir. Teyp bugün de arşivde yaşar; LTO-10 kartuşu sıkıştırmasız 40 TB'a kadar veri tutar.

10.10.2 Mantıksal döküm algoritması

UNIX'in artımlı mantıksal döküm algoritması dört aşamada çalışır ve bir i-node biteşlemi kullanır. Şekil 10.47'deki ağaçta kareler dizinleri, daireler dosyaları gösterir; sayılar i-node numaralarıdır. Kök 1'in altında 2, 3 ve 4 vardır. 2'nin altında 5 ve 6, 4'ün altında 7 ve 8, 7'nin altında 9, 8'in altında 10 bulunur. Son dökümden beri değişen dosyalar 3, 6 ve 10'dur.

1. **Aşama 1.** Ağaçta kökten başlayarak gezinilir. Değişen her dosya ve değişmiş olsun olmasın bütün dizinler işaretlenir: 1, 2, 3, 4, 6, 7, 8, 10.
2. **Aşama 2.** Ağaçta yeniden gezinilir. Altında değişen hiçbir şey olmayan dizinlerin işareti kaldırılır: 7, çünkü yalnızca değişmemiş 9'u içerir.
3. **Aşama 3.** İşaretli dizinler i-node sırasıyla dökülür: 1, 2, 4, 8. Bu dizinler değişmemiş olsa da dökülür, çünkü değişen dosyaların yollarını geri kurmak için gereklidir.
4. **Aşama 4.** İşaretli dosyalar dökülür: 3, 6, 10.



Şekil 10.47: Mantıksal dökümün son aşaması: işaretli dosyalar 3, 6, 10 dökülür. Biteşlemde 5, 7 ve 9 işaretsizdir.

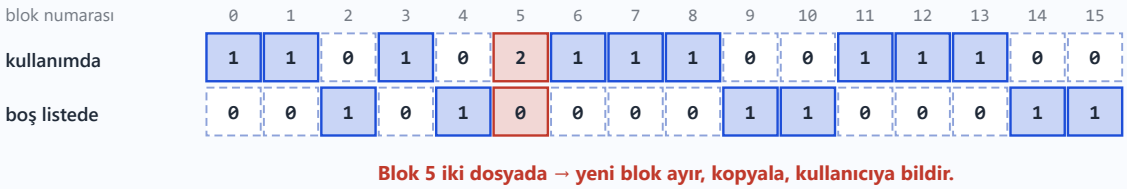
Dizinlerin dosyalardan önce dökülmesi geri yüklemeyi kolaylaştırır: önce dizin ağacı kurulur, sonra dosyalar yerlerine konur. Birkaç ayrıntı daha vardır. Boş liste dökülmez; geri yüklemeden sonra yeniden kurulur. Birden çok katı bağı olan bir dosya yalnızca bir kez dökülmeli, geri yüklemede bütün adları yeniden bağlanmalıdır. Seyrek dosyaların delikleri sıfırlarla doldurulmamalıdır; yoksa geri yüklenen dosya diskte gerçek boyuta şişer. FIFO'lar ve aygıt dosyaları da içerik olarak değil, tür olarak dökülür.

10.10.3 Tutarlılık denetimi: fsck

Bir dosya sistemi çökme sonrasında tutarsız kalabilir: değişikliklerin bir kısmı diske ulaşmış, bir kısmı ulaşmamıştır. UNIX'in `fsck` (Windows'ta `chkdsk`) programı bu tutarsızlıkları bulup onarır. İki tür denetim yapar: blok tutarlılığı ve dosya tutarlılığı.

Blok tutarlılığında fsck her blok için iki sayaç tutar. Bütün i-node'ları okur ve her i-node'un kullandığı blokların "kullanımda" sayacını artırır. Sonra boş listeyi ya da biteşlemi okur ve "boş" sayacını artırır. Tutarlı bir dosya sisteminde her blok için iki sayacın toplamı 1'dir: blok ya bir dosyadadır ya boştur (Şekil 10.48). Üç sorun çıkabilir:

- **Eksik blok** (0, 0): blok hiçbir yerde görünmez. Zararsızdır ama alan kaybıdır; blok boş listeye eklenir.
- **Boş listede yinelenen blok** (0, 2): aynı blok ileride iki dosyaya birden verilebilirdi. Boş liste yeniden kurulur. Biteşlemde bu durum oluşamaz, çünkü bir bit ya 0'dır ya 1.
- **Yinelenen veri bloğu** (2, 0): iki dosya aynı bloğu kullanıyor. En tehlikelisidir: dosyalardan biri değişince öbürü de bozulur. fsck yeni bir blok ayırır, içeriği kopyalar, kopyayı dosyalardan birine verir ve kullanıcıyı bilgilendirir.



Şekil 10.48: fsck sayaçları: blok 5 iki dosyada görünür (yinelenen veri bloğu). Yeni blok ayrılır, içerik kopyalanır, kullanıcıya bildirilir.

Bir blok aynı anda hem kullanımda hem boş listede görünürse (1, 1) boş listeden çıkarılır.

Dosya tutarlılığında fsck kökten başlayarak bütün dizinlerde gezinir ve her i-node için bir görülme sayacı tutar. Katı bağlar yüzünden bir i-node birden çok dizinde görünebilir. Sayaç sonunda i-node'daki bağ sayısıyla karşılaştırılır. Bağ sayısı görülme sayısından büyükse yalnızca alan israfı vardır: bütün adlar silinse bile bağ sayısı 0'a inmez ve dosya hiç serbest kalmaz. Küçükse durum tehlikelidir: adlardan biri silinince bağ sayısı 0'a iner ve dosya, başka bir adı hâlâ varken serbest bırakılır. Her iki durumda bağ sayısı görülme sayısına eşitlenir. Hiçbir dizinde görünmeyen ama kullanımda olan i-node'lar `lost+found` dizinine konur.

Örnek 10.12

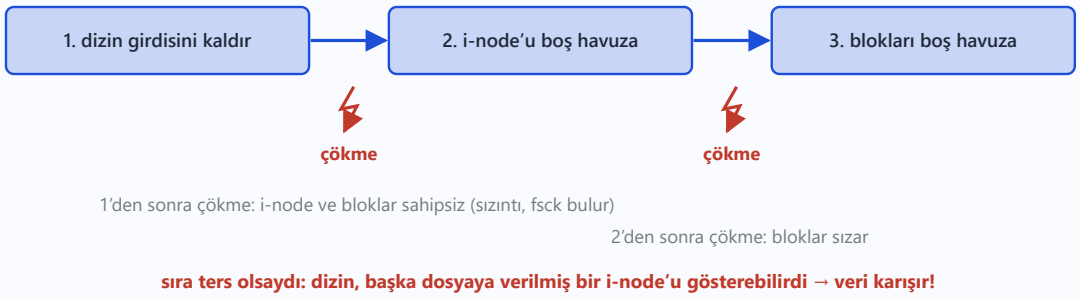
fsck'in sayaçları şöyle çıktı (blok 0–9): kullanımda 1 0 1 1 0 2 1 0 0 1; boş listede 0 1 0 0 1 0 0 0 2 0. Sorunlu bloklar hangileridir?

1. Her blok için toplam 1 olmalı. Blok 5: (2, 0), yinelenen veri bloğu; en tehlikelisi.
2. Blok 7: (0, 0), eksik blok; boş listeye eklenir.
3. Blok 8: (0, 2), boş listede iki kez; liste yeniden kurulur.

fsck'in büyük sorunu süresidir. Bütün i-node'ları ve bütün dizinleri okumak gerekir. Terabaytlık bir diskte tam tarama saatler sürer ve sistem bu sürede kullanılamaz. Günlükleme ve yazınca kopyalama bu yüzden doğdu.

10.10.4 Çökme ve atomiklik

Sorunun kökü şudur: bir dosya işlemi birden çok bloğu değiştirir, ama disk yalnızca tek bir sektörün yazılmasını atomik olarak garanti eder. Bir dosyayı silmek üç adımdır (Şekil 10.49): dizin girdisini kaldırmak, i-node'u boş havuza vermek ve blokları boş havuza vermek. Birinci adımdan sonra çökme olursa i-node ve bloklar sahipsiz kalır; bu bir sızıntıdır, fsck bulur. İkinci adımdan sonra çökme olursa bloklar sızar. Sıra ters olsaydı durum çok daha kötü olurdu. i-node önce serbest bırakılıp başka bir dosyaya verilirse ve dizin girdisi henüz silinmeden çökme olursa dizin yanlış dosyayı gösterir. İki dosyanın verisi karışır.



Şekil 10.49: Dosya silmenin üç adımı ve çökme noktaları.

Bu soruna dört çözüm ailesi vardır. (1) Sonra onarmak: fsck. (2) Önce niyeti yazmak: günlükleme. (3) Hiçbir zaman üzerine yazmamak: yazınca kopyalama ve günlük yapılı dosya sistemleri. (4) Yazma sırasını dikkatle ayarlamak: BSD'nin *soft updates* yöntemi, yazmaları çökme hangi anda olursa olsun yalnızca zararsız sızıntılar bırakacak sırada yapar.

Tanım 10.14

Bir kez de birçok kez de uygulansa aynı sonucu veren işleme **eşgüçlü** (*idempotent*) işlem denir.

Günlükleme eşgüçlülüğe dayanır. Kurtarma sırasında bir işlem birden çok kez yeniden oynatılabilir: çökme, kurtarmanın ortasında bile olabilir. "Blok n'yi boş işaretle" eşgüçlüdür; kaç kez yapılsa blok boştur. "Boş listenin sonuna n'yi ekle" eşgüçlü değildir; iki kez yapılırsa n listede iki kez bulunur. Günlüğe bu yüzden işlemler değil, blokların yeni içerikleri yazılır.

10.10.5 Günlükleme

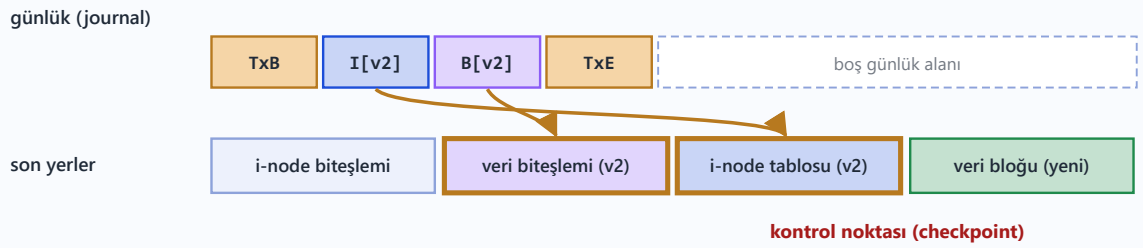
Tanım 10.15

Günlükleme (*journaling*), değişiklikleri önce diskteki ayrı bir alana, **günlüğe**, yazıp ancak sonra son yerlerine uygulamaktır. Veritabanlarında bu fikre önceden yazma günlüğü (*write-ahead logging*) denir.

Bir dosyaya yeni bir blok eklendiğini düşünelim. Üç blok değişir: dosyanın i-node'u (I), veri biteşlemi (B) ve yeni veri bloğu (Db). ext4'ün varsayılan **sıralı** (*ordered*) modunda işlem beş adımda yürür (Şekil 10.50).

1. **Veri.** Yeni veri bloğu Db önce son yerine yazılır.
2. **Günlüğe yazma.** İşlem başı kaydı TxB ve değişecek üst veri blokları, yani I ve B'nin yeni sürümleri, günlüğe sıralı yazılır.

3. **Onaylama.** Önceki yazmaların diske ulaştığı kesinleşince (bariyer ya da FLUSH komutu) işlem sonu kaydı TxE yazılır. TxE diskteyse işlem kesinleşmiştir.
4. **Kontrol noktası.** I ve B son yerlerine yazılır. Bu yazmalar arada kesilse de sorun yoktur; günlükten tekrarlanabilir.
5. **Serbest bırakma.** Günlüğün süperbloğunda işlem tamamlandı olarak işaretlenir. Günlük dairesel bir tampon gibi yeniden kullanılır.



Şekil 10.50: Kontrol noktası: günlükte TxB, I[v2], B[v2], TxE; son yerlerde biteşlem ve i-node tablosu v2'ye güncellendi.

Kurtarma basittir. Açılışta günlük taranır. TxE'si olan işlemler baştan oynatılır: blokları son yerlerine yeniden yazılır (*redo*). Yeniden yazmak eşgüçlü olduğu için blokların bir kısmının çökme öncesinde zaten yazılmış olması sorun değildir. TxE'si olmayan işlemler atılır; son yerlerine hiç dokunulmamış olduğu için dosya sistemi eski, tutarlı durumundadır. fsck gerekmez; kurtarma saniyeler sürer.

Çökmenin TxE'den önce mi sonra mı olduğu bu yüzden belirleyicidir. TxE yazılmadan çökme olursa işlem hiç olmamış sayılır ve değişiklik kaybolur. TxE yazıldıktan sonra, örneğin kontrol noktasının ortasında çökme olursa işlem günlükten yeniden oynatılır ve değişiklik korunur. Her iki durumda da dosya sistemi tutarlı kalır.

ext4'ün üç modu vardır. **journal** modunda veri de günlüğe yazılır; en güvenlisi ama en yavaşıdır, çünkü her veri iki kez yazılır. **ordered** modunda yalnızca üst veri günlüğe girer, ama veri her zaman üst veriden önce yazılır; böylece bir i-node hiçbir zaman çöp içeren bir bloğu göstermez. **writeback** modunda veri için sıra garantisi yoktur. Günlük kullanan sistemler arasında ext3/ext4 (jbd2), XFS, NTFS (`$LogFile`) ve JFS sayılabilir. Apple'da günlüklü HFS+ yerini yazınca kopyalama kullanan APFS'e bıraktı.

10.10.6 Kod izleme: günlükten kurtarma

Aşağıdaki `kurtar` fonksiyonu n kayıtlık bir günlüğü baştan tarar. Kayıtların türü `TXB`, `BLOK` ya da `TXE`'dir; `BLOK` kayıtları hedef bloğun numarasını ve yeni içeriğini taşır.

```
void kurtar(Kayit g[], int n) {
  int i = 0;
  while (i < n && g[i].tur == TXB) {
    int j = i + 1;
    while (j < n && g[j].tur == BLOK)
      j++;
    if (j == n || g[j].tur != TXE)
      break;              /* onaysız: at */
    for (int k = i + 1; k < j; k++)
      yaz(g[k].hedef, g[k].veri);
    i = j + 1;          /* sonraki */
  }
}
```

`i` bir işlemin başını gösterir. Dış döngü, `g[i]` bir TxB olduğu sürece sürer. İç döngü `j` 'yi blok kayıtlarının üzerinden geçirir ve işlemin sonuna götürür. `j` günlüğün sonuna geldiyse ya da orada bir TxE yoksa işlem onaylanmamıştır: döngü kırılır ve işlem atılır. Onaylı işlemin blokları `yaz` ile son yerlerine kopyalanır ve `i` TxE'nin bir sonrasına geçer. Fonksiyon günlüğü bir kez tarar ve her blok kaydını en çok bir kez yazar: O(n).

Sistem üçüncü işlem yazılırken çökmüştür. Günlükte on kayıt vardır: TxB 1, I7 v1, B v1, TxE 1, TxB 2, I9 v2, D40 v2, TxE 2, TxB 3, I7 v3. Son yerlerdeki bütün bloklar hâlâ v0'dır. Tablo 10.22 izlemeyi verir.

i	İç döngü sonunda j	g[j]	Karar	Son yerler sonra (I7, B, I9, D40)
0 (TxB 1)	3	TxE 1	oynat: I7 ← v1, B ← v1	v1, v1, v0, v0
4 (TxB 2)	7	TxE 2	oynat: I9 ← v2, D40 ← v2	v1, v1, v2, v2
8 (TxB 3)	10 = n	—	onaysız: at, döngüden çık	v1, v1, v2, v2

Tablo 10.22: `kurtar` izlemesi (Python'a birebir çevrilerek çalıştırıldı).



Şekil 10.51: Kurtarmanın sonu: işlem 1 ve 2 uygulandı; işlem 3'ün I7 v3 sürümü hiç yazılmadı.

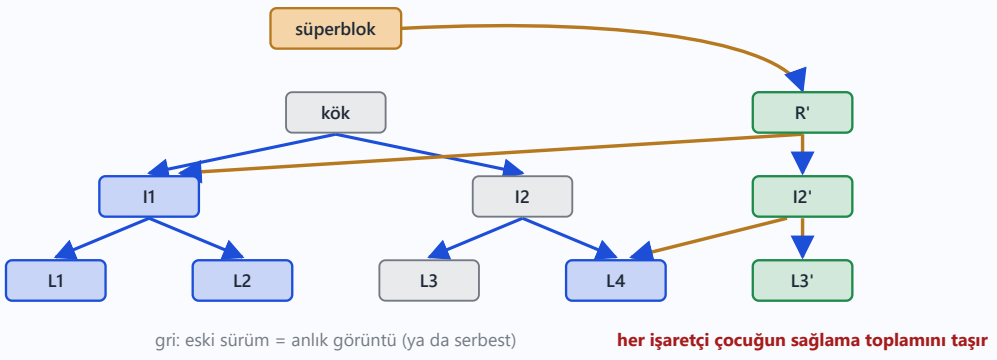
İşlem 1 ve 2 tam olarak uygulanır, işlem 3 hiç olmamış gibidir. I7'nin v1 sürümü çökmeden önce son yerine yazılmış olabilir; yeniden yazmak bir şey değiştirmez. Dosya sistemi tutarlıdır ve diski baştan sona tarayan bir fsck gerekmez.

> **Sık yapılan hata**
>
> Günlükte bulunan her bloğu yeniden oynatmak. İşlem 3'ün I7 v3 kaydı günlükte durur, ama TxE'si yoktur. Onu yazmak, işlemin yalnızca bir kısmını uygulamak demektir; tutarsızlığın ta kendisidir. Yalnızca onaylanmış işlemler oynatılır.

10.10.7 Yazınca kopyalama

Günlükleme her değişikliği iki kez yazar: önce günlüğe, sonra son yerine. **Yazınca kopyalama** (*copy-on-write*, CoW) kullanan dosya sistemleri başka bir yol izler: hiçbir bloğun üzerine yazmazlar. Dosya sistemi bir blok ağacıdır (Şekil 10.52). En üstte süperblok (ZFS'te *uberblock*), onun altında kök, iç düğümler ve en altta veri blokları vardır.

1. **Yaprak.** L3 bloğu değişecek olsun. L3 yerinde güncellenmez; yeni sürümü L3' boş bir yere yazılır. Eski L3 hâlâ sağlamdır.
2. **Yol.** L3'ü gösteren üst düğümler de değişmek zorundadır, çünkü işaretçileri değişecektir. I2 ve kök de kopyalanır: I2', R'. Değişmeyen alt ağaçlar (I1, L4) yeni sürümle paylaşılır.
3. **Atomik geçiş.** Son olarak süperblok R''yü gösterecek biçimde tek bir atomik yazmayla güncellenir. Çökme bu yazmadan önce olursa eski ağaç, sonra olursa yeni ağaç görünür. Ara durum yoktur; günlük gerekmez.



Şekil 10.52: Yazınca kopyalama: değişen yaprak ve yolu yeni yere yazılır, süperblok atomik güncellenir. Gri: eski sürüm, anlık görüntü olarak saklanabilir.

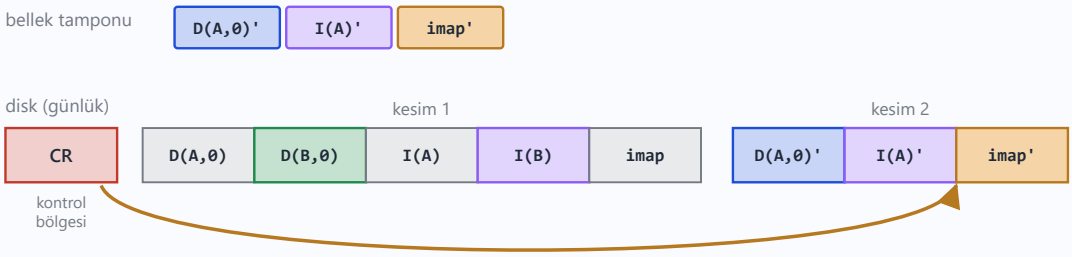
Bu tasarımın iki güzel yan ürünü vardır. Eski kök saklanırsa eski ağacın tamamı hâlâ geçerlidir: **anlık görüntü** bedavaya gelir. ZFS ve Btrfs ayrıca her işaretçinin yanında gösterdiği bloğun **sağlama toplamını** tutar. Diskin bir bloğu sessizce bozulursa okunurken yakalanır ve ayna kopyadan onarılır. Bütün diski okuyup bu denetimi yapan işleme *scrub* denir.

Yazınca kopyalama kullanan sistemler ZFS (Sun, 2005; bugün OpenZFS), Btrfs (Linux), APFS (Apple, 2017) ve ReFS'tir (Windows Server). Bedeli rastgele üzerine yazmalarda ortaya çıkar. Dosyanın ortasındaki her küçük değişiklik yeni bir yere yazıldığı için dosya zamanla parçalanır. Bir yaprak değişikliği bütün yolun kopyalanmasını gerektirdiği için yazma büyütmesi de artar. Veritabanı ve sanal makine disk dosyalarında CoW bu yüzden kapatılabilir.

10.10.8 Günlük yapılı dosya sistemi

Rosenblum ve Ousterhout 1991'de Sprite işletim sistemi için radikal bir tasarım önerdi: **günlük yapılı dosya sistemi** (*log-structured file system*, LFS). Tasarımın varsayımı şuydu: bellek büyüdükçe okumaların çoğu önbellekten karşılanır, diske giden trafik ağırlıkla yazmadır. O hâlde yazmalar en hızlı biçimde, yani tek bir uzun sıralı yazmayla yapılmalıdır.

LFS'te bütün disk tek bir günlüktür. Yazmalar bellekte biriktirilir: veri blokları, değişen i-node'lar ve i-node haritasının (imap) parçaları. Tampon dolunca hepsi tek bir **kesim** (*segment*, ör. 1–2 MiB) olarak günlüğün sonuna sıralı yazılır (Şekil 10.53). i-node'ların sabit bir yeri yoktur; her yazmada yer değiştirirler. imap bir i-node numarasından onun son konumunu bulur. Diskin sabit bir yerindeki **kontrol bölgesi** (CR) en son imap parçalarını gösterir.



Şekil 10.53: A'nın bloğu değişince yeni sürüm kesim 2'ye yazılır; kesim 1'deki eski D(A,0), I(A) ve imap ölü olur.

A dosyasının bir bloğu değişince yeni sürümü, yeni i-node'u ve yeni imap parçası yeni bir kesime yazılır. Kesim 1'deki eski kopyalar artık ölüdür, ama yer kaplamaya devam eder. Disk dolmadan bu alanın geri kazanılması gerekir. Bu işi **temizleyici** (*cleaner*) yapar: az canlı bloğu olan kesimleri okur, canlı blokları yeni bir kesime sıkıştırır ve eski kesimleri boşaltır. Disk doluyken ve yük altındayken temizleme maliyeti büyür; LFS'in zayıf noktası budur.

LFS, yazma bant genişliğini neredeyse diskin sıralı hızına çıkarır. Fikir bugün de birçok yerde yaşar: Android'in F2FS'i, Linux'un NILFS2'si, NetApp'ın WAFL'ı, ZFS ve Btrfs'in yazınca kopyalaması ve SSD'lerin FTL'leri. SSD'nin içi zaten bir günlük yapılı sistemdir.

10.11 VFS ve ağ dosya sistemi

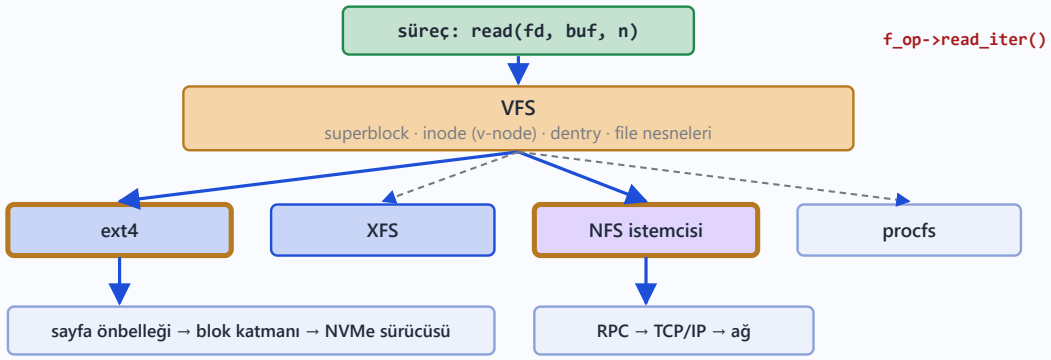
10.11.1 Sanal dosya sistemi

Bir Linux makinesinde aynı anda birçok dosya sistemi bağlıdır: kök dizinde ext4, bir veri diskinde XFS, takılı bir USB bellekte exFAT, ağdan bağlanmış bir NFS paylaşımı, bellekte yaşayan `/proc`. Uygulama bunların hepsine aynı `open` ve `read` çağrılarıyla erişir. Bu birliği çekirdeğin bir katmanı sağlar.

> **Tanım 10.16**
>
> **Sanal dosya sistemi** (*virtual file system*, VFS), farklı dosya sistemlerini ortak bir nesne arayüzünün arkasında birleştiren çekirdek katmanıdır. Her somut dosya sistemi bu arayüzün işlemlerini kendi işlevleriyle gerçekler.

Linux VFS'i dört nesne türüyle çalışır: **superblock** (bağlı bir dosya sistemi), **inode** (bir dosya; Sun'ın özgün tasarımındaki adıyla v-node), **dentry** (yoldaki bir ad bileşeni) ve **file** (bir açılış). Her nesne bir işlem tablosu taşır: işlevlere işaretçilerden oluşan bir yapı. Mekanizma dört adımda kurulur (Şekil 10.54).

1. **Kayıt.** Bir dosya sistemi, açılışta ya da modülü yüklenince `register_filesystem()` ile VFS'ye kaydolur ve VFS'nin çağıracağı işlevlerin adres tablosunu verir.
2. **Bağlama.** VFS o dosya sisteminden bir süperblok nesnesi kurmasını ister. Nesne süperblok işlemleri tablosunu (`s_op`) taşır.
3. **open.** Yol dentry'lerle çözülür. Somut dosya sistemi i-node bilgisini bir VFS i-node'una yerleştirir; i-node işlemleri (`i_op`) ve dosya işlemleri (`f_op`) tabloları bağlanır. Bir file nesnesi oluşur ve sürecin tanımlayıcı tablosuna eklenir.
4. **read.** VFS yalnızca `file->f_op->read_iter` fonksiyon işaretçisini çağırır. Dosya ext4'teyse ext4'ün işlevi çalışır ve sayfa önbelleği, blok katmanı ve NVMe sürücüsü üzerinden veriyi getirir. Dosya NFS'teyse NFS istemcisinin işlevi çalışır ve ağ üzerinden bir uzak yordam çağrısı (RPC) gönderir.



Şekil 10.54: VFS katmanı: `read` çağrısı file nesnesindeki `f_op->read_iter` işaretçisiyle somut dosya sisteminin işlevine yönlenir.

Bu, C'de yazılmış çok biçimliliktir: nesne yönelimli bir dildeki sanal metot çağrısının aynısı. Uygulama farkı görmez. Ek maliyet yalnızca bir dolaylı çağrıdır; performans kaybı küçüktür. VFS'nin amacı farklı dosya sistemlerine *aynı anda* tek bir arayüz sunmaktır. Aynı diskte birden çok dosya sistemi bulundurmak için VFS gerekmez; bölümleme bunu zaten sağlar. Windows'ta aynı rolü G/Ç yöneticisi ve dosya sistemi sürücüleri oynar.

Sık yapılan hata

Uygulamanın dosya sistemine göre farklı bir sistem çağrısı kullandığını ya da çekirdeğin bütün dosya sistemlerinin kodunu sırayla denediğini sanmak. Sistem çağrısı aynıdır. Doğru kod, `open` sırasında file nesnesine bağlanan işlem tablosundan tek bir dolaylı çağrıyla seçilir.

10.11.2 NFS

Ağ dosya sistemi (*network file system*, NFS), uzaktaki bir makinenin dizinini yerel ağaca bağlar. Uygulama uzak dosyaları yerel dosyalar gibi kullanır. Sun, NFS'i 1984'te SunOS için tasarladı ve protokolü açık bir standart olarak yayımladı. Bu yüzden bugün hemen her UNIX benzeri sistemde bulunur.



Şekil 10.55: NFS istemci ve sunucu katmanları: istemcide VFS'nin altındaki NFS istemcisi, sunucuda nfsd ve

yerel dosya sistemi.

Şekil 10.55 katmanları gösterir. İstemcide `mount sunucu:/export /mnt/proje` uzak dizini bağlar. Süreç `/mnt/proje` altındaki bir dosyayı okuyunca çağrı VFS'den NFS istemcisine gider. İstemci bir `READ(dosya tanıtıcısı, ofset, n)` isteği hazırlar ve TCP 2049 portu üzerinden sunucuya gönderir. Sunucudaki `nfsd` isteği kendi VFS'si üzerinden yerel dosya sistemine (ext4, XFS, ZFS) iletir ve yanıtı geri yollar.

NFS makine ve mimari bağımsızlığını iki araçla sağlar. Uzak yordam çağrısı (RPC) istek-yanıt alışverişini bir işlev çağrısı gibi gösterir. Dış veri gösterimi (XDR) ise sayıları ve yapıları bayt sırasından bağımsız bir biçimde kodlar; böylece farklı mimarilerdeki istemci ve sunucu anlaşır. NFS'in sürümleri tasarımda önemli bir ayrımı da gösterir. NFSv3 **durumsuzdur**: sunucu hangi istemcinin hangi dosyayı açtığını hatırlamaz, her istek gereken her şeyi (dosya tanıtıcısı, bağıl konum) taşır. Sunucu çöküp yeniden başlasa da istemciler yalnızca isteklerini tekrarlar. NFSv4 ise **durumludur**. Yalnızca TCP ve tek bir port (2049) kullanır; Kerberos güvenliği, ACL, dosya kilitleme ve devretme (*delegation*) içerir. v4.1'deki pNFS, verinin birden çok sunucudan paralel okunmasını sağlar. NFS başlangıçta UDP/IP kullanıyordu; bugün TCP tercih edilir. Tek şart bir IP ağıdır. Windows dünyasında karşılığı şifreleme ve çok kanallı bağlantı destekleyen SMB 3'tür.

10.12 Özet

- Disk yalnızca numaralı bloklar sunar. **Dosya sistemi** bunun üzerine adlı, kalıcı bayt dizileri olan **dosyaları** ve onları eşleyen **dizinleri** kurar. Dosya türü uzantıyla değil, **sihirli sayıyla** güvenilir biçimde anlaşılır.
- Öznitelikler i-node'da, ad dizin girdisindedir. `stat` öznitelikleri okur; mtime içeriğin, ctime i-node'un değiştiği andır.
- Açık bir dosya üç düzeyde izlenir: süreç başına **tanımlayıcı tablosu** → sistem geneli **açık dosya tablosu** (konum, mod) → bellek içi **i-node**. `fork` ve `dup` konumu paylaşır, ikinci `open` ayrı bir konum açar. Kısa okuma hata değildir; 0 dönüşü dosya sonudur.
- Güvenli kaydetme: geçici dosyaya yaz → `fsync` → `rename`.
- Dizin yapıları tek düzeyden ağaca ve döngüsüz çizgeye gelişti; UNIX dizinlere katı bağı yasaklar. Yol **bileşen bileşen** çözülür: kök i-node'u bellekteyse k bileşen 2k okuma, ilk veri bloğuyla 2k + 1 okuma ister.
- **Katı bağ** aynı i-node'a ikinci addır ve bağ sayısını artırır; **sembolik bağ** hedefin yolunu saklar ve hedef silinince askıda kalır. **Bağlama** ayrı dosya sistemlerini tek ağaçta birleştirir.
- rwx bitleri sahip, grup ve diğerleri için ayrı tutulur; ilk eşleşen sınıf uygulanır. Dizinde w girdi ekleyip silmeyi, x içinden geçmeyi sağlar. setuid, sticky ve umask (izin = istek & ~umask) bu düzeni tamamlar.

- MBR 32 bit LBA ile 2 TiB'ta sınırlıdır; GPT 64 bit LBA, 128 bölüm ve yedek başlık sunar. i-node'un yeri numarasından hesaplanır: bağıl konum = n × boy, disk bloğu = tablo başı + bağıl konum div blok.
- **Bitişik tahsis** hızlıdır ama dış parçalanma üretir. **Bağlı liste** esnektir ama rastgele erişimi O(n) disk okumasıdır. **FAT** zinciri belleğe taşır ama tablo diskle büyür. **i-node** yalnızca açık dosyalar için bellek ister; 12 doğrudan, tek, çift ve üç dolaylı işaretçiyle 4 KiB blokta ≈ 4 TiB'a ulaşır. Extent'ler bitişik blokları tek kayıtla anlatır.
- Dizinler doğrusal liste, hash ya da B+ ağacıyla tutulur; büyük dizinlerde ağaç hem O(log n) arama hem sıralı listeleme verir.
- Boş alan **biteşlem**, bağlı liste, **gruplama** ya da **sayma** ile izlenir. Biteşlem sabit boyutludur; liste yalnızca disk doluyken küçüktür. Blok boyutu veri hızı ile alan verimliliği arasında bir ödünleşimdir.
- Önbelleksiz bir dosya oluşturup 12 KiB yazmak 25 G/Ç'dir. **Tampon önbelleği** hash ve LRU ile çalışır; kirli bir tamponu atmak ıskanın maliyetini ikiye katlar. Birleşik sayfa önbelleği çift önbellekleme yi önler. Önden okuma ve yakın yerleşim sıralı erişimi hızlandırır; SSD'ye silinen bloklar **TRIM** ile bildirilir.
- Yedek için 3-2-1 kuralı geçerlidir; mantıksal döküm dört aşamalıdır. **fsck** blok ve bağ sayılarını denetler ama yavaştır. **Günlükleme** TxE'si olan işlemleri yeniden oynatır, olmayanları atar. **Yazınca kopyalama** hiçbir bloğun üzerine yazmaz ve süperbloğu atomik değiştirir. **LFS** her şeyi sıralı bir günlüğe yazar ve temizleyiciyle alan kazanır.
- **VFS** fonksiyon işaretçisi tablolarıyla farklı dosya sistemlerine tek arayüz sunar. **NFS** uzak dizinleri RPC ve XDR ile yerel ağaca bağlar.

10.13 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

dosya sistemi *file system* · dosya tanımlayıcısı *file descriptor* · açık dosya tablosu *open-file table* · dizin, dizin girdisi *directory, directory entry* · yol adı (mutlak / göreli) *path name* · katı bağ / sembolik bağ *hard link / symbolic link* · bağlama *mount* · izin bitleri *permission bits (rwx)* · erişim denetim listesi (ACL) *access control list* · süperblok *superblock* · i-node · bitişik tahsis *contiguous allocation* · bağlı liste ile tahsis *linked allocation* · indisli tahsis *indexed allocation* · iç / dış parçalanma *internal / external fragmentation* · biteşlem *bitmap* · tampon önbelleği *buffer cache* · sayfa önbelleği *page cache* · günlükleme *journaling* · yazınca kopyalama *copy-on-write* · sanal dosya sistemi (VFS) *virtual file system* · ağ dosya sistemi (NFS) *network file system*

10.14 Kendini Dene

1. 9000 baytlık bir dosya 2048 baytlık bir tamponla, `read` 0 döndürene kadar okunuyor. `read` çağrıları sırasıyla ne döndürür?

   Cevap

   2048, 2048, 2048, 2048, 808 ve 0. Dört tam okuma 8192 bayt getirir; kalan 9000 − 8192 = 808 bayt kısa bir okumayla gelir, son çağrı dosya sonunu bildirir. Toplam 6 çağrı.

2. `mkdir p`, `mkdir p/x`, `mkdir p/y`, `echo z > p/f`, `ln p/f p/g` ve `ln -s f p/h` çalıştırılıyor. `p`, `p/x`, `p/f` ve `p/h` 'nin bağ sayıları nedir?

   Cevap

   `p`: 4 (üst dizindeki girdi, kendi `.`, x ve y'nin `..` girdileri). `p/x`: 2. `p/f`: 2 (f ve g aynı i-node). `p/h`: 1 (sembolik bağın kendi i-node'u).

3. i-node tablosu blok 3'ten başlıyor; blok 4 KiB, i-node 128 B, numaralar 0'dan. i-node 50 hangi disk bloğunda ve hangi sektörde (512 B) başlar?

   Cevap

   50 × 128 = 6400 = 1 × 4096 + 2304: tablonun 1. bloğu, yani disk bloğu 3 + 1 = 4, blok içinde 2304. bayt. Bayt adresi 4 × 4096 + 2304 = 18.688; 18.688 / 512 = 36,5 → sektör 36, içinde 256. bayt.

4. Küme 2 KiB; FAT'ta FAT[8] = 3, FAT[3] = 13, FAT[13] = 6, FAT[6] = EOF. İlk kümesi 8 olan dosyanın 5000. baytı hangi kümede ve kümenin kaçıncı baytındadır?

   Cevap

   5000 div 2048 = 2: dosyanın 2. kümesi (0'dan). Zincir 8 → 3 → 13 → 6 olduğundan 2. küme fiziksel 13'tür. Küme içi konum 5000 mod 2048 = 904. Zincir bellekte izlendiği için disk okuması yalnızca veri kümesi içindir.

5. Blok 8 KiB, adres 4 B; i-node'da 12 doğrudan, 1 tek, 1 çift, 1 üç dolaylı işaretçi var. En büyük dosya yaklaşık ne kadardır?

   Cevap

   P = 8192 / 4 = 2048. Blok sayısı $12 + 2048 + 2048^2 + 2048^3 = 8.594.130.956$; × 8192 B ≈ $70{,}4 \times 10^{12}$ B ≈ 64 TiB. Blok boyunu ikiye katlamak kapasiteyi yaklaşık $2^4 = 16$ kat artırır: hem blok iki kat büyür hem üç düzeyin her biri iki kat fazla adres tutar.

6. 4 KiB blok, 4 B adres, 12 doğrudan işaretçi. 10.000.000. bayt hangi düzeydedir ve i-node bellekteyse kaç disk okuması gerekir?

   Cevap

   10.000.000 div 4096 = 2441. 2441 − 12 − 1024 = 1405 ≥ 0 olduğundan çift dolaylı bölgededir: dış indis 1405 div 1024 = 1, iç indis 1405 mod 1024 = 381. Blok içi konum 1664. Okumalar: çift dolaylı blok, tek dolaylı blok, veri = 3.

7. 24 bloklu bir diskte boş bloklar 1, 2, 3, 7, 8, 9, 10, 15, 20 ve 21. Bu boş alan sayma yöntemiyle ve biteşlemle (1 = boş) nasıl gösterilir?

   Cevap

   Sayma: (1, 3), (7, 4), (15, 1), (20, 2). Biteşlem: 01110001 11100001 00001100. On boş blok dört çifte iner.

8. 3 tamponlu bir önbellekte LRU sırası 3, 8, 6 (3 en eski ve kirli). İstekler 8, 1, 3, 6. `bread` kaç okuma ve kaç yazma yapar?

   Cevap

   8 isabettir (sıra 3, 6, 8). 1 ıskadır; en eski 3 kirli olduğu için önce yazılır, sonra 1 okunur (6, 8, 1). 3 ıskadır: az önce atılan blok yeniden okunur, 6 atılır (8, 1, 3). 6 ıskadır, 8 atılır (1, 3, 6). Toplam 3 okuma, 1 yazma. Bu küçük önbellekte LRU, hemen geri istenecek blokları atar.

9. Çökmeden sonra günlükte şu kayıtlar var: TxB 1, I4 v1, TxE 1, TxB 2, D8 v2, B v2. Son yerlerde her şey v0. `kurtar` sonunda I4, D8 ve B hangi sürümdedir?

   Cevap

   İşlem 1 onaylıdır: I4 ← v1. İşlem 2'nin TxE'si yoktur; j günlüğün sonuna (6) ulaşır ve işlem atılır. Sonuç I4 v1, D8 v0, B v0. D8 v2 günlükte bulunsa da yazılmaz.

10. Bir dosyanın izni `-rw-r--r--`, bulunduğu dizinin izni `drwxrwxrwx`. Dosyanın sahibi olmayan sıradan bir kullanıcı dosyayı silebilir mi? Dizinin izni `drwxrwxrwt` olsaydı ne değişirdi?

    Cevap

    Silebilir. Silme bir dizin girdisini kaldırmaktır ve dizinin w hakkını ister; dosyanın kendi izni önemsizdir. Sticky bit (sondaki t) açık olsaydı girdiyi yalnızca dosyanın sahibi, dizinin sahibi ya da root silebilirdi.

10.15 İleri okuma

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, "File System Implementation", "Crash Consistency: FSCK and Journaling" ve "Log-structured File Systems" bölümleri: bu nottaki G/Ç sayımlarını ve günlükleme adımlarını çok daha ayrıntılı anlatır. Kitap ostep.org'da ücretsizdir.
- M. K. McKusick, W. N. Joy, S. J. Leffler, R. S. Fabry, "A Fast File System for UNIX", ACM TOCS, 1984: silindir gruplarını ve yakın yerleşim fikrini ortaya koyan, bugünkü ext4'ün düzenini açıklayan klasik makale.
- M. Rosenblum ve J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", ACM TOCS, 1992: LFS'in tasarımını, temizleyici politikalarını ve ölçümlerini anlatır.
- Linux çekirdek belgeleri, "Overview of the Linux Virtual File System" ve "ext4 Data Structures and Algorithms": VFS nesnelerini ve ext4'ün disk düzenini, extent ağacını ve günlüğünü birinci elden anlatır.

10.16 Kaynaklar

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 13 (File-System Interface), 14 (File-System Implementation), 15 (File-System Internals).
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 4: File Systems.
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Files and Directories", "File System Implementation", "Fast File System", "Crash Consistency: FSCK and Journaling", "Log-structured File Systems", "Flash-based SSDs", "Data Integrity", "Sun's NFS".
4. M. Rosenblum, J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", ACM TOCS, 1992; M. K. McKusick ve diğ., "A Fast File System for UNIX", ACM TOCS, 1984.
5. Linux çekirdek belgeleri: "Overview of the Linux Virtual File System", "ext4 Data Structures and Algorithms"; Microsoft Learn: NTFS ve exFAT belirtimi; OpenZFS ve Btrfs belgeleri; UEFI belirtimi (GPT).

KISIM III · BÖLÜM 11

Giriş/Çıkış

Bir bilgisayar aynı anda çok farklı aygıtları yönetir. Klavye saniyede birkaç bayt gönderir; NVMe disk ise saniyede milyonlarca isteği karşılar. Bu aygıtların hızları dokuz basamak, arayüzleri ise birbirinden tamamen farklıdır. İşletim sisteminin G/Ç (giriş/çıkış) kısmı bu çeşitliliği iki yoldan denetler. Donanım tarafında aygıtlara komut verir, durumlarını okur, kesmeleri yakalar ve veriyi CPU'yu yormadan taşır. Yazılım tarafında ise bütün aygıtları uygulamaya tek tip bir arayüzle sunar. Not, denetleyiciler, aygıt yazmaçları ve iki adresleme yöntemiyle başlar. Ardından yoklama, kesme güdümlü G/Ç ve DMA aynı yazıcı örneği üzerinde karşılaştırılır. G/Ç yazılımının katmanları, tamponlama, eş zamansız arayüzler ve saatler bunu izler. Son kesimler sabit diskin erişim süresini, disk kolu çizelgeleme algoritmalarını, SSD'leri, RAID dizilerini ve kararlı depolamayı ele alır. Kafa hareketleri, erişim süreleri, tampon izlemeleri ve XOR eşliği gibi bütün sayısal örnekler algoritmalar gerçekten çalıştırılarak denetlenmiştir.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum11` · **Laboratuvar:** `github.com/sercankulcu/operating-systems-labs/lab11`

ÖĞRENME ÇIKTILARI

Donanım
Aygıt denetleyicisini, aygıt yazmaçlarını ve PCIe, USB, NVMe gibi veri yollarını tanımak; kapı eşlemeli ve bellek eşlemeli G/Ç'yi açıklamak.

Aktarım yöntemleri
Yoklama, kesme güdümlü G/Ç ve DMA'yı CPU maliyetiyle karşılaştırmak; MSI-X ve IOMMU'nun görevini söylemek.

Yazılım katmanları
Kesme işleyicisi, aygıt sürücüsü, aygıttan bağımsız katman ve kullanıcı kütüphanesini bir `read()` çağrısının yolunda izlemek; dairesel tamponu ve zamanlayıcı listesini kodla izlemek.

Arayüzler
Bloke eden, bloke etmeyen ve eş zamansız G/Ç'yi ayırt etmek; epoll ile io_uring'in modellerini karşılaştırmak.

Diskler
Erişim süresini hesaplamak; FCFS, SSTF, SCAN, C-SCAN, LOOK ve C-LOOK için kafanın yolunu ve toplam hareketini bulmak.

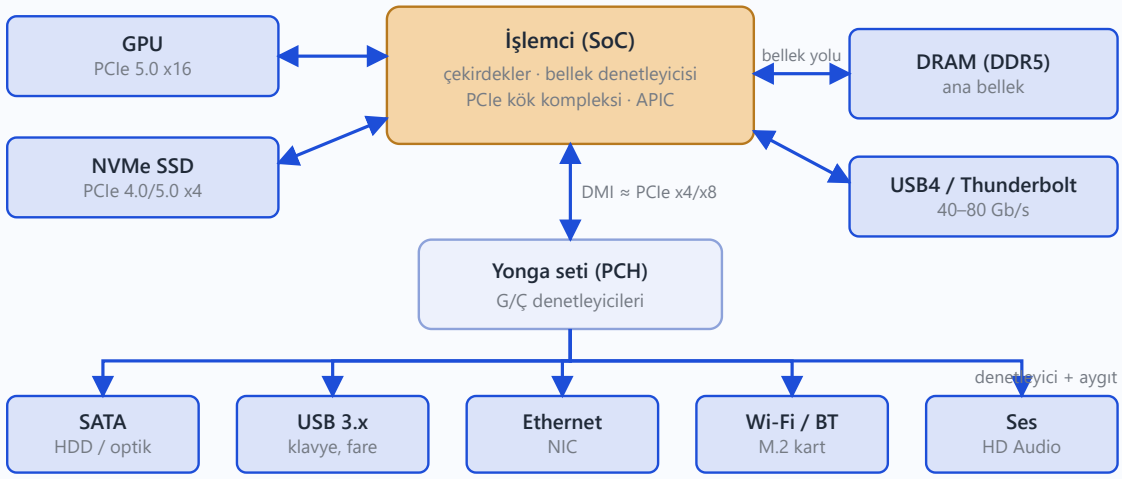
SSD ve RAID
FTL, aşınma dengeleme, TRIM ve blk-mq'yu açıklamak; RAID 0/1/5/6/10 düzenlerini, XOR eşliğini, küçük yazmayı ve kurtarmayı hesapla göstermek.

11.1 G/Ç sistemine genel bakış

11.1.1 İşletim sistemi aygıtlarla ne yapar?

Bir programın dosya okuması, ekrana yazı basması ya da ağdan paket alması, sonunda bir aygıtla konuşmayı gerektirir. Bu konuşmayı programın kendisi yapmaz. İşletim sistemi aygıtlara komut verir, durumlarını okur, işin bittiğini haber veren kesmeleri yakalar ve hataları ele alır. Uygulamaya ise aygıttan bağımsız, katmanlı ve tek tip bir arayüz sunar. Aynı `read()` çağrısı bir SSD'den, bir USB bellekten ya da bir seri porttan okuyabilir.

İşin büyüklüğünü bir sayı iyi anlatır. Linux kaynak kodunun yaklaşık %60–70'i `drivers/` dizinindedir. Çekirdeğin en büyük parçası süreç yönetimi ya da bellek yönetimi değil, binlerce farklı aygıtın sürücüleridir. Her yeni ağ kartı, ekran kartı ya da sensör, çekirdeğe yeni bir sürücü ekler.



Şekil 11.1: Modern bir bilgisayarın G/Ç düzeni. Bellek denetleyicisi ve PCIe kök kompleksi işlemcinin içindedir. Hızlı aygıtlar (GPU, NVMe SSD) PCIe ile doğrudan işlemciye bağlanır. Yavaş aygıtlar yonga setindeki (PCH) denetleyiciler üzerinden gelir.

Şekil 11.1 bugünkü bir masaüstü bilgisayarın G/Ç düzenini gösterir. Her aygıt iki parçadan oluşur: bir denetleyici ve onun sürdüğü asıl aygıt. İşletim sistemi yalnız denetleyiciyle konuşur. Denetleyiciler de çeşitli veri yollarıyla işlemciye bağlanır.

11.1.2 Veri yolları: PCIe, USB4, SATA

Tanım 11.1

Veri yolu (*bus*), bileşenler arasında veri, adres ve denetim sinyallerini taşıyan bağlantıdır. Eski PC'lerde birçok aygıtın paylaştığı paralel yollar (ISA, PCI) vardı. Bugün noktadan noktaya seri hatlar kullanılır: PCIe, USB, SATA.

Paylaşılan paralel yolda aynı anda yalnız bir aygıt konuşabilir. Yol genişledikçe teller arasındaki zamanlama farkı da sorun olur. Seri hatlar bu iki sorunu birlikte çözer. Her bağlantı yalnız iki uç arasındadır ve veri tek bir hat çifti üzerinde çok yüksek frekansla akar. PCIe'de bir **hat** (*lane*) tam çift yönlüdür; bağlantı x1, x4, x8 ya da x16 hattı birleştirir. Bant genişliği her yönde ayrı ayrıdır: x4 bir NVMe disk okurken de yazarken de aynı hızı kullanabilir.

Bağlantı	Ham hız	Etkin veri hızı	Not
PCIe 4.0	16 GT/s / hat	≈ 1,97 GB/s / hat; x4 ≈ 7,9 GB/s	128b/130b kodlama; NVMe SSD'ler
PCIe 5.0	32 GT/s / hat	≈ 3,94 GB/s / hat; x16 ≈ 63 GB/s	GPU, 14 GB/s sınıfı SSD
PCIe 6.0	64 GT/s (PAM4)	≈ 7,6 GB/s / hat; x16 ≈ 121 GB/s	FLIT + FEC; sunucularda yaygınlaşıyor
USB 2.0 / 3.2	480 Mb/s / 5–20 Gb/s	≤ 60 MB/s / ≤ 2,5 GB/s	Ana makine yoklamalı protokol
USB4 v1 / v2	40 / 80 Gb/s	PCIe, DisplayPort tünelleme	Thunderbolt 4/5 ile uyumlu
SATA III	6 Gb/s	≈ 600 MB/s (8b/10b)	AHCI: tek kuyruk, 32 komut
Ethernet	1 / 10 / 100 / 400 Gb/s	125 MB/s … 50 GB/s	Veri merkezi: 400–800 GbE

Tablo 11.1: Bugünkü başlıca bağlantılar ve hızları.

Örnek 11.1

PCIe 4.0'ın bir hattı 16 GT/s (saniyede 16 milyar aktarım) hızındadır ve 128b/130b kodlama kullanır. Bir hattın ve x4 bağlantının etkin veri hızı nedir?

1. **Kodlama payı.** Her 130 bitin 128'i veridir; kalan 2 bit eş zamanlama başlığıdır. Etkin bit hızı 16 × 128 / 130 ≈ 15,75 Gb/s olur.
2. **Bayta çevirme.** 15,75 / 8 ≈ 1,97 GB/s. Bu, bir hattın tek yöndeki hızıdır.
3. **x4 bağlantı.** 4 × 1,97 ≈ 7,9 GB/s. Piyasadaki PCIe 4.0 NVMe disklerin 7 GB/s civarında tıkanması bu yüzdendir: protokol yükü de eklenince sınır buraya yakındır.

Aynı hesap PCIe 5.0 için 32 × 128 / 130 / 8 ≈ 3,94 GB/s verir; x16 bağlantıda bu 63 GB/s eder. PCIe 6.0 kodlamayı tamamen değiştirir: PAM4 sinyali her darbede iki bit taşır ve veri, hata düzeltme kodu (FEC) içeren 256 baytlık FLIT paketlerinde gider. Bir FLIT'in 242 baytı yararlı yüktür. Böylece 64 × 242 / 256 / 8 ≈ 7,6 GB/s bulunur.

Tablodaki bazı satırlar da tarihseldir. FireWire, 56K modem ve 10/100 Mb/s Ethernet artık ders kitabı örneği olarak kalmıştır. Pentium döneminin "kuzey köprüsü" de ortadan kalktı. Bellek denetleyicisi işlemcinin içine taşındı (AMD 2003, Intel 2008). Bugün hızlı aygıtlar PCIe ile doğrudan işlemciye, yavaş aygıtlar ise yonga setine bağlanır.

11.1.3 Aygıt veri hızları

Aygıtları ayıran ilk özellik hızlarıdır. Şekil 11.2 tipik üst sınırları logaritmik bir eksende gösterir. Eksendeki her çizgi bir öncekinin 1000 katıdır.



Şekil 11.2: Aygıt veri hızları logaritmik eksende: klavye yaklaşık 10 B/s, Gigabit Ethernet 125 MB/s, PCIe 4.0 x4 NVMe disk 7 GB/s, PCIe 5.0 x16 bağlantı 63 GB/s.

Klavye ile PCIe 5.0 x16 arasında $63 \times 10^9 / 10 \approx 6 \times 10^9$ kat fark vardır. Aynı işletim sistemi hem saniyede birkaç tuşu hem de saniyede milyonlarca NVMe isteğini yönetmek zorundadır. Bu yüzden tek bir aktarım yöntemi her aygıta uymaz. Yavaş aygıtta CPU'nun her baytla tek tek ilgilenmesi sorun yaratmaz. Hızlı aygıtta ise CPU'nun veri yoluna hiç girmemesi gerekir. Şekildeki değerler tipik üst sınırlardır. Gerçek hız sinyal kalitesine, protokol yüküne ve aygıtın kendisine bağlıdır.

11.1.4 Aygıt sınıfları: blok, karakter, ağ

Hızdan sonra ikinci ayrım, verinin nasıl adreslendiğidir. Unix geleneği aygıtları üç sınıfa ayırır.

Tanım 11.2

Blok aygıtı (*block device*) veriyi sabit boyutlu, numaralı bloklarda tutar; her blok ötekilerden bağımsız okunup yazılabilir. **Karakter aygıtı** (*character device*) bir bayt akışı üretir ya da tüketir; bu akış adreslenemez ve konumlanma (*seek*) işlemi yoktur. **Ağ aygıtı** paketlerle çalışır ve soket arayüzüyle kullanılır.

Sınıf	Örnek	Özellik	Tipik arayüz
Blok	HDD, SSD, USB bellek, SD kart	Sabit boyutlu, adreslenebilir bloklar (512 B / 4 KiB); her blok bağımsız okunur ve yazılır	`read`, `write`, `lseek`; dosya sistemi, `mmap`
Karakter	Klavye, fare, seri port, yazıcı, ses	Bayt akışı; adreslenemez, konum yok	`read` / `write`, `ioctl`; `/dev/ttyS0`
Ağ	Ethernet, Wi-Fi	Paket tabanlı; ne blok ne karakter	Soket API'si (`send`, `recv`); `/dev`'de dosyası yok
Diğer	Saatler, zamanlayıcılar, GPU	Ne akış ne blok: kesme üretir ya da bellek eşlemelidir	`timerfd`, DRM/KMS, `mmap`

Tablo 11.2: Aygıt sınıfları.

Linux'ta blok ve karakter aygıtları `/dev` altında özel dosyalar olarak görünür. `ls -l` çıktısının ilk harfi sınıfı söyler:

```
$ ls -l /dev/nvme0n1 /dev/ttyS0
brw-rw---- 1 root disk    259, 0  nvme0n1   b = blok, ana/yan numara 259,0
crw-rw---- 1 root dialout   4, 64 ttyS0     c = karakter
```

`b` blok, `c` karakter aygıtıdır. Dosya boyutunun bulunacağı yerdeki iki sayı ana ve yan numaradır. Ana numara (259) çekirdeğe hangi sürücünün çağrılacağını, yan numara (0) o sürücünün hangi biriminin kastedildiğini söyler. Dosya sistemleri yalnız blok aygıtlarının üzerine kurulabilir, çünkü bir dosyanın herhangi bir bloğuna numarasıyla gitmek gerekir.

Sık yapılan hata

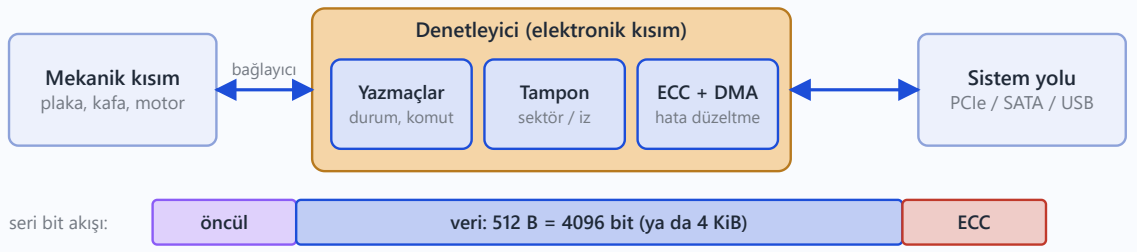
Ağ arabirimini karakter aygıtı saymak. Unix ve Linux'ta ağ ayrı bir sınıftır; `eth0` ya da `wlan0` için `/dev`'de bir dosya bulunmaz ve bu arabirimlere `open()` ile değil, soketlerle erişilir. Sınıflandırma her aygıta da tam oturmaz. Teyp sürücüsü blok yazar ama konumlanması çok yavaştır. Saat ise ne bayt akışı ne de blok üretir; yalnız kesme üretir.

11.2 G/Ç donanımı

11.2.1 Aygıt denetleyicileri

Tanım 11.3

Bir G/Ç birimi iki parçadan oluşur: mekanik ya da asıl aygıt kısmı ve onu süren elektronik **aygıt denetleyicisi** (*device controller*). Denetleyici bir yonga ya da bir karttır ve aygıta bir bağlayıcıyla bağlanır. İşletim sistemi aygıtla değil, denetleyiciyle konuşur.



Şekil 11.3: Disk denetleyicisi mekanik kısım ile sistem yolu arasında durur. İçinde yazmaçlar, bir sektör tamponu ve hata düzeltme birimi vardır. Diskten gelen seri bit akışı öncül, veri ve ECC alanlarından oluşur.

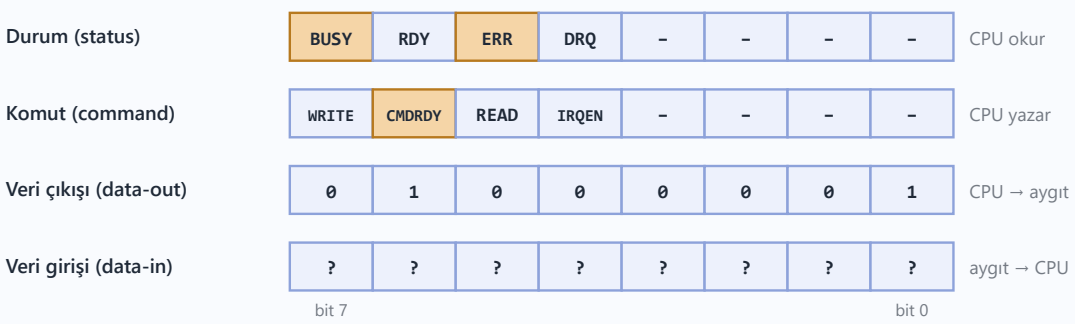
Denetleyicinin işi, aygıtın ham sinyallerini işletim sisteminin anlayacağı birimlere çevirmektir. Şekil 11.3'teki disk denetleyicisini düşünelim. Okuma kafasının altından bir bit akışı geçer. Akış bir **öncülle** (*preamble*) başlar. Öncül, denetleyicinin saatini akışa eşleyen bir desen ile silindir ve sektör numarasını taşır. Ardından 512 bayt (4096 bit) ya da 4 KiB veri, en sonda da hata düzeltme kodu (ECC) gelir. Denetleyici bit akışını bloğa çevirir, ECC ile hataları düzeltir ve bloğu kendi tamponuna koyar. Blok ancak bundan sonra belleğe aktarılır.

Denetleyicinin standart bir arayüzü olması işletim sistemi için büyük kolaylıktır. SATA ya da NVMe arayüzü sayesinde tek bir sürücü, birçok üreticinin diskini sürebilir. İşletim sistemi açılışta ve bir aygıt sıcak takıldığında yolları tarar. PCIe yapılandırma alanını ve USB tanımlayıcılarını okur, aygıtın kimliğini öğrenir ve uygun sürücüyü yükler. Bu düzene **tak-çalıştır** denir.

Disklerin sektör boyutu da değişti. Modern diskler 4 KiB fiziksel sektör kullanır (*Advanced Format*). Eski yazılımla uyum için dışarıya 512 B sektör gösterenlere 512e denir. Böyle bir diske 512 baytlık hizasız bir yazma geldiğinde denetleyici önce 4 KiB'lık sektörün tamamını okur, değişen kısmı yerleştirir ve sektörü geri yazar.

11.2.2 Aygıt yazmaçları: durum, komut, veri

Denetleyici, işletim sistemiyle yazmaçları üzerinden konuşur. Bir denetleyicinin genellikle dört yazmacı vardır.



Şekil 11.4: Örnek bir denetleyicinin dört yazmacı. Durum yazmacında meşgul (BUSY), hazır (RDY) ve hata (ERR) bitleri, komut yazmacında yaz (WRITE) ve komut hazır (CMDRDY) bitleri bulunur. Veri çıkış yazmacında-

ki 0100 0001, 'A' karakterinin kodudur (0x41).

- **Durum yazmacı** (*status*): CPU okur. Aygıtın meşgul olup olmadığını, sonucun hazır olup olmadığını, hata oluşup oluşmadığını bildirir.
- **Komut yazmacı** (*command*): CPU yazar. Denetleyiciye ne yapacağını söyler: okuma mı, yazma mı, hangi kipte.
- **Veri çıkış yazmacı** (*data-out*): CPU'dan aygıta giden veri buraya konur.
- **Veri giriş yazmacı** (*data-in*): aygıttan gelen veri CPU'nun okuması için burada bekler.

Kural basittir: yazmaca yazmak komut vermek, yazmacı okumak durum öğrenmektir. Bazı denetleyicilerde yazmaçlara ek olarak büyük veri tamponları da bulunur. Ekran kartının çerçeve tamponu ya da ağ kartının (NIC) gönderme ve alma halkaları buna örnektir. Yazmaçların düzeni aygıta özgüdür ve bu ayrıntıyı yalnız sürücü bilir. Şekil 11.4'teki bit adları da örnek amaçlıdır; gerçek bir SATA ya da NVMe denetleyicisinin yazmaç düzeni farklıdır.

11.2.3 Kapı eşlemeli ve bellek eşlemeli G/Ç

CPU'nun bir aygıt yazmacına nasıl eriştiği sorusunun iki klasik cevabı vardır.

Tanım 11.4

Kapı eşlemeli G/Ç (*port-mapped I/O*) aygıt yazmaçlarını bellekten ayrı bir adres uzayına, G/Ç kapılarına yerleştirir. CPU bu kapılara özel `IN` / `OUT` komutlarıyla erişir. **Bellek eşlemeli G/Ç** (*memory-mapped I/O*, MMIO) ise yazmaçları sıradan bellek adreslerine koyar. Erişim normal yükleme ve saklama komutlarıyla yapılır; adres çözücü erişimi ilgili aygıta yönlendirir.



Şekil 11.5: Üç yaklaşım: (a) bellek ve G/Ç kapıları için ayrı adres uzayları, (b) tek adres alanının bir aralığı aygıt yazmaçlarına ayrılmış bellek eşlemeli düzen, (c) denetim yazmaçları kapılarda, veri tamponları bellekte duran hibrit düzen.

x86 ailesi iki yöntemi birlikte taşır. IBM PC'den kalma birçok aygıt hâlâ sabit kapı numaralarındadır (Tablo 11.3). Şekil 11.5'teki hibrit düzenin klasik örneği eski PC'nin VGA kartıdır: denetim yazmaçları kapılarda, ekran belleği ise bellek adreslerindeydi.

Kapı (x86)	Aygıt
0x20–0x21	8259 kesme denetleyicisi
0x40–0x43	8254 zamanlayıcı (PIT)
0x60, 0x64	8042 klavye denetleyicisi
0x3F8–0x3FF	COM1 seri port
0xCF8–0xCFF	PCI yapılandırma

Tablo 11.3: x86'da bazı sabit G/Ç kapıları.

İki erişim biçimi makine kodunda şöyle görünür:

```
in  al, 0x60          ; kapı: özel komut
mov eax, [rbx+0x14]   ; MMIO: sıradan yükleme

Linux: inb()/outb()  ·  ioremap() + readl()/writel()
/proc/ioports  ·  /proc/iomem
```

İlk satır 0x60 kapısından, yani klavye denetleyicisinin veri yazmacından bir bayt okur. Bunun için ayrı bir komut (`in`) gerekir. İkinci satır sıradan bir bellek yüklemesidir. `rbx` bir aygıtın eşlendiği adresi gösteriyorsa bu komut o aygıtın 0x14 konumundaki yazmacını okur; CPU açısından bellekten okumaktan farkı yoktur. Linux çekirdeğinde kapılara `inb()` / `outb()` ile, bellek eşlemeli yazmaçlara ise önce `ioremap()` ile eşleyip sonra `readl()` / `writel()` ile erişilir. Hangi kapıların ve hangi fiziksel adres aralıklarının hangi aygıta ait olduğu `/proc/ioports` ve `/proc/iomem` dosyalarında görülür.

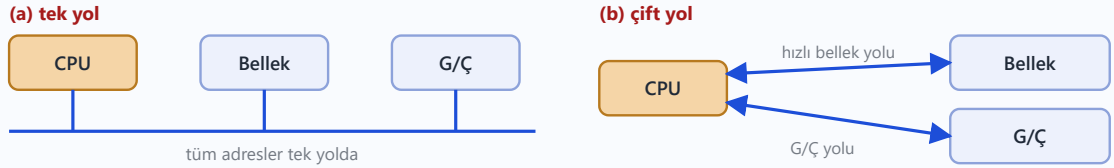
Bugün ağırlık açıkça bellek eşlemeli G/Ç'dedir. ARM ve RISC-V işlemcilerinde kapı komutları hiç yoktur; yalnız MMIO kullanılır. PCIe aygıtları da x86'da bile yazmaçlarını **BAR** (*base address register*) denen adres pencereleriyle bellek eşlemeli olarak sunar.

11.2.4 Bellek eşlemeli G/Ç: artılar ve eksiler

Bellek eşlemeli G/Ç'nin yaygınlaşmasının üç nedeni vardır. Birincisi, özel komut gerekmez. Sürücü tamamen C ile yazılabilir; yazmaç bir `volatile` işaretçi üzerinden okunup yazılır. İkincisi, koruma sayfa tablosuyla gelir. Yazmaçların bulunduğu sayfa kullanıcı uzayına eşlenmezse hiçbir kullanıcı programı aygıta dokunamaz. İstenirse bu sayfa tek bir sürece verilebilir; VFIO ve DPDK gibi kullanıcı alanı sürücüleri bu yolla çalışır. Üçüncüsü, belleğe erişen her komut yazmaçla da çalışır: `TEST` ya da `AND` gibi komutlar doğrudan yazmacın üzerinde işlem yapabilir.

Yöntemin bedelleri de vardır.

- **Önbellek.** Durum yazmacı önbelleğe alınırsa CPU sonraki okumalarda yazmacın kendisine değil, önbellekteki eski kopyaya bakar. Aygıt hazır olsa bile program bunu göremez. Bu yüzden yazmaç sayfaları önbelleklenemez olarak işaretlenir (x86'da PAT/MTRR; Linux'ta `ioremap` bunu yapar).
- **Birden çok yol.** Bellek ayrı ve hızlı bir yoldaysa G/Ç aygıtları o yoldaki adresleri göremez (Şekil 11.6). Hangi adres aralığının belleğe, hangisinin aygıtlara gideceğine köprü ya da kök kompleks karar verir.
- **Derleyici ve işlemci sıralaması.** Derleyici, değeri değişmiyor gibi görünen bir okumayı döngü dışına alabilir ya da iki yazmayı yer değiştirebilir. Yazmaç işaretçisi `volatile` olmalı, sıranın önemli olduğu yerlere bellek bariyerleri konmalıdır.



Şekil 11.6: (a) Tek veri yolunda CPU, bellek ve G/Ç aygıtları bütün adresleri görür. (b) Çift yolda bellek ayrı ve hızlı bir yoldadır; G/Ç aygıtları bellek adreslerini göremez, yönlendirme adres aralıklarıyla yapılır.

Sık yapılan hata

Durum yazmacını `while ((*durum & HAZIR) == 0) ;` döngüsüyle bekleyen sürücüde işaretçiyi `volatile` yapmamak. Derleyici `*durum` değerinin döngü içinde değişmediğini varsayar, okumayı bir kez yapar ve döngü hiç bitmez. Aynı belirti, yazmaç sayfası önbelleklenebilir eşlendiğinde de görülür. İki önlem birlikte gerekir: `volatile` işaretçi ve önbelleklenemez eşleme.

11.3 Programlanmış G/Ç

11.3.1 Yoklama ve el sıkışma

CPU ile denetleyicinin bir baytı nasıl devrettiğini en yalın biçimiyle inceleyelim. İki taraf, durum ve komut yazmaçlarındaki birkaç bit üzerinden anlaşır. Bu anlaşmaya **el sıkışma** (*handshaking*) denir.

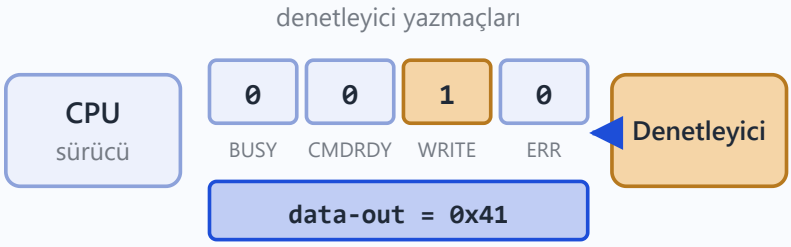
Tanım 11.5

Yoklama (*polling*), CPU'nun aygıtın durum yazmacını aygıt hazır olana kadar tekrar tekrar okumasıdır. Bekleme süresince CPU başka iş yapmadığı için buna **meşgul bekleme** (*busy waiting*) de denir. Verinin her baytını CPU'nun kendisinin taşıdığı bu yönteme **programlanmış G/Ç** (*programmed I/O*, PIO) adı verilir.

```
while (status & BUSY)
    ;          /* meşgul bekleme */
command = WRITE;
data_out = 'A';
command |= CMDRDY;
/* --- denetleyici tarafı --- */
/* CMDRDY görüldü → BUSY = 1 */
/* komutu oku, baytı aygıta yaz */
/* CMDRDY = ERR = BUSY = 0 */
```

Kodun üst yarısını sürücü, alt yarısındaki yorumlarla anlatılan adımları denetleyici yürütür. Bir baytın, 'A' karakterinin (0x41) yazılması sekiz adımda biter:

1. **Bekleme.** Sürücü durum yazmacını okur. Denetleyici önceki işi bitirmediği için BUSY = 1'dir ve `while` döngüsü döner. CPU bu sırada başka iş yapamaz.
2. **Çıkış.** BUSY 0 olunca döngüden çıkılır. Bu bekleme birkaç döngüden milisaniyelere kadar sürebilir; süre aygıta bağlıdır.
3. **Komut.** Komut yazmacına WRITE biti yazılır: yapılacak iş yazmadır.
4. **Veri.** Gönderilecek bayt (0x41) veri çıkış yazmacına konur.
5. **Komut hazır.** CMDRDY = 1 yapılır. Bu bit denetleyiciye işin başlayabileceğini söyler. Sıra önemlidir: CMDRDY, komut ve veri yerine konduktan sonra kurulmalıdır.
6. **Denetleyici meşgul.** Denetleyici CMDRDY'yi görür ve BUSY = 1 yapar. Böylece ana makineye meşgul olduğunu bildirir.
7. **İş.** Denetleyici komut yazmacını okur (WRITE), veri çıkış yazmacındaki baytı alır ve aygıta gönderir.
8. **Bitiş.** CMDRDY, ERR ve BUSY temizlenir. ERR = 0, işin başarıyla bittiğini gösterir. Sonraki bayt için döngü baştan başlar.



Şekil 11.7: El sıkışmanın sonu: BUSY = 0, CMDRDY = 0, ERR = 0. Veri çıkış yazmacında son gönderilen bayt (0x41) durur.

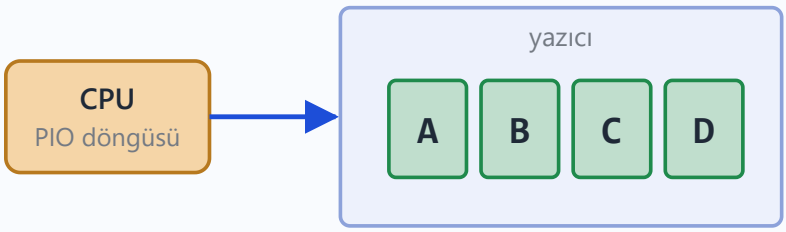
Programlanmış G/Ç basittir ve ek donanım istemez. Ama CPU bekleme boyunca boşa döner. Daha kötüsü, CPU başka bir sürece geçer ve bir baytı zamanında almazsa veri kaybolabilir. Denetleyicinin tamponu taşarsa yeni gelen baytın konacak yeri kalmaz.

11.3.2 Yazıcı örneği

Tanenbaum'un klasik örneği, bir kullanıcı sürecinin “ABCD” dizisini yazıcıya göndermesidir. Sistem çağrısının çekirdekteki kısmı şöyledir:

```
copy_from_user(buffer, p, count);
for (i = 0; i < count; i++) {
    while (*printer_status_reg != READY)
        ;   /* meşgul bekleme */
    *printer_data_register = p[i];
}
return_to_user();
```

İlk satır kullanıcının dizisini çekirdek tamponuna kopyalar (`copy_from_user`). Çekirdek kullanıcı belleğini doğrudan kullanmaz, çünkü süreç o belleği başka bir iş parçacığıyla değiştirebilir ya da adres geçersiz olabilir. Ardından her karakter için iki iş yapılır. İç `while` döngüsü durum yazmacı READY olana kadar döner. Yazıcı hazır olunca karakter veri yazmacına yazılır; yazıcı karakteri basar ve yeniden meşgul olur. Dört karakterin dördü de basıldıktan sonra sürece dönülür.



Şekil 11.8: PIO döngüsünün sonu: yazıcı A, B, C, D karakterlerini bastı. CPU bütün süre boyunca bu sürece bağlı kaldı.

Örnek 11.2

Yazıcı saniyede 100 karakter basıyor. PIO ile her karakter için CPU ne kadar bekler? 2 GHz'lik bir CPU bu sürede kaç tur harcar?

1. **Karakter süresi.** 1 s / 100 = 10 ms. CPU her karakterden sonra yaklaşık 10 ms durum yazmacını yoklar.
2. **Boşa giden tur.** 10 ms × 2 × 10^9 tur/s = 2 × 10^7 tur. CPU bu sürede on milyonlarca komut çalıştırabilirdi.
3. **Sonuç.** Dört karakterlik iş yaklaşık 40 ms sürer ve tamamen yazıcının hızıyla sınırlıdır. CPU'nun hızlı olması hiçbir şeyi değiştirmez.

Yine de PIO her zaman kötü değildir. Gömülü bir sistemde CPU'nun başka işi yoksa ya da aygıt çok hızlı yanıt veriyorsa yoklama en basit ve en hızlı çözümdür. Aşağıdaki örnek, yoklamanın ne zaman zararsız, ne zaman imkânsız olduğunu sayılarla gösterir.

Örnek 11.3

2 GHz'lik bir CPU'da bir yoklama (durumu oku, karşılaştır, veriyi al) 600 tur sürüyor. Saniyede 100 kez yoklanan bir klavye ile 50 MB/s veri üreten ve her yoklamada 32 bayt veren bir aygıt için CPU zamanının yüzde kaçı gider?

1. **Oran.** CPU payı = (yoklama/s × tur/yoklama) / (tur/s).
2. **Klavye.** $100 \times 600 = 60.000$ tur/s; $60.000 / (2 \times 10^9) = \%0{,}003$. Önemsizdir.
3. **Hızlı aygıt.** $50 \times 10^6 / 32 \approx 1{,}56 \times 10^6$ yoklama/s. Bu, $1{,}56 \times 10^6 \times 600 \approx 9{,}4 \times 10^8$ tur/s eder; CPU zamanının yaklaşık %47'si yalnız yoklamaya gider.
4. **Yorum.** Seyrek olaylı yavaş aygıtta yoklama zararsızdır. Yüksek hızlı blok aktarımında ise CPU'nun her parçayla tek tek ilgilenmesi kabul edilemez; çözüm DMA'dır.

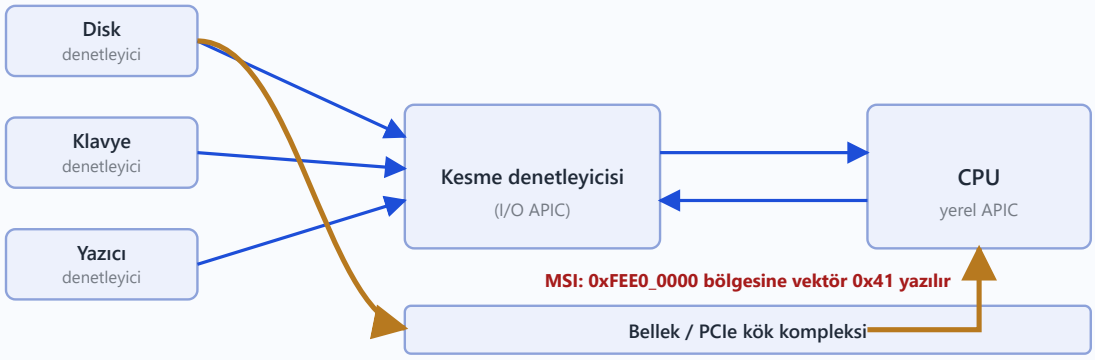
11.4 Kesmeler

11.4.1 Kesme donanımı: hattan MSI'ya

Yoklamanın kusuru, CPU'nun aygıtı sürekli sorgulamasıdır. Daha iyi bir düzen, işi bitince aygıtın CPU'ya haber vermesidir.

Tanım 11.6

Kesme (*interrupt*), bir aygıtın CPU'ya “dikkatini iste” sinyali göndermesidir. CPU o anki komutu bitirir, çalışan programı askıya alır ve aygıta ait **kesme işleyicisini** (*interrupt handler*) çalıştırır. İşleyici bitince askıya alınan programa dönülür.



Şekil 11.9: Kesme donanımı. Aygıt denetleyicileri kesme denetleyicisine (I/O APIC), o da çekirdeğin yerel APIC'ine bağlıdır. Modern PCIe aygıtı ise kablo kullanmaz: 0xFEE0_0000 bölgesine vektör numarasını (burada 0x41) içeren bir bellek yazması gönderir.

Klasik düzende her aygıtın kesme denetleyicisine giden bir hattı (IRQ) vardır. Denetleyici CPU'nun tek kesme girişini aygıtlar arasında paylaştırır ve öncelik belirler. Bir disk okuması şu sırayla haber verilir:

1. **Sinyal.** Disk denetleyicisi işini bitirince kendi kesme hattına sinyal verir.
2. **İletme.** Kesme denetleyicisi o anda başka kesme işlenmiyorsa ya da yeni kesme daha öncelikliyse CPU'ya kesme iletir. Değilse istek bekletilir; CPU onaylayana kadar hat etkin kalır.
3. **Vektör.** CPU kesmeyi kabul eder ve denetleyici bir vektör numarası verir. CPU bu sayıyı kesme vektör tablosunda indis olarak kullanır ve işleyicinin adresini bulur.

Bu anlatım tarihsel PC düzenine dayanır: iki 8259 PIC ve 15 IRQ hattı. Denetleyicinin vektörü "adres yoluna koyması" da yalnız o düzen için geçerlidir. Bugün x86'da bir I/O APIC ve her çekirdekte bir yerel APIC (x2APIC) bulunur; ARM'da bu görevi GIC üstlenir.

Tanım 11.7

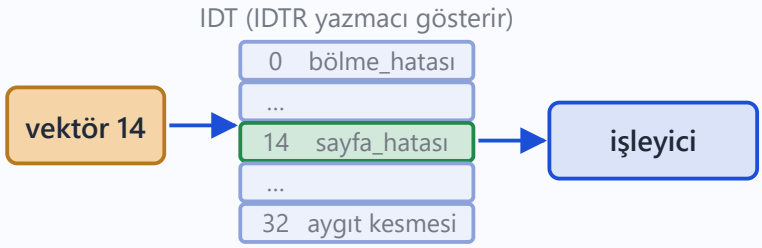
MSI (*message-signaled interrupts*) ve onun genişletilmiş biçimi **MSI-X**, kesmeyi bir kablo yerine bellek yazmasıyla iletir. Aygıt özel bir adres bölgesine (x86'da 0xFEE0_0000) vektör numarasını içeren bir yazma gönderir; hedef çekirdeğin yerel APIC'i bunu kesmeye çevirir.

MSI'nın iki üstünlüğü vardır. Hat paylaşımı ortadan kalkar, çünkü her aygıt istediği kadar ayrı vektör alabilir. Kesme ayrıca istenen çekirdeğe yönlendirilebilir. MSI-X bir PCIe işlevine 2048 vektöre kadar izin verir. NVMe diskler ve ağ kartları her kuyruk için ayrı vektör alır ve o kuyruğun kesmesini kuyruğu kullanan çekirdeğe gönderir.

11.4.2 Vektörler, öncelik, NMI ve istisnalar

Tanım 11.8

Kesme vektörü (*interrupt vector*), kesme vektör tablosundaki bir indistir. Tablonun her girdisi bir işleyicinin adresini tutar. x86'da bu tabloya IDT denir; 256 girdisi vardır ve yerini IDTR yazmacı gösterir.



Şekil 11.10: Vektör 14 (sayfa hatası) IDT'nin 14. girdisini seçer; girdi işleyicinin adresini verir.

Vektörlerin bir kısmı aygıtlara değil, işlemcinin kendisine aittir. x86'da 0–31 arası işlemci istisnalarına ayrılmıştır (Tablo 11.4).

Vektör	Olay	Tür
0	Bölme hatası	hata (*fault*)
1 / 3	Hata ayıklama / kesme noktası	tuzak (*trap*)
2	NMI (maskelenemez kesme)	kesme
6	Geçersiz işlem kodu	hata
13	Genel koruma hatası	hata
14	Sayfa hatası	hata
18	Makine denetimi	iptal (*abort*)
32–255	Aygıt kesmeleri, IPI, APIC zamanlayıcısı	maskelenebilir

Tablo 11.4: x86'da bazı kesme vektörleri.

Tanım 11.9

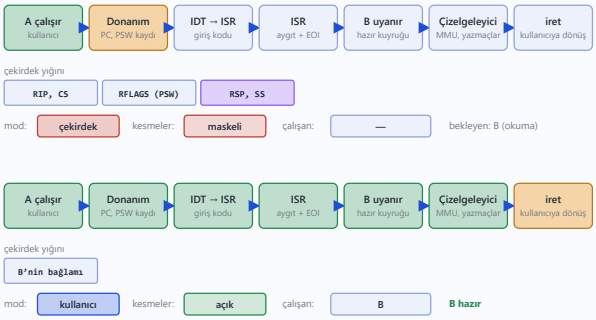
İstisna (*exception*), çalışan komutun kendisinin yol açtığı, eş zamanlı bir olaydır. İki türü öne çıkar. **Hata** (sayfa hatası gibi) komutun tamamlanmasını engeller; işleyiciden sonra aynı komut yeniden çalıştırılır. **Tuzak** (*trap*; sistem çağrısı, `int3`) bilerek tetiklenir; işleyiciden sonra bir sonraki komuttan devam edilir.

Kesmelerin de kendi aralarında farkları vardır. Maskelenebilir kesmeler kritik bir bölgede ertelenebilir; x86'da `cli` komutu kesmeleri kapatır, `sti` yeniden açar. NMI (*non-maskable interrupt*) ise ertelenemez. Donanım hataları, bellek ECC hataları ve izleme köpeği (*watchdog*) bu yolu kullanır, çünkü bu olaylar beklemeye tahammül etmez. Öncelik de önemlidir. APIC kesmeleri vektör sınıfına göre sıralar; yüksek öncelikli bir kesme, düşük öncelikli bir işleyiciyi kesebilir.

Aygıt sayısı vektör sayısından fazla olduğunda bir vektör birden çok aygıta düşebilir. Bu durumda tablonun girdisi tek bir işleyiciyi değil, bir işleyici listesini gösterir. Kesme geldiğinde listedeki işleyiciler sırayla "bu kesme benim mi?" diye kendi aygıtına bakar. Bu düzene **kesme zinciri** ya da paylaşılan IRQ denir. Linux'ta hangi vektörün hangi çekirdekte kaç kez geldiği `/proc/interrupts` dosyasında görülür; `irqbalance` hizmeti kesmeleri çekirdeklere dağıtır.

11.4.3 Kesme işleme adım adım

Bir kesmenin baştan sona nasıl işlendiğini somut bir durumda izleyelim. Süreç A çalışıyor; süreç B ise bir disk okumasının bitmesini bekliyor. Disk okuması biter ve denetleyici kesme ister.



Şekil 11.11: Kesme işleme akışının iki anı. Üstte donanım bağlamı kaydediyor; altta `iret` ile seçilen sürece dönülüyor.

1. **Kabul.** CPU o anki komutu bitirir, sonra kesmeyi kabul eder. Kesme ancak komut sınırında alınır.
2. **Donanımın kaydı.** Donanım program sayacını (x86-64'te RIP), durum sözcüğünü (PSW, RFLAGS) ve yığın işaretçisini çekirdek yığınına iter. Çekirdek moduna geçer ve kesmeleri maskeler.
3. **İşleyiciyi bulma.** Vektör numarası IDT'de indis olur ve işleyicinin adresi bulunur. Giriş kodu (assembly) donanımın kaydetmediği yazmaçları da kaydeder. Artık C ile yazılmış kesme hizmet yordamı (ISR) için bağlam ve yığın hazırdır.
4. **ISR.** İşleyici aygıtın durumunu okur, veriyi alır ve kesme denetleyicisine onay (EOI, *end of interrupt*) gönderir. Bundan sonra yeni kesmeler gelebilir. Uzun sürecek iş daha sonraya, alt yarıya ertelenir.
5. **Uyandırma.** Bekleyen süreç B, bloke durumundan hazır kuyruğuna alınır.
6. **Çizelgeleme.** Çizelgeleyici sıradaki süreci seçer; burada B'yi seçtiğini varsayalım. B'nin sayfa tablosu MMU'ya yüklenir (x86'da CR3 yazmacı), yazmaçları süreç tablosundan ya da yığından geri alınır.

7. **Dönüş.** `iret` komutu kaydedilen PC ve PSW'yi geri yükler, kullanıcı moduna döner ve kesmeleri açar. B, kaldığı komuttan devam eder.

Bu sıra Tanenbaum'un kesme işleyicisi adımlarını izler. Çizelgeleyici A'yı da seçebilirdi; bu seçim politikaya bağlıdır. Kesme, bekleyen bir süreci hazır yapmak dışında çizelgeleme kararını kendiliğinden değiştirmez.

Sık yapılan hata

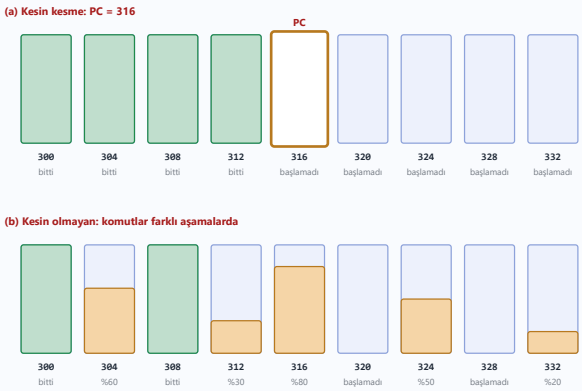
Kesme işleyicisinin, durumu kaydetmeden önce çalıştığını sanmak. İşleyici A'nın yazmaçlarını kullanacağı için önce donanım dönüş adresini ve durum sözcüğünü, sonra giriş kodu öteki yazmaçları kaydetmelidir. Kayıt yapılmasaydı dönüş adresi kaybolurdu ve A kaldığı yerden devam edemezdi. Doğru sıra: aygıt ister, CPU komut sınırında durur, durum kaydedilir, vektörle işleyici bulunur, işleyici aygıtı onaylar ve bekleyeni uyandırır, çizelgeleyici seçer, `iret` ile dönülür.

11.4.4 Kesin ve kesin olmayan kesmeler

Kesmeden sonra programın kaldığı yerden devam edebilmesi için "kaldığı yer"in iyi tanımlanmış olması gerekir. Boru hattıyla çalışan işlemcide bu kendiliğinden sağlanmaz.

Tanım 11.10

Bir kesme şu dört koşulu sağlıyorsa **kesindir** (*precise interrupt*): (1) PC bilinen bir yere kaydedilmiştir; (2) PC'den önceki bütün komutlar tamamen bitmiştir; (3) PC'den sonraki komutların hiçbiri başlamamıştır; (4) PC'deki komutun durumu bilinir. Bu koşulları sağlamayan kesmeye **kesin olmayan** kesme denir.



Şekil 11.12: (a) Kesin kesme: PC = 316; 300–312 adresindeki komutlar bitmiş, 316 ve sonrası başlamamış. (b) Kesin olmayan kesme: komutlar farklı aşamalarda kalmış (304 %60, 312 %30, 316 %80 …).

Boru hattında aynı anda birçok komut farklı aşamalardadır. Üstün ölçekli bir işlemcide komutlar sıra dışı da tamamlanabilir. Şekil 11.12'nin (b) kısmındaki durumda tek bir PC değeri programın nerede kaldığını anlatmaz. Yeniden başlamak için ya donanım yarım işleri geri almalı ya da işletim sistemi çok miktarda iç durumu kaydedip çözümlemelidir. İkincisi işleyiciyi hem yavaşlatır hem karmaşıklaştırır.

Modern x86, ARM ve RISC-V işlemcileri kesin kesme sağlar. Bunu **yeniden sıralama arabelleği** (ROB, *reorder buffer*) ile yapar. Komutlar sıra dışı yürütülse de ROB onları program sırasıyla "emekliye ayırır", yani sonuçlarını kalıcı hâle getirir. Kesme geldiğinde henüz emekliye ayrılmamış komutların etkileri atılır. Bedeli yonga alanı ve tasarım karmaşıklığıdır, ama işletim sistemi bu sayede sade kalır.

11.4.5 Kesme güdümlü G/Ç: yazıcı örneği

Kesmelerle yazıcı örneği baştan yazılabilir. Amaç, CPU'nun yazıcıyı beklerken boşa dönmemesidir.

> **Tanım 11.11**
>
> **Kesme güdümlü G/Ç** (*interrupt-driven I/O*) yönteminde CPU aygıta bir iş verir ve başka bir süreci çalıştırmaya gider. Aygıt işi bitirince kesme üretir; kesme işleyicisi sonraki işi verir ya da bekleyen süreci uyandırır.

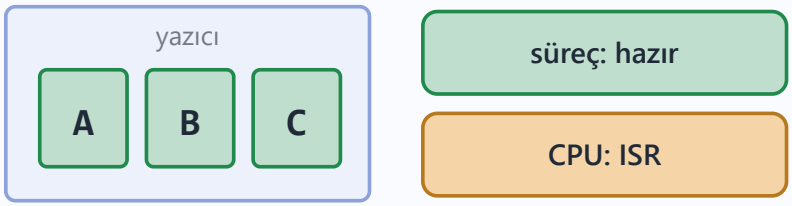
```
/* sistem çağrısı */
copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
i = 1; count = count - 1;
scheduler();      /* süreç bloke */
/* kesme hizmet yordamı */
if (count == 0) {
    unblock_user();
} else {
    *printer_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
acknowledge_interrupt();
return_from_interrupt();
```

Kod iki parçadan oluşur. Üst kısım sistem çağrısıdır. Diziyi çekirdek tamponuna kopyalar, yazıcının kesmelerini açar, yazıcı hazır olunca yalnız ilk karakteri yazar, sayaçları günceller ve çizelgeleyiciyi çağırır. Süreç bloke olur ve CPU başka bir süreci çalıştırır. Alt kısım kesme hizmet yordamıdır. Yazıcı her

karakteri bastığında bir kesme gelir. Yordam `count`'a bakar: sıfırsa bütün karakterler basılmıştır ve kullanıcı süreci uyandırılır. Değilse sıradaki karakter yazılır, `count` bir azalır, `i` bir artar. Her iki durumda kesme onaylanır ve kesilen sürece dönülür.

Olay	Yazılan	`i`	`count`	Süreç
Sistem çağrısı	p[0] = 'A'	1	2	bloke
1. kesme ('A' basıldı)	p[1] = 'B'	2	1	bloke
2. kesme ('B' basıldı)	p[2] = 'C'	3	0	bloke
3. kesme ('C' basıldı)	—	3	0	hazır (`unblock_user`)

Tablo 11.5: "ABC" dizisinin kesme güdümlü yazdırılması (başta `count` = 3).



Şekil 11.13: Üçüncü kesmeden sonra: A, B, C basıldı; ISR kullanıcı sürecini hazır kuyruğuna aldı.

Tablo 11.5 bir ayrıntıyı ortaya koyar: üç karakter için üç kesme gelir, ama üçüncü kesme yeni bir karakter yazmaz. Yalnız işin bittiğini bildirir. Sistem çağrısı ilk karakteri yazdığı için kesmelerin sayısı karakter sayısına eşittir. Yöntemin bedeli kesme başına bağlam kaydı ve işleyici süresidir; bu da tipik olarak 1–2 µs'dir. Yazıcı beklerken CPU başka süreçleri çalıştırır; bu kazanç, birkaç mikrosaniyelik kesme maliyetinden çok büyüktür.

11.4.6 Üst yarı, alt yarı ve hibrit yoklama

Kesme işleyicisi çalışırken en azından aynı hattaki kesmeler kapalıdır. İşleyici uzun sürerse başka aygıtların kesmeleri gecikir, hatta kaybolur. Bu yüzden işletim sistemleri işleyiciyi ikiye böler.

Tanım 11.12

Üst yarı (*top half*, Linux'ta *hardirq*) asıl kesme işleyicisidir: kısa tutulur, aygıtı onaylar, veriyi bir tampona alır ve geri kalan işi planlar. **Alt yarı** (*bottom half*) ertelenmiş iştir ve kesmeler açıkken çalışır. Linux'ta softirq, tasklet, workqueue ya da iş parçacıklı IRQ (`request_threaded_irq`) biçiminde, Windows'ta ISR + DPC (*deferred procedure call*) biçiminde bulunur.

Kesme sayısını azaltmanın başka yolları da vardır.

- **Kesme birleştirme** (*interrupt coalescing*). Ağ kartları ve NVMe diskler birkaç tamamlanmayı tek bir kesmede bildirir. Kesme sayısı azalır, gecikme biraz artar.
- **NAPI: kesme ile yoklamanın birleşimi.** Yoğun ağ trafiğinde Linux sürücüsü kesmeyi kapatır ve kuyruğu yoklar. Trafik azalınca kesmeye döner. Seyrek olayda kesme, sık olayda yoklama verimlidir; hibrit yöntem ikisinin iyi yanını alır.
- **Saf yoklama.** Olay hızı çok yüksek ve gecikme kritikse kesme hiç kullanılmaz. DPDK ve SPDK, NVMe yoklamalı kuyrukları ve io_uring'in IOPOLL kipi bu yolu seçer. Bir çekirdek tamamen yoklamaya ayrılır.

Kesmelerin bir de arıza durumu vardır. Hatalı bir aygıt sürekli kesme üretirse işlemci yalnız işleyiciyi çalıştırır ve sistem kilitlenir. Buna **kesme fırtınası** denir. Linux, hiçbir işleyicinin sahiplenmediği kesmeler çok sayıda gelirse “nobody cared” iletisiyle hattı kapatır; kesme hızını sınırlayan mekanizmalar da vardır.

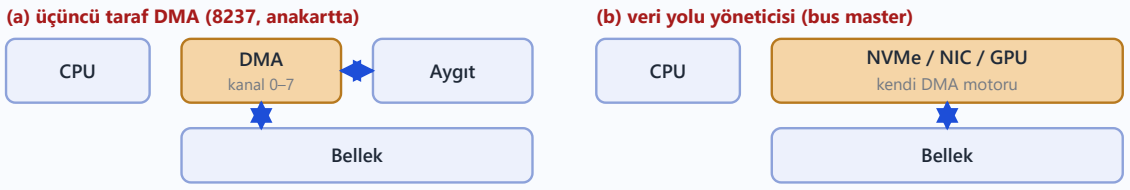
11.5 Doğrudan bellek erişimi (DMA)

11.5.1 DMA denetleyicisi

Kesme güdümlü G/Ç beklemeyi ortadan kaldırır, ama her bayt ya da sözcük için hâlâ bir kesme gelir. 4 KiB'lık bir disk bloğunu sözcük sözcük taşımak binlerce kesme demektir. Çözüm, taşıma işini CPU'dan tamamen almaktır.

> **Tanım 11.13**
>
> **Doğrudan bellek erişimi** (DMA, *direct memory access*), bir aygıt ile bellek arasında blok aktarımını CPU'nun katılımı olmadan yapan donanım düzenidir. CPU yalnız aktarımı kurar ve aktarım bitince gelen tek bir kesmeyle ilgilenir.

Bir DMA denetleyicisinin yazmaçları aktarımı tarif eder: bellek adresi, bayt sayısı ve denetim yazmacı. Denetim yazmacı G/Ç kapısını, yönü (okuma mı, yazma mı), aktarım birimini (bayt mı, sözcük mü) ve bir seferde kaç birim taşınacağını (patlama boyutu) belirler. Çok kanallı bir denetleyici her kanal için ayrı bir yazmaç takımı tutar ve aynı anda birçok aktarımı yönetir.



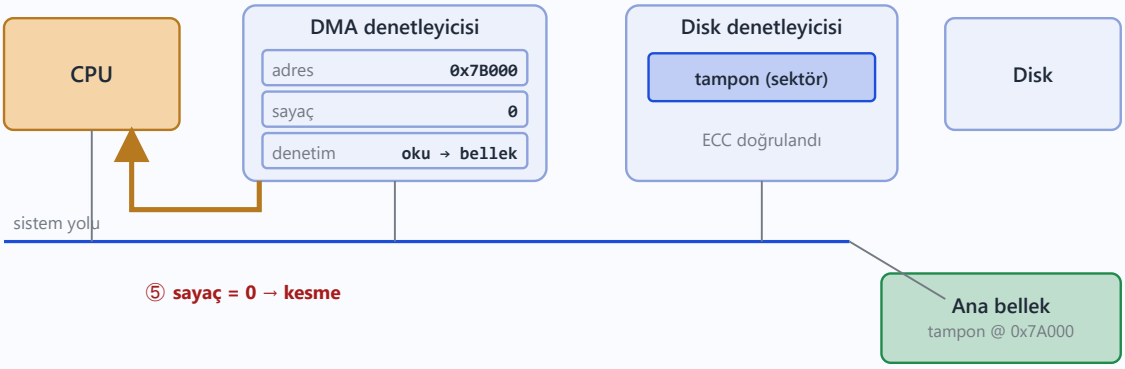
Şekil 11.14: (a) Üçüncü taraf DMA: anakartta tek bir DMA denetleyicisi (8237) aygıtlar adına aktarım yapar. (b) Veri yolu yöneticisi (*bus master*): NVMe disk, ağ kartı ya da GPU kendi DMA motoruyla belleğe doğrudan

yazar.

Anakartta tek bir DMA denetleyicisinin bütün aygıtlara hizmet ettiği düzen yalnız tarihsel 8237 yongası için geçerlidir (Şekil 11.14 (a)). Bugün aygıtların çoğunda kendi DMA motoru vardır ve bu aygıtlar belleğe doğrudan PCIe yazması yapar. Kavram aynı kalır: aktarımı kuran CPU'dur, taşıyan aygıttır.

11.5.2 DMA aktarımı adım adım

Bir disk sektörünün DMA ile belleğe okunmasını izleyelim. Sektör 4096 bayttır ve bellekte 0x7A000 adresine yazılacaktır.



Şekil 11.15: DMA aktarımının son adımı: sayaç 0'a indi, DMA denetleyicisi CPU'ya kesme gönderiyor. Veri zaten bellekte.

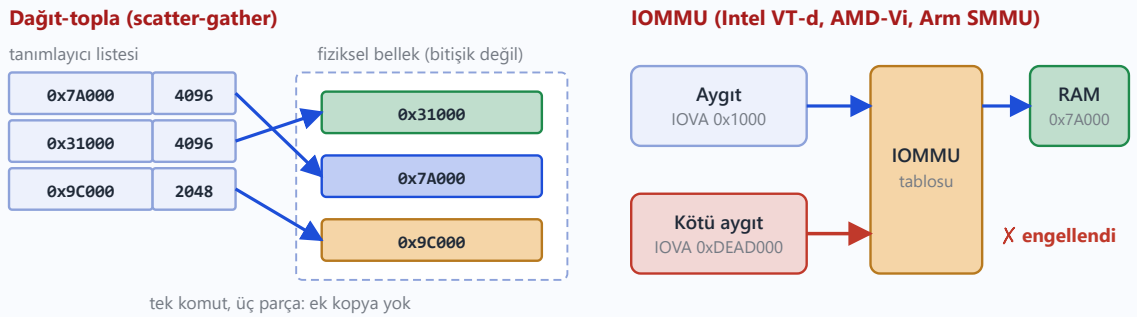
1. **Kurulum.** CPU, DMA denetleyicisini (adres = 0x7A000, sayaç = 4096, yön = okuma) ve disk denetleyicisini ("sektörü oku") programlar, sonra başka işe döner.
2. **Tampon.** Disk denetleyicisi sektörü kendi tamponuna okur ve ECC ile doğrular. Tampon sayesinde yol o an meşgulse veri kaybolmaz.
3. **İstek.** DMA denetleyicisi yolu ister, bellek adresini yola koyar ve disk denetleyicisine okuma isteği gönderir.
4. **Yazma.** Disk denetleyicisi sözcüğü doğrudan belleğe yazar. CPU bu aktarımın içinde değildir.
5. **Onay.** Disk denetleyicisi onay gönderir; DMA denetleyicisi adresi artırır, sayacı azaltır. Sayaç 0 olana kadar istek, yazma ve onay adımları tekrarlanır.
6. **Kesme.** Sayaç 0'a inince DMA denetleyicisi CPU'yu kesmeyle uyarır. İşletim sistemi kesmeyi işlediğinde veri zaten bellektedir.

DMA'nın veri yolunu nasıl kullandığı da önemlidir. **Döngü çalma** (*cycle stealing*) kipinde denetleyici her sözcük için yolu birkaç turluğuna CPU'dan alır. CPU tamamen durmaz, yalnız belleğe erişmek istediği anlarda biraz bekler. **Patlama** (*burst*) kipinde yol bir blok boyunca tutulur; aktarım hızlanır ama

CPU daha uzun bekleyebilir. Bir ayrım daha vardır. **Geçişsiz** (*fly-by*) kipte veri aygıttan belleğe doğrudan gider. Öteki kipte DMA denetleyicisi sözcüğü önce kendisi okur, sonra hedefe yazar; bu iki yol döngüsü harcar ama bellekten belleğe kopyalamaya da izin verir.

11.5.3 Dağıt-topla ve IOMMU

Sanal bellek kullanan bir sistemde bir sürecin 12 KiB'lık tamponu sanal adreslerde bitişiktir, ama fiziksel bellekte üç ayrı sayfaya dağılmış olabilir. Basit bir DMA denetleyicisi yalnız tek bir bitişik bölgeye yazabilir. Bu durumda çekirdek ya veriyi bitişik bir tampona alıp sonra kopyalamalı ya da üç ayrı aktarım yapmalıdır.



Şekil 11.16: Solda dağıt-topla: tanımlayıcı listesi üç parçayı (0x7A000/4096, 0x31000/4096, 0x9C000/2048) tek komutta verir. Sağda IOMMU: aygıtın gördüğü adres (IOVA) tablo üzerinden fiziksel adrese çevrilir; tabloda olmayan adrese yapılan erişim engellenir.

> **Tanım 11.14**
>
> **Dağıt-topla** (*scatter-gather*) DMA'da aygıta bir (adres, uzunluk) çiftleri listesi verilir; aygıt bitişik olmayan sayfalara tek işlemde yazar ya da onlardan okur. **IOMMU**, aygıtın kullandığı adresleri (IOVA) bir sayfa tablosu üzerinden fiziksel adreslere çeviren birimdir.

Dağıt-topla, Linux'ta `readv` / `writev` çağrılarının, parçalı ağ paketlerinin ve NVMe'deki PRP ve SGL listelerinin temelidir. Ek kopya gerekmez. IOMMU'nun ise üç görevi vardır. Birincisi korumadır: aygıt yalnız kendisine eşlenmiş sayfalara erişebilir. Thunderbolt gibi dışarıdan takılan bir aygıtın bütün belleği okumaya çalıştığı DMA saldırıları bu sayede engellenir. İkincisi sanallaştırmadır: bir aygıt doğrudan bir sanal makineye atanabilir (VFIO); konuk işletim sistemi kendi "fiziksel" adreslerini verir, IOMMU bunları gerçek adreslere çevirir. Üçüncüsü, yalnız 32 bit adresleyebilen eski aygıtların 4 GiB'nin üstündeki belleği kullanabilmesidir. Intel'de bu birimin adı VT-d, AMD'de AMD-Vi, ARM'da SMMU'dur.

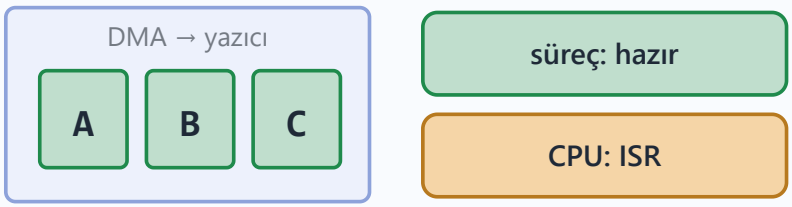
11.5.4 DMA ile yazıcı örneği

Yazıcı örneğinin üçüncü sürümü DMA kullanır:

```
/* sistem çağrısı */
copy_from_user(buffer, p, count);
set_up_DMA_controller();
scheduler();     /* süreç bloke */

/* kesme hizmet yordamı */
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

Sistem çağrısı veriyi çekirdek tamponuna kopyalar, DMA denetleyicisini programlar (tampon adresi, sayaç = 3, hedef olarak yazıcı kapısı) ve süreci bloke eder. DMA denetleyicisi karakterleri CPU'suz olarak yazıcıya iletir. Bu sırada CPU'ya hiç kesme gelmez. Bütün karakterler aktarılınca DMA tek bir kesme üretir. Kesme hizmet yordamı yalnız iki iş yapar: kesmeyi onaylar ve kullanıcı sürecini uyandırır. Karakter başına değil, tampon başına bir kesme gelir.



Şekil 11.17: DMA ile yazdırmanın sonu: üç karakter aktarıldı, tek kesmeyle süreç uyandırıldı.

DMA her zaman kazandırmaz. DMA denetleyicisi genellikle CPU'dan yavaştır. Aygıt çok yavaşsa ya da CPU'nun yapacak başka bir işi yoksa, kesme güdümlü G/Ç ya da PIO daha basit ve yeterli olabilir. Tanenbaum bu noktayı özellikle vurgular: DMA'nın kazancı, CPU'nun boşalan zamanda yararlı iş yapabilmesine bağlıdır.

11.5.5 Üç yöntemin maliyeti

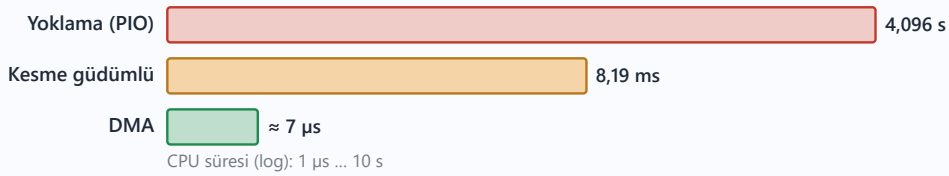
Üç yöntemi aynı iş üzerinde sayılarla karşılaştıralım.

Örnek 11.4

Yazıcı saniyede 1000 karakter basıyor; 4096 karakter yazdırılacak. Kesme işleme maliyeti 2 µs, DMA kurulumu 5 µs. Her yöntemde CPU bu iş için ne kadar zaman harcar?

1. **Yoklama.** CPU her karakteri bekler. Karakter başına 1 ms olduğundan 4096 × 1 ms = 4,096 s boyunca CPU meşguldür. Programlanmış G/Ç'de CPU'nun harcadığı süre aygıtın hızına bağlıdır.

2. **Kesme güdümlü.** Karakter başına bir kesme gelir: 4096 × 2 µs = 8192 µs ≈ 8,19 ms. Bu, işin toplam süresinin (4,096 s) %0,2'sidir. Bekleme ortadan kalkar; bedel kesme sayısıdır.
3. **DMA.** 5 µs kurulum ve 1 kesme × 2 µs: toplam ≈ 7 µs. Kesme güdümlü yönteme göre 8192 / 7 ≈ 1170 kat daha az CPU zamanı harcanır.
4. **Yorum.** Yoklama ile DMA arasında yaklaşık altı basamak (4,096 s / 7 µs ≈ 585.000 kat), kesme güdümlü ile DMA arasında üç basamaktan fazla fark vardır (Şekil 11.18). DMA yine de veri yolundan döngü çalar ve kurulum maliyeti küçük aktarımlarda kendini geri ödemez.



Şekil 11.18: 4096 karakterlik yazdırmada üç yöntemin CPU süresi logaritmik eksende: yoklama 4,096 s, kesme güdümlü 8,19 ms, DMA ≈ 7 µs.

Örnek 11.5

Bir disk denetleyicisinden 512 KiB veri okunuyor. Kesme güdümlü yöntemde her 512 baytlık sektör için bir kesme geliyor ve işleyici sektörü belleğe kopyalayarak 5 µs harcıyor. DMA'da kurulum 8 µs, aktarım sonundaki tek kesme 5 µs. CPU yükleri nedir?

1. **Sektör sayısı.** 512 KiB = 524.288 B; 524.288 / 512 = 1024 sektör.
2. **Kesme güdümlü.** 1024 × 5 µs = 5120 µs ≈ 5,1 ms.
3. **DMA.** 8 + 5 = 13 µs. CPU yükü 5120 / 13 ≈ 394 kat azalır.
4. **Kenar durum.** Aynı sistemde tek bir 512 baytlık sektör okunursa kesme güdümlü yöntem 5 µs, DMA 13 µs harcar. Sabit kurulum maliyeti küçük aktarımda kazancı siler.

Bu iki örnek kuralı açık eder. Aktarım büyüdükçe DMA'nın sabit maliyeti önemsizleşir ve CPU yükü aktarım boyundan bağımsız hâle gelir. Aktarım küçüldükçe, aygıt yavaşladıkça ya da CPU boşta bekledikçe DMA'nın kazancı kaybolur. Kesme, DMA ve G/Ç Akışı Simülatörü üç yöntemi aynı aktarımla çalıştırır ve CPU kullanımlarını yan yana gösterir.

11.6 G/Ç yazılımı

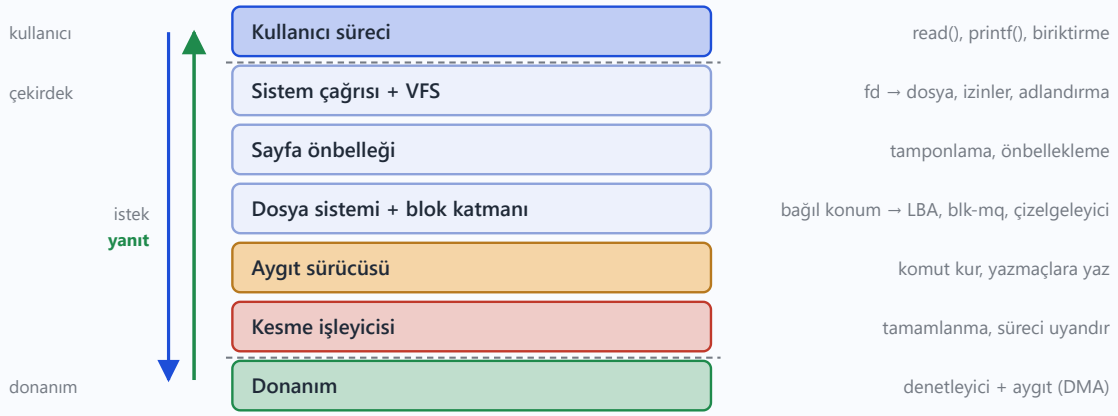
11.6.1 G/Ç yazılımının hedefleri

Donanımın bu kadar çeşitli olması, yazılımın düzenli olmasını zorunlu kılar. G/Ç yazılımı tasarlanırken altı hedef gözetilir.

- **Aygıt bağımsızlığı.** Program hangi aygıtla çalıştığını bilmeden çalışabilmelidir. `sort < girdi` komutu, `girdi` dosyası SSD'de de, USB bellekte de dursa aynı biçimde çalışır.
- **Tek tip adlandırma.** Bir dosyanın ya da aygıtın adı aygıt türüne bağlı olmamalıdır. `/dev/nvme0n1` ve `/mnt/usb/rapor.txt` aynı ağacın parçalarıdır; farklı aygıtlar bağlama (*mount*) ile tek bir ağaca eklenir.
- **Hata ele alma.** Hata donanıma olabildiğince yakın çözülmelidir. Denetleyici bit hatasını ECC ile düzeltir, sürücü başarısız işlemi yeniden dener. Üst katman ancak gerektiğinde hatadan haberdar olur.
- **Eş zamanlı ve eş zamansız aktarım.** Donanım düzeyinde aktarımların çoğu eş zamansızdır: CPU işi başlatır ve kesmeyi bekler. Kullanıcıya ise bloke eden bir arayüz sunmak programlamayı kolaylaştırır. İşletim sistemi bu ikisi arasında köprü kurar.
- **Tamponlama.** Ağdan gelen bir paket, nereye gideceği belli olmadan önce bir yere konmalıdır. Ses gibi gerçek zamanlı aygıtlar da tampona dayanır.
- **Paylaşılan ve adanmış aygıtlar.** Bir disk aynı anda birçok kullanıcıya hizmet verebilir. Yazıcı ya da tarayıcı ise belli bir anda tek bir kullanıcıya adanır. Adanmış aygıtlar kilitlenme riski doğurur: iki süreç birbirinin tuttuğu aygıtı bekleyebilir.

11.6.2 G/Ç yazılım katmanları

Bu hedeflere ulaşmanın yolu katmanlı bir yapıdır. Her katmanın iyi tanımlı bir görevi ve üst katmana sunduğu bir arayüz vardır.



Şekil 11.19: G/Ç yazılım katmanları: kullanıcı süreci, sistem çağrısı ve VFS, sayfa önbelleği, dosya sistemi ve blok katmanı, aygıt sürücüsü, kesme işleyicisi ve donanım. İstek yukarıdan aşağı, yanıt aşağıdan yukarı akar.

Tanenbaum dört katman sayar: kullanıcı düzeyi G/Ç yazılımı, aygıttan bağımsız yazılım, aygıt sürücüleri ve kesme işleyicileri. Linux'ta "aygıttan bağımsız" kısım kendi içinde birkaç parçaya ayrılır: VFS (sanal dosya sistemi), sayfa önbelleği, dosya sistemi ve blok katmanı (Şekil 11.19). Aşağıdaki örnek, bir isteğin bu katmanlardan nasıl geçtiğini gösterir.

11.6.3 Bir `read()` çağrısının yolu



Şekil 11.20: `read()` isteğinin katmanlardaki yolu. Bu anda SSD veriyi DMA ile önbellek sayfasına yazmış ve MSI-X kesmesi üretmiştir.

Örnek 11.6

Bir süreç NVMe SSD'deki bir dosyadan `read(fd, buf, 4096)` ile 4 KiB okuyor ve istenen sayfa bellekte yok. İstek hangi adımlardan geçer?

1. **Sistem çağrısı.** libc sarmalayıcısı `syscall` komutuyla çekirdeğe geçer.

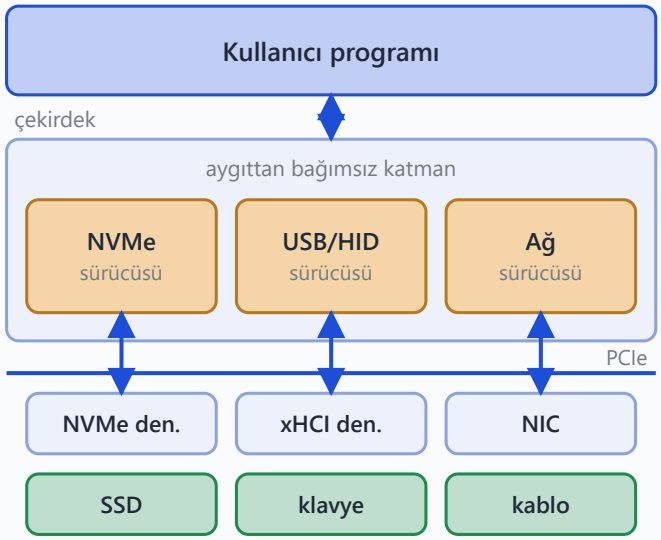
2. **VFS.** Dosya tanımlayıcısı açık dosya nesnesine, açık dosya da i-düğüme (inode) çevrilir. Erişim izni dosya açılırken zaten denetlenmiştir.
3. **Sayfa önbelleği.** İstenen sayfa önbellekte yoktur. Bir önbellek sayfası ayrılır ve okuma başlatılır.
4. **Dosya sistemi ve blok katmanı.** Dosya sistemi dosya bağıl konumunu disk bloğuna (LBA, mantıksal blok adresi) çevirir. Blok katmanı bir istek (bio) oluşturur ve donanım kuyruğuna verir.
5. **Sürücü.** NVMe sürücüsü okuma komutunu gönderim kuyruğuna yazar ve kapı zili (*doorbell*) yazmacına yazarak aygıta haber verir. Süreç bloke olur; Linux'ta durumu D (kesintisiz uyku) görünür.
6. **Aygıt.** SSD veriyi DMA ile önbellek sayfasına yazar, tamamlanma kuyruğuna bir kayıt ekler ve MSI-X kesmesi üretir.
7. **Kesme işleyicisi.** Tamamlanma kaydı işlenir, sayfa "güncel" olarak işaretlenir ve bekleyen süreç hazır olur.
8. **Dönüş.** Süreç yeniden çalışınca VFS veriyi önbellekten kullanıcı tamponuna kopyalar ve çağrı 4096 döndürür.

Aynı veri ikinci kez okunursa üçüncü adımda sayfa önbellekte bulunur. Aygıta hiç gidilmez; işlem mikrosaniyeler içinde biter. Bu fark, işletim sisteminin boş belleği neden önbelleğe ayırdığını da açıklar.

11.6.4 Aygıt sürücüleri

Tanım 11.15

Aygıt sürücüsü (*device driver*), belirli bir aygıta ya da aygıt ailesine özgü kodu kapsülleyen çekirdek bileşenidir. Aygıtın yazmaçlarını, komutlarını ve tuhaflıklarını yalnız sürücü bilir.



Şekil 11.21: Aygıt sürücüleri çekirdekte çalışır ve veri yolu üzerinden denetleyicilerle konuşur.

Bir sürücü, üst katmandan gelen her istek için benzer bir iş sırası izler. Önce parametreleri doğrular. Sonra isteği aygıtın anlayacağı bir komuta çevirir; bir disk için bu, LBA ve kuyruk girdisi demektir. Gerekirse aygıtı başlatır, yazmaçlara yazar ve kesmeyi bekler, yani bloke olur. İş bitince hatayı denetler ve sonucu üst katmana döndürür.

Sürücü yazmanın iki zorluğu vardır. Birincisi, sürücü **yeniden girilebilir** (*reentrant*) olmalıdır: bir isteği işlerken yeni bir istek ya da bir kesme gelebilir. İkincisi, sıcak takılabilen bir aygıt çalışırken ortadan kaybolabilir; sürücü bunu çökmeden karşılamalıdır.

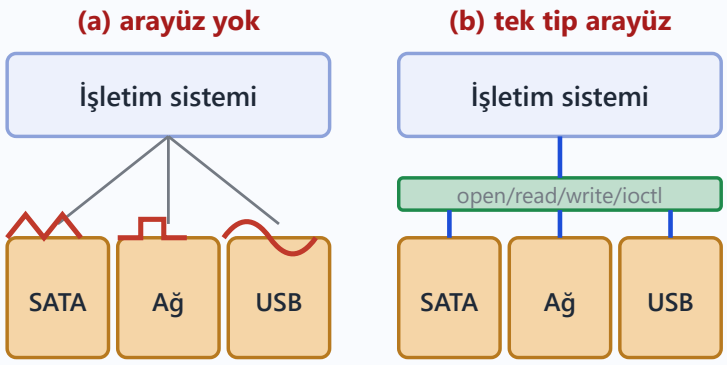
Linux'ta sürücülerin çoğu çekirdek kaynak ağacının içindedir; bazıları üreticiden ayrıca gelir. Sürücüler genellikle yüklenebilir modüllerdir: `modprobe` modülü yükler, `lsmod` yüklü modülleri listeler. Ama çekirdekte çalışan hatalı bir sürücü bütün sistemi çökertebilir. Bu riski azaltmak için sürücüleri kullanıcı alanına taşıyan tasarımlar vardır. MINIX 3'te bütün sürücüler kullanıcı alanındadır. Linux'ta FUSE, VFIO/UIO, DPDK ve SPDK, Windows'ta UMDF bu amaçla kullanılır.

11.6.5 Aygıttan bağımsız G/Ç yazılımı

Sürücünün üstündeki katman, bütün sürücüler için ortak olan işleri üstlenir. Bunların başında tek tip arayüz gelir. Her aygıt sınıfı aynı işlevleri sunar ve çekirdek sürücüyü bir fonksiyon işaretçisi tablosu üzerinden çağırır:

```
struct file_operations nvme_fops = {
  .open           = nvme_open,
  .release        = nvme_release,
  .read_iter      = nvme_read,
  .unlocked_ioctl = nvme_ioctl,
  .poll           = nvme_poll,
};
```

Bu tablo C'de nesne yönelimli programlamanın karşılığıdır. VFS bir dosyada `read` istendiğinde hangi aygıt olduğunu sormaz; dosyaya bağlı tablodaki `read_iter` işaretçisini çağırır. NVMe için bu `nvme_read`'dir, bir seri port için başka bir işlevdir. Yeni bir aygıt eklemek, çekirdeğin geri kalanına dokunmadan yeni bir tablo yazmak demektir.



Şekil 11.22: (a) Standart arayüz yoksa çekirdek her sürücüyü farklı biçimde çağırır. (b) Standart arayüzle bütün sürücüler aynı işlevleri sunar.

Aynı katman başka işleri de yapar. **Adlandırma**: `/dev/nvme0n1` gibi bir ad, ana numara (sürücü) ve yan numara (birim) çiftine çevrilir. **Koruma**: aygıt dosyasının izinleri, kimin aygıta erişebileceğini belirler. Ayrıca aygıttan bağımsız bir blok boyutu sunar, tamponlamayı ve hata raporlamayı yönetir, adanmış aygıtların ayrılmasını ve bırakılmasını düzenler.

11.6.6 Ara belleğe alma (tamponlama)

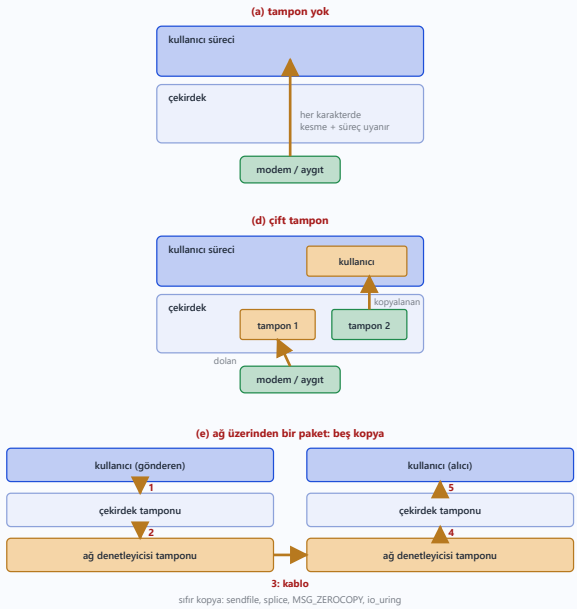
Tanım 11.16

Tampon (*buffer*), iki aygıt ya da bir aygıt ile bir uygulama arasında aktarılan veriyi geçici olarak tutan bellek alanıdır. Bu alanın kullanılmasına **tamponlama** ya da ara belleğe alma denir.

Silberschatz tamponlamanın üç nedenini sayar. Birincisi, üretici ile tüketici arasındaki **hız farkıdır**: modemden gelen yavaş akış diske tek tek yazılmaz, bir tamponda biriktirilir. İkincisi, **aktarım boyutu farkıdır**: ağdaki küçük paketler alıcı tarafta birleştirilir. Üçüncüsü **kopya anlamıdır**: `write` döndük-

ten sonra uygulama kendi tamponunu değiştirebilmelidir; diske yazılacak veri o anki hâliyle çekirdekte kopyalanmış olmalıdır.

Bir modemden ya da seri porttan karakter okuyan bir süreci düşünelim. Tamponlamanın beş farklı düzeyi vardır (Şekil 11.23).



Şekil 11.23: Tamponlama seçenekleri. Üstte tamponsuz düzen, ortada çift tampon, altta ağ üzerinden bir paketin beş kopyası.

1. **Tamponsuz.** Her karakter için bir kesme gelir ve süreç uyandırılır. Çok sayıda bağlam değiştirme olur; verimsizdir.
2. **Kullanıcı tamponu.** Kesme işleyicisi karakterleri doğrudan süreçteki bir tampona yazar; süreç ancak tampon dolunca uyandırılır. Ama tampon sayfası o sırada diske atılmışsa işleyici nereye yazacak? Sayfa bellekte kilitlenmelidir. Çok sayıda süreç sayfa kilitlerse kullanılabilir bellek azalır.
3. **Çekirdek tamponu.** Karakterler önce çekirdekteki bir tamponda birikir; tampon dolunca tek seferde kullanıcıya kopyalanır. Peki kopyalama sürerken gelen karakter nereye konacak?
4. **Çift tampon.** Çekirdekte iki tampon vardır: biri dolarken öteki kullanıcıya kopyalanır, sonra roller değişir. Bu düzenin genel biçimi dairesel tampondur.
5. **Ağda kopyalar.** Bir paket ağ üzerinden giderken beş kez kopyalanabilir: gönderen kullanıcıdan çekirdeğe, çekirdekten ağ kartına, kablo üzerinden alıcının ağ kartına, oradan çekirdeğe ve en sonda kullanıcıya. Her kopya bellek bant genişliği ve gecikme demektir. Bu yüzden `sendfile`, `splice`, `MSG_ZEROCOPY` ve io_uring gibi **sıfır kopya** teknikleri geliştirilmiştir.

Örnek 11.7

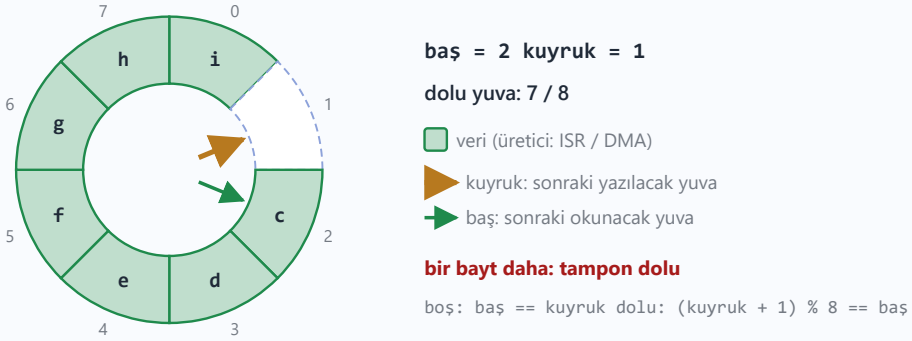
Bir aygıttan bir bloğun gelmesi T = 10 ms, sürecin bir bloğu işlemesi C = 6 ms, tampondan kullanıcı alanına kopyalama M = 2 ms sürüyor. Kararlı durumda blok başına süre nedir?

1. **Tamponsuz.** Okuma ve işleme sırayla yapılır: T + C = 16 ms.
2. **Tek tampon.** Sonraki blok çekirdek tamponuna gelirken önceki blok işlenir. Ama kopyalama sırasında tampon meşguldür ve yeni blok başlayamaz: max(T, C) + M = 10 + 2 = 12 ms.
3. **Çift tampon.** Biri dolarken öteki kopyalanır; kopyalama da örtüşür. C + M = 8 ≤ T olduğundan darboğaz aygıttır: max(T, C) = 10 ms.
4. **Yorum.** Artık sınır aygıtın hızıdır (T > C). Daha fazla tampon eklemek ortalama hızı artırmaz; yalnız ani patlamaları emer.

11.6.7 Dairesel tampon

Tanım 11.17

Dairesel tampon (*circular buffer*, *ring buffer*) N yuvalı bir dizinin uçları birleşmiş gibi kullanılmasıdır. Üretici *kuyruk* indisindeki yuvaya yazar, tüketici *baş* indisindeki yuvadan okur; iki indis de sona gelince 0'a sarar (mod N). Baş = kuyruk ise tampon boştur; (kuyruk + 1) mod N = baş ise doludur.



Şekil 11.24: Sekiz yuvalı dairesel tampon dolu: baş = 2, kuyruk = 1. Yedi yuvada c, d, e, f, g, h, i durur; bir yuva hep boş kalır.

Örnek 11.8

N = 8 yuvalı tampon boştur (baş = kuyruk = 0). Kesme işleyicisi 3 bayt (a, b, c) yazar, süreç 2 bayt okur, ardından üretici 5 bayt (d … h) ve bir bayt daha (i) yazar. İndisler nasıl değişir?

1. **Boş.** Baş ve kuyruk aynı yuvayı gösterir.
2. **Üç yazma.** a, b, c yuva 0, 1, 2'ye gider; kuyruk = 3. Tüketici henüz okumamıştır.
3. **İki okuma.** a ve b okunur; baş = 2. Yuva 0 ve 1 yeniden kullanılabilir.
4. **Beş yazma.** d … h yuva 3–7'ye gider. Kuyruk sona gelince başa sarar: 8 mod 8 = 0.

5. **Son yazma.** i yuva 0'a yazılır, kuyruk = 1. Şimdi (1 + 1) mod 8 = 2 = baş: tampon doludur. Tamponda 3 + 5 + 1 − 2 = 7 bayt vardır, yani N − 1.

Dolu tamponda üretici iki şey yapabilir: tüketicinin yer açmasını bekler ya da veriyi düşürür. Ağ kartında ikincisi olur ve paket kaybolur. Bir yuvanın hep boş bırakılmasının nedeni basittir. Bütün yuvalar dolabilseydi dolu tamponda da baş = kuyruk olurdu ve boş ile dolu ayırt edilemezdi. Bunun bedeli, N yuvalı tamponun en çok N − 1 bayt tutmasıdır.

Dairesel tamponlar G/Ç'nin her yerindedir: klavye ve seri port sürücüleri, ağ kartlarının alma ve gönderme halkaları, NVMe'nin gönderim ve tamamlanma kuyrukları, io_uring. Tek üretici ve tek tüketici varsa kilit gerekmez. Üretici yalnız kuyruğu, tüketici yalnız başı değiştirir; doğru sırayı korumak için yalnız bellek bariyerleri gerekir.

11.6.8 Kod izleme: dairesel tampon

Aşağıdaki iki işlev dairesel tamponu C ile gerçekler. `koy` kesme işleyicisinde, `al` süreçte çalışır.

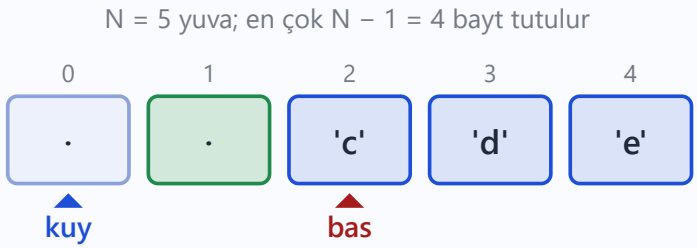
```
int koy(char c) {          /* üretici */
  int s = (kuy + 1) % N;
  if (s == bas) return -1; /* dolu */
  buf[kuy] = c;
  kuy = s;
  return 0;
}
int al(char *c) {          /* tüketici */
  if (bas == kuy) return -1; /* boş */
  *c = buf[bas];
  bas = (bas + 1) % N;
  return 0;
}
```

`koy` önce kuyruğun bir sonraki değerini `s`'de hesaplar. `s` başa eşitse yazmak tamponu "boş" görünür hâle getirirdi; bu yüzden tampon dolu sayılır ve `-1` döner. Değilse bayt `buf[kuy]`'a yazılır ve kuyruk ilerler. Burada sıra önemlidir: önce veri yazılır, sonra indis güncellenir. Tersi olsaydı tüketici henüz yazılmamış bir yuvayı okuyabilirdi. `al` ise başın kuyruğa eşit olup olmadığına bakar; eşitse tampon boştur. Değilse `buf[bas]` okunur ve baş ilerler. Her iki işlev O(1) zamanda çalışır. Tamponun alanı N yuvadır ve hiç değişmez.

Çağrı	Ayrıntı	bas	kuy	Dönüş
—	tampon boş	0	0	
`koy('a')`	buf[0] = 'a'	0	1	0
`koy('b')`	buf[1] = 'b'	0	2	0
`koy('c')`	buf[2] = 'c'	0	3	0

Çağrı	Ayrıntı	bas	kuy	Dönüş
al	bas ≠ kuy; 'a' okunur	1	3	0
koy('d')	buf[3] = 'd'	1	4	0
koy('e')	buf[4] = 'e'; kuyruk başa sarar	1	0	0
koy('f')	s = (0 + 1) mod 5 = 1 = bas: dolu	1	0	−1
al	'b' okunur	2	0	0

Tablo 11.6: N = 5 için koy ve al çağrıları. Her satır çağrıdan sonraki durumu gösterir.



Şekil 11.25: İzlemenin sonu: yuva 2, 3, 4'te 'c', 'd', 'e' durur; bas = 2, kuy = 0.

'f' reddedildiğinde tamponda dört bayt vardır ('b', 'c', 'd', 'e'): N − 1 = 4. Bir okuma yapılınca yer açılır ve bir sonraki koy başarılı olur.

Sık yapılan hata

Dolu koşulunu kuy == bas olarak yazmak. Bu koşul boş tampon için de doğrudur; tampon dolunca üretici bütün veriyi "boş" bir tamponun üstüne yazmış olur ve tüketici hiçbir şey okuyamaz. Bir yuvayı boş bırakmak istenmiyorsa ayrı bir sayaç tutmak gerekir; ama o zaman iki taraf aynı değişkeni değiştirir ve kilitsiz çalışma bozulur.

11.6.9 Önbellek, biriktirme ve hata ele alma

Önbellek ve tampon farklı şeylerdir. Tampon, verinin tek kopyasını geçici olarak tutar. Önbellek (*cache*) ise başka bir yerde duran verinin hızlı erişilen bir kopyasıdır. Linux'un sayfa önbelleği ikisini birleştirir: diske yazılacak veri orada tamponlanır, diskten okunan veri orada önbelleklenir. Boşta kalan RAM önbelleğe gider; free -h çıktısındaki buff/cache sütunu bunu gösterir.

Tanım 11.18

Biriktirme (*spooling*), adanmış bir aygıta gidecek işleri diskte bir kuyruğa alıp tek bir arka plan sürecinin bunları sırayla aygıta göndermesidir.

Yazıcı biriktirmenin klasik örneğidir. Kullanıcı yazdırılacak dosyayı biriktirme dizinine koyar; tek bir arka plan süreci (Linux'ta CUPS, dizin `/var/spool/cups`) dosyaları sırayla yazdırır. Hiçbir kullanıcı süreci yazıcıyı kilitlemez; bu yüzden yazıcı yüzünden kilitlenme de oluşmaz.

Hata ele alma iki tür hatayı ayırır. Programlama hatalarında, örneğin geçersiz bir dosya tanımlayıcısında, çağrı bir hata kodu döndürür (`EBADF`, `EINVAL`). Gerçek bir G/Ç hatasında sürücü işlemi yeniden dener; hata kalıcıysa çağrı `EIO` döndürür ve olay çekirdek günlüğüne yazılır (`dmesg`).

Ayırma ve koruma. Adanmış bir aygıt `open` ile ayrılır. Aygıt meşgulse istek kuyruğa alınır ya da çağrı `EBUSY` döndürür. G/Ç komutlarının hepsi ayrıcalıklıdır; kullanıcı süreci G/Ç'yi yalnız sistem çağrısıyla yapabilir.

En üstte kullanıcı alanındaki G/Ç yazılımı durur. `printf` ve `scanf` gibi kütüphane yordamları biçimlendirme yapar ve kendi tamponlarını tutar. stdio terminale satır satır, dosyaya ise blok blok yazar. Tampon dolunca ya da satır bitince `write` sistem çağrısı yapılır. Böylece her `printf` bir sistem çağrısına dönüşmez.

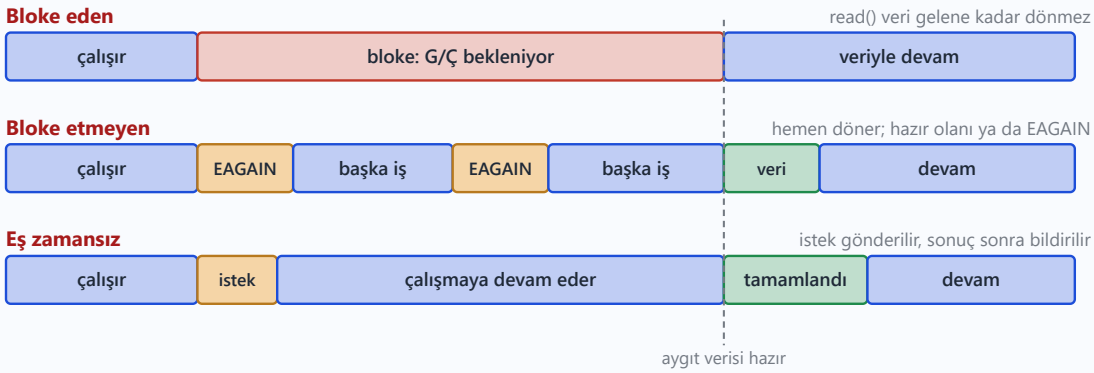
11.7 G/Ç arayüzleri

11.7.1 Bloke eden, bloke etmeyen ve eş zamansız G/Ç

Bir uygulama G/Ç istediğinde çağrının ne zaman döneceği, programın bütün yapısını belirler. Üç model vardır.

Tanım 11.19

Bloke eden G/Ç (*blocking I/O*) çağrısı, istenen veri gelene kadar dönmez; süreç bu sürede uyur. **Bloke etmeyen G/Ç** (*nonblocking I/O*) çağrısı hemen döner: o an ne kadar veri varsa onu verir, hiç yoksa `EAGAIN` hatası döndürür. **Eş zamansız G/Ç** (*asynchronous I/O*) çağrısı işi başlatıp hemen döner; iş arka planda tamamlanır ve süreç sonradan bilgilendirilir.



Şekil 11.26: Üç modelin zaman çizgileri. Bloke eden çağrıda süreç veri gelene kadar uyur. Bloke etmeyen çağrıda süreç `EAGAIN` aldıkça başka iş yapar ve yeniden dener. Eş zamansız çağrıda süreç çalışmaya devam eder ve iş bitince haber alır.

Bloke eden model en kolay programlanandır ve çoğu programın kullandığı modeldir. Ama bir süreç aynı anda binlerce ağ bağlantısına hizmet edecekse her bağlantı için bir iş parçacığı açmak pahalıdır. Tek bir iş parçacığının birçok tanımlayıcıyı izleyebilmesi için çekirdek "hangi tanımlayıcı hazır?" sorusuna cevap veren arayüzler sunar (Tablo 11.7).

Arayüz	Model	Not
`select` / `poll`	hazır olma bildirimi	her çağrıda bütün tanımlayıcı listesi taranır: O(n)
`epoll` (Linux), `kqueue` (BSD)	hazır olma bildirimi	ilgi listesi çekirdekte; yalnız hazır olanlar döner
io_uring, Windows IOCP	tamamlanma bildirimi	gerçek eş zamansız G/Ç; disk dahil

Tablo 11.7: Çoklu tanımlayıcı arayüzleri.

11.7.2 Olay döngüsü: epoll

```
int ep = epoll_create1(0);
for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
    epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
for (;;) {
  int n = epoll_wait(ep, evs, 64, -1);
  for (int i = 0; i < n; i++) {
    int fd = evs[i].data.fd;
    ssize_t r = read(fd, buf, sizeof buf);
    if (r < 0 && errno == EAGAIN) continue;
    handle(fd, buf, r);
  }
}
```

Kod tipik bir sunucunun olay döngüsüdür. `epoll_create1` çekirdekte bir epoll nesnesi oluşturur. Bu nesne iki liste tutar: izlenen tanımlayıcıların **ilgi listesi** ve o an hazır olanların **hazır listesi**. Ardından soketler bloke etmeyen kipte (`O_NONBLOCK`) ilgi listesine eklenir. Bu işlem bir kez yapılır;

`select`'teki gibi her turda listenin yeniden verilmesi gerekmez.

Sonsuz döngüde `epoll_wait` hiçbir soket hazır değilken süreci bloke eder; CPU boşa dönmez. Diyelim ki soket 7'ye bir paket geldi. Ağ kartı kesme üretir, ağ yığını paketi işler ve fd 7'yi hazır listesine ekler; süreç uyanır ve `n = 1` döner. İç döngü yalnız hazır tanımlayıcıyı işler: 512 bayt okunur ve `handle` çağrılır. Maliyet hazır tanımlayıcı sayısıyla orantılıdır, toplam tanımlayıcı sayısıyla değil. Bir sonraki `read` `EAGAIN` döndürürse okunacak veri kalmamıştır ve döngü yeniden `epoll_wait`'e gider.



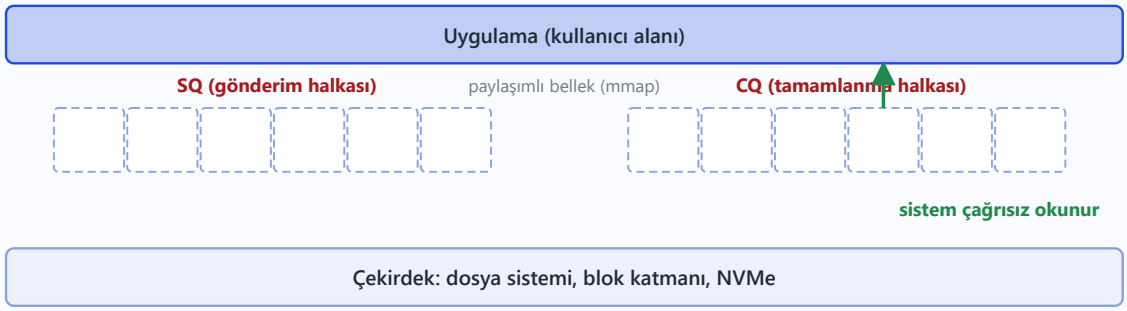
Şekil 11.27: Veri işlendikten sonra: ilgi listesinde fd 5, 6, 7; hazır listesi boş.

Bu düzenle tek bir iş parçacığı on binlerce bağlantıyı yönetebilir. nginx, Node.js ve Redis böyle çalışır. epoll'un bir sınırı da vardır. epoll yalnız "okuyabilirsin" der; işi yine `read` yapar ve kopyalama sürecin zamanından gider. Disk dosyaları ise her zaman "hazır" görünür, çünkü çekirdek bir disk okumasını hiçbir zaman `EAGAIN` ile reddetmez; bloke ederek tamamlar. Bu yüzden disk için gerçek eş zamansızlık io_uring ile gelir.

11.7.3 io_uring: paylaşılan halkalar

Tanım 11.20

io_uring (Linux 5.1, 2019), uygulama ile çekirdeğin paylaşımlı bellekte iki dairesel tamponu ortak kullandığı eş zamansız G/Ç arayüzüdür. Uygulama **gönderim halkasına** (SQ) istek kayıtları (SQE) yazar; çekirdek tamamlanan işlerin sonuçlarını **tamamlanma halkasına** (CQ) kayıt (CQE) olarak ekler.



Şekil 11.28: io_uring: uygulama ile çekirdek arasında paylaşılan gönderim (SQ) ve tamamlanma (CQ) halkaları. Uygulama tamamlanmaları sistem çağrısı yapmadan okur.

1. **Halkalar.** Uygulama ile çekirdek iki halkayı `mmap` ile paylaşımlı bellekte görür. İstekler ve sonuçlar kopyalanmaz.
2. **Gönderim kayıtları.** Uygulama üç SQE yazar: iki okuma ve bir yazma. Her kayıt işlem kodunu, dosya tanımlayıcısını, tampon adresini ve bağıl konumu taşır.
3. **Toplu gönderim.** Tek bir `io_uring_enter` çağrısı üç isteği birden gönderir. SQPOLL kipinde bir çekirdek iş parçacığı halkayı sürekli yoklar ve hiç sistem çağrısı gerekmez.
4. **Tamamlanma.** Aygıtlar işleri bitirdikçe çekirdek CQE'leri halkaya ekler. Her CQE'nin sonuç alanı aktarılan bayt sayısını ya da negatif hata kodunu (−errno) taşır.
5. **Toplama.** Uygulama CQE'leri doğrudan bellekten okur ve halkanın başını ilerletir. Toplu gönderim ve toplu toplama sayesinde bağlam değiştirme çok azalır.

io_uring bir **tamamlanma modelidir**: çekirdek "işin bitti" der, Windows'taki IOCP gibi. epoll ise bir **hazır olma modelidir**: "artık okuyabilirsin" der. io_uring ağ, dosya, `accept`, zamanlayıcı dahil yüzlerce işlem türünü destekler. Karmaşık bir arayüz olduğu için çok sayıda güvenlik açığı çıkmıştır; bu yüzden Android ve bazı konteyner ortamları gibi yerlerde kapalıdır.

Sık yapılan hata

Bloke etmeyen G/Ç ile eş zamansız G/Ç'yi aynı şey saymak. Bloke etmeyen çağrı işi yapmaz, yalnız o an yapılabilecek kadarını yapar ve geri kalanını sürece bırakır; süreç yeniden denemek zorundadır. Eş zamansız çağrı ise işin tamamını çekirdeğe bırakır ve sonucu sonra alır. Bedeli programlamanın karmaşıklaşmasıdır: tamponlar iş bitene kadar geçerli kalmalı ve her isteğin hata yolu ayrıca izlenmelidir.

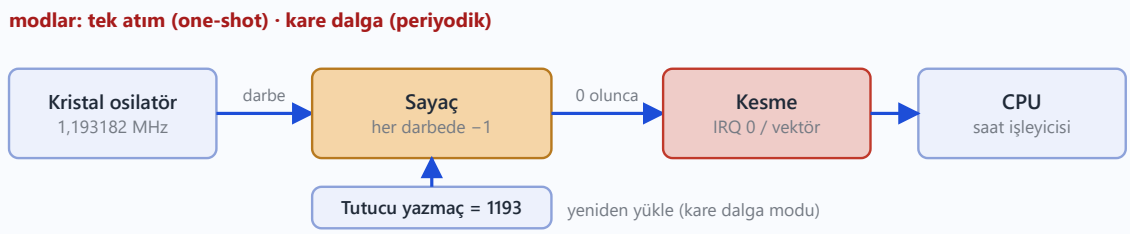
11.8 Saatler ve diğer aygıtlar

11.8.1 Saat donanımı

Saatler ne blok ne karakter aygıtıdır, ama işletim sisteminin en çok kullandığı aygıtlardandır. Zaman dilimlerini, günün saatini ve bütün zamanlayıcıları onlar yürütür.

Tanım 11.21

Programlanabilir saat, bir kristal osilatörden gelen her darbede bir sayacı bir azaltan ve sayaç sıfıra inince kesme üreten devredir. **Tek atım** (*one-shot*) kipinde saat bir kez kesme üretir ve durur. **Kare dalga** (periyodik) kipinde sayaç bir tutucu yazmaçtan yeniden yüklenir ve kesmeler düzenli aralıklarla gelir. Her kesmeye bir **tik** (*tick*) denir.



Şekil 11.29: 8254 PIT: 1,193182 MHz'lik osilatör her darbede sayacı bir azaltır; sayaç 0'a inince IRQ 0 üretilir ve sayaç tutucu yazmaçtan (1193) yeniden yüklenir.

Örnek 11.9

8254 PIT saniyede 1.193.182 darbe sayıyor. 1 kHz ve 100 Hz'lik tik için tutucu yazmaca ne yüklenir? Gerçek frekanslar nedir?

1. **1 kHz.** 1.193.182 / 1000 = 1193,18. Sayaç tam sayı olmalıdır: 1193. Gerçek frekans 1.193.182 / 1193 ≈ 1000,15 Hz.
2. **100 Hz.** 1.193.182 / 100 = 11.931,82 → 11.932. Gerçek frekans 1.193.182 / 11.932 ≈ 99,998 Hz.
3. **Sonuç.** Yuvarlama yüzünden saat istenen frekanstan biraz sapar. Uzun sürede biriken bu sapma, günün saatinin ara sıra gerçek bir saat kaynağıyla düzeltilmesini gerektirir.

8254 bugün yalnız uyumluluk için durur. Modern x86 işlemcilerinde TSC (zaman damgası sayacı), HPET ve her çekirdekteki yerel APIC zamanlayıcısı kullanılır. APIC zamanlayıcısının TSC-deadline kipinde işletim sistemi doğrudan “TSC şu değere ulaşınca kesme üret” der. ARM işlemcilerinde bu göre-

vi genel zamanlayıcı (*generic timer*) üstlenir. Pil destekli gerçek zaman saati (RTC) ise bilgisayar kapalıyken tarihi tutar; açılışta günün saati ondan okunur.

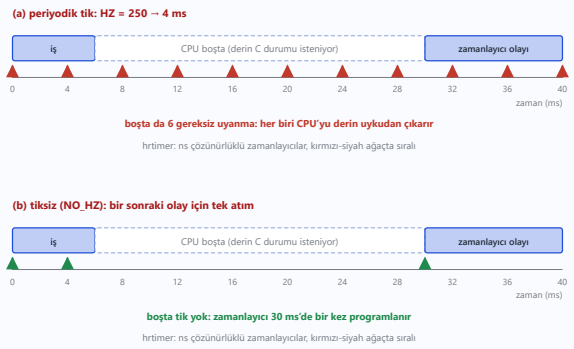
Saat yazılımının görevleri şunlardır: günün saatini tutmak, süreçlerin zaman dilimi tükenince çizelgeleyiciyi çağırmak, CPU kullanım muhasebesi yapmak, süreçlerin istediği alarmları ve zamanlayıcıları çalıştırmak, izleme köpeği zamanlayıcılarını beslemek ve profil çıkarmak için örnek almak.

Sık yapılan hata

Günün saatini 32 bitlik bir saniye sayacında tutmak. İşaretli 32 bit `time_t`, 1 Ocak 1970'ten itibaren en çok $2^{31} - 1$ saniye sayabilir. Bu sınıra 19 Ocak 2038, 03:14:07 UTC'de ulaşılır ve sayaç negatife taşar. Günün saati 64 bitlik bir sayaçla tutulmalıdır. Linux 64 bitlik `time_t` 'ye geçmiştir, ama eski gömülü sistemlerde sorun sürüyor.

11.8.2 Tik ve tiksiz çekirdek

Klasik çekirdek saati periyodik kipte kullanır. Zamanlayıcı her 1/HZ saniyede bir kesme üretir. Linux'ta HZ yapılandırma sırasında seçilir: 100, 250 ya da 1000; tik aralığı sırasıyla 10, 4 ve 1 ms'dir. Her tikte işleyici zamanı ilerletir, çalışan sürecin kalan zaman dilimini azaltır ve süresi dolan zamanlayıcıları çalıştırır.



Şekil 11.30: (a) Periyodik tik (HZ = 250, 4 ms): CPU boşta olsa bile her tikte uyanır; şekildeki boş aralıkta altı gereksiz uyanma vardır. (b) Tiksiz çekirdek: boşta tik yoktur, saat bir sonraki olaya (30 ms) tek atımla kurulur.

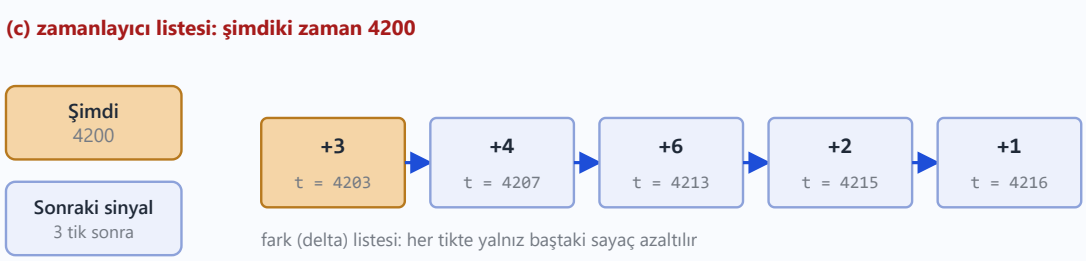
Periyodik tikin sorunu, CPU'nun boştayken de uyanmasıdır. Her uyanma CPU'yu derin uyku durumundan çıkarır ve enerji harcar. **Tiksiz çekirdek** (Linux'ta NO_HZ_IDLE, 2007) CPU boşken tiki durdurur ve saati bir sonraki olayın zamanına tek atımla kurar. `nohz_full` seçeneği bir adım daha ileri gider. Tek bir sürecin çalıştığı çekirdekte, süreç çalışırken bile tiki kaldırır. Bu, yüksek başarımlı hesaplama ve gerçek zamanlı iş yükleri için önemlidir. Tiksiz çalışma dizüstü bilgisayarlarda pil, sanal makinelerde ise işlemci zamanı kazandırır. Mikrosaniye düzeyindeki kısa beklemeler için yumuşak zamanlayıcılar ve yoklama birlikte kullanılır. Linux'un yüksek çözünürlüklü zamanlayıcıları (hrtimer) nanosaniye çözünürlüklüdür ve kırmızı-siyah ağaçta sıralı tutulur.

11.8.3 Zamanlayıcı listesi

Birçok süreç "şu kadar tik sonra bana haber ver" diye alarm kurabilir. Saat yazılımı bu alarmları her tikte hepsine bakmadan nasıl izleyebilir? Tanenbaum'un cevabı fark listesidir.

Tanım 11.22

Fark listesi (*delta list*), bekleyen zamanlayıcıları dolma zamanına göre sıralı tutan bağlı listedir. Her düğüm mutlak zamanı değil, bir önceki düğümden kaç tik sonra dolacağını (farkı) saklar. İlk düğümün farkı şimdiki zamana göredir.



Şekil 11.31: Şimdiki zaman 4200; bekleyen sinyaller 4203, 4207, 4213, 4215 ve 4216'da. Liste yalnız farkları tutar: 3, 4, 6, 2, 1.

Bu düzenin üstünlüğü tik işleyicisindedir: her tikte yalnız baştaki sayaç bir azaltılır. Baştaki fark 0'a inince olay tetiklenir ve düğüm listeden çıkar. Listede kaç zamanlayıcı olursa olsun tik başına iş O(1)'dir. Bedel eklemededir: yeni zamanlayıcının yeri bulunurken liste gezilir ve bu O(n) sürer.

11.8.4 Kod izleme: `ekle` ve `tik`

```
void ekle(Zam *z, int t) {  /* t: tik */
  Zam **p = &bas;
  while (*p && (*p)->fark <= t) {
    t -= (*p)->fark;
    p = &(*p)->sonraki;
  }
  z->fark = t; z->sonraki = *p;
  if (*p) (*p)->fark -= t;
  *p = z;
}
void tik(void) {
  if (bas) bas->fark--;
  while (bas && bas->fark == 0)
    bas = calistir(bas);   /* çıkar */
}
```

`ekle` yeni zamanlayıcıyı listeye t tik sonrası için ekler. `p` bir işaretçinin işaretçisidir: önce liste başını (`&bas`), sonra her adımda bir önceki düğümün `sonraki` alanını gösterir. Böylece başa ekleme ile araya ekleme ayrı ayrı yazılmaz. Döngü, geçilen her düğümün farkını t'den çıkarır; t artık "o düğümden sonra kaç tik" anlamına gelir. Farkı t'den büyük bir düğüme gelince döngü durur. Yeni düğüm oraya girer ve arkasındaki düğümün farkı t kadar azaltılır; böylece arkadaki düğümün mutlak zamanı değişmez. `tik` ise yalnız baştaki farkı azaltır ve farkı 0 olan bütün düğümleri sırayla çalıştırıp çıkarır. Aynı anda dolan iki zamanlayıcıdan ikincisinin farkı 0'dır; `while` döngüsü bu yüzden vardır.

Örnek 11.10

Liste A 3, B 4, C 6, D 2, E 1 (mutlak zamanlar 4203, 4207, 4213, 4215, 4216). Şimdi 4200'deyken X zamanlayıcısı 10 tik sonrası (4210) için ekleniyor: `ekle(X, 10)`. Ardından üç tik geçiyor.

1. **A.** A'nın farkı 3 ≤ 10: X ondan sonra gelir. t = 10 − 3 = 7; `*p` B'ye geçer.
2. **B.** B'nin farkı 4 ≤ 7: X ondan da sonra gelir. t = 7 − 4 = 3; `*p` C'ye geçer.
3. **C.** C'nin farkı 6 > 3: X, C'den önce girmelidir; döngü biter.
4. **Ekleme.** X.fark = 3 olarak araya girer; C'nin farkı 6 − 3 = 3 olur. Liste A 3, B 4, X 3, C 3, D 2, E 1. C'nin mutlak zamanı 4203 + 4 + 3 + 3 = 4213, yani değişmedi. Yalnız bir düğüm güncellendi.
5. **Tikler.** İlk iki tikte yalnız A'nın farkı azalır: 2, sonra 1. Üçüncü tikte (4203) fark 0 olur; A çalıştırılır ve listeden çıkarılır. Liste B 4, X 3, C 3, D 2, E 1.



Şekil 11.32: Solda X eklendikten sonra: A 3, B 4, X 3, C 3, D 2, E 1. Sağda 4203'te A çıktıktan sonra: liste B'den başlar.

Örnek 11.11

Şimdi 8000. tik; alarmlar 8002, 8006, 8006, 8011 ve 8020'de. Fark listesi nedir? 8 tik sonra hangi alarmlar tetiklenmiş olur?

1. **Farklar.** 8002 − 8000 = 2, 8006 − 8002 = 4, 8006 − 8006 = 0, 8011 − 8006 = 5, 8020 − 8011 = 9. Liste 2, 4, 0, 5, 9.
2. **İlk alarm.** 2 tik sonra baştaki fark 0 olur: 8002'deki alarm tetiklenir.
3. **Eş zamanlı alarmlar.** 4 tik daha geçince (8006) baştaki fark 0 olur. Döngü bu düğümü çıkarır; arkasındaki düğümün farkı da 0 olduğu için onu da çıkarır. 8006'daki iki alarm birlikte tetiklenir.
4. **Kalan.** 8008'e gelindiğinde iki tik daha geçmiştir: baştaki fark 5 − 2 = 3. Liste 3, 9 (8011 ve 8020).

11.8.5 Terminaller, giriş aygıtları ve güç yönetimi

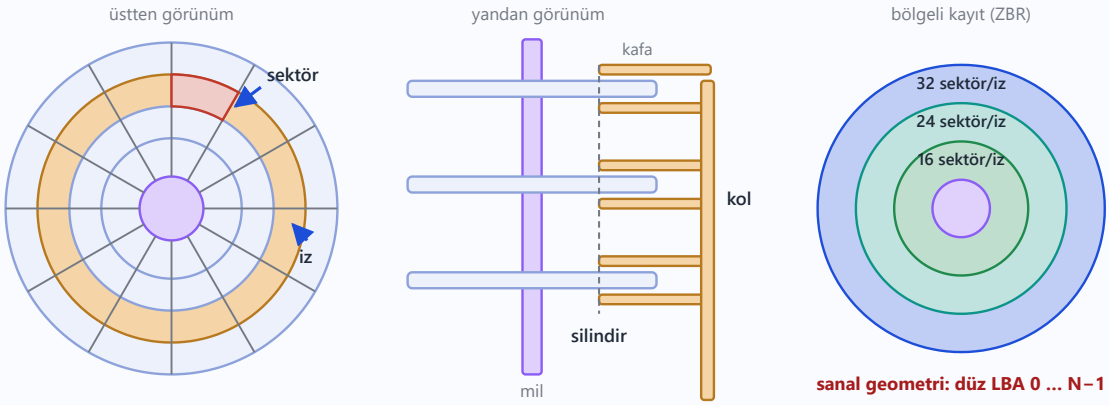
Terminaller. Unix'te terminal sürücüsü (tty) ile uygulama arasında bir **satır disiplini** bulunur. Kanonik kipte satır disiplini satırı düzenler: geri silme ve satır silme orada işlenir, satır ancak Enter'a basılınca uygulamaya teslim edilir. Ham kipte her tuş hemen uygulamaya gider; metin düzenleyiciler bu kipi kullanır (`stty -icanon`). Terminal öykünücüleri ve `ssh` oturumları sözde terminal (pty) kullanır. Ekran denetimi, imleci taşıyan ya da rengi değiştiren ANSI kaçış dizileriyle yapılır.

Klavye ve giriş aygıtları. Klavye karakter değil, tarama kodu gönderir: her tuşun basılması ve bırakılması ayrı bir koddur. Sürücü tuş eşlemesini kullanarak bunları karaktere çevirir. Shift tuşu bu yüzden ayrı bir olaydır. USB klavye ve fareler (HID aygıtları) kesme üretmez; ana makine onları saniyede 125–1000 kez yoklar. Linux'ta giriş olayları `input` alt sisteminden `/dev/input/event*` (evdev) dosyalarına, oradan libinput'a ve Wayland birleştiricisine, en sonda pencereye ulaşır.

Güç yönetimi. ACPI sistem için uyku durumları tanımlar: S0 çalışır durumdur (S0ix modern bekleme), S3 RAM'e askıya alma, S4 diske hazırda bekleme, S5 kapalıdır. İşlemcide iki ayrı ayar vardır. C durumları boşta ne kadar derin uyunacağını, P durumları çalışırken hangi frekans ve gerilimin kullanılacağını belirler; Linux'ta P durumlarını `schedutil` yöneticisi seçer. Aygıtların da D0 (açık) ile D3 (kapalı) arasında durumları vardır. Çalışma zamanı güç yönetimi, kullanılmayan aygıtı sistem çalışırken uyutur: NVMe'de APST, PCIe'de ASPM, Wi-Fi güç tasarrufu ve ekranın kararması buna örnektir. İşletim sistemi aygıtları doğru sırayla uyutup uyandırmalı ve hangi aygıtların (klavye, ağ, saat) sistemi uyandırabileceğini belirlemelidir. Bir dizüstü bilgisayarın pil ömrü, büyük ölçüde boşta geçen sürede ne kadar az güç harcandığına bağlıdır.

11.9 Diskler

11.9.1 Manyetik disk geometrisi



Şekil 11.33: Sabit disk: plakalar, izler, sektörler, silindir, okuma-yazma kafaları ve kol. Sağda bölgeli kayıt (ZBR): dış bölgelerde iz başına daha çok sektör vardır; işletim sistemi bunu düz bir LBA dizisi olarak görür.

Tanım 11.23

Sabit disk (HDD) aynı mil üzerinde dönen bir ya da daha çok manyetik **plakadan** oluşur. Her yüzeyin bir okuma-yazma kafası vardır; kafalar ortak bir kolla birlikte hareket eder. Bir yüzeydeki eş merkezli halkalara **iz**, izlerin parçalarına **sektör**, bütün yüzeylerde aynı yarıçaptaki izlerin oluşturduğu kümeye **silindir** denir.

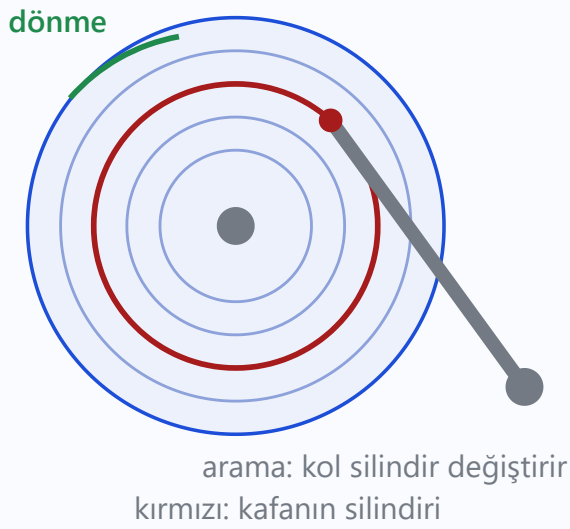
Plakalar masaüstü disklerde 5400–7200 rpm, sunucu disklerinde 10.000–15.000 rpm hızla döner. Dış izler daha uzun olduğu için dış bölgelerde iz başına daha çok sektör vardır. Buna **bölgeli kayıt** (*zoned bit recording*) denir; disk dış izlerde daha hızlı aktarım yapar.

İşletim sistemi bu geometriyi bugün görmez. Eskiden diskler silindir-kafa-sektör (CHS) üçlüsüyle adreslenirdi. Bugün işletim sistemi yalnız **LBA** (*logical block address*), yani 0'dan N − 1'e kadar numaralı bloklar görür. Disk içindeki denetleyici LBA'yı gerçek geometriye, yedek sektörlere ve kendi önbelleğine kendisi çevirir. Eski CHS arayüzünün sınırı 1024 silindir × 255 kafa × 63 sektör × 512 B ≈ 8,4 GB idi.

11.9.2 Erişim süresi

Tanım 11.24

Bir disk isteğinin **erişim süresi** üç bileşenin toplamıdır. **Arama süresi** (*seek time*) kolun doğru silindire gitmesi için geçen süredir. **Dönme gecikmesi** (*rotational latency*) istenen sektörün kafanın altına gelmesini beklemektir; ortalaması yarım turdur. **Aktarım süresi** verinin kafanın altından geçme süresidir.

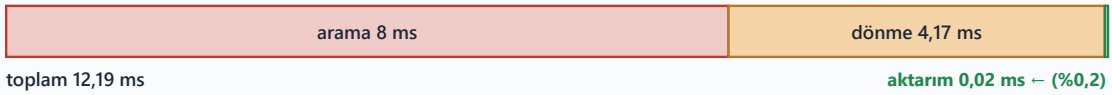


Şekil 11.34: Arama, kolun silindir değiştirmesidir; dönme gecikmesi, plakanın istenen sektörü kafanın altına getirmesidir.

Örnek 11.12

Disk 7200 rpm dönüyor; ortalama arama süresi 8 ms, ortam aktarım hızı 200 MB/s. 4 KiB'lık rastgele bir blok okumak ne kadar sürer?

1. **Bir tur.** 60.000 ms / 7200 = 8,33 ms.
2. **Dönme gecikmesi.** İstenen sektör ortalamada yarım tur uzaktadır: 8,33 / 2 = 4,17 ms.
3. **Aktarım.** 4096 B / (200 × 10^6 B/s) = 0,020 ms.
4. **Toplam.** 8 + 4,17 + 0,02 = 12,19 ms. Aktarım süresi, küçük de olsa, üç bileşenden biridir ve sayılmalıdır.



Şekil 11.35: 4 KiB rastgele okumada arama (8 ms), dönme gecikmesi (4,17 ms) ve aktarım (0,02 ms; toplamın %0,2'si).

Sonuç açıktır: süre neredeyse tamamen mekanik hareketlerde geçer. CPU bu 12 ms'de on milyonlarca komut çalıştırabilir; işletim sistemi beklemez, başka bir süreç çalıştırır. Aynı disk erişim düzenine göre çok farklı hızlar verir:

- **Rastgele 4 KiB.** Saniyede 1000 / 12,19 ≈ 82 istek (IOPS) yapılabilir. Bu, 82 × 4096 B ≈ 0,34 MB/s eder.
- **Ardışık 1 MiB.** Aktarım 1.048.576 / (200 × 10^6) = 5,24 ms; toplam 8 + 4,17 + 5,24 = 17,41 ms. Hız 1.048.576 B / 17,41 ms ≈ 60 MB/s olur.

Aynı disk, büyük ardışık okumada küçük rastgele okumaya göre yaklaşık 180 kat hızlıdır. Dosya sistemlerinin blokları yakın yerleştirmeye çalışması ve disk kolu çizelgelemenin önemi bu farktan gelir. Karşılaştırma için: bir NVMe SSD'de 4 KiB rastgele okuma 50–100 µs sürer ve paralel kuyruklarla saniyede bir milyonun üzerinde istek yapılabilir.

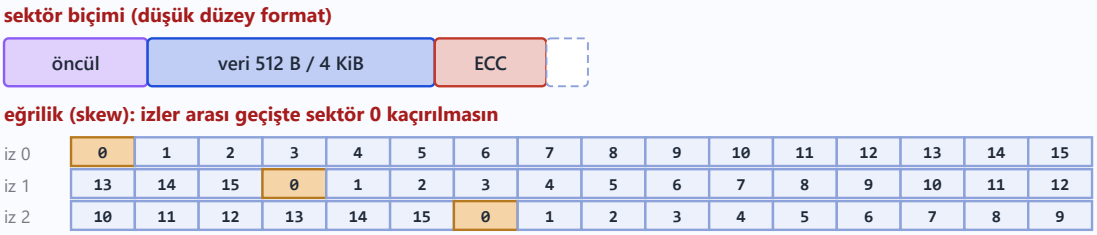
Örnek 11.13

15.000 rpm dönen bir sunucu diskinin ortalama arama süresi 3,5 ms, ortam hızı 250 MB/s (1 MB = 10^6 B). 64 KiB ardışık okuma ve diske dağılmış 16 adet 4 KiB blok okuma ne kadar sürer?

1. **Tur ve dönme.** 60.000 / 15.000 = 4 ms; ortalama dönme gecikmesi 2 ms.
2. **64 KiB ardışık.** Aktarım 65.536 / (250 × 10^6) s = 0,262 ms; toplam 3,5 + 2 + 0,262 ≈ 5,76 ms.
3. **16 × 4 KiB dağınık.** Blok başına 3,5 + 2 + 0,016 = 5,516 ms; 16 blok 16 × 5,516 ≈ 88,3 ms. Aynı veri yaklaşık 15 kat yavaş okunur.
4. **Hızlı ortam işe yarar mı?** Ortam hızı iki katına çıksa dağınık okumadaki aktarım payı 16 × 0,016 = 0,26 ms'den 0,13 ms'ye iner: toplamın yalnız %0,15'i kadar kazanç. Küçük rastgele okumada süreye arama ve dönme hâkimdir.

11.9.3 Biçimlendirme ve silindir eğriliği

Bir diskin kullanılabilmesi için iki düzeyde biçimlendirilmesi gerekir. **Düşük düzey format** fabrikada yapılır. İzler ve sektörler yüzeye yazılır; her sektöre öncül, veri alanı ve ECC konur, yedek sektörler ayrılır. Bu yüzden biçimlendirilmiş kapasite ham kapasiteden azdır. **Üst düzey format** ise bir bölüme boş bir dosya sistemi yazar: önyükleme bloğu, boş alan yönetimi yapıları ve kök dizin.



Şekil 11.36: Sektör biçimi ve silindir eğriliği. Her izin sektör 0'ı bir öncekine göre kaydırılmıştır; kafa iz değiştirirken sektör 0 kaçırılmaz.

Tanım 11.25

Silindir eğriliği (*cylinder skew*), ardışık izlerin sektör 0 konumlarının birbirine göre kaydırılmasıdır. Kafa bir izi bitirip sonrakine geçerken disk dönmeye devam eder; kaydırma, kafa yeni ize vardığında sektör 0'ın tam gelmek üzere olmasını sağlar.

Örnek 11.14

Disk 10.000 rpm dönüyor ve izde 300 sektör var. İz değiştirme 800 µs sürüyor. Eğrilik kaç sektör olmalıdır?

1. **Tur.** 60.000 / 10.000 = 6 ms.
2. **Sektör süresi.** 6 ms / 300 = 20 µs.
3. **Eğrilik.** İz değiştirme sırasında 800 / 20 = 40 sektör kafanın altından geçer. Sonraki izin sektör 0'ı 40 sektör kaydırılmalıdır.

Eskiden yavaş denetleyiciler için bir de **serpiştirme** (*interleaving*) yapılırdı: denetleyici bir sektörü belleğe aktarırken sonraki sektör kafanın altından geçip kaçmasın diye ardışık sektörler izde aralıklı dizilirdi. Bugün denetleyici bütün izi kendi tamponuna okuduğu için serpiştirme gereksizdir.

11.9.4 Optik diskler

Optik disklerde veri, manyetik diskteki eş merkezli izler yerine içten dışa uzanan tek bir sarmal iz boyunca durur. İzde çukurlar (*pit*) ve düzlükler (*land*) vardır; çukur ile düzlük arasındaki her geçiş 1 bitini, geçiş olmaması 0'ı gösterir. Basılı disklerde çukurlar polikarbonata kalıpla basılır, lazerle yakılmaz. Kaydedilebilir CD-R ve DVD-R disklerinde ise lazer organik bir boya katmanını ısıtarak yansıtıcılığını değiştirir.



Şekil 11.37: Optik diskte veri tek bir sarmal iz boyunca, manyetik diskte eş merkezli izlerde durur.

	CD (1982)	DVD (1996)	Blu-ray (2006)
Kapasite	650–700 MB	4,7 / 8,5 / 9,4 / 17 GB	25 / 50 GB; BDXL 100–128 GB
Lazer	780 nm (kızılötesi)	650 nm (kırmızı)	405 nm (mavi-mor)
En küçük çukur	0,83 µm	0,40 µm	0,15 µm
İz aralığı	1,6 µm	0,74 µm	0,32 µm

Tablo 11.8: Optik disk kuşakları.

Kuşaklar arasındaki kapasite artışı, Tablo 11.8'deki gibi lazer dalga boyunun kısalmasından gelir: daha kısa dalga boyu daha küçük bir noktaya odaklanır, çukurlar ve iz aralığı küçülür. CD-ROM'un kodlaması ayrıca ilginçtir. Her bayt, EFM (*eight-to-fourteen modulation*) ile 14 bitlik bir kanal sembolüne çevrilir. Bu bir modülasyon kodudur, hata düzeltme kodu değildir; hata düzeltme CIRC ile yapılır. 98 çerçevenin her biri 24 veri baytı taşır ve 98 × 24 = 2352 B'lık bir sektör oluşur. Mod 1'de bunun 2048 B'ı kullanıcı verisidir; geri kalanı eş zamanlama, başlık ve ek ECC/EDC alanıdır.

DVD'nin iki katmanlı türlerinde üst katman yarı saydamdır; lazer odağını değiştirerek alttaki katmanı okur. Optik ortam bugün çoğunlukla arşiv ve dağıtım için kullanılır; yoğunlukta manyetik ve flaş ortamın gerisindedir.

11.10 Disk kolu çizelgeleme

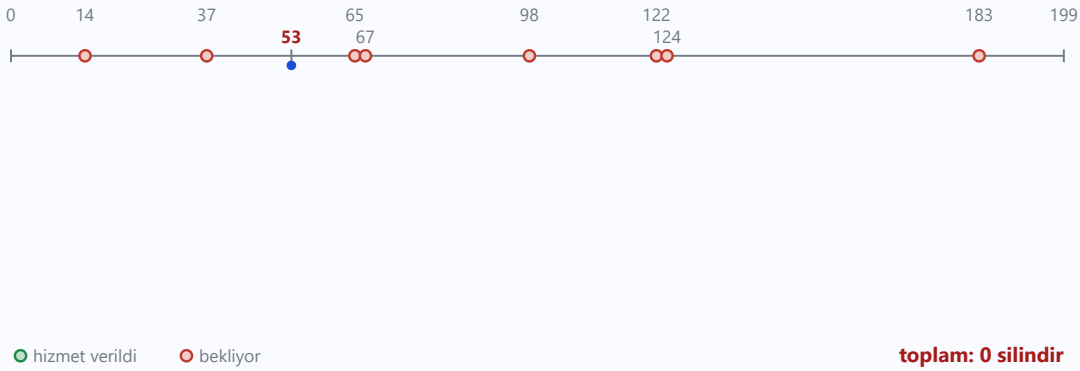
11.10.1 Amaç ve ölçüt

Erişim süresi örneği, bir disk isteğinin süresinin büyük kısmının arama ve dönmede geçtiğini gösterdi. Diskte aynı anda birçok istek bekliyorsa bunların hangi sırayla karşılanacağı, kolun ne kadar yol alacağını belirler.

Tanım 11.26

Disk kolu çizelgeleme (*disk scheduling*), bekleyen disk isteklerinin karşılanma sırasını seçerek kol hareketini (dolayısıyla arama süresini) azaltmayı ve verimi artırmayı amaçlar. Sürücü her istek için silindir numarasını ve geliş zamanını tutar. Algoritmaları karşılaştırmak için kullanılan ölçüt, kafanın silindir cinsinden **toplam hareketidir**.

Bu modelin bir basitleştirme olduğunu baştan söylemek gerekir. Bugün LBA'nın fiziksel konuma eşlenmesi diskin içindedir; işletim sistemi silindirleri bilmez. Disk ayrıca NCQ (*native command queuing*) ile 32 isteği dönme konumuna göre kendisi yeniden sıralar. İşletim sistemi bu yüzden yaklaşık olarak "LBA yakınlığı" ile çizelgeler: numaraları yakın blokların diskte de yakın olduğunu varsayar.



Şekil 11.38: Ortak kuyruk: kafa 53. silindirde; bekleyen istekler 98, 183, 37, 122, 14, 124, 65, 67. Silindirler 0–199.

Algoritmaları karşılaştırmak için hepsine aynı kuyruğu vereceğiz. Silberschatz'ın klasik örneğinde silindirler 0–199 arasıdır, kafa 53'tedir ve istekler geliş sırasıyla 98, 183, 37, 122, 14, 124, 65, 67'dir (Şekil 11.38). Tablo 11.9 algoritmaların özünü önceden özetler.

Algoritma	Fikir	Sorun
FCFS	Geliş sırası	Adil ama kol çok gezer
SSTF	Kafaya en yakın istek (açgözlü)	Uçtaki istekler açlık çekebilir
SCAN (asansör)	Bir yönde uca kadar git, dön	Uçlar ve orta eşit beklemez
C-SCAN	Tek yönde tara, uçtan başa sıçra	Dönüş hareketi boşa
LOOK / C-LOOK	SCAN / C-SCAN, ama son isteğe kadar	Uygulamada kullanılan biçimler

Tablo 11.9: Disk kolu çizelgeleme algoritmaları.

11.10.2 FCFS

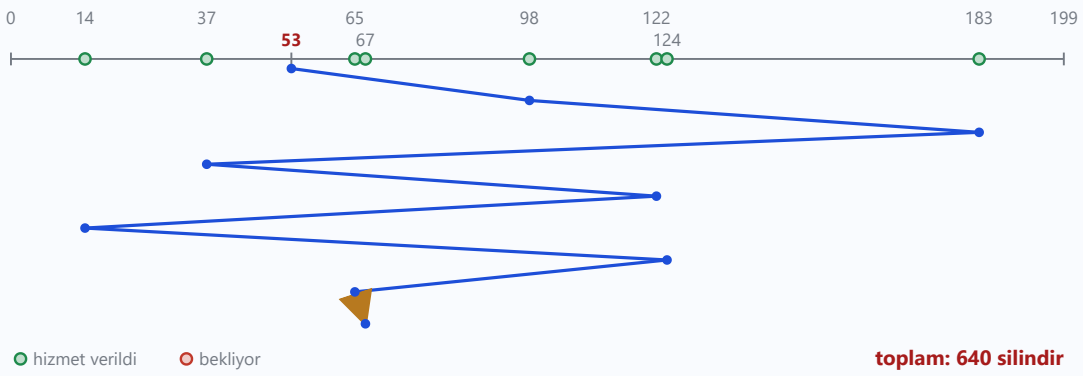
Tanım 11.27

FCFS (*first-come, first-served*, ilk gelen ilk hizmet alır) istekleri geliş sırasıyla karşılar.

Örnek 11.15

Ortak kuyrukta FCFS'nin toplam kafa hareketi nedir?

1. 53 → 98: 45 silindir; toplam 45.
2. 98 → 183: 85; toplam 130.
3. 183 → 37: 146; toplam 276.
4. 37 → 122: 85; toplam 361.
5. 122 → 14: 108; toplam 469.
6. 14 → 124: 110; toplam 579.
7. 124 → 65: 59; toplam 638.
8. 65 → 67: 2; toplam **640** silindir.



Şekil 11.39: FCFS'de kafanın yolu: 53, 98, 183, 37, 122, 14, 124, 65, 67. Toplam 640 silindir.

FCFS adildir ve açlık yaratmaz: her istek, önündekiler bitince mutlaka karşılanır. Ama 122 → 14 → 124 gibi gidip gelmeler çok pahalıdır. Kol diskin bir ucundan öbürüne iki kez gidip gelir; 14 ile 124'e giden iki istek yalnız 218 silindir tüketir.

11.10.3 SSTF

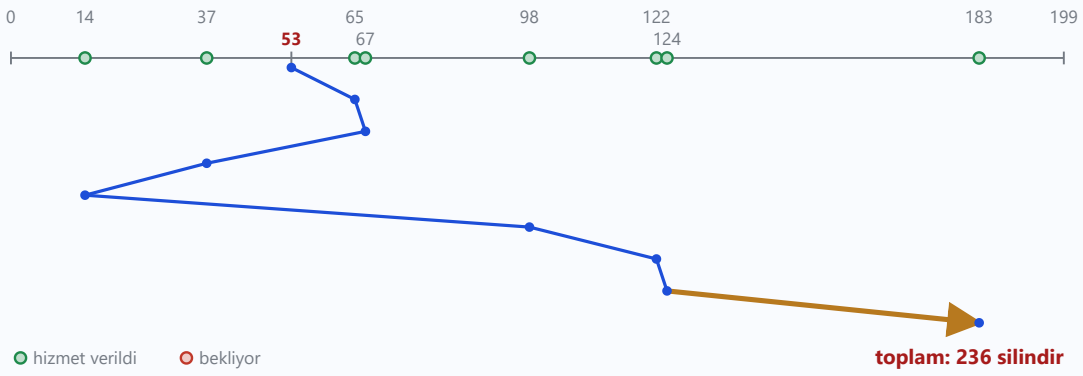
Tanım 11.28

SSTF (*shortest seek time first*, en kısa arama süresi önce) her adımda kafaya en yakın bekleyen isteği seçer.

Örnek 11.16

Ortak kuyrukta SSTF hangi sırayı izler, toplam hareket nedir?

1. Kafa 53: adaylar 65 (12 uzakta) ve 37 (16 uzakta). 53 → 65: 12; toplam 12.
2. 65 → 67: 2; toplam 14.
3. Kafa 67: 37 (30 uzakta), 98'den (31 uzakta) yakındır. 67 → 37: 30; toplam 44.
4. 37 → 14: 23; toplam 67.
5. Alt bölge bitti. 14 → 98: 84; toplam 151.
6. 98 → 122: 24; toplam 175.
7. 122 → 124: 2; toplam 177.
8. 124 → 183: 59; toplam **236**.



Şekil 11.40: SSTF'de kafanın yolu: 53, 65, 67, 37, 14, 98, 122, 124, 183. Toplam 236 silindir.

236, FCFS'nin 640'ının yaklaşık %37'sidir. Ama SSTF açgözlüdür ve en iyi sonucu garanti etmez. Kafa önce aşağı inseydi sıra 53 → 37 → 14 → 65 → 67 → 98 → 122 → 124 → 183 olurdu. Toplam 16 + 23 + 51 + 2 + 31 + 24 + 2 + 59 = 208 çıkar. SSTF'de üçüncü adımdaki 30 silindirlik kısa adım, sonra 84 silindirlik bir geri dönüşe yol açmıştır.

Sık yapılan hata

SSTF'yi en iyi algoritma sanmak. SSTF her adımda en kısa hareketi seçer, ama toplam hareketi en aza indirmez; ortak kuyrukta 236 yapar, en iyi sıra ise 208'dir. Daha önemli sorun adalettir. Yoğun yükte isteklerin sürekli diskin orta bölgesine geldiğini düşünelim: kafa hep ortada kalır ve 190. silindirdeki bir istek dakikalarca bekleyebilir. Bu bir **açlık** durumudur. FCFS'de ve tarama algoritmalarında açlık yoktur.

11.10.4 Kod izleme: `sstf()`

SSTF'nin C ile yazılmış bir sürümü aşağıdadır. `bitti[]` dizisi hangi isteklerin karşılandığını tutar.

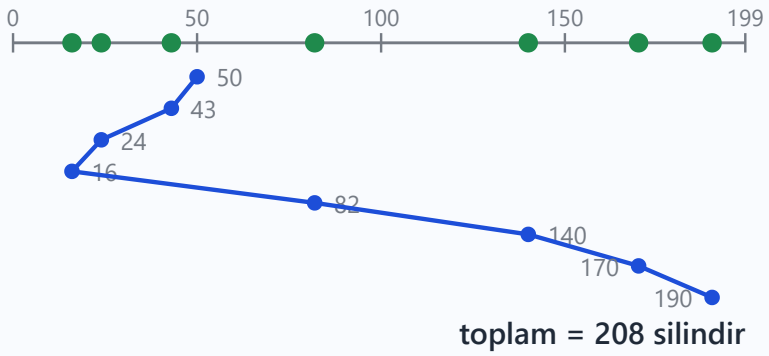
```
int sstf(int bas, int q[], int n) {
  int top = 0;
  for (int k = 0; k < n; k++) {
    int en = -1;
    for (int i = 0; i < n; i++)
      if (!bitti[i] && (en < 0 ||
        abs(q[i]-bas) < abs(q[en]-bas)))
        en = i;            /* en yakın */
    top += abs(q[en] - bas);
    bas = q[en];
    bitti[en] = 1;
  }
  return top;
}
```

Dış döngü her turda bir istek karşılar ve n tur sürer. İç döngü bekleyen isteklerin hepsine bakar; `en`, şu ana kadar bulunan en yakın isteğin indisidir. Koşuldaki `en < 0 ||` parçası ilk bekleyen isteği aday yapar; sonraki istekler ancak kesin olarak daha yakınsa adayın yerini alır. Bu yüzden eşit uzaklıkta iki istek varsa geliş sırasında önce olan seçilir. Tur sonunda uzaklık `top`'a eklenir, kafa o isteğe taşınır ve istek bitti işaretlenir. Her turda n istek tarandığı için algoritma $O(n^2)$ zaman ve `bitti[]` için $O(n)$ ek alan kullanır.

k	Kafa	En yakın (uzaklık)	Sonraki aday (uzaklık)	top
0	50	43 (7)	24 (26)	7
1	43	24 (19)	16 (27)	26
2	24	16 (8)	82 (58)	34
3	16	82 (66)	140 (124)	100
4	82	140 (58)	170 (88)	158
5	140	170 (30)	190 (50)	188

k	Kafa	En yakın (uzaklık)	Sonraki aday (uzaklık)	top
6	170	190 (20)	—	208

Tablo 11.10: `sstf()` izlemesi. Kafa 50; kuyruk (geliş sırası) 82, 170, 43, 140, 24, 16, 190.



Şekil 11.41: `sstf()` izlemesinin sonu: bütün istekler karşılandı, `top` = 208.

Bu kuyrukta FCFS 642 silindir harcardı; SSTF 208 ile üçte birin altına iner. k = 3 satırı SSTF'nin zayıf noktasını gösterir: aşağıdaki istekler bitince kol 66 silindirlik uzun bir yolu geri döner. Ortadaki istekler sürekli gelseydi uçtakiler bekleyebilirdi.

11.10.5 SCAN (asansör)

Tanım 11.29

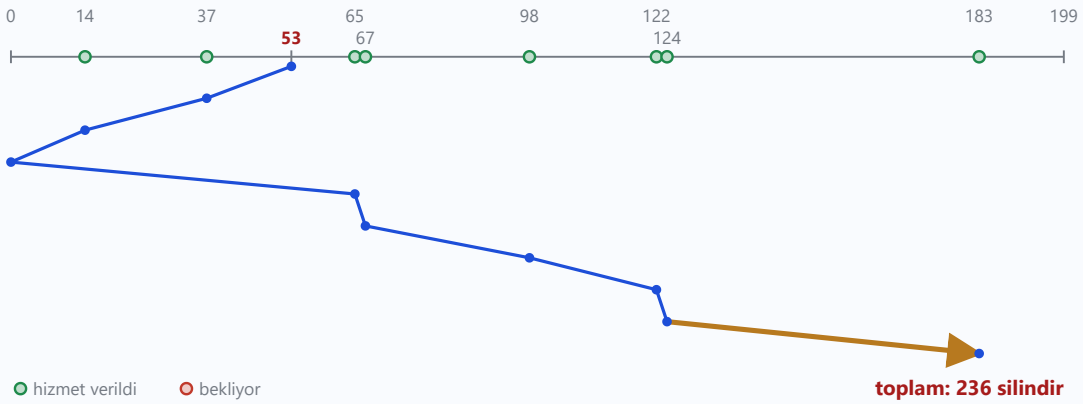
SCAN algoritmasında kafa bir yönde diskin ucuna kadar gider ve yolu üzerindeki istekleri karşılar; uca varınca yön değiştirir ve öbür yöndeki istekleri karşılar. Bir binanın katları arasında gidip gelen asansöre benzediği için **asansör algoritması** da denir.

Örnek 11.17

Ortak kuyrukta kafa 0'a doğru (aşağı) hareket ediyor. SCAN'in toplam hareketi nedir?

1. 53 → 37: 16; toplam 16.
2. 37 → 14: 23; toplam 39.
3. 14 → 0: 14; toplam 53. Aşağıda istek kalmadığı hâlde kafa diskin ucuna kadar gider.
4. Yön değişir. 0 → 65: 65; toplam 118.
5. 65 → 67: 2; toplam 120.
6. 67 → 98: 31; toplam 151.
7. 98 → 122: 24; toplam 175.

8. 122 → 124: 2; toplam 177.
9. 124 → 183: 59; toplam **236**.



Şekil 11.42: SCAN (aşağı yönde): 53, 37, 14, 0, 65, 67, 98, 122, 124, 183. Toplam 236 silindir.

SCAN bu kuyrukta SSTF ile aynı toplamı verir, ama açlığı önler: kafa yolundaki her isteği bir tarama içinde karşılar. Yeni gelen istekler kafanın önündeyse aynı taramada, arkasındaysa dönüşte karşılanır. Uca kadar gitmenin (14 → 0) bedeli boşa harcanan 14 silindirdir. Yön seçimi de sonucu değiştirir: aynı kuyrukta kafa yukarı yönde başlasaydı 53 → 199 (146) ve 199 → 14 (185) ile toplam 331 olurdu.

SCAN'in bir adalet kusuru vardır. Kafa bir uca varıp döndüğünde, hemen arkasındaki bölgeye yeni hizmet vermiştir; orada bekleyen istek azdır. İsteklerin en yoğun olduğu yer diskin öbür ucudur ve o istekler en uzun bekleyenlerdir. Kafa ortadaki silindirlerden her gidiş-dönüşte eşit aralıklarla iki kez geçer. Uçtaki silindirlerden ise kısa arayla iki kez geçer, ardından uzun bir boşluk gelir. Bu yüzden uçlara düşen istekler ortalamada daha uzun bekler.

11.10.6 C-SCAN

Tanım 11.30

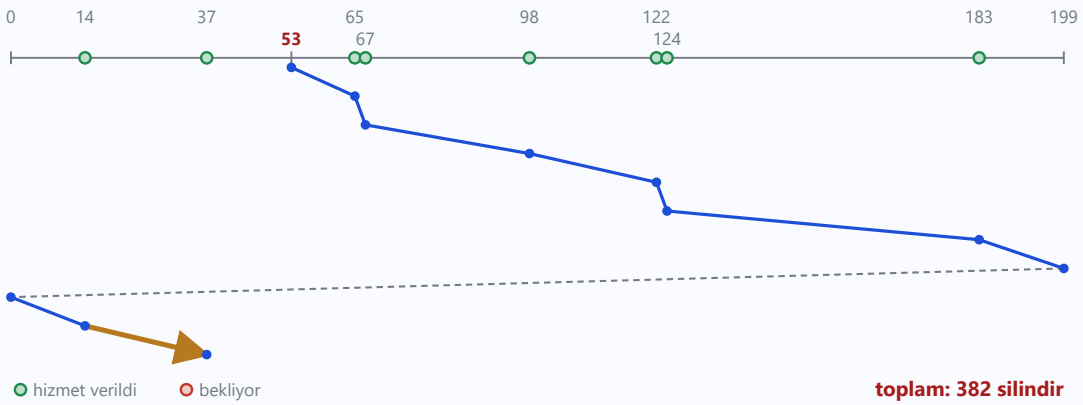
C-SCAN (dairesel SCAN) yalnız bir yönde hizmet verir. Kafa yukarı yönde uca kadar gider; uca varınca hizmet vermeden diskin başına (0'a) döner ve yeniden yukarı taramaya başlar. Disk bir halka gibi görülür.

Örnek 11.18

Ortak kuyrukta C-SCAN (yukarı yönde) hangi yolu izler? Dönüş hareketi de sayılırsa toplam nedir?

1. 53 → 65 → 67 → 98 → 122 → 124 → 183: 12 + 2 + 31 + 24 + 2 + 59 = 130.

2. 183 → 199: 16; toplam 146. Yukarıda istek yoktur ama kafa uca kadar gider.
3. 199 → 0: geri dönüş, 199 silindir, hizmet yok; toplam 345.
4. 0 → 14: 14; toplam 359.
5. 14 → 37: 23; toplam **382**.



Şekil 11.43: C-SCAN (yukarı yönde): 53 ... 183, 199, geri dönüş 0, 14, 37. Dönüş sayılırsa toplam 382, sayılmazsa 183 silindir.

Dönüş hareketi sayılmazsa toplam 382 − 199 = 183 olur. Kaynaklar bu konuda farklı davranır; bir sınav sorusunda dönüşün sayılıp sayılmadığı mutlaka belirtilmelidir. Dönüş gerçekte de hızlıdır: kol hiçbir sektörde durmadan tek hamlede geri gelir. C-SCAN'in kazancı toplam hareketten değil, adaletten gelir. Her istek, kafanın yukarı taramasında bir kez karşılanır; uçtaki ve ortadaki istekler yaklaşık eşit bekler.

11.10.7 LOOK ve C-LOOK

Tanım 11.31

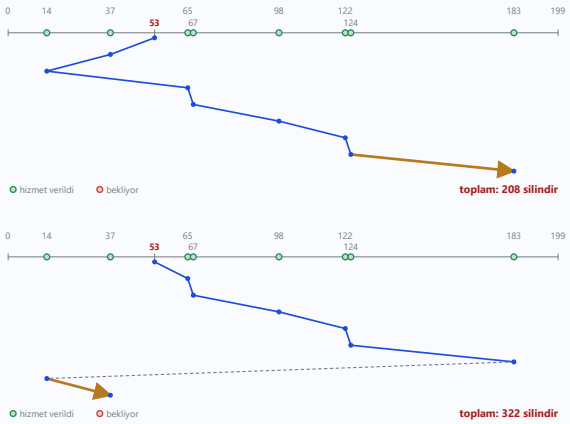
LOOK, SCAN gibi çalışır ama diskin ucuna gitmez: o yöndeki son istekte döner. **C-LOOK**, C-SCAN gibi tek yönde hizmet verir ama uca gitmez: yukarıdaki son istekten sonra doğrudan en küçük bekleyen isteğe sıçrar.

Örnek 11.19

Ortak kuyrukta LOOK (aşağı yönde) ve C-LOOK (yukarı yönde) için toplam hareket nedir?

1. **LOOK, aşağı.** 53 → 37 → 14: 16 + 23 = 39. Aşağıda istek kalmadı; 0'a gidilmez.

2. **LOOK, yukarı.** Yön değişir: 14 → 65 → 67 → 98 → 122 → 124 → 183: 51 + 2 + 31 + 24 + 2 + 59 = 169. Toplam 39 + 169 = **208**; SCAN'e göre 28 silindir az, çünkü 14 → 0 → 14 yolu (2 × 14) yapılmaz.
3. **C-LOOK, yukarı.** 53 → 65 … 183: 130.
4. **Sıçrama.** 183 → 14: en küçük bekleyen isteğe 169 silindir; 199 ve 0'a gidilmez. Toplam 299.
5. **Son.** 14 → 37: 23; toplam **322**. Sıçrama sayılmazsa 322 − 169 = 153.



Şekil 11.44: Üstte LOOK (aşağı yönde): 53, 37, 14, 65 … 183; toplam 208. Altta C-LOOK (yukarı yönde): 53 … 183, sıçrama 14, 37; toplam 322 (sıçramasız 153).

Uygulamada “asansör” denen davranış çoğunlukla LOOK'tur; hiçbir gerçek sürücü istek yokken kolu diskin ucuna götürmez. Linux'un eski `deadline` ve CFQ çizelgeleyicileri ile bugünkü `mq-deadline` istekleri LBA'ya göre artan sırada gönderir ve en yüksek LBA'dan sonra baştan başlar. Bu, C-LOOK'a benzer bir davranıştır.

11.10.8 Kod izleme: `look()`

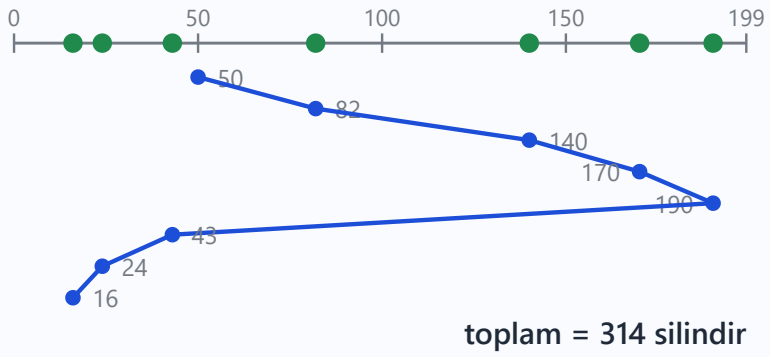
LOOK'un yukarı yönde başlayan sürümü, sıralı bir dizi üzerinde iki döngüyle yazılabilir:

```c
/* s[] sıralı; önce yukarı, sonra aşağı */
int look(int bas, int s[], int n) {
  int i = 0, top = 0;
  while (i < n && s[i] < bas) i++;
  for (int j = i; j < n; j++) {
    top += s[j] - bas; bas = s[j];
  }
  for (int j = i - 1; j >= 0; j--) {
    top += bas - s[j]; bas = s[j];
  }
  return top;
}
```

İlk `while` döngüsü kafadan büyük ya da kafaya eşit ilk isteğin indisini (`i`) bulur. Birinci `for` döngüsü `i` 'den sona kadar yukarı yönde ilerler ve her adımın uzaklığını (`s[j] - bas`) toplar. İkinci `for` döngüsü `i - 1` 'den 0'a kadar aşağı iner; bu kez uzaklık `bas - s[j]` 'dir. Kafa yukarıdaki son istekten sonra uca gitmeden döndüğü için kod LOOK'tur, SCAN değildir. Fonksiyon dizi sıralı verildiğinde O(n) çalışır; sıralama O(n log n) sürer.

Adım	`j`	Hedef	Hareket	`top`
`while`	`i` = 3: 50'den büyük ilk istek s[3] = 82. Altındaki 16, 24, 43 dönüşte alınacak.			
yukarı	3	82	+32	32
yukarı	4	140	+58	90
yukarı	5	170	+30	120
yukarı	6	190	+20	140
aşağı	2	43	+147	287
aşağı	1	24	+19	306
aşağı	0	16	+8	314

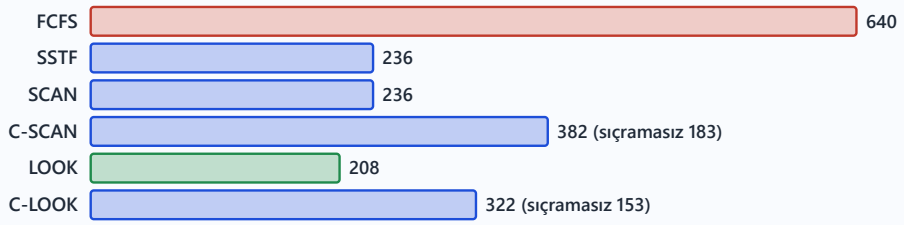
Tablo 11.11: `look()` izlemesi. s = 16, 24, 43, 82, 140, 170, 190; kafa 50, yukarı yönde.



Şekil 11.45: `look()` izlemesinin sonu: toplam 314 silindir.

Aynı kuyrukta SSTF 208 yapmıştı; LOOK 314 yaptı. Fark yön seçiminden gelir. Kafa aşağı yönde başlasaydı LOOK önce 50 → 16 (34 silindir), sonra 16 → 190 (174 silindir) giderdi: 34 + 174 = 208. Yön seçimi sonucu belirler, ama yön ne olursa olsun hiçbir istek aç kalmaz.

11.10.9 Algoritmaların karşılaştırması



Şekil 11.46: Ortak kuyrukta altı algoritmanın toplam kafa hareketi: FCFS 640, SSTF 236, SCAN 236, C-SCAN 382 (sıçramasız 183), LOOK 208, C-LOOK 322 (sıçramasız 153).

Şekil 11.46 ortak kuyruktaki sonuçları bir arada gösterir. Toplam hareket tek başına bir algoritmayı seçmeye yetmez. En az hareketi LOOK (208) verir; SSTF ve SCAN 236 ile onu izler, ama SSTF açlık yaratabilir. C-SCAN ve C-LOOK daha uzun yol alır, buna karşılık bekleme sürelerini daha eşit dağıtır. Yük hafifken bütün algoritmalar FCFS'ye benzer davranır, çünkü kuyrukta aynı anda genellikle tek istek vardır.

Örnek 11.20

Tanenbaum'un örneği: kafa 11. silindirde, istekler geliş sırasıyla 1, 36, 16, 34, 9, 12. FCFS, SSTF ve yukarı yönde asansör kaç silindir hareket eder?

1. **FCFS.** 11 → 1 → 36 → 16 → 34 → 9 → 12: 10 + 35 + 20 + 18 + 25 + 3 = **111**.
2. **SSTF.** Sıra 12, 9, 16, 1, 34, 36: 1 + 3 + 7 + 15 + 33 + 2 = **61**. Kafa 9'dayken 16 (7 uzakta), 1'den (8 uzakta) yakındır.
3. **Asansör, yukarı.** Sıra 12, 16, 34, 36, 9, 1: 1 + 4 + 18 + 2 + 27 + 8 = **60**.
4. **Yorum.** Tanenbaum'un asansörü yukarıdaki son istekte (36) döner, uca gitmez; yani aslında LOOK'tur.

Örnek 11.21

Silindirler 0–199; kafa 84'te ve son olarak 70'e hizmet verdiği için yukarı yönde gidiyor. İstekler geliş sırasıyla 184, 142, 170, 8, 60, 155, 116, 37. Altı algoritmanın toplam hareketi nedir? C-SCAN ve C-LOOK'ta geri dönüş sayılsın.

1. **FCFS.** 100 + 42 + 28 + 162 + 52 + 95 + 39 + 79 = 597.

2. **SSTF.** 84 → 60 (24) → 37 (23) → 8 (29) → 116 (108) → 142 (26) → 155 (13) → 170 (15) → 184 (14): 252. Başta 60 (24 uzakta), 116'dan (32 uzakta) yakındır.
3. **SCAN.** 84 → 116 … 184 → 199, sonra 60, 37, 8: (199 − 84) + (199 − 8) = 115 + 191 = 306.
4. **C-SCAN.** 84 → 199 (115), 199 → 0 (199), 0 → 8 → 37 → 60 (60): 374. Dönüş sayılmazsa 175.
5. **LOOK.** 84 → 184 (100), 184 → 8 (176): 276.
6. **C-LOOK.** 84 → 184 (100), sıçrama 184 → 8 (176), 8 → 60 (52): 328. Sıçrama sayılmazsa 152.
7. **Sıralama.** SSTF 252 < LOOK 276 < SCAN 306 < C-LOOK 328 < C-SCAN 374 < FCFS 597. LOOK, SCAN'den yalnız uca (199) boşuna gitmemesiyle ayrılır.

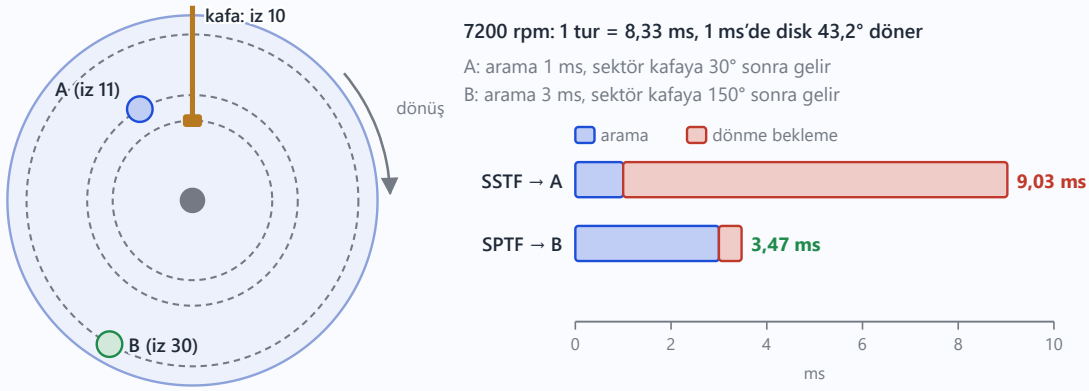
Disk Kolu Çizelgeleme Simülatörü, farklı istek kuyrukları ve başlangıç yönleriyle bu altı algoritmayı yan yana çalıştırır ve kafanın yolunu canlandırır.

11.10.10 SPTF: dönme gecikmesini de saymak

Buraya kadarki bütün algoritmalar yalnız silindir uzaklığına baktı. Oysa bir isteğin süresini arama ve dönme birlikte belirler. Yakındaki bir sektör kafanın altından yeni geçmişse, onun için neredeyse bir tur beklemek gerekir.

Tanım 11.32

SPTF (*shortest positioning time first*, en kısa konumlanma süresi önce) bekleyen isteklerden arama süresi ile dönme gecikmesinin toplamı en küçük olanı seçer.



Şekil 11.47: Kafa iz 10'da. A (iz 11) yakın ama sektörü 30° ötede; B (iz 30) uzak ama sektörü 150° ötede. SSTF A'yı seçer ve 9,03 ms harcar; SPTF B'yi seçer ve 3,47 ms harcar.

Örnek 11.22

Disk 7200 rpm dönüyor. Kafa iz 10'da. A isteği iz 11'de: arama 1 ms, sektörü kafaya göre dönme yönünde 30° ötede. B isteği iz 30'da: arama 3 ms, sektörü 150° ötede. SSTF ve SPTF hangisini seçer, ne kadar sürer?

1. **Dönme hızı.** Bir tur 60.000 / 7200 = 8,33 ms; disk 1 ms'de 360 / 8,33 = 43,2° döner.
2. **SSTF → A.** 1 ms'lik aramada disk 43,2° döner. Sektör 30° ötedeydi; arama bitince kafayı 13,2° geçmiştir. Sektörün yeniden gelmesi için 360 − 13,2 = 346,8° beklenir: 346,8 / 43,2 = 8,03 ms. Toplam 1 + 8,03 = 9,03 ms.
3. **SPTF → B.** 3 ms'lik aramada disk 129,6° döner; B'ye 150 − 129,6 = 20,4° kalır. Bekleme 20,4 / 43,2 = 0,47 ms. Toplam 3 + 0,47 = 3,47 ms.
4. **Sonuç.** Uzak iz, yakın izden 9,03 / 3,47 ≈ 2,6 kat çabuk sonuç verdi.

SPTF konumlanma süresini en aza indirir, ama işletim sistemi sektörlerin açısal konumunu bilmez. Bu yüzden SPTF diskin içinde, NCQ ile uygulanır. SPTF de SSTF gibi açlığa açıktır. Açlığa karşı kullanılan bir yöntem **F-SCAN**'dir: tarama başlarken kuyruk dondurulur; tarama sırasında gelen yeni istekler bir sonraki taramayı bekler. Böylece kafanın önüne sürekli yeni istek dökülerek eski istekler geride bırakılamaz.

11.10.11 Denetleyici önbelleği ve bozuk sektörler

Denetleyici önbelleği. Disk kendi DRAM'inde (yaklaşık 256 MB) bir iz önbelleği tutar. Kafanın altından geçen ve henüz istenmemiş blokları da okur; sonraki ardışık istek doğrudan önbellekten karşılanır. Bu önbellek işletim sisteminin sayfa önbelleğinden ayrıdır.

Yazma önbelleği ve NCQ. Disk bir yazmayı önbelleğine alıp hemen “tamam” diyebilir. Veri henüz plakaya ulaşmamıştır; güç kesilirse kaybolur. Kalıcılık gerektiğinde işletim sistemi FLUSH komutu ya da FUA (*force unit access*) bayraklı yazma kullanır. NCQ ise kuyruktaki 32 isteği dönme konumuna göre sıralar; bu, donanımın içindeki bir SPTF'dir.

Bozuk sektörler. Üretimde ya da kullanımda bozulan bir sektöre yazılan veri geri okunamaz. Denetleyici fabrikada bulunan bozuk sektörlerin listesini ve kullanımda bulduklarını izler. SMART verilerindeki “Reallocated Sectors” sayısı bu listenin büyüklüğünü gösterir. Bozuk bir sektörün iki çözümü vardır (Şekil 11.48).



Şekil 11.48: Bozuk sektör 7'yi ele almanın iki yolu. Üstte sektör 7 bir yedekle eşlenir. Altta 8–15 numaralı sektörler bir kaydırılır ve bir yedek kullanılır; sıralı okuma bozulmaz.

1. **Yedekle eşleme.** Bozuk sektör izin sonundaki bir yedek sektörle eşlenir. Basittir, ama sektör 6, 7, 8'i okuyan ardışık bir istek 7 için izin sonuna gidip gelmek zorunda kalır ve bir tur daha ödeyebilir.
2. **Kaydırma.** Bozuk sektörden sonraki bütün sektörler bir kaydırılır. İzin tamamı yeniden yazılmalıdır, ama sıra korunur.

İşletim sistemi de bozuk blok listesi tutabilir: Linux'ta `badblocks` aracı ve dosya sistemlerinin bozuk blok listeleri bu amaçla vardır. Bugünkü disklerde yeniden eşleme tamamen disk içinde yapıldığı için bu liste çoğu zaman boş kalır.

11.11 SSD ve NVMe

11.11.1 NAND flaş

Tanım 11.33

SSD (*solid-state drive*), veriyi NAND flaş bellek yongalarında tutan, hareketli parçası olmayan depolama aygıtıdır. Flaş belleğin okuma ve yazma birimi **sayfa** (yaklaşık 16 KiB), silme birimi ise çok sayıda sayfadan oluşan **bloktur** (yaklaşık 10–40 MiB).



Şekil 11.49: SSD iç yapısı: denetleyici (FTL, LDPC hata düzeltme, DRAM önbelleği) 4–8 kanal üzerinden NAND kalıplarına bağlanır. Kanallar ve kalıplar paralel çalışır.

SSD'nin sabit diskten iki temel farkı vardır. Birincisi, hareketli parça olmadığı için arama ve dönme gecikmesi yoktur. Rastgele erişim ardışık erişimle hemen hemen aynı hızdadır. İkincisi, flaşta **yerinde güncelleme yoktur**. Bir sayfa ancak silinmiş durumdayken yazılabilir ve silme yalnız blok düzeyinde yapılır. Ayrıca her hücrenin dayanabileceği silme sayısı sınırlıdır. Bu sayıya P/E (*program/erase*) döngüsü denir ve bir hücrede kaç bit tutulduğuna bağlıdır (Tablo 11.12).

Hücre	Bit/hücre	Tipik P/E döngüsü	Kullanım
SLC	1	≈ 50–100 bin	önbellek bölgesi, endüstriyel
MLC	2	≈ 3–10 bin	eski kurumsal
TLC	3	≈ 1–3 bin	çoğu tüketici ve sunucu SSD'si
QLC	4	≈ 0,1–1 bin	yüksek kapasite, okuma ağırlıklı

Tablo 11.12: NAND hücre türleri.

Hücre başına bit arttıkça hücrenin ayırt etmesi gereken gerilim düzeyi sayısı 2, 4, 8, 16 diye artar. Düzeyler sıklaştıkça okuma yavaşlar ve hücre daha çabuk yıpranır. Kapasiteyi artırmanın öteki yolu dikey yığmadır: bugünkü 3D NAND yongalarında 200–300'ün üzerinde katman bulunur. SSD ile HDD arasındaki GB başına fiyat farkı azalmıştır; yine de büyük kapasite ve arşiv için HDD hâlâ daha ucuzdur.

11.11.2 FTL, çöp toplama, TRIM ve aşınma dengeleme

Yerinde güncelleme yapılamıyorsa işletim sisteminin “LBA 1'in üzerine yaz” isteği nasıl karşılanır? Cevap, SSD denetleyicisindeki bir çeviri katmanıdır.

Tanım 11.34

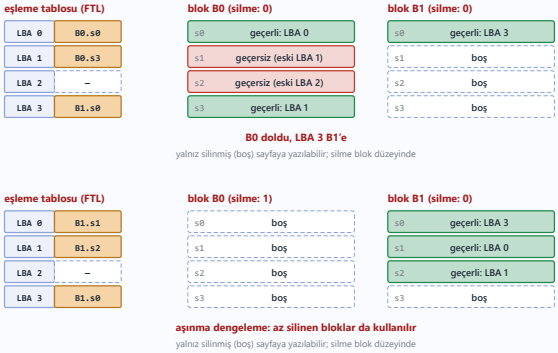
Flaş çeviri katmanı (FTL, *flash translation layer*), mantıksal blok adreslerini (LBA) fiziksel flaş sayfalarına eşleyen denetleyici yazılımıdır. Güncellenen veri yeni bir boş sayfaya yazılır, eşleme tablosu değiştirilir ve eski sayfa geçersiz sayılır. Buna **yer dışı güncelleme** denir.

Örnek 11.23

Her biri dört sayfalık iki blok (B0, B1) olan küçük bir SSD'de şu işlemler yapılıyor: LBA 0, 1, 2 yazılır; LBA 1 yeniden yazılır; LBA 2'nin bulunduğu dosya silinir; LBA 3 yazılır. Sonra B0 temizlenir.

1. **İlk yazmalar.** FTL üç LBA'nın verisini B0'ın boş sayfalarına (s0, s1, s2) koyar ve tabloya kaydeder.
2. **Güncelleme.** LBA 1 yeniden yazılır. Eski sayfanın üzerine yazılamaz; yeni veri B0.s3'e gider, B0.s1 geçersiz olur.

3. **TRIM.** Dosya silinince dosya sistemi TRIM (NVMe'de Deallocate) komutu gönderir. SSD, LBA 2'nin sayfasının artık gereksiz olduğunu öğrenir ve onu geçersiz sayar.
4. **Blok doldu.** B0'da boş sayfa kalmamıştır; LBA 3 B1'e yazılır. B0'ın yarısı geçersizdir.
5. **Çöp toplama.** B0'daki geçerli sayfalar (LBA 0 ve LBA 1) B1'e kopyalanır, B0 silinir. Kullanıcının istemediği iki ek sayfa yazması yapılmıştır; buna **yazma büyütmesi** denir.



Şekil 11.50: FTL'nin iki anı. Üstte B0 dolmuş ve yarısı geçersizdir; LBA 3 B1'e yazılmıştır. Altta çöp toplamadan sonra: geçerli sayfalar B1'e taşınmış, B0 silinmiş (silme sayısı 1).

TRIM olmasaydı SSD, silinmiş dosyanın sayfalarını hâlâ geçerli sanır ve çöp toplamada onları da boşuna kopyalardı. Yazma büyütmesi artar, hem performans hem ömür düşerdi.

Tanım 11.35

Aşınma dengeleme (*wear leveling*), bütün flaş bloklarının yaklaşık eşit sayıda silinmesini sağlayan FTL politikasıdır. Yeni yazmalar az silinmiş bloklara yönlendirilir; uzun süre değişmeyen (soğuk) veri de ara sıra taşınır ki onun tuttuğu bloklar da kullanılabilsin.

Aşınma dengeleme olmasaydı sık güncellenen birkaç blok kısa sürede ömrünü doldurur, geri kalanlar ise neredeyse hiç kullanılmamış kalırdı. SSD'ler ayrıca kullanıcıya gösterilmeyen bir yedek alan (*over-provisioning*) ayırır. Yedek alan, çöp toplamanın her zaman boş blok bulmasını kolaylaştırır ve yazma büyütmesini azaltır.

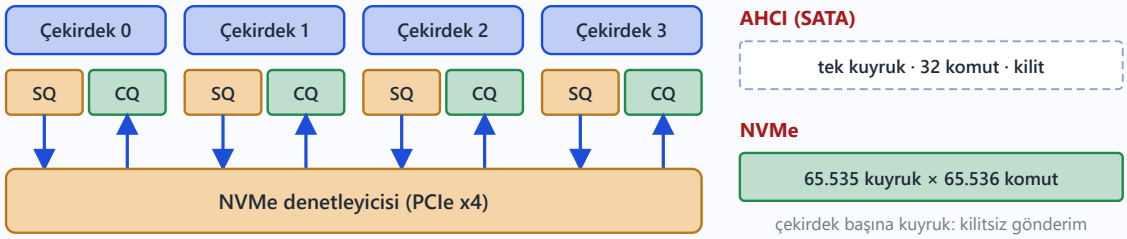
Sık yapılan hata

SSD'yi sabit disk gibi birleştirmek (defragmente etmek). SSD'de dosyanın parçalarının yan yana olması hız kazandırmaz, çünkü arama yoktur; birleştirme yalnız gereksiz yazma ve aşınma ekler. Windows'taki "İyileştir" komutu SSD'de birleştirme yapmaz, TRIM gönderir. Bunun yerine bölümleri 1 MiB sınırına hizalamak önemlidir: mantıksal bloklar flaş sayfalarına bölünmeden oturur ve tek bir yazma iki sayfaya taşmaz.

11.11.3 NVMe ve çoklu kuyruk

Tanım 11.36

NVMe (*NVM Express*), PCIe üzerine bağlanan flaş depolama için tasarlanmış komut protokolüdür. Her çekirdeğe kendi gönderim (SQ) ve tamamlanma (CQ) kuyruğu çifti verilebilir; kuyruklar bellekte duran dairesel tamponlardır.

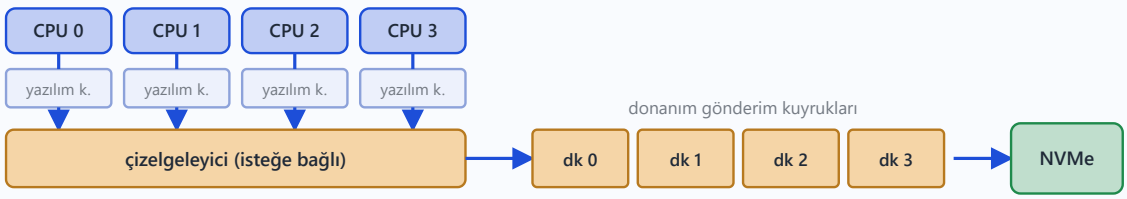


Şekil 11.51: NVMe'de her çekirdeğin kendi SQ/CQ çifti vardır ve gönderim kilitsizdir. AHCI'da (SATA) tek bir kuyruk ve 32 komut vardır; bütün çekirdekler aynı kilidi paylaşır.

SATA ve onun AHCI arayüzü dönen diskler için tasarlanmıştı. Tek bir kuyruğu ve en çok 32 komutu vardır; bu sınır bir sabit disk için yeterliydi, çünkü disk zaten aynı anda tek bir isteği yerine getirir. Flaş ise kanallar ve kalıplar boyunca yüzlerce işi paralel yürütebilir. NVMe bu paralelliği açığa çıkarır: 64 K kuyruk ve kuyruk başına 64 K komut tanımlar. Her çekirdek kendi kuyruğuna yazdığı için çekirdekler arasında kilit çekişmesi olmaz; tamamlanma kesmesi de MSI-X ile aynı çekirdeğe gelir.

Birkaç terim sık karıştırılır. NVMe bir protokoldür; M.2 ise bir kart biçimidir (form faktörü). M.2 bir kart SATA da olabilir. U.2/U.3 ve E1.S/E3.S sunucularda kullanılan kablolu ya da yuvalı biçimlerdir. NVMe sürücü desteği 2013'ten beri bütün yaygın işletim sistemlerinde yerleşiktir; AHCI ile uyumsuzluk yalnız protokol düzeyindedir. NVMe disklerde gecikme 10–100 µs arasındadır; PCIe 5.0 x4 diskler yaklaşık 14 GB/s ve saniyede 2 milyonun üzerinde istek sunar. NVMe-oF aynı protokolü ağ üzerinden taşır; ZNS (bölgeli ad alanları) ise flaşın blok yapısını doğrudan yazılıma açar.

11.11.4 Linux blk-mq ve G/Ç çizelgeleyicileri



Şekil 11.52: Linux blk-mq: CPU başına yazılım kuyrukları, isteğe bağlı bir çizelgeleyici ve aygıtın donanım gönderim kuyrukları.

Linux'un blok katmanı NVMe'nin çoklu kuyruk modelini izler. **blk-mq** (*multi-queue block layer*) her CPU'ya bir yazılım kuyruğu verir ve bunları aygıtın donanım kuyruklarına bağlar. Tek kuyruklu eski tasarımda milyonlarca IOPS'de bütün çekirdekler aynı kuyruk kilidini beklerdi; blk-mq bu çekişmeyi ortadan kaldırır. Linux 5.0'da tek kuyruklu eski blok katmanı ve onun çizelgeleyicileri (noop, deadline, CFQ) tamamen kaldırıldı. Bugünkü **G/Ç çizelgeleyicileri** (*I/O scheduler*) Tablo 11.13'te özetlenmiştir.

Çizelgeleyici	Fikir	Uygun
`none`	Sıralama yok, doğrudan gönder	NVMe (varsayılan), hızlı SSD
`mq-deadline`	LBA'ya göre sıralı + okuma/yazma son tarihi	SATA HDD/SSD (çoğu dağıtımda varsayılan)
`bfq`	Süreç başına bütçe, adil paylaşım	Masaüstü, yavaş aygıtlarda etkileşim
`kyber`	Okuma/yazma gecikme hedefleriyle sınırlama	Hızlı çok kuyruklu aygıtlar

Tablo 11.13: Linux blk-mq G/Ç çizelgeleyicileri.

Bir aygıtın çizelgeleyicisi `sysfs` üzerinden okunur ve değiştirilir:

```
$ cat /sys/block/nvme0n1/queue/scheduler
[none] mq-deadline kyber bfq
$ echo bfq | sudo tee /sys/block/sda/queue/scheduler
```

Köşeli parantez içindeki ad etkin çizelgeleyicidir; NVMe diskte bu `none`'dır. İkinci komut bir SATA diskin çizelgeleyicisini `bfq` yapar. `sudo tee` kullanılır, çünkü yönlendirme (`>`) `sudo`'nun dışında, kullanıcının yetkisiyle yapılırdı.

NVMe'de neden hiç sıralama yapılmaz? Kol çizelgelemenin varlık nedeni arama süresini azaltmaktır. SSD'de arama yoktur ve erişim süresi konumdan neredeyse bağımsızdır. Çoklu donanım kuyrukları zaten her çekirdeğe paralel bir yol verir. Sıralama yalnız CPU harcar ve gecikme ekler. SSD'de çizelgeleyicinin işi sıralamak değil, gerektiğinde adaleti ve gecikme hedeflerini korumaktır; bunun için `mq-deadline`, `bfq` ya da `kyber` seçilebilir.

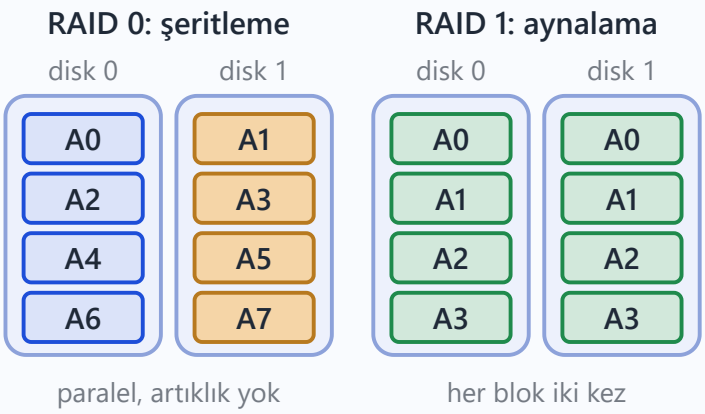
11.12 RAID ve kararlı depolama

11.12.1 RAID: diskleri birleştirmek

Tanım 11.37

RAID (*Redundant Array of Independent Disks*), birçok diski işletim sistemine tek bir disk gibi gösteren düzendir. **Şeritleme** (*striping*) ardışık blokları farklı disklere dağıtarak paralellik sağlar. **Aynalama** (*mirroring*) her bloğun kopyasını başka bir diskte tutar. **Eşlik** (*parity*) ise birkaç bloğun XOR'unu ayrı bir blokta saklayarak kaybolan bir bloğun yeniden hesaplanmasını sağlar.

RAID fikri Patterson, Gibson ve Katz'ın 1988 tarihli makalesine dayanır. Makale, pahalı tek büyük disk (SLED, *single large expensive disk*) yerine ucuz disklerden bir dizi kurmayı önerdi. Açılımdaki "I" bu yüzden ilk başta *inexpensive* idi. Bugün "bağımsız" (*independent*) olarak okunur. Standart düzeyler 0–6 arasıdır. RAID bir donanım kartıyla ya da yazılımla kurulabilir: Linux'ta `mdadm`, ZFS'te RAID-Z, Btrfs ve Windows Depolama Alanları yazılım RAID örnekleridir.



Şekil 11.53: RAID 0 ardışık blokları (A0, A1, ...) disklere sırayla dağıtır: paralel ama artıklık yok. RAID 1 her bloğu iki diske birden yazar.

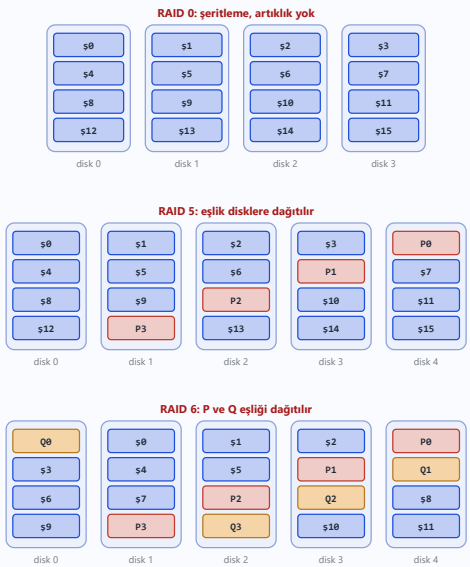
Düzen	Kullanılabilir	Dayanır	Küçük rastgele yazma maliyeti
RAID 0 (şeritleme)	16 TB	0 disk	1 G/Ç
RAID 1 (4 kopya ayna)	4 TB	3 disk	4 G/Ç (her kopyaya)
RAID 10 (ayna + şerit)	8 TB	1 (çiftler farklıysa 2)	2 G/Ç
RAID 5 (dağıtılmış eşlik)	12 TB	1 disk	4 G/Ç (2 oku + 2 yaz)
RAID 6 (çift eşlik)	8 TB	2 disk	6 G/Ç (3 oku + 3 yaz)

Tablo 11.14: Dört adet 4 TB diskle RAID düzenleri.

Sık yapılan hata

RAID'i yedek saymak. RAID disk arızasına karşı korur, ama kullanıcı hatasına, yazılım hatasına ya da fidye yazılımına karşı korumaz. Silinen bir dosya bütün disklerde aynı anda silinir; bozulan bir dosya her kopyada bozulur. Yedek, ayrı bir ortamda ve zamanda alınmış bir kopyadır.

11.12.2 RAID düzenleri



Şekil 11.54: Üstte RAID 0: şeritler (ş0, ş1, ...) dört diske sırayla dağılır. Ortada RAID 5: eşlik blokları (P) her şeritte başka bir diskte durur. Altta RAID 6: P ve Q eşlikleri beş diske dağıtılır.

- **RAID 0.** Ardışık şeritler farklı disklere gider. Büyük isteklerde n disk n kat bant genişliği verir. Artıklık yoktur: bir disk bozulursa dizideki her şey kaybolur.
- **RAID 1.** Her yazma iki kez yapılır. Okuma iki diskten birinden yapılabilir; yük diskler arasında dağıtılır. Kapasitenin yarısı artıklığa gider.
- **RAID 10.** Önce aynalanır, sonra şeritlenir. Yazma hızlıdır ve bozulan disk yalnız aynasından kopyalanarak hızla yeniden kurulur. Veritabanlarında yaygındır.
- **RAID 2 ve 3.** RAID 2 sözcük düzeyinde Hamming kodu, RAID 3 bayt düzeyinde tek bir eşlik diski kullanır. Her iki düzende de bütün disklerin eş zamanlı dönmesi gerekir. Diskler zaten kendi içlerinde ECC yaptığı için bugün kullanılmazlar.
- **RAID 4.** Blok düzeyinde şeritleme ve ayrı bir eşlik diski. Her küçük yazma eşlik diskini de günceller; eşlik diski darboğaz olur.
- **RAID 5.** Eşlik blokları disklere sırayla dağıtılır ve darboğaz kalkar. n diskin n − 1'inin kapasitesi kullanılır; bir disk kaybına dayanır.

- **RAID 6.** İki eşlik tutar: P (XOR) ve Q ($GF(2^8)$ cisminde Reed–Solomon kodu). İki disk kaybına dayanır. Büyük disklerde yeniden kurma saatler sürdüğü için, bu sürede ikinci bir arıza riskine karşı tercih edilir.

11.12.3 Eşlik ve XOR

Eşliğin temeli XOR ($\oplus$) işlemidir. İki bit farklıysa XOR 1, aynıysa 0 verir. Bit dizileri üzerinde XOR, her bit konumunda ayrı ayrı uygulanır. İşlemin üç özelliği RAID'in bütün hesaplarını taşır: $x \oplus x = 0$, $x \oplus 0 = x$, ve işlem değişme ve birleşme özelliklidir.

Teorem 11.1 (Eşlikle kurtarma)

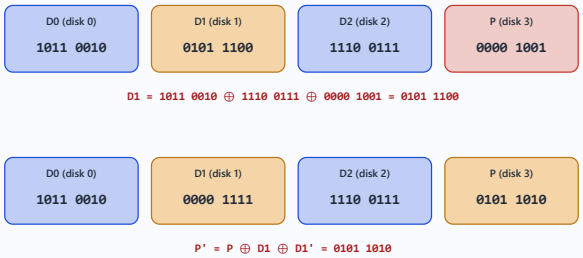
$P = D_0 \oplus D_1 \oplus \dots \oplus D_{n-1}$ olsun. Hangisinin kaybolduğu bilinen herhangi bir D_k bloğu, geri kalan veri bloklarının ve P'nin XOR'una eşittir: $D_k = P \oplus$ (k dışındaki bütün D_i'lerin XOR'u).

Kanıt kısadır. P'nin tanımında D_k dışındaki blokları her iki yana XOR'layalım. Her D_i ($i \neq k$) sağ tarafta iki kez geçer ve $x \oplus x = 0$ olduğu için yok olur; geriye D_k kalır. Tek bir eşliğin yetmesinin nedeni, hangi diskin bozulduğunun bilinmesidir: disk ya hiç yanıt vermez ya da ECC hatası bildirir. Eşlik bozuk bloğu bulmaz, yalnız yerini bildiğimiz bloğu yeniden hesaplar.

Örnek 11.24

Bir RAID 5 şeridinde D0 = 1011 0010, D1 = 0101 1100, D2 = 1110 0111 (8 bitlik örnek bloklar). Eşlik nedir? Disk 1 bozulursa D1 nasıl bulunur? D1 yerine 0000 1111 yazılırsa yeni eşlik nedir?

1. **Eşlik.** D0 $\oplus$ D1 = 1110 1110; $\oplus$ D2 = 0000 1001. P = 0000 1001. Her bit konumunda 1'lerin sayısı artık çifttir.
2. **Kurtarma.** D1 = D0 $\oplus$ D2 $\oplus$ P = 1011 0010 $\oplus$ 1110 0111 $\oplus$ 0000 1001. İlk ikisi 0101 0101, P ile 0101 1100 verir: özgün D1. Yeni diske bütün şeritler böyle yeniden yazılır; buna yeniden kurma denir.
3. **Küçük yazma.** D1' = 0000 1111. Eski D1 ve eski P okunur. P' = P $\oplus$ D1 $\oplus$ D1' = 0000 1001 $\oplus$ 0101 1100 $\oplus$ 0000 1111 = 0101 1010. Sonra D1' ve P' yazılır: 4 G/Ç (oku-değiştir-yaz).
4. **Denetim.** D0 $\oplus$ D1' $\oplus$ D2 = 1011 0010 $\oplus$ 0000 1111 $\oplus$ 1110 0111 = 0101 1010 = P'. Şeridin tamamı okunmadan doğru eşlik bulunmuştur.



Şekil 11.55: Üstte kurtarma: D1 = D0 ⊕ D2 ⊕ P = 0101 1100. Altta küçük yazma: D1' = 0000 1111 yazılınca P' = P ⊕ D1 ⊕ D1' = 0101 1010.

Yeniden kurmanın bir riski de vardır. Tüketici sınıfı disklerde düzeltilemeyen okuma hatası (URE, *unrecoverable read error*) oranı yaklaşık 10^{14} bitte 1'dir. Bozulan diski kurmak için kalan disklerin tamamı okunmalıdır.

Örnek 11.25

RAID 5 dizisinde bir disk bozuldu; yeniden kurma için kalan disklerden toplam 12 TB okunacak. URE oranı 10^{14} bitte 1. Okuma sırasında beklenen hata sayısı ve en az bir hata olasılığı nedir?

1. **Bit sayısı.** 12×10^{12} B × 8 = $9{,}6 \times 10^{13}$ bit.
2. **Beklenen hata.** $9{,}6 \times 10^{13} / 10^{14} = 0{,}96$.
3. **En az bir hata.** Hatalar bağımsız ve seyrekse sayıları yaklaşık Poisson dağılır: $1 - e^{-0{,}96} \approx 0{,}62$. Yeniden kurmanın en az bir okunamayan sektöre takılma olasılığı yaklaşık %62'dir.
4. **Yorum.** RAID 5'te bu anda artıklık kalmamıştır; takılan sektörün şeridi kurtarılamaz. Büyük dizilerde RAID 6 bu yüzden seçilir: ikinci eşlik, yeniden kurma sırasındaki bu hatayı da düzeltir.

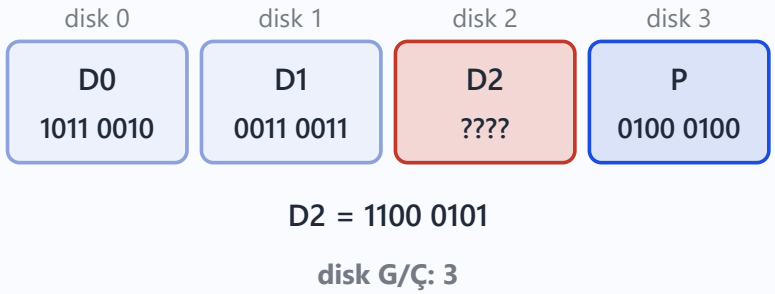
11.12.4 Kod izleme: küçük yazma ve kurtarma

```
void kucuk_yaz(int d, uint8_t yeni) {
  uint8_t eski = oku(d);   /* G/Ç 1 */
  uint8_t p = oku(P);      /* G/Ç 2 */
  p = p ^ eski ^ yeni;
  yaz(d, yeni);            /* G/Ç 3 */
  yaz(P, p);               /* G/Ç 4 */
}
uint8_t kur(int bozuk) {
  uint8_t x = 0;
  for (int d = 0; d <= P; d++)
    if (d != bozuk)
      x ^= oku(d);         /* P dahil */
  return x;
}
```

Kodda diskler 0'dan numaralanır; `P`, eşlik diskinin numarasıdır (burada 3). `oku` ve `yaz` birer disk G/Ç'sidir. `kucuk_yaz` d diskindeki bloğu `yeni` ile değiştirir. Önce eski veri ve eski eşlik okunur (2 G/Ç). `p ^ eski` eski verinin etkisini eşlikten çıkarır, çünkü x ⊕ x = 0'dır; `^ yeni` yeni değeri ekler. Sonra yeni veri ve yeni eşlik yazılır (2 G/Ç). Şeridin öteki veri diskleri hiç okunmaz. `kur` ise bozuk disk dışındaki bütün diskleri, eşlik dahil, XOR'layarak kaybolan bloğu hesaplar.

Adım	İşlem	Değer	G/Ç
1	`eski = oku(1)`	0110 1001	1
2	`p = oku(P)`	0001 1110	2
3	`p = p ^ eski ^ yeni`	0001 1110 ⊕ 0110 1001 ⊕ 0011 0011 = 0100 0100	2
4	`yaz(1, yeni)`	D1 = 0011 0011	3
5	`yaz(P, p)`	P = 0100 0100	4
6	`kur(2)`, d = 0	x = 1011 0010	
7	d = 1	x = 1011 0010 ⊕ 0011 0011 = 1000 0001	
8	d = 2 atlanır; d = 3 (P)	x = 1000 0001 ⊕ 0100 0100 = 1100 0101	3 okuma

Tablo 11.15: `kucuk_yaz(1, 0011 0011)` ve ardından `kur(2)`. Başta D0 = 1011 0010, D1 = 0110 1001, D2 = 1100 0101, P = 0001 1110.



Şekil 11.56: `kur(2)` 'nin sonucu: x = 1100 0101, kaybolan D2 ile aynı.

Kurtarılan değer özgün D2 ile aynıdır. Yeni diske her şerit için 3 okuma yapılarak yeniden kurulur ve bu süre boyunca dizi korumasızdır. Küçük yazmanın da bir zayıf noktası vardır: 4. ve 5. adımdaki iki yazma arasında sistem çökerse veri yeni, eşlik eski kalır ve şerit tutarsız olur. Buna **yazma deliği** (*write hole*) denir. Ardından bir disk bozulursa yanlış eşlikten yanlış veri hesaplanır. Donanım RAID kartları bunu pil destekli önbellekle, ZFS ise şeritleri hiçbir zaman yerinde güncellemeyerek önler.

Örnek 11.26

Dört veri diskli bir şeritte D0 = 1100 1010, D1 = 0011 0110, D2 = 1010 0001, D3 = 0101 1100. P'yi bulun; D3'ün diski bozulursa D3'ü kurtarın; D0 yerine 1111 0000 yazılırsa yeni P nedir?

1. **P.** D0 ⊕ D1 = 1111 1100; ⊕ D2 = 0101 1101; ⊕ D3 = 0000 0001. P = 0000 0001.
2. **Kurtarma.** D0 ⊕ D1 ⊕ D2 ⊕ P = 0101 1101 ⊕ 0000 0001 = 0101 1100: özgün D3.
3. **Küçük yazma.** Değişen bitler D0 ⊕ Y = 1100 1010 ⊕ 1111 0000 = 0011 1010. P' = P ⊕ 0011 1010 = 0011 1011.
4. **Denetim.** Y ⊕ D1 ⊕ D2 ⊕ D3 = 1111 0000 ⊕ 0011 0110 ⊕ 1010 0001 ⊕ 0101 1100 = 0011 1011. Sonuç aynıdır.

11.12.5 RAID performansı

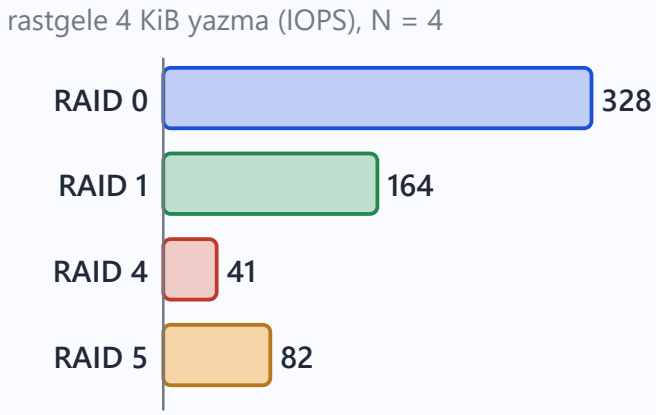
Bir RAID düzeninin hızı, tek bir diskin iki ölçüsüyle kolayca ifade edilir. Bunlar rastgele küçük isteklerdeki hız R (IOPS) ve büyük ardışık aktarımdaki hız S (MB/s)'dir. Arpaci-Dusseau'ların modeli bu iki sayıdan yola çıkar.

Örnek 11.27

Tek disk 7200 rpm, 8 ms ortalama arama, 200 MB/s ortam hızına sahip. Dizi N = 4 diskten oluşuyor. R ve S nedir? RAID 0, 1, 4 ve 5'in hızları nedir?

1. **R.** Rastgele 4 KiB okuma 12,19 ms sürer; R = 1000 / 12,19 ≈ 82 IOPS (0,34 MB/s).

2. **S.** Ardışık 10 MiB parça: aktarım 10.485.760 / (200 × 10^6) = 52,43 ms; toplam 8 + 4,17 + 52,43 = 64,6 ms. S = 10,49 MB / 64,6 ms ≈ 162 MB/s.
3. **RAID 0.** Her istek tek diske gider ve dört disk paralel çalışır: ardışık 4 · S ≈ 649 MB/s, rastgele okuma ve yazma 4 · R ≈ 328 IOPS.
4. **RAID 1 (iki ayna çifti).** Kapasite 2 disk. Rastgele okuma her iki kopyadan yapılabilir: 4 · R ≈ 328. Her yazma iki diske gider: 2 · R ≈ 164. Ardışık okuma ise yalnız 2 · S ≈ 325 MB/s'dir.
5. **RAID 4.** Kapasite 3 disk. Ardışık 3 · S ≈ 487 MB/s; rastgele okuma 3 · R ≈ 246. Küçük yazmada eşlik diski her yazmada 1 okuma ve 1 yazma yapar: R / 2 ≈ 41 IOPS.
6. **RAID 5.** Kapasite 3 disk. Ardışık 3 · S ≈ 487 MB/s; rastgele okuma bütün diskleri kullanır: 4 · R ≈ 328. Küçük yazmanın 4 G/Ç'si disklere yayılır: (4 / 4) · R ≈ 82 IOPS.



Şekil 11.57: Dört diskte rastgele küçük yazma hızı: RAID 0 328, RAID 1 164, RAID 4 41, RAID 5 82 IOPS.

Düzen	Kapasite	Ardışık okuma/yazma	Rastgele okuma	Rastgele yazma	Yazma gecikmesi
RAID 0	N = 4	N·S ≈ 649 MB/s	N·R ≈ 328	N·R ≈ 328	D
RAID 1 (ikili ayna)	N/2 = 2	(N/2)·S ≈ 325 MB/s	N·R ≈ 328	(N/2)·R ≈ 164	D (yavaş kopya)
RAID 4	N − 1 = 3	(N−1)·S ≈ 487 MB/s	(N−1)·R ≈ 246	R/2 ≈ 41	2D
RAID 5	N − 1 = 3	(N−1)·S ≈ 487 MB/s	N·R ≈ 328	(N/4)·R ≈ 82	2D

Tablo 11.16: N diskli dizilerin hızları (R: tek diskin rastgele IOPS'si, S: ardışık hızı, D: tek isteğin gecikmesi). Sayılar N = 4, R ≈ 82, S ≈ 162 MB/s içindir.

Tablonun üç satırı açıklama ister. **Aynada ardışık okuma** neden yalnız (N/2)·S'dir? İki kopya blokları paylaşabilir: bir disk çift numaralı blokları, öteki tek numaralıları okur. Ama her disk atladığı blokların da altından dönerek geçmek zorundadır; atlanan bloklar zaman kazandırmaz. **RAID 4'te küçük yazma** her seferinde eşlik diskinde bir okuma ve bir yazma ister. Dizinin hızı tek diskin yarısına iner ve disk eklemek hiç işe yaramaz. **RAID 5** aynı 4 G/Ç'yi bütün disklere yaydığı için küçük yazma hızı N·R/4 olur ve disk sayısıyla büyür. Tam bir şerit yazıldığında ise eski veri okunmaz; eşlik yeni verilerden hesaplanır ve hız (N − 1)·S olur.

Örnek 11.28

Tek disk 150 rastgele IOPS veriyor. 8 diskli dizide küçük rastgele yazma hızı nedir: RAID 0, RAID 10 (4 aynalı çift), RAID 4, RAID 5, RAID 6? Disk sayısı 16'ya çıkarsa hangisi hiç hızlanmaz?

1. **RAID 0.** Yazma başına 1 G/Ç: 8 × 150 = 1200 IOPS.
2. **RAID 10.** İki kopya, 2 G/Ç: 1200 / 2 = 600.
3. **RAID 4.** 4 G/Ç, ama ikisi hep eşlik diskinde: 150 / 2 = 75.
4. **RAID 5.** 4 G/Ç, disklere yayılır: 1200 / 4 = 300.
5. **RAID 6.** 6 G/Ç (3 oku + 3 yaz): 1200 / 6 = 200.
6. **16 disk.** RAID 4 yine 75'te kalır; darboğaz tek eşlik diskidir. RAID 5 600'e çıkar.

11.12.6 Kararlı depolama

RAID bozulan sektörlere ve disklere karşı korur, ama yazma sırasında bir çökme olursa ne olacağına cevap vermez. Aynalı iki diske aynı anda yazılırken güç kesilirse iki kopya da yarım kalabilir.

Tanım 11.38

Kararlı depolama (*stable storage*), bir yazmanın ya doğru biçimde tamamlandığını ya da eski verinin yerinde kaldığını garanti eden depolama düzenidir. Tanenbaum'un modeli üç varsayıma dayanır: iki diskte aynı bloğun aynı anda bozulma olasılığı ihmal edilir; hatalı veri ECC ile fark edilir; CPU çökerse süren yazma durur.

Düzen iki diskle ve üç işlemle kurulur. **Kararlı yazma**: blok önce disk 1'e yazılır, geri okunur ve doğrulanır; gerekirse n kez yeniden denenir, olmazsa yedek sektöre yazılır. Disk 1 başarıyla yazılınca aynı işlem disk 2'ye yapılır. **Kararlı okuma**: blok disk 1'den okunur; geçerli ECC ile okunamazsa disk 2'den okunur. **Kurtarma**: çökmeden sonra iki diskteki bloklar karşılaştırılır ve tutarsızlık giderilir.



Şekil 11.58: Çökmeden sonra kurtarma. Üstte çökme disk 1'e yazılırken oldu: disk 1 bozuk (ECC), eski değer disk 2'den geri getirilir. Altta disk 1 yazıldı, disk 2 yazılmadan çöktü: yeni değer disk 2'ye kopyalanır.

Örnek 11.29

Kararlı yazmanın beş anında çökme olursa kurtarma ne yapar?

1. **Disk 1'e yazmadan önce.** İki kopya da eskidir. Bir şey yapılmaz; yazma hiç olmamış sayılır.
2. **Disk 1'e yazarken.** Disk 1'deki blok yarım kalmıştır ve ECC hatası verir. Eski değer disk 2'den disk 1'e kopyalanır.
3. **Disk 1 yazıldı, disk 2'ye başlanmadı.** İki kopya da geçerli ama farklıdır. Kural: disk 1 kazanır. Yeni değer disk 2'ye yazılır ve yazma tamamlanır.
4. **Disk 2'ye yazarken.** Disk 2 bozuktur (ECC); yeni değer disk 1'den kopyalanır.
5. **İki kopya da yeni.** Yazma tamamlanmıştır; bir şey yapılmaz.

Her durumda sonuç ya tamamen eski ya tamamen yenidir. Bunu sağlayan yazma sırasının sabit olmasıdır. Herhangi bir anda en az bir kopya ya tamamen eski ya tamamen yeni ve geçerlidir; kurtarma o kopyayı ötekine yayar. RAID 1 bunu tek başına sağlamaz, çünkü iki diske aynı anda yazar ve hangisinin "doğru" olduğuna dair bir sıra tutmaz.

Aynı fikir bugün başka biçimlerde yaşar. Dosya sistemleri değişiklikleri önce bir günlüğe yazar (*journaling*) ve ancak günlük kalıcı olunca asıl yerlerine işler. Yazma bariyerleri ve FUA, belirli yazmaların diskte kalıcı olduğundan emin olmayı sağlar. Kurumsal SSD'lerde güç kaybı koruma kapasitörleri, elektrik kesildiğinde önbellekteki veriyi flaşa yazmaya yetecek enerjiyi saklar.

11.13 Özet

- İşletim sistemi aygıtla değil, **aygıt denetleyicisiyle** konuşur. Denetleyicinin durum, komut, veri giriş ve veri çıkış yazmaçları vardır: yazmaca yazmak komut vermek, okumak durum öğrenmektir.
- Yazmaçlara **kapı eşlemeli** G/Ç'de özel `IN` / `OUT` komutlarıyla, **bellek eşlemeli** G/Ç'de sıradan yükleme ve saklama komutlarıyla erişilir. MMIO'da yazmaç sayfası önbelleklenemez eşlenmeli, işaretçi `volatile` olmalıdır. Bugün PCIe aygıtları BAR'larla MMIO kullanır.
- Üç aktarım yöntemi vardır. **Yoklamada** CPU durum yazmacını döngüyle bekler; **kesme güdümlü** G/Ç'de her birim için bir kesme gelir; **DMA**'da CPU yalnız kurulumu ve bitiş kesmesini işler. 4096 karakterlik örnekte CPU maliyetleri 4,096 s, 8,19 ms ve yaklaşık 7 µs'dir.
- Kesme geldiğinde CPU komut sınırında durur, donanım PC ve PSW'yi kaydeder, vektör IDT'de indis olur, işleyici aygıtı onaylar ve bekleyeni uyandırır, çizelgeleyici seçer, `iret` ile dönülür. Modern aygıtlar kesmeyi **MSI-X** ile bellek yazması olarak gönderir. İşleyici üst yarı ve alt yarı olarak ikiye bölünür.
- **Dağıt-topla** bitişik olmayan sayfalara tek DMA işlemiyle yazar; **IOMMU** aygıt adreslerini çevirir ve aygıtları birbirinden yalıtır.

- G/Ç yazılımı katmanlıdır: kullanıcı kütüphanesi, aygıttan bağımsız yazılım (Linux'ta VFS, sayfa önbelleği, dosya sistemi, blok katmanı), aygıt sürücüsü ve kesme işleyicisi. Sürücüler ortak bir işlev tablosu (`file_operations`) sunar.
- **Tamponlama** hız farkını, boyut farkını ve kopya anlamını karşılar. **Dairesel tampon** N yuvada en çok N − 1 bayt tutar: baş = kuyruk boş, (kuyruk + 1) mod N = baş dolu demektir. Tek üretici ve tek tüketicide kilit gerekmez.
- **Bloke eden** çağrı veri gelene kadar dönmez; **bloke etmeyen** çağrı ne varsa onu ya da `EAGAIN` döndürür; **eş zamansız** çağrı işi başlatır ve sonra haber verir. epoll hazır olmayı, io_uring tamamlanmayı bildirir.
- Saat donanımı her darbede bir sayacı azaltır ve sıfırda kesme üretir. **Tiksiz çekirdek** boşta tiki durdurur. Bekleyen zamanlayıcılar **fark listesinde** tutulur: tik başına iş O(1), ekleme O(n)'dir.
- Disk erişim süresi = arama + dönme gecikmesi + aktarım. 7200 rpm, 8 ms arama ve 200 MB/s ile rastgele 4 KiB okuma 12,19 ms (≈ 82 IOPS), ardışık 1 MiB okuma 17,41 ms (≈ 60 MB/s) sürer.
- Disk kolu çizelgelemede ortak kuyruğun (kafa 53) toplam hareketleri: FCFS 640, SSTF 236, SCAN 236, C-SCAN 382 (dönüşsüz 183), LOOK 208, C-LOOK 322 (sıçramasız 153). SSTF açlık yaratabilir; tarama algoritmaları yaratmaz. **SPTF** dönme gecikmesini de hesaba katar ve diskin içinde NCQ ile uygulanır.
- **SSD**'de arama yoktur ve yerinde güncelleme yapılamaz. FTL yer dışı güncelleme, çöp toplama ve **aşınma dengeleme** yapar; TRIM yazma büyütmesini azaltır. NVMe çekirdek başına kuyruk sunar; Linux blk-mq bu modeli izler ve NVMe'de varsayılan çizelgeleyici `none` 'dır.
- **RAID** şeritleme, aynalama ve eşlikle diskleri birleştirir. Eşlik XOR'dur; kaybolan blok kalanların XOR'udur. RAID 5'te küçük yazma 4 G/Ç, RAID 6'da 6 G/Ç ister. RAID 4'te eşlik diski darboğazdır. RAID yedek değildir.
- **Kararlı depolama** iki diske sırayla yazarak her çökmeden sonra ya tamamen eski ya tamamen yeni veriyi garanti eder. Bugün aynı fikri günlükleme, FUA ve güç kaybı koruması sürdürür.

11.14 Kavram listesi

veri yolu *bus* · aygıt denetleyicisi *device controller* · blok / karakter aygıtı *block / character device* · bellek eşlemeli G/Ç *memory-mapped I/O* · yoklama *polling* · meşgul bekleme *busy waiting* · programlanmış G/Ç *programmed I/O* · kesme *interrupt* · kesme vektörü *interrupt vector* · kesme işleyicisi *interrupt handler* · istisna *exception* · tuzak *trap* · boru hattı *pipeline* · kesme güdümlü G/Ç *interrupt-driven I/O* · doğrudan bellek erişimi (DMA) *direct memory access* · aygıt sürücüsü *device driver* · tampon / tamponlama *buffer / buffering* · sayfa önbelleği *page cache* · biriktirme *spooling* · bloke eden / bloke etmeyen / eş zamansız G/Ç *blocking / nonblocking / asynchronous I/O* · zamanlayıcı *timer* · sabit disk (HDD) *hard disk drive*

arama süresi / dönme gecikmesi *seek time / rotational latency* · disk kolu çizelgeleme *disk scheduling* · SSD *solid-state drive* · aşınma dengeleme *wear leveling* · G/Ç çizelgeleyicisi *I/O scheduler* · RAID · eşlik *parity* · günlükleme *journaling*

11.15 Kendini Dene

1. Silindirler 0–199; kafa 100'de ve yukarı yönde gidiyor. İstekler geliş sırasıyla 55, 58, 39, 18, 90, 160, 150, 38, 184. FCFS, SSTF, LOOK ve C-LOOK (sıçrama sayılsın) toplam hareketi nedir?

   Cevap

   FCFS: 45 + 3 + 19 + 21 + 72 + 70 + 10 + 112 + 146 = 498. SSTF: 90, 58, 55, 39, 38, 18, 150, 160, 184 sırasıyla 10 + 32 + 3 + 16 + 1 + 20 + 132 + 10 + 24 = 248. LOOK: 100 → 184 (84), 184 → 18 (166): 250. C-LOOK: 100 → 184 (84), sıçrama 184 → 18 (166), 18 → 90 (72): 322.

2. 5400 rpm dönen bir diskin ortalama arama süresi 10 ms, ortam hızı 100 MB/s (1 MB = 10^6 B). 8 KiB'lık rastgele bir okuma ortalama ne kadar sürer? Disk saniyede yaklaşık kaç rastgele okuma yapabilir?

   Cevap

   Bir tur 60.000 / 5400 ≈ 11,11 ms; ortalama dönme gecikmesi 5,56 ms. Aktarım 8192 / 10^8 s ≈ 0,08 ms. Toplam 10 + 5,56 + 0,08 ≈ 15,64 ms; saniyede 1000 / 15,64 ≈ 64 okuma.

3. N = 4 yuvalı dairesel tampon boş. Çağrılar sırayla: koy x, koy y, al, koy z, koy w, koy v, al, koy u. Hangi çağrı başarısız olur? Sonunda baş ve kuyruk nedir?

   Cevap

   x yuva 0'a, y yuva 1'e yazılır (kuyruk 2). al x'i okur (baş 1). z yuva 2'ye, w yuva 3'e yazılır; kuyruk başa sarar (0). koy v'de (0 + 1) mod 4 = 1 = baş: tampon dolu, −1 döner. al y'yi okur (baş 2). u yuva 0'a yazılır (kuyruk 1). Sonuç: baş = 2, kuyruk = 1; tamponda z, w, u vardır.

4. Fark listesi A 5, B 2, C 6. 9 tik sonra dolacak X eklenirse liste ne olur? `ekle` kaç düğüm geçer?

   Cevap

   A'nın farkı 5 ≤ 9 → t = 4; B'nin farkı 2 ≤ 4 → t = 2; C'nin farkı 6 > 2, döngü durur. X 2 farkıyla C'den önce girer, C'nin farkı 6 − 2 = 4 olur. Liste A 5, B 2, X 2, C 4. İki düğüm geçilir. C'nin mutlak zamanı 13 tik sonrası olarak kalır.

5. RAID 5 şeridinde D0 = 0110 1101, D2 = 1001 0011, P = 1010 0110. D1'in diski bozuldu. D1 nedir?

   Cevap

   D1 = D0 ⊕ D2 ⊕ P. D0 ⊕ D2 = 1111 1110; ⊕ P = 0101 1000. D1 = 0101 1000.

6. Bir ağ kartı 2000 paketlik bir patlama alıyor. Kesme güdümlü yöntemde her paket için 3 µs'lik bir kesme, DMA'lı halkada ise 10 µs kurulum ve patlama sonunda tek bir 3 µs'lik kesme gerekiyor. CPU maliyetleri nedir? Tek paketlik trafikte hangisi daha ucuzdur?

   Cevap

   Kesme güdümlü: 2000 × 3 = 6000 µs = 6 ms. DMA: 10 + 3 = 13 µs; yaklaşık 460 kat az. Tek pakette kesme güdümlü 3 µs, DMA 13 µs harcar: sabit kurulum maliyeti küçük aktarımda kendini geri ödemez.

7. Bir program diskteki bir dosyayı epoll ile izleyip "hazır" olduğunda okumak istiyor. Bu neden eş zamansız okuma sağlamaz? Hangi arayüz sağlar?

   Cevap

   epoll yalnız hazır olmayı bildirir. Disk dosyaları her zaman hazır görünür, çünkü çekirdek bir disk okumasını `EAGAIN` ile reddetmez, bloke ederek tamamlar. Okuma yine `read` içinde beklenir. İşi başlatıp tamamlanmayı sonra bildiren io_uring (ya da Windows'ta IOCP) gerçek eş zamansızlık sağlar.

8. Bir SSD'de dosya sistemi TRIM göndermiyor. Kullanıcı büyük bir dosyayı sildikten sonra disk yeniden doldukça neden yavaşlar ve neden daha çabuk yıpranır?

   Cevap

   SSD silinen dosyanın sayfalarını hâlâ geçerli sanır. Çöp toplama bir bloğu temizlerken bu sayfaları da başka bloğa kopyalar. Kopyalanan her sayfa ek bir yazmadır: yazma büyütmesi artar, yazmalar çöp toplamayı bekler ve hücreler gereksiz yere silinip yazılır.

9. Bir sürücü durum yazmacını `int *durum` işaretçisiyle okuyan bir bekleme döngüsü içeriyor. Optimizasyon açılınca döngü hiç bitmiyor. Neden? İki düzeltme nedir?

   Cevap

   Derleyici `*durum`'un döngüde değişmediğini varsayar ve okumayı bir kez yapar. İşaretçi `volatile int *` yapılmalıdır. Ayrıca yazmaç sayfası önbelleklenemez eşlenmelidir (Linux'ta `ioremap`); yoksa CPU önbellekteki eski değeri okur.

10. Kararlı depolamada iki kopyanın ikisi de geçerli ECC veriyor ama değerleri farklı. Kurtarma ne yapar? Bu durum hangi anda çökme olduğunu gösterir?

    Cevap

    Disk 1 kazanır; değeri disk 2'ye kopyalanır ve yazma tamamlanır. Bu durum, disk 1'e yazmanın bitip doğrulandığını, disk 2'ye yazmanın ise henüz başlamadığını gösterir. Disk 2'ye yazarken çökülseydi disk 2'deki blok ECC hatası verirdi.

11.16 İleri okuma

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, "I/O Devices", "Hard Disk Drives", "Redundant Arrays of Inexpensive Disks" ve "Flash-based SSDs" bölümleri: disk zamanlamasını, RAID başarım modelini ve FTL'yi kısa ve hesaplı anlatır; ostep.org'da ücretsizdir.
- D. A. Patterson, G. Gibson ve R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", SIGMOD 1988: RAID düzeylerini ilk kez tanımlayan, bugün de okunabilir kısa makaledir.
- J. Axboe, "Efficient IO with io_uring" (kernel.dk): io_uring'in tasarımcısının arayüzü, halkaları ve SQPOLL kipini anlattığı belgedir.
- J. Corbet, A. Rubini ve G. Kroah-Hartman, *Linux Device Drivers*, 3. baskı, O'Reilly, 2005: karakter sürücüleri, kesme işleme ve DMA üzerine örnekli klasik kaynaktır; çekirdek arayüzlerinin bir kısmı eskimiştir ama kavramlar geçerlidir.

11.17 Kaynaklar

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 11: Mass-Storage Structure; Bölüm 12: I/O Systems.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 5: Input/Output.
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "I/O Devices", "Hard Disk Drives", "RAID", "Flash-based SSDs".
4. D. A. Patterson, G. Gibson, R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", SIGMOD 1988.
5. NVM Express Base Specification 2.x; PCI-SIG PCIe 5.0/6.0; USB4 Specification; UEFI Specification (GPT).
6. Linux çekirdek belgeleri: "Multi-Queue Block IO Queueing Mechanism (blk-mq)", "NO_HZ: Reducing Scheduling-Clock Ticks", io_uring (J. Axboe, "Efficient IO with io_uring"); J. Corbet, A. Rubini, G. Kroah-Hartman, *Linux Device Drivers*, 3. baskı.

KISIM IV

İleri Konular

Bölüm 12–15

KISIM IV · BÖLÜM 12

Çok İşlemcili Sistemler

Bugün sıradan bir dizüstü bilgisayarda bile sekiz ya da on altı çekirdek bulunur. Sunucularda bu sayı yüzleri aşar; bir veri merkezinde ise binlerce bilgisayar tek bir iş üzerinde birlikte çalışır. Tek işlemci için tasarlanmış bir işletim sistemi bu donanımdan yararlanamaz. Çekirdeklerin aynı belleği nasıl paylaştığı, önbelleklerin nasıl tutarlı kaldığı, kilitlerin bu ortamda nasıl gerçeklendiği ve iş parçacıklarının işlemcilere nasıl dağıtıldığı yeni sorulardır. Not, önce paylaşımlı bellekli çok işlemcilerin donanımını anlatır: veri yolu, çapraz çubuk ve omega ağı, NUMA, dizin tabanlı tutarlılık, MESI protokolü ve yanlış paylaşım. Ardından çok işlemcili işletim sistemi modellerine, kilitlere ve çizelgelemeye geçer. Belleği paylaşmayan çoklu bilgisayarlar, ileti geçirme, uzak prosedür çağrısı ve yük dengeleme bunu izler. Son kesimler sanallaştırmayı, konteynerleri, dağıtık sistemlerdeki ara yazılımı ve bulutu ele alır.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum12` · **Kod (Java):** `github.com/sercankulcu/operating-systems-java/Bolum12/src`

ÖĞRENME ÇIKTILARI

Donanım
UMA (veri yolu, çapraz çubuk, omega) ve NUMA mimarilerini karşılaştırmak; dizin tabanlı tutarlılığı bir adres üzerinde izlemek.

Önbellek tutarlılığı
MESI durum geçişlerini bir okuma/yazma dizisine uygulamak; yanlış paylaşımı bulup gidermek.

İşletim sistemi ve kilitler
Ayrı işletim sistemi, usta-köle ve SMP modellerini; TSL, bilet, TTAS ve MCS kilitlerini; dönme ile geçişi değerlendirmek.

Çizelgeleme
Ortak hazır kuyruğunun sınırlarını, alan paylaşımını ve çete çizelgelemeyi bir örnek üzerinde uygulamak.

Çoklu bilgisayarlar
Topolojilerin çapını hesaplamak; ileti geçirme, RPC, DSM ve yük dengeleme yöntemlerini açıklamak.

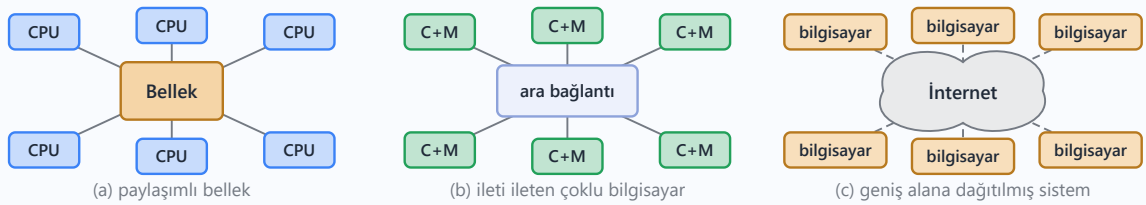
Sanallaştırma ve dağıtık sistemler
Hipervizör türlerini, VT-x ve EPT'yi, konteynerleri; ara yazılım türlerini ve bulutu tanımak.

12.1 Neden çok işlemci?

Bir bilgisayarı hızlandırmanın en eski yolu saat hızını artırmaktı. 1980'lerden 2000'lerin başına kadar işlemcilerin saat hızı her birkaç yılda bir ikiye katlandı. Bu yol 2005 dolaylarında tıkandı. Bugün de masaüstü işlemcilerin saat hızı 3 ile 5 GHz arasındadır. Yarı iletken üreticileri her yıl daha çok transistörü bir yongaya sığdırmayı sürdürüyor. Ama bu transistörler artık daha hızlı tek bir çekirdeğe değil, daha çok çekirdeğe harcanıyor. Bu kesim, paralel sistemlerin üç temel düzenini ve paralelliğin getirdiği kazancın sınırını anlatır.

12.1.1 Üç paralel düzen

Birden çok işlemciyi bir arada çalıştırmanın üç ana yolu vardır (Şekil 12.1). Birincisinde bütün işlemciler tek bir ortak belleğe bağlıdır. Her işlemci her bellek sözcüğünü sıradan `LOAD` ve `STORE` komutlarıyla okuyup yazabilir. Buna **paylaşımlı bellekli çok işlemci** (*shared-memory multiprocessor*) denir. İkincisinde her işlemcinin kendi belleği vardır ve işlemciler hızlı bir ara bağlantıyla birbirine bağlanır. Bir işlemci başkasının belleğini doğrudan okuyamaz; veri ancak iletiyle gönderilir. Bu düzene **çoklu bilgisayar** (*multicomputer*) ya da küme denir. Üçüncüsünde tam bilgisayarlar, yani kendi diski, çevre birimleri ve işletim sistemi olan makineler, İnternet gibi geniş alanlı bir ağla bağlanır. Bu da **dağıtık sistemdir**.



Şekil 12.1: Üç paralel düzen. (a) Paylaşımlı bellekli çok işlemci, (b) ileti geçiren çoklu bilgisayar (C+M: CPU ve bellek), (c) geniş alana dağıtılmış sistem.

Üç düzen arasındaki fark bağlantının sıkılığıdır. Çok işlemcide iletişim nanosaniyeler düzeyinde bir bellek erişimidir. Çoklu bilgisayarda bir ileti birkaç mikrosaniye sürer. Dağıtık sistemde iletiler milisaniyeler alır. Bağlantı gevşedikçe sistem büyütülebilir, ama programlamak ve yönetmek zorlaşır.

Paylaşımlı bellekli çok işlemciler kendi içinde de ikiye ayrılır. **Simetrik çok işlemcide** (*symmetric multiprocessor*, SMP) bütün işlemciler belleğe ve giriş/çıkış (G/Ç) aygıtlarına eşit hakla erişir. Çekirdek kodu her işlemcide çalışabilir. **Asimetrik çok işlemcide** (AMP) işlemciler eşit değildir. Ya iş-

letim sistemini yalnız biri yürütür ya da her işlemci ayrı bir yazılım çalıştırır. Bugünkü masaüstü ve sunucu işletim sistemlerinin hepsi SMP'dir. Asimetrik düzen daha çok gömülü yongalarda karşımıza çıkar.

Çok işlemci işletim sistemine yeni işler yükler. Hangi iş parçacığı hangi işlemcide çalışacak? Aynı çekirdek tablosuna iki işlemci aynı anda dokunursa ne olur? İşlemciler arasında veri nasıl taşınır? Paylaşılan verinin kopyaları nasıl tutarlı tutulur? Bu bölümün geri kalanı bu soruların yanıtlarıdır.

12.1.2 Saat hızının sınırı ve Amdahl yasası

Saat hızının neden durduğunu anlamak için güç tüketimine bakmak gerekir. Uzun yıllar boyunca transistörler küçüldükçe çalışma gerilimi de düştü. Böylece birim alandaki güç sabit kaldı. Bu gözleme **Dennard ölçeklemesi** denir. 2000'lerin ortasında gerilim daha fazla düşürülemez hâle geldi. Saat hızını artırmak artık yonganın aşırı ısınması demekti. Üreticiler de transistör bütçesini daha çok çekirdeğe harcamaya başladı.

Çekirdek sayısını artırmak her programı aynı oranda hızlandırmaz. Programın bir kısmı ancak sırayla yürütülebilir: girdiyi okumak, sonuçları birleştirmek, ortak bir yapıyı kilitle korumak gibi. Bu sıralı kısım, çekirdek sayısı ne olursa olsun aynı sürede biter.

Teorem 12.1 (Amdahl yasası)

Bir programın tek işlemcideki çalışma süresinin p kesri paralelleştirilebiliyor, 1 − p kesri sıralı kalıyorsa, n işlemcideki hızlanma

> $S(n) = 1 / ((1 - p) + p / n)$

olur. n sonsuza giderken p / n sıfıra yaklaşır; hızlanma hiçbir zaman 1 / (1 − p) değerini aşamaz.

Formülün nereden geldiği kolayca görülür. Tek işlemcide süre 1 birim olsun. Sıralı kısım n işlemcide de 1 − p sürer. Paralel kısım n işlemciye eşit bölünürse p / n sürer. Yeni süre bu ikisinin toplamıdır; hızlanma da eski sürenin yeni süreye oranıdır.

Örnek 12.1

Bir programın %95'i paralelleştirilebiliyor (p = 0,95). 8 ve 64 işlemcide hızlanma nedir? En fazla ne kadar hızlanabilir? 10 kat hızlanma için kaç işlemci gerekir?

1. **n = 8.** Paralel kısım 0,95 / 8 = 0,11875 sürer. Toplam süre 0,05 + 0,11875 = 0,16875 olur. Hızlanma 1 / 0,16875 ≈ 5,93'tür. Sekiz işlemci, beklenen 8 kat yerine yaklaşık 6 kat kazandırır.

2. **n = 64.** 0,05 + 0,95 / 64 = 0,05 + 0,0148 = 0,0648. Hızlanma ≈ 15,42'dir. İşlemci sayısı sekiz kat arttı, hızlanma üç kat bile artmadı.
3. **Sınır.** n büyüdükçe süre 0,05'e yaklaşır. Hızlanma 1 / 0,05 = 20'yi geçemez. 1024 işlemcide bile hızlanma ≈ 19,64'tür.
4. **10 kat için.** 0,05 + 0,95 / n = 0,1 olmalıdır. Buradan 0,95 / n = 0,05 ve n = 19 bulunur.

"İşlemci ekledikçe performans artar" sözü ancak sıralı kısım çok küçükse doğrudur. %5'lik sıralı bir kısım bile kazancı 20 katla sınırlar.

Amdahl yasası sabit bir problem boyutu varsayar. Gerçekte işlemci sayısı arttıkça çözülmek istenen problem de büyür: daha büyük bir benzetim, daha çok veri. Bu durumda sıralı kısmın payı küçülür ve kazanç Amdahl'ın öngördüğünden fazla olur. Bu bakışa Gustafson yasası denir. Yine de Amdahl'ın dersi geçerlidir: çekirdek sayısı arttıkça yazılımın sıralı kısmını küçültmek, çekirdek eklemekten daha önemli hâle gelir. İşletim sisteminin kendi kilitleri de bu sıralı kısmın bir parçasıdır. Tek bir çekirdek kilidi bütün işlemcileri sıraya sokar ve sistemin ölçeklenmesini sınırlar.

Sık yapılan hata

Hızlanmayı p × n diye hesaplamak. %95'i paralel bir program 64 işlemcide 0,95 × 64 ≈ 61 kat hızlanmaz. Sıralı kısım her işlemci sayısında aynı süreyi alır ve paydada sabit kalır; doğru değer yaklaşık 15,4'tür.

12.2 Çok işlemcili donanım

İşletim sisteminin çok işlemcide nasıl davranması gerektiğini anlamak için önce donanımın nasıl kurulduğunu bilmek gerekir. Bu kesim işlemcileri belleğe bağlamanın yollarını, bellek erişim süresinin neden her yerde aynı olmadığını ve önbelleklerin nasıl tutarlı tutulduğunu anlatır.

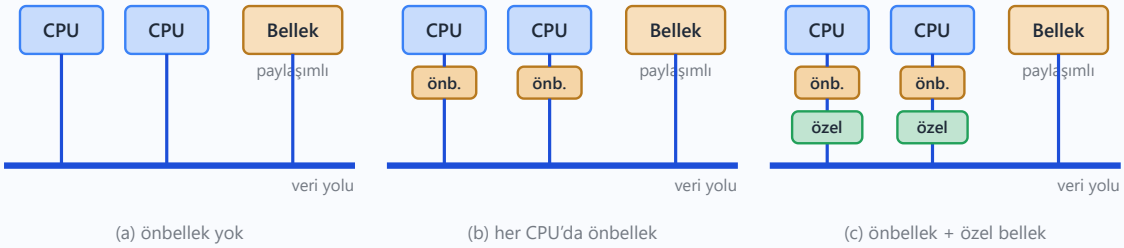
12.2.1 UMA: veri yolu tabanlı mimari

Tanım 12.1

UMA (*Uniform Memory Access*, tekdüze bellek erişimi) çok işlemcisinde her CPU her bellek sözcüğüne aynı sürede erişir. **NUMA** (*Non-Uniform Memory Access*) çok işlemcisinde ise erişim süresi sözcüğün hangi belleğe bağlı olduğuna göre değişir.

UMA programcı için en rahat düzendir. Verinin nereye yerleştiği performansı değiştirmez; bir program tek işlemcide nasıl yazılıyorsa çok işlemcide de öyle yazılabilir. En basit UMA makinesi bütün işlemcileri ve belleği tek bir **veri yoluna** bağlar (Şekil 12.2 (a)). Bir CPU bellekten okumak istediğinde

önce yolun boş olup olmadığına bakar. Yol boşsa adresi yola koyar, birkaç denetim sinyali verir ve bellek yanıtı yola koyana kadar bekler. Yol meşgulse boşalmasını bekler.



(a) önbellek yok (b) her CPU'da önbellek (c) önbellek + özel bellek

Şekil 12.2: Veri yolu tabanlı UMA çok işlemcileri: (a) önbelleksiz, (b) her CPU'da önbellek, (c) önbellek ve özel bellek.

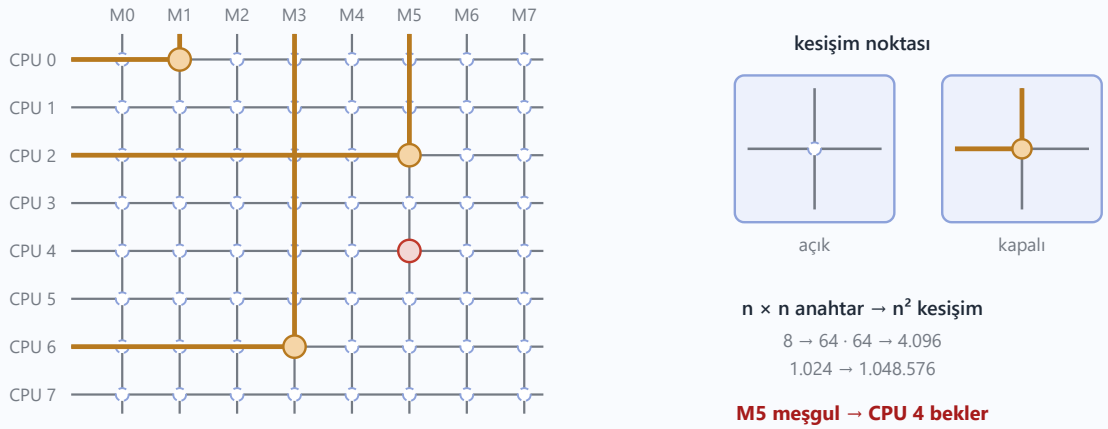
Bu düzen iki ya da üç işlemcide iyi çalışır. İşlemci sayısı arttıkça yol doyar. Otuz iki ya da altmış dört işlemcide CPU'lar zamanlarının çoğunu yolu beklemekle geçirir; yeni işlemci eklemek sistemi hızlandırmaz. (b) düzeninde her CPU'ya bir önbellek eklenir. Okumaların çoğu yerel önbellekte karşılanır ve yola çıkmaz; trafik büyük ölçüde azalır. Bunun bedeli yeni bir sorundur: aynı bellek satırının birden çok önbellekte kopyası bulunabilir. Bir CPU kendi kopyasını değiştirirse diğer kopyalar eskir. Bu kopyaları tutarlı tutmak donanımın görevidir ve ilerideki kesimlerde ayrıntılı anlatılır.

(c) düzeninde her CPU'nun bir de **özel belleği** vardır. Derleyici programın kodunu, sabitleri, yığını ve yerel değişkenleri bu özel belleğe koyar. Paylaşımlı bellek yalnız gerçekten paylaşılan değişkenler için kullanılır. Böylece yol trafiği daha da azalır. Ama derleyicinin hangi verinin paylaşıldığını bilmesi gerekir; bu da programlamayı zorlaştırır.

Bugünkü yongalarda tek bir paylaşılan veri yolu kalmadı. Çekirdekler, önbellek dilimleri ve bellek denetleyicileri yonga içinde bir halka ya da ızgara ağıyla bağlanır. Intel'in “mesh” ağı ve AMD'nin Infinity Fabric'i bunun örnekleridir. Düşünce yine aynıdır: ortak bir iletişim ortamı ve o ortamın sınırlı bant genişliğini korumak için büyük önbellekler.

12.2.2 Çapraz çubuk anahtar

Tek veri yolunun darboğazından kurtulmanın en doğrudan yolu her CPU'yu her bellek modülüne ayrı bir yolla bağlamaktır. **Çapraz çubuk anahtar** (*crossbar switch*) bunu bir ızgarayla yapar (Şekil 12.3). Yatay çizgiler CPU'ları, dikey çizgiler bellek modüllerini temsil eder. Her kesişim noktasında küçük bir elektronik anahtar vardır. Anahtar kapanınca o CPU ile o bellek modülü arasında doğrudan bir yol kurulur.



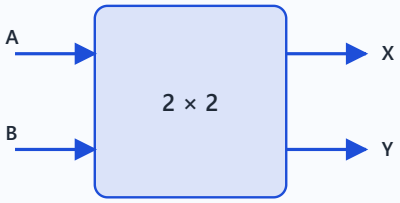
Şekil 12.3: 8 × 8 çapraz çubuk. CPU 0 → M1, CPU 2 → M5 ve CPU 6 → M3 bağlantıları aynı anda kurulu; M5 meşgul olduğu için CPU 4 bekler.

Şekilde önce CPU 2, M5'e erişmek ister: (2, 5) kesişim noktası kapanır ve yol kurulur. Ardından CPU 0 → M1 ve CPU 6 → M3 bağlantıları da aynı anda kurulur. Farklı bellek modüllerine giden istekler birbirini hiç beklemez. Bu nedenle çapraz çubuk **engellemesiz** (*non-blocking*) bir ağdır. CPU 4 de M5'i isterse ağda yol vardır, ama modülün kendisi meşguldür. Tek çekişme noktası bellek modülüdür, ağın kendisi değil.

Çapraz çubuğun bedeli boyutudur. n CPU ve n bellek modülü için n^2 kesişim noktası gerekir. 8 × 8 bir anahtarda 64 kesişim vardır; 64 × 64'te 4096, 1024 × 1024'te 1.048.576. Bu yüzden büyük çapraz çubuklar pahalıdır. Eski Sun Fire E25K sunucusu 18 × 18 bir çapraz çubuk kullanıyordu. Bugün çapraz çubuklar daha çok yonga içinde, çekirdekler ile L3 önbellek dilimleri arasında küçük ölçekte kullanılır.

12.2.3 Omega ağı

n^2 maliyetten kaçmak için ağ küçük anahtarlardan aşamalı olarak kurulabilir. Yapı taşı 2 × 2 anahtardır: iki girişi, iki çıkışı vardır. Bir girişten gelen ileti iki çıkıştan birine yönlendirilir. İleti dört alan taşır: hedef bellek modülü (Modül), modül içindeki adres, işlem kodu (okuma ya da yazma) ve yazılacak değer. Anahtar yalnız Modül alanına bakar (Şekil 12.4).



her giriş her çıkışa bağlanabilir
bit = 0 → üst çıkış (X)
bit = 1 → alt çıkış (Y)

ileti biçimi



Modül: hangi bellek modülü (yönlendirme etiketi)
Adres: modül içindeki sözcük
İşlem kodu: READ / WRITE
Değer: yazılacak sözcük (isteğe bağlı)

n CPU · n bellek → $\log_2 n$ aşama × n/2 anahtar
n = 8: 3 × 4 = 12 anahtar (çapraz çubukta 64 kesişim)
n = 1.024: 10 × 512 = 5.120 anahtar (1.048.576)

Şekil 12.4: 2 × 2 anahtar ve bir iletinin alanları. Anahtar yönlendirme kararını yalnız Modül alanından verir.

Omega ağı bu anahtarları $\log_2 n$ aşamaya dizer. 8 CPU ve 8 bellek için 3 aşama gerekir; her aşamada dört anahtar vardır (1A–1D, 2A–2D, 3A–3D). Aşamalar arasındaki kablolama **mükemmel karıştırmadır** (*perfect shuffle*). Bir çıkışın numarası bir sola döndürülerek sonraki aşamadaki girişin numarası bulunur. Örneğin 011 numaralı hat 110 numaralı girişe bağlanır.

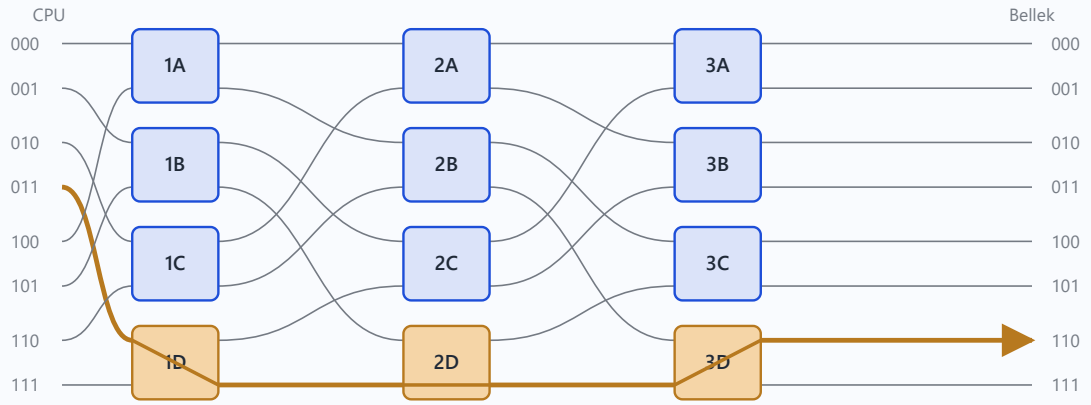
Yönlendirme kuralı çok basittir. k. aşamadaki anahtar, hedef adresin soldan k. bitine bakar. Bit 0 ise iletiyi üst çıkışa, 1 ise alt çıkışa gönderir. Kaynak adres yönlendirmeye hiç karışmaz.

Örnek 12.2

8 × 8 omega ağında CPU 011, bellek 110'u okumak istiyor. İleti hangi anahtarlardan geçer?

1. **Girişe varış.** 011 hattı karıştırmayla 110 numaralı girişe bağlanır. Bu giriş 1D anahtarına aittir (girişler 110 ve 111).
2. **1. aşama.** 1D hedefin 1. bitine bakar: 1. İleti alt çıkıştan çıkar. Bu çıkışın numarası 111'dir.
3. **2. aşama.** 111 karıştırmayla yine 111'e gider; ileti 2D'ye girer. Hedefin 2. biti 1'dir; ileti 2D'nin alt çıkışından çıkar.
4. **3. aşama.** İleti 3D'ye girer. Hedefin 3. biti 0'dır; ileti üst çıkıştan çıkar ve bellek 110'a varır.

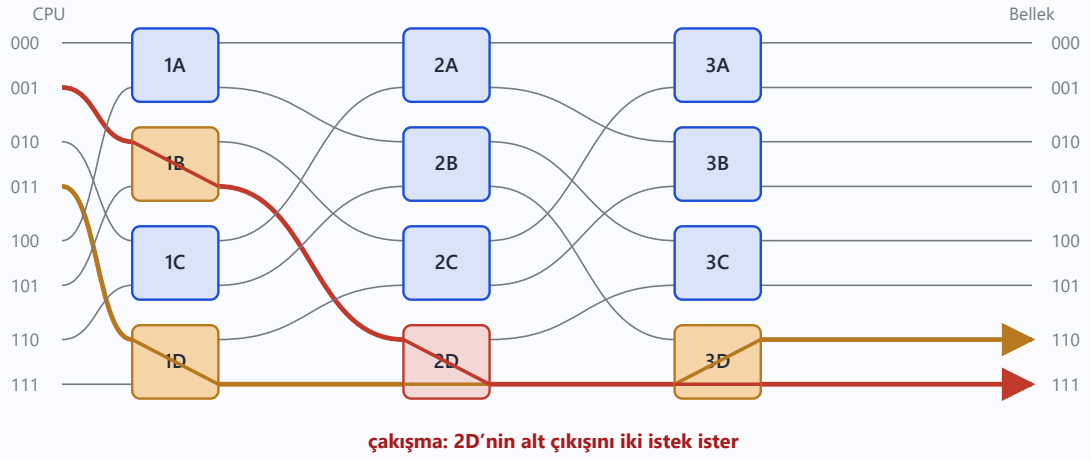
Yol 1D (alt) → 2D (alt) → 3D (üst) olur. (Bu yollar ve aşağıdaki çakışma, karıştırma kuralını uygulayan kısa bir Python benzetimiyle de doğrulandı.)



Şekil 12.5: CPU 011'den bellek 110'a giden istek. Her aşama hedef adresin bir bitini kullanır.

Yanıt aynı ağdan geri döner. Bunun için her anahtar, iletiyi geçirirken hangi girişten aldığını, yani kaynak adresin bir bitini Modül alanına yazar. İleti belleğe vardığında Modül alanında kaynak adres, 011, birikmiş olur. Bellek yanıtı bu etiketle geri gönderir ve yanıt aynı kuralla CPU 011'e ulaşır.

Omega ağı birçok aktarımı aynı anda taşıyabilir, ama her zaman değil. Aynı anda CPU 001'in bellek 111'e gitmek istediğini düşünelim. Bu istek 1B'den alt çıkışla çıkar ve karıştırmadan sonra 2D'ye girer. Hedefin 2. biti 1 olduğu için o da 2D'nin alt çıkışını ister. İki istek aynı çıkışa düşer ve biri beklemek zorunda kalır (Şekil 12.6). Bu yüzden omega ağı **engellemeli** (*blocking*) bir ağdır.



Şekil 12.6: 011 → 110 ve 001 → 111 istekleri 2D anahtarının alt çıkışında çakışır; biri bekler.

Omega ağı $n / 2 \times \log_2 n$ anahtar kullanır. 8 girişte $4 \times 3 = 12$ anahtar, 64 girişte $32 \times 6 = 192$, 1024 girişte $512 \times 10 = 5120$ anahtar gerekir. 1024×1024 bir çapraz çubuğun bir milyonu aşan kesişim noktasıyla karşılaştırıldığında bu çok küçük bir donanımdır. Bedeli ikidir: istekler çakışınca bekler ve her istek $\log_2 n$ aşamadan geçtiği için gecikme artar.

n	Çapraz çubuk: n^2 kesişim	Omega: $(n/2)\cdot\log_2 n$ anahtar	Omega aşama sayısı
8	64	12	3
64	4.096	192	6
1.024	1.048.576	5.120	10

Tablo 12.1: Çapraz çubuk ve omega ağının donanım maliyeti.

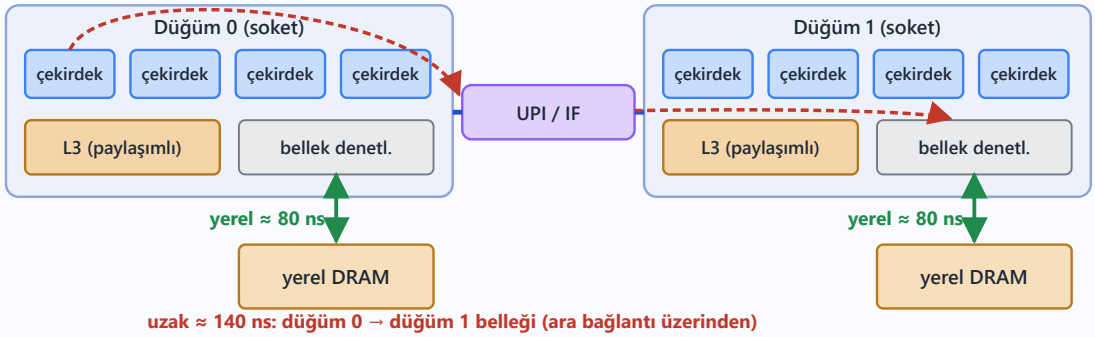
Sık yapılan hata

Omega ağında yönlendirmeyi kaynak adresin bitleriyle yapmak. Gidişte anahtarlar yalnız hedef adrese bakar; kaynak adresin bitleri ancak yanıtın geri dönebilmesi için yol boyunca kaydedilir. Bir diğer yanlış da omega ağını engellemesiz sanmaktır: iki istek aynı anahtar çıkışını isterse biri bekler.

12.2.4 NUMA çok işlemcileri

Veri yolu ya da anahtar ağıyla kurulan UMA makineleri birkaç düzine işlemcinin ötesine büyütülemez. Yüzlerce işlemcili makineler için bir ödün vermek gerekir: her bellek sözcüğüne aynı sürede erişme isteğinden vazgeçilir. Bu makinelere NUMA çok işlemcileri denir (Tanım 12.1).

Bir NUMA makinesinde bellek düğümlere bölünür. Her düğümde birkaç CPU ve onlara yakın bir bellek bulunur. Bugünkü iki soketli bir sunucuda her soket bir düğümdür (Şekil 12.7). Soketin çekirdekleri kendi bellek denetleyicisine bağlı DRAM'den yaklaşık 80 ns'de okur. Diğer soketin belleğine ise soketler arası bağlantı (Intel'de UPI, AMD'de Infinity Fabric) üzerinden, örneğin 140 ns'de ulaşır. Bu gecikmeler örnek değerlerdir; gerçek değerler makineden makineye değişir.



Şekil 12.7: İki düğümlü bir NUMA sunucusu. Her soket kendi belleğine hızlı, diğer soketin belleğine ara bağlantı üzerinden daha yavaş erişir.

NUMA makinelerinin üç ortak özelliği vardır. Birincisi, bütün CPU'lar tek bir fiziksel adres alanı görür. İkincisi, uzak belleğe de sıradan `LOAD` ve `STORE` komutlarıyla erişilir; program iletiler göndermez. Üçüncüsü, uzak erişim yerel erişimden yavaştır. NUMA makineleri önbellek tutarlılığı olup olmaması-

na göre ikiye ayrılır. Önbellek tutarlılığı olmayanlara **NC-NUMA**, olanlara **CC-NUMA** (*cache-coherent NUMA*) denir. Bugünkü sunucuların hepsi CC-NUMA'dır ve tutarlılığı çoğunlukla bir dizinle sağlar.

NUMA işletim sistemi için yeni bir sorumluluk demektir: veriyi onu kullanan işlemcinin yanında tutmak. Ortalama erişim süresi, erişimlerin ne kadarının yerel olduğuna bağlıdır.

> ortalama erişim = $y \cdot t_{\text{yerel}} + (1 - y) \cdot t_{\text{uzak}}$

Burada y, erişimlerin yerel kesridir.

Örnek 12.3

Yerel erişim 80 ns, uzak erişim 140 ns'dir. Erişimlerin %90'ı ve %50'si yerelken ortalama erişim süresi nedir?

1. **y = 0,9.** 0,9 · 80 + 0,1 · 140 = 72 + 14 = 86 ns.
2. **y = 0,5.** 0,5 · 80 + 0,5 · 140 = 40 + 70 = 110 ns.
3. **Karşılaştırma.** 110 / 86 ≈ 1,28. Yerel erişim oranı %90'dan %50'ye düşünce program yaklaşık %28 yavaşlar.

Bu kaybı önlemek için işletim sistemi üç şey yapar. İlki **ilk dokunma** (*first touch*) ilkesidir: bir sayfayı ilk kez hangi düğümdeki CPU kullanırsa sayfa o düğümün belleğinden ayrılır. İkincisi, çizelgeleyici bir iş parçacığını mümkün olduğunca verisinin bulunduğu düğümde çalıştırır. Yük dengelemesi önce düğüm içinde yapılır, düğümler arasına ancak gerekirse geçilir. Üçüncüsü, Linux'un **otomatik NUMA dengelemesi** bir sürecin hangi sayfalara hangi düğümden eriştiğini örnekleyerek izler. Sayfaları süreçlerin yanına ya da süreçleri sayfaların yanına taşır. Uygulama tarafında da `numactl` aracıyla bir süreç ve belleği belirli bir düğüme bağlanabilir.

NUMA yalnız çok soketli sunuculara özgü değildir. Tek soketli büyük yongalar da içeride NUMA olabilir. Örneğin AMD EPYC'de NPS (*nodes per socket*) ayarı, Intel'de SNC (*sub-NUMA clustering*) bir soketi birkaç NUMA düğümüne böler.

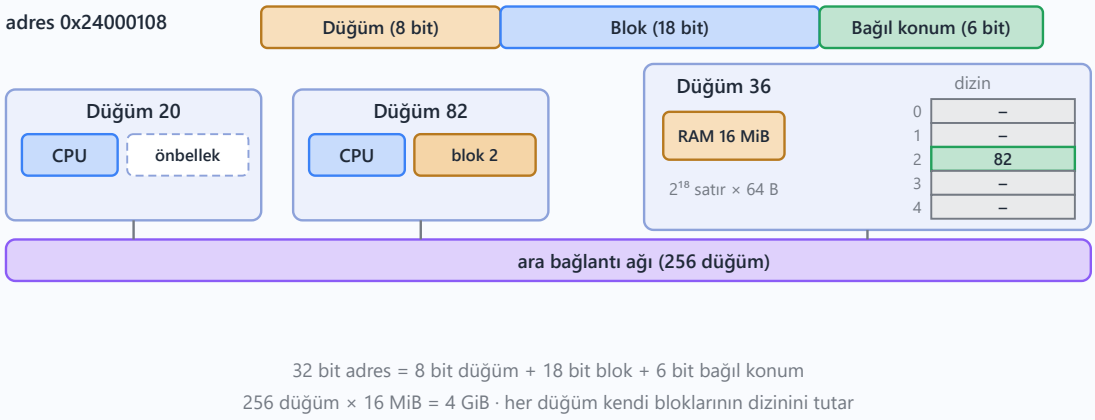
12.2.5 Dizin tabanlı CC-NUMA

Az sayıda işlemcide önbellekler ortak yolu dinleyerek tutarlı tutulabilir. Yüzlerce düğümde ise her isteği bütün düğümlere yaymak ağı boğar. Çözüm, her bellek satırının hangi önbelleklerde bulunduğunu bir kayıtta tutmaktır. Bu kayda **dizin** (*directory*) denir. Bir satır için ileti yalnız o satırın kopyasını tutan düğümlere gönderilir.

Tanenbaum'un örneği düşünceyi somutlaştırır. Sistemde 256 düğüm vardır. Her düğümde bir CPU ve 16 MiB bellek bulunur; toplam bellek 256 × 16 MiB = 2^{32} bayttır. Önbellek satırları 64 bayttır. 32 bitlik bir fiziksel adres üç alana bölünür (Şekil 12.8):

- en soldaki 8 bit **düğüm** numarasıdır (256 düğüm);

- sonraki 18 bit, o düğümün belleğindeki **blok** (satır) numarasıdır: 16 MiB / 64 B = 2^{18} satır;
- son 6 bit satır içindeki **bağıl konumdur** (64 bayt).



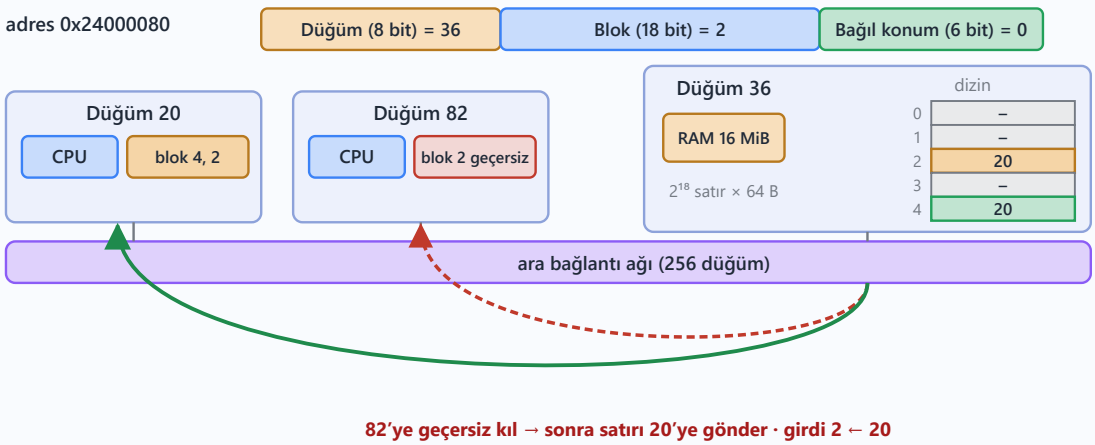
Şekil 12.8: 256 düğümlü dizin tabanlı CC-NUMA. Her düğüm kendi belleğindeki satırların dizinini tutar.

Her düğüm, kendi belleğindeki 2^{18} satırın her biri için bir dizin girdisi tutar. Bu basit tasarımda bir girdi, satır şu anda bir önbellekteyse o önbelleğin düğüm numarasını saklar. Bir satır aynı anda yalnız bir düğümde önbelleklenebilir.

Örnek 12.4

Düğüm 20'deki CPU önce `LOAD 0x24000108`, sonra `LOAD 0x24000080` komutlarını yürütüyor. Başlangıçta düğüm 36'nın dizininde girdi 2'de 82 yazıyor, girdi 4 boş. Her istekte ne olur?

1. **Adresin çözülmesi.** 0x24000108'in en soldaki baytı 0x24 = 36'dır: satır düğüm 36'nın belleğindedir. Kalan 24 bit 0x000108 = 264'tür; 264 = 4 · 64 + 8, yani blok 4, bağıl konum 8.
2. **İstek.** Adres yerel değildir. Düğüm 20'nin bellek denetleyicisi düğüm 36'ya "blok 4'ü istiyorum" iletisi gönderir.
3. **Dizin.** Düğüm 36 dizinine bakar: girdi 4 boştur, satır hiçbir önbellekte değildir. Güncel kopya düğüm 36'nın kendi RAM'indedir.
4. **Yanıt.** Düğüm 36 satırı düğüm 20'ye gönderir ve girdi 4'e 20 yazar: "bu satırın kopyası 20'de".
5. **İkinci istek.** 0x24000080'de düğüm yine 36'dır; 0x80 = 128 = 2 · 64 + 0 olduğu için blok 2, bağıl konum 0'dır. Dizinde girdi 2'de 82 yazar: satır düğüm 82'nin önbelleğindedir.
6. **Geçersiz kılma.** Düğüm 36, düğüm 82'ye "satırı geçersiz kıl" iletisi gönderir. Satır 82'de değiştirilmişse önce geri yazılır. Ardından satır düğüm 20'ye gönderilir ve girdi 2'ye 20 yazılır.



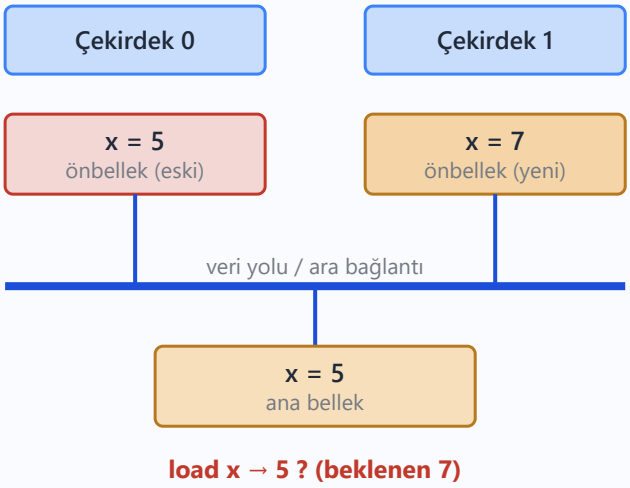
Şekil 12.9: 0x24000080 isteği. Düğüm 36 önce 82'deki kopyayı geçersiz kılar, sonra satırı 20'ye gönderir; dizin girdisi 2 artık 20'yi gösterir.

Bütün bu iletişim noktadan noktaya iletilerle yapılır. Bir veri yolunu bütün önbelleklerin dinlemesi gerekmez. Dizinin bedeli bellektir. Her girdi 8 bitlik bir düğüm numarası ve bir geçerli biti tutarsa 9 bit eder. Bir düğümün dizini $2^{18} \times 9$ bit = 294.912 bayt olur. Bu, 64 baytlık her satır için 9 bit, yani belleğin yaklaşık %1,76'sı kadar bir ek yüktür.

Bir satırın tek bir düğümde önbelleklenebilmesi gerçekçi değildir; sık okunan bir satırın birçok kopyası olmalıdır. Gerçek sistemler kopya kümesini her düğüm için bir bit içeren bir **bit vektörüyle** ya da sınırlı sayıda işaretçiden oluşan bir listeyle tutar. Çok çekirdekli yongaların içinde aynı işi "snoop filter" adı verilen yapılar görür: hangi satırın hangi çekirdeğin özel önbelleklerinde olduğunu izler.

12.2.6 Önbellek tutarlılığı sorunu

Her çekirdeğin kendi özel önbelleği vardır ve bu önbellekler aynı belleğin kopyalarını tutar. Şekil 12.10 sorunu gösterir. Bellekte x = 5'tir. İki çekirdek de x'i okuyup önbelleğine almıştır. Çekirdek 1, x'e 7 yazar. Yazma yalnız kendi önbelleğindeki kopyayı değiştirir. Çekirdek 0 şimdi x'i okursa kendi önbelleğindeki eski 5 değerini görür. Program açısından bu yanlıştır: x'e en son 7 yazılmıştı.



Şekil 12.10: Önbellek tutarsızlığı. Çekirdek 1 x'i 7 yaptı; çekirdek 0 kendi önbelleğindeki eski 5'i okur.

Tanım 12.2

Önbellek tutarlılığı (*cache coherence*), aynı adrese yapılan yazmaların bütün çekirdeklerce aynı sırada görülmesi ve bir okumanın o adrese en son yazılan değeri döndürmesi koşuludur.

Donanım tutarlılığı iki yoldan sağlar. **Dinleme** (*snooping*) yönteminde bütün önbellek denetleyicileri ortak yolu sürekli dinler. Bir önbellek, kendinde kopyası olan bir satıra başka biri yazacaksa bunu yoldan görür ve kopyasını günceller ya da geçersiz kılar. Bu yöntem az sayıda çekirdekte uygundur. **Dizin** yönteminde ise kopyaların listesi tutulur ve iletiler yalnız ilgili düğümlere gider. Bu yöntem çok çekirdekli yongalara ve NUMA makinelerine ölçeklenir.

Yazmalarla ilgili iki temel politika vardır. **Yazınca geçersiz kıl** (*write-invalidate*) politikasında bir çekirdek yazmadan önce diğer kopyaları geçersiz kılar; onlar bir sonraki okumada satırı yeniden ister. **Yazınca güncelle** (*write-update*) politikasında yeni değer bütün kopyalara gönderilir. Aynı satıra art arda yapılan yazmalarda güncelleme her seferinde trafik üretir, geçersiz kılma ise yalnız ilk yazmada. Bu yüzden bugünkü işlemcilerin hemen hepsi geçersiz kılma kullanır.

Tutarlılık tek bir adresle ilgilidir. Farklı adreslere yapılan yazmaların başka çekirdeklerce hangi sırada görüleceği ayrı bir konudur. Bu konuya **bellek tutarlılık modeli** (*memory consistency model*) denir. x86'nın modeli (x86-TSO) oldukça güçlüdür; ARM ve RISC-V daha zayıf modeller kullanır. Kilit gerçeklemeleri bu yüzden bellek bariyerleri içerir: kritik bölgedeki yazmaların, kilit bırakılmadan önce herkese görünmesini zorlar.

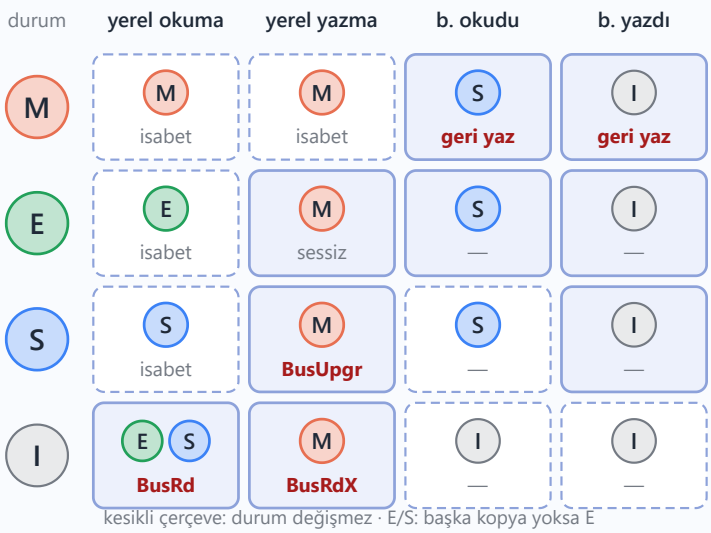
12.2.7 MESI protokolü

Yazınca geçersiz kılan protokollerin en yaygını MESI'dir. Adını bir önbellek satırının alabileceği dört durumun baş harflerinden alır (Tablo 12.2).

Durum	Anlamı
M (Modified, değiştirilmiş)	Satırın tek kopyası bu önbellektedir ve kirlidir: bellekteki değer eskidir. Satır önbellekten atılırken belleğe geri yazılır.
E (Exclusive, özel)	Tek kopya buradadır ve temizdir: bellekle aynıdır. Yazma, veri yoluna çıkmadan satırı M yapar.
S (Shared, paylaşılan)	Birden çok önbellekte temiz kopya olabilir. Yazmak için önce diğer kopyalar geçersiz kılınır.
I (Invalid, geçersiz)	Kopya geçersizdir; erişim önbellek ıskasıdır.

Tablo 12.2: MESI durumları.

Bir satırın durumu iki tür olayla değişir: kendi çekirdeğinin okuma ve yazmaları ile başka bir çekirdeğin yolda görülen istekleri. Şekil 12.11 bütün geçişleri bir arada gösterir. Veri yolunda üç tür istek dolaşır. `BusRd` bir satırı okumak için ister. `BusRdX` bir satırı yazmak amacıyla, özel olarak ister; diğer kopyalar geçersiz kılınır. `BusUpgr` ise S durumundaki bir satırı yazmak için yalnız geçersiz kılma ister; veri zaten elde olduğu için satır taşınmaz.



Şekil 12.11: MESI geçiş tablosu. Satırlar mevcut durum, sütunlar olay: yerel okuma, yerel yazma, başkasının okuması (b. okudu), başkasının yazması (b. yazdı). Hücrede yeni durum ve gerekiyorsa veri yolu işlemi yazar.

Tablonun okunuşu şöyledir. I durumundaki bir satır yerel olarak okunursa `BusRd` gönderilir. Başka bir önbellek "bende var" derse satır S, kimse demezse E olur. I durumundaki bir satıra yazılacaksa `BusRdX` gönderilir ve satır M olur. E durumundaki satıra yazmak sessizdir: başka kopya olmadığı bilindiği için yola hiçbir şey çıkmaz. S durumundaki satıra yazmak `BusUpgr` ister. M durumundaki bir satırı başka bir çekirdek okumak isterse sahibi veriyi yola koyar (*Flush*), belleğe de geri yazar ve satırı S yapar. Başka bir çekirdek yazmak isterse yine veriyi verir ve satırı I yapar.

E durumunun varlık nedeni sık görülen bir durumu ucuzlatmaktır: bir iş parçacığı özel verisini önce okur, sonra yazar. E olmasaydı okuma satırı S yapar, ardından gelen yazma gereksiz bir `BusUpgr` üretirdi. Bu protokolün ticari türevleri de vardır. Intel'in MESIF'inde F (*Forward*) durumu, paylaşılan

kopyalardan hangisinin istekleri yanıtlayacağını belirler. AMD'nin MOESI'sinde O (*Owned*) durumu kirli ama paylaşılan bir satırı gösterir; satır belleğe geri yazılmadan başka önbelleklere verilebilir.

12.2.8 MESI adım adım

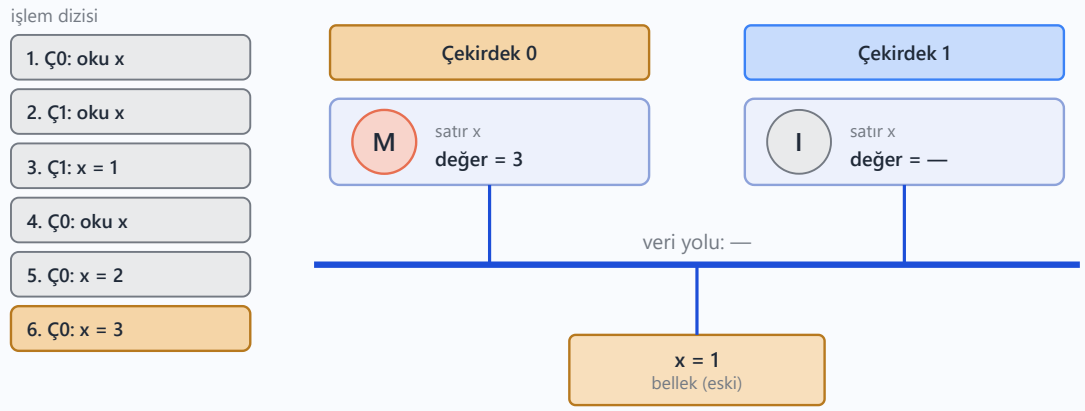
Protokolün nasıl işlediği bir işlem dizisi üzerinde en iyi görülür. İki çekirdekli bir sistemde x başlangıçta yalnız bellektedir ve değeri 0'dır. İki önbellekte de x'in satırı I durumundadır.

Örnek 12.5

Ç0 ve Ç1 sırayla şu işlemleri yapıyor: Ç0 x'i okur, Ç1 x'i okur, Ç1 x = 1 yazar, Ç0 x'i okur, Ç0 x = 2 yazar, Ç0 x = 3 yazar. Her adımdan sonra satırın iki önbellekteki durumu, bellekteki değer ve veri yolu iletisi nedir?

Adım	İşlem	Ç0	Ç1	Bellek	Veri yolu	Açıklama
0	—	I	I	0	—	x yalnız bellekte
1	Ç0 okur	E	I	0	BusRd	başka kopya yok: tek ve temiz
2	Ç1 okur	S	S	0	BusRd	Ç0 isteği dinler, "bende var" der
3	Ç1: x = 1	I	M	0	BusUpgr	S'den yazmak için geçersiz kılma; bellek artık eski
4	Ç0 okur	S	S	1	BusRd, Flush	Ç1 kirli satırı hem Ç0'a hem belleğe verir
5	Ç0: x = 2	M	I	1	BusUpgr	yine geçersiz kılma
6	Ç0: x = 3	M	I	1	—	zaten M: veri yolu trafiği yok

Altı işlemde beş veri yolu işlemi oluşur. Dizinin sonunda x'in güncel değeri 3'tür ve yalnız Ç0'ın önbelleğindedir; bellekte hâlâ 1 yazar. (Tablo, MESI kurallarını uygulayan bir Python benzetimiyle üretildi.)



Şekil 12.12: Dizinin son adımı. Ç0 satırı M durumunda tuttuğu için yeni yazma veri yoluna çıkmaz; bellekteki değer eskidir.

Son adım önemli bir sonucu gösterir. Bir çekirdeğin yalnız kendisinin kullandığı veriye art arda yazması ucuzdur: satır M durumunda kaldıkça hiçbir ileti üretmez. Pahalı olan, aynı satıra farklı çekirdeklerin sırayla yazmasıdır. Her yazma satırı bir önbellekten ötekine taşır. Bir sonraki kesimdeki yanlış paylaşım bu gözleme dayanır.

Sık yapılan hata

E ve S durumlarını karıştırmak. İkisi de temizdir, ama E "başka kopya yok" demektir; E'deki satıra yazma sessizce M olur. S'deki satıra yazmak için önce diğer kopyalar geçersiz kılınmalıdır. Bir başka yanlış da 4. adımda belleğin güncellenmediğini sanmaktır: M durumundaki satır başkasına verilirken belleğe de geri yazılır.

12.2.9 Yanlış paylaşım

Tutarlılık protokolleri değişkenleri değil önbellek satırlarını izler. Bugünkü işlemcilerde bir satır 64 bayttır. Birbirinden tamamen bağımsız iki değişken aynı satıra düşerse donanım onları paylaşılan veri gibi ele alır.

Tanım 12.3

Yanlış paylaşım (*false sharing*), farklı çekirdeklerin kullandığı farklı değişkenlerin aynı önbellek satırında bulunması yüzünden oluşan gereksiz tutarlılık trafiğidir.

Aşağıdaki kodda iki iş parçacığı iki ayrı sayacı artırır. Mantıkta hiçbir paylaşım yoktur ve kilit de gerekmez.

```
struct sayaclar {      /* kötü */
  long a;   /* bağıl konum 0 */
  long b;   /* bağıl konum 8 */
};

struct dolgulu {       /* iyi */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

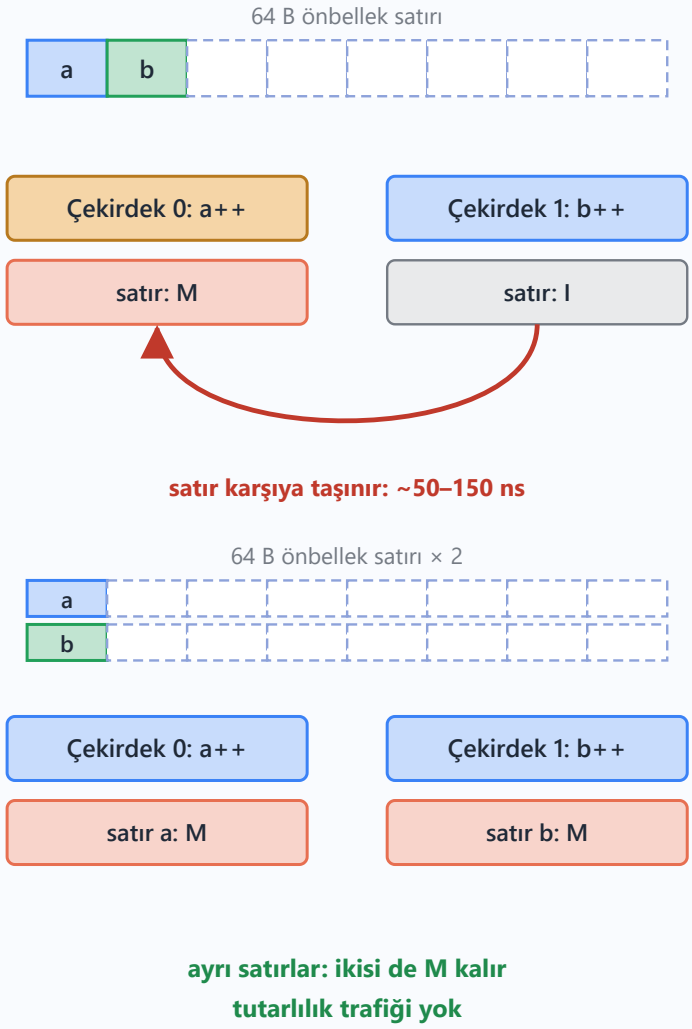
/* iş parçacığı 0: for(..) s.a++;
   iş parçacığı 1: for(..) s.b++; */
```

`struct sayaclar`'da `a` 0, `b` 8 numaralı bayttadır; ikisi de aynı 64 baytlık satıra düşer. İş parçacığı 0, Ç0'da `s.a++`'yı döngüde yürütür, iş parçacığı 1, Ç1'de `s.b++`'yı. Olaylar şöyle gelişir (Şekil 12.13):

1. Ç0 `a++` yapar. Satırı M durumunda alır; Ç1'in kopyası geçersiz olur.

2. Ç1 `b++` yapar. Satır Ç1'e taşınır ve orada M olur; Ç0'daki kopya geçersiz olur.
3. Ç0 yeniden `a++` yapar ve satırı geri çeker.

Her artırmada satır bir çekirdekten ötekine gider. Buna "ping-pong" denir. Bir satırın taşınması onlarca ile yüz elli nanosaniye arasında sürer; artırmanın kendisi ise birkaç döngüdür. Sonuçta iki iş parçacıklı program, aynı işi yapan tek iş parçacıklı sürümden bile yavaş çalışabilir.



Şekil 12.13: Üstte a ve b aynı satırda: her yazma satırı karşı çekirdeğe taşır. Altta dolgulu yapı: her sayaç ayrı satırda, ikisi de M durumunda kalır ve tutarlılık trafiği oluşmaz.

Çözüm her sayacı ayrı bir satıra koymaktır. `struct dolgulu`'da C11'in `_Alignas(64)` belirteci her alanı 64 baytlık sınıra hizalar. Yapının boyu 128 bayta çıkar ve sayaç başına 56 bayt boşa gider. Karşılığında ping-pong tamamen kalkar; kazanç çoğu zaman kat kat hızdır. Java'da aynı iş `@Contended` notuyla, Linux çekirdeğinde `____cacheline_aligned` makrosuyla yapılır. Çekirdeğin CPU başına sayaçları da bu nedenle her CPU için ayrı satırlarda tutulur.

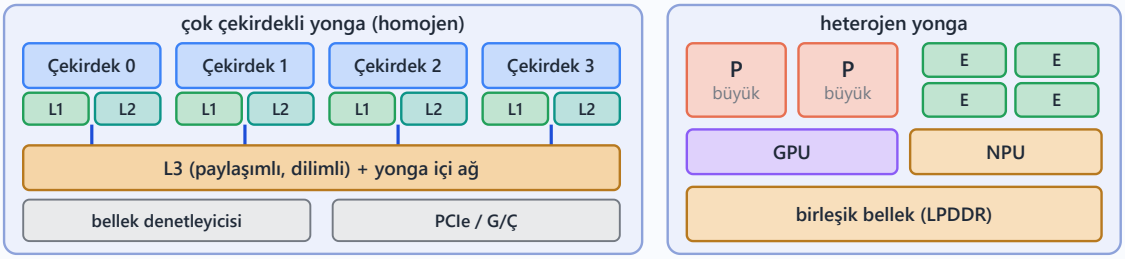
Sık yapılan hata

Yanlış paylaşımı yarış durumu sanıp mutex eklemek. Ayrı değişkenlerde yarış yoktur; sonuçlar doğrudur, yalnız program yavaştır. Mutex eklemek sorunu çözmez, aksine kilit satırı da çekirdekler arasında gidip gelmeye başlar. Doğru çözüm veriyi ayrı satırlara yerleştirmektir.

Aynı sorun daha büyük ölçekte de karşımıza çıkar. Dağıtık paylaşımlı bellekte tutarlılık birimi satır değil sayfadır. Bağımsız iki değişken aynı 4 KiB'lık sayfadaysa sayfa makineler arasında ağ üzerinden gidip gelir.

12.2.10 Çok çekirdekli ve heterojen yongalar

Bugünkü işlemcilerin çoğu tek bir yongada birden çok bağımsız çekirdek barındırır. Bu yongalara **çok çekirdekli** (*multicore*) ya da yonga üzerinde çok işlemci (*chip multiprocessor*, CMP) denir (Şekil 12.14). Tipik bir düzende her çekirdeğin özel L1 ve L2 önbellekleri vardır. L3 önbellek paylaşılır ve dilimler hâlinde yonga içi ağa dağıtılır. Sunucu soketlerinde çekirdek sayısı yüzleri bulur: AMD EPYC 9005 serisinde 192 çekirdek, Intel Xeon 6900E serisinde 288 verimlilik çekirdeği vardır.



Şekil 12.14: Solda homojen çok çekirdekli yonga: özel L1/L2, paylaşımlı L3, bellek denetleyicisi. Sağda büyük (P) ve küçük (E) çekirdekler, GPU ve NPU içeren heterojen yonga.

Çok çekirdekli yongaların işletim sistemini ilgilendiren dört özelliği vardır:

- **Yongacıklar** (*chiplets*): Büyük işlemciler artık tek parça değildir. Çekirdek grupları ayrı kalıplara konur ve bir paket içinde birleştirilir. Tek bir sokette bile bir çekirdeğin L3'e ve belleğe uzaklığı her yerde aynı değildir. Çizelgeleyici birlikte çalışan iş parçacıklarını aynı yongacıkta tutmaya çalışır.
- **Büyük–küçük çekirdekler**: ARM'ın big.LITTLE düzeni, Apple yongaları ve Intel'in P/E çekirdekleri hızlı ama çok güç harcayan çekirdekleri yavaş ama verimli olanlarla birleştirir. İşletim sistemi işi enerjiye göre yerleştirir. Linux'ta bu işi enerjiye duyarlı çizelgeleme (EAS) yapar. Intel'in Thread Director donanımı da Windows'a hangi iş parçacığının hangi çekirdeğe uygun olduğu konusunda ipuçları verir.
- **Hızlandırıcılar**: GPU ve yapay zekâ işlemleri için NPU aynı yongada bulunur ve aynı belleği paylaşır. Bunlar doğrudan çizelgelenmez; işletim sistemi onlara sürücü üzerinden iş gönderir.

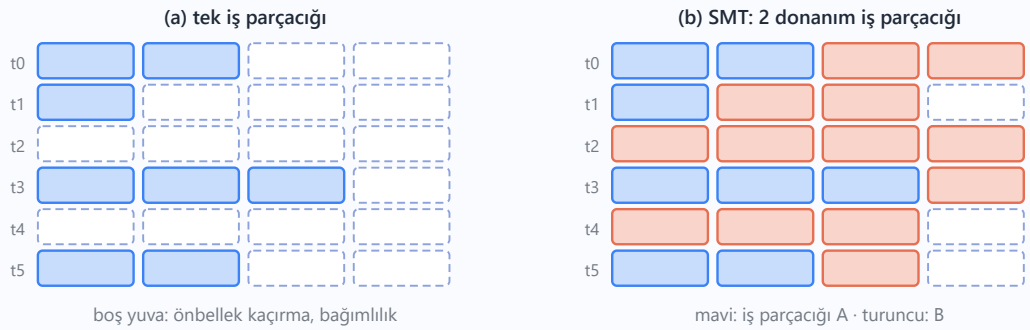
- **Paylaşılan kaynaklar**: L3 önbellek ve bellek bant genişliği ortaktır. Bir çekirdekteki bellek yoğun bir iş, başka bir çekirdekteki işi yavaşlatabilir.

12.2.11 Eş zamanlı çoklu iş parçacığı

Süperskalar bir çekirdek her döngüde birkaç komut yürütebilir. Ama tek bir iş parçacığı bu yuvaları nadiren doldurur. Bir önbellek ıskası ya da komutlar arasındaki bir bağımlılık çekirdeği bekletir ve yürütme birimleri boş kalır (Şekil 12.15 (a)).

Tanım 12.4

Eş zamanlı çoklu iş parçacığı (*simultaneous multithreading*, SMT), tek bir fiziksel çekirdeğin iki ya da daha çok donanım iş parçacığının komutlarını aynı döngüde yürütmesidir. Çekirdek işletim sistemine birden çok mantıksal CPU olarak görünür.



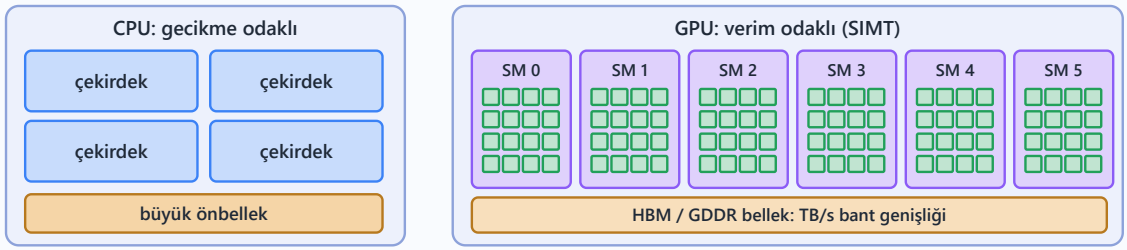
Şekil 12.15: (a) Tek iş parçacığında boş kalan yürütme yuvaları; (b) SMT'de iki iş parçacığı (mavi A, turuncu B) aynı yuvaları paylaşır.

Intel buna Hyper-Threading der; AMD de SMT adını kullanır. Mantıksal CPU'ların yazmaç kümeleri ayrıdır. Yürütme birimleri, L1 ve L2 önbellekleri ise ortaktır. Kazanç tipik olarak %10 ile %30 arasındadır. İki iş parçacığı aynı birimi, örneğin kayan nokta birimini isterse kazanç hiç olmayabilir. Bu nedenle çizelgeleyici işleri önce boş fiziksel çekirdeklere yerleştirir. Bir fiziksel çekirdeğin iki mantıksal CPU'sunu ancak bütün fiziksel çekirdekler doluyken birlikte kullanır.

SMT'nin bir de güvenlik yanı vardır. Aynı çekirdeği paylaşan iki iş parçacığı ortak önbellekler üzerinden birbirinin verisini sezebilir. L1TF ve MDS gibi yan kanal saldırıları bunu kullanır. Linux'un **çekirdek çizelgelemesi** (*core scheduling*) yalnız birbirine güvenen iş parçacıklarını aynı fiziksel çekirdeğe koyar. Bazı üreticiler SMT'den tamamen vazgeçmeye başladı; Intel'in Lunar Lake (2024) yongasının P çekirdeklerinde SMT yoktur.

12.2.12 GPU ve heterojen hesaplama

CPU ve GPU farklı amaçlar için tasarlanmıştır (Şekil 12.16). CPU az sayıda büyük ve karmaşık çekirdekten oluşur; büyük önbellekler ve dal tahmini gibi düzeneklerle tek bir iş parçacığının gecikmesini en aza indirmeye çalışır. GPU ise binlerce basit şerit içerir ve toplam verimi en çoklar. Şeritler 32'lik gruplar hâlinde aynı komutu yürütür. NVIDIA bu gruplara *warp*, AMD *wavefront* der. Bu düzen her veri öğesine aynı işlemin uygulandığı veri paralel işlere uygundur: grafik, yapay zekâ ve bilimsel hesap.



Şekil 12.16: Gecikme odaklı CPU ve verim odaklı GPU. GPU'nun akış çoklu işlemcileri (SM) yüksek bant genişlikli belleğe bağlıdır.

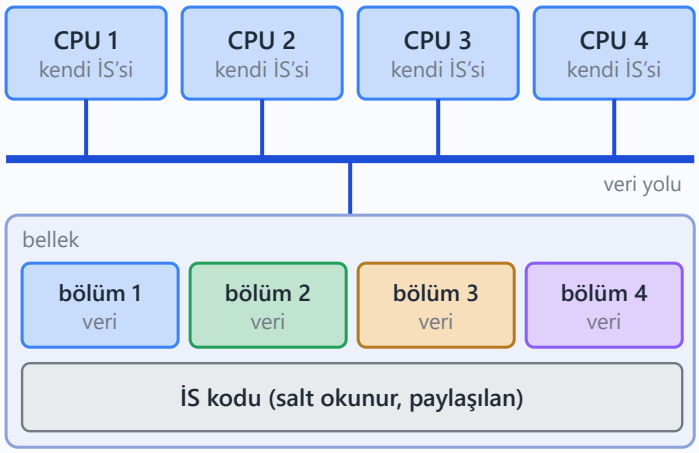
GPU programları CUDA, ROCm/HIP, OpenCL ya da SYCL ile yazılır. CPU, GPU üzerinde çalışacak işlevi (buna da çekirdek, *kernel*, denir; işletim sistemi çekirdeğiyle karıştırılmamalıdır) başlatır, veriyi GPU belleğine taşır ve sonucu bekler. İşletim sistemi ve sürücü bu sırada GPU belleğini yönetir ve farklı süreçlerin GPU bağlamlarını zaman paylaşımıyla çalıştırır. Bugünkü sistemler CPU ile GPU arasında birleşik ya da paylaşılan bir sanal bellek de sunar. GPU bir sayfaya eriştiğinde, IOMMU üzerinden sayfa hatası oluşabilir ve sayfa GPU'ya taşınır. Büyük yapay zekâ kümelerinde her düğümde 4–8 GPU bulunur. GPU'lar birbirine NVLink ile, düğümler birbirine InfiniBand ya da RoCE ile bağlanır ve veriyi RDMA ile doğrudan taşır.

12.3 Çok işlemcili işletim sistemi türleri

Donanım birden çok işlemciyi aynı belleğe bağladıktan sonra işletim sisteminin bu işlemcileri nasıl kullanacağına karar vermek gerekir. Tarihte üç model ortaya çıktı. Bu kesim üçünü sırayla anlatır ve karşılaştırır.

12.3.1 Her CPU'nun kendi işletim sistemi

En basit yol, belleği CPU sayısı kadar bölüme ayırmak ve her CPU'ya kendi bölümünde bağımsız bir işletim sistemi kopyası çalıştırmaktır (Şekil 12.17). İşletim sisteminin kodu salt okunur olduğu için paylaşılabilir; verisi ise her kopyada ayrıdır. Bir süreç sistem çağrısı yaptığında çağrı kendi CPU'sunda yakalanır ve o CPU'nun işletim sistemi onu işler.



Şekil 12.17: Her CPU'nun kendi işletim sistemi. Bellek dört bölüme ayrılmış; işletim sistemi kodu paylaşılıyor, verisi ayrı.

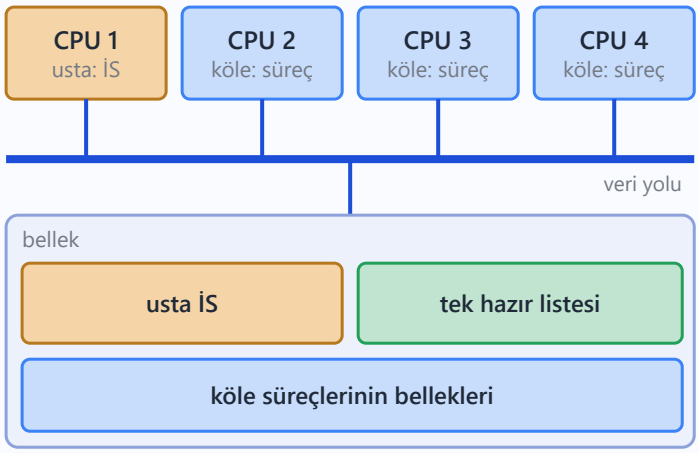
Bu modelin çekiciliği basitliğindedir. Tek işlemcili bir işletim sistemi neredeyse hiç değiştirilmeden kullanılabilir. Bölümler birbirinden bağımsız olduğu için aralarında kilit de gerekmez. Ama paylaşımın olmaması dört sorun doğurur:

- Süreçler paylaşılmaz. Her işletim sisteminin kendi süreç tablosu vardır. Bir CPU aşırı yüklüyken diğeri boşta kalabilir; iş birinden ötekine kolayca aktarılamaz.
- Sayfalar paylaşılmaz. Bir bölümde bellek tükenirken başka bir bölümde boş sayfa kalabilir.
- Her işletim sistemi kendi tampon önbelleğini tutar. Aynı disk bloğu iki önbellekte farklı hâllerde bulunabilir. İki kopya da değiştirilirse disk bloğu tutarsız olur. Bu sorundan kaçmanın tek yolu tampon önbelleklerini kaldırmaktır; bu da performansı düşürür.
- Bir CPU'nun işletim sistemi, diğerlerinin ne yaptığını bilmez; ortak bir çizelgeleme yapılamaz.

Bu model bugün başka adlarla yaşıyor. Büyük sunucuların donanım düzeyinde bölümlenmesi ve bir hipervizör üzerinde çalışan ayrı sanal makineler aynı düşünceye dayanır. Araştırmada ise Barrelfish işletim sistemi bunu bilinçli bir tasarım olarak kullanır. Bu "çoklu çekirdek" (*multikernel*) tasarımında her çekirdek, iletiyle haberleşen ayrı bir düğüm gibi ele alınır.

12.3.2 Usta-köle modeli

İkinci modelde işletim sisteminin tek bir kopyası ve bütün tabloları yalnız bir CPU'dadır; bu CPU'ya **usta** denir (Şekil 12.18). Diğer CPU'lar, yani **köleler**, yalnız kullanıcı süreçlerini çalıştırır. Bir köle CPU'daki süreç sistem çağrısı yaptığında çağrı ustaya yönlendirilir ve orada işlenir.



Şekil 12.18: Usta-köle modeli. İşletim sistemi ve tek hazır listesi ustadadır; köleler kullanıcı süreçlerini çalıştırır.

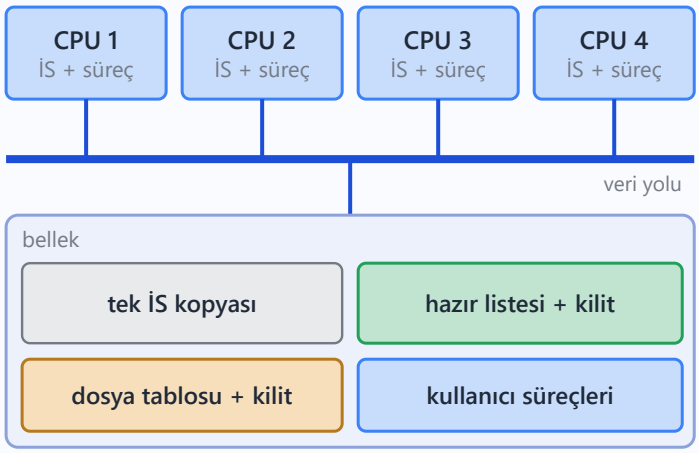
Usta-köle modeli ayrı işletim sistemlerinin sorunlarının çoğunu çözer. Tek bir hazır listesi vardır. Bir köle boşalınca ustadan yeni bir süreç ister; böylece yük kendiliğinden dengelenir. Bellek sayfaları tek bir havuzdan dağıtılır ve tek bir tampon önbelleği olduğu için tutarsızlık oluşmaz. Çekirdek verisine yalnız usta dokunduğu için kilit de gerekmez.

Modelin zayıf yanı ölçeklenememesidir. Bütün sistem çağrıları tek bir CPU'da işlenir. Sistem çağrısı yoğun bir iş yükünde usta darboğaz olur. Köle sayısı arttıkça çağrılar ustanın önünde sıraya girer ve yeni CPU eklemek az kazandırır. Usta çökerse de bütün sistem durur. Bu yüzden usta-köle modeli az sayıda CPU'su olan sistemlerde işe yarar.

Asimetrik çok işlemciliğin (AMP) klasik biçimi budur. Bugün asimetrik düzen daha çok gömülü yongalarda karşımıza çıkar. Örneğin Linux çalıştıran Cortex-A çekirdeklerinin yanında gerçek zamanlı bir işletim sistemi çalıştıran bir Cortex-M çekirdeği bulunur. Linux bu çekirdeği `remoteproc` altyapısıyla başlatır ve onunla OpenAMP üzerinden haberleşir.

12.3.3 Simetrik çok işlemciler

Üçüncü model ustanın darboğazını kaldırır. Bellekte işletim sisteminin tek bir kopyası vardır, ama her CPU onu çalıştırabilir (Şekil 12.19). Bir sistem çağrısı, çağrıyı yapan CPU'da işlenir. Yük ve bellek usta-kölede olduğu gibi dinamik olarak dengelenir. Bu modele **simetrik çok işlemci** (SMP) denir.



Şekil 12.19: SMP modeli. Tek işletim sistemi kopyasını bütün CPU'lar çalıştırır; hazır listesi ve dosya tablosu gibi paylaşılan yapılar kilitlerle korunur.

SMP'nin bedeli eş zamanlılıktır. İki CPU aynı anda aynı çekirdek tablosuna dokunabilir: biri hazır listesine bir süreç eklerken öteki aynı listeden süreç çıkarır. Bu yarışları önlemek için çekirdeğin paylaşılan yapıları kilitlerle korunmalıdır. En kolay ilk adım bütün çekirdeği tek bir kilitle korumaktır. Bir CPU çekirdeğe girerken bu kilidi alır, çıkarken bırakır. Linux 2.0 tam olarak böyle başladı; bu kilide "büyük çekirdek kilidi" (*Big Kernel Lock*, BKL) denir. Tek kilitli bir çekirdek, aslında usta-köle modelinin biraz daha esnek bir hâlidir: aynı anda çekirdekte yalnız bir CPU bulunabilir.

Ölçeklenebilmek için çekirdek, birbirinden bağımsız kritik bölgelere bölünür ve her bölge kendi kilidiyle korunur. Linux BKL'yi on yıldan uzun bir sürede adım adım küçülttü ve 2.6.39 sürümünde (2011) tamamen kaldırdı. Bugün çekirdekte binlerce ince taneli kilit vardır. Bunların yanında okuyucuların hiç kilit almadığı RCU (*read-copy-update*) düzeneği ve her CPU'nun yalnız kendi kopyasına dokunduğu CPU başına veri yapıları kullanılır.

İnce taneli kilitlerin kendi tehlikesi vardır. Bir kod yolu A ve B kilitlerini bu sırayla, başka bir yol ters sırayla alırsa iki CPU birbirini sonsuza dek bekleyebilir. Çekirdek geliştiricileri bunu önlemek için kilitlere sıra verir ve her kod yolunun kilitleri aynı sırayla almasını şart koşar. Linux'un `lockdep` aracı bu kuralın ihlallerini çalışma sırasında yakalar.

12.3.4 Üç modelin karşılaştırması

Tablo 12.3 üç modeli yan yana koyar.

	Ayrı işletim sistemi	Usta-köle	SMP
İşletim sistemi kopyası	CPU başına bir	bir (ustada)	bir (herkes çalıştırır)
Sistem çağrısı nerede?	yerel CPU	usta CPU	yerel CPU

	Ayrı işletim sistemi	Usta-köle	SMP
Hazır listesi	CPU başına	tek, ustada	tek ya da CPU başına + dengeleme
Yük dengeleme	yok	iyi	iyi
Ölçeklenebilirlik	iyi ama paylaşım yok	usta darboğaz	kilit tasarımına bağlı, en iyi
Karmaşıklık	en düşük	düşük	en yüksek
Günümüz	sanal makineler, bölümleme	gömülü AMP, eski sistemler	Linux, Windows, macOS, BSD

Tablo 12.3: Çok işlemcili işletim sistemi modellerinin karşılaştırması.



Şekil 12.20: Üç model bir arada: ayrı işletim sistemlerinde bellek dörde bölünür; usta-kölede çağrılar yalnız CPU1'de işlenir; SMP'de tek çekirdeği herkes çalıştırır.

Bu üç model paylaşımlı bellekli çok işlemciler içindir. Çoklu bilgisayar işletim sistemlerinden farkını akılda tutmak gerekir. Çok işlemcide bütün CPU'lar aynı fiziksel belleği görür. Çoklu bilgisayarda ise her düğümün kendi belleği ve kendi işletim sistemi kopyası vardır; düğümler yalnız iletiyle haberleşir.

Sık yapılan hata

Sorunları yanlış modele bağlamak. Tutarsız tampon önbellekleri ve dengesiz yük "her CPU'nun kendi işletim sistemi" modelinin sorunudur. Ustanın darboğaz olması usta-köle modelinin, çekirdek tablolarındaki yarışlar ve kilit sırası hataları ise SMP'nin sorunudur.

12.4 Senkronizasyon

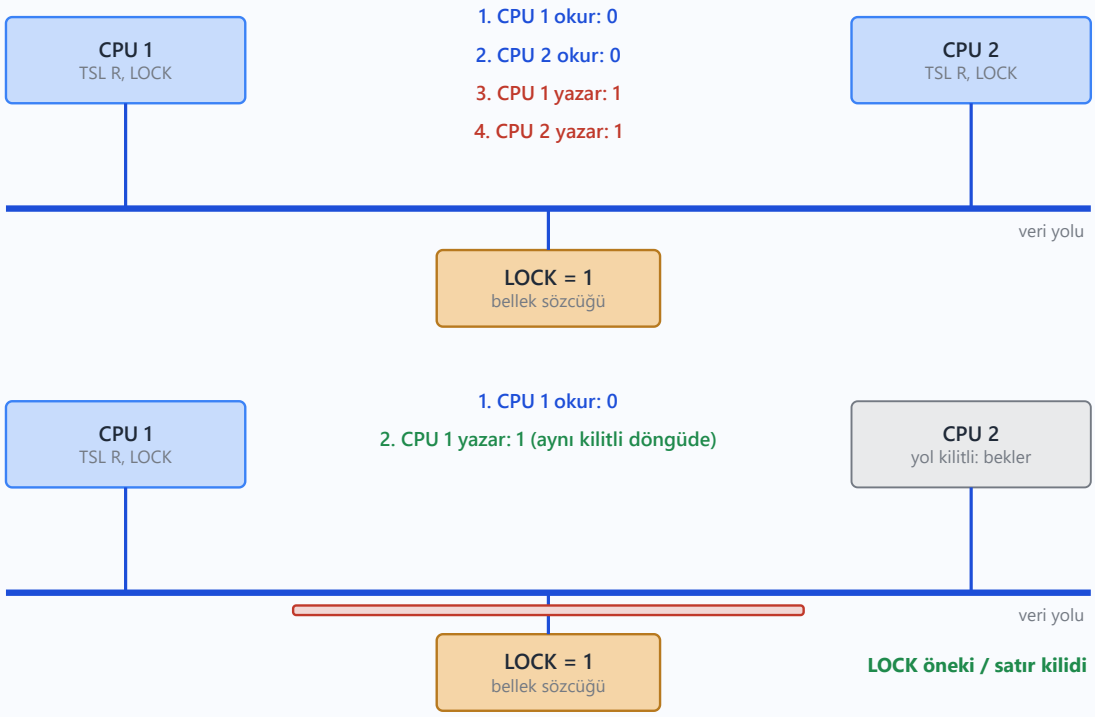
SMP çekirdeği paylaşılan yapılarını kilitlerle korur. Tek işlemcide bir kilit, kesmeleri kapatarak ya da bölünmez bir komutla kolayca gerçeklenebilir. Çok işlemcide ise her iki yol da yeni sorunlar çıkarır. Bu kesim kilidin donanımdaki temelini, daha ölçeklenebilir kilit türlerini ve beklerken dönmek ile uyumak arasındaki seçimi anlatır.

12.4.1 TSL ve veri yolu kilidi

TSL (*test and set lock*) komutu bir bellek sözcüğünü okur, eski değerini bir yazmaca koyar ve sözcüğe 1 yazar. Tek işlemcide bu komut bölünmezdir: komutun ortasında kesme gelmez. Kilit sözcüğü 0 ise serbest, 1 ise tutuluyor demektir. TSL eski değer olarak 0 döndürürse kilit alınmıştır.

Çok işlemcide komutun bölünmezliği tek başına yetmez. TSL'nin okuması ve yazması iki ayrı veri yolu döngüsüdür. Başka bir CPU bu iki döngünün arasına girebilir (Şekil 12.21):

1. CPU 1 kilit sözcüğünü okur: 0, yani serbest.
2. CPU 1 yazmadan önce CPU 2 de okur: yine 0.
3. CPU 1 sözcüğe 1 yazar; CPU 2 de 1 yazar.
4. İkisi de "kilit benim" sanır ve karşılıklı dışlama bozulur.



Şekil 12.21: Üstte veri yolu kilitlenmeyen TSL: iki CPU da 0 okur ve ikisi de kilidi alır. Altta çözüm: TSL okuma ve yazma süresince yolu (ya da önbellek satırını) kilitler; CPU 2 bekler.

Kesmeleri kapatmak da işe yaramaz. Kesmeyi kapatmak yalnız komutu yürüten CPU'yu korur; diğer CPU'lar belleğe erişmeyi sürdürür. Çözüm donanımdadır: TSL, okuma ve yazma süresince veri yolunu kilitler. Bu sırada başka hiçbir CPU belleğe erişemez. Bugünkü x86 işlemcilerinde `LOCK` öneki çoğunlukla bütün yolu değil, yalnız ilgili önbellek satırını kilitler. Satır işlem süresince M durumunda tutulur ve diğer çekirdeklerin istekleri bekletilir. ARM ise aynı işi iki yoldan yapar: `LDXR` / `STXR` komut çifti (aradaki sürede başka biri yazdıysa yazma başarısız olur ve döngü tekrarlanır) ya da LSE uzantısındaki `CAS` gibi tek komutlu atomik işlemler.

12.4.2 Bilet kilidi

Doğru çalışan bir TSL kilidinin iki eksiği kalır. Birincisi adil değildir: kilit serbest kalınca bekleyenlerden hangisinin alacağı rastlantıdır ve biri sürekli kaybedip aç kalabilir. İkincisi her denemede bir yazma yapar ve önbellek trafiği üretir. **Bilet kilidi** (*ticket lock*) birinci sorunu tamamen, ikincisini büyük ölçüde çözer.

Düşünce bir bankadaki sıra numarası makinesidir. İki sayaç vardır: `next` sıradaki bileti dağıtır, `serving` sırası geleni gösterir. Kilidi isteyen iş parçacığı `next`'i atomik olarak artırır ve eski değeri kendi bileti olarak alır. Ardından `serving` kendi biletine eşit olana kadar bekler. Kilidi bırakan `serving`'i bir artırır. Sunumdaki Java programı iki iş parçacığının ortak bir sayacı bu kilitle korumasını gösterir:

```
import java.util.concurrent.atomic.*;
public class Bilet {
  static AtomicInteger next = new AtomicInteger();
  static volatile int serving = 0;
  static int sayac = 0;
  static void lock() {
    int my = next.getAndIncrement();
    while (serving != my) Thread.onSpinWait();
  }
  static void unlock() { serving++; }
  public static void main(String[] a)
      throws Exception {
    Runnable is = () -> {
      for (int i = 0; i < 100_000; i++) {
        lock(); sayac++; unlock(); } };
    Thread t0 = new Thread(is);
    Thread t1 = new Thread(is);
    t0.start(); t1.start(); t0.join(); t1.join();
    System.out.println("sayac = " + sayac);
  }
}
```

Kodun önemli satırları şunlardır. `next` bir `AtomicInteger`'dır; `getAndIncrement()` sayacı bölünmez biçimde artırır ve eski değeri döndürür. x86'da bu tek bir `LOCK XADD` komutudur. İki iş parçacığı aynı anda çağırsa bile farklı biletler alır. `serving` `volatile`'dır. Her okuma gerçek değeri görür. `unlock()`'taki yazma da kritik bölgedeki yazmaların herkese görünmesini sağlar. `Thread.onSpinWait()` işlemciye “dönüyorum” ipucu verir; x86'da `PAUSE` komutuna dönüşür ve güç tüketimini azaltır. `unlock()`'taki `serving++` atomik olmak zorunda değildir, çünkü `serving`'e yalnız kilidin sahibi yazar.

Örnek 12.6

T0 ve T1 sırayla `lock()` çağırıyor; T0 kritik bölgeyi bitirince bırakıyor. Sayaçlar ve iş parçacıklarının durumu nasıl değişir?

Adım	Olay	next	serving	T0.my	T1.my	Durum
0	başlangıç	0	0	—	—	kilit serbest
1	T0 bilet alır	1	0	0	—	serving == 0: T0 girer, `sayac++`
2	T1 bilet alır	2	0	0	1	serving ≠ 1: T1 döner
3	T1 döner	2	0	0	1	serving'i kendi önbelleğindeki S kopyadan okur
4	T0 bırakır	2	1	—	1	yazma T1'in kopyasını geçersiz kılar
5	T1 yeniden okur	2	1	—	1	1 == my: T1 girer

Program 2 × 100.000 artırmanın sonunda `sayac = 200000` yazar. Program JDK 21 ile beş kez çalıştırıldı ve her seferinde 200000 yazdı. Kilit çağrıları kaldırılıp yalnız `sayac++` bırakıldığında altı çalıştırmada 118.108 ile 151.847 arasında değerler çıktı: artırmalar birbirinin üzerine yazdı.

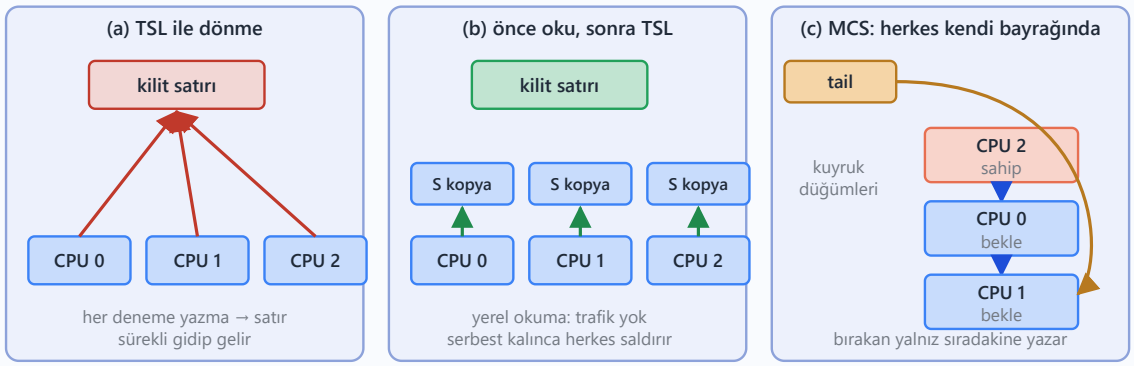


Şekil 12.22: Programın sonunda iki sayaç 200000'e eşittir; dağıtılan her bilet hizmet görmüştür.

Bilet kilidi **adildir**: iş parçacıkları kilide bilet sırasıyla, yani varış sırasıyla girer (FIFO). Hiçbir iş parçacığı aç kalmaz. Bekleyenler yalnız okuma yapar: `serving`'i kendi önbelleklerindeki S kopyadan okurlar ve veri yolunu meşgul etmezler. Kilit bırakılınca tek bir tutarlılık iletisiyle bütün kopyalar geçersiz olur. Yine de bir zayıflığı kalır. Bırakmadan sonra bütün bekleyenler satırı yeniden okur, ama yalnız biri girebilir. Çok sayıda bekleyen varsa bu okuma dalgası pahalıdır.

12.4.3 Spinlock'lar ve önbellek trafiği

Kilidi beklerken döngüde dönen kilitlere **spinlock** denir. Çok işlemcide bir spinlock'un maliyeti, beklerken ürettiği önbellek trafiğidir. Şekil 12.23 üç tasarımı karşılaştırır.



Şekil 12.23: (a) Saf TSL döngüsü: her deneme bir yazmadır, kilit satırı CPU'lar arasında gidip gelir. (b) Önce oku, sonra TSL: bekleyenler kendi S kopyalarını okur. (c) MCS: her bekleyen kendi bayrağında döner.

(a) Saf TSL döngüsü. Bekleyen CPU durmadan TSL yürütür. Her TSL bir yazmadır: satırı M durumunda çeker ve diğer kopyaları geçersiz kılar. Kilit satırı bekleyenler arasında sürekli taşınır. Kilidin sahibi de bu satıra erişmek istediğinde sıraya girer. Sonuçta bekleyenler kilidin sahibini yavaşlatır ve kilidin bırakılması da gecikir.

(b) Önce oku, sonra TSL. Bu yönteme TTAS (*test-and-test-and-set*) denir. Bekleyen CPU önce kilidi sıradan bir okumayla dener. Kilit tutuluyorsa okuma kendi önbelleğindeki kopyadan karşılanır ve yola hiçbir şey çıkmaz. Kilit serbest görününce TSL'yi dener. Ek olarak **üstel geri çekilme** (*exponential backoff*) kullanılabilir: başarısız bir TSL'den sonra CPU bir süre bekler ve bu süreyi her başarısızlıkta ikiye katlar. Böylece bekleyenlerin aynı anda saldırması azalır.

(c) MCS kilidi. Mellor-Crummey ve Scott'ın 1991'de önerdiği kilitte bekleyenler bir bağlı liste oluşturur. Her bekleyen kendi düğümündeki bayrakta döner ve her düğüm ayrı bir önbellek satırındadır. Kilit, listenin sonunu gösteren tek bir `tail` işaretçisiyle yönetilir. Kilidi bırakan, listedeki bir sonrakinin bayrağına yazar. Bu tek bir uzak yazmadır ve yalnız sıradaki CPU'yu uyandırır. MCS hem adildir hem de bekleyen sayısından bağımsız olarak sabit trafik üretir. Linux'un bugünkü `qspinlock`'u bu düşünceye dayanır.

Bu tasarımların hiçbiri veri tutarsızlığını önlemek için yapılmaz; saf TSL de doğru çalışır. Amaç tutarlılık trafiğini, yani satırların önbellekler arasında gereksiz yere taşınmasını azaltmaktır.

12.4.4 Kod izleme: TTAS kilidi

TTAS kilidinin C koduyla gerçeklemesi kısadır. `tsl(k)` çağrısı, `*k`'ya bölünmez biçimde 1 yazan ve eski değeri döndüren donanım komutunu temsil eder.

```
void kilitle(int *k) {
  for (;;) {
    while (*k == 1)   /* yerelde dön */
      ;
    if (tsl(k) == 0)  /* serbest göründü */
      return;         /* kilit alındı */
  }
}
void ac(int *k) {
  *k = 0;
}
```

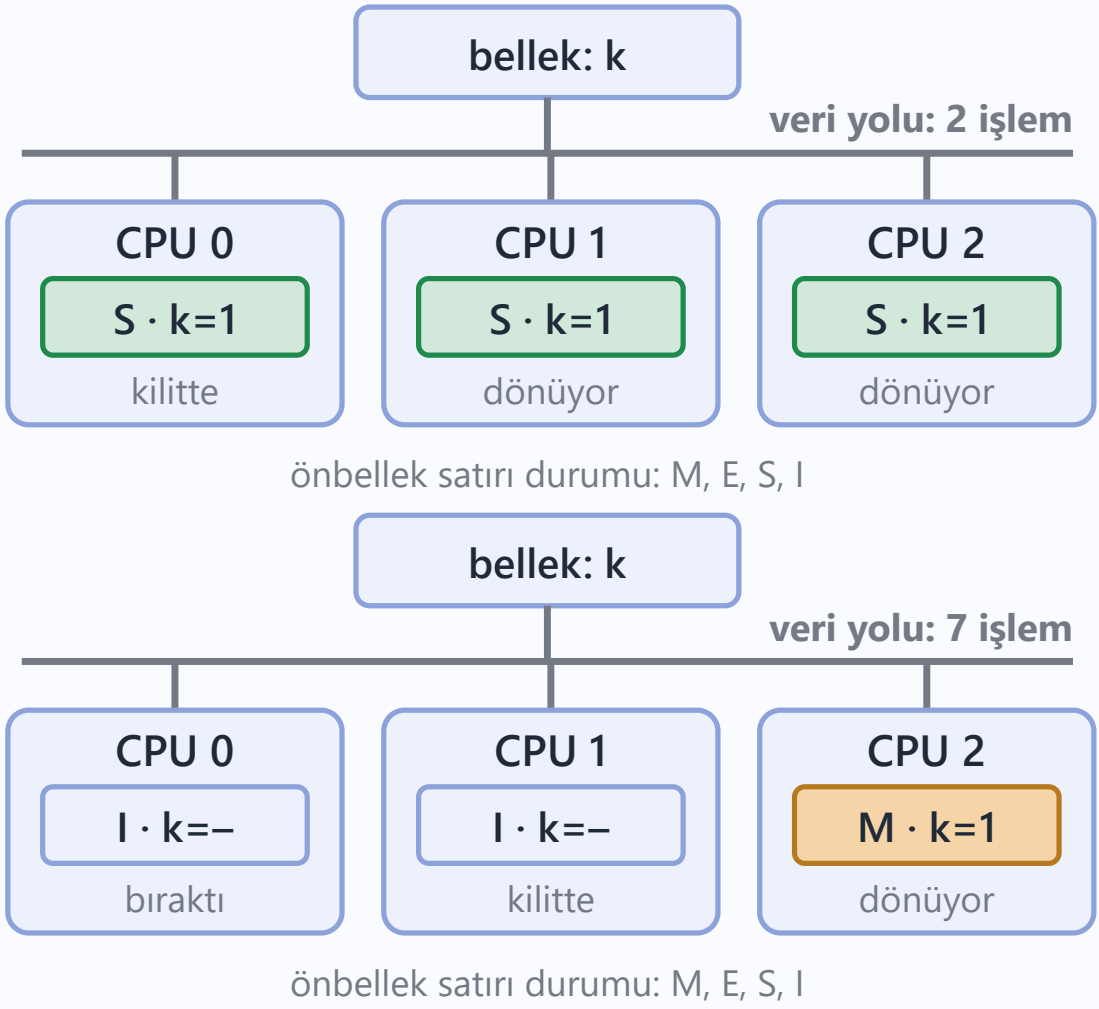
`kilitle()` sonsuz bir döngüdür. İçteki `while` kilit 1 oldukça döner; bu bir okumadır ve kopya önbellekteyse yola çıkmaz. Kilit 0 görününce `tsl()` denenir. Eski değer 0 ise kilit alınmıştır ve işlev döner. Başka bir CPU araya girip kilidi önce aldıysa eski değer 1'dir; döngü yeniden yerel okumaya döner. `ac()` kilit sözcüğüne 0 yazar.

Örnek 12.7

CPU 0 kilidi tutuyor (k = 1, satır CPU 0'ın önbelleğinde M). CPU 1 ve CPU 2 `kilitle()` çağırıyor. CPU 0 kilidi bırakıyor ve iki bekleyen yarışıyor. Kaç veri yolu işlemi oluşur?

Adım	Olay	CPU 0	CPU 1	CPU 2	Yol işlemi (toplam)
0	başlangıç	M, k=1, kilitte	I	I	0
1	CPU 1 `*k` 'yı okur (ıska)	S	S, k=1	I	1
2	CPU 2 okur (ıska)	S	S	S, k=1	2
3	ikisi de yerelde döner	S	S	S	2
4	CPU 0 `ac()` : k = 0	M, k=0	I	I	3
5	CPU 1 okur (ıska), k = 0 görür	S	S	I	4
6	CPU 2 okur (ıska), k = 0 görür	S	S	S	5
7	CPU 1 `tsl` : eski değer 0, kilit alındı	I	M, k=1	I	6
8	CPU 2 `tsl` : eski değer 1, başarısız	I	I	M, k=1	7
9	CPU 2 yerelde döner	I	I	M	7

Toplam yedi veri yolu işlemi oluşur. 3. adımdaki dönme ne kadar uzun sürerse sürsün bu sayı artmaz; saf TSL döngüsünde ise her deneme bir işlem daha üretirdi. (Durumlar, aynı MESI benzetimiyle doğrulandı.)



Şekil 12.24: Üstte 2. adımın sonu: üç kopya S, 2 yol işlemi. Altta izlemenin sonu: CPU 1 kilitte, CPU 2 başarısız TSL'nin çektiği M kopyada dönüyor; toplam 7 işlem.

8. adımda ilginç bir ayrıntı vardır. CPU 2'nin başarısız TSL'si de bir yazmadır. Kilit değişmez (1 üzerine 1 yazılır), ama satır CPU 1'den CPU 2'ye taşınır. CPU 1 kritik bölgeden çıkıp `ac()` çağırdığında satırı yeniden çekmek zorunda kalır. Yarışın bedeli bu fazladan işlemlerdir. Bekleyen sayısı arttıkça her bırakmada bütün bekleyenler okur ve TSL dener; MCS kilidi bu yüzden daha iyi ölçeklenir.

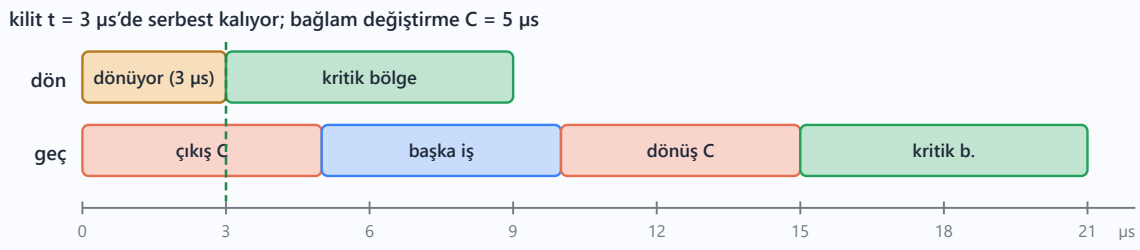
Sık yapılan hata

TTAS'ta içteki okuma döngüsünü yeterli sanıp TSL'yi kaldırmak. Okuma ile yazma arasında başka bir CPU kilidi alabilir; 5. ve 6. adımlarda iki CPU da k = 0 görmüştür. Kilidin doğruluğunu atomik TSL sağlar; içteki okuma yalnız trafiği azaltır.

12.4.5 Dönmek mi, geçiş yapmak mı?

Bir iş parçacığı kilidi alamayınca iki şey yapabilir. **Dönebilir** (*spin*): CPU'yu bırakmadan kilidi yoklamayı sürdürür. CPU bu sürede boşa harcanır, ama kilit serbest kalır kalmaz iş parçacığı devam eder. Ya da **geçiş yapabilir** (*switch*, *block*): uyur ve CPU başka bir iş parçacığına verilir. Bunun bedeli iki bağlam değiştirmedir: biri uyurken, biri uyanırken. Uyanan iş parçacığı soğuk bir önbellekle de karşılaşır.

Şekil 12.25 iki seçeneği bir zaman çizelgesinde karşılaştırır. Kilit t = 3 µs'de serbest kalıyor, bir bağlam değiştirme C = 5 µs sürüyor. Dönen iş parçacığı 3 µs bekleyip kritik bölgeye girer. Geçiş yapan iş parçacığı 5 µs'de çıkar ve CPU başka bir işe geçer. İş parçacığı 5 µs'de geri döner ve kritik bölgeye ancak 15. mikrosaniyede girer.



Şekil 12.25: Kilit 3 µs sonra serbest kalırken dönmek 3 µs, geçiş yapmak 2C = 10 µs harcar.

Karar kuralı buradan çıkar. Beklenen bekleme süresi geçişin bedelinden, yani 2C'den kısaysa dönmek; değilse uyumak ucuzdur. Kısa kritik bölgeler için dönmek, disk G/Ç'si içeren ya da uzun süren bölgeler için uyumak doğrudur.

Sorun, bekleme süresinin önceden bilinmemesidir. Bunun için iyi bilinen bir ödünleşim vardır: önce geçişin bedeli kadar, yani 2C süre dönmek, kilit hâlâ alınamadıysa uyumak. Bu stratejinin maliyeti en kötü durumda, bekleme süresini önceden bilen ideal bir algoritmanın maliyetinin iki katını geçmez. Böyle bir garantisi olan çevrim içi stratejiye **rekabetçi** (*competitive*) strateji denir.

Örnek 12.8

C = 5 µs, yani geçişin bedeli 2C = 10 µs. İş parçacığı önce 10 µs döner, sonra uyur. Bekleme t = 3, 10,1 ve 50 µs olduğunda bu stratejinin maliyeti idealinkiyle nasıl karşılaştırılır?

1. **t = 3.** 3 µs dönülür ve kilit alınır; maliyet 3 µs. İdeal maliyet de min(3, 10) = 3 µs'dir; oran 1.
2. **t = 10,1.** 10 µs dönülür, sonra uyunur: 10 + 10 = 20 µs. İdeal olan baştan uyumaktır: 10 µs. Oran 2'dir; en kötü durum budur.
3. **t = 50.** Yine 10 + 10 = 20 µs harcanır; ideal 10 µs'dir. Oran 2'dir.

Eşik C seçilseydi t, C'yi biraz aştığında maliyet C + 2C = 3C, ideal ise yaklaşık C olurdu; oran 3'e çıkardı. (Oranlar, t'nin binlerce değeri için Python'da hesaplanarak doğrulandı.)

Gerçek sistemler bu düşünceyi uyarlamalı kilitlerle uygular. Linux'un `futex` tabanlı mutex'i, kilidin sahibi başka bir CPU'da çalışıyorsa kısa süre döner, sahip uyuyorsa hemen uyur. Windows'ta `CRITICAL_SECTION` için bir dönme sayısı ayarlanabilir. Java sanal makinesi de çekişmesiz kilitleri ucuz bir biçimde tutar ve çekişme arttığında onları işletim sistemi kilidine dönüştürür; buna kilit şişirme denir.

Çekirdeğin içinde bir kural daha geçerlidir: kesmelerin kapalı olduğu bir bölgede uyunamaz, çünkü uyuyan kodu uyandıracak kesme gelemez. Bu tür bölgelerde yalnız spinlock kullanılabilir.

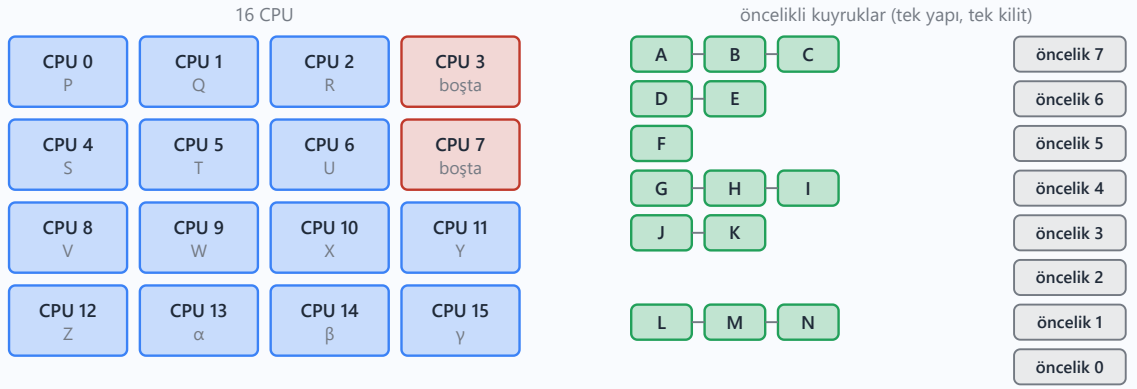
> **Sık yapılan hata**
>
> Tek CPU'lu bir sistemde spinlock'ta dönmek. Kilidin sahibi aynı CPU'da hazır kuyruğunda bekliyorsa, dönen iş parçacığı CPU'yu bırakmadıkça sahip hiç ilerleyemez. Dönme yalnız kilidin sahibi başka bir CPU'da çalışırken anlamlıdır.

12.5 Çok işlemcili çizelgeleme

Tek işlemcide çizelgeleyici yalnız bir soruyu yanıtlar: sıradaki iş parçacığı hangisi? Çok işlemcide ikinci bir soru eklenir: bu iş parçacığı hangi CPU'da çalışacak? İş parçacıkları birbirinden bağımsızsa ikinci soru kolaydır. Aynı uygulamanın iş parçacıkları birbiriyle haberleşiyorsa iki soru birlikte yanıtlanmalıdır. Bu kesim üç yaklaşımı anlatır: zaman paylaşımı, alan paylaşımı ve çete çizelgeleme.

12.5.1 Zaman paylaşımı: ortak hazır kuyruğu

Birbirinden bağımsız süreçler için en basit yaklaşım, bütün CPU'ların tek bir ortak hazır kuyruğundan iş almasıdır (Şekil 12.26). Kuyruk, tek işlemcili sistemlerdeki gibi öncelik düzeylerine göre düzenlenir. Bir CPU boşaldığında kuyruğu kilitler, en yüksek öncelikli iş parçacığını çıkarır, kilidi bırakır ve o iş parçacığını çalıştırır. Şekilde 16 CPU'dan 14'ü çalışıyor; CPU 3 ve CPU 7 boşta. Boştaki CPU'lardan kuyruğu ilk kilitleyen, öncelik 7 kuyruğunun başındaki A'yı alacaktır.



Şekil 12.26: Zaman paylaşımı: 16 CPU tek bir öncelikli hazır kuyruğundan iş alır. Kuyruk tek bir yapıdır ve tek bir kilitle korunur.

Bu yaklaşımın en büyük artısı yükün kendiliğinden dengelenmesidir. Bir CPU iş beklerken başka bir CPU'da iş yığılmaz. Üç eksiği vardır:

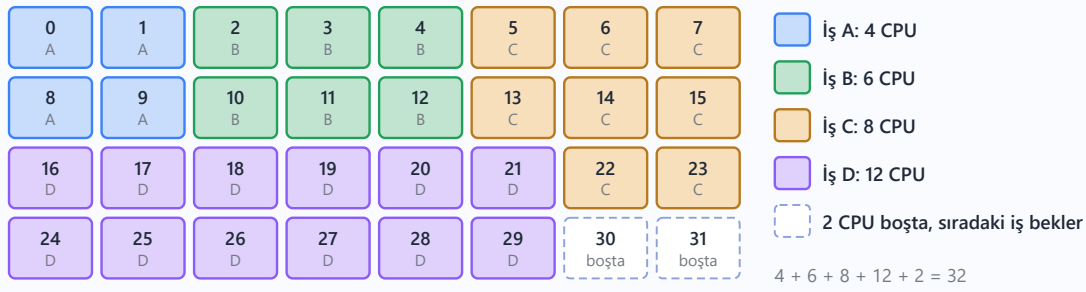
- **Tek kilit darboğazdır.** CPU sayısı arttıkça kuyruğun kilidi için çekişme de artar. Yüzlerce CPU'lu bir sistemde CPU'lar sıradaki işi almak için birbirini bekler.
- **Önbellek soğuk kalır.** Bir iş parçacığı bir süre CPU 4'te çalıştıysa önbelleği ve TLB'si onun verisiyle doludur. Kuyruktan yeniden çıktığında boşta olan herhangi bir CPU'ya, örneğin CPU 12'ye düşebilir. Orada önbellek boştur ve iş parçacığı ilk milisaniyelerini ıskalarla geçirir. Bunu önlemek için çizelgeleyici iş parçacığını mümkünse son çalıştığı CPU'da çalıştırmaya çalışır. Bu eğilime **işlemci afinitesi** (*processor affinity*) denir.
- **Kilit tutarken kesilme.** Bir iş parçacığı bir spinlock tutarken zaman dilimi biterse kesilir. Kilidi isteyen diğer CPU'lar, sahip yeniden çalışana kadar boşuna döner.

Son sorun için Tanenbaum'un anlattığı **akıllı çizelgeleme** (*smart scheduling*) düşüncesi kullanılabilir. Spinlock alan iş parçacığı süreç genelinde görülebilen bir bayrak koyar ve kilidi bırakınca bayrağı kaldırır. Çizelgeleyici bayraklı bir iş parçacığını kesmez. Ona kritik bölgesini bitirmesi için biraz ek süre verir. Benzer bir düzenek bugün sanal makinelerde de kullanılır: kilit tutan bir sanal CPU'nun kesilip diğer sanal CPU'ların boşuna dönmesi sorununa “lock holder preemption” denir.

Bugünkü işletim sistemleri tek ortak kuyruk kullanmaz. Linux ve Windows her CPU için ayrı bir hazır kuyruğu tutar. Kilit çekişmesi kalkar ve iş parçacığı kendiliğinden aynı CPU'da kalır. Kuyruklar arasındaki dengesizlik ise ayrı bir yük dengeleme düzeneğiyle düzeltilir. Yükü az olan CPU başka bir CPU'nun kuyruğundan iş çeker (*pull*), ya da yük dengeleyici ara sıra işleri yüklü CPU'lardan boş olanlara iter (*push*). Linux bu dengelemeyi “çizelgeleme alanları” (*scheduling domains*) adı verilen bir hiyerarşiyle yapar. Önce aynı fiziksel çekirdeğin SMT kardeşleri arasında dengeler, sonra aynı L3'ü paylaşan çekirdekler arasında, en son da NUMA düğümleri arasında. Bağlantıdaki çok çekirdekli çizelgeleme simülatörü bu davranışları denemeye olanak verir.

12.5.2 Alan paylaşımı

Birbiriyle sıkı haberleşen iş parçacıkları için başka bir yaklaşım daha uygundur: aynı uygulamanın bütün iş parçacıklarını aynı anda, ayrı CPU'larda çalıştırmak. Buna **alan paylaşımı** (*space sharing*) denir. Bir iş başladığında istediği sayıda CPU ona ayrılır. İş bitene kadar bu CPU'ları tutar. CPU başına çok programlama yoktur, dolayısıyla bağlam değiştirme de yoktur. Yetecek kadar boş CPU yoksa iş, yeterli CPU boşalana kadar bekler. En basit sıralama ilk gelen ilk hizmet alır (FCFS) biçimindedir.

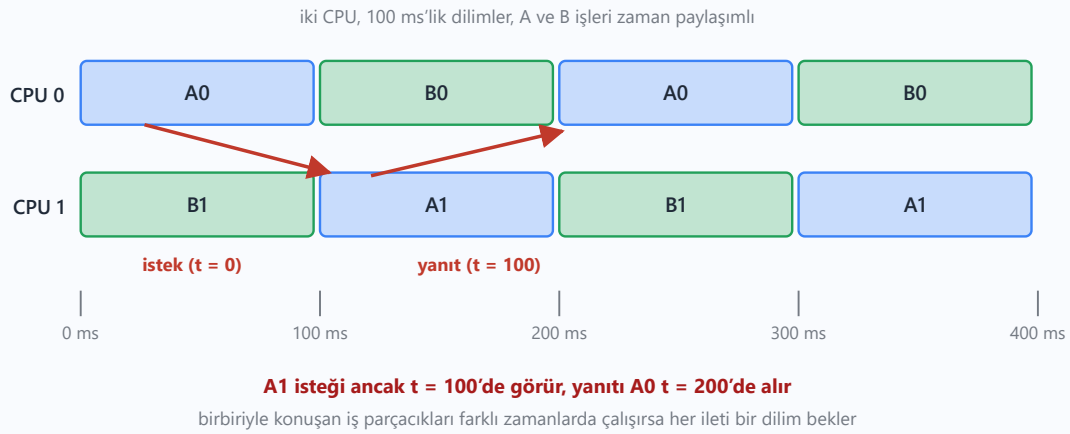


Şekil 12.27: 32 CPU'luk bir sistem dört işe bölünmüş: A 4, B 6, C 8, D 12 CPU kullanıyor; 2 CPU boşta. Sıradaki iş 2 CPU'dan fazlasını isterse bekler.

Şekil 12.27 örneğinde 32 CPU dört işe dağıtılmıştır: 4 + 6 + 8 + 12 = 30 CPU kullanılıyor, 2'si boşta. Kuyruktaki bir sonraki iş 3 CPU isterse, bu işlerden biri bitene kadar bekler. Bu sırada iki CPU boşta kalır. Alan paylaşımının asıl sakıncası budur. Bir iş parçacığı G/Ç için bloke olduğunda onun CPU'su başka bir işe verilmez ve boşa durur. Bugün HPC kümelerindeki iş çizelgeleyicileri (Slurm, PBS) düğümleri tam olarak böyle bölümler: bir iş istediği düğümleri alır ve bitene kadar tutar.

12.5.3 Çete çizelgeleme

Zaman paylaşımında haberleşen iş parçacıklarının neden yavaşladığını bir örnek gösterir (Şekil 12.28). İki CPU var, zaman dilimi 100 ms ve iki iş zaman paylaşımıyla çalışıyor. A işinin iki iş parçacığı A0 ve A1 birbirine iletiler gönderiyor. CPU 0'da sıra A0, B0, A0, B0...; CPU 1'de B1, A1, B1, A1... biçimindedir. A0, t = 0'da A1'e bir istek gönderir. A1 o sırada çalışmıyordur ve isteği ancak t = 100'de görür. Yanıtı hemen gönderir, ama bu kez A0 çalışmıyordur. A0 yanıtı t = 200'de alır. Tek bir istek-yanıt 200 ms sürmüştür.



Şekil 12.28: Çete olmadan: A0 ve A1 kaymış dilimlerde çalışır; her ileti bir dilim bekler ve istek-yanıt 200 ms sürer.

Tanım 12.5

Çete çizelgeleme (*gang scheduling*), birbirine bağlı iş parçacıklarından oluşan bir grubun (çetenin) bütün üyelerinin aynı zaman diliminde, farklı CPU'larda birlikte çalıştırılmasıdır. Bütün CPU'lar dilimlere eş zamanlı başlar ve dilimleri eş zamanlı bitirir.

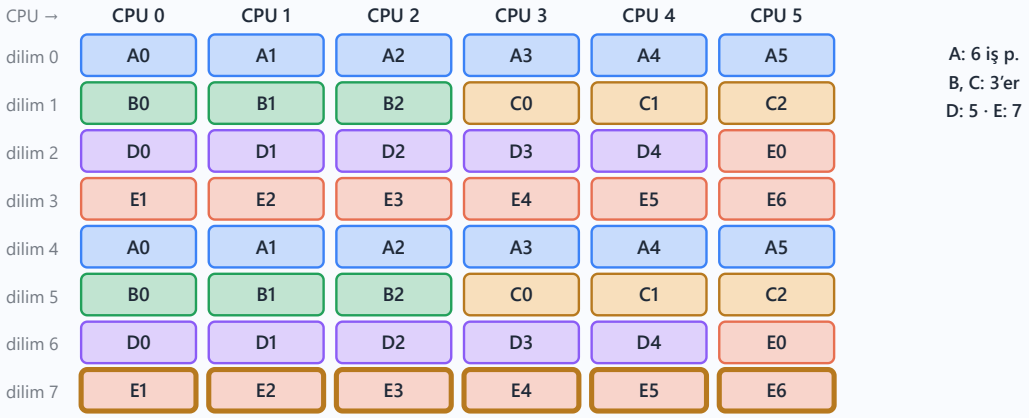
Çete çizelgeleme zaman paylaşımı ile alan paylaşımını birleştirir. Üç parçası vardır. Birbirine bağlı iş parçacıkları bir çete olarak gruplanır. Bir çetenin bütün üyeleri aynı anda, farklı CPU'larda çalışır. Bütün çete üyeleri dilimlerine birlikte başlar ve birlikte bitirir. Böylece bir üyenin gönderdiği iletiyi alıcısı hemen görür ve yanıt da aynı dilimde döner.

Örnek 12.9

6 CPU'lu bir sistemde beş iş var: A 6, B 3, C 3, D 5, E 7 iş parçacığından oluşuyor. Çete çizelgelemeyle dilimlere nasıl yerleşir?

1. **Dilim 0.** A'nın 6 iş parçacığı 6 CPU'yu doldurur.
2. **Dilim 1.** B (3) ve C (3) birlikte 6 CPU'yu doldurur.
3. **Dilim 2.** D'nin 5 iş parçacığı ve E0. E'nin 7 iş parçacığı tek dilime sığmaz; ilk üyesi burada çalışır.
4. **Dilim 3.** E'nin kalan 6 iş parçacığı (E1–E6).
5. **Döngü.** Dört dilimden sonra sıra baştan tekrarlanır.

Toplam 6 + 3 + 3 + 5 + 7 = 24 iş parçacığı 4 dilim × 6 CPU = 24 yuvaya tam sığar; kullanım %100'dür. Tanenbaum'un örneğinde E büyük bir çete olduğu için iki dilime bölünmüştür. Daha katı bir tanımda bir çetenin hiçbir üyesi ötekilerden ayrı çalışmaz; o zaman E bir dilime sığmadığı için bu düzen kurulamazdı.



Şekil 12.29: Çete çizelgeleme: 6 CPU ve 8 dilim. A, B+C, D+E0 ve E1–E6 dilimleri dört dilimlik bir döngüde tekrarlanır.

Çete çizelgelemenin iki koşulu kısaca şöyle özetlenir: çete üyeleri aynı anda başlar ve hepsi farklı CPU'lardadır. Bu düşünce bugün sanal makinelerde de karşımıza çıkar. Çok sanal CPU'lu bir sanal makinenin vCPU'ları birbirini kilitlerle bekler. Hipervizör bunları birlikte çizelgelerse (*co-scheduling*) bir vCPU'nun, kesilmiş bir kardeşini beklerken boşuna dönmesi önlenir.

Sık yapılan hata

Çete çizelgelemede yalnız "aynı anda" koşulunu akılda tutmak. Üyeler aynı anda ama aynı CPU'da sırayla çalışsaydı yine birbirini beklerlerdi. İki koşul birlikte gereklidir: aynı dilim ve farklı CPU'lar.

12.6 Çoklu bilgisayarlar

Paylaşımlı bellekli çok işlemciler birkaç yüz CPU'nun ötesine büyütülemez; büyütmek de pahalıdır. Daha büyük sistemler belleği paylaşmayan bilgisayarlardan kurulur. Bu kesim çoklu bilgisayarların donanımını, iletilerin ağda nasıl taşındığını ve bu sistemlerde kullanılan programlama ve yük dağıtma yöntemlerini anlatır.

12.6.1 Düğümler ve ağ arayüzleri

Tanım 12.6

Çoklu bilgisayar (*multicomputer*), her biri CPU, RAM ve ağ arayüzünden oluşan düğümlerin hızlı bir ara bağlantıyla birleştirildiği sistemdir. Bellek paylaşılmaz; her düğümde kendi işletim sistemi kopyası çalışır ve düğümler yalnız iletiyle haberleşir.

Çoklu bilgisayarlara küme de denir. Bir düğüm çoğu zaman sıradan bir sunucudur; onu özel yapan ağ arayüzü ve ara bağlantıdır. Büyük ölçekli benzetimler, yüksek başarımlı hesaplama ve yapay zekâ eğitimi bu sistemlerde çalışır. Dünyanın en hızlı bilgisayarlarını sıralayan TOP500 listesindeki sistemler on binlerce düğümlü çoklu bilgisayarlardır.

Çoklu bilgisayarda performansın önündeki en büyük engel iletilerin kopyalanmasıdır (Şekil 12.30). Bir ileti en kötü durumda beş kez kopyalanır:

1. gönderen sürecin kullanıcı tamponundan çekirdek tamponuna,
2. çekirdek tamponundan ağ arayüz kartının (NIC) belleğine,
3. NIC'ten kablo üzerinden karşı NIC'e,
4. karşı NIC'in belleğinden alıcının çekirdek tamponuna,
5. çekirdek tamponundan alıcı sürecin kullanıcı tamponuna.

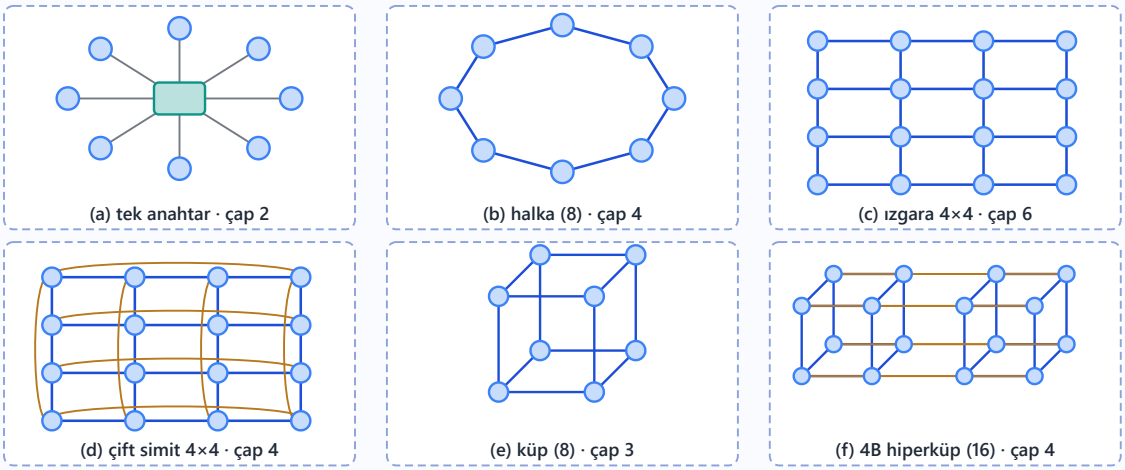


Şekil 12.30: Bir iletinin kopyaları. 3 numaralı kopya kablodur; RDMA'da NIC doğrudan kullanıcı tamponuna yazar ve 1, 2, 4, 5 numaralı kopyalar ortadan kalkar.

Her kopya mikrosaniyeler alır ve ağın kendisi bundan daha hızlı olabilir. Kopyaları azaltmanın yolları şunlardır. NIC belleği doğrudan kullanıcı alanına eşlenebilir; süreç iletiyi oraya kendisi yazar. NIC, DMA ile ana belleği doğrudan okuyup yazabilir. En ileri çözüm **RDMA**'dır (*remote direct memory access*). InfiniBand ve RoCE ağlarında NIC, karşı makinedeki bir sürecin belleğine çekirdek hiç devreye girmeden yazar. Bir ileti böylece birkaç mikrosaniye yerine bir mikrosaniyenin altında karşıya ulaşır.

12.6.2 Ara bağlantı topolojileri

Düğümlerin birbirine nasıl bağlandığına **topoloji** denir. Bir topolojiyi değerlendiren iki temel ölçü vardır. **Çap**, en uzak iki düğüm arasındaki en kısa yolun atlama sayısıdır; en kötü durumdaki gecikmeyi belirler. **Bağlantı sayısı** ise maliyeti belirler. Şekil 12.31 altı yaygın topolojiyi gösterir.



Şekil 12.31: Ara bağlantı topolojileri ve çapları: (a) tek anahtar, (b) halka, (c) ızgara, (d) çift simit, (e) küp, (f) dört boyutlu hiperküp.

- **Tek anahtar** (yıldız): her düğüm merkezdeki anahtara bağlıdır. Çap 2'dir, ama anahtar tek hata noktasıdır ve büyütülemez.
- **Halka**: her düğümün iki komşusu vardır. n düğümlü çift yönlü halkada çap n / 2'dir.
- **Izgara**: düğümler bir kare düzende yerleşir. k × k ızgarada en uzak iki köşe arasında 2(k − 1) atlama vardır.
- **Simit** (*torus*): ızgaranın kenarları karşı kenara sarılır. k × k simitte çap 2 · ⌊k / 2⌋ olur.
- **Hiperküp**: d boyutlu hiperküpte 2^d düğüm vardır ve her düğüm d komşuya bağlıdır. Çap d = $\log_2$n'dir.

Örnek 12.10

1.024 düğümlü bir sistem için halka, 32 × 32 ızgara, 32 × 32 simit ve 10 boyutlu hiperküpün çapı ve bağlantı sayısı nedir?

1. **Halka.** Çap 1024 / 2 = 512, bağlantı 1024.
2. **Izgara.** Çap (32 − 1) + (32 − 1) = 62. Her satırda 31 yatay, her sütunda 31 dikey bağlantı vardır: 2 × 32 × 31 = 1984.
3. **Simit.** Sarma bağlantılarıyla her boyutta en fazla 16 atlama gerekir: çap 32. Her düğümden 4 bağlantı çıkar ve her bağlantı iki düğümce paylaşılır: 1024 × 4 / 2 = 2048.
4. **Hiperküp.** Çap $\log_2 1024 = 10$. Her düğümün 10 bağlantısı vardır: 1024 × 10 / 2 = 5120.

> Değerler Tablo 12.4'te toplanmıştır. 16 düğümlü küçük topolojilerin çapları ayrıca bir genişlik öncelikli arama ile hesaplanıp şekildeki değerlerle karşılaştırıldı.

n = 1.024	Halka	Izgara 32 × 32	Simit 32 × 32	Hiperküp (10B)
Çap	512	62	32	10
Bağlantı sayısı	1.024	1.984	2.048	5.120

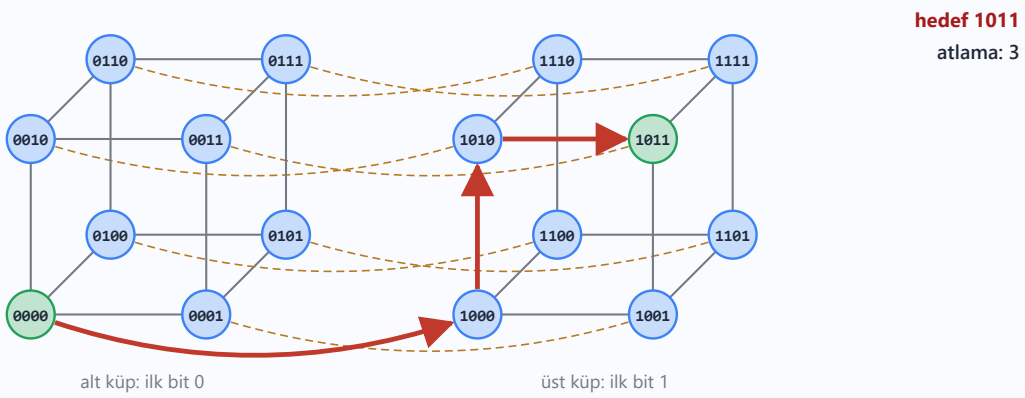
Tablo 12.4: 1.024 düğümde dört topolojinin çapı ve bağlantı sayısı.

Tablo bir ödünleşimi gösterir. Hiperküpün çapı en küçüktür, ama bağlantı sayısı da en büyüktür. Üstelik hiperküpte düğüm başına bağlantı sayısı boyutla birlikte artar; düğüm sayısı iki katına çıkınca her düğüme bir bağlantı daha eklenmelidir. Bugünkü süper bilgisayarlar çoğunlukla *fat-tree* ve *dragonfly* topolojilerini kullanır. Japonya'nın Fugaku süper bilgisayarı ise altı boyutlu bir simit ağı (Tofu) kullanır.

12.6.3 Hiperküpte yönlendirme

Hiperküpte düğümler d bitlik numaralar taşır ve iki düğüm yalnız tek bir bitte farklıysa komşudur. 4 boyutlu hiperküpte 16 düğüm vardır; her düğümün 4 komşusu bulunur. Örneğin 0000'ın komşuları 1000, 0100, 0010 ve 0001'dir. Bu yapı yönlendirmeyi çok kolaylaştırır. İki düğüm arasındaki en kısa yolun uzunluğu, numaralarının kaç bitte farklı olduğuna, yani **Hamming uzaklığına** eşittir. İleti her adımda farklı bitlerden birini düzelterek ilerler.

E-küp yönlendirmesi (*e-cube routing*) bu bitleri her zaman aynı sırada, soldan sağa düzeltir. 0000'dan 1011'e giden bir ileti için kaynak $\oplus$ hedef = 1011'dir; üç bit farklıdır. Soldaki ilk farklı bit düzeltilir: 1000. Bu adım iletiyi alt 3 boyutlu küpten üst küpe geçirir. Sonra 1010'a, en son 1011'e gelinir (Şekil 12.32). Atlama sayısı 3'tür; 4 boyutlu hiperküpte en kötü durum $\log_2 16 = 4$ atlamadır.



Şekil 12.32: 4 boyutlu hiperküpte 0000 → 1000 → 1010 → 1011 yolu. Sol küpte ilk bit 0, sağ küpte 1'dir; kesikli kenarlar iki küpü bağlar.

Kuralın önemli bir özelliği yerel olmasıdır. Her düğüm yalnız kendi numarasına ve hedefin numarasına bakarak iletiyi hangi komşusuna göndereceğine karar verir. Yönlendirme tablosu gerekmez. Bitler hep aynı sırada düzeltildiği için yol döngüye de girmez.

12.6.4 Kod izleme: e-küp yönlendirme

E-küp yönlendirmesi C'de birkaç satırdır. `src` kaynak, `dst` hedef düğüm, `d` hiperküpün boyutudur; işlev geçilen her düğümü yazdırır.

```
void yonlendir(int src, int dst, int d) {
  int u = src;
  for (int b = d - 1; b >= 0; b--) {
    int m = 1 << b;
    if ((u ^ dst) & m) {  /* bit farklı */
      u ^= m;             /* komşuya geç */
      printf("%d ", u);
    }
  }
}
```

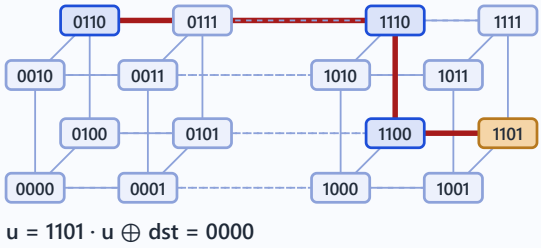
`u` iletinin o anda bulunduğu düğümdür. Döngü en soldaki bitten (`b = d - 1`) başlayıp sağa doğru ilerler. `m = 1 << b` , yalnız b. biti 1 olan maskedir. `(u ^ dst) & m` sıfırdan farklıysa u ile hedef bu bitte farklıdır. `u ^= m` o biti çevirir; bu, iletinin o boyuttaki komşuya geçmesi demektir. Döngü d kez döner; atlama sayısı farklı bit sayısı kadardır.

Örnek 12.11

`yonlendir(6, 13, 4)` çağrısı, yani 0110'dan 1101'e giden ileti adım adım nasıl ilerler?

b	m	u (önce)	u ^ dst	Bit farklı mı?	u (sonra)	Çıktı
—	—	0110	1011	—	0110	—
3	1000	0110	1011	evet	1110	14
2	0100	1110	0011	hayır	1110	—
1	0010	1110	0011	evet	1100	12
0	0001	1100	0001	evet	1101	13

Çıktı `14 12 13` 'tür. İleti 0110 → 1110 → 1100 → 1101 yolunu 3 atlamada izler. Başlangıçtaki u ⊕ dst = 1011 üç bitin farklı olduğunu baştan söylüyordu. (Döngü Python'a birebir aktarılıp çalıştırıldı.)



Şekil 12.33: İzlemenin sonu: u = 1101 ve u ⊕ dst = 0000; ileti hedefe vardı.

Sık yapılan hata

Atlama sayısını düğüm numaralarının farkından hesaplamak. 6 ile 13 arasındaki fark 7'dir, ama atlama sayısı farklı bit sayısıdır: 0110 ⊕ 1101 = 1011, yani 3. Komşuluk sayısal yakınlık değil, tek bit farkıdır.

12.6.5 Sakla ve ilet ya da kesip geç

Bir paket kaynaktan hedefe birkaç anahtardan geçerek gider. Anahtarların paketi nasıl ilettiği, toplam gecikmeyi büyük ölçüde değiştirir. Örnek olarak 4 KiB'lık bir paket ve 10 Gb/s hızında bağlantılar düşünelim. Paketin bir bağlantıdan geçmesi 4096×8 bit / 10^{10} bit/s ≈ 3,28 µs sürer.

Sakla ve ilet (*store-and-forward*) yönteminde her anahtar paketin tamamını tamponuna alır, denetler ve ancak ondan sonra bir sonraki bağlantıya gönderir. A'dan iki anahtar üzerinden B'ye giden paket üç bağlantı geçer. Her bağlantıda paketin tamamı beklenir: 3 × 3,28 = 9,83 µs. Gecikme atlama sayısıyla doğrusal büyür.

Kesip geç (*cut-through*) yönteminde anahtar paketin başlığını okur okumaz yolu kurar ve paketin geri kalanını beklemeden iletmeye başlar. Paket boru gibi akar; ön kısmı hedefe varırken arka kısmı hâlâ kaynaktan çıkar. **Solucan deliği** (*wormhole*) yönlendirmesi bu düşüncenin küçük parçalara (*flit*) bölünmüş paketlerle kullanılan biçimidir. Toplam gecikme yaklaşık bir paket süresi artı anahtarların küçük başlık gecikmeleridir. Her anahtar başlığı 0,05 µs'de işlerse 3,28 + 2 × 0,05 ≈ 3,38 µs olur (Şekil 12.34).





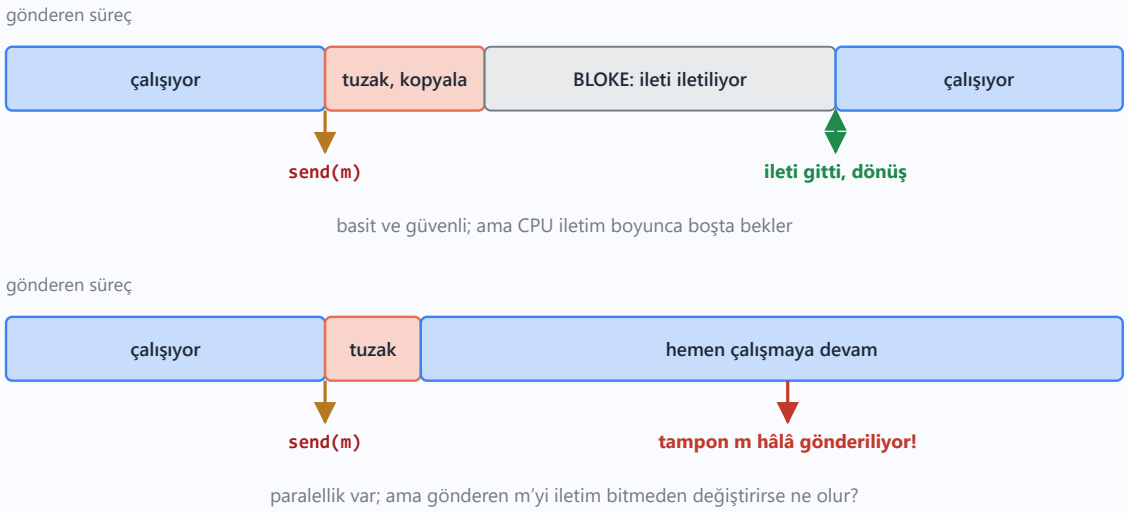
Şekil 12.34: Üstte sakla ve ilet: paket her anahtarda tamamen alınır, toplam 9,83 µs. Altta kesip geç: paketin parçaları aynı anda birden çok bağlantıdadır, toplam ≈ 3,38 µs.

Atlama sayısı arttıkça fark büyür. Sakla ve iletin gecikmesi atlama sayısıyla orantılıdır; kesip geçte ise her atlama yalnız küçük bir başlık gecikmesi ekler. Yüksek başarımlı hesaplama ağları ve veri merkezi anahtarları bu yüzden kesip geç ya da solucan deliği yöntemini kullanır.

12.6.6 Bloke eden ve etmeyen gönderim

Çoklu bilgisayarda süreçler `send` ve `receive` çağrılarıyla haberleşir. Bu çağrıların en önemli tasarım sorusu, gönderenin iletinin gitmesini bekleyip beklemediğidir.

Bloke eden (*blocking*) ya da senkron gönderimde `send` çağrısı ileti iletilene kadar dönmez (Şekil 12.35 (a)). Süreç tuzakla çekirdeğe girer, ileti kopyalanır ve gönderilir; ancak ondan sonra süreç devam eder. Bu yöntem basit ve güvenlidir: `send` döndüğünde tampon yeniden kullanılabilir. Bedeli, CPU'nun iletim boyunca boşta beklemesidir.



Şekil 12.35: (a) Bloke eden gönderim: süreç ileti gidene kadar bekler. (b) Bloke etmeyen gönderim: süreç hemen devam eder, ama tampon m'nin gönderimi sürer.

Bloke etmeyen (*nonblocking*) ya da eş zamansız gönderimde `send` hemen döner ve süreç çalışmayı sürdürür (b). Hesaplama ile iletişim paralel yürür. Ama yeni bir sorun doğar: gönderen, iletim bitmeden tampondaki `m`'yi değiştirirse karşıya hangi veri gider? Tanenbaum bu soruna dört yanıt sıralar (Tablo 12.5).

Seçenek	Bedel
Bloke eden gönderim	CPU iletim boyunca boşta
Bloke etmeyen + çekirdeğe kopyalama	fazladan kopya için CPU zamanı
Bloke etmeyen + bitince kesme	programlaması zor (yarış durumları)
Yazınca kopyala (CoW)	tampon yazılırsa yine fazladan kopya

Tablo 12.5: Gönderim seçenekleri ve bedelleri.

İkinci seçenekte çekirdek iletiyi kendi tamponuna kopyalar ve süreç hemen devam eder; tampon artık serbesttir. Üçüncüsünde iletim bitince gönderene bir kesme ya da sinyal gelir; süreç o ana kadar tampona dokunmamalıdır. Dördüncüsünde tamponun sayfası salt okunur işaretlenir; süreç ancak tampona yazmaya kalkarsa sayfanın kopyası çıkarılır.

Alıcı tarafta da benzer seçenekler vardır. Bloke eden `receive` ileti gelene kadar bekler. Yoklama (*polling*) ile süreç ara sıra ileti gelip gelmediğine bakar. Üçüncü yol, ileti gelince yeni bir iş parçacığı açmak ya da iletinin içindeki kodu doğrudan çalıştırmaktır (*active message*). Bu kavramların bugünkü karşılıkları da tanıdıktır. MPI kütüphanesinde `MPI_Send` / `MPI_Recv` bloke eden, `MPI_Isend` / `MPI_Irecv` ile `MPI_Wait` bloke etmeyen çağrılardır. Linux'ta `io_uring` ve sıfır kopyalı `MSG_ZEROCOPY` aynı soruna verilmiş modern yanıtlardır.

12.6.7 Uzak prosedür çağrısı

`send` ve `receive` bir G/Ç modelidir; programcıyı iletileri kendisi oluşturmaya zorlar. Birrell ve Nelson 1984'te daha tanıdık bir model önerdi: başka bir makinedeki bir prosedürü yerel bir prosedür gibi çağırmak.

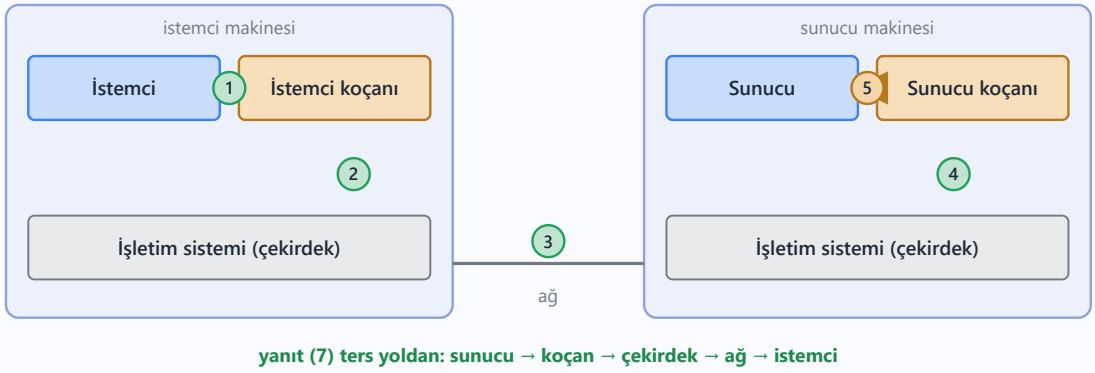
> **Tanım 12.7**
>
> **Uzak prosedür çağrısı** (*remote procedure call*, RPC), bir makinedeki programın başka bir makinedeki bir prosedürü sıradan bir yerel çağrı sözdizimiyle çağırmasıdır. İletilerin oluşturulması ve gönderilmesi programcıdan gizlenir.

Gizlemeyi **koçan** (*stub*) adı verilen küçük prosedürler yapar. İstemci `sonuc = topla(3, 4)` yazdığında aslında istemci koçanını çağırır. Bir çağrı şu adımlardan geçer (Şekil 12.36):

1. İstemci, istemci koçanını yerel bir prosedür gibi çağırır. Parametreler her zamanki gibi yığına konur.

2. Koçan parametreleri bir iletiye paketler ve çekirdeğe sistem çağrısı yapar. Paketlemeye **sıralama** (*marshalling*) denir.
3. Çekirdek iletiyi ağ üzerinden sunucu makinesine gönderir.
4. Sunucunun çekirdeği iletiyi sunucu koçanına verir.
5. Sunucu koçanı parametreleri açar ve sunucu prosedürünü yerel olarak çağırır.

Yanıt aynı yolu ters yönde izler: sunucu → koçan → çekirdek → ağ → istemci çekirdeği → istemci koçanı → istemci. İstemci açısından `topla` sıradan bir prosedürdür; yalnız biraz yavaştır.



Şekil 12.36: RPC'nin adımları: istemci, istemci koçanı, çekirdek, ağ, sunucu çekirdeği, sunucu koçanı ve sunucu. Yanıt ters yoldan döner.

Görünüşteki bu sadelik birkaç zorluğu saklar:

- **İşaretçiler.** İstemcideki bir adres sunucunun adres alanında anlamsızdır. Bir dizi parametresi değer olarak kopyalanıp gönderilir, sunucunun yaptığı değişiklikler de dönüşte geri kopyalanır. Bu yönteme **kopyala-geri yükle** (*copy-restore*) denir. Bağlı liste gibi işaretçilerle kurulu yapılar ise bu yolla kolayca gönderilemez.
- **Uzunluğu bilinmeyen diziler.** C'de bir dizinin uzunluğu dizinin kendisinde yazmaz. Koçan kaç bayt göndereceğini bilemez; uzunluk ayrı bir parametre olarak verilmelidir.
- **Küresel değişkenler.** Sunucu, istemcinin küresel değişkenlerini göremez.
- **Kısmi hatalar.** Yerel bir çağrıda çağıran ve çağrılan birlikte çöker. RPC'de sunucu ya da ağ tek başına çökebilir. İstemci yanıt alamayınca çağrıyı yinelerse prosedür iki kez çalışabilir. Bu yüzden bir RPC sisteminin “en az bir kez” mi, “en çok bir kez” mi anlamını sağladığı belirtilmelidir.
- **Veri gösterimi.** İki makinenin bayt sırası ya da tam sayı boyutu farklı olabilir; koçan veriyi ortak bir biçime dönüştürür.

Koçanlar elle yazılmaz; bir arayüz tanımından otomatik üretilir. Bugün en yaygın örnekler gRPC ve onun arayüz dili Protocol Buffers, Apache Thrift ve Java RMI'dır.

Sık yapılan hata

Koçanı bir ağ sürücüsü ya da uzak kodu yerelde çalıştıran bir düzenek sanmak. Koçan yalnız parametreleri iletiye paketleyen, çekirdeğe gönderen ve yanıtı açan sıradan bir prosedürdür. Uzak prosedür sunucuda çalışır.

12.6.8 Dağıtık paylaşımlı bellek

RPC ileti geçirmeyi prosedür çağrısının arkasına gizler. Daha iddialı bir düşünce, belleği paylaşmayan makinelerde paylaşımlı bellek yanılsaması yaratmaktır. Bu düşünceye **dağıtık paylaşımlı bellek** (*distributed shared memory*, DSM) denir. Yanılsamayı sağlayan katman donanımda (NUMA), işletim sisteminde ya da bir kullanıcı kütüphanesinde olabilir. Li ve Hudak'ın 1989'da tanımladığı sayfa tabanlı DSM, işletim sisteminin sanal bellek düzeneğini kullanır.

Bütün makineler aynı sanal adres alanını paylaşır, ama her sayfa fiziksel olarak tek bir makinenin belleğindedir (Şekil 12.37). Bir CPU, belleğinde olmayan bir sayfaya erişince MMU sayfa hatası üretir. İşletim sistemi, örneğin bir dizin aracılığıyla, sayfanın hangi makinede olduğunu bulur ve onu ister. Sayfa ağ üzerinden bu CPU'ya taşınır, sayfa tablosu güncellenir ve hataya yol açan komut yeniden yürütülür. Süreç açısından bu, diskten gelen sıradan bir sayfa hatasından farksızdır; yalnız sayfa diskten değil başka bir makineden gelir.



Şekil 12.37: 16 sayfa dört CPU'ya dağıtılmış. Sayfa 10 salt okunur olduğu için çoğaltılmıştır: CPU 0 ve CPU 1'de birer kopyası vardır.

Sık okunan sayfaları sürekli taşımak pahalıdır. Salt okunur bir sayfa bu yüzden **çoğaltılır** (*replication*): şekildeki sayfa 10'un hem CPU 0'da hem CPU 1'de kopyası vardır ve iki CPU da yerel okur. Bir CPU çoğaltılmış bir sayfaya yazmak isterse diğer kopyalar geçersiz kılınır. Bu, MESI'nin sayfa boyutundaki karşılığıdır. Aynı nedenle yanlış paylaşım da sayfa boyutunda ortaya çıkar. Farklı makinelerde kullanılan bağımsız iki değişken aynı sayfadaysa sayfa iki makine arasında gidip gelir.

Sayfa tabanlı DSM bugün yaygın değildir. 4 KiB'lık bir sayfayı ağ üzerinden taşımak pahalıdır ve sayfa paylaşım için kaba bir birimdir. Yüksek başarımlı hesaplama ileti geçirmeye (MPI) ve PGAS dillerine (bölümlenmiş küresel adres alanı) yöneldi. Düşüncenin bugünkü karşılığı donanımdadır: CXL stan-

dardı birden çok makinenin ortak bir bellek havuzunu kullanmasına olanak verir.

12.6.9 Çizge kuramsal deterministik atama

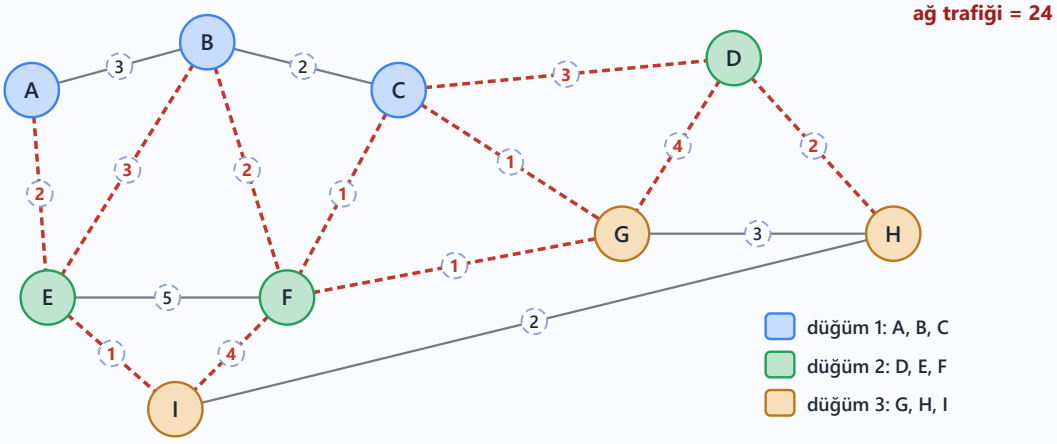
Çoklu bilgisayarda süreçlerin hangi düğümde çalışacağı da önemlidir. Çok haberleşen iki süreç ayrı düğümlere konursa her iletileri ağdan geçer. Aynı düğüme konursa iletileri yerel kalır. Süreçler arasındaki trafik önceden biliniyorsa atama bir çizge problemi olarak çözülebilir. Süreçler düğümler, aralarındaki trafik kenar ağırlıklarıdır. Amaç, düğümleri eşit büyüklükte kümelere bölerken kümeler arasında kalan kenarların toplam ağırlığını en aza indirmektir.

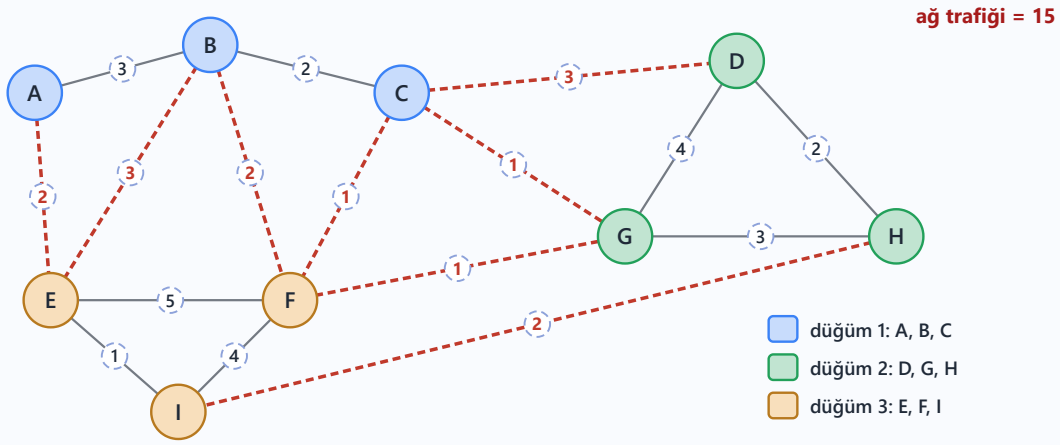
Örnek 12.12

A–I dokuz süreç 3 makineye üçer üçer atanacak. Süreçler arasındaki trafik Şekil 12.38'deki kenar ağırlıklarıdır; toplam trafik 39'dur. Alfabetik atama ile en iyi atamanın ağ trafiği ne kadardır?

1. **Alfabetik atama.** {A, B, C}, {D, E, F}, {G, H, I}. Makine içinde yalnız A–B (3), B–C (2), E–F (5), G–H (3) ve H–I (2) kenarları kalır: toplam 15. Ağ trafiği 39 − 15 = 24 olur.
2. **Daha iyi atama.** {A, B, C}, {D, G, H}, {E, F, I}. Makine içinde A–B (3), B–C (2), D–G (4), D–H (2), G–H (3), E–F (5), E–I (1), F–I (4) kalır: toplam 24. Ağ trafiği 39 − 24 = 15'tir.
3. **En iyi mi?** Dokuz süreç 9! / (3!^{3} · 3!) = 280 farklı biçimde üçer kişilik üç gruba ayrılabilir. Hepsi denendiğinde en küçük ağ trafiği 15 çıkar ve bu değeri yalnız yukarıdaki atama verir.

(280 bölümlemenin tamamı Python'da sayılıp kesim ağırlıkları hesaplandı.)





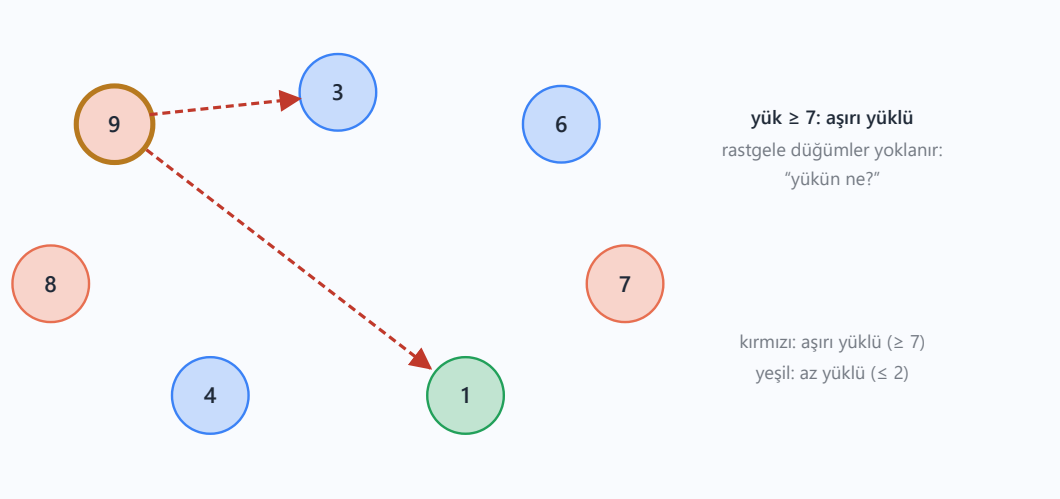
Şekil 12.38: Dokuz süreç ve aralarındaki trafik. Üstte alfabetik atama (ağ trafiği 24), altta en iyi atama (15). Kesikli kırmızı kenarlar makineler arasında kalan trafiktir.

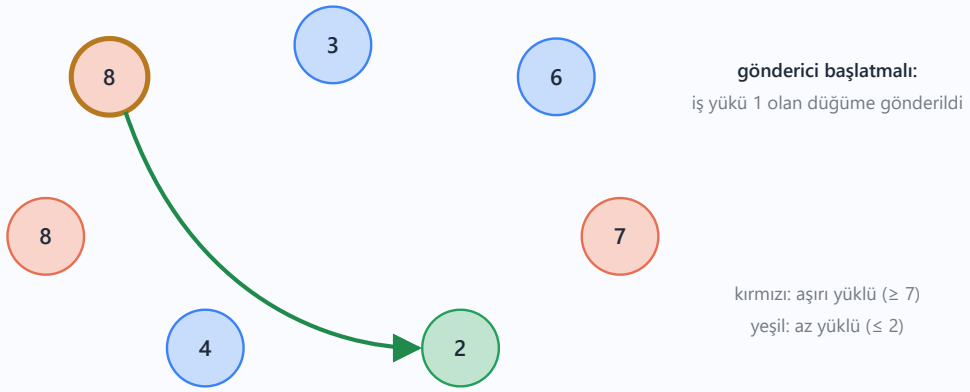
Çok haberleşen süreçler aynı makinede toplanınca ağ trafiği üçte bir azalmıştır. Ama bu yöntemin iki sınırı vardır. Birincisi, trafik önceden bilinmelidir. İkincisi, problem genel hâlinde NP-zordur: süreç sayısı arttıkça olası bölümlemelerin sayısı patlar ve hepsi denenemez. Pratikte sezgisel yöntemler kullanılır.

12.6.10 Dağıtık sezgisel yük dengeleme

Trafik bilinmiyorsa ya da işler sürekli gelip gidiyorsa yük, düğümlerin kendi aralarında yürüttüğü basit kurallarla dağıtılır. İki temel algoritma vardır.

Gönderici başlatmalı (*sender-initiated*) algoritmada yeni bir iş oluşturan düğüm kendi yüküne bakar. Yükü bir eşiği aşıyorsa rastgele seçtiği başka bir düğüme "yükün ne?" diye sorar (Şekil 12.39). Yoklama sınırlı sayıda yapılır, örneğin en fazla N = 3 düğüm. Eşiğin altında bir düğüm bulunursa iş ona gönderilir; bulunamazsa yerelde çalışır. Eager ve arkadaşları 1986'da bu algoritmayı incelediler. Şekildeki örnekte eşik 7'dir ve yükü 9 olan düğüm 3 ve 1 yüklü iki düğümü yoklar. İşi en az yüklü olana, yükü 1 olan düğüme gönderir.

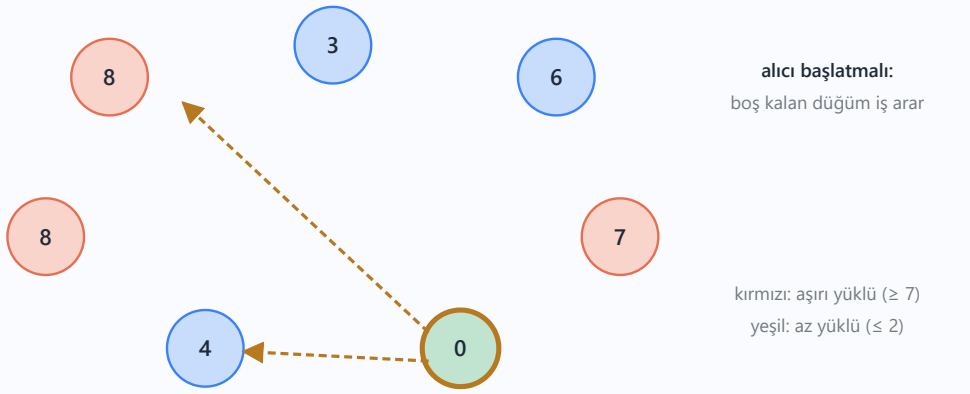




Şekil 12.39: Gönderici başlatmalı yük dengeleme. Üstte yükü 9 olan düğüm rastgele iki düğümü yokluyor; altta iş, yükü 1 olan düğüme gönderildi. Kırmızı: aşırı yüklü (≥ 7), yeşil: az yüklü (≤ 2).

Bu algoritmanın sakıncası, sistem bütünüyle yüklüyken ortaya çıkar. Her düğüm iş yollamaya çalışır, her yoklama boşa gider. Yoklamaların kendisi de ağa ve düğümlere ek yük bindirir. Yük arttıkça sistem daha çok zamanını yoklamaya harcar.

Alıcı başlatmalı (*receiver-initiated*) algoritma sorunu tersinden ele alır. Bir işi biten ve boş kalan düğüm kendisi iş arar. Rastgele düğümleri yoklar ve yüklü birini bulursa ondan bir iş alır (Şekil 12.40). Yoklama maliyeti zaten boş olan düğüme düşer. Sistem yüklüyken boş düğüm azdır ve yoklama da az olur. Bu yüzden alıcı başlatmalı algoritma yük yüksekken iyi çalışır; gönderici başlatmalı ise yük düşükken. İkisini birleştiren algoritmalar da vardır.



Şekil 12.40: Alıcı başlatmalı yük dengeleme: işi biten ve yükü 0'a düşen düğüm aşırı yüklü düğümlerden iş ister.

Alıcı başlatmalı dengelemenin bugünkü karşılığı **iş çalma**dır (*work stealing*). Her işçi iş parçacığının kendi iş kuyruğu vardır. Kuyruğu boşalan işçi başka bir işçinin kuyruğunun öbür ucundan iş çalar. Cilk, Java'nın `ForkJoinPool`'u, Go'nun ve Rust'taki Tokio'nun çalışma zamanları bu yöntemi kullanır.

12.7 Sanallaştırma

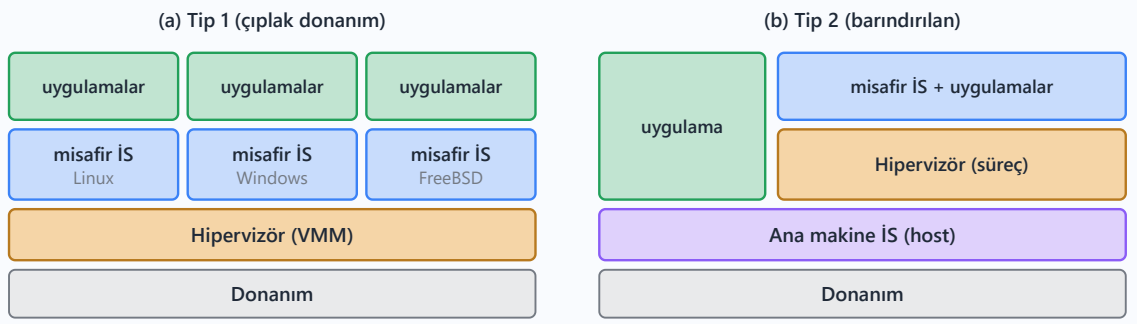
Bir veri merkezindeki sunucuların çoğu zamanlarının büyük kısmında az yüklüdür. Her hizmet için ayrı bir fiziksel makine ayırmak donanımı, enerjiyi ve yeri israf eder. Sanallaştırma tek bir fiziksel makinede birden çok sanal makine çalıştırarak bu sorunu çözer. Her sanal makine kendi işletim sistemini çalıştırır ve diğerlerinden yalıtılmıştır. Bu kesim sanal makineleri çalıştıran yazılımı, işlemcinin ve belleğin nasıl sanallaştırıldığını ve konteynerleri anlatır.

12.7.1 Hipervizörler: tip 1 ve tip 2

Tanım 12.8

Hipervizör (*hypervisor*) ya da sanal makine izleyicisi (*virtual machine monitor*, VMM), sanal makineleri oluşturan ve çalıştıran yazılım katmanıdır. Her sanal makineye, gerçek bir bilgisayarın donanımını taklit eden bir ortam sunar. Sanal makinede çalışan işletim sistemine **misafir** (*guest*) denir.

Hipervizörler nerede çalıştıklarına göre ikiye ayrılır (Şekil 12.41). **Tip 1** hipervizör doğrudan donanım üzerinde çalışır; altında başka bir işletim sistemi yoktur. Çıplak donanım (*bare metal*) hipervizörü de denir. VMware ESXi, Xen (bugün XCP-ng) ve Microsoft Hyper-V bu türdendir. Linux'un KVM modülü, Linux çekirdeğinin kendisini bir tip 1 hipervizöre dönüştürür. Bulutların çoğu KVM ya da onun türevlerini kullanır; AWS'nin Nitro sistemi ve Google'ın bulutu bunlara örnektir. Windows'ta Hyper-V etkinleştirildiğinde Windows da hipervizörün üzerinde, ayrıcalıklı bir "kök bölümde" çalışmaya başlar.



Şekil 12.41: (a) Tip 1 hipervizör doğrudan donanımda çalışır; (b) tip 2 hipervizör ana makine işletim sisteminin üzerinde bir süreçtir.

Tip 2 hipervizör ise sıradan bir uygulama gibi bir ana makine (*host*) işletim sistemi üzerinde çalışır. VirtualBox, VMware Workstation ve Fusion, Parallels ve UTM bu türdendir. Aygıt erişimini ana makine işletim sistemine bırakır; bu yüzden G/Ç'de ek yük daha fazladır. Saf bir tip 2 hipervizör misafirin

komutlarını yorumlar ya da ikili çeviriyle dönüştürür; KVM'siz QEMU böyle çalışır. Uygulamadaki tip 2 hipervizörler ise bir çekirdek modülü yükleyerek işlemcinin sanallaştırma desteğini kullanır. Bir hipervizörün türünü, donanım desteği kullanıp kullanmadığı değil, nerede çalıştığı belirler.

Bir makinenin verimli biçimde sanallaştırılabilmesi için gereken koşulu Popek ve Goldberg 1974'te ortaya koydular. Komutları iki gruba ayıralım. **Hassas** komutlar makinenin durumunu, örneğin sayfa tablosunu ya da G/Ç'yi etkileyen ya da ona göre farklı davranan komutlardır. **Ayrıcalıklı** komutlar ise kullanıcı modunda yürütülmeye çalışıldığında tuzağa düşen komutlardır.

> **Teorem 12.2 (Popek–Goldberg)**
>
> Bir makinenin hassas komutlarının hepsi ayrıcalıklıysa, yani kullanıcı modunda tuzağa düşüyorsa, o makine için verimli bir hipervizör kurulabilir.

Koşulun anlamı şudur. Misafir işletim sistemi kullanıcı modunda çalıştırılır. Sıradan komutları doğrudan donanımda yürütülür. Hassas bir komut yürütmeye kalktığında tuzağa düşer ve denetim hipervizöre geçer. Hipervizör komutun etkisini sanal makine için taklit eder. Klasik x86 bu koşulu sağlamıyordu. Örneğin `POPF` komutu kesme bayrağını da değiştirir; ama kullanıcı modunda bu değişikliği tuzağa düşmeden, sessizce yok sayar. Misafir işletim sistemi kesmeleri kapattığını sanır, hipervizörün de bundan haberi olmaz. Bu sorun önce ikili çeviriyle, sonra donanım desteğiyle çözüldü.

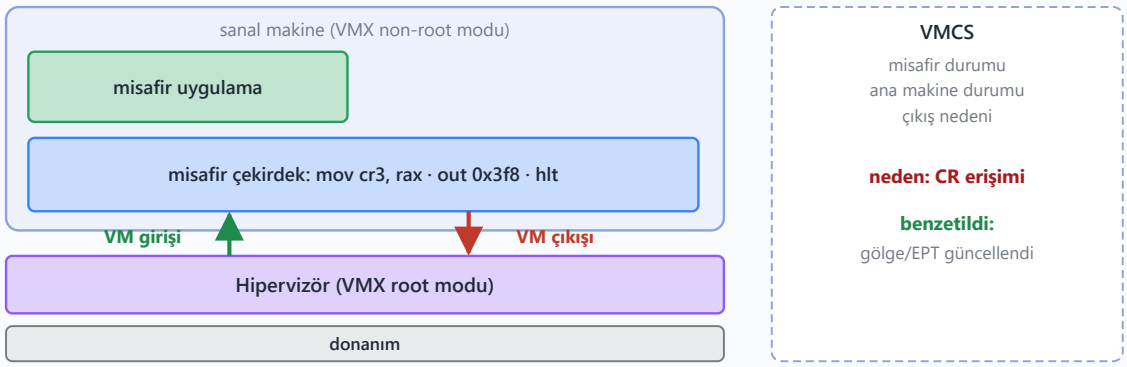
Sanallaştırmanın kullanım nedenleri bugün iyi bilinir. Birçok sunucu tek bir makinede birleştirilir. Çalışan bir sanal makine başka bir fiziksel makineye kesintisiz taşınabilir; buna canlı göç denir. Bir sanal makinenin anlık görüntüsü alınıp gerektiğinde ona geri dönülebilir. Bulut bilişimin temeli de budur.

12.7.2 Tuzak ve benzet: VT-x ve AMD-V

Intel 2005'te VT-x, AMD 2006'da AMD-V ile x86'ya donanım sanallaştırma desteği ekledi. Bu destek işlemciye yeni bir çalışma kipi kazandırır. Hipervizör **root** kipinde, misafir **non-root** kipinde çalışır. Misafir işletim sistemi non-root kipinin kendi çekirdek modunda çalışır ve kendini gerçekten çekirdek modunda sanar. Ama gerçek denetim hipervizördedir.

Misafir çekirdek hassas bir komut yürütmek istediğinde şunlar olur (Şekil 12.42):

1. Misafir çekirdek hassas bir komut yürütür: sayfa tablosu tabanını değiştiren `mov cr3`, bir G/Ç portuna yazan `out` ya da işlemciyi durduran `hlt`.
2. Donanım komutu çalıştırmaz. Bunun yerine bir **VM çıkışı** (*VM exit*) ile denetimi hipervizöre verir. Misafirin durumu ve çıkışın nedeni VMCS adlı bir bellek yapısına yazılır.
3. Hipervizör komutun etkisini **benzetir** (*emulate*). Örneğin gölge tablo ya da EPT'yi günceller, sanal aygıtın durumunu değiştirir.
4. Hipervizör bir **VM girişi** (*VM entry*) ile misafire geri döner; misafir bir sonraki komuttan devam eder.



Şekil 12.42: Tuzak ve benzet. Misafirin hassas komutu VM çıkışına yol açar; neden VMCS'ye yazılır, hipervizör komutu benzetir ve VM girişiyle misafire döner.

Misafirin yalnız hassas komutları tuzağa düşer; geri kalan komutları doğrudan donanımda, tam hızda çalışır. Yine de bir VM çıkışı yaklaşık bir mikrosaniye sürer. Sık çıkış yapan bir iş yükü bu yüzden yavaşlar. Donanım desteğinden önce VMware (1999) bu sorunu **ikili çeviriyle** çözüyordu. Misafir çekirdeğin kodu çalıştırılmadan önce taranır, hassas komutlar hipervizörü çağıran güvenli kod parçalarıyla değiştirilir. Çevrilmiş kod önbelleğe alınır ve tekrar tekrar kullanılır.

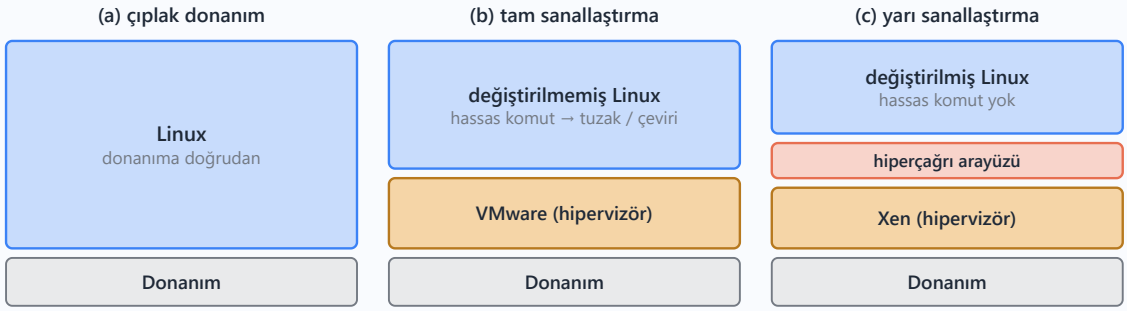
12.7.3 Yarı sanallaştırma

Tuzak ve benzetmenin bedelinden kaçmanın bir yolu daha vardır: misafir işletim sistemini değiştirmek.

Tanım 12.9

Yarı sanallaştırmada (*paravirtualization*) misafir işletim sistemi bir hipervizör üzerinde çalıştığını bilir. Hassas komutlar yerine hipervizöre doğrudan çağrılar yapar; bu çağrılara **hiperçağrı** (*hypercall*) denir.

Hiperçağrı bir sistem çağrısının bir alt katmandaki benzeridir. Uygulama işletim sisteminden nasıl hizmet istiyorsa, misafir işletim sistemi de hipervizörden öyle ister. Tuzak ve benzetme maliyeti olmadığı için yöntem daha hızlıdır. Bedeli, misafir işletim sisteminin kaynak kodunda değişiklik gerektirmesidir. Xen (2003) bu yaklaşımı ilk yaygınlaştıran hipervizördür; Linux'un değiştirilmiş bir sürümü Xen üzerinde neredeyse doğal hızda çalışıyordu (Şekil 12.43).



Şekil 12.43: (a) Çıplak donanımda Linux, (b) VMware üzerinde değiştirilmemiş Linux (hassas komutlar tuzağa düşer ya da çevrilir), (c) Xen üzerinde hiperçağrı arayüzü kullanan değiştirilmiş Linux.

Her hipervizörün kendi hiperçağrı arayüzü olması bir sorundu: bir çekirdeğin ayrı ayrı sürümleri gerekiyordu. Tek bir çekirdeğin hem çıplak donanımda hem de farklı hipervizörlerde çalışabilmesi için VMI (*Virtual Machine Interface*) adlı ortak bir arayüz önerildi. Donanım desteği yaygınlaşınca gerek kalmadı ve VMI 2011'de Linux'tan kaldırıldı. Bugün yarı sanallaştırma daha çok sürücülerde yaşar. `virtio` disk ve ağ sürücüleri, yarı sanal saat ve Hyper-V'nin "aydınlatmaları" (*enlightenments*) bunun örnekleridir. Günümüz bulutlarında CPU ve bellek donanım desteğiyle tam sanallaştırılır. G/Ç ise ya `virtio` gibi yarı sanal sürücülerle ya da SR-IOV ile bir aygıtın parçasının doğrudan sanal makineye atanmasıyla yapılır.

12.7.4 Bellek sanallaştırma: gölge tablo ve EPT

Sanal makinede adres çevirisi iki katmanlıdır (Şekil 12.44). Misafirdeki bir süreç **misafir sanal adresi** (GVA) üretir. Misafir işletim sisteminin sayfa tabloları onu **misafir fiziksel adresine** (GPA) çevirir. Ama misafirin "fiziksel" belleği de aslında hipervizörün ayırdığı bir bellek parçasıdır. Hipervizör GPA'yı gerçek **ana makine fiziksel adresine** (HPA) çevirmelidir.



Şekil 12.44: İki düzeyli adres çevirisi: misafir sayfa tablosu GVA → GPA, EPT/NPT GPA → HPA. Gölge sayfa tablosu GVA → HPA eşlemesini doğrudan tutar.

Donanım desteğinden önceki çözüm **gölge sayfa tablolarıdır** (*shadow page tables*). Hipervizör her misafir süreci için GVA'yı doğrudan HPA'ya eşleyen bir tablo tutar ve MMU'ya bu tabloyu verir. Misafirin kendi tablolarını salt okunur yapar. Misafir bir tablo girdisini değiştirmeye kalkınca tuzağa düşer; hipervizör değişikliği gölge tabloya da yansıtır. Çeviri hızlıdır, ama misafirin her tablo değişikliği bir VM çıkışı demektir. Süreç oluşturan ya da çok sayfa hatası alan iş yükleri bu yüzden çok yavaşlar.

Intel'in EPT'si (*Extended Page Tables*, 2008) ve AMD'nin NPT'si (*Nested Page Tables*) bu işi donanıma verir. MMU iki tabloyu birlikte yürür. Misafir kendi tablosunu tuzağa düşmeden değiştirir; hipervizör yalnız GPA → HPA tablosunu yönetir. Bedel TLB ıskasındadır. Misafir tablosunun her girdisi bir GPA'dır; okunabilmesi için önce EPT'den geçirilmelidir.

Örnek 12.13

Misafir ve ana makine 4 düzeyli sayfa tabloları kullanıyor. TLB ıskasında ve hiçbir tablo girdisi önbellekte değilken en kötü durumda kaç bellek erişimi gerekir (son veri erişimi hariç)?

1. **Misafir tablosunun bir düzeyi.** Girdinin adresi bir GPA'dır. Önce 4 EPT düzeyi yürünerek HPA'ya çevrilir (4 erişim), sonra girdinin kendisi okunur (1 erişim): 5 erişim.
2. **Dört düzey.** 4 × 5 = 20 erişim. Sonunda verinin GPA'sı elde edilir.
3. **Son çeviri.** Verinin GPA'sı da EPT'den geçmelidir: 4 erişim daha. Toplam 20 + 4 = 24.

Genel formül $(n + 1)(m + 1) - 1$'dir; n misafir, m EPT düzey sayısıdır. $n = m = 4$ için $(4 + 1)^2 - 1 = 24$ erişim eder. İki taraf da 5 düzeyli olursa $(5 + 1)^2 - 1 = 35$ erişim gerekir. Sanallaştırma olmadan aynı ıska yalnız 4 erişim isterdi.

Bu uzun yürüyüşün etkisini azaltmanın iki yolu vardır. Büyük sayfalar (2 MiB ya da 1 GiB) hem tablo düzeyi sayısını hem de TLB ıskalarını azaltır. TLB girdilerinin sanal makine kimliğiyle etiketlenmesi (Intel'de VPID, AMD'de ASID) ise sanal makineler arasında geçişte TLB'nin boşaltılmasını gereksiz kılar.

Hipervizörler belleği misafirlere fiziksel bellekten fazla da dağıtabilir (aşırı tahsis). İki düzenek bunu destekler. **Balon sürücüsü**, misafirin içinde çalışan bir sürücüdür. Hipervizör bellek isteyince misafirden bellek ayırır ve bu sayfaları hipervizöre geri verir; misafir kendi sayfa yer değiştirme politikasıyla neyin diske gideceğine kendisi karar verir. Linux'un **KSM** (*Kernel Same-page Merging*) düzeneği ise farklı sanal makinelerdeki aynı içerikli sayfaları bulur ve tek bir kopyada birleştirir. Biri sayfaya yazarsa yazınca kopyala ile ayrı bir kopya çıkarılır.

12.7.5 Konteynerler ve sanal makineler

Her sanal makine kendi çekirdeğini çalıştırır; başlaması saniyeler alır ve gigabaytlarca bellek tutar. Birçok uygulama için bu fazladır. Uygulamaları birbirinden yalıtmak çoğu zaman yeterlidir; ayrı bir çekirdek gerekmez.

Tanım 12.10

Konteyner (*container*), ana makinenin tek çekirdeğini paylaşan ve ad alanları ile cgroups yardımıyla diğer süreçlerden yalıtılmış bir süreç grubudur.

Linux konteynerleri üç çekirdek düzeneğine dayanır. **Ad alanları** (*namespaces*) bir süreç grubuna kendi süreç numaralarını, dosya sistemi ağacını, ağ arayüzlerini ve makine adını gösterir; grup diğer süreçleri göremez. **Denetim grupları** (cgroups) grubun kullanabileceği CPU, bellek ve G/Ç miktarını sınırlar. **seccomp** ise grubun yapabileceği sistem çağrılarını kısıtlar. Bir konteynerin içinde bir dağıtımın kütüphaneleri ve araçları bulunur, ama çekirdek ana makinenindir (Şekil 12.45).



Şekil 12.45: (a) Sanal makinelerin her birinde ayrı bir misafir çekirdek vardır; (b) konteynerler tek ana makine çekirdeğini paylaşır.

	Sanal makine	**Konteyner**
Yalıtım sınırı	donanım düzeyi, ayrı çekirdek	aynı çekirdek: namespace, cgroup, seccomp
Başlama / ek yük	saniyeler, GB bellek	milisaniyeler, MB bellek
Farklı işletim sistemi	evet (Linux üstünde Windows)	hayır (aynı çekirdek ailesi)
Örnek	KVM, Hyper-V, ESXi	Docker, Podman, Kubernetes pod'ları

Tablo 12.6: Sanal makine ve konteynerin karşılaştırılması.

Konteynerin yalıtımı sanal makinenin yalıtımından zayıftır. Bütün konteynerler aynı çekirdeği paylaştığı için çekirdekteki bir güvenlik açığı hepsini etkiler. Arada kalan çözümler bu yüzden yaygınlaşıyor. AWS Lambda'nın kullandığı Firecracker, milisaniyeler içinde açılan küçük sanal makineler (mikro-VM) üretir. gVisor, konteynerin sistem çağrılarını kullanıcı alanındaki bir çekirdekte karşılar. Kata Containers her konteyneri hafif bir sanal makinede çalıştırır. Windows'taki WSL 2 de hafif bir Hyper-V sanal makinesinde gerçek bir Linux çekirdeği çalıştırır; Windows'taki Linux konteynerleri bu sanal makinede koşar.

Sık yapılan hata

Konteynerin kendi çekirdeği olduğunu sanmak. Linux 6.8 çalışan bir sunucuda Ubuntu 20.04 tabanlı bir konteynerde `uname -r` yine 6.8'i gösterir; konteynerde yalnız kullanıcı alanı Ubuntu 20.04'tür. Aynı sunucudaki bir sanal makine ise kendi misafir çekirdeğini önyükler ve onun sürümünü gösterir.

12.8 Dağıtık sistemler

Çoklu bilgisayarın düğümleri aynı odadadır, aynı kurumun yönetimindedir ve özel bir ağla bağlıdır. Dağıtık sistemde bu varsayımların hepsi gevşer. Bu kesim dağıtık sistemleri, onları bir arada tutan ağı ve ara yazılımın dört ana türünü anlatır.

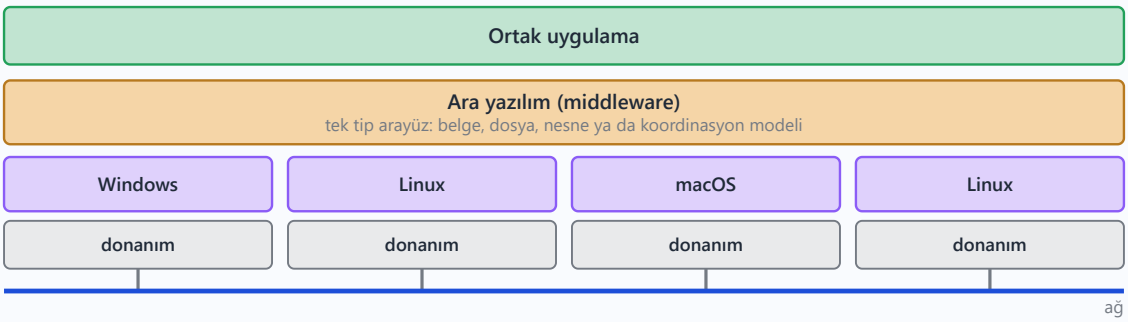
12.8.1 Dağıtık sistem ve ara yazılım

Dağıtık sistem, ağ üzerinden haberleşen özerk bilgisayarlardan oluşur. Her bilgisayar kendi işletim sistemini, belleğini ve kaynaklarını yönetir. Sistem kullanıcıya yine de tek ve tutarlı bir sistem gibi görünmeye çalışır. Düğümler farklı donanımlar ve farklı işletim sistemleri çalıştırabilir, farklı kurumlara ait olabilir ve dünyaya yayılabilir.

Tanım 12.11

Ara yazılım (*middleware*), dağıtık sistemde işletim sistemleri ile uygulamalar arasında duran ve uygulamalara bütün düğümlerde aynı olan bir model sunan yazılım katmanıdır.

Farklı işletim sistemleri ortak bir dili ancak ara yazılım sayesinde konuşur (Şekil 12.46). Ara yazılım dört temel paradigmadan birine dayanır. Belge tabanlı ara yazılımda her şey bir belgedir; Web bunun örneğidir. Dosya tabanlı ara yazılımda her şey bir dosyadır; NFS ve SMB böyledir. Nesne tabanlı ara yazılımda her şey bir nesnedir; CORBA, Java RMI ve gRPC bu türdendir. Koordinasyon tabanlı ara yazılımda ise süreçler ortak bir uzay ya da aracı üzerinden haberleşir; Linda ve yayınla-abone ol sistemleri bu türdendir.



Şekil 12.46: Ara yazılımın konumu: farklı işletim sistemleri üzerinde ortak bir katman ve onun üzerinde ortak uygulama.

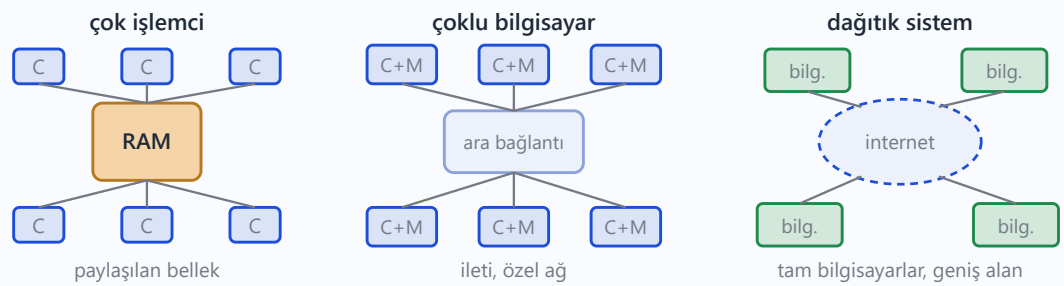
Bütün düğümleri tek bir işletim sistemi gibi yöneten gerçek dağıtık işletim sistemleri de denendi. Amoeba ve Plan 9 bunların en bilinenleridir. Bunlar araştırma sistemleri olarak kaldı. Bugün her düğüm kendi işletim sistemini çalıştırır; dağıtımı ara yazılım ve Kubernetes gibi küme yöneticileri üstlenir.

12.8.2 Üç sistem türünün karşılaştırması

Bu bölümde üç paralel düzen ele alındı: çok işlemci, çoklu bilgisayar ve dağıtık sistem. Tablo 12.7 onları Tanenbaum'un karşılaştırmasıyla yan yana koyar.

Özellik	Çok işlemci	Çoklu bilgisayar	Dağıtık sistem
Düğüm yapısı	CPU	CPU, RAM, ağ arayüzü	tam bilgisayar
Çevre birimleri	hepsi paylaşılır	paylaşılır (belki disk hariç)	her düğümde tam set
Konum	aynı raf	aynı oda	dünya çapında olabilir
Düğümler arası iletişim	paylaşılan RAM	özel ara bağlantı	geleneksel ağ
İşletim sistemi	bir, paylaşılan	çok, aynı	çok, farklı olabilir
Dosya sistemi	bir, paylaşılan	bir, paylaşılan	her düğümün kendi
Yönetim	tek kurum	tek kurum	birçok kurum

Tablo 12.7: Üç sistem türünün karşılaştırması.

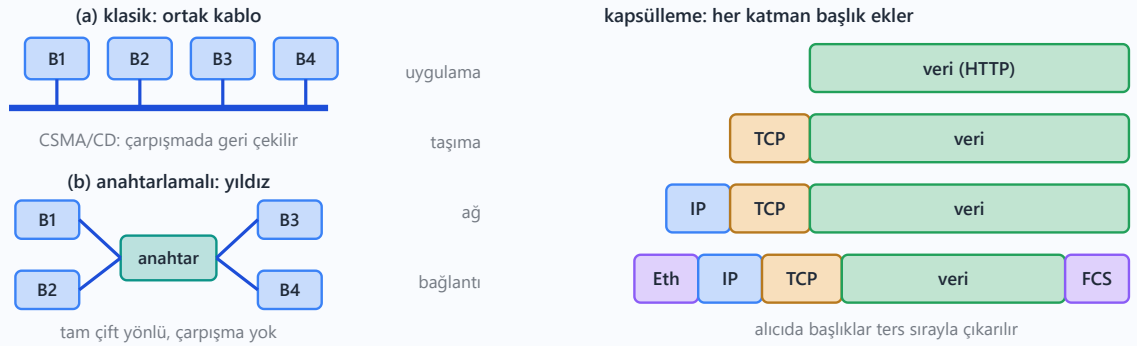


Şekil 12.47: Üç sistem türü: paylaşılan belleğe bağlı CPU'lar; kendi belleği olan düğümler ve özel ara bağlantı;

geniş alan ağıyla bağlı tam bilgisayarlar.

12.8.3 Ağ donanımı ve kapsülleme

Dağıtık sistemin düğümlerini bir ağ bağlar. Yerel ağların en yaygın türü Ethernet'tir (Şekil 12.48). 1976'daki klasik Ethernet'te bütün bilgisayarlar tek bir kabloyu paylaşıyordu. Bir bilgisayar göndermeden önce kabloyu dinler, boşsa gönderirdi. İki bilgisayar aynı anda gönderirse çarpışma olur, ikisi de rastgele bir süre geri çekilip yeniden denerdi. Bu yönteme CSMA/CD denir. Bugün her bilgisayar bir anahtara ayrı bir kabloyla bağlıdır. Bağlantılar 1 ile 400 Gb/s arasında hızlarda ve tam çift yönlü çalışır; çarpışma olmaz.



Şekil 12.48: (a) Klasik ve (b) anahtarlamalı Ethernet; sağda her katmanın pakete başlık eklediği kapsülleme.

Yerel ağlar yönlendiricilerle birbirine bağlanır. Kurumların ağları İnternet servis sağlayıcılarına, onlar da İnternet'in omurgasına bağlanır. Omurga yüksek hızlı fiber hatlardan ve ağların birbirine bağlandığı değişim noktalarından oluşur.

Ağ yazılımı katmanlar hâlinde düzenlenir. Bir uygulama verisi gönderilirken her katman kendi başlığını ekler. Taşıma katmanı TCP başlığını, ağ katmanı IP başlığını, bağlantı katmanı Ethernet başlığını ve sondaki denetim toplamını (FCS) ekler. Buna **kapsülleme** denir. Alıcı tarafta başlıklar ters sırayla çıkarılır ve her katman kendi başlığını yorumlar.

12.8.4 Ağ protokolleri ve hizmet türleri

Ağ hizmetleri iki açıdan sınıflandırılır. **Bağlantı yönelimli** hizmette önce bir bağlantı kurulur, sonra veri gönderilir, en sonda bağlantı sonlandırılır; telefon görüşmesine benzer. **Bağlantısız** hizmette kurulum maliyeti yoktur; her paket bağımsız gönderilir, posta gibi. **Güvenilir** hizmet verinin teslim edildiğini ve sırasını garanti eder; alıcı her paketi bir alındı onayıyla (ACK) bildirir. Güvenilmez hizmet bu garantiyi vermez. Tablo 12.8 İnternet'in temel protokollerini özetler.

Protokol	Katman	Hizmet	Not
IP (v4/v6)	ağ	bağlantısız, güvenilmez	adresleme ve yönlendirme

Protokol	Katman	Hizmet	Not
TCP	taşıma	bağlantı yönelimli, güvenilir bayt akışı	onay, yeniden iletim, akış ve tıkanıklık denetimi
UDP	taşıma	bağlantısız, güvenilmez datagram	sağlama toplamı var, yeniden iletim yok; DNS, oyun, ses
QUIC	taşıma (UDP üstü)	bağlantı yönelimli, güvenilir, şifreli	HTTP/3 (2022) bunu kullanır
DNS	uygulama	ad → IP adresi	hiyerarşik, dağıtık, önbellekli
HTTP/HTTPS	uygulama	Web belgeleri	HTTPS = HTTP + TLS, port 443
SMTP	uygulama	e-posta aktarımı	sunucular arası; okuma IMAP ile
FTP → SFTP	uygulama	dosya aktarımı	FTP şifresizdir; yerini SFTP/HTTPS aldı
SNMP	uygulama	ağ aygıtı izleme	v3 kimlik doğrulama ve şifreleme ekler

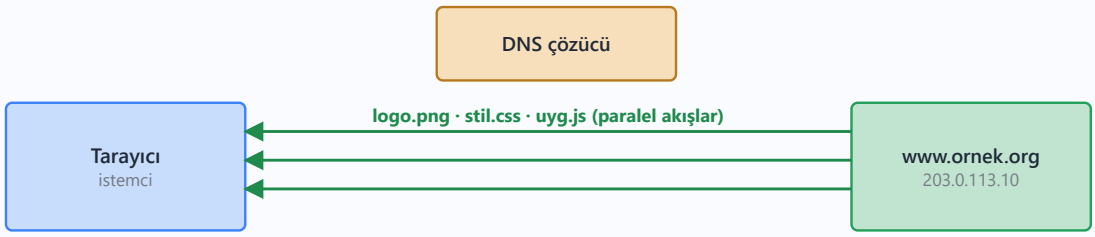
Tablo 12.8: Başlıca İnternet protokolleri.

UDP'nin sağlama toplamı bozuk bir paketi algılar, ama onu düzeltmez ve yeniden göndermez; bozuk paket yalnız atılır. Kayıp ya da bozuk paketleri yeniden göndermek TCP'nin ya da uygulamanın işidir.

12.8.5 Belge tabanlı ara yazılım: Web

Web'de her şey bir belgedir. Belgeler birbirine bağlantılarla bağlanır ve dev bir yönlendirilmiş çizge oluşturur. Kullanıcı tarayıcıya `www.ornek.org/index.html` adresini yazdığında şunlar olur (Şekil 12.49):

1. Tarayıcı, işletim sisteminin çözücüsü üzerinden DNS'e `www.ornek.org`'un IP adresini sorar.
2. Yanıt gelir: 203.0.113.10. Yanıt bir süre önbellekte tutulur.
3. Tarayıcı sunucunun 443 numaralı portuna bir TCP bağlantısı açar ve TLS el sıkışmasını yapar. HTTP/3'te bu iki adım QUIC ile tek adımda birleşir.
4. Tarayıcı `GET /index.html` isteğini gönderir.
5. Sunucu HTML dosyasını döndürür ve tarayıcı metni görüntülemeye başlar.
6. HTML'nin başvurduğu resimler, CSS ve betikler aynı bağlantı üzerinden alınır; HTTP/2 ve HTTP/3 bunları paralel akışlarla taşır. Bağlantı bir süre açık kalır.

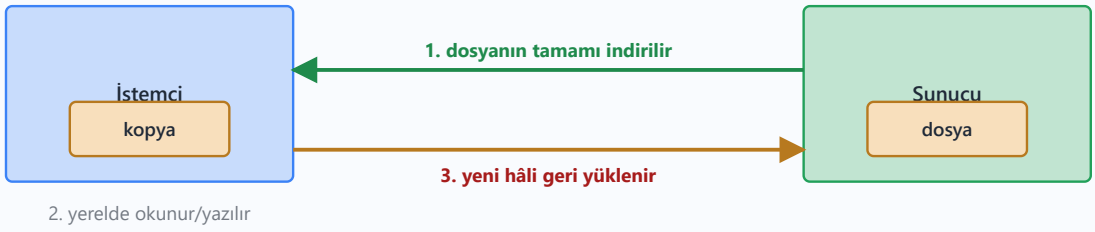


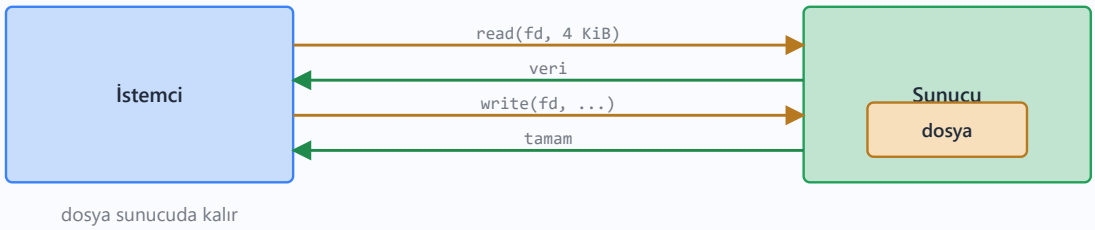
Şekil 12.49: Bir web sayfasının alınmasının son adımı: `logo.png`, `stil.css` ve `uyg.js` aynı bağlantı üzerinden paralel akışlarla gelir.

Eski ders kitaplarında port 80'e bağlanılır ve her nesne için yeni bir TCP bağlantısı açılıp kapatılırdı. Bu HTTP/1.0 dönemine aittir. Bugün varsayılan HTTPS'dir (port 443); bağlantılar kalıcıdır ve HTTP/2–3 çoğullama kullanır.

12.8.6 Dosya sistemi tabanlı ara yazılım

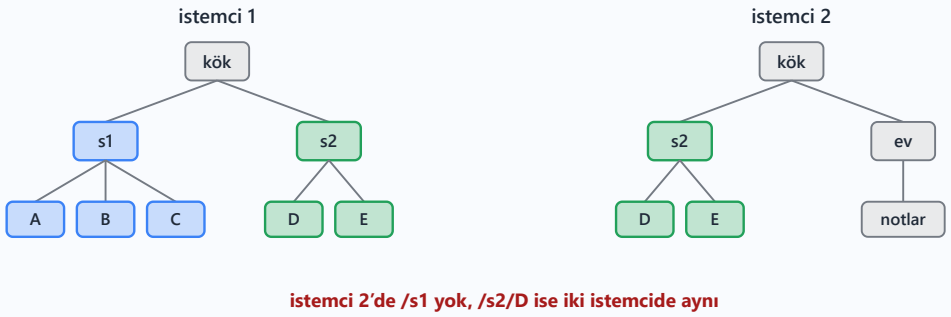
Dosya tabanlı ara yazılım dağıtık sistemi bir dosya sistemi gibi gösterir. İlk tasarım sorusu, dosyaya nasıl erişileceğidir (Şekil 12.50). **Yükleme/indirme modelinde** istemci dosyanın tamamını kendine kopyalar, yerelde okur ve yazar, işi bitince yeni hâlini sunucuya geri yükler. Bu model basittir, ama büyük bir dosyanın birkaç baytını okumak için bütün dosyayı taşımak pahalıdır. FTP ve eski AFS bu modeli kullanır. **Uzaktan erişim modelinde** dosya sunucuda kalır. İstemci `read` ve `write` isteklerini ağ üzerinden gönderir ve yalnız istediği bloklar taşınır. NFS ve SMB bu modeli kullanır.





Şekil 12.50: Üstte yükleme/indirme modeli: dosyanın tamamı indirilir, yerelde kullanılır, geri yüklenir. Altta uzaktan erişim modeli: dosya sunucuda kalır, istekler ağdan gider.

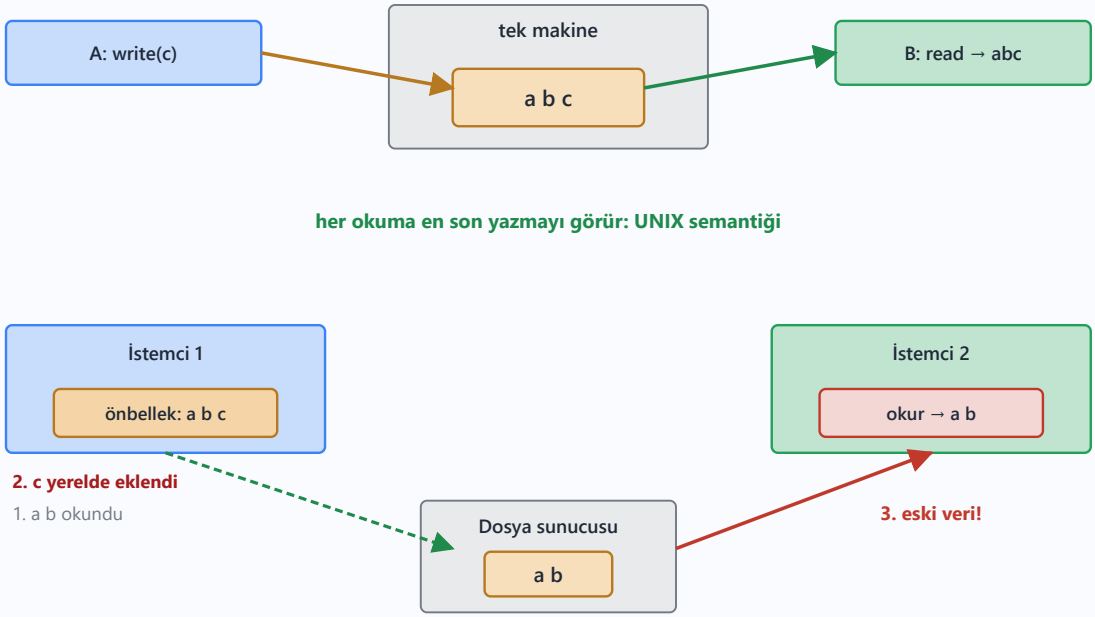
İkinci soru adlandırmadır. En basit çözümde dosya adı makine adını içerir: `makine:/yol`. Bu ad konum şeffaf değildir; dosya başka bir makineye taşınırsa adı değişir. Daha iyi bir çözüm, uzak dosya sistemini yerel ağacın bir dizinine bağlamaktır (*mount*). Bütün istemciler sunucuları aynı yere bağlarsa herkes tek ve ortak bir ad alanı görür. İstemciler farklı şeyler bağlarsa her biri farklı bir görünüm görür (Şekil 12.51). Aynı yol farklı istemcilerde farklı dosyaları gösterebilir. En güçlü çözüm, bütün makinelerde aynı olan ve dosyanın konumunu hiç içermeyen tek bir ad alanıdır. Buna konumdan bağımsızlık denir. Bugünkü örnekler NFSv4, SMB 3 ve Amazon S3 gibi nesne depolama sistemleridir.



Şekil 12.51: İki istemci farklı şeyler bağlamış: istemci 2'de `/s1` yoktur, `/s2/D` ise iki istemcide aynı dosyadır.

12.8.7 Dosya paylaşımının anlamı

Birden çok istemci aynı dosyayı paylaştığında bir yazmanın başkalarına ne zaman görüneceği tanımlanmalıdır. Tek bir makinede yanıt açıktır. A süreci bir dosyaya "c" ekler, ardından B okursa "abc" görür. Her okuma en son yazmayı görür. Buna **UNIX semantiği** ya da sıralı tutarlılık denir (Şekil 12.52, üstte).



Şekil 12.52: Üstte tek makinede UNIX semantiği. Altta istemci 1 "c"yi kendi önbelleğinde eklemiştir; sunucu bunu bilmez ve istemci 2 eski "ab"yi okur.

Dağıtık sistemde istemciler performans için dosyaları önbelleğe alır. İstemci 1 dosyayı önbelleğinde değiştirir ama sunucu bunu henüz bilmez. İstemci 2 sunucudan okur ve eski "ab"yi görür. UNIX semantiğini korumanın tek yolu önbellek kullanmamak ve her işlemi sunucuya göndermektir; bu da çok yavaştır. Dağıtık sistemler bu yüzden daha gevşek anlamlar kullanır (Tablo 12.9).

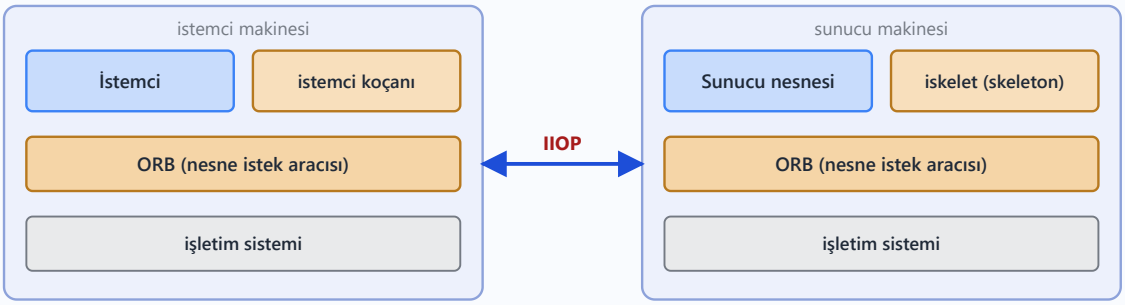
Yaklaşım	Kural	Örnek
UNIX semantiği	her yazma hemen herkese görünür	tek makine; önbelleksiz uzak erişim
Oturum semantiği	değişiklik dosya kapanınca görünür	AFS; NFS'in kapat-aç tutarlılığı
Değişmez dosyalar	dosya yazılamaz, yenisi oluşturulur	nesne depolama, Git nesneleri
İşlemler (*transactions*)	hep ya da hiç, sıralanabilir	veritabanları

Tablo 12.9: Dosya paylaşım anlamları.

Oturum semantiğinde bir istemcinin değişiklikleri dosyayı kapatana kadar yalnız kendisine görünür. Dosya kapatıldıktan sonra açan istemciler yeni sürümü görür. **Değişmez dosyalarda** sorun hiç çıkmaz, çünkü bir dosya hiçbir zaman değiştirilmez; değişiklik yeni bir dosya üretir. **İşlemlerde** bir dizi işlem ya bütünüyle ya da hiç yapılır ve eş zamanlı işlemler sanki sırayla yapılmış gibi sonuç verir. Her çözüm tutarlılık ile performans arasında farklı bir noktada durur.

12.8.8 Nesne tabanlı ara yazılım: CORBA

Nesne tabanlı ara yazılımda dağıtık sistem bir nesneler topluluğudur. Bir istemci, başka bir makinedeki bir nesnenin yöntemini yerel bir nesneninki gibi çağırır. Bu yaklaşımın en kapsamlı örneği OMG'nin 1991'de tanımladığı CORBA'dır (Şekil 12.53).



Şekil 12.53: CORBA'nın temel unsurları: istemci ve istemci koçanı, sunucu nesnesi ve iskeleti, iki taraftaki ORB'ler ve aralarındaki IIOP protokolü.

- **IDL** (*Interface Definition Language*): nesnelerin arayüzleri programlama dilinden bağımsız bir dille tanımlanır. IDL derleyicisi bu tanımdan istemci koçanını ve sunucu tarafındaki **iskeleti** (*skeleton*) üretir.
- **ORB** (*Object Request Broker*, nesne istek aracısı): nesnenin yerini bulur, çağrıyı paketler ve IIOP protokolüyle karşı tarafa iletir. İstemci nesnenin hangi makinede olduğunu bilmez; buna ağ şeffaflığı denir.
- **Statik çağırma** (SII): istemci, derleme zamanında IDL'den üretilmiş koçanları kullanır. Hızlıdır ve türler derleme zamanında denetlenir.
- **Dinamik çağırma** (DII): istemci, çağıracağı arayüzü çalışma zamanında bir arayüz deposundan keşfeder ve çağrıyı kendisi kurar.

SII ve DII iki ayrı düzenektir; SII, DII'nin derlenmiş bir sürümü değildir. CORBA bugün daha çok eski sistemlerde yaşar. Yeni sistemlerde gRPC ile Protocol Buffers, REST ve Java RMI tercih edilir. Ama IDL'den koçan üretmek, ORB benzeri bir aracı ve konumdan bağımsız nesne başvuruları gibi fikirleri bu sistemlere geçmiştir.

12.8.9 Koordinasyon tabanlı ara yazılım: Linda

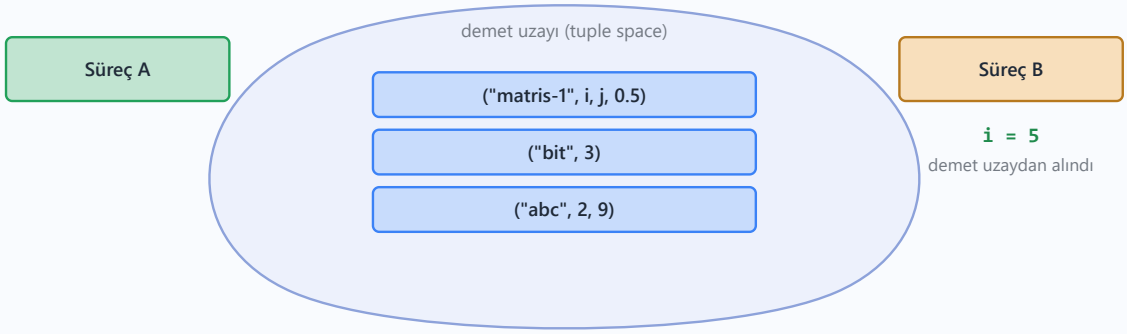
Koordinasyon tabanlı ara yazılımda süreçler birbirini tanımaz. Doğrudan haberleşmek yerine ortak bir ortama veri koyar ve oradan veri alır. Bu yaklaşımın klasik örneği Gelernter'in 1985'te tanıttığı Linda'dır.

Tanım 12.12

Demet uzayı (*tuple space*), Linda'da bütün süreçlerin paylaştığı kalıcı bir demetler kümesidir. **Demet** (*tuple*), türlü alanlardan oluşan bir kayıttır: `("abc", 2, 9)` gibi.

Linda'nın dört temel işlemi vardır (Şekil 12.54):

- `out` bir demeti uzaya koyar ve bloke olmaz.
- `in` bir **şablonla** eşleşen bir demeti arar, uzaydan çıkarır ve alanlarını değişkenlere atar. Eşleşen demet yoksa çağıran bloke olur ve uyan bir demet konana kadar bekler.
- `rd` `in` gibi çalışır, ama demeti uzaydan çıkarmaz; yalnız okur.
- `eval` yeni bir süreç başlatır; süreç bitince sonucu bir demet olarak uzaya konur.



Şekil 12.54: Demet uzayı. B süreci `in("abc", 2, ? i)` çağırdı; eşleşen `("abc", 2, 5)` demeti uzaydan alındı ve i = 5 oldu.

Bir şablonun bir demetle eşleşmesi için üç koşul gerekir. Alan sayıları aynı olmalıdır. Karşılıklı alanlar aynı türde olmalıdır. Şablondaki sabit alanlar da demettekilere eşit olmalıdır. `? i` biçimindeki alanlar **biçimsel parametredir**: türü uyan her değerle eşleşir ve o değer i'ye atanır. Birden çok demet eşleşirse hangisinin seçileceğini Linda tanımlamaz.

Bu basit işlemlerle senkronizasyon da kendiliğinden gelir. Örneğin bir semafor, uzaya konan özdeş demetlerle gerçeklenebilir: `out("sem")` V işlemi, `in("sem")` P işlemidir. Bir iş havuzu da benzer biçimde kurulur: yönetici işleri `out` ile koyar, işçiler `in` ile alır.

12.8.10 Kod izleme: Linda `in()`

Aşağıdaki işlev, demet uzayını bir bağlı liste olarak tutan basit bir Linda gerçeklemesinde `in()`'in arama kısmıdır. `s` şablondur; `n` alan sayısını, `tur[k]` k. alanın türünü, `sabit[k]` o alanın sabit olup olmadığını tutar.

```
Demet *in_ara(Sablon *s) {
  for (Demet *d = uzay; d; d = d->snr) {
    if (d->n != s->n) continue;
    int ok = 1;
    for (int k = 0; ok && k < s->n; k++)
      if (d->tur[k] != s->tur[k] ||
          (s->sabit[k] && !esit(d, s, k)))
        ok = 0;          /* alan uymadı */
    if (ok) { cikar(d); return d; }
  }
  return NULL;           /* bloke ol */
}
```

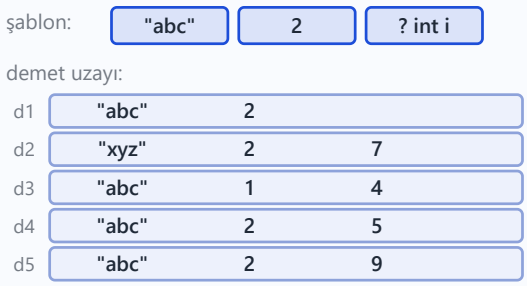
Dış döngü uzaydaki demetler üzerinde sırayla gezinir. Alan sayısı farklıysa demet hemen atlanır (`continue`). Aksi hâlde iç döngü alanları tek tek karşılaştırır. Tür farklıysa ya da alan sabit olup değeri farklıysa `ok` 0 olur ve iç döngü durur. Bütün alanlar uyarsa demet uzaydan çıkarılır ve döndürülür. Hiçbir demet uymazsa işlev `NULL` döndürür; çağıran süreç bloke olur.

Örnek 12.14

Uzayda sırayla d1 = ("abc", 2), d2 = ("xyz", 2, 7), d3 = ("abc", 1, 4), d4 = ("abc", 2, 5) ve d5 = ("abc", 2, 9) var. Süreç `in("abc", 2, ? int i)` çağrısını üç kez yapıyor.

Çağrı	Demet	Denetim	Sonuç
1	d1	2 alan ≠ 3 alan	`continue`
1	d2	alan 0: "xyz" ≠ "abc"	ok = 0
1	d3	alan 0 uyar; alan 1: 1 ≠ 2	ok = 0
1	d4	sabitler eşit, alan 2 tam sayı	d4 çıkarılır, i = 5
2	d1, d2, d3	yukarıdaki gibi	atlanır
2	d5	sabitler eşit, alan 2 tam sayı	d5 çıkarılır, i = 9
3	d1, d2, d3	yukarıdaki gibi	`NULL` : süreç bloke olur

İlk çağrıda d5 de eşleşirdi, ama listede önce d4 bulunduğu için o alındı. Üçüncü çağrı ancak biri uyan bir demeti `out` ile koyduğunda sürer. (Arama döngüsü Python'a birebir aktarılıp aynı uzayla çalıştırıldı.)



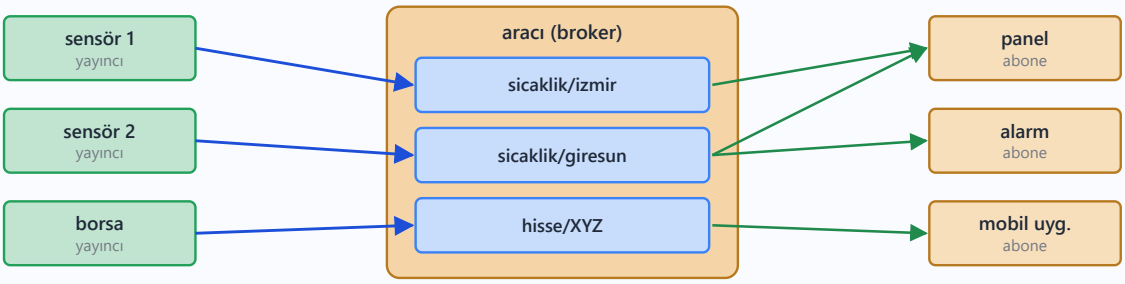
Şekil 12.55: İzlemenin başı: şablon ve uzaydaki beş demet.

Sık yapılan hata

Eşleşmede yalnız değerlere bakmak. `in("abc", ? int x, 5)` şablonu `("abc", 2.0, 5)` demetiyle eşleşmez: ikinci alanın türü kayan noktalıdır, tam sayı değildir. Tür denetimi değer denetiminden önce gelir.

12.8.11 Yayınla-abone ol ve Jini

Koordinasyonun bugün en yaygın biçimi **yayınla-abone ol** (*publish-subscribe*) modelidir (Şekil 12.56). Yayıncı bir iletiyi bir konuya yayınlar. Abone ilgilendiği konulara abone olur. Aradaki **aracı** (*broker*) her iletiyi o konunun bütün abonelerine iletir. Yayıncı ile abone birbirinin kimliğini bilmez; aynı anda çalışmaları da gerekmez. Bu iki özelliğe uzayda ve zamanda ayrıklık denir.



Şekil 12.56: Yayınla-abone ol: sensörler ve borsa konulara yayınlar, aracı iletileri konuya abone olan panel, alarm ve mobil uygulamaya dağıtır.

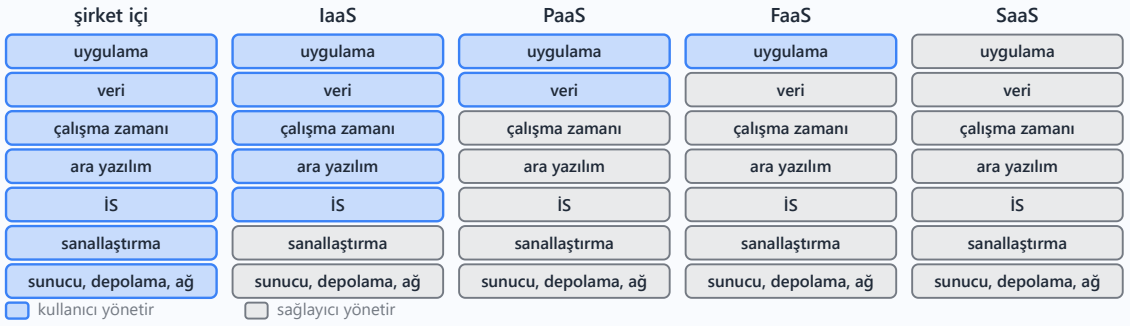
Aboneler ve konular çalışma sırasında eklenip çıkarılabilir; sistem bu yüzden esnek ve ölçeklenebilirdir. Teslim garantisi aracının ayarına bağlıdır: en çok bir kez, en az bir kez ya da iletilerin kalıcı bir kuyrukta saklanması. Bugünkü örnekler nesnelerin interneti için MQTT, Apache Kafka, RabbitMQ ve bulut sağlayıcılarının kuyruk hizmetleridir.

Jini, Sun'ın 1998'de tanıttığı bir koordinasyon sistemidir. Ağa katılan Java aygıtları önceden yapılandırma gerekmeden bir arama hizmetini bulur ve sundukları hizmetleri ona kaydeder. Bir istemci arama hizmetinden uygun bir hizmet bulur ve onu Java RMI ile kullanır. Kayıtlar bir **kiralama** (*lease*) süresiyle yapılır. Süre yenilenmezse kayıt kendiliğinden düşer; böylece çöken aygıtlar hizmet listesinden temizlenir. Jini, Apache River adıyla sürdürüldü ama bugün etkin değildir. Fikirleri ise servis keşfinde yaşıyor: mDNS/DNS-SD, Consul ve Kubernetes servisleri.

12.8.12 Bulut ve kümeler

Küme, hızlı bir ağla bağlı ve tek bir yönetim altındaki sunuculardır; yüksek başarımlı hesaplama kümeleri ve veri merkezleri buna örnektir. **Bulut** ise bu kaynakların ağ üzerinden, isteğe bağlı olarak ve kullandıkça öde biçiminde sunulmasıdır. AWS, Azure ve Google Cloud en büyük bulut sağlayıcılarıdır.

Bulut hizmetleri, katmanların ne kadarını sağlayıcının yönettiğine göre ayrılır (Şekil 12.57). Şirket içi bir sistemde her katmanı kullanıcı yönetir. IaaS'ta (*Infrastructure as a Service*) sağlayıcı sunucuları, depolamayı, ağı ve sanallaştırmayı yönetir; kullanıcı sanal makinesinde işletim sisteminden yukarısını yönetir. PaaS'ta (*Platform as a Service*) kullanıcı yalnız uygulamasını ve verisini getirir. FaaS'ta (*Function as a Service*) kullanıcı yalnız tek tek işlevler yazar; sağlayıcı onları her istekte çalıştırır. SaaS'ta (*Software as a Service*) kullanıcı hazır bir uygulamayı kullanır ve hiçbir katmanı yönetmez.



Şekil 12.57: Bulut hizmet modelleri: şirket içi, IaaS, PaaS, FaaS ve SaaS. Mavi katmanları kullanıcı, gri katmanları sağlayıcı yönetir.

Bulutun altyapısı bu bölümün konularının bir araya gelmesidir. Fiziksel sunucular çok çekirdekli NUMA makineleridir. Üzerlerinde KVM ya da Nitro gibi hipervizörler sanal makineler çalıştırır. Sanal makinelerin ya da sunucuların üzerinde konteynerler koşar. Kubernetes bu konteynerleri (pod'ları) düğümlere çizelgeler, çökenleri yeniden başlatır ve yükü dağıtır; bir bakıma dağıtık işletim sisteminin bugünkü karşılığıdır. Düğümler RDMA destekli ağlarla bağlanır, iş yükleri yük dengeleyicilerle dağıtılır ve hizmetler yayınla-abone ol kuyruklarıyla haberleşir.

12.9 Özet

- Paralel sistemler üç düzende kurulur: paylaşımlı bellekli **çok işlemci**, ileti geçiren **çoklu bilgisayar** ve geniş alana yayılmış **dağıtık sistem**. Saat hızı 2005'ten beri yerinde saydığı için transistör bütçesi çekirdek sayısına gidiyor. **Amdahl yasasına** göre hızlanma 1 / ((1 − p) + p / n)'dir ve 1 / (1 − p)'yi aşamaz.
- **UMA** makinelerinde her bellek sözcüğüne aynı sürede erişilir. Tek veri yolu birkaç CPU'da doyar. **Çapraz çubuk** engellemesizdir ama n^2 kesişim ister. **Omega ağı** $(n/2)\cdot\log_2 n$ anahtarla kurulur, hedef adresin bitleriyle yönlendirir ve engellemelidir.
- **NUMA**'da uzak erişim yerelden yavaştır; ortalama erişim $y \cdot t_{yerel} + (1 - y) \cdot t_{uzak}$'tır. İşletim sistemi ilk dokunma, afinite ve otomatik NUMA dengelemesiyle veriyi işin yanında tutar. CC-NUMA'da tutarlılık bir **dizinle** sağlanır; iletiler yalnız ilgili düğümlere gider.
- **Önbellek tutarlılığı**, aynı adrese yapılan yazmaların herkesçe aynı sırada görülmesidir. **MESI**'de E'deki satıra yazmak sessizdir, S'deki satıra yazmak geçersiz kılma ister, M'deki satır başkasına verilirken belleğe de yazılır.
- **Yanlış paylaşımda** bağımsız değişkenler aynı 64 baytlık satırda olduğu için satır çekirdekler arasında gidip gelir. Çözüm hizalama ve dolgudur.
- Çok çekirdekli yongalar yongacıklardan, büyük–küçük çekirdeklerden ve hızlandırıcılardan oluşur. **SMT** bir çekirdeği birden çok mantıksal CPU olarak gösterir; kazanç %10–30'dur ve güvenlik riski taşır.
- Üç işletim sistemi modeli vardır. Her CPU'nun kendi işletim sistemi paylaşım sunmaz. **Usta-kölede** usta darboğazdır. **SMP**'de herkes çekirdeği çalıştırır ve paylaşılan yapılar ince taneli kilitlerle korunur.
- Çok işlemcide **TSL** okuma ve yazma süresince yolu ya da satırı kilitlemelidir. **Bilet kilidi** adildir; **TTAS** bekleyenleri yerel kopyada döndürür; **MCS** her bekleyene ayrı bir bayrak verir. Beklenen bekleme 2C'den kısaysa dönmek, uzunsa uyumak ucuzdur; 2C dönüp sonra uyumak en kötü durumda iki kat maliyet verir.
- Ortak hazır kuyruğu yükü dengeler ama tek kilit ve soğuk önbellek sorunu taşır; bugünkü sistemler CPU başına kuyruk ve yük dengeleme kullanır. **Alan paylaşımı** bir işe CPU'ları bitene kadar ayırır. **Çete çizelgelemede** birbirine bağlı iş parçacıkları aynı dilimde, farklı CPU'larda çalışır.
- Çoklu bilgisayarda iletiler beş kez kopyalanabilir; RDMA kopyaları kaldırır. Topolojilerin çapı halkada n / 2, k × k ızgarada 2(k − 1), hiperküpte $\log_2 n$'dir. **E-küp** yönlendirmesi farklı bitleri soldan sağa düzeltir. **Kesip geç**, sakla ve iletin atlama sayısıyla büyüyen gecikmesini ortadan kaldırır.
- Gönderim bloke eden ya da etmeyen olabilir. **RPC** uzak çağrıyı koçanlarla yerel çağrı gibi gösterir; işaretçiler ve kısmi hatalar zorluk çıkarır. **DSM** sayfa hatalarıyla paylaşımlı bellek yanılsaması yaratır. Süreç ataması bir çizge bölümleme problemidir; dağıtık yük dengeleme gönderici ya da alıcı başlatmalı olabilir.

- **Hipervizörler** tip 1 (çıplak donanım) ve tip 2 (ana makine üzerinde) olarak ayrılır. Popek–Goldberg koşulunu sağlamayan x86 önce ikili çeviriyle, sonra VT-x/AMD-V ile sanallaştırıldı. **Yarı sanallaştırma** hiperçağrı kullanır. **EPT** gölge tabloların VM çıkışlarını kaldırır, ama TLB ıskası $(n + 1)(m + 1) - 1$ erişime çıkar. **Konteynerler** tek çekirdeği paylaşır.
- Dağıtık sistemde **ara yazılım** belge (Web), dosya (NFS, SMB), nesne (CORBA, gRPC) ya da koordinasyon (Linda, yayınla-abone ol) modeli sunar. Dosya paylaşımında UNIX semantiği pahalıdır; oturum semantiği, değişmez dosyalar ve işlemler ödünleşimlerdir. Bulut bütün bu katmanları hizmet olarak sunar.

12.10 Kavram listesi

Amdahl yasası *Amdahl's law* · simetrik çok işlemci (SMP) *symmetric multiprocessor* · UMA / NUMA *uniform / non-uniform memory access* · çapraz çubuk anahtar *crossbar switch* · omega ağı *omega network* · önbellek tutarlılığı *cache coherence* · dinleme / dizin *snooping / directory* · MESI protokolü *MESI protocol* · yanlış paylaşım *false sharing* · çok çekirdekli *multicore* · eş zamanlı çoklu iş parçacığı (SMT) *simultaneous multithreading* · usta-köle modeli *master-slave model* · spinlock *spinlock* · bilet kilidi *ticket lock* · işlemci afinitesi *processor affinity* · yük dengeleme *load balancing* · alan paylaşımı *space sharing* · çete çizelgeleme *gang scheduling* · çoklu bilgisayar *multicomputer* · hiperküp *hypercube* · ileti geçirme *message passing* · uzak prosedür çağrısı (RPC) *remote procedure call* · dağıtık paylaşımlı bellek *distributed shared memory* · hipervizör *hypervisor* · yarı sanallaştırma *paravirtualization* · sanal makine *virtual machine* · konteyner *container* · ara yazılım *middleware* · demet uzayı *tuple space* · yayınla-abone ol *publish-subscribe*

12.11 Kendini Dene

1. Bir programın %90'ı paralelleştirilebiliyor. 16 çekirdekte hızlanma nedir ve çekirdek başına verim (hızlanma / n) ne olur? Hızlanmanın üst sınırı kaçtır?

   Cevap

   $S(16) = 1 / (0{,}1 + 0{,}9 / 16) = 1 / (0{,}1 + 0{,}05625) = 1 / 0{,}15625 = 6{,}4$. Verim $6{,}4 / 16 = 0{,}4$: çekirdeklerin gücünün ancak %40'ı kullanılır. Üst sınır $1 / 0{,}1 = 10$'dur.

2. 8×8 omega ağında CPU 110, bellek 001'e erişiyor. İstek hangi anahtarlardan, hangi çıkışlarla

geçer? Aynı anda CPU 000 bellek 000'ı okumak isterse çakışma olur mu?

Cevap

110 karıştırmayla 101 numaralı girişe, yani 1C'ye gelir. Hedef 001'in bitleri 0, 0, 1 olduğu için yol 1C (üst) → 2A (üst) → 3A (alt) olur. CPU 000'ın isteği 1A (üst) → 2A (üst) → 3A (üst) yolunu izler. İki istek de 2A'nın üst çıkışını kullanır; biri bekler. Hedef modüller farklı olduğu hâlde ağın içinde çakışma oluşmuştur.

3. Bir NUMA makinesinde yerel erişim 90 ns, uzak erişim 150 ns'dir. Erişimlerin %80'i yerelse ortalama süre nedir? Ortalamanın 100 ns'yi geçmemesi için yerel erişim oranı en az ne olmalıdır?

Cevap

$0{,}8 \cdot 90 + 0{,}2 \cdot 150 = 72 + 30 = 102$ ns. $90y + 150(1 - y) \le 100$ koşulundan $150 - 60y \le 100$, yani $y \ge 50 / 60 \approx 0{,}833$ bulunur. Erişimlerin en az yaklaşık %83,3'ü yerel olmalıdır.

4. 256 düğümlü, düğüm başına 16 MiB ve 64 baytlık satırlı dizin tabanlı sistemde 0x5A0003C4 adresinin satırı hangi düğümün belleğindedir? Blok numarası ve bağıl konum nedir?

Cevap

En soldaki 8 bit 0x5A = 90'dır: satır düğüm 90'dadır. Kalan 24 bit 0x0003C4 = 964'tür. $964 = 15 \cdot 64 + 4$ olduğu için blok 15, bağıl konum 4'tür. İstek düğüm 90'a gider ve onun dizininin 15. girdisine bakılır.

5. Üç çekirdekli bir MESI sisteminde x başlangıçta yalnız bellektedir ve 0'dır. Sırayla Ç0 x'i okur, Ç0 x = 5 yazar, Ç1 okur, Ç2 okur, Ç1 x = 6 yazar. Sonda satırın durumları nedir, kaç veri yolu işlemi oluşmuştur ve bellekte hangi değer vardır?

Cevap

Ç0 okur: E (BusRd). Ç0 yazar: E → M, sessiz. Ç1 okur: BusRd; Ç0 veriyi verir ve belleğe yazar, ikisi S. Ç2 okur: BusRd, üçü de S. Ç1 yazar: BusUpgr; Ç0 ve Ç2 I, Ç1 M. Son durumlar I, M, I'dır. Dört veri yolu işlemi oluşur. Bellekte 5 vardır; güncel değer 6 yalnız Ç1'dedir.

6. Bağlam değiştirme maliyeti C = 2 µs'dir. Bir kilidin beklenen bekleme süresi 3 µs ise ve 6 µs ise hangisi daha ucuzdur: dönmek mi, uyumak mı?

Cevap

Geçişin bedeli 2C = 4 µs'dir. 3 µs < 4 µs olduğu için ilk durumda dönmek ucuzdur. 6 µs > 4 µs olduğu için ikinci durumda uyumak ucuzdur. Bekleme süresi bilinmiyorsa 4 µs dönüp sonra uyumak, en kötü durumda idealin iki katı maliyet verir.

7. 4 CPU'lu bir sistemde dört iş çete çizelgelemeyle çalışacak: A 3, B 2, C 2, D 1 iş parçacığından oluşuyor. Bir tur en az kaç dilim sürer ve CPU kullanımı nedir?

   Cevap

   Toplam 8 iş parçacığı ve dilim başına 4 CPU vardır; en az ⌈8 / 4⌉ = 2 dilim gerekir. Yerleşim: dilim 0'da A (3) ve D (1), dilim 1'de B (2) ve C (2). Her dilim dolu olduğu için kullanım %100'dür.

8. 4 boyutlu hiperküpte e-küp yönlendirmesiyle 0101'den 1010'a giden ileti hangi düğümlerden geçer? Bu yol neden en kısadır?

   Cevap

   $0101 \oplus 1010 = 1111$; dört bit de farklıdır. Bitler soldan sağa düzeltilir: $1101 \rightarrow 1001 \rightarrow 1011 \rightarrow 1010$. Her atlama tek bir farklı biti düzeltir ve hiçbir atlama farkı artırmaz; bu yüzden atlama sayısı Hamming uzaklığına, yani 4'e eşittir. Bu, 4 boyutlu hiperküpün çapıdır.

9. 1 KiB'lık bir paket 4 anahtar, yani 5 bağlantı üzerinden gidiyor. Bağlantılar 1 Gb/s'dir ve her anahtar başlığı 0,1 µs'de işler. Sakla ve ilet ile kesip geçin gecikmeleri nedir?

   Cevap

   Bir bağlantıdan geçiş $1024 \times 8 / 10^9$ s = 8,192 µs'dir. Sakla ve ilette her bağlantıda paketin tamamı beklenir: 5 × 8,192 + 4 × 0,1 = 40,96 + 0,4 = 41,36 µs. Kesip geçte paket bir kez akar ve yalnız başlık gecikmeleri eklenir: 8,192 + 4 × 0,1 ≈ 8,59 µs.

10. 32 bitlik bir misafir işletim sistemi 2 düzeyli sayfa tabloları kullanıyor; ana makinenin EPT'si 4 düzeylidir. TLB ıskasında ve hiçbir girdi önbellekte değilken en kötü durumda kaç bellek erişimi gerekir (son veri erişimi hariç)?

    Cevap

    Misafir tablosunun her düzeyi için girdinin GPA'sı 4 EPT erişimiyle çevrilir ve girdi 1 erişimle okunur: 2 × 5 = 10. Verinin GPA'sı da 4 erişimle çevrilir. Toplam $10 + 4 = 14 = (2 + 1)(4 + 1) - 1$.

12.12 İleri okuma

- A. S. Tanenbaum ve H. Bos, *Modern Operating Systems*, Bölüm 8 “Multiple Processor Systems” ve Bölüm 7 “Virtualization and the Cloud”. Kitap, bu nottaki omega ağı, dizin, çete çizelgeleme ve yük dengeleme örneklerinin özgün kaynağıdır.

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, "Multiprocessor Scheduling", "Locks" ve "Distributed Systems" bölümleri: önbellek afinitesini, kilit gerçeklemelerini ve RPC'yi kısa ve anlaşılır biçimde anlatır; kitap ostep.org'da ücretsizdir.
- J. L. Hennessy ve D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Bölüm 5: dinleme ve dizin tabanlı tutarlılık protokollerini, bellek tutarlılık modellerini ve senkronizasyon komutlarını donanım açısından ayrıntılı inceler.
- J. M. Mellor-Crummey ve M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM TOCS*, 1991: bilet ve MCS kilitlerini ölçümlerle karşılaştıran klasik makaledir.

12.13 Kaynaklar

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 8: Multiple Processor Systems; Bölüm 7: Virtualization and the Cloud.
2. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 5.5 (Multi-Processor Scheduling), 18 (Virtual Machines), 19 (Networks and Distributed Systems).
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Multiprocessor Scheduling", "Locks", "Distributed Systems", "Sun's NFS", "AFS", Ek: "Virtual Machine Monitors".
4. J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6. baskı, Morgan Kaufmann, 2017 — Bölüm 5: Thread-Level Parallelism (tutarlılık, MESI, dizin protokolleri).
5. J. M. Mellor-Crummey, M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM TOCS*, 1991; A. D. Birrell, B. J. Nelson, "Implementing Remote Procedure Calls", *ACM TOCS*, 1984; G. J. Popek, R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures", *CACM*, 1974.
6. Linux çekirdek belgeleri: "CFS/EEVDF Scheduler", "Scheduler Domains", "NUMA Memory Policy", "Core Scheduling"; *Intel 64 and IA-32 Architectures Software Developer's Manual*, Cilt 3 (VMX, EPT).

KISIM IV · BÖLÜM 13

Güvenlik

İşletim sistemi bir bilgisayardaki bütün kaynakların bekçisidir. Hangi sürecin hangi dosyayı okuyacağına, hangi kullanıcının sisteme gireceğine, hangi kodun çekirdek kipinde çalışacağına o karar verir. Bu kararlar yanlış verilirse en iyi çizelgeleyici ya da en hızlı dosya sistemi de bir işe yaramaz: veriler sızar, değiştirilir ya da kullanılamaz hâle gelir. Not, önce güvenliğin hedeflerini ve tehditlerini tanımlar. Ardından işletim sisteminin kullandığı kriptografi araçlarını anlatır: şifreleme, özet, imza, sertifika ve TLS. Koruma kısmı erişim matrisini, erişim denetim listelerini, yetenekleri ve en az ayrıcalık ilkesini ele alır. Biçimsel modeller ve gizli kanallar bunu izler. Son kesimler kimlik doğrulamayı, bellek güvenliği açıklarını, kötü amaçlı yazılımları ve bugünkü savunmaları işler: korumalı alan, konteyner, zorunlu erişim denetimi, güvenli önyükleme.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum13`

ÖĞRENME ÇIKTILARI

Hedefler ve tehditler
Gizlilik, bütünlük ve erişilebilirlik hedeflerini; saldırı türlerini ve saldırgan profillerini ayırt etmek.

Kriptografi
Simetrik ve asimetrik şifrelemeyi, özet fonksiyonlarını, dijital imzayı, sertifikaları ve TLS el sıkışmasını açıklamak; Diffie-Hellman değişimini adım adım hesaplamak.

Koruma
Erişim matrisini ACL ve yetenek biçiminde okumak; en az ayrıcalığı, setuid'i ve Linux yeteneklerini uygulamak.

Biçimsel modeller
Bell-LaPadula ve Biba kurallarını uygulamak; gizli kanalların neden kapatılamadığını açıklamak.

Kimlik doğrulama
Tuzlu yavaş parola özetlerini, MFA'yı, TOTP'yi ve FIDO2 geçiş anahtarlarını karşılaştırmak.

Saldırılar ve savunmalar
Bellek güvenliği hatalarını, TOCTOU'yu ve kötü amaçlı yazılımları; kanarya, NX, ASLR, CFI, korumalı alan ve güvenli önyüklemeyi tanımak.

13.1 Güvenlik hedefleri ve tehditler

Bir sistemi korumaya başlamadan önce neyin, kime karşı ve hangi amaçla korunduğunu söylemek gerekir. Bu kesim güvenliğin üç temel hedefini, bu hedefleri çiğneyen olay türlerini ve saldırganları tanıtır. Sonunda da savunmanın genel mimarisi gelir: katmanlı savunma ve güvenilir bilgi işleme tabanı.

13.1.1 Güvenlik ve koruma

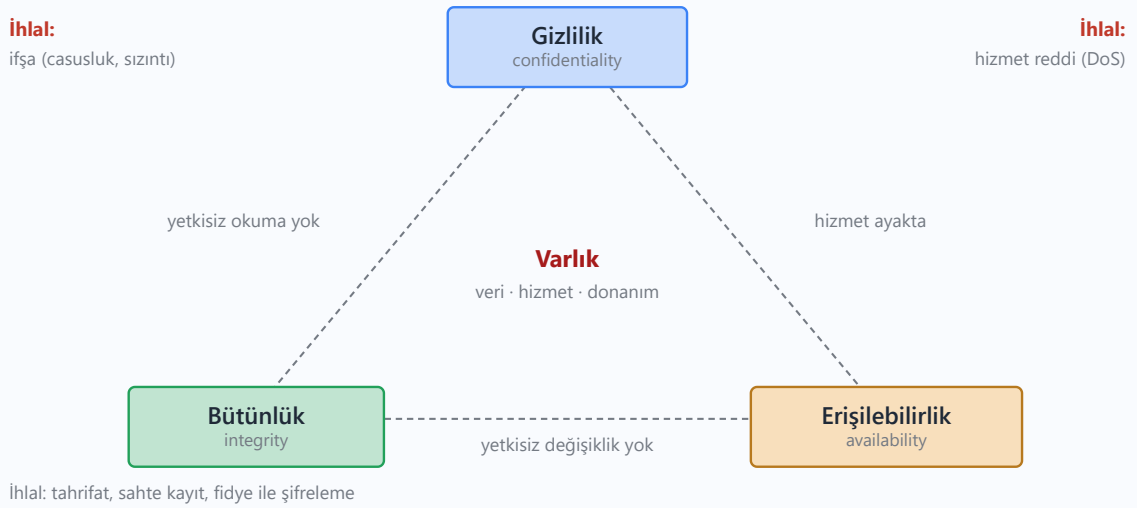
İşletim sistemleri kitaplarında iki yakın kavram ayrı ayrı anılır. **Koruma** (*protection*), sistemin içindeki mekanizmalardır: hangi sürecin hangi nesneye hangi hakla erişebileceğini belirleyen ve bunu denetleyen kurallar. Dosya izin bitleri, sayfa tablosundaki salt okunur bit, çekirdek ve kullanıcı kipi ayrımı birer koruma mekanizmasıdır. **Güvenlik** (*security*) ise daha geniş bir sorudur. Sistem, içeriden ve dışarıdan gelen kasıtlı saldırılara ve kazalara karşı ne kadar dayanıklıdır? Koruma mekanizmaları kusursuz çalışsa bile parola çalınırsa, yönetici kandırılırsa ya da yedek alınmamışsa sistem güvenli değildir.

Bu ayrımın pratik bir sonucu vardır. Koruma, işletim sisteminin doğrudan gerçeklediği ve doğruluğu kanıtlanabilen bir alt sistemdir. Güvenlik ise insanları, ağı, donanımı ve yazılımın bütün hatalarını kapsar; hiçbir zaman tam olarak sağlanamaz, yalnız saldırının maliyeti yükseltilebilir. Bu not ikisini birlikte ele alır. Önce hedefler ve tehditler, sonra koruma mekanizmaları gelir. En sonda bu mekanizmaları aşmaya çalışan saldırılar ve onlara karşı geliştirilen savunmalar anlatılır.

13.1.2 CIA üçlüsü

Güvenlik hedefleri genellikle üç sözcükle özetlenir. İngilizce baş harflerinden ötürü bu üçlüye **CIA üçlüsü** denir (Şekil 13.1).

> **Tanım 13.1**
>
> **Gizlilik** (*confidentiality*), verinin yalnız yetkili kişilerce okunabilmesidir. **Bütünlük** (*integrity*), verinin yalnız yetkili kişilerce ve yetkili yollarla değiştirilebilmesidir. **Erişilebilirlik** (*availability*), meşru kullanıcıların veriye ve hizmete ihtiyaç duydukları anda ulaşabilmesidir.



Şekil 13.1: CIA üçlüsü. Korunan varlık veri, hizmet ya da donanım olabilir; her hedefin ihlali ayrı bir zarar türüdür.

Her hedefin karşısında bir ihlal türü durur. Gizliliğin ihlali *ifşadır*: bir şirketin müşteri veritabanının sızdırılması, bir casus yazılımın e-postaları kopyalaması. Bütünlüğün ihlali *tahrifattır*: bir öğrencinin not kaydını değiştirmesi, bir saldırganın sistem programının yerine kendi sürümünü koyması. Erişilebilirliğin ihlali *hizmet reddidir*: bir web sunucusunun sahte isteklerle boğulması ya da bir fidye yazılımının diski şifreleyip kullanılamaz hâle getirmesi.

Üç hedef birbirinden bağımsız değildir ve kimi zaman çatışır. Verinin üç ayrı ülkede kopyasını tutmak erişilebilirliği artırır, ama sızıntı riski olan yerlerin sayısını da üçe çıkarır. Bir disk şifrelendiğinde gizlilik artar; anahtar kaybolursa veri sonsuza dek erişilemez olur. Güvenlik tasarımı bu yüzden hedefleri dengeleme işidir.

Üçlüye sık eklenen iki hedef daha vardır. **Özgünlük** (*authenticity*), bir iletinin ya da kullanıcının gerçekten iddia ettiği kaynaktan geldiğinin doğrulanabilmesidir. **Hesap verebilirlik** (*accountability*) ise bir işlemi kimin yaptığının sonradan kanıtlanabilmesidir. Hesap verebilirliğin iki aracı vardır: denetim kayıtları ve inkâr edilemezlik. İnkâr edilemezlikte (*non-repudiation*) bir kişi imzaladığı bir belgeyi sonradan “ben imzalamadım” diyerek reddedemez. Dijital imzanın asıl kazancı budur.

Örnek 13.1

Bir hastanenin dosya sunucusundaki kayıtlarla ilgili dört olayı ele alalım. Her biri hangi hedefi çiğner?

1. **Bir çalışan, merak ettiği bir ünlünün tahlil sonuçlarını açıp okuyor.** Veri değişmedi, hizmet aksamadı; ama yetkisiz biri okudu. Gizlilik ihlalidir.
2. **Bir saldırgan bir hastanın kan grubunu değiştiriyor.** Kimse fark etmeden okunan veri artık yanlıştır. Bütünlük ihlalidir ve gizlilik ihlalinden de tehlikeli olabilir.

3. **Fidye yazılımı bütün dosyaları şifreliyor; doktorlar saatlerce kayıtlara ulaşamıyor.** Veri dışarı çıkmadı ve içeriği bozulmadı; ama kullanılamıyor. Erişilebilirlik ihlalidir.
4. **Bir doktor, yazdığı reçeteyi sonradan yazmadığını iddia ediyor ve kayıtlarda bunu çürütecek bir iz yok.** Hesap verebilirlik eksiktir. Reçeteler doktorun özel anahtarıyla imzalansaydı inkâr mümkün olmazdı.

13.1.3 Tehdit, zafiyet, saldırı ve istismar

Güvenlik dilinde dört sözcük birbirine karıştırılır, ama her biri zincirin ayrı bir halkasıdır (Şekil 13.2).

Tanım 13.2

Tehdit (*threat*) olası bir zarardır. **Zafiyet** (*vulnerability*) bu zararı mümkün kılan zayıflıktır. **Saldırı** (*attack*) zafiyeti kullanma girişimidir. **İstismar** (*exploit*) ise bu girişimi yapan kod ya da yöntemdir.



Şekil 13.2: Tehditten ihlale giden zincir. Her halkanın kendi savunması vardır: yama zafiyeti, kimlik doğrulama saldırıyı, kanarya ve NX istismarı, yedek de ihlalin sonucunu sınırlar.

Bir örnekle zincir somutlaşır. “Müşteri verilerinin çalınması” bir tehdittir. Web uygulamasındaki denetlenmemiş bir girdi alanı zafiyettir. Saldırganın bu alana özel hazırlanmış bir sorgu yazması saldırıdır. Bu sorgunun kendisi, yani zafiyeti tetikleyen dizgi ya da program, istismardır. Sonunda veritabanı dışarı aktarılırsa gizlilik ihlal edilmiş olur.

Zincir görüşünün değeri, savunmanın da her halkaya yerleştirilebilmesidir. Yama zafiyeti ortadan kaldırır. Güçlü kimlik doğrulama saldırganın sisteme girmesini zorlaştırır. Yığın kanaryası ve yürütülemez sayfalar gibi önlemler istismarın çalışmasını engeller. Yedekler ise her şey başarısız olsa bile ihlalin sonucunu sınırlar. Bir halkadaki savunma kırılsa bile sonraki halkalar devrede kalır.

İhlaller sunumdaki gibi yedi kategoride toplanabilir (Tablo 13.1). İlk üçü CIA hedeflerinin doğrudan karşılığıdır. Sonraki dördü bu hedefleri dolaylı yoldan bozar.

Kategori	Tanım	Örnek
Gizlilik ihlali	Verinin izinsiz okunması	Kredi kartı veritabanının sızdırılması
Bütünlük ihlali	Verinin izinsiz değiştirilmesi	Not sisteminde kayıt tahrifatı
Erişilebilirlik ihlali	Verinin ya da hizmetin yok edilmesi	Fidye yazılımının diski şifrelemesi
Hizmet hırsızlığı	Kaynakların izinsiz kullanımı	Ele geçirilen sunucuda kripto para kazma

Kategori	Tanım	Örnek
Hizmet reddi (DoS/DDoS)	Meşru kullanımın engellenmesi	Botnet ile SYN seli
Kimlik değiştirme	Başkası gibi davranma	Oltalama ile çalınan parola
Ayrıcalık yükseltme	Hak edilenden fazla yetki kazanma	setuid programdaki hatadan root olma

Tablo 13.1: İhlal kategorileri.

Hizmet reddi (*denial of service*, DoS) saldırısında amaç veriyi okumak ya da değiştirmek değildir. Saldırgan sistemi meşru isteklere yanıt veremeyecek kadar meşgul eder. Bir TCP sunucusu her bağlantı isteği için yarı açık bir bağlantı kaydı tutar. Saldırgan saniyede binlerce sahte istek (SYN) gönderir ve el sıkışmayı hiç tamamlamaz. Bu tablo dolar ve gerçek kullanıcılar bağlanamaz. Saldırı binlerce ele geçirilmiş makineden aynı anda yapılırsa adı **dağıtık hizmet reddi** (DDoS) olur. Bu durumda tek bir kaynak adresi engellemek de işe yaramaz.

Kimlik değiştirme (*masquerading*) saldırısında saldırgan başka bir kullanıcının ya da makinenin kimliğine bürünür. **Ayrıcalık yükseltme** (*privilege escalation*) ise sıradan bir kullanıcının ya da sürecin, bir hatadan yararlanarak yönetici haklarına ulaşmasıdır. İşletim sistemi açısından en tehlikeli kategori budur, çünkü yönetici haklarıyla bütün öteki korumalar devre dışı bırakılabilir.

13.1.4 Ağ saldırıları

Bugün hemen her bilgisayar ağa bağlıdır ve işletim sisteminin dışarıyla konuştuğu kanal da saldırı yüzeyinin parçasıdır. Güvenlik kitaplarında bu saldırılar geleneksel olarak üç kişiyle anlatılır. Alice ve Bob konuşmak isteyen iki taraftır. Eve (*eavesdropper*) yalnız dinleyen pasif saldırgandır. Mallory ise iletileri değiştirebilen etkin saldırgandır. Kanal şifresiz ve kimlik doğrulamasızsa dört saldırı da mümkündür.

Dinleme (*eavesdropping*)
: Pasif saldırgan trafiğin bir kopyasını alır ve okur. İletilere dokunmadığı için fark edilmesi çok zordur. Gizliliği bozar. Çaresi şifrelemedir: bugün bunu çoğunlukla TLS yapar.

Tekrar saldırısı (*replay*)
: Saldırgan geçerli bir iletiyi yakalar ve sonradan aynen yeniden gönderir. İleti şifreli ve imzalı olabilir: saldırganın onu okuması ya da değiştirmesi gerekmez. “1000 TL öde” iletisi iki kez işlenirse zarar iki katına çıkar. Çaresi her iletiye bir kez kullanılabilen bir değer eklemektir: tek kullanımlık sayı (*nonce*), zaman damgası ya da sıra numarası.

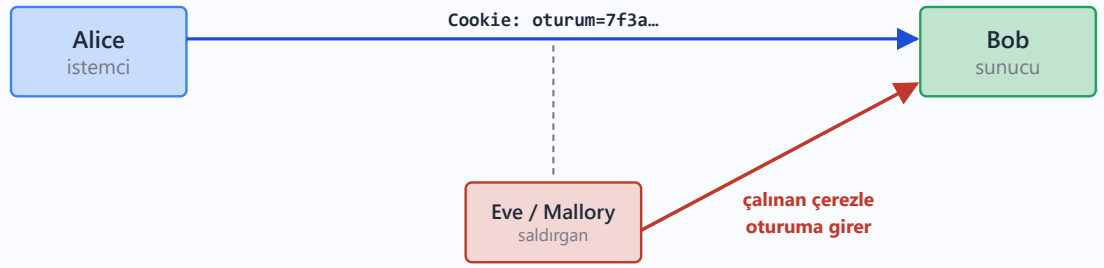
Ortadaki adam (*man-in-the-middle*, MITM)
: Saldırgan iki tarafın arasına girer ve her birine öteki gibi görünür. Alice, Bob'la konuştuğunu sanır; aslında Mallory ile konuşur. Mallory iletileri okur, değiştirir ve iletir. Şifreleme tek başına çare değildir: Alice'in şifreli kanal kurduğu taraf Mallory'nin kendisi olabilir. Çaresi karşı tarafın kimliğini doğrulamaktır, yani sertifikalardır.

Oturum kaçırma (*session hijacking*)
: Kullanıcı bir kez kimlik doğruladıktan sonra sunucu ona bir oturum belirteci verir; çoğunlukla bir çerez. Saldırgan bu belirteci çalarsa parolayı bilmeden oturuma girer. Çaresi belirteci yalnız

şifreli kanaldan göndermek (`Secure`), betiklerin okumasını engellemek (`HttpOnly`) ve ömrünü kısa tutmaktır.



Şekil 13.3: Ortadaki adam: Mallory iki tarafa da karşı taraf gibi görünür; iletileri okur ve değiştirir.



Şekil 13.4: Oturum kaçırma: kimlik doğrulamadan sonra verilen çerez çalınırsa saldırgan parolasız oturuma girer.

Sık yapılan hata

Şifrelemenin tekrar saldırısını önlediğini düşünmek. Saldırgan şifreli iletiyi çözmeden de yeniden gönderebilir. Alıcı aynı iletiyi daha önce gördüğünü anlamıyorsa ikinci kopyayı da geçerli sayar. Tekrarı yalnız iletiye eklenen tek kullanımlık bir değer ve alıcının bu değerleri izlemesi önler.

13.1.5 Saldırganlar ve kazayla veri kaybı

Savunmanın gücü, karşısındaki saldırganın kim olduğuna göre ayarlanır. Bir öğrenci laboratuvarındaki makineyi meraklı bir arkadaşa karşı korumakla bir bankanın sunucusunu devlet destekli bir gruba karşı korumak aynı iş değildir. Saldırganlar kabaca dört gruba ayrılır.

Meraklı içeriden kişi
: Teknik bilgisi azdır; başkasının e-postasına ya da dosyasına "bakmak" ister. Çoğu zaman bir izin hatasından yararlanır. Davranışı etik dışıdır ve çoğu zaman yasaya da aykırıdır. Doğru dosya izinleri bu gruba karşı çoğunlukla yeterlidir.

Kötü niyetli çalışan
: Sistemde meşru bir hesabı ve yetkisi vardır; bu yetkiyi kötüye kullanır. Veri sızdırabilir, sabotaj yapabilir ya da işten çıkarılırsa tetiklenecek bir mantık bombası bırakabilir. Dışarıdaki saldırganlara karşı kurulan duvarların içindedir; bu yüzden fark edilmesi en zor gruptur.

Siber suçlu
: Amacı kârdır: fidye yazılımı, kimlik ve kart bilgisi hırsızlığı, ele geçirdiği makinelerden oluşan ağı (botnet) kiralamak. Bugün en yaygın tehdit budur. Bu grup çoğunlukla bilinen ve yaması çıkmış açıkları kullanır; yama yapılmamış sistemleri otomatik araçlarla tarar.

Devlet destekli gruplar
: Literatürde "gelişmiş kalıcı tehdit" (*advanced persistent threat*, APT) olarak anılırlar. Amaçları askerî ya da ticari casusluk ve kritik altyapı sabotajıdır. Yıllarca sessizce içeride kalabilirler. Henüz kimsenin bilmediği açıkları (*zero-day*, sıfırıncı gün açıkları) ve tedarik zinciri saldırılarını kullanırlar. 2010'da ortaya çıkan Stuxnet, uranyum zenginleştirme santrifüjlerini hedef alan böyle bir yazılımdı.

Veri kaybının tek nedeni saldırganlar değildir. Kazalar, saldırılardan çok daha sık veri kaybettirir (Tablo 13.2). Bir disk arızalanır, bir yönetici yanlış diski biçimlendirir, bir dizüstü bilgisayar trende unutulur. Tanenbaum'un alaycı örneğiyle, yedek bantlarını kemiren fareler bile bu listeye girer.

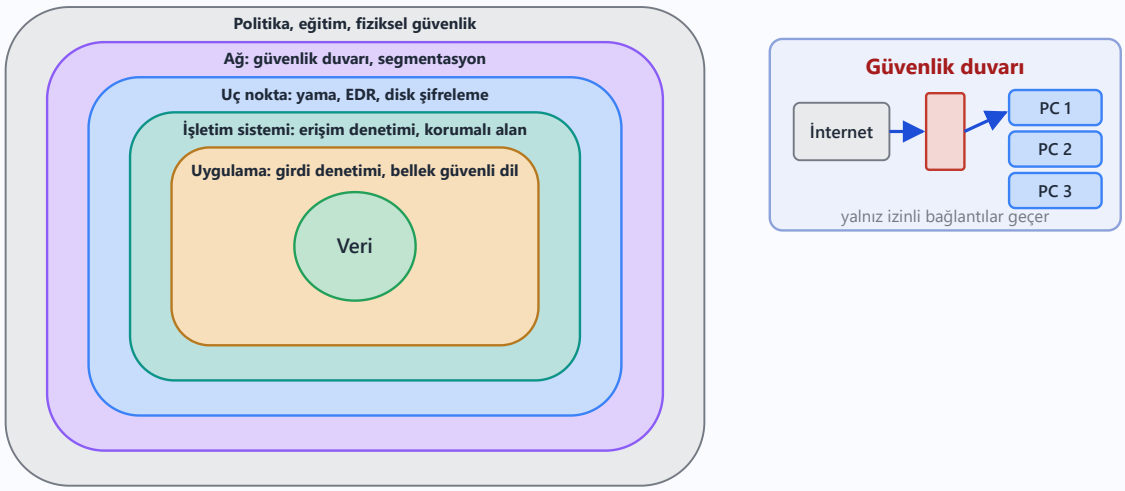
Neden	Örnek	Önlem
Doğal olaylar	yangın, sel, deprem	coğrafi olarak ayrı yedek
Donanım ve yazılım hatası	disk arızası, CPU hatası, yazılım hatası	RAID, sağlama toplamı, sürüm tutma
İnsan hatası	yanlış komut, yanlış diski biçimleme, kayıp aygıt	3-2-1 yedekleme, şifreli disk

Tablo 13.2: Kazayla veri kaybının nedenleri ve önlemleri.

Yaygın yedekleme kuralı **3-2-1** diye anılır: verinin üç kopyası, iki farklı ortamda tutulur ve bir kopya başka bir yerde durur. Bugün buna bir koşul eklenir: uzaktaki kopyanın değiştirilemez (*immutable*) olması. Fidye yazılımları ağdan erişebildikleri yedekleri de şifreler ya da siler. Değiştirilemez bir kopya bu saldırıdan sağ çıkar. Böylece aynı kural hem kazaya hem saldırıya karşı işe yarar.

13.1.6 Derinlemesine savunma

Hiçbir savunma kusursuz değildir. Güvenlik duvarının bir kuralı yanlış yazılır, bir çalışan oltalama e-postasına tıklar, bir kütüphanede yeni bir açık bulunur. Bu yüzden tek bir savunmaya güvenilmez; savunmalar iç içe katmanlar hâlinde dizilir. Bu ilkeye **derinlemesine savunma** (*defense in depth*) denir. Ortaçağ kalelerinin hendek, sur, iç sur ve kale burcundan oluşan düzenine benzer: saldırgan bir katmanı aşsa bile bir sonrakine takılır (Şekil 13.5).



Şekil 13.5: Derinlemesine savunmanın katmanları (solda) ve bir ağın önündeki güvenlik duvarı (sağda).

Katmanlar dıştan içe şöyle sıralanır. En dışta politikalar, kullanıcı eğitimi ve fiziksel güvenlik vardır: sunucu odasının kilidi bu katmandadır. Ağ katmanında **güvenlik duvarı** (*firewall*) yalnız izinli bağlantıları geçirir; ağ bölütlemesi de bir bölümün ele geçirilmesinin ötekilere yayılmasını önler. Uç nokta katmanında yamalar, uç nokta tespit ve yanıt araçları (EDR) ve disk şifreleme bulunur. İşletim sistemi katmanı erişim denetimini ve korumalı alanı sağlar. Uygulama katmanında girdi denetimi ve bellek güvenli diller yer alır. En içte de korunan varlık, yani veri durur.

İyi uygulamalar da bu katmanlara dağılır. Güvenlik güncellemeleri hızla uygulanır, çünkü saldırıların çoğu yaması çoktan çıkmış açıkları kullanır. Güçlü kimlik doğrulama ve en az ayrıcalıkla çalışan erişim denetimi temel kuraldır. Günlük kayıtları tutulur ve izlenir; aksi hâlde bir ihlal aylarca fark edilmeyebilir. Düzenli sızma testleriyle savunmanın zayıf noktaları önceden aranır. Kullanıcılar oltalamaya karşı eğitilir.

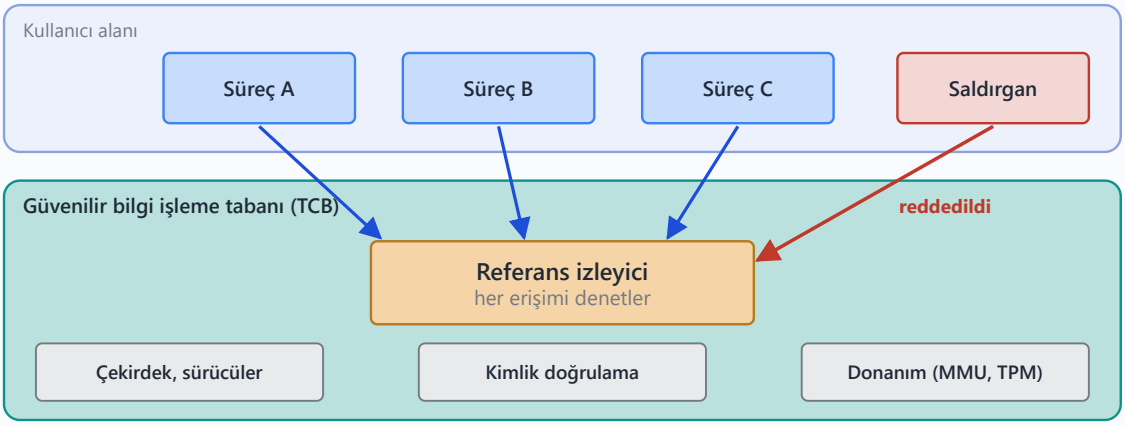
Son yıllarda yaygınlaşan **sıfır güven** (*zero trust*) yaklaşımı derinlemesine savunmanın bir adım ötesidir. Geleneksel düzende güvenlik duvarının içi güvenilir sayılırdı: içerideki bir makine öteki makinelere serbestçe erişebilirdi. Sıfır güven bu varsayımı reddeder. Ağın içinden gelen istek de dışarıdan gelen gibi kimlik doğrulamasından ve yetki denetiminden geçer. Böylece içeri sızan bir saldırganın yanal hareketi zorlaşır.

13.1.7 Güvenilir bilgi işleme tabanı

Her güvenlik iddiası bazı bileşenlerin doğru çalıştığı varsayımına dayanır. Dosya izinleri ancak onları denetleyen çekirdek kodu hatasızsa bir anlam taşır. Bu bileşenlerin toplamına güvenilir bilgi işleme tabanı denir (Şekil 13.6).

Tanım 13.3

Güvenilir bilgi işleme tabanı (*trusted computing base*, TCB), sistemin güvenlik politikasını uygulamaktan sorumlu bütün donanım, bellenim ve yazılım bileşenlerinin kümesidir. TCB'deki bir hata bütün sistemin güvenliğini bozabilir; TCB dışındaki bir hata ise politikayı çiğneyemez.



Şekil 13.6: Kullanıcı süreçlerinin bütün erişimleri TCB'nin içindeki referans izleyiciden geçer. Saldırganın isteği de oradan geçer ve reddedilir.

TCB'nin kalbi **referans izleyicidir** (*reference monitor*). Bu bileşen her erişim isteğini alır, politikaya bakar ve izin ya da ret kararı verir. Bir referans izleyicinin güvenilir olması için üç koşul sağlanmalıdır. Birincisi, her erişimde çağrılmalıdır; atlanabilen bir denetim, hiç denetim olmaması kadar kötüdür. İkincisi, kurcalanamaz olmalıdır; kullanıcı süreçleri onun kodunu ya da verisini değiştirememelidir. Üçüncüsü, doğruluğu denetlenebilecek kadar küçük olmalıdır.

Bu üçüncü koşul işletim sistemi tasarımında önemli bir sonuç doğurur. TCB ne kadar büyükse içinde hata bulunma olasılığı o kadar yüksektir. Linux çekirdeği on milyonlarca satırdır ve aygıt sürücüleri de dahil olmak üzere hepsi çekirdek kipinde, yani TCB'nin içinde çalışır. Mikro çekirdekler bu yüzden ilgi görür: dosya sistemini ve sürücüleri kullanıcı kipine taşıyarak TCB'yi küçültürler. seL4 mikro çekirdeği daha da ileri gider. Yaklaşık on bin satırlık kodunun, belirtimine uygun çalıştığı biçimsel olarak kanıtlanmıştır.

Sık yapılan hata

TCB'yi "güvenilir olduğu kanıtlanmış bileşenler" diye anlamak. Tanım tersini söyler: TCB, güvenilmek *zorunda* olunan bileşenlerdir. Bir yazıcı sürücüsü çekirdek kipinde çalışıyorsa, ne kadar özensiz yazılmış olursa olsun TCB'nin parçasıdır. Güvenli tasarımın amacı bu kümeyi olabildiğince küçük tutmaktır.

13.2 Kriptografi

İşletim sistemi kriptografiyi her yerde kullanır: parolaları saklarken, diski şifrelerken, güncelleme paketlerinin imzasını doğrularken, ağ bağlantılarını korurken, önyükleme zincirini denetlerken. Bu kesim, bu kullanımları anlamaya yetecek temel araçları tanıtır. Amaç şifre tasarlamak değildir. Her aracın neyi sağladığını ve neyi sağlamadığını bilmek yeterlidir.

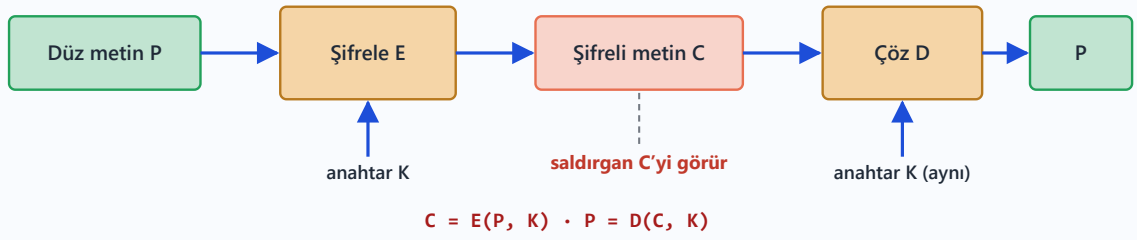
13.2.1 Şifrelemenin temelleri

Şifrelemenin amacı, okunabilir bir iletiyi anahtarı bilmeyen birinin anlayamayacağı bir biçime dönüştürmektir. Okunabilir ileti **düz metin** (*plaintext*, P), dönüşmüş biçimi **şifreli metin** (*ciphertext*, C) diye anılır. Dönüşümü yapan fonksiyon şifreleme E, geri dönüşümü yapan fonksiyon çözme D'dir. İkisi de bir anahtar K kullanır (Şekil 13.7).

Tanım 13.4

Şifreleme (*encryption*), veriyi anahtar olmadan okunamaz hâle getirmektir. **Simetrik** ya da **gizli anahtarlı şifrelemede** şifreleme ve çözme aynı anahtarı kullanır:

> $C = E(P, K), \quad P = D(C, K)$



Şekil 13.7: Simetrik şifreleme modeli. Saldırgan şifreli metni görebilir; aynı K'yı bilmeden düz metni elde edemez.

Modelde saldırganın kanalı dinleyebildiği ve C'yi görebildiği varsayılır. Peki saldırgan şifreleme algoritmasını da biliyor mu? 1883'te Auguste Kerckhoffs'un ortaya koyduğu ilkeye göre bilmesi varsayılmalıdır. **Kerckhoffs ilkesi**, bir şifreleme sisteminin güvenliğinin yalnız anahtarın gizliliğine dayanması gerektiğini söyler. Algoritma açık olabilir, hatta açık olmalıdır. Gizli tutulan bir algoritma er geç tersine mühendislikle ortaya çıkar. Açık bir algoritma ise yıllarca dünyanın her yerinden kriptografların saldırısına uğrar; ayakta kalırsa güvenilir sayılır. Bugünkü bütün standart şifreler bu yüzden herkese açıktır.

Kerckhoffs ilkesinin bir sonucu, anahtar uzayının büyüklüğünün önemidir. Algoritmayı bilen bir saldırgan her anahtarı tek tek deneyebilir. Bu yönteme **kaba kuvvet** (*brute force*) saldırısı denir. Eski DES standardı 56 bitlik anahtar kullanırdı. Bu 2^{56} anahtar 1998'de özel bir donanımla birkaç günde taranabildi. 128 bitlik bir anahtar uzayı ise bütün dünyanın hesaplama gücüyle bile taranamaz. Ancak büyük bir anahtar uzayı tek başına yeterli değildir; bir sonraki örnek bunu gösterir.

13.2.2 Yerine koyma şifresi ve sıklık analizi

En eski şifrelerden biri **tek alfabeli yerine koyma şifresidir** (*monoalphabetic substitution*). Alfabenin her harfi sabit bir başka harfle değiştirilir. Anahtar, 26 harfin bir permütasyonudur. Sunumdaki anahtar, klavyenin üst sırasından başlayarak harfleri dizer: A → Q, B → W, C → E ve böyle sürer (Tablo 13.3).

açık	A	B	C	D	E	F	G	H	I	J	K	L	M
şifreli	Q	W	E	R	T	Y	U	I	O	P	A	S	D
açık	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
şifreli	F	G	H	J	K	L	Z	X	C	V	B	N	M

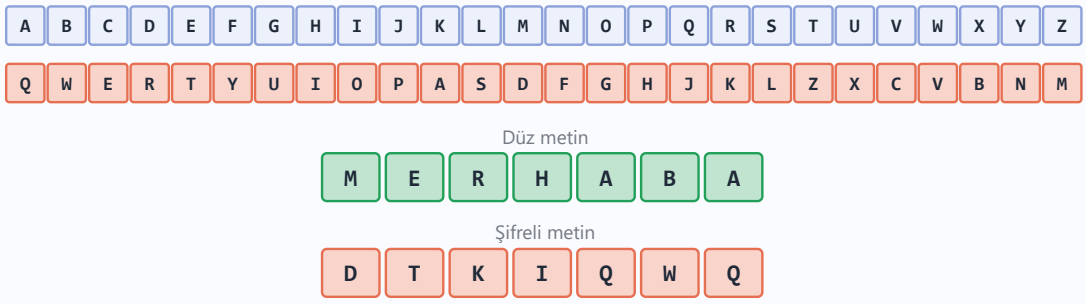
Tablo 13.3: Yerine koyma şifresinin anahtarı.

Örnek 13.2

MERHABA sözcüğünü bu anahtarla şifreleyelim.

1. M'nin altındaki harf D'dir: M → D.
2. E → T, R → K, H → I.
3. A → Q, B → W, A → Q.

Sonuç: MERHABA → DTKIQWQ. Çözmek için tablo ters yönde okunur: şifreli metindeki D'nin üstünde M vardır. Sözcükteki üç A'nın üçü de Q olmuştur.



Şekil 13.8: MERHABA'nın şifrelenmesi. Aynı düz harf hep aynı şifreli harfe gider.

Bu şifrenin anahtar uzayı çok büyüktür: 26 harfin permütasyon sayısı $26! \approx 4 \times 10^{26}$'dır, yani yaklaşık 2^{88}. Saniyede bir milyar anahtar deneyen bir bilgisayar bu uzayı milyarlarca yılda tarayabilir. Buna karşın şifre birkaç satırlık bir metinle, kalem kâğıtla kırılır. Neden?

Örnekteki üç A'nın üçünün de Q olması zayıflığı açığa vurur. Şifre harfleri değiştirir, ama harf sıklıklarını değiştirmez. Türkçe bir metinde en sık harfler A, E ve İ'dir. Şifreli metinde en sık geçen harf büyük olasılıkla A'nın karşılığıdır. Ardından sık geçen iki harfli ve üç harfli kalıplar ("ve", "bir", "ler") eşlenir. Birkaç tahminden sonra anahtar adım adım ortaya çıkar. Bu yönteme **sıklık analizi** denir. Yöntemi ilk kez dokuzuncu yüzyılda Arap matematikçi el-Kindî anlatmıştır.

> **Sık yapılan hata**
>
> Büyük anahtar uzayını güvenlikle eşitlemek. Anahtar uzayı yalnız kaba kuvvete karşı bir alt sınırdır. Şifrenin yapısı düz metnin istatistiğini sızdırıyorsa saldırgan anahtarları tek tek denemez, doğrudan yapıyı kullanır. Yerine koyma şifresinde 2^{88}'lik uzay bu yüzden hiçbir işe yaramaz.

13.2.3 Bugünkü simetrik şifreler

Bugünkü simetrik şifreler sıklık bilgisini gizlemek için iki yapıdan birini kullanır. **Blok şifreleri** veriyi sabit boylu bloklara böler ve her bloğu birçok tur boyunca karıştırır. Standart blok şifresi AES'tir (*Advanced Encryption Standard*, 2001). 128 bitlik bloklar ve 128, 192 ya da 256 bitlik anahtarlar kullanır. Bugünkü işlemcilerde AES için özel komutlar (AES-NI) bulunur ve şifreleme saniyede gigabaytlar hızına ulaşır. **Akış şifreleri** ise anahtardan uzun bir rastgele görünümlü bayt dizisi üretir ve veriyle XOR'lar. ChaCha20 bu türün yaygın bir örneğidir; AES donanımı bulunmayan telefonlarda hızlıdır.

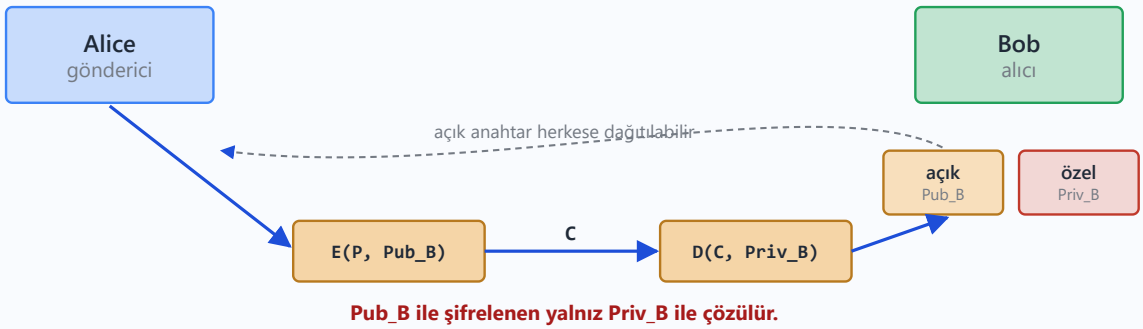
Bir şifrenin yalnız gizliliği sağlaması yetmez. Saldırgan şifreli metnin bitlerini değiştirirse alıcı bozuk bir düz metin elde eder ve bunu fark etmeyebilir. Bu yüzden bugün şifreleme her zaman bir bütünlük denetimiyle birlikte kullanılır. Bu birleşik yapıya **kimlik doğrulamalı şifreleme** (*authenticated encryption*) denir. AES-256-GCM ve ChaCha20-Poly1305 bu türün iki standart örneğidir: şifreli metnin yanına kısa bir doğrulama etiketi eklerler. Tek bir bit değişse etiket tutmaz ve alıcı iletiyi reddeder.

13.2.4 Açık anahtarlı şifreleme

Simetrik şifrelemenin bir temel sorunu vardır: iki tarafın önceden aynı anahtarı paylaşmış olması gerekir. İnternette ilk kez bağlandığımız bir siteyle bu anahtarı nasıl paylaşırız? Anahtarı ağdan göndermek, dinleyen herkese vermek demektir. 1976'da Diffie ve Hellman bu soruya yeni bir cevap verdi: iki ayrı anahtar.

Tanım 13.5

Açık anahtarlı (*public-key*) ya da **asimetrik şifrelemede** her kullanıcının bir anahtar çifti vardır. **Açık anahtar** herkese dağıtılır; **özel anahtarı** yalnız kullanıcının kendisi bilir. Açık anahtarla şifrelenen bir ileti yalnız eşi olan özel anahtarla çözülebilir.



Şekil 13.9: Alice, Bob'un açık anahtarı Pub_B ile şifreler; iletiyi yalnız Priv_B'yi bilen Bob çözebilir.

Alice, Bob'a gizli bir ileti göndermek istediğinde Bob'un açık anahtarını alır ve iletiyi onunla şifreler (Şekil 13.9). Şifreli metni dinleyen Eve de Bob'un açık anahtarını bilir, ama bu anahtarla iletiyi çözemez. Çözme yalnız özel anahtarla mümkündür ve o anahtar Bob'un makinesinden hiç çıkmaz. Böylece önceden paylaşılmış bir sır gerekmez.

Bu düzenin matematik temeli **tek yönlü tuzak kapılı fonksiyonlardır**. Böyle bir fonksiyonu bir yönde hesaplamak kolaydır, ters yönde hesaplamak çok zordur. Ancak gizli bir bilgi (tuzak kapı) bilinirse ters yön de kolaylaşır. RSA'da (1977) bu fonksiyon büyük asal sayıların çarpımıdır: iki 1024 bitlik asalı çarpmak bir anlıktır, ama çarpımı çarpanlarına ayırmanın bilinen hızlı bir yolu yoktur. Eliptik eğri kriptografisinde (ECC) zorluk, eğri üzerindeki ayrık logaritma problemidir.

Örnek 13.3

RSA'nın işleyişini oyuncak boyutlu sayılarla izleyelim. Gerçek sistemlerde sayılar yüzlerce basamaklıdır; adımlar aynıdır.

1. **Anahtar üretimi.** Bob iki asal seçer: $p = 3$, $q = 11$. Çarpım $n = 33$'tür. $\varphi = (p-1)(q-1) = 2 \times 10 = 20$. Açık üs $e = 3$ seçilir (20 ile ortak böleni yoktur). Özel üs d, $e \cdot d \equiv 1 \pmod{20}$ koşulunu sağlayan sayıdır: $3 \times 7 = 21 \equiv 1$, yani $d = 7$. Açık anahtar $(n, e) = (33, 3)$, özel anahtar $d = 7$'dir.
2. **Şifreleme.** Alice $m = 4$ iletisini gönderir: $c = m^e \bmod n = 4^3 \bmod 33 = 64 \bmod 33 = 31$.
3. **Çözme.** Bob $c^d \bmod n$ hesaplar: $31^7 \bmod 33 = 4$. Özgün ileti geri gelir.

Eve n = 33'ü ve e = 3'ü bilir. d'yi bulmak için φ'yi, yani p ve q'yu bilmesi gerekir. 33'ü çarpanlarına ayırmak kolaydır. 2048 bitlik bir n'yi ayırmak ise bugünkü bilgisayarların yapabileceği bir iş değildir. Değerler Python ile doğrulandı.

Asimetrik şifrelemenin bedeli hızdır: simetrik şifrelemeden yüz ile bin kat yavaştır (Tablo 13.4). Bu yüzden gerçek sistemler ikisini birlikte kullanır. Asimetrik yöntem yalnız kısa bir işi yapar: iki taraf arasında bir oturum anahtarı kurar. Asıl veri bu oturum anahtarıyla simetrik olarak şifrelenir. TLS, SSH ve disk şifreleme bu karma düzeni kullanır.

	Simetrik	**Asimetrik (açık anahtarlı)**
Anahtar	Tek, paylaşılan	Açık + özel çift
Hız	Çok hızlı (AES-NI)	100–1000 kat yavaş
Kullanım	Toplu veri	Anahtar değişimi, imza
Örnek	AES, ChaCha20	RSA, ECDH/X25519, Ed25519, ML-KEM

Tablo 13.4: Simetrik ve asimetrik şifrelemenin karşılaştırılması.

Tablodaki son algoritma yeni bir tehdide yanıttır. Yeterince büyük bir kuantum bilgisayarı, Shor algoritmasıyla hem çarpanlara ayırmayı hem ayrık logaritmayı hızla çözebilir. Böyle bir makine bugün yoktur. Ancak bugün kaydedilen trafik yıllar sonra çözülebilir; bu yüzden geçiş şimdiden başlamıştır. ABD standart kurumu NIST, 2024'te kafes tabanlı ML-KEM algoritmasını kuantuma dayanıklı anahtar kurma standardı olarak yayımladı (FIPS 203).

13.2.5 Özet fonksiyonları

Kriptografinin üçüncü temel aracı şifreleme değildir; hiçbir anahtar kullanmaz ve geri çevrilemez.

Tanım 13.6

Kriptografik özet fonksiyonu (*cryptographic hash function*), her boydaki girdiden sabit boylu bir çıktı, yani **özet** üretir ve şu özellikleri taşır:

- **Tek yönlülük:** bir özetten onu veren girdiyi bulmak pratikte imkânsızdır.
- **Çakışma direnci:** aynı özeti veren iki farklı girdi bulmak pratikte imkânsızdır.
- **Çığ etkisi:** girdideki tek bir bitlik değişiklik, özetin bitlerinin yaklaşık yarısını değiştirir.

Bugünkü standartlar SHA-256 ve SHA-3'tür. SHA-256 her girdiden 256 bitlik, yani 64 onaltılık haneli bir özet üretir. Çığ etkisini görmek için iki sözcüğün özetini karşılaştıralım.

Örnek 13.4

"MERHABA" ile "MERHABa" yalnız son harfte ayrılır. ASCII'de A (0x41) ile a (0x61) tek bir bitte farklıdır. SHA-256 özetleri şöyledir:

```
MERHABA  cc44cc878588056ef8376b36fc7a2ba5124656c38597eb31291704e0af986112
MERHABa  57821b9db50b174af36bc659771a47e0eaa6aef8144b1a9e5b803c93e3280b18
```

İki özet arasında hiçbir benzerlik yoktur. Bit bit karşılaştırıldığında 256 bitin 124'ü farklıdır; bu, beklenen yarıya (128) çok yakındır. Özetler Python'un `hashlib` kütüphanesiyle hesaplandı.

Özet fonksiyonları işletim sisteminde birçok yerde karşımıza çıkar. İndirilen bir dosyanın özeti yayımlanan özetle karşılaştırılarak dosyanın bozulmadığı anlaşılır. Parolalar açık hâlde değil, özet olarak saklanır. Bütünlük denetimi yapan araçlar sistem dosyalarının özetlerini saklar ve değişeni bulur. Git her nesneyi özetiyle adlandırır.

Bir özet fonksiyonunun kırılması, çakışma bulmanın pratik hâle gelmesi demektir. MD5'te çakışmalar 2004'te, SHA-1'de 2017'de pratik olarak üretildi. İkisi de güvenlik amacıyla artık kullanılmamalıdır.

Sık yapılan hata

Özeti bir sıkıştırma ya da tek başına bir kanıt saymak. Özetten girdi geri elde edilemez; bu yüzden sıkıştırma değildir. İmzasız bir özet de bütünlüğü kanıtlamaz: belgeyi değiştiren saldırgan, yanındaki özeti de yeniden hesaplayıp değiştirebilir. Özetin güvenilir bir kanaldan gelmesi ya da imzalanması gerekir.

13.2.6 Dijital imza

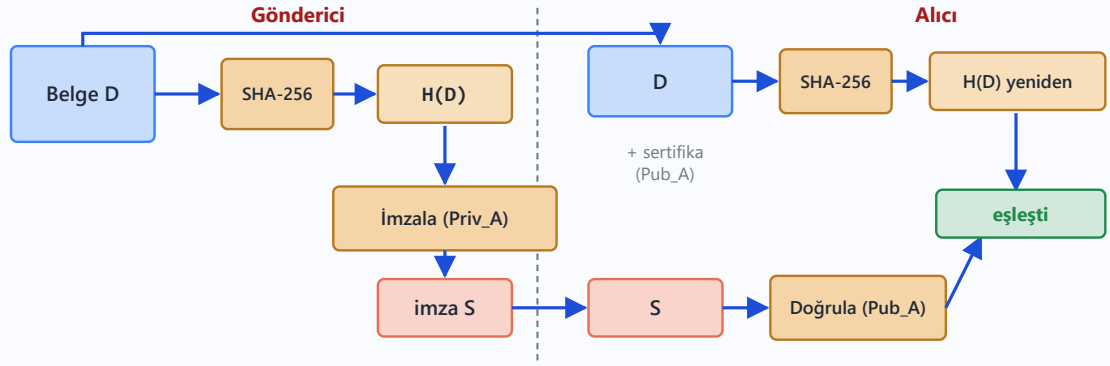
Açık anahtarlı şifreleme ters yönde de kullanılabilir. Özel anahtarla yapılan bir işlemi herkes açık anahtarla denetleyebilir; ama işlemi yalnız özel anahtarın sahibi yapabilir. Dijital imza bu gözleme dayanır.

Tanım 13.7

Dijital imza (*digital signature*), bir belgenin özetinin göndericinin özel anahtarıyla işlenmesiyle elde edilen değerdir. Herkes göndericinin açık anahtarıyla imzayı doğrulayabilir. Geçerli bir imza iki şeyi kanıtlar: belge değişmemiştir (bütünlük) ve özel anahtarın sahibinden gelmiştir (özgünlük, inkâr edilemezlik).

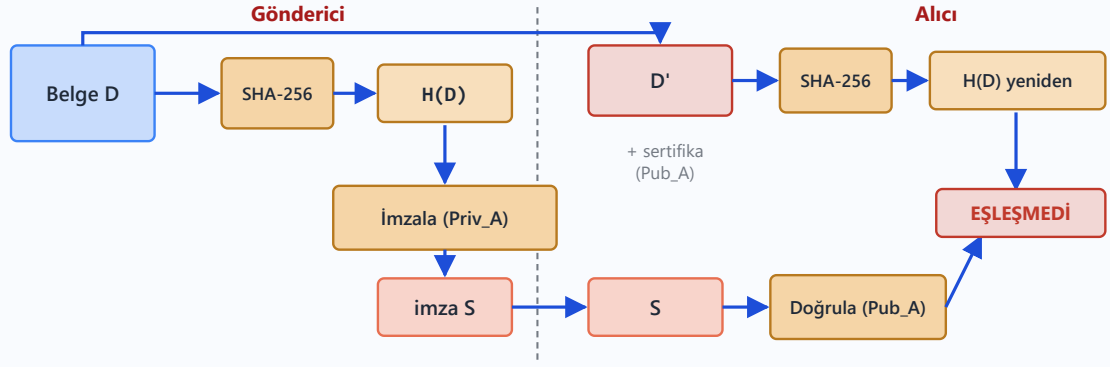
İmzalama ve doğrulama şu adımlarla yürür (Şekil 13.10).

1. **Özet.** Gönderici Alice, D belgesinin özetini hesaplar: H(D). Belgenin kendisini değil özetini imzalamak hem hızlıdır hem de sabit boylu bir girdiyle çalışır.
2. **İmza.** Alice özeti özel anahtarı Priv_A ile imzalar ve S imzasını elde eder. Bugünkü algoritmalar RSA-PSS, ECDSA ve Ed25519'dur.
3. **Gönderme.** Belge, imza ve Alice'in açık anahtarını taşıyan sertifika birlikte gönderilir. Gizlilik istenmiyorsa belge şifrelenmez.
4. **Doğrulama.** Alıcı Bob, belgenin özetini kendisi yeniden hesaplar ve imzayı Pub_A ile doğrular. Doğrulama, imzanın tam bu özete Priv_A ile atıldığını söylüyorsa belge bozulmamıştır ve Alice'ten gelmiştir.



Şekil 13.10: Dijital imzanın akışı: gönderici özeti özel anahtarıyla imzalar, alıcı özeti yeniden hesaplar ve imzayı açık anahtarla doğrular.

Yolda belgenin tek bir baytı değişirse ne olur? Bob değişmiş D′ belgesinin özetini hesaplar. Çığ etkisi yüzünden bu özet H(D)'den tamamen farklıdır ve imza onunla eşleşmez (Şekil 13.11). Saldırgan yeni bir geçerli imza da üretemez, çünkü Priv_A onda yoktur.



Şekil 13.11: Belge yolda D′'ye dönüşürse yeniden hesaplanan özet imzayla eşleşmez ve doğrulama başarısız olur.

Örnek 13.5

Oyuncak RSA anahtarıyla (n = 33, e = 3, d = 7) imzalamayı izleyelim. Belgenin özeti h = 4 olsun.

1. Bob özel anahtarıyla imzalar: $s = h^d \bmod n = 4^7 \bmod 33 = 16$.
2. Alıcı açık anahtarla doğrular: $s^e \bmod n = 16^3 \bmod 33 = 4$. Sonuç kendi hesapladığı özete eşittir; imza geçerlidir.
3. Saldırgan belgeyi değiştirir ve özet 5 olur. $16^3 \bmod 33$ yine 4'tür ve 5'e eşit değildir; doğrulama başarısız olur. Yeni özete uyan imzayı üretmek için d = 7'yi bilmek gerekir.

İmza işlemi, şifrelemeyle karıştırılmaya çok yatkındır. İki işlem anahtarları ters yönde kullanır. Şifrelemede gönderici *alıcının açık* anahtarını kullanır ve yalnız alıcı çözebilir. İmzada gönderici *kendi özel* anahtarını kullanır ve herkes doğrulayabilir. Paylaşılan gizli bir anahtarla üretilen doğrulama kodu (MAC) bütünlüğü sağlar, ama inkâr edilemezliği sağlamaz: anahtarı alıcı da bildiği için aynı kodu alıcı da üretebilirdi.

13.2.7 Sertifikalar ve açık anahtar altyapısı

Açık anahtarlı şifreleme bir soruyu açık bırakır. Bob, elindeki açık anahtarın gerçekten Alice'e ait olduğunu nereden bilir? Mallory kendi açık anahtarını "Alice'in anahtarı" diye dağıtırsa, Alice'e gönderilen bütün iletileri o okur. Ortadaki adam saldırısı tam olarak budur. Çözüm, anahtarı kimliğe bağlayan ve güvenilir bir üçüncü tarafın imzaladığı bir belgedir.

Tanım 13.8

Sertifika (X.509 biçiminde), bir açık anahtarı sahibinin kimliğine, örneğin bir alan adına bağlayan belgedir. Belgede geçerlilik süresi ve bunları onaylayan bir **sertifika otoritesinin** (*certificate authority*, CA) imzası da bulunur. Sertifikaları üreten, dağıtan ve geri alan kurum ve kuralların bütününe **açık anahtar altyapısı** (*public key infrastructure*, PKI) denir.



Şekil 13.12: Sertifika zinciri: kök CA ara CA'yı, ara CA sunucu sertifikasını imzalar. Sağda tarayıcının doğrulama adımları.

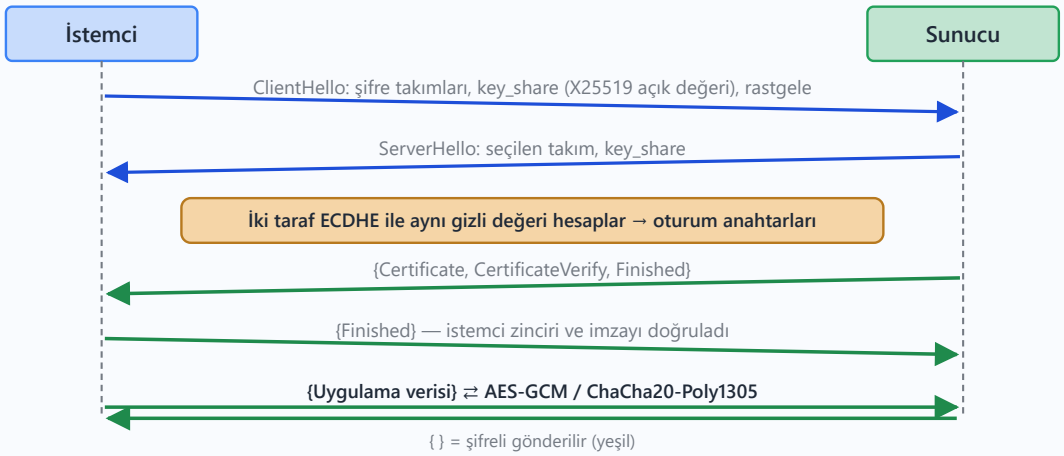
Sertifikalar bir zincir oluşturur (Şekil 13.12). Zincirin başında **kök CA** vardır. Kök sertifika öz imzalıdır; onu kimse imzalamaz. Ona duyulan güven, işletim sisteminin ya da tarayıcının *güven deposunda* önceden yüklü olmasından gelir. Kök CA'lar anahtarlarını çevrim dışı tutar ve günlük işi ara CA'lara bırakır. Ara CA'nın imzaladığı sunucu sertifikası ise örneğin `www.ornek.edu.tr` alan adını sunucunun açık anahtarına bağlar.

Tarayıcı bir HTTPS sitesine bağlandığında şu denetimleri yapar. Sunucu kendi sertifikasını ve ara CA sertifikasını gönderir. Tarayıcı imzaları zincir boyunca doğrular ve zincirin güven deposundaki bir köke ulaştığını denetler. Sertifikadaki alan adının bağlanılan adla eşleştiğine ve sürenin dolmadığına bakar. Sertifikanın geri alınıp alınmadığını (OCSP ya da CRL ile) ve herkese açık sertifika şeffaflığı (CT) kayıtlarında bulunup bulunmadığını denetler. Hepsi tamamsa bağlantı güvenilir sayılır. Mallory'nin sahte sertifikası ise güvenilir bir CA'nın imzasını taşımadığı için zincir kurulamaz ve tarayıcı uyarı verir.

Sertifika dünyası son on yılda hızla değişti. Let's Encrypt, ACME protokolüyle sertifika almayı ücretsiz ve otomatik hâle getirdi; sertifikaları 90 günlüktür. Kısa ömür, çalınan bir anahtarın zararını sınırlar ve yenilemeyi otomasyona zorlar. Tarayıcı üreticileri ile CA'ların ortak kurulu, sertifika ömrünün 2029'a kadar kademeli olarak 47 güne indirilmesini kararlaştırdı.

13.2.8 TLS 1.3 el sıkışması

Önceki araçların hepsi tek bir protokolde birleşir: TLS (*Transport Layer Security*). HTTPS, güvenli e-posta ve birçok başka protokol TLS'nin üzerinde çalışır. Güncel sürüm TLS 1.3'tür (RFC 8446, 2018). Bağlantının başındaki **el sıkışması** üç işi birlikte yapar: ortak bir oturum anahtarı kurar, sunucunun kimliğini doğrular ve el sıkışmasının değiştirilmediğini onaylar (Şekil 13.13).



Şekil 13.13: TLS 1.3 el sıkışması (basitleştirilmiş). Süslü ayraç içindeki iletiler şifreli gider.

1. **ClientHello.** İstemci desteklediği şifre takımlarını, bir rastgele değeri ve geçici (*ephemeral*) bir Diffie-Hellman açık değerini gönderir. Bugün bu değer çoğunlukla X25519 eğrisindendir.
2. **ServerHello.** Sunucu bir şifre takımı seçer ve kendi Diffie-Hellman açık değerini gönderir. Buraya kadarki her şey açık gider.
3. **Ortak sır.** İki taraf da karşı tarafın açık değeri ile kendi gizli değerinden aynı ortak sırrı hesaplar. Bu sır ağda hiç dolaşmaz. Oturum anahtarları ondan türetilir ve bundan sonraki her ileti şifrelidir.
4. **Sunucunun kimliği.** Sunucu sertifikasını gönderir. Ardından el sıkışmasının o ana kadarki özetini özel anahtarıyla imzalar (CertificateVerify) ve Finished iletisini ekler. Ortadaki bir saldırgan bu imzayı üretemez, çünkü sunucunun özel anahtarı onda yoktur.
5. **İstemcinin onayı.** İstemci sertifika zincirini ve imzayı doğrular. Kendi Finished iletisiyle el sıkışmasının değiştirilmediğini onaylar. Bütün işlem tek bir gidiş-dönüşte (1-RTT) biter.
6. **Uygulama verisi.** Veri artık simetrik, kimlik doğrulamalı şifreyle akar: AES-GCM ya da ChaCha20-Poly1305.

Geçici Diffie-Hellman değerleri önemli bir özellik kazandırır: **ileriye dönük gizlilik** (*forward secrecy*). Her oturumun anahtarı o oturum için üretilen geçici değerlerden türer ve oturum bitince bu değerler silinir. Sunucunun uzun ömürlü özel anahtarı yalnız imza için kullanılır. Bu anahtar yıllar sonra çalınsa bile, saldırganın kaydettiği eski trafik çözülemez.

13.2.9 Kod izleme: Diffie-Hellman anahtar değişimi

TLS'nin kalbindeki Diffie-Hellman değişimi küçük sayılarla elle izlenebilir. İki taraf önceden iki açık sayı üzerinde anlaşır: bir asal p ve bir taban g. Her taraf gizli bir üs seçer, tabanı bu üsse yükseltir ve sonucu açıkça gönderir. Sonra karşı tarafın açık değerini kendi gizli üssüne yükseltir. Üs alma işleminin yer değiştirme özelliği sayesinde iki taraf aynı sayıya ulaşır:

$$A = g^a \bmod p, \quad B = g^b \bmod p, \quad s = B^a = A^b = g^{ab} \bmod p$$

Sunumdaki kod, büyük üsleri verimli hesaplayan `modpow` fonksiyonunu ve değişimin dört hesabını içerir.

```
long modpow(long g, long e, long p) {
    long r = 1;
    while (e > 0) {
        if (e & 1) r = r * g % p;
        g = g * g % p;
        e >>= 1;
    }
    return r;
}
int main(void) {
    /* açık: p = 23, g = 5 */
    long A  = modpow(5, 6, 23);
    long B  = modpow(5, 15, 23);
    long sA = modpow(B, 6, 23);
    long sB = modpow(A, 15, 23);
}
```

`modpow` "kare al ve çarp" yöntemini kullanır. Üs `e`'nin bitleri sağdan sola taranır. Her turda bit 1 ise sonuç `r`, o anki taban `g` ile çarpılır. Sonra taban kendisiyle çarpılarak bir sonraki bitin ağırlığına getirilir: g, g^2, g^4, g^8 … Her çarpmadan sonra mod p alındığı için sayılar hiç büyümez. Üs n bitse döngü n tur döner. 5^{15} için düz yöntem 14 çarpma isterken bu yöntem 4 turda biter. 2048 bitlik üslerde fark, mümkün ile imkânsız arasındaki farktır.

Alice gizli a = 6'yı seçmiştir. A = modpow(5, 6, 23) hesabı Tablo 13.5 gibi ilerler. 6'nın ikili gösterimi 110'dur.

Tur	e & 1	r	g (sonra)	e (sonra)	Açıklama
1	0	1	5 · 5 mod 23 = 2	3	bit 0: r değişmez
2	1	1 · 2 = 2	2 · 2 = 4	1	bit 1: r = r · g
3	1	2 · 4 = 8	4 · 4 = 16	0	bit 2: r = r · g; döngü biter

Tablo 13.5: `modpow(5, 6, 23)` izlemesi. Başlangıçta r = 1, g = 5, e = 6. Değerler Python ile yeniden hesaplandı.

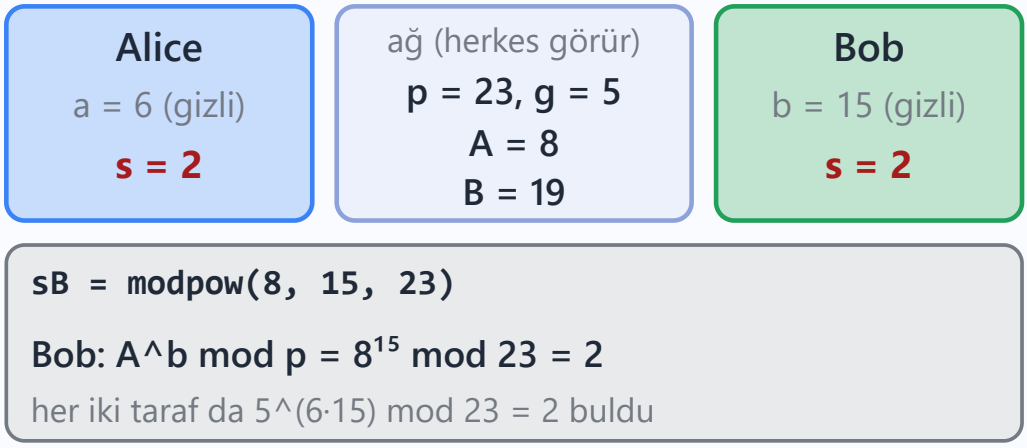
Fonksiyon 8 döndürür: A = 8. Bu değer ağa açık gönderilir; a = 6 ise Alice'in makinesinden hiç çıkmaz. Bob da gizli b = 15 ile aynı fonksiyonu çalıştırır (Tablo 13.6). 15'in ikili gösterimi 1111'dir; dört bitin hepsi 1 olduğu için her turda çarpma yapılır.

Tur	e & 1	r	g (sonra)	e (sonra)
1	1	1 · 5 = 5	25 mod 23 = 2	7
2	1	5 · 2 = 10	4	3
3	1	10 · 4 = 40 mod 23 = 17	16	1

Tur	e & 1	r	g (sonra)	e (sonra)
4	1	17 · 16 = 272 mod 23 = 19	256 mod 23 = 3	0

Tablo 13.6: `modpow(5, 15, 23)` izlemesi. Başlangıçta r = 1, g = 5, e = 15.

B = 19 ağdan Alice'e gider. Şimdi iki taraf karşı tarafın açık değerini kendi gizli üssüne yükseltir. Alice sA = modpow(19, 6, 23) hesaplar: turlar r = 1, 16, 2 değerlerinden geçer ve sonuç 2 olur. Bob sB = modpow(8, 15, 23) hesaplar: r = 8, 6, 12, 2 ve sonuç yine 2'dir. İki taraf da $5^{6 \cdot 15} \bmod 23 = 5^{90} \bmod 23$ değerini hesaplamıştır.



Şekil 13.14: Değişimin sonu: Alice ve Bob ortak sır s = 2'ye ulaştı; ağdan yalnız p, g, A ve B geçti.

Ağı dinleyen Eve p = 23, g = 5, A = 8 ve B = 19'u görür. Ortak sırrı bulmak için a'yı ya da b'yi bilmesi gerekir. a'yı bulmak, $5^a \bmod 23 = 8$ denklemini çözmektir. Buna **ayrık logaritma problemi** denir. Bu küçük örnekte Eve 22 olasılığı deneyerek a = 6'yı hemen bulur. Gerçekte p 2048 bit uzunluğundadır ya da X25519 eğrisi kullanılır. Bilinen hiçbir algoritma bu boyda bir problemi makul sürede çözemez.

Sık yapılan hata

Diffie-Hellman'ın tek başına ortadaki adam saldırısını önlediğini sanmak. Değişim yalnız “ağı dinleyen” Eve'e karşı güvenlidir. Mallory araya girip Alice ile bir, Bob ile ayrı bir değişim yaparsa iki tarafla da ayrı birer ortak sır kurar ve bütün trafiği okur. Bu yüzden TLS'de sunucu, el sıkışmasını sertifikasındaki anahtarla imzalar. Kimlik doğrulamasız Diffie-Hellman güvenli değildir.

13.3 Koruma

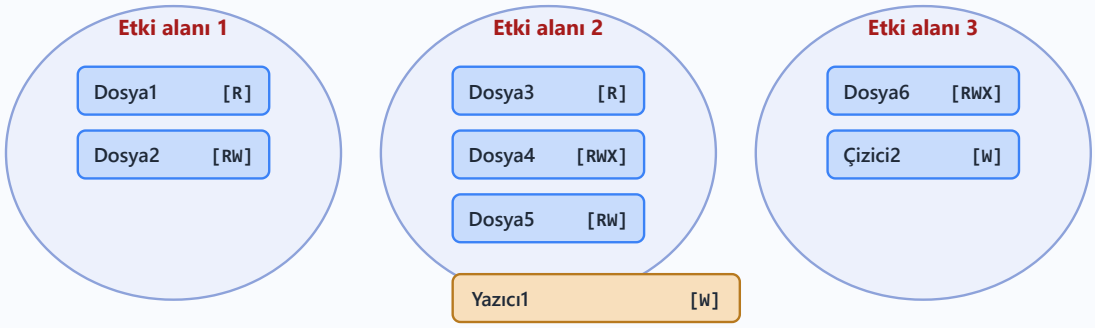
Kriptografi verinin sistem dışında, ağda ya da diskte korunmasını sağlar. Sistemin içinde ise başka bir soru sorulur: çalışan bir süreç hangi nesnelere hangi haklarla erişebilir? Bu kesim bu sorunun modelini (koruma matrisi), modelin iki uygulama biçimini (erişim denetim listeleri ve yetenekler) ve hakları en aza indirme yollarını anlatır.

13.3.1 Koruma etki alanları

Bir bilgisayar sisteminde korunması gereken **nesneler** vardır: dosyalar, bellek kesimleri, yazıcılar, semaforlar, süreçler. Her nesnenin kendine özgü işlemleri bulunur: bir dosya okunur, yazılır, çalıştırılır; bir yazıcıya yalnız yazılır. Bir süreç belirli bir anda bu nesnelerin bir kısmına belirli haklarla erişebilir. Bu (nesne, hak) çiftlerinin kümesine etki alanı denir (Şekil 13.15).

Tanım 13.9

Koruma etki alanı (*protection domain*), (nesne, haklar) çiftlerinden oluşan bir kümedir. Her süreç her an bir etki alanında çalışır ve yalnız o alanın izin verdiği işlemleri yapabilir.



Şekil 13.15: Üç koruma etki alanı. Yazıcı1 hem ikinci hem üçüncü alanda bulunur: nesneler etki alanları arasında paylaşılabilir.

Şekil 13.15'te birinci etki alanı Dosya1'i okuyabilir, Dosya2'yi okuyup yazabilir. İkinci alan Dosya3'ü okur, Dosya4 üzerinde bütün hakları kullanır, Dosya5'i okuyup yazar ve Yazıcı1'e yazar. Üçüncü alanda Dosya6 (RWX), Çizici2 (W) ve yine Yazıcı1 (W) vardır. Etki alanları ayrık olmak zorunda değildir: aynı nesne birden çok alanda, hatta farklı haklarla bulunabilir.

UNIX ve Linux'ta bir etki alanı, kullanıcı kimliği (UID) ile grup kimliklerinin (GID) kümesiyle belirlenir. Aynı UID ve GID'lerle çalışan iki süreç aynı nesnelere aynı haklarla erişir. Bunun yanında çekirdek kipi ve kullanıcı kipi de iki ayrı etki alanıdır. Çekirdek kipinde çalışan kod bütün belleğe ve bütün aygıtlara erişebilir.

Süreçler çalışırken etki alanı değiştirebilir. Bir sistem çağrısı süreci kullanıcı kipinden çekirdek kipine geçirir; çağrı bitince süreç eski alanına döner. Bir süreç `setuid` bitli bir programı çalıştırdığında ise programın sahibinin etki alanına geçer.

13.3.2 Koruma matrisi

Bütün etki alanlarının bütün haklarını tek bir tabloda göstermek mümkündür. Lampson'ın 1971'de önerdiği bu tabloya koruma matrisi ya da erişim matrisi denir.

> **Tanım 13.10**
>
> **Koruma matrisinde** (*access matrix*) her satır bir etki alanını, her sütun bir nesneyi gösterir. Satır i ile sütun j'nin kesiştiği hücre, i etki alanındaki bir sürecin j nesnesi üzerindeki haklarını içerir.

	Dosya1	Dosya2	Dosya3	Dosya4	Dosya5	Yazıcı1	Dosya6	Çizici2	D1	D2	D3
D1	R	RW								Gir	
D2			R	RWX	RW	W					
D3						W	RWX	W			

Tablo 13.7: Etki alanlarının da nesne olarak yer aldığı koruma matrisi. “Gir” hakkı başka bir etki alanına geçmeye izin verir.

Tablo 13.7 matrise ilginç bir özellik ekler: etki alanları da birer nesnedir. D1 satırının D2 sütunundaki **Gir** (*enter*) hakkı, D1'de çalışan bir sürecin D2'ye geçebileceğini söyler. Böylece etki alanı değiştirme de matrisin denetimi altına girer. Aynı yolla matrisin kendisini değiştirme hakları da modellenebilir: bir hakkı başkasına kopyalama, bir nesnenin sahibi olarak haklarını düzenleme gibi.

Matrisin her durumu güvenlik politikasıyla karşılaştırılabilir. Politikaya göre Eve'in bir dosyayı hiçbir zaman okuyamaması gerekiyorsa, Eve'in satırında o dosyanın hücresinde R görülmemelidir. Bir dizi meşru komuttan sonra o hücrede R belirirse sistem **yetkili** durumdan **yetkisiz** duruma geçmiş demektir. Doğal bir soru akla gelir: bir başlangıç durumundan ve izinli komutlardan, yetkisiz bir duruma varılıp varılamayacağı önceden hesaplanabilir mi? Harrison, Ruzzo ve Ullman 1976'da bu sorunun genel durumda **karar verilemez** olduğunu gösterdi. “Bir hak sızabilir mi?” sorusunu her matris için yanıtlayan bir algoritma yoktur. Bu sonuç, kuralları baştan kısıtlayan Bell-LaPadula ve Biba gibi modellerin neden geliştirildiğini açıklar.

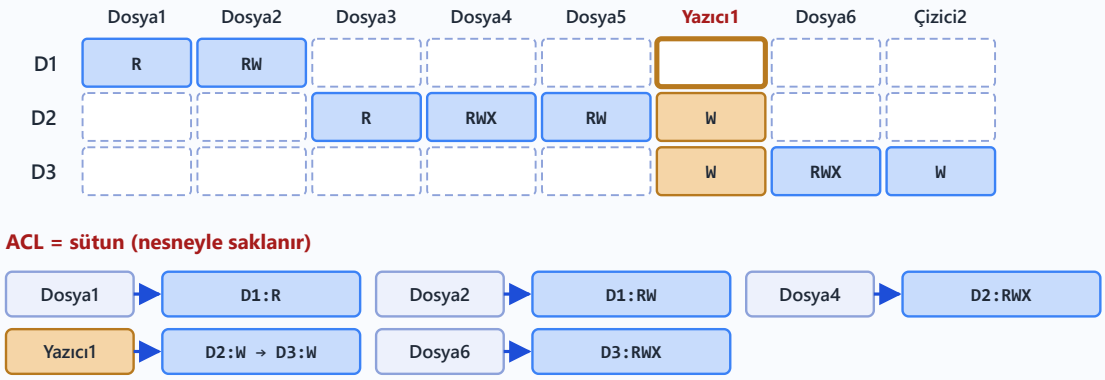
Matrisin uygulamadaki sorunu boyutudur. Binlerce kullanıcı ve milyonlarca dosya bulunan bir sistemde matrisin milyarlarca hücresi olur; bunların çok büyük çoğunluğu boştur. Matris çok **seyrektir**. Tablo 13.7'deki küçük örnekte bile, yalnız sekiz nesne sütunu ele alındığında 24 hücreden 15'i boştur. Bu yüzden matrisin tamamı hiçbir zaman saklanmaz. Ya sütunlar ya da satırlar saklanır ve boş hücreler atlanır.

13.3.3 Aynı matrisin iki biçimi: ACL ve yetenek

Matrisi sütun sütun saklamak, her nesnenin yanına “bu nesneye kimler hangi haklarla erişebilir?” listesini koymak demektir. Satır satır saklamak ise her etki alanına “bu alan hangi nesnelere hangi haklarla erişebilir?” listesini vermektir (Şekil 13.16 ve Şekil 13.17).

Tanım 13.11

Erişim denetim listesi (*access control list*, ACL), matrisin bir sütunudur. Bir nesneye hangi etki alanlarının hangi haklarla erişebileceğini sıralar ve nesneyle birlikte saklanır. **Yetenek listesi** (*capability list*, C-listesi) ise matrisin bir satırıdır: bir etki alanının erişebildiği nesneleri ve haklarını sıralar. Listedeki her girdiye **yetenek** (*capability*) denir.



Şekil 13.16: Matrisin boş olmayan sütunları birer ACL olur. Örneğin Yazıcı1'in ACL'si D2:W ve D3:W'dir.



Şekil 13.17: Aynı matrisin bir satırı: D2'nin yetenek listesi Dosya3:R, Dosya4:RWX, Dosya5:RW ve Yazıcı1:W'dir.

ACL düzeninde bir süreç bir nesneye erişmek istediğinde çekirdek nesnenin ACL'sini açar ve sürecin kimliğini bu listede arar. Yetenek düzeninde ise süreç erişirken elindeki yeteneği gösterir. Çekirdeğin yalnız yeteneğin geçerli olduğunu denetlemesi yeterlidir; bir ad çözümlemesi ya da liste taraması gerekmez.

İki gösterim aynı bilgiyi taşır, ama farklı soruları kolay yanıtlar (Tablo 13.8).

Soru	ACL (sütun)	Yetenek (satır)
Nerede saklanır?	Nesnenin yanında	Etki alanında (süreçte ya da çekirdekte)
“Bu nesneye kim erişebilir?”	Kolay: listeye bakılır	Zor: bütün süreçler taranır
“Bu süreç neye erişebilir?”	Zor: bütün nesneler taranır	Kolay: listeye bakılır
Bir hakkı geri alma	Kolay: tek girdi silinir	Zor: bütün kopyalar bulunmalı
Bir hakkı devretme	Nesnenin listesi değişmeli	Kolay: yetenek kopyalanır
Erişim anındaki maliyet	Liste taraması	Yeteneğin geçerliliği denetlenir

Tablo 13.8: ACL ve yetenek listesinin karşılaştırılması.

Örnek 13.6

Tablo 13.7'deki matriste D3'ün Yazıcı1 hakkı hemen kaldırılmak isteniyor. Ayrıca D2, Dosya5 üzerindeki okuma hakkını D1'e devretmek istiyor. Her iş hangi gösterimde kolaydır?

1. **Geri alma.** ACL düzeninde Yazıcı1'in listesi {D2:W, D3:W}'dir; D3 girdisi silinir ve iş biter. Yetenek düzeninde D3'ün listesindeki yetenek silinmelidir. Ama D3 bu yeteneği daha önce başka süreçlere kopyalamış olabilir. Bütün kopyaları bulmak için sistemdeki bütün yetenek listelerini taramak gerekir.
2. **Devretme.** Yetenek düzeninde D2, Dosya5:R yeteneğinin bir kopyasını D1'e verir; Dosya5'in kendisine dokunulmaz. ACL düzeninde ise Dosya5'in listesine yeni bir D1:R girdisi eklenmelidir; bunu yalnız nesnenin sahibi yapabilir.

Gerçek sistemler çoğu zaman iki gösterimi birleştirir. UNIX, bir dosyayı açarken izin bitlerini, yani bir tür ACL'yi denetler. Açılan dosya için sürece bir **dosya tanımlayıcısı** verir. Bu tanımlayıcı aslında bir yetenektir: sonraki `read` ve `write` çağrılarında izin yeniden denetlenmez. Dosyanın izinleri bu arada değiştirilse bile açık tanımlayıcı çalışmayı sürdürür.

13.3.4 UNIX izinleri ve POSIX ACL'leri

Klasik UNIX, ACL fikrinin çok sıkıştırılmış bir biçimini kullanır. Her dosyanın bir sahibi ve bir grubu vardır. Dosyanın i-düğümünde üç sınıf için üçer bit tutulur: sahip (u), grup (g) ve diğerleri (o). Her üçlüde okuma (r), yazma (w) ve çalıştırma (x) bitleri vardır. Toplam dokuz bit, sekizli tabanda üç rakamla yazılır: r = 4, w = 2, x = 1.

```
$ ls -l rapor.txt
-rw-r----- 1 ali muhasebe 4096 rapor.txt
 rw-  sahip   r--  grup   ---  diğer

$ setfacl -m u:ayse:rw rapor.txt
$ getfacl rapor.txt
user::rw-
user:ayse:rw-
group::r--
mask::rw-
other::---
```

Çıktıdaki `-rw-r-----` dizgisi ilk karakterden sonra üç üçlüye bölünür. Sahip ali okur ve yazar (rw- = 4 + 2 = 6). muhasebe grubunun üyeleri yalnız okur (r-- = 4). Diğerlerinin hiçbir hakkı yoktur (--- = 0). Dosyanın sekizli modu 640'tır ve `chmod 640 rapor.txt` komutuyla bu hâle getirilebilir.

Yeni oluşturulan dosyaların izinleri **umask** değeriyle belirlenir. Program genellikle dosya için 666, dizin için 777 ister. Çekirdek umask'te 1 olan bitleri bu değerden siler. Yaygın umask 022 ile dosyalar 644 (rw-r--r--), dizinler 755 (rwxr-xr-x) olur. Daha sıkı bir umask olan 077 ile dosyalar yalnız sahibine açıktır: 600.

Dokuz bitlik düzen hızlıdır ve i-düğümüne sığar, ama kaba tanelidir. Ali dosyasını muhasebe grubunun tamamına değil, yalnız Ayşe'ye yazılabilir kılmak isterse dokuz bit yetmez. POSIX ACL'leri bu boşluğu doldurur. `setfacl -m u:ayse:rw rapor.txt` komutu listeye adlı bir kullanıcı girdisi ekler. `getfacl` çıktısında bu girdiyi ve yeni bir satırı görürüz: `mask`. Maske, adlı kullanıcı ve grup girdilerinin ulaşabileceği en fazla hakkı sınırlar. Ayşe'nin girdisi rw- olsa bile maske r-- yapılırsa Ayşe etkin olarak yalnız okuyabilir. ACL taşıyan dosyalar `ls -l` çıktısında izin dizgisinin sonundaki `+` işaretiyle ayırt edilir.

Örnek 13.7

Bir betik dosyasının modu 754'tür. Sahip, grup üyeleri ve diğerleri neler yapabilir? umask 027 iken oluşturulan yeni bir dosya ve dizin hangi izinleri alır?

1. 7 = 4 + 2 + 1: sahip okur, yazar ve çalıştırır (rwx).
2. 5 = 4 + 1: grup okur ve çalıştırır (r-x).
3. 4: diğerleri yalnız okur (r--). Dizgi `rwxr-xr--` 'dur.
4. umask 027, gruptan yazmayı (2) ve diğerlerinden her şeyi (7) siler. Dosya: 666 → 640 (rw-r-----). Dizin: 777 → 750 (rwxr-x---).

Windows'un NTFS dosya sistemi daha zengin bir ACL modeli kullanır. Her nesnenin isteğe bağlı erişim denetim listesi (DACL) sıralı girdilerden oluşur. Her girdi bir güvenlik kimliği (SID) ile bir hak maskesini ve bir izin ya da ret türünü taşır. Açık ret girdileri izinlerden önce gelir. Bir klasörün ACL'si

içindeki dosyalara kalıtım yoluyla geçer. Komut satırında `icacls` aracı bu listeleri gösterir ve düzenler. POSIX, NFSv4 ve NTFS ACL'lerinin anlamları tam örtüşmez. Bu yüzden dosyalar sistemler arasında taşınırken haklar bazen beklenmedik biçimde değişir.

ACL düzeninin artısı anlaşılır olmasıdır. Bir nesneye kimin erişebildiği tek bakışta görülür ve bir hak nesne başına kolayca geri alınır. Eksisi büyük sistemlerde yönetimin zorlaşması ve her erişimde listenin taranmasıdır. Tarama maliyeti önbellekle hafifletilir: UNIX'te izin denetimi yalnız `open` anında yapılır.

Örnek 13.8

Kullanıcılar dört gruba ve bir yönetici rolüne ayrılmış, dört nesne (A, B, C, D) var (Tablo 13.9). C nesnesinin ACL'si nedir? Grup 4 hangi nesnelere erişebilir?

Grup	A	B	C	D
Grup 1	R	RWX	—	—
Grup 2	W	—	—	—
Grup 3	—	—	X	R
Grup 4	—	—	—	—
Yönetici	RWX	RWX	RWX	RWX

Tablo 13.9: Gruplar ve nesneler için koruma matrisi.

C'nin ACL'si sütunun boş olmayan hücreleridir: {Grup 3: X, Yönetici: RWX}. Grup 4'ün satırı tamamen boştur, yani hiçbir ACL'de yer almaz. Güvenli varsayılan gereği bu grup hiçbir nesneye erişemez.

13.3.5 Kod izleme: erişim denetim listesi

Çekirdeğin bir ACL'yi nasıl taradığını küçük bir C fonksiyonuyla izleyebiliriz. A dosyasının ACL'si koruma matrisinin A sütunudur: Grup 1 okur, Grup 2 yazar, yönetici her şeyi yapar. Haklar bit olarak kodlanır: R = 4, W = 2, X = 1.

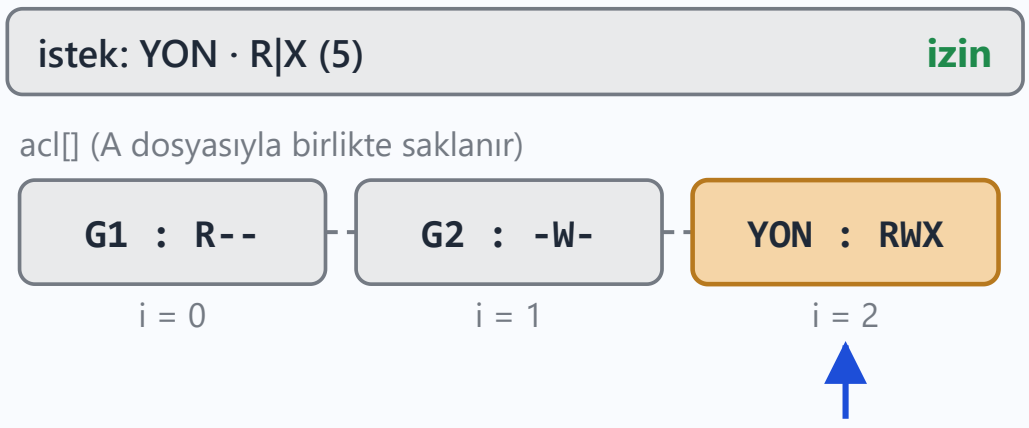
```c
/* A dosyasının ACL'si: R=4 W=2 X=1 */
struct girdi { int grup, hak; } acl[] =
    {{G1, 4}, {G2, 2}, {YON, 7}};

int erisim(int grup, int istek) {
    for (int i = 0; i < 3; i++)
        if (acl[i].grup == grup) {
            int h = acl[i].hak;
            return (h & istek) == istek;
        }
    return 0;   /* listede yok: yasak */
}
```

Dizide yalnız üç girdi vardır; Grup 3 ve Grup 4'ün A üzerindeki hücreleri boş olduğu için saklanmaz. `erisim` fonksiyonu listeyi baştan tarar. İsteyen grubu bulduğu ilk girdide kararı verir ve döner. Karar ifadesi `(h & istek) == istek`'tir. İstenen hakların hepsi girdide varsa VE işleminin sonucu isteğin kendisine eşit olur. Biri bile eksikse sonuç farklıdır ve erişim reddedilir. Grup listede yoksa döngü tamamlanır ve fonksiyon 0 döndürür. Tablo 13.10 dört çağrıyı izler.

Çağrı	Bakılan i	Eşleşen girdi	h & istek	Sonuç
`erisim(G2, W)`	0, 1	i = 1, h = 2	2 & 2 = 2	izin
`erisim(G1, W)`	0	i = 0, h = 4	4 & 2 = 0 ≠ 2	ret
`erisim(G4, R)`	0, 1, 2	yok	—	ret (0)
`erisim(YON, R|X)`	0, 1, 2	i = 2, h = 7	7 & 5 = 5	izin

Tablo 13.10: `erisim` fonksiyonunun dört çağrısı. Değerler Python ile yeniden hesaplandı.



Şekil 13.18: Son çağrı: yönetici okuma ve çalıştırma hakkını birlikte istiyor (5); üçüncü girdide eşleşme bulunur ve 7 & 5 = 5 olduğu için izin verilir.

İzleme üç dersi açıkça gösterir. Birincisi, karar ilk eşleşen girdide verilir: Grup 1'in yazma isteği reddedilir ve liste daha fazla taranmaz. NTFS'te ret girdilerinin başa konmasının nedeni de budur. İkincisi, listede bulunmayan bir grubun isteği için bütün liste taranır. Uzun ACL'lerde her erişimin maliyeti budur: n girdili bir listede en kötü durumda O(n) karşılaştırma. Üçüncüsü, güvenli varsayılan ilkesi koda gömülüdür. Açıkça izin verilmeyen her şey yasaktır; fonksiyonun son satırı bunu söyler.

Sık yapılan hata

Karar ifadesini `(h & istek) != 0` diye yazmak. Bu ifade istenen haklardan *biri* varsa izin verir. Grup 1 okuma ve yazmayı birlikte isterse (6) sonuç 4 & 6 = 4 ≠ 0 olur ve yazma hakkı olmadığı hâlde izin verilir. Doğru ifade, istenen bitlerin *hepsinin* girdide bulunmasını ister.

13.3.6 Yetenek listeleri

Yetenek düzeninde bir yetenek, sahibine bir nesneye erişme hakkı veren bir bilettir. Biletin en önemli özelliği sahte üretilememesidir. Bir süreç kendi kendine "Dosya7'ye yazma yeteneği" üretebilseydi düzen hiçbir şey korumazdı. Yetenekleri sahte üretimden korumanın üç yolu vardır.

Çekirdekte tutmak
: Yetenek listesi çekirdeğin belleğinde durur. Süreç yalnız listedeki bir girdinin indisini bilir ve bu indisi sistem çağrısına verir: `read(cap 2, …)`. UNIX dosya tanımlayıcıları tam olarak böyle çalışır (Şekil 13.19).

Etiketli donanım
: Bellekteki her sözcüğe, onun bir yetenek olup olmadığını gösteren ek bir bit eklenir. Bu bit yalnız özel komutlarla değiştirilebilir. Cambridge Üniversitesi'nin CHERI mimarisi bu yaklaşımı bugünkü işlemcilere taşır.

Şifreli koruma
: Yetenek kullanıcının elinde durur, ama bir doğrulama alanı taşır. Bu alan, nesne ve hakların yalnız sunucunun bildiği bir gizli değerle birlikte tek yönlü bir fonksiyondan (örneğin HMAC) geçirilmesiyle elde edilir. Kullanıcı hakları değiştirirse doğrulama alanı tutmaz. Amoeba dağıtık işletim sistemi bu yöntemi kullanırdı.



Şekil 13.19: Solda çekirdekte tutulan C-listesi: süreç yalnız indis görür. Sağda şifreli korumalı yetenek biçimi.

Yetenekler üzerinde dört temel işlem tanımlanır. *Yeteneği kopyalamak* aynı nesne için yeni bir yetenek üretir; devretme budur. *Nesneyi kopyalamak* yeni bir nesne ve ona ait yeni bir yetenek üretir. *Yeteneği kaldırmak* listeden bir girdiyi siler; nesne yerinde kalır. *Nesneyi yok etmek* ise nesneyi ve ona ait yeteneği kalıcı olarak siler.

Saf yetenek sistemleri bugün araştırma ve özel alanlarda yaşar: biçimsel olarak doğrulanmış seL4 mikro çekirdeği ve Google'ın Fuchsia sistemi baştan sona yeteneklerle çalışır. FreeBSD'nin Capsicum altyapısı ise sıradan bir UNIX'e yetenek kipi ekler. Bu kipe geçen bir süreç artık dosya adlarıyla yeni dosya açamaz; yalnız elindeki tanımlayıcıları kullanabilir.

13.3.7 En az ayrıcalık ve setuid

Saltzer ve Schroeder'in 1975'teki ünlü makalesi, güvenli sistem tasarımının ilkelerini sıralar. Bunların en çok bilineni şudur.

> **Tanım 13.12**
>
> **En az ayrıcalık ilkesi** (*principle of least privilege*): her bileşen, işine yetecek en küçük hak kümesiyle çalışmalıdır. Bu haklar da yalnız gerektiği süre boyunca tutulmalıdır. Bileşen ele geçirilirse verebileceği zarar bu kümeyle sınırlı kalır.

UNIX'te bu ilkeyle çatışan klasik bir durum vardır. Parolaların özetleri `/etc/shadow` dosyasında durur. Dosyanın sahibi root'tur ve modu `-rw-r-----`'dır: sıradan kullanıcılar onu okuyamaz bile. Peki Ali kendi parolasını nasıl değiştirir? Çözüm **setuid** bitidir. `/usr/bin/passwd` programının sahibi root'tur ve izinlerinde x yerine s görülür: `-rwsr-xr-x`. Bu bit açıkken program, onu çalıştıranın değil dosya sahibinin haklarıyla çalışır.

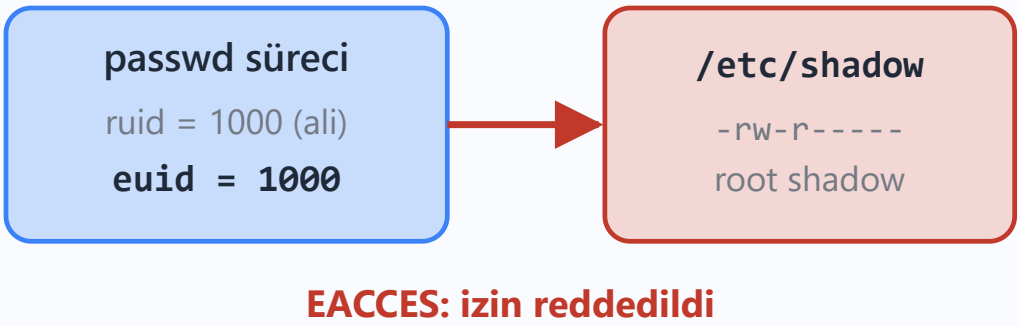
Bu davranışı anlamak için bir sürecin üç kullanıcı kimliğini bilmek gerekir. **Gerçek kimlik** (ruid) süreci başlatan kullanıcıdır. **Etkin kimlik** (euid) izin denetimlerinde kullanılan kimliktir. **Saklı kimlik** (suid), sürecin daha sonra geri dönebileceği bir kimliği saklar. Normalde üçü eşittir. setuid bitli bir program `execve` ile başlatıldığında etkin ve saklı kimlik dosya sahibine çevrilir; gerçek kimlik değişmez.

```c
/* /usr/bin/passwd  -rwsr-xr-x root */
int main(void) {
    uid_t me = getuid();
    int fd = open("/etc/shadow", O_RDWR);
    update_own_entry(fd, me);
    close(fd);
    setresuid(me, me, me);
    /* ... kalan iş ... */
    fd = open("/etc/shadow", O_RDWR);
    if (fd < 0) perror("open");
    else close(fd);
    return 0;
}
```

Kod, ayrıcalığı yalnız gerektiği kadar kullanan bir `passwd` taslağıdır. `getuid()` gerçek kimliği döndürür. Program kimin adına çalıştığını buradan öğrenir ve yalnız onun satırını değiştirir. `open` çağrısında çekirdek erişimi etkin kimliğe göre denetler. euid 0 olduğu için dosya açılır. İş biter bitmez `setresuid(me, me, me)` üç kimliğin üçünü de 1000 yapar. Saklı kimlik de değiştiği için geri dönüş yoktur. Sondaki ikinci `open` artık EACCES hatasıyla başarısız olur ve −1 döner; program dönüş değerini denetler ve `perror` hatayı “open: Permission denied” olarak yazar (Tablo 13.11).

Adım	ruid	euid	suid	Sonuç
kabuk (ali)	1000	1000	1000	ayrıcalıksız
`execve` (setuid root)	1000	0	0	etkin kimlik dosya sahibine döner
`getuid()`	1000	0	0	1000 döner
`open("/etc/shadow")`	1000	0	0	izin verildi (euid 0)
`setresuid(1000, 1000, 1000)`	1000	1000	1000	ayrıcalık kalıcı bırakıldı
`open("/etc/shadow")`	1000	1000	1000	EACCES, fd = −1

Tablo 13.11: Ali (uid 1000) `passwd`'yi çalıştırdığında kimliklerin izlenmesi. Değerler Python benzetimiyle doğrulandı.



Şekil 13.20: Ayrıcalık bırakıldıktan sonra euid 1000'dir ve `/etc/shadow`'a erişim reddedilir.

Ayrıcalığı bu kadar erken bırakmanın değeri, programın geri kalanında bir hata çıktığında anlaşılır. Diyelim ki programın sonraki kısmında bir tampon taşması var ve saldırgan bu hatayı istismar ederek kendi kodunu çalıştırıyor. Ayrıcalık bırakılmasaydı saldırganın kodu root olarak çalışırdı. Bırakıldığı için Ali'nin haklarıyla çalışır ve Ali'nin zaten yapabildiğinden fazlasını yapamaz. setuid root programlar bu yüzden ayrıcalık yükseltme saldırılarının klasik hedefidir. Her biri dikkatle incelenmeli ve sayıları en aza indirilmelidir.

Sık yapılan hata

Ayrıcalığı `seteuid(getuid())` ile bıraktığını sanmak. Bu çağrı yalnız etkin kimliği değiştirir; saklı kimlik 0 olarak kalır. Ayrıcalıksız bir süreç etkin kimliğini gerçek ya da saklı kimliğine çevirebildiği için, saldırganın kodu `seteuid(0)` ile yeniden root olur. Kalıcı bırakmada saklı kimlik de değiştirilmelidir: `setresuid(me, me, me)`.

13.3.8 Linux yetenekleri ve tasarım ilkeleri

Geleneksel UNIX'te root ya her şeyi yapabilir ya da hiçbir şeyi. 1024'ün altındaki portlar ayrıcalıklıdır. Bu yüzden 80. portu dinlemek isteyen bir web sunucusu root olmak zorundadır. Böylece web sunucusu dosya izinlerini aşma, saati değiştirme, çekirdek modülü yükleme haklarını da gereksiz yere kazanır. Linux, root'un tek parça yetkisini yaklaşık 41 ayrı parçaya böler ve bunlara **yetenek** (*capability*) der (Şekil 13.21). Buradaki "yetenek" sözcüğü, yetenek listesi düzenindekiyle aynı değildir; anlamı "ayrıcalık parçası"dır.



Şekil 13.21: Root yetkisinin Linux yeteneklerine bölünmesi. CAP_SYS_ADMIN o kadar geniştir ki "yeni root" diye anılır.

Örneğin CAP_NET_BIND_SERVICE 1024'ün altındaki portları açma hakkıdır. CAP_DAC_OVERRIDE dosya izinlerini aşar, CAP_KILL başka kullanıcıların süreçlerine sinyal gönderir, CAP_SYS_TIME sistem saatini ayarlar. Her iş parçacığının dört yetenek kümesi vardır: izinli, etkin, kalıtılabilir ve sınırlayıcı. Bir dosyaya da yetenek atanabilir. Aşağıdaki komutlar web sunucusuna yalnız düşük numaralı port açma hakkını verir.

```
$ sudo setcap cap_net_bind_service=+ep /usr/local/bin/websrv
$ getcap /usr/local/bin/websrv
/usr/local/bin/websrv cap_net_bind_service=ep     # 80. kapıyı açabilir, başka hiçbir root
hakkı yok
```

`=ep` soneki yeteneğin izinli (p) ve etkin (e) kümelere eklendiğini söyler. Program artık root olmadan 80. portu dinleyebilir. Ele geçirilse bile dosya izinlerini aşamaz ya da başka süreçleri öldüremez.

Saltzer ve Schroeder'in listesi en az ayrıcalıkla bitmez. Öteki ilkeler de bugünkü sistemlerde karşılık bulur.

Güvenli varsayılan
: Açıkça izin verilmeyen her şey yasaktır. ACL taramasının son satırındaki `return 0` bu ilkedir.

Tam aracılık
: Her erişim denetlenir; bir kez verilen izin önbellekte sonsuza dek geçerli sayılmaz. Referans izleyicinin birinci koşuludur.

Açık tasarım
: Güvenlik, tasarımın gizli kalmasına dayanmaz. Kerckhoffs ilkesinin genel biçimidir.

Ayrıcalıkların ayrılması
: Kritik bir işlem birden çok koşula ya da kişiye bağlanır. İki anahtarlı kasa ve iki kişi onayı bu ilkenin örnekleridir.

Psikolojik kabul edilebilirlik
: Güvenlik mekanizması kullanıcıyı bezdirmemelidir. Aksi hâlde kullanıcılar onu atlatmanın yolunu bulur: parolayı ekrana yapıştırır, izinleri 777 yapar.

Bugünkü yazılımlar bu ilkeleri **ayrıcalık ayrımı** (*privilege separation*) ile uygular. OpenSSH sunucusu buna iyi bir örnektir. Küçük bir root süreci yalnız kimlik doğrulama gibi ayrıcalık gerektiren işleri yapar. Ağdan gelen verinin ayrıştırılması gibi riskli işler, hiçbir ayrıcalığı olmayan bir çocuk süreçte yürür. systemd de hizmet dosyalarında `CapabilityBoundingSet=` ile bir hizmetin hangi yetenekleri kullanabileceğini sınırlar. `NoNewPrivileges=` ise hizmetin setuid programlar aracılığıyla yeni ayrıcalık kazanmasını engeller.

13.4 Biçimsel güvenlik modelleri

Koruma matrisi her politikayı ifade edebilir, ama bir hakkın sızıp sızmayacağı genel durumda hesaplanamaz. Biçimsel modeller bu sorunu kuralları baştan kısıtlayarak çözer. Kurallar o kadar basittir ki uyulduğu sürece belirli bir zararın imkânsız olduğu kanıtlanabilir. Bu kesimde iki klasik model ve onların bile kapatamadığı bir sızıntı yolu, gizli kanallar ele alınır.

İki modelde de denetimi kullanıcı değil sistem yapar. Sıradan UNIX izinlerinde dosyanın sahibi izinleri istediği gibi değiştirebilir; buna **isteğe bağlı erişim denetimi** (*discretionary access control*, DAC) denir. Biçimsel modellerde ise politikayı yönetici belirler ve hiçbir süreç onu gevşetemez; buna **zorunlu erişim denetimi** (*mandatory access control*, MAC) denir.

13.4.1 Bell-LaPadula modeli

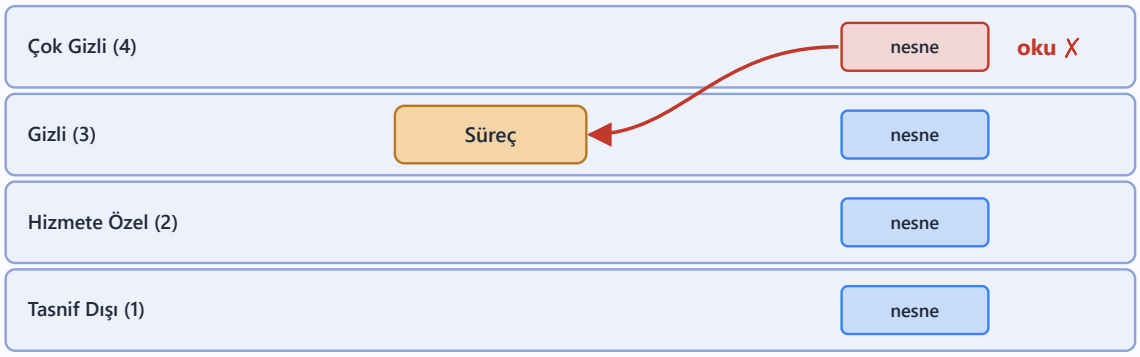
Askerî kuruluşlarda belgeler gizlilik düzeylerine ayrılır. Sunumdaki düzenle bu düzeyler büyükten küçüğe Çok Gizli (4), Gizli (3), Hizmete Özel (2) ve Tasnif Dışı (1)'dir. Her kişinin de bir yetki düzeyi vardır. Bell ve LaPadula'nın 1973'te tanımladığı model, bu düzenin bilgisayardaki karşılığıdır. Amacı tek bir şeydir: bilginin yüksek düzeyden alçak düzeye sızmasını engellemek.

Tanım 13.13

Bell-LaPadula modelinde (BLP) her süreç ve her nesne bir gizlilik düzeyi taşır. Model iki kural koyar:

- **Basit güvenlik özelliği:** k düzeyindeki bir süreç yalnız düzeyi k'ye eşit ya da k'den küçük nesneleri okuyabilir (*yukarı okuma yok*).
- *** özelliği** (yıldız özelliği): k düzeyindeki bir süreç yalnız düzeyi k'ye eşit ya da k'den büyük nesnelere yazabilir (*aşağı yazma yok*).

Birinci kural sezgiyle uyumludur: Gizli yetkili bir kişi Çok Gizli bir belgeyi okuyamaz. İkinci kural ilk bakışta tuhaf görünür. Gizli düzeydeki bir süreç neden Tasnif Dışı bir dosyaya yazamasın? Cevap, sürecin okuduğu bilgiyi yazdığı yere taşıyabilmesidir. Gizli bir belgeyi okuyan süreç, onun içeriğini Tasnif Dışı bir dosyaya kopyalarsa bilgi aşağı sızmış olur. Yukarı yazmak ise gizliliği bozmaz: Gizli bilgi Çok Gizli bir dosyaya geçerse daha sıkı korunur. Bu tür yazma genellikle "kör" bir eklemedir; süreç yazdığı dosyayı okuyamaz.



Şekil 13.22: Yukarı okuma yok: Gizli düzeydeki süreç Çok Gizli nesneyi okuyamaz.



Şekil 13.23: Aşağı yazma yok: süreç içinde çalışan bir Truva atı bile bilgiyi Tasnif Dışı düzeye yazamaz.

Örnek 13.9

Gizli (3) düzeyde çalışan bir süreç dört düzeydeki nesnelere okuma ve yazma isteği yapıyor. BLP hangilerine izin verir?

1. **Okuma.** Nesne düzeyi ≤ 3 olmalıdır. Çok Gizli (4): ret. Gizli, Hizmete Özel, Tasnif Dışı: izin.
2. **Yazma.** Nesne düzeyi ≥ 3 olmalıdır. Çok Gizli ve Gizli: izin. Hizmete Özel ve Tasnif Dışı: ret.

Süreç kendi düzeyinde hem okur hem yazar. Aşağıyı yalnız okur, yukarıya yalnız yazar.

BLP'nin asıl kazancı, kuralların kullanıcının iyi niyetine bağlı olmamasıdır. Gizli yetkili bir kullanıcı, içinde bir **Truva atı** gizlenmiş bir program çalıştırsın. Program kullanıcının haklarıyla çalışır ve Gizli belgeleri okuyabilir. DAC düzeninde program okuduğunu herkese açık bir dosyaya yazabilirdi. BLP'de ise bu yazma aşağı yazma olduğu için sistem reddeder. Kural, programın ne yapmaya çalıştığından bağımsız olarak işler.

13.4.2 Biba modeli

BLP yalnız gizliliği korur; bütünlük hakkında hiçbir şey söylemez. Tasnif Dışı bir süreç Çok Gizli bir dosyaya yazabilir ve onu bozabilir. Biba 1977'de bütünlüğü koruyan bir model önerdi. Biba, BLP'nin aynasıdır: düzeyler bu kez gizliliği değil *güvenilirliği* gösterir. Sunumdaki örnekte düzeyler Yüksek (sistem), Orta (kullanıcı) ve Düşük (İnternet)'tir.

Tanım 13.14

Biba modelinde her süreç ve nesne bir bütünlük düzeyi taşır. Model iki kural koyar:

- **Basit bütünlük özelliği:** k düzeyindeki bir süreç yalnız düzeyi k'ye eşit ya da k'den büyük nesneleri okuyabilir (*aşağı okuma yok*).
- *** bütünlük özelliği:** k düzeyindeki bir süreç yalnız düzeyi k'ye eşit ya da k'den küçük nesnelere yazabilir (*yukarı yazma yok*).

Yukarı yazma yasağının anlamı açıktır: kullanıcı süreçleri sistem dosyalarını değiştiremez. Aşağı okuma yasağı daha incedir. İnternetten gelen güvenilmeyen bir veriyi okuyup ona göre davranan bir süreç "kirlenir". Kötü niyetli bir girdi sürecin davranışını değiştirebilir ve süreç sonra bu davranışla daha güvenilir nesnelere yazabilir. Biba bunu kaynağında engeller.



Şekil 13.24: Biba'da yukarı yazma yok: Düşük bütünlüklü tarayıcı süreci Orta düzeydeki kullanıcı dosyalarına yazamaz.

Biba'nın gerçek bir uygulaması Windows'un **zorunlu bütünlük denetimidir** (*Mandatory Integrity Control*, MIC). Windows'ta süreçler ve nesneler Düşük, Orta, Yüksek ve Sistem bütünlük düzeylerinden birini taşır. Sıradan uygulamalar Orta düzeyde çalışır. Tarayıcıların sekme süreçleri ise Düşük düzeyde çalışır. Bir web sayfasındaki istismar tarayıcı sürecini ele geçirse bile, kullanıcının Orta düzeydeki belgelerine yazamaz. Bu, yukarı yazma yasağının ta kendisidir.

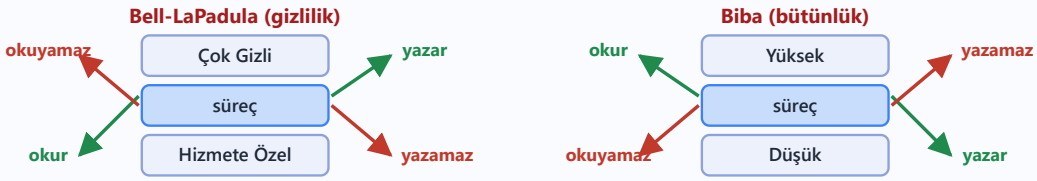
13.4.3 İki modelin karşılaştırılması

İki model aynı yapıyı ters yönde kullanır (Tablo 13.12).

	Bell-LaPadula (1973)	**Biba (1977)**
Korur	Gizlilik	Bütünlük
Okuma	Aşağı serbest, yukarı yok	Yukarı serbest, aşağı yok
Yazma	Yukarı serbest, aşağı yok	Aşağı serbest, yukarı yok

	Bell-LaPadula (1973)	Biba (1977)
Engellenen	Bilginin aşağı sızması	Kirli verinin yukarı bulaşması
Bugün	SELinux MLS, askerî sistemler	Windows MIC, FreeBSD ve Linux bütünlük politikaları

Tablo 13.12: Bell-LaPadula ve Biba.



Şekil 13.25: İki modelde sürecin okuyup yazabildiği yönler. Okları ezberlemek yerine her modelin neyi engellediği akılda tutulur.

Örnek 13.10

Bir süreç Orta düzeyde. Yüksek'i okumak, Yüksek'e yazmak, Düşük'ü okumak ve Düşük'e yazmak isteklerine iki model ne der? İkisi birlikte uygulanırsa süreç neye erişebilir?

İstek	BLP (gizlilik)	Biba (bütünlük)
Yüksek'i okumak (yukarı okuma)	ret	izin
Yüksek'e yazmak (yukarı yazma)	izin	ret
Düşük'ü okumak (aşağı okuma)	izin	ret
Düşük'e yazmak (aşağı yazma)	ret	izin

Tablo 13.13: Orta düzeydeki sürecin dört isteği. Kararlar Python ile doğrulandı.

Her satırda modellerden biri reddeder. İki model birlikte uygulanırsa süreç yalnız kendi düzeyindeki nesneleri okuyup yazabilir. Biri gizliliği, öteki bütünlüğü korur ve hedefleri çatışır.

Adlandırmalar karıştırılmaya çok yatkındır. BLP'de basit güvenlik özelliği yukarı okumayı, * özelliği aşağı yazmayı yasaklar. Biba'da basit bütünlük özelliği aşağı okumayı, * bütünlük özelliği yukarı yazmayı yasaklar. İki modelde de “basit” sözcüğü okuma kuralını, yıldız da yazma kuralını gösterir.

Sık yapılan hata

BLP kurallarını “yukarı yazma yok” diye hatırlamak. Gizlilikte tehlike bilginin *aşağı* akmasıdır; yasaklanan yazma da aşağı doğru olandır. Kuralı çıkarmanın yolu şudur: önce modelin neyi korumak istediği sorulur, sonra bilginin hangi yönde akarsa bu hedefi bozacağı düşünülür.

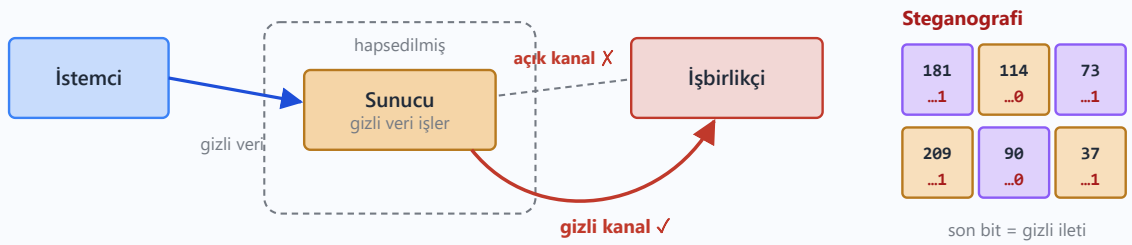
İki modelin ortak bir sınırı vardır: yalnız açık okuma ve yazma işlemlerini denetlerler. Bilgi bir dosyaya yazılmadan da aktarılabilir. Bir sonraki konu bu yolları ele alır.

13.4.4 Gizli kanallar ve steganografi

Lampson 1973'te bir soru sordu: bir program, verdiği hizmet sırasında işlediği gizli bilgiyi dışarı sızdırmasın diye tam olarak hapsedilebilir mi? Buna **hapsetme problemi** (*confinement problem*) denir. Senaryo şöyledir: bir istemci, gizli verisini işlemesi için bir sunucuya verir. Sunucu kötü niyetlidir ve veriyi dışarıdaki bir işbirlikçiye aktarmak ister. İşletim sistemi sunucuyu hapseder: dosya yazamaz, ağa bağlanamaz, işbirlikçiyle ileti alışverişi yapamaz. Lampson'ın cevabı olumsuzdur.

Tanım 13.15

Gizli kanal (*covert channel*), sistem tasarımcısının iletişim için öngörmediği ve güvenlik politikasını atlayan bir bilgi aktarma yoludur.



Şekil 13.26: Hapsedilmiş sunucunun açık kanalı kapalıdır, ama işbirlikçiye gizli bir kanaldan veri sızdırabilir. Sağda steganografi: piksellerin son bitlerinde gizlenen ileti.

Paylaşılan her kaynak bir gizli kanal olabilir. Kanallar üç türe ayrılır.

Depolama kanalı
: Paylaşılan bir durumun değişmesiyle bilgi aktarılır: bir dosyanın var olup olmaması, kilitli ya da açık olması, diskin doluluk oranı.

Zamanlama kanalı
: Olayların zamanlamasıyla bilgi aktarılır. Sunucu CPU'yu yoğun kullanırsa 1, boşta kalırsa 0 gönderir. İşbirlikçi kendi işinin ne kadar yavaşladığını ölçerek bitleri okur. Önbellek gecikmesi de böyle bir kanaldır.

Kaynak kanalı
: Paylaşılan bir kaynağın kullanım miktarıyla bilgi aktarılır: sayfa hatası oranı, ağ trafiğinin hacmi.

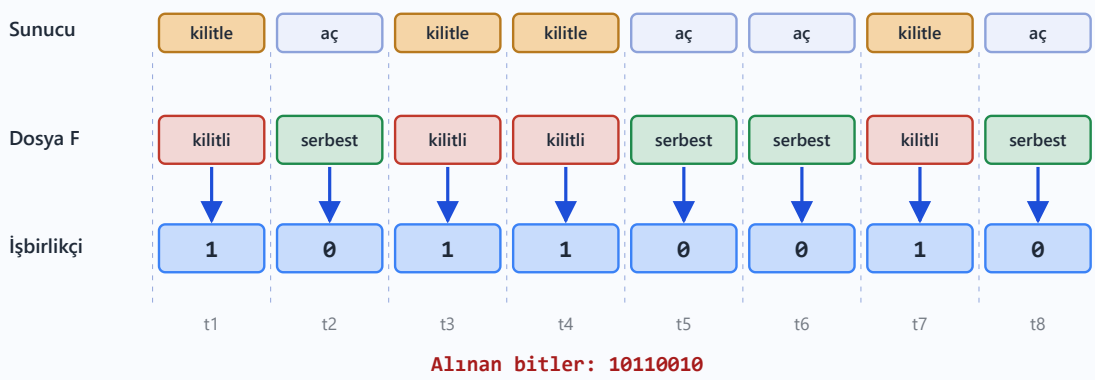
Bütün gizli kanalları kapatmak pratikte imkânsızdır, çünkü süreçlerin paylaştığı her kaynağın durumunu öteki süreçler gözleyebilir. Yapılabilecek olan kanalların **bant genişliğini**, yani saniyede aktarabildikleri bit sayısını düşürmektir. Gürültü eklemek, kaynakları düzeylere göre ayırmak ve saatlerin

çözünürlüğünü azaltmak bu yöntemlerdendir.

Gizli kanalın yakın bir akrabası **steganografidir**: bilgiyi şifrelemek yerine varlığını gizlemek. Bir renkli görüntüde her pikselin kırmızı, yeşil ve mavi bileşenleri sekizer bittir. Her bileşenin en düşük anlamlı bitini değiştirmek rengi en fazla 1/256 oranında değiştirir. Göz bu farkı göremez. Şekil 13.26'daki altı değerin son bitleri sırayla 1, 0, 1, 1, 0, 1'dir. Kanal başına bir bitle 1920 × 1080'lik bir fotoğraf 777.600 bayt saklayabilir. Tanenbaum'un bilinen örneğinde üç zebralı bir fotoğrafın içine beş Shakespeare oyunu gizlenir; görüntü gözle hiç değişmez.

13.4.5 Dosya kilidiyle gizli kanal

Somut bir depolama kanalı kuralım. Sunucu ve işbirlikçi önceden anlaşır: her zaman aralığında F dosyası kilitliyse 1, serbestse 0 gönderilmiş sayılır. Sunucu bir bit göndermek için F'yi kilitler ya da bırakır. İşbirlikçi aynı aralıkta F'yi kilitlemeyi dener. Deneme başarısız olursa dosya kilitlidir ve bit 1'dir; başarılı olursa bit 0'dır. Bu düzende ne okuma ne yazma vardır; F'nin içeriği hiç değişmez. BLP'nin denetlediği hiçbir işlem yapılmaz, ama bilgi akar.



Şekil 13.27: Sekiz aralıkta bir bayt: işbirlikçi kilit denemelerinden 10110010 dizisini okur.

Şekil 13.27'deki zaman çizelgesinde sunucu sekiz aralıkta F'yi kilitle, aç, kilitle, kilitle, aç, aç, kilitle, aç sırasıyla değiştirir. İşbirlikçi 10110010 dizisini, yani 0xB2 = 178 değerini alır. Aralık 10 ms ise saniyede 100 aralık, dolayısıyla yaklaşık 100 bit/s'lik bir kanal elde edilir. 256 bitlik bir şifreleme anahtarı bu hızla 2,56 saniyede sızar.

Örnek 13.11

Sunucu `sir = 'K'` değerini en yüksek bitten başlayarak gönderiyor:

```
for (int i = 7; i >= 0; i--) {
    if ((sir >> i) & 1) kilitle(F);
    else                birak(F);
    bekle(10);                      /* ms */
}
```

1. 'K'nin ASCII kodu 0x4B'dir; ikili gösterimi 01001011.
2. i = 7'den 0'a bitler 0, 1, 0, 0, 1, 0, 1, 1'dir. Sunucu sırayla bırak, kilitle, bırak, bırak, kilitle, bırak, kilitle, kilitle yapar.
3. İşbirlikçi her aralıkta `bit = kilitlemeyi_dene(F) ? 0 : 1` hesaplar ve 80 ms sonunda 01001011 = 0x4B = 'K' değerini bulur.

Kanalı bulan bir yönetici aralığı uzatabilir ya da kilit durumuna rastgele gecikmeler ekleyebilir. Bu önlemler kanalın hızını düşürür, ama sıfırlamaz.

13.5 Kimlik doğrulama

Koruma matrisinin bütün kararları, sürecin hangi kullanıcıya ait olduğu bilgisine dayanır. Bu bilgi sistem açılırken kullanıcının kendini tanıtmasıyla elde edilir. Kimlik doğrulama zayıfsa en iyi erişim denetimi de işe yaramaz: saldırgan meşru bir kullanıcının kimliğiyle içeri girer ve onun bütün haklarını kullanır.

13.5.1 Kimlik doğrulama etkenleri

Tanım 13.16

Kimlik doğrulama (*authentication*), bir kullanıcının iddia ettiği kişi olduğunu doğrulamaktır. Hangi işlemlere izin verileceğine karar vermek ise ayrı bir adımdır ve **yetkilendirme** (*authorization*) diye anılır.

Kimlik doğrulama dört tür kanıta, yani **etkene** dayanabilir.

Bilinen bir şey
: Parola, PIN, güvenlik sorusu. Ucuzdur ve her yerde kullanılabilir. Ama çalınabilir, tahmin edilebilir ve kullanıcılar aynı parolayı birçok sitede yeniden kullanır.

Elde tutulan bir şey
: Telefon, akıllı kart, FIDO2 güvenlik anahtarı, TOTP üreteci. Fiziksel olarak çalınabilir, ama uzaktan kopyalanamaz.

Kişinin kendisi
: Parmak izi, yüz, iris. Kullanımı kolaydır, ama değiştirilemez: sızarsa kalıcı olarak “yanar”.

Bulunulan yer ya da davranış
: Konum, IP adresi, yazım ritmi, aygıt parmak izi. Genellikle tek başına değil, ek bir risk sinyali olarak kullanılır.

Çok etkenli kimlik doğrulama (*multi-factor authentication*, MFA), farklı kategorilerden en az iki etkenin birlikte istenmesidir. Kategori koşulu önemlidir. "Parola + güvenlik sorusu" MFA değildir, çünkü ikisi de bilinen bir şeydir ve aynı oltalama sayfası ikisini birden toplayabilir. MFA, çalınan parolalara karşı tek başına en etkili önlemdir.

Eski sistemlerde bir tür **soru-yanıt** (*challenge-response*) doğrulaması kullanılırdı. Kullanıcıya kayıt sırasında sorular sorulur, girişte bunlardan biri rastgele seçilirdi. Tipik sorular annenin kızlık soyadı ya da ilk evcil hayvanın adıydı. Bu yanıtların çoğu bugün sosyal medyadan kolayca bulunabildiği için yöntem zayıf kabul edilir. Soru-yanıt fikrinin güçlü biçimi ise kriptografiktir: sunucu rastgele bir sayı gönderir, kullanıcının aygıtı onu gizli bir anahtarla işler. FIDO2 bu fikri kullanır.

13.5.2 Parola saklama: tuz ve yavaş özet

Parolalar hiçbir zaman açık hâlde saklanmaz. Saklansaydı parola dosyasını ele geçiren bir saldırgan, ya da dosyayı okuyabilen bir yönetici, bütün parolaları anında öğrenirdi. Kullanıcılar aynı parolayı başka yerlerde de kullandığı için zarar tek sistemle sınırlı kalmazdı. Sistem bunun yerine parolanın özetini saklar. Girişte kullanıcının yazdığı parolanın özeti hesaplanır ve saklananla karşılaştırılır.

Ancak düz bir özet yeterli değildir (Şekil 13.28). İnsanlar sık sık aynı parolaları seçer. Aynı parola aynı özeti verdiği için, özet dosyasına bakan saldırgan hangi kullanıcıların aynı parolayı seçtiğini hemen görür. Daha kötüsü, saldırgan milyonlarca yaygın parolanın özetini önceden bir kez hesaplayıp bir tabloya koyabilir. Bu tür tablolara **gökkuşağı tablosu** (*rainbow table*) denir. Sızan dosyadaki her özet bu tabloda aranır ve tek seferde binlerce parola kırılır.



Şekil 13.28: Tuzsuz özette ali ile veli'nin parolası aynı olduğu için özetleri de aynıdır; saldırgan tekrarı görür.

Tanım 13.17

Tuz (*salt*), her kullanıcı için ayrı üretilen ve özetten önce parolaya eklenen rastgele bir değerdir. Tuz gizli değildir; özetin yanında açıkça saklanır.

Tuzun etkisi basittir: aynı parola artık farklı bir girdidir. ali ile veli'nin parolası aynı olsa da tuzları farklı olduğu için özetleri tamamen farklıdır. Önceden hesaplanmış tablolar da işe yaramaz, çünkü her olası tuz için ayrı bir tablo gerekir. 1970'lerin UNIX'i bile 2 karakterlik, yani 12 bitlik bir tuz kullanırdı. Bu, her parolanın 4096 farklı biçimde saklanabileceği ve tablo boyunun 4096 kat büyüyeceği anlamına geliyordu.

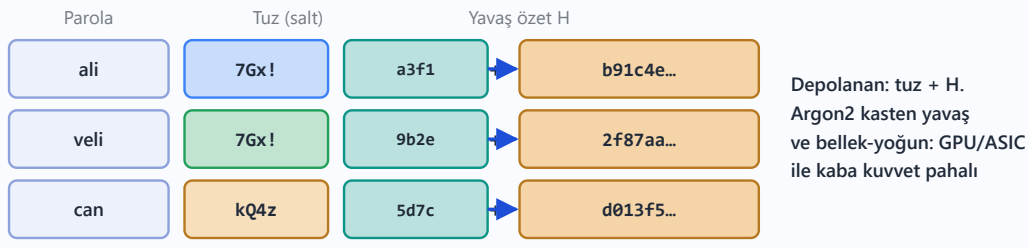
Tuz tek başına yetmez. SHA-256 gibi genel amaçlı özet fonksiyonları hızlı olsun diye tasarlanmıştır. Bugünkü bir ekran kartı saniyede milyarlarca SHA-256 hesaplayabilir. Parola saklamada istenen tam tersidir: her deneme pahalı olmalıdır. Bu yüzden parolalar kasıtlı olarak yavaş ve bellek yoğun fonksiyonlarla özetlenir. Güncel öneri **Argon2id**'dir (RFC 9106); scrypt ve bcrypt de kullanılır. Tek bir doğrulama birkaç milisaniye sürer ve kullanıcı bunu fark etmez. Milyarlarca deneme ise yıllar alır.

```
# Kayıt / parola değiştirme
salt = os.urandom(16)
h = argon2(password, salt)
store(user, salt, h)

# Giriş denemesi
salt, h = lookup(user)
g = argon2(attempt, salt)
if constant_eq(g, h): ok()
```

Sunumdaki sözde kod iki işlemi gösterir. Kayıt ya da parola değiştirmede 16 baytlık rastgele bir tuz üretilir. Parola bu tuzla Argon2'den geçirilir ve kullanıcı adı, tuz ve özet birlikte saklanır. Girişte kullanıcının tuzu ve özeti bulunur, girilen deneme aynı tuzla özetlenir ve iki özet karşılaştırılır.

Karşılaştırmanın `constant_eq` ile yapılması bir ayrıntı değildir. Sıradan bir karşılaştırma ilk farklı baytta durur. Saldırgan yanıt süresini çok hassas ölçebiliyorsa, ilk kaç baytın doğru olduğunu süreden çıkarabilir. Sabit zamanlı karşılaştırma her zaman bütün baytları karşılaştırır ve bu zamanlama sızıntısını önler.



Şekil 13.29: Her kullanıcının ayrı tuzu vardır; aynı parola farklı özetler verir. Depolanan değer tuz ve yavaş özettir.

Sık yapılan hata

Tuzun gizli tutulması gerektiğini ya da tuz sayesinde zayıf parolaların güvenli hâle geldiğini düşünmek. Tuz yalnız toplu saldırıyı ve önceden hesaplamayı engeller. Saldırgan tek bir kullanıcıyı hedeflerse o kullanıcının tuzunu bilir ve sözlükteki parolaları tek tek dener. Zayıf parolaya karşı yavaş özet ve parolanın uzunluğu işe yarar. Bir sızıntıdan sonra da parolalar her durumda sıfırlanmalıdır.

13.5.3 Kaba kuvvet süresi

Tuzun ve yavaşlığın ayrı ayrı katkısını sayılarla görmek öğreticidir.

Örnek 13.12

Parolalar 8 küçük harften oluşuyor (26 harfli alfabe). Saldırgan sızan veritabanındaki tek bir kullanıcının özetini kırmak istiyor. En kötü durumda ne kadar sürer?

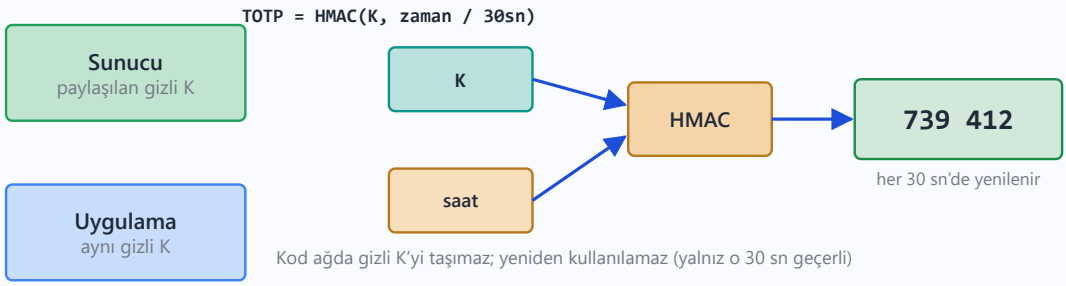
1. **Olası parola sayısı.** Her konum için 26 seçenek vardır: $26^8 = 208.827.064.576 \approx 2{,}09 \times 10^{11}$.
2. **Düz SHA-256.** Saniyede 10^{10} deneme yapılırsa: $2{,}09 \times 10^{11} / 10^{10} \approx 21$ saniye.
3. **Argon2id.** Her deneme pahalı olduğu için saniyede yalnız 10^4 deneme yapılabilsin: $2{,}09 \times 10^{11} / 10^4 \approx 2{,}09 \times 10^7$ saniye ≈ 242 gün. Yavaş özet her denemeyi 10^6 kat pahalılaştırmıştır.
4. **Daha uzun parola.** 10 karakter ve 62 simge (büyük, küçük harf ve rakam) ile uzay $62^{10} \approx 8{,}4 \times 10^{17}$ olur. Düz SHA-256'da bile yaklaşık 2,66 yıl, Argon2id ile yaklaşık 2,66 milyon yıl sürer.

Bütün değerler Python ile hesaplandı.

Bu hesapta tuz nerede? Tek bir kullanıcının süresini tuz değiştirmez. Tuzun etkisi, saldırgan bütün kullanıcıları kırmak istediğinde ortaya çıkar. Tuzsuz düzende saldırgan her adayın özetini bir kez hesaplar ve bir milyon kullanıcının özetiyle birden karşılaştırır: iş, tek kullanıcınınkiyle aynıdır. Tuzlu düzende her kullanıcı için baştan başlamak zorundadır. Bir milyon kullanıcı ve düz SHA-256 ile 21 saniyelik iş $10^6 \times 21$ saniyeye, yani yaklaşık 242 güne çıkar. Tuz işi kullanıcı sayısıyla çarpar; yavaşlık her denemenin fiyatını artırır. İkisi birbirinin yerini tutmaz.

13.5.4 MFA ve TOTP

Parolanın yanına eklenen en yaygın ikinci etken telefondur. Telefon, ya sunucudan gelen bir kodu alır ya da kodu kendisi üretir (Şekil 13.30).



Şekil 13.30: TOTP: sunucu ve uygulama aynı gizli K'yı ve aynı saati bilir; kod HMAC(K, zaman / 30 s)'den türetilir.

SMS kodu

: Yaygındır ama zayıftır. Saldırgan operatörü kandırıp numarayı kendi SIM kartına taşıyabilir (SIM takası) ya da telefon ağının SS7 protokolündeki açıklarla iletileri yönlendirebilir. Yine de hiç ikinci etken olmamasından iyidir.

TOTP üreteci uygulaması

: **Zaman tabanlı tek kullanımlık parola** (*time-based one-time password*, TOTP, RFC 6238). Kayıt sırasında sunucu ile uygulama, çoğunlukla bir QR kod aracılığıyla, gizli bir anahtar paylaşır. Uygulama bu anahtar ve saatten her 30 saniyede yeni bir kod üretir. Çevrim dışı çalışır. Ama oltalamaya açıktır: kullanıcı sahte bir siteye kodu da yazabilir.

Anlık bildirim onayı

: Kullanıcı telefona gelen giriş onayı bildirimine dokunur. Kolaydır, ama *onay yorgunluğu* saldırısına açıktır: saldırgan art arda onlarca istek gönderir ve kullanıcı sonunda bıkıp birini onaylar. Bu yüzden ekrandaki bir sayıyı telefona yazdıran *numara eşleştirme* eklenir.

TOTP kodunun ağda gizli anahtarı taşımadığına dikkat edilmelidir. Kod, anahtarın ve zamanın tek yönlü bir fonksiyonudur. Saldırgan bir kodu görse bile ondan anahtarı çıkaramaz. Kod da yalnız kendi 30 saniyelik penceresinde geçerlidir.

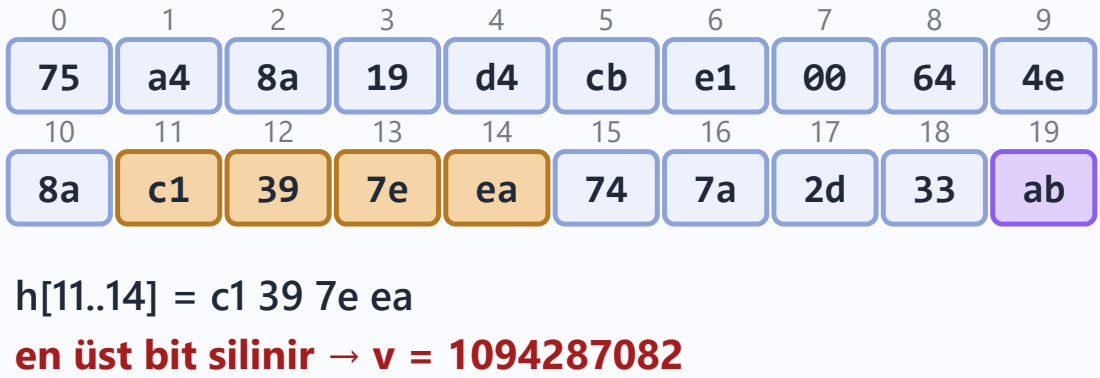
13.5.5 Kod izleme: TOTP kodu üretimi

Telefon ile sunucunun aynı kodu nasıl ürettiği sunumdaki C fonksiyonunda görülür. İzlemede RFC 6238'in test anahtarı kullanılır: ASCII “12345678901234567890” dizgisinin 20 baytı.

```
uint32_t totp(uint8_t *K, uint64_t t) {
    uint64_t T = t / 30;    /* 30 s */
    uint8_t h[20];
    hmac_sha1(K, T, h);
    int o = h[19] & 0x0F;   /* bağıl konum */
    uint32_t v = (h[o] & 0x7F) << 24
               | h[o+1] << 16
               | h[o+2] << 8
               | h[o+3];
    return v % 1000000;     /* 6 hane */
}
```

Fonksiyon dört adımda çalışır.

1. **Zaman adımı.** Unix zamanı t (1970'ten bu yana geçen saniye) 30'a bölünür: `T = t / 30`. Tam sayı bölmesi olduğu için aynı 30 saniyelik penceredeki her an aynı T'yi verir. 0–29 saniye T = 0'a, 30–59 saniye T = 1'e, 60–89 saniye T = 2'ye düşer. t = 59 için T = 1'dir.
2. **HMAC.** T, 8 baytlık büyük uçlu bir sayı olarak K ile birlikte HMAC-SHA1'den geçirilir. Sonuç 20 baytlık bir özettir: `75 a4 8a 19 d4 cb e1 00 64 4e 8a c1 39 7e ea 74 7a 2d 33 ab`. K'yı bilmeyen biri bu değeri üretemez.
3. **Dinamik kesme.** 20 bayttan 4'ünü seçmek gerekir. Seçimin yerini özetin kendisi belirler: son baytın alt yarısı bağıl konumdur. h[19] = 0xab'dir; 0xab & 0x0F = 0xb = 11. Dolayısıyla o = 11.
4. **Kod.** h[11] ... h[14] baytları, yani c1 39 7e ea, 32 bitlik bir sayıya birleştirilir. İlk baytın en üst biti `& 0x7F` ile silinir (0xc1 → 0x41). Bunun nedeni, sayının işaretli yorumlandığı sistemlerde negatif çıkmasını önlemektir. v = 0x41397eea = 1094287082'dir. Son altı hane v mod 1.000.000 = 287082 olur.



Şekil 13.31: h[19] = 0xab bağıl konumu 11 seçer; h[11..14] = c1 39 7e ea, üst bit silinince v = 1094287082.

Telefon 287082'yi gösterir; sunucu aynı hesabı yapıp kullanıcının yazdığı değerle karşılaştırır. 30 saniye sonra durum tamamen değişir (Tablo 13.14). t = 75'te T = 2 olur. Yalnız bir artan T, tamamen farklı bir özet verir. Bu kez son bayt 0x44'tür, bağıl konum 4'tür ve kod 359152 olur.

t	T	h[19]	o	h[o..o+3]	v	Kod
59	1	0xab	11	c1 39 7e ea	1094287082	287082
75	2	0x44	4	08 2f ef 30	137359152	359152

Tablo 13.14: İki zaman anında TOTP hesabı. Değerler Python'un `hmac` kütüphanesiyle yeniden hesaplandı; 8 haneli sonuç (94287082) RFC 6238'in test vektörüyle aynıdır.

Telefonun saati sunucununkinden birkaç saniye sapabilir. Sunucular bu yüzden genellikle bir önceki ve bir sonraki adımın kodlarını da kabul eder: T − 1, T ve T + 1. Bu, kodun yaklaşık 90 saniyelik bir pencerede geçerli olması demektir. Rastgele tahminle doğru kodu bulma olasılığı her denemede $3 / 10^6$'dır. Sunucu deneme sayısını sınırladığı sürece bu olasılık ihmal edilebilir.

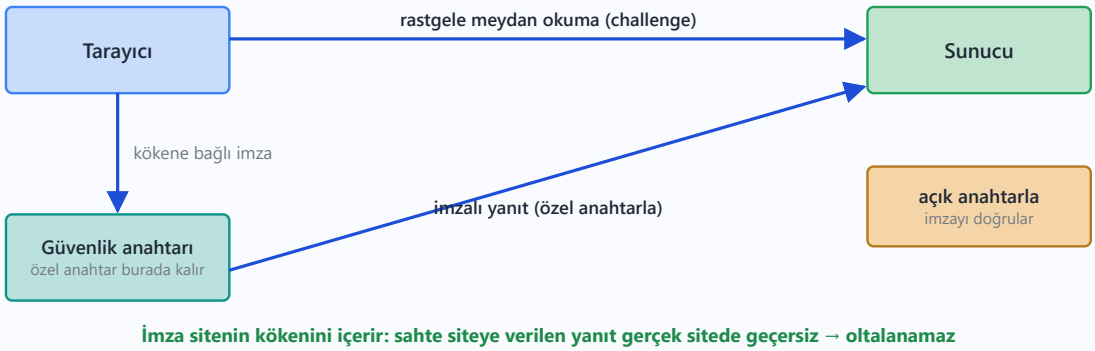
Örnek 13.13

t = 1.700.000.000 anında T kaçtır? Aynı kod hangi aralıkta geçerlidir? Sunucu T − 1 … T + 1'i kabul ediyorsa, şu anda girilen bir kod hangi aralıkta üretilmiş olabilir?

1. 1.700.000.000 / 30 = 56.666.666,67; tam sayı kısmı T = 56.666.666.
2. Pencere T · 30 = 1.699.999.980'de başlar ve 30 saniye sürer: 1.699.999.980 … 1.700.000.009.
3. T − 1'in başı 1.699.999.950, T + 1'in sonu 1.700.000.039'dur. Kabul edilen pencere 90 saniyedir.

13.5.6 FIDO2, WebAuthn ve geçiş anahtarları

TOTP parolanın çalınmasına karşı güçlüdür, ama oltalamaya karşı zayıftır. Kullanıcı sahte bir giriş sayfasına hem parolasını hem de o anki kodu yazar. Saldırgan bu ikisini birkaç saniye içinde gerçek siteye iletir ve kod henüz geçerliyken içeri girer. Sorunun kökü, kullanıcının hangi siteye kod verdiğini denetleyen bir mekanizmanın olmamasıdır. FIDO2 ve onun web arayüzü WebAuthn bu açığı kapatır (Şekil 13.32).



Şekil 13.32: FIDO2: güvenlik anahtarı sunucunun rastgele sınamasını, sitenin kökenine bağlı olarak imzalar; sunucu imzayı açık anahtarla doğrular.

Düzen açık anahtarlı kriptografiye dayanır. Kullanıcı bir siteye kaydolurken aygıtı o site için yeni bir anahtar çifti üretir. Özel anahtar aygıtın güvenli donanımından, yani TPM'den, Secure Enclave'den ya da USB güvenlik anahtarından hiç çıkmaz. Sunucuya yalnız açık anahtar gönderilir. Girişte sunucu rastgele bir sınama (*challenge*) gönderir. Tarayıcı bu sınamaya bağlandığı sitenin **kökenini** (*origin*), yani protokol, alan adı ve port üçlüsünü ekler. Aygıt bu ikisini özel anahtarla imzalar ve sunucu imzayı saklı açık anahtarla doğrular.

Oltalamaya karşı direnç bu kökenden gelir. Kullanıcı `ornek-giris.com` gibi sahte bir siteye girerse tarayıcı imzaya o sahte kökeni koyar. Ayrıca aygıt, sahte site için kayıtlı bir anahtar bulamaz. Saldırgan yanıtı gerçek siteye iletse bile köken eşleşmediği için imza geçersizdir. Kullanıcının dikkatsizliği sunucu değiştirmez; denetimi tarayıcı yapar.

Düzenin öteki kazançları da önemlidir. Sunucu yalnız açık anahtarları saklar; veritabanı sızsa bile bu anahtarlarla kimse giriş yapamaz. Paylaşılan bir sır yoktur ve aynı anahtar başka sitede yeniden kullanılmaz, çünkü her siteye ayrı çift üretilir. **Geçiş anahtarları** (*passkeys*), FIDO2 kimlik bilgilerinin kullanıcının aygıtları arasında güvenli biçimde eşitlenebilen biçimidir. Parolaların yerini almak üzere tasarlanmıştır.

13.5.7 Biyometrik kimlik doğrulama

Biyometri, kişinin fiziksel ya da davranışsal bir özelliğini ölçer: parmak izi, yüz, iris, ses. İlk bakışta ideal görünür: unutulmaz, kaybolmaz, devredilemez. Ancak her ölçüm biraz farklıdır ve sistem ölçülen değeri kayıtlı şablonla bir eşik değerine göre karşılaştırır. Bu yüzden iki tür hata vardır (Tablo 13.15).

Ölçüt	Açıklama
Yanlış kabul oranı (FAR)	Yabancıyı kabul etme oranı: güvenlik riski
Yanlış ret oranı (FRR)	Gerçek kullanıcıyı reddetme oranı: kullanılabilirlik sorunu
Canlılık testi	Fotoğraf, maske ya da parmak kalıbına karşı denetim

Ölçüt	Açıklama
İptal edilemezlik	Parmak izi sızarsa değiştirilemez; parola değiştirilebilir

Tablo 13.15: Biyometrik sistemlerin ölçütleri.

FAR ile FRR birbirine bağlıdır. Eşik sıkılaştırılırsa yabancılar daha az kabul edilir, ama gerçek kullanıcı da daha sık reddedilir. Eşik gevşetilirse tersi olur. Küçük bir FAR bile büyük sayılarda önem kazanır. FAR 1/100.000 olan bir sistemde bin farklı yabancının denemesinden en az birinin kabul edilme olasılığı $1 - (1 - 10^{-5})^{1000} \approx$ %1'dir.

Biyometrinin temel sınırı şudur: bir kimlik için iyidir, bir sır için kötüdür. Yüz ve parmak izi gizli değildir; insanlar her gün dokundukları yüzeylere parmak izi, sosyal medyaya fotoğraf bırakır. Bu yüzden doğru kullanım yerel kullanımdır. Biyometri, aygıtın içindeki güvenli donanımın kilidini açan bir etken olarak çalışır; örneğin Face ID Secure Enclave'deki anahtarlara erişimi açar. Ham şablon aygıttan hiç çıkmaz. Tek başına uzaktan kimlik doğrulama için uygun değildir ve genellikle elde tutulan bir aygıtla birleştirilir. Parmak uzunluğunu ölçen ilk aygıtlar bugün tarihseldir. Bugünkü sistemler parmak izi çizgilerinin ayrıntı noktalarını (*minutiae*), üç boyutlu yüz şeklini ve iris desenini kullanır. Donanım destekli canlılık testi de şarttır.

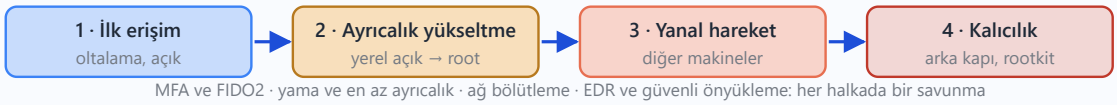
13.5.8 Saldırganlar sisteme nasıl girer?

Kimlik doğrulamanın zayıf noktaları, saldırganların en sık kullandığı giriş yollarıdır (Tablo 13.16).

Yol	Açıklama	Önlem
Oltalama (*phishing*)	Sahte giriş sayfasıyla parola toplama	FIDO2, kullanıcı eğitimi
Kimlik bilgisi doldurma	Sızan parolaları başka sitelerde deneme	Benzersiz parola, MFA
Kaba kuvvet / sözlük	Zayıf parolayı tahmin etme	Yavaş özet, hız sınırı, kilitleme
Yama yapılmamış açık	Bilinen bir zafiyeti kullanma	Hızlı yama yönetimi
Sosyal mühendislik	Yardım masasını kandırma	Süreç ve doğrulama
Varsayılan ya da zayıf kimlik	admin/admin, açık bırakılmış port	Sağlamlaştırma (*hardening*)

Tablo 13.16: Yaygın ilk erişim yolları ve önlemleri.

Gerçek bir saldırı nadiren tek bir adımdır; genellikle bir zincirdir (Şekil 13.33). Saldırgan önce oltalama ya da bir açıkla *ilk erişimi* elde eder. Bu erişim çoğunlukla sıradan bir kullanıcı hesabıdır. Ardından yerel bir açıkla *ayrıcalık yükseltme* yapar ve root olur. Sonra aynı ağdaki öteki makinelere *yanal hareketle* yayılır. Sonunda bir arka kapı ya da rootkit kurarak *kalıcılık* sağlar: makine yeniden başlatılsa da içeride kalır.



Şekil 13.33: Saldırı zinciri ve her halkadaki savunma: MFA ve FIDO2, yama ve en az ayrıcalık, ağ bölütleme, EDR ve güvenli önyükleme.

Zincir görüşü savunmanın da katmanlı olması gerektiğini bir kez daha gösterir. MFA ilk erişimi zorlaştırır. Yama ve en az ayrıcalık yükseltmeyi engeller. Ağ bölütlemesi yanal hareketi durdurur. Uç nokta izleme ve güvenli önyükleme kalıcılığı yakalar. Tek bir halkanın kopması bütün sistemin ele geçirilmesine yol açmamalıdır.

13.6 Yazılım açıkları

Kimlik doğrulama ve erişim denetimi kusursuz olsa bile programlardaki hatalar saldırganlara yeni kapılar açar. Bir hata, programın tasarımcısının hiç öngörmediği bir şey yapmasına yol açıyorsa güvenlik açığına dönüşür. En tehlikelileri, saldırganın kendi seçtiği kodu çalıştırmasına izin veren bellek hatalarıdır. Bu kesim bunların en klasiğini, yığın tampon taşmasını, ona karşı geliştirilen savunmaları ve bir yarış durumu açığı olan TOCTOU'yu ele alır.

13.6.1 Yığın tampon taşması

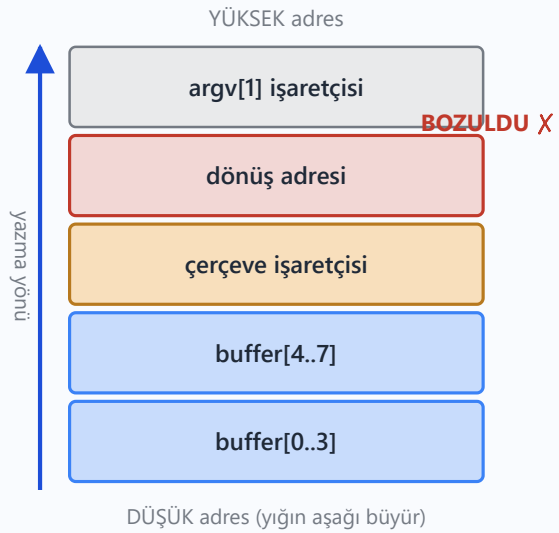
C'de bir fonksiyon çağrıldığında yığında onun için bir **çerçeve** açılır. Çerçevede fonksiyonun yerel değişkenleri, çağıranın kayıtlı çerçeve işaretçisi ve **dönüş adresi** bulunur. Dönüş adresi, fonksiyon bittiğinde işlemcinin yürütmeye devam edeceği yerdir. x86 ailesinde yığın yüksek adreslerden düşük adreslere doğru büyür. Bu yüzden önce çağrılan fonksiyonun dönüş adresi yığına konur, yerel diziler ondan daha düşük adreslerde durur.

Tanım 13.18

Tampon taşması (*buffer overflow*), bir tampona boyundan fazla veri yazılması ve fazla baytların bitişik belleğin üzerine taşmasıdır. Yığındaki bir tampon taşarsa en tehlikeli kurban dönüş adresidir.

```
/* küçük, kasıtlı olarak hatalı örnek */
void A(char *in) {
    char buffer[8];
    strcpy(buffer, in);   // sınır denetimi YOK
    return;
}
A(argv[1]);               // in: kullanıcı girdisi
```

Bu küçük ve kasıtlı olarak hatalı fonksiyon, kullanıcıdan gelen bir dizgiyi 8 baytlık bir diziye kopyalar. `strcpy` hedefin boyunu bilmez; kaynaktaki sonlandırıcı `\0`'a kadar kopyalamayı sürdürür. Girdi 7 karakter ya da daha kısa olduğu sürece sorun yoktur: sonlandırıcıyla birlikte en çok 8 bayt yazılır ve dizi taşmaz. Girdi daha uzunsa fazla baytlar dizinin üstündeki hücrelere yazılır. Dizi düşük adreste, dönüş adresi yüksek adreste durduğu ve kopyalama artan adres yönünde ilerlediği için taşma doğrudan dönüş adresine doğru gider (Şekil 13.34).



Şekil 13.34: Yığın çerçevesi. Yazma yönü yukarı doğrudur; uzun bir girdi önce çerçeve işaretçisini, sonra dönüş adresini bozar.

Fonksiyon `return` ettiğinde işlemci yığındaki dönüş adresini okur ve oraya atlar. Adres rastgele baytlarla bozulmuşsa program büyük olasılıkla bir segmentasyon hatasıyla (SIGSEGV) çöker. Ama girdiyi seçen bir saldırgan, dönüş adresinin yerine kendi seçtiği bir adresi koyabilir. O zaman yürütme akışı saldırganın istediği yere gider. Klasik saldırıda bu adres, saldırganın aynı girdiyle tampona yerleştirdiği makine koduna işaret eder. Program setuid root ise ya da bir ağ hizmetiyse, saldırgan root haklarıyla bir kabuk elde eder. Aleph One'ın 1996'da yayımladığı “Smashing the Stack for Fun and Profit” makalesi bu tekniği ayrıntılı anlatarak bir saldırı kuşağının başlangıcı oldu.

Örnek 13.14

64 bitlik bir makinede `char ad[8]` dizisine `strcpy` ile “Giresun-Ordu-Samsun” kopyalanıyor. Kanarya ve dolgu yoktur: çerçevede 8 baytlık `ad`, 8 baytlık kayıtlı çerçeve işaretçisi (rbp) ve 8 baytlık dönüş adresi vardır. Kaç bayt yazılır, kaçı taşar, dönüş adresinin kaç baytı bozulur?

1. Dizgi 19 karakterdir. `strcpy` sonlandırıcıyla birlikte 19 + 1 = 20 bayt yazar.
2. İlk 8 bayt “Giresun-” diziye tam sığar. Henüz `\0` gelmediği için kopyalama sürer.
3. Sonraki 8 bayt “Ordu-Sam” kayıtlı rbp'nin üzerine yazılır.

4. Kalan “sun” ve `\0` dönüş adresinin ilk dört baytına yazılır: 73 75 6e 00. Makine küçük uçlu olduğu için bu baytlar adresin düşük dört baytıdır; adres ...006e7573 olur.

Toplam 20 bayttan 20 − 8 = 12'si taşmıştır; dönüş adresinin 4 baytı bozulmuştur. Fonksiyon dönerken işlemci bu geçersiz adrese atlar ve program çöker. Değerler Python ile doğrulandı.

Düzeltme, kopyalamayı hedefin boyuyla sınırlamaktır. `snprintf` en çok `sizeof buffer - 1` karakter kopyalar ve her zaman bir sonlandırıcı ekler.

```
void A(const char *in) {
    char buffer[8];
    snprintf(buffer, sizeof buffer, "%s", in);   /* en çok 7 karakter + '\0' */
}
```

Sık yapılan hata

Sonlandırıcıyı saymamak. 8 baytlık bir diziye 8 karakterlik bir dizgi sığmaz: `strcpy` 9 bayt yazar ve bir bayt taşar. Taşan tek bir bayt bile, örneğin kayıtlı çerçeve işaretçisinin en düşük baytını sıfırlayarak, istismar edilebilir. Bu tür hatalara “birer fazla” (*off-by-one*) hatası denir.

13.6.2 Tampon taşmasına karşı savunmalar

Tampon taşması kırk yılı aşkın bir süredir bilinir ve ona karşı birçok savunma katmanı geliştirilmiştir. Her savunma saldırının farklı bir adımını hedefler.

Yığın kanaryası
: Derleyici, dönüş adresinin hemen altına süreç başında rastgele seçilen bir değer koyar. Fonksiyon dönmeden önce bu değerin bozulup bozulmadığına bakar. Dönüş adresine ulaşmak isteyen bir taşma önce kanaryayı ezmek zorundadır. Bozulma görülürse program durdurulur. GCC'de bu özellik yığın parçalama koruması (SSP) adıyla bilinir.

DEP / NX
: Veri sayfaları sayfa tablosunda yürütülemez olarak işaretlenir (x86-64'teki NX biti). Yığına enjekte edilen kod, işlemci oraya atlasa bile çalışmaz; bir sayfa hatası oluşur. Bir sayfanın hem yazılabilir hem çalıştırılabilir olmaması kuralına W^X da denir.

ASLR
: Adres alanı yerleşiminin rastgeleleştirilmesi (*address space layout randomization*): yığın, heap ve kütüphaneler her çalıştırmada farklı adreslere yüklenir. Saldırgan hangi adrese atlayacağını bilemez. Programın kendisi de konumdan bağımsız (PIE) derlenirse onun adresi de rastgele olur. Linux'ta 64 bitlik süreçlerde kütüphane yerleşimi 28 bitlik bir rastgelelik taşır; yaklaşık 268 milyon olasılık.

CFI

Denetim akışı bütünlüğü (*control-flow integrity*): dolaylı atlamaların ve çağrıların yalnız derleyicinin önceden belirlediği geçerli hedeflere gitmesine izin verilir.

Gölge yığın

Dönüş adresinin bir kopyası, programın sıradan yazma komutlarıyla değiştiremeyeceği ayrı bir yığında tutulur. Dönüşte iki adres karşılaştırılır. Intel CET ve ARM GCS bunu donanımda sağlar. ARM'ın işaretçi doğrulama (PAC) ve dal hedefi belirleme (BTI) özellikleri ayrı ama benzer amaçlı korumalardır.

Bellek güvenli dil

Rust, Go, Java ve C# gibi dillerde dizi sınırları ve nesne ömürleri derleyici ya da çalışma zamanında denetlenir. Tampon taşması bu dillerde baştan oluşamaz. En köklü çözüm budur.

Savunmalar geliştikçe saldırılar da değişti. NX, yığına enjekte edilen kodun çalışmasını engelleyince saldırganlar yeni kod getirmeyi bıraktı. Bunun yerine bellekte zaten var olan kodu kullanmaya başladılar. **return-to-libc** saldırısında dönüş adresi libc'deki `system()` fonksiyonuna yönlendirilir ve ona "/bin/sh" argümanı verilir. Daha genel biçimi **dönüş yönelimli programlamadır** (*return-oriented programming*, ROP). Saldırgan, programın ve kütüphanelerin içinde `ret` komutuyla biten küçük kod parçaları (*gadget*) bulur. Yığına bu parçaların adreslerini art arda yazar. Her parça birkaç komut çalıştırır ve `ret` ile bir sonrakine geçer. Yeterince parçayla istenen her hesap yapılabilir. Bu kodların hepsi zaten çalıştırılabilir sayfalarda durduğu için NX bu saldırıyı durduramaz. ASLR parçaların adreslerini gizler, CFI ve gölge yığın da dönüşlerin beklenmedik yerlere gitmesini engeller. Bu yüzden savunmalar birlikte kullanılır; tek başına hiçbiri yeterli değildir.

13.6.3 Kod izleme: yığın kanaryası

Kanaryanın nasıl çalıştığını bayt bayt izleyelim. Aşağıdaki kodda kanarya satırlarını programcı değil derleyici ekler; gerçek derlenmiş kodda bu satırlar fonksiyonun giriş ve çıkış komutlarıdır.

```
/* kanarya satırlarını derleyici ekler */
void kopyala(const char *girdi) {
    long kanarya = __stack_chk_guard;
    char buf[8];
    strcpy(buf, girdi);
    if (kanarya != __stack_chk_guard)
        __stack_chk_fail();
}
/* girdi: "AAAAAAAABBBBBBBB" */
```

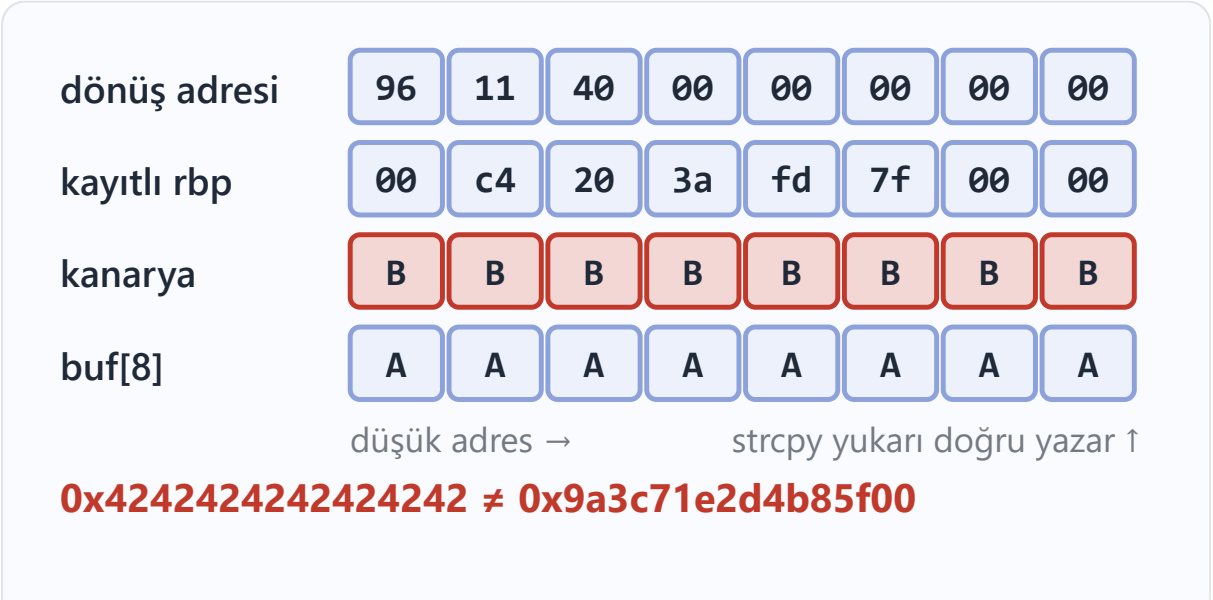
Fonksiyon "AAAAAAAABBBBBBBB" girdisiyle çağrılır: 16 karakter ve bir sonlandırıcı. `buf` ise 8 bayttır. Çerçevede düşük adresten yükseğe doğru `buf`, kanarya, kayıtlı rbp ve dönüş adresi bulunur. İzleme Tablo 13.17 adımlarıyla ilerler.

Adım	buf[8]	kanarya	kayıtlı rbp	dönüş adresi
çağrı	(belirsiz)	(belirsiz)	f0 c4 20 3a fd 7f 00 00	96 11 40 00 00 00 00 00

Adım	buf[8]	kanarya	kayıtlı rbp	dönüş adresi
giriş kodu	(belirsiz)	00 5f b8 d4 e2 71 3c 9a	aynı	aynı
ilk 8 bayt	A A A A A A A A	aynı	aynı	aynı
sonraki 8 bayt	A × 8	B B B B B B B B	aynı	aynı
`\0`	A × 8	B × 8	**00** c4 20 3a fd 7f 00 00	96 11 40 00 00 00 00 00
çıkış denetimi	0x4242424242424242 ≠ 0x9a3c71e2d4b85f00 → `__stack_chk_fail()`			

Tablo 13.17: `kopyala("AAAAAAAABBBBBBBB")` çağrısında çerçevenin durumu. Baytlar bellekteki sırayla (düşük adresten yükseğe) yazılmıştır. Değerler Python ile doğrulandı.

1. **Çağrı.** `call` komutu dönüş adresini (0x401196) yığına koyar. Fonksiyonun giriş kodu çağıranın çerçeve işaretçisini (0x7ffd3a20c4f0) kaydeder. Küçük uçlu düzende adresin en düşük baytı önce gelir: 0x401196, bellekte 96 11 40 00 … diye görünür.
2. **Kanarya.** Derleyicinin eklediği giriş kodu, süreç başında rastgele seçilmiş `__stack_chk_guard` değerini `buf` ile kayıtlı rbp arasına koyar. Bu örnekte değer 0x9a3c71e2d4b85f00'dır. İlk baytının, yani bellekteki en düşük baytının 0x00 olması bilinçli bir seçimdir. Dizgi işlevleri bu baytta durduğu için kanarya bir dizgi okumasıyla sızdırılamaz ve `strcpy` ile aynen yazılamaz. Geri kalan 56 bit rastgeledir: 2^{56} olasılık.
3. **İlk 8 bayt.** `strcpy` "AAAAAAAA"yı `buf` 'a yazar. Henüz bir sorun yoktur.
4. **Taşma.** "BBBBBBBB" `buf` 'ın dışına, kanaryanın üzerine yazılır. Kanarya artık 0x4242424242424242'dir (B'nin ASCII kodu 0x42).
5. **Sonlandırıcı.** `\0` kayıtlı rbp'nin ilk baytına yazılır: f0 → 00. Girdi biraz daha uzun olsaydı sıra dönüş adresine gelirdi. Ama dönüş adresine ulaşmanın tek yolu kanaryanın üzerinden geçmektir.
6. **Çıkış denetimi.** Fonksiyon dönmeden önce çıkış kodu yığındaki kanaryayı süreç başındaki değerle karşılaştırır. İkisi farklıdır.
7. **Durdurma.** `__stack_chk_fail` "*** stack smashing detected ***: terminated" iletisini yazar ve süreci SIGABRT ile sonlandırır. Bozulmuş çerçeveye hiç dönülmez.



Şekil 13.35: Çıkış denetimi: yığındaki kanarya (B'ler) süreç başındaki değerle eşleşmiyor.

Kanaryanın kazancı, bir kod yürütme saldırısını bir çökmeye dönüştürmesidir. Bu hâlâ bir kayıptır, çünkü hizmet durur ve erişilebilirlik bozulur. Ama saldırgan kodunu çalıştıramaz. Kanaryanın sınırları da vardır. Saldırgan başka bir hatayla kanaryanın değerini okuyabilirse onu aynen geri yazabilir. Taşma dönüş adresi yerine bir fonksiyon işaretçisini bozuyorsa kanarya hiç devreye girmez.

13.6.4 Diğer bellek ve enjeksiyon hataları

Tampon taşması, geniş bir hata ailesinin yalnız bir üyesidir (Tablo 13.18).

Açık	Neden tehlikeli	Önlem
Biçim dizgisi	`printf(user)` : %x ile belleği okur, %n ile yazar	`printf("%s", user)`
Tam sayı taşması	`len = a + b` sarması küçük ayırmaya yol açar	Denetimli aritmetik, boyut tipleri
Serbest sonrası kullanım	Serbest bırakılmış belleğe erişim (UAF)	Sahiplik (Rust), akıllı işaretçi, karantina
Çifte serbest bırakma	Aynı blok iki kez `free` edilir; heap bozulur	`free` sonrası NULL, güçlendirilmiş ayırıcı
Sınır dışı okuma	Heartbleed gibi bellek sızıntısı	Sınır denetimi, bulanıklaştırma
Komut/SQL enjeksiyonu	Girdi koda ya da sorguya karışır	Parametreli sorgu, kaçış, izin listesi

Tablo 13.18: Bellek ve enjeksiyon hataları.

Biçim dizgisi hatasında programcı kullanıcının girdisini doğrudan `printf`'e biçim dizgisi olarak verir. Kullanıcı girdiye `%x` yazarsa `printf` olmayan argümanları yığından okur. Böylece belleğin içeriği ekrana dökülür. `%n` ise o ana kadar yazılan karakter sayısını bir adrese *yazar*; saldırgan bununla belleği değiştirebilir. **Tam sayı taşmasında** iki uzunluğun toplamı 32 bitlik bir değişkeni aşar ve küçük bir sayıya sarar. Program bu küçük değerle bellek ayırır, sonra büyük veriyi bu alana kopyalar ve heap'te bir taşma oluşur.

Serbest sonrası kullanımda (*use-after-free*) program serbest bıraktığı bir bloğa eski işaretçi üzerinden erişmeyi sürdürür. Ayırıcı bu arada bloğu başka bir nesneye vermişse, saldırgan o nesnenin içeriğini denetleyerek programın eski nesneyi nasıl yorumlayacağını belirler. Tarayıcı açıklarının büyük bölümü bu türdendir. **Sınır dışı okumanın** ünlü örneği, 2014'te OpenSSL'de bulunan Heartbleed'dir. Sunucu, istemcinin bildirdiği uzunluğu denetlemeden o kadar baytı yanıta kopyalıyordu. Her istekte belleğinden 64 KiB'a kadar veri, içinde özel anahtarlar ve parolalar olabilecek biçimde sızıyordu.

Bu hataların ortak kökü, verinin kodla ya da denetim bilgisiyle karışması ve güvenilmeyen girdinin denetimsiz kullanılmasıdır. Biçim dizgisinde veri biçim komutu olarak yorumlanır. SQL enjeksiyonunda kullanıcının yazdığı tırnak işareti sorgunun yapısını değiştirir. Savunma da ortaktır: girdi doğrulanır, veri koddan ayrılır, bellek güvenli diller kullanılır. Programa rastgele bozuk girdiler veren **bulanıklaştırma** (*fuzzing*) araçları da hataları saldırganlardan önce bulur.

13.6.5 TOCTOU yarış durumu

Bazı güvenlik açıkları bellek hatası değil, zamanlama hatasıdır. Bir program bir koşulu denetler ve bir süre sonra bu denetime dayanarak bir işlem yapar. İki an arasında koşul değişebiliyorsa program artık geçersiz bir bilgiye göre davranır.

Tanım 13.19

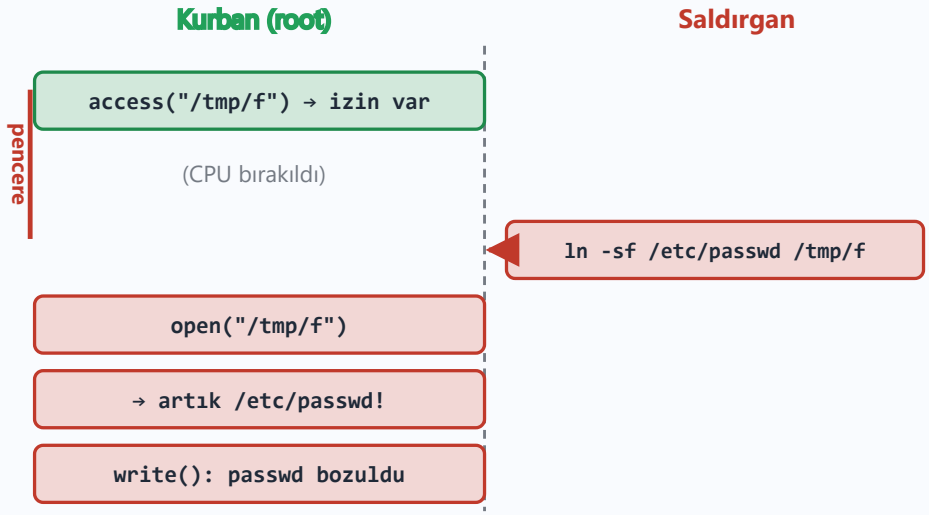
TOCTOU (*time of check to time of use*, kontrol zamanı ile kullanım zamanı) açığı bir yarış durumudur. Saldırgan, kaynağın denetlendiği an ile kullanıldığı an arasında kaynağı değiştirebilir.

```
/* TOCTOU: setuid root yardımcı */
if (access(path, W_OK) == 0) {   // KONTROL
    fd = open(path, O_WRONLY);     // KULLANIM
    write(fd, data, n);
}
```

Kod, setuid root bir yardımcı programın parçasıdır. Program, kendisini çalıştıran kullanıcının yazma izni olmayan bir dosyaya root haklarıyla yazmasını önlemek ister. Bunun için `access` çağrısını kullanır. `access` izni etkin kimliğe değil *gerçek* kimliğe göre denetler: “ali bu dosyaya yazabilir mi?” sorusunu yanıtlar. Cevap olumluysa program dosyayı `open` ile açar. `open` ise etkin kimliği, yani root'u kullanır.

Sorun, iki çağrının dosya adını ayrı ayrı çözmesidir (Şekil 13.36).

1. **Kontrol.** `/tmp/f` sıradan, Ali'ye ait bir dosyadır. `access` izin olduğunu söyler.
2. **Pencere.** Kontrol ile kullanım arasında küçük bir zaman aralığı vardır. Çizelgeleyici tam bu sırada süreci CPU'dan alabilir.
3. **Saldırı.** Saldırgan bu aralıkta `/tmp/f`'yi siler ve yerine `/etc/passwd`'ye işaret eden bir sembolik bağ koyar: `ln -sf /etc/passwd /tmp/f`. Dosya adı aynıdır, ama hedef değişmiştir.
4. **Kullanım.** `open("/tmp/f")` adı yeniden çözer ve bağı izleyerek `/etc/passwd`'yi root haklarıyla açar. `write` bu kritik dosyayı bozar.



Şekil 13.36: TOCTOU zaman çizelgesi: kontrol ile kullanım arasındaki pencerede saldırgan dosya adını sembolik bağla değiştirir.

Pencere çok kısa olabilir, ama saldırgan denemeyi binlerce kez tekrarlar. Çok çekirdekli bir makinede kendi sürecini ayrı bir çekirdekte sürekli bağ değiştirecek biçimde çalıştırarak şansını artırır. Bir kez tutturması yeterlidir.

Çözümün ilkesi, adı iki kez çözmemektir. Dosya bir kez açılır ve sonraki bütün denetimler aynı dosya tanımlayıcısı üzerinde yapılır. Tanımlayıcı açıldığı anda hangi dosyaya bağlandıysa ona bağlı kalır; adın sonradan değişmesi onu etkilemez. Ayrıca program, açmadan önce ayrıcalığını kullanıcının kimliğine düşürür. Böylece izin denetimini `access` değil, `open`'ın kendisi doğru kimlikle yapar. `O_NOFOLLOW` bayrağı ise son bileşen bir sembolik bağsa açmayı reddeder.

```
/* düzeltilmiş: ad bir kez çözülür, denetim aynı tanımlayıcı üzerinde yapılır */
uid_t ruid = getuid();
seteuid(ruid);                              /* önce ayrıcalık düşürülür */
int fd = open(path, O_WRONLY | O_NOFOLLOW);   /* izin denetimini çekirdek yapar */
if (fd >= 0) {
    struct stat st;
    if (fstat(fd, &st) == 0 && S_ISREG(st.st_mode))
        write(fd, data, n);
    close(fd);
}
```

Güvenli bir dizinde çalışan programlar `openat` ailesini kullanarak yolu dizin tanımlayıcısına göre çözer; bu da yolun ara bileşenlerinin değiştirilmesini engeller.

Sık yapılan hata

Pencereyi küçültmeyi çözüm saymak. `access` ile `open` art arda yazıldığı için pencere mikrosaniyelerle ölçülür; ama küçük bir pencere yalnız saldırının deneme sayısını artırır. Çözüm pencereyi kapatmaktır: denetim ve kullanım aynı nesne üzerinde, tek bir ad çözümlemesiyle yapılmalıdır.

13.7 İçeriden saldırılar

Saldırıların bir kısmı sistemi dışarıdan değil, içeriden gelir. Saldırgan sistemi geliştiren ya da işleten biridir: bir programcı, bir yönetici, bir yüklenici. Meşru erişimi olduğu için dış savunmaların hepsini zaten aşmıştır. Bu grubun üç klasik yöntemi vardır.

Mantık bombası (*logic bomb*)
: Belirli bir koşulda tetiklenen zararlı koddur. Koşul bir tarih olabilir ya da programcının adının bordrodan silinmesi gibi bir olay. Çoğu zaman hoşnutsuz bir çalışan koyar: kod her gün çalışır, programcının hâlâ şirkette olup olmadığına bakar ve değilse dosyaları siler.

Arka kapı (*back door*)
: Kimlik doğrulamayı atlayan gizli bir giriştir (Şekil 13.37). Arka kapılar tedarik zincirine de sokulabilir. 2024'te Linux dağıtımlarının kullandığı xz sıkıştırma kütüphanesinde bir arka kapı son anda fark edildi. Arka kapıyı yıllarca güven kazanmış bir katkıcı eklemişti; kapı SSH kimlik doğrulamasını etkiliyordu.

Giriş sahtekârlığı (*login spoofing*)
: Saldırgan, gerçek giriş ekranına benzeyen bir program çalıştırır ve terminalin başından ayrılır. Gelen kullanıcı adını ve parolasını yazar. Program bunları kaydeder, "parola yanlış" iletisi gösterir ve gerçek giriş ekranına devreder. Kullanıcı ikinci denemede girer ve hiçbir şeyden kuşkulanmaz.

(a) Normal kod

```
if (user_ok(name, pw))
grant_access();
else
deny();
```

(b) Arka kapılı kod

```
if (user_ok(name, pw)
|| eq(name, "zzz"))
grant_access();
else
deny();
```

"zzz" her zaman girer

Şekil 13.37: (a) Normal oturum açma kodu; (b) "zzz" kullanıcı adıyla her zaman giriş izni veren arka kapılı sürüm.

Giriş sahtekârlığına karşı **güvenli dikkat dizisi** (*secure attention key*) kullanılır. Windows'ta Ctrl+Alt+Del birleşimini hiçbir kullanıcı programı yakalayamaz; bu tuşlar her zaman doğrudan işletim sisteminin gerçek giriş ekranını çağırır. Kullanıcı giriş yapmadan önce bu tuşlara basmaya alıştırılırsa, ekranda sahte bir program çalışıyor olsa bile gerçek ekran öne gelir.

Arka kapılara karşı savunma kurumsal ve tekniktir. Kod incelemesinde her değişikliği en az bir başka kişi görür. Kritik değişiklikler için iki kişi onayı istenir. **Yeniden üretilebilir derleme** (*reproducible builds*) ise aynı kaynak koddan herkesin bayt bayt aynı ikili dosyayı elde etmesini sağlar. Böylece dağıtılan programın yayımlanan kaynaktan derlendiği bağımsız olarak doğrulanabilir. Ken Thompson'ın 1984'teki Turing Ödülü konuşması bu konunun sınırını gösterir. Thompson, kendini derlerken arka kapıyı yeniden ekleyen bir derleyicinin kaynak kodda hiçbir iz bırakmayacağını anlatır. Sonuç olarak her güven zinciri bir yerde, kendimiz yazmadığımız bir araca güvenmekle başlar.

13.8 Kötü amaçlı yazılımlar

Kötü amaçlı yazılım (*malware*), sistem sahibinin isteği dışında çalışan programların genel adıdır. Bu programlar zarar vermek, veri çalmak ya da kaynak kullanmak için tasarlanır. Önceki kesimlerdeki açıklar, bu programların sisteme girmek için kullandığı kapılardır.

13.8.1 Türler

Kötü amaçlı yazılımlar yayılma biçimlerine ve amaçlarına göre sınıflandırılır (Tablo 13.19).

Tür	Yayılma	Amaç / davranış
Virüs	Başka bir programa kendini ekler; kullanıcı o programı çalıştırınca bulaşır	Dosyaları bozar, yayılır
Solucan	Ağ üzerinden kendi kendine; kullanıcı gerekmez	Hızlı yayılır, kaynak tüketir
Truva atı	Yararlı görünen bir programın içinde	Arka kapı açar, veri çalar
Fidye yazılımı	Oltalama, açık, RDP	Dosyaları şifreler, fidye ister
Rootkit	Ayrıcalık kazanıldıktan sonra kurulur	Kendini ve öteki zararlıları gizler
Casus yazılım	Truva atı, paket yazılım, sahte eklenti	Veri toplar, uzağa gönderir
Bot / botnet	Solucan + arka kapı	Uzaktan komutla DDoS, spam, kazım

Tablo 13.19: Kötü amaçlı yazılım türleri.

Virüs ile solucan arasındaki fark yayılmanın kime bağlı olduğudur. **Virüs** bir taşıyıcıya ihtiyaç duyar: başka bir programa ya da belgeye yerleşir ve o dosya çalıştırıldığında ya da açıldığında etkinleşir. **Solucan** (*worm*) ise bağımsız bir programdır. Ağdaki yamalanmamış bir hizmeti kullanarak makineden makineye kendini kopyalar ve kullanıcının hiçbir şey yapmasına gerek kalmaz. Kasım 1988'deki

Morris solucanı, o günkü İnternet'e bağlı makinelerin önemli bir kısmını birkaç saatte etkiledi. 2017'deki WannaCry ise fidye yazılımı taşıyan bir solucandı ve Windows'un dosya paylaşım protokolündeki bir açıkla yayıldı.

Truva atı (*Trojan horse*) adını Truva savaşındaki tahta attan alır. Kullanıcının kendi isteğiyle kurduğu, yararlı görünen bir oyun, eklenti ya da "ücretsiz" bir yazılımdır. Kullanıcının haklarıyla çalıştığı için o kullanıcının erişebildiği her şeye erişir. **Casus yazılım** (*spyware*) kullanıcının etkinliklerini izler ve uzağa gönderir. Tarayıcının ana sayfasını ya da arama motorunu değiştirebilir, istenmeyen araç çubukları ekleyebilir, durdurulamayan reklam pencereleri açabilir.

Bugün sınırlar bulanıktır. Modern bir zararlı genellikle birkaç türü birleştirir: solucan gibi yayılır, rootkit ile gizlenir, bir botnet'e bağlanır ve sonunda fidye yazılımı yükünü çalıştırır.

13.8.2 Parazitik virüsler

Klasik virüslerin en yaygın türü, çalıştırılabilir dosyalara yerleşen **parazitik virüslerdir**. Virüs bir dosyaya bulaşırken iki iş yapar: kendi kodunu dosyaya ekler ve programın giriş noktasını değiştirir. Program çalıştırıldığında önce virüs çalışır, yeni dosyalara bulaşır, sonra asıl programa atlar. Kullanıcı programın normal çalıştığını görür ve bir şeyden kuşkulanmaz. Virüs kodunu dosyanın neresine koyduğuna göre üç türe ayrılır.

Başta yerleşik
: Virüs kodu programın başına eklenir ve giriş noktası virüse çevrilir.

Sonda yerleşik
: Virüs programın sonuna eklenir; programın başına virüse atlayan bir komut konur. En yaygın türdür, çünkü programın geri kalanını kaydırmak gerekmez.

Boşluklara yayılmış
: Çalıştırılabilir dosyalarda hizalama yüzünden kullanılmayan boşluklar bulunur. Virüs kendini parçalara bölüp bu boşluklara dağıtır. Dosya boyu değişmediği için tespit edilmesi zorlaşır (Şekil 13.38).



Şekil 13.38: Boşluklara yayılmış virüs: parçalar programın kullanılmayan alanlarına dağılır; dosya boyu değişmez.

Başka türler de vardır. *Eşlik eden virüs* asıl programa dokunmaz: MS-DOS'ta aynı adlı .com dosyası .exe'den önce çalıştığı için, prog.exe'nin yanına prog.com adıyla yerleşir. *Bellekte yerleşik virüsler* bellekte kalıp kesme vektörlerini ele geçirir. *Önyükleme sektörü virüsleri* işletim sisteminden önce çalışır. *Aygıt sürücüsü virüsleri* çekirdek kipinde çalışır. *Makro virüsleri* belgelerin içindeki makrolara, *kaynak kodu virüsleri* derlenmeden önce kaynak dosyalara yerleşir. Bunların çoğu bugün tarihseldir. Modern zararlılar betikleri, diske hiç dosya yazmayan bellek içi (*fileless*) teknikleri ve tedarik zinciri yollarını kullanır.

13.8.3 Fidye yazılımı

Fidye yazılımı (*ransomware*), kurbanın verilerini şifreleyip çözme anahtarı karşılığında para isteyen kötü amaçlı yazılımdır. Bugün kurumların karşılaştığı en yıkıcı tehditlerden biridir. Tipik bir saldırı beş adımda ilerler (Şekil 13.39).

1. **İlk erişim.** Çoğunlukla bir oltalama e-postasının eki, internete açık ve yaması yapılmamış bir hizmet ya da çalınmış bir uzak masaüstü (RDP) parolasıyla olur.
2. **Yayılma.** Zararlı çalışır ve ağda yanal hareketle olabildiğince çok makineye ulaşır. Saldırganlar bu aşamada günler ya da haftalar geçirebilir.
3. **Yedeklerin silinmesi.** Kurtarmayı engellemek için gölge kopyalar ve ağdan erişilebilen yedekler silinir. Çevrim dışı ya da değiştirilemez yedek bu yüzden şarttır.
4. **Şifreleme.** Dosyalar güçlü bir simetrik şifreyle, örneğin AES ile kilitlenir. Simetrik anahtar saldırganın açık anahtarıyla şifrelenir ya da saldırganın sunucusuna gönderilir. Kurbanın elinde anahtarın hiçbir kopyası kalmaz.
5. **Fidye notu.** Ekranda ödeme yönergeleri görüntülenir. Son yıllarda saldırganlar şifrelemeden önce veriyi dışarı da kopyalar ve ödeme yapılmazsa yayımlamakla tehdit eder.



Şekil 13.39: Fidye yazılımı saldırı zinciri. Ödeme anahtarın verileceğini garanti etmez; yedek ve yama tek güvenilir savunmadır.

Fidye yazılımı, sağlam kriptografinin kötü amaçla nasıl kullanılabileceğini gösterir. AES doğru kullanıldığında anahtar olmadan çözülemez. Dolayısıyla şifreleme başladıktan sonra yapılabilecek pek bir şey kalmaz; savunma önceden yapılmalıdır. Ödeme, anahtarın verileceğini garanti etmez ve yeni sal-

dırıları teşvik eder. Savunma katmanları bilinen araçlardır: 3-2-1 kuralına uygun ve değiştirilemez yedek, hızlı yama, MFA, ağ bölütlemesi ve şüpheli davranışı yakalayan uç nokta izleme (EDR).

13.8.4 Rootkit ve gizlenme

Rootkit, sisteme yerleşmiş bir saldırganın varlığını gizlemek için kullandığı programlar topluluğudur. Adını UNIX'teki root hesabından alır. Rootkit, dosyaları, süreçleri ve ağ bağlantılarını işletim sisteminden ve onun araçlarından saklar. Gizlendiği katmana göre türleri vardır (Şekil 13.40).



Şekil 13.40: Rootkitlerin saklanabileceği katmanlar. Aşağı inildikçe gizlenme güçlenir, kaldırmak zorlaşır.

Uygulama düzeyi
: `ls` ve `ps` gibi araçlar, çıktılarından zararlının dosyalarını ve süreçlerini çıkaran sürümlerle değiştirilir. Bütünlük denetimiyle kolayca yakalanır.

Kütüphane düzeyi
: libc'deki fonksiyonlar saptırılır. Örneğin dizin okuyan fonksiyon belirli adları atlar. Bu kütüphaneyi kullanan bütün programlar aldatılır.

Çekirdek düzeyi
: Rootkit bir çekirdek modülü olarak yüklenir ve sistem çağrı tablosunu ya da çekirdeğin iç veri yapılarını değiştirir. Çekirdeğe sorulan her soru yalan bir cevap alır.

Hipervizör düzeyi
: Rootkit, çalışan işletim sistemini fark ettirmeden bir sanal makinenin içine alır ve onun altında çalışır. İşletim sisteminin içinden bakıldığında hiçbir iz görülmez.

Bellenim düzeyi
: UEFI belleniminde ya da bir çevre biriminin belleniminde saklanır. İşletim sistemi yeniden kurulsa ya da disk değiştirilse bile silinmez.

Rootkit ne kadar derine yerleşirse o kadar güçlüdür, ama kurması da o kadar zordur. Çekirdek ve altındaki rootkitlere karşı içeriden tarama güvenilir değildir, çünkü tarayıcının sorduğu sorulara rootkit cevap verir. Savunma birkaç yönden gelir. İmzalı sürücü zorunluluğu, çekirdeğe yalnız güvenilir mo-

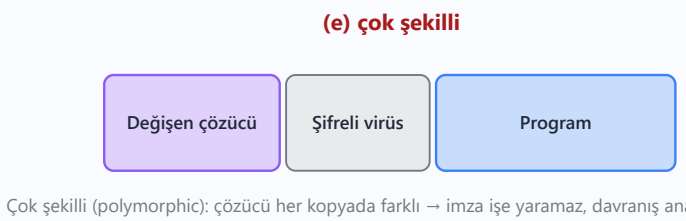
düllerin yüklenmesini sağlar. Güvenli önyükleme, değiştirilmiş bir önyükleyicinin ya da çekirdeğin çalışmasını engeller. Ölçülü önyükleme ve TPM, önyükleme bileşenlerinin özetini kaydeder ve değişikliği ortaya çıkarır. Son çare olarak disk, çevrim dışı olarak, temiz bir sistemden taranır.

13.8.5 Antivirüs ve kaçırma teknikleri

Antivirüs yazılımları ile virüs yazarları arasında uzun süredir bir yarış vardır. Her tespit yöntemi yeni bir kaçırma tekniği doğurmuştur.

En eski yöntem **imza taramadır**. Bilinen her virüsün kodundan karakteristik bir bayt dizisi, yani imza çıkarılır. Tarayıcı dosyalarda bu dizileri arar. Hızlıdır ve yanlış alarmı azdır, ama yalnız bilinen virüsleri yakalar. Virüs yazarları ilk olarak **sıkıştırmaya** başvurdu: virüslü program sıkıştırılınca imza görünmez olur. Tarayıcılar buna dosyaları açıp içlerini tarayarak karşılık verdi. Sonra **şifreli virüsler** çıktı: virüsün gövdesi her kopyada farklı bir anahtarla şifrelenir, yalnız küçük bir çözücü açıkta kalır. Tarayıcılar bu kez çözücünün imzasını aramaya başladı.

Bunun da yanıtı **çok şekilli** (*polymorphic*) virüslerdir (Şekil 13.41). Bu virüsler çözücüyü de her kopyada değiştirir. Aynı işi yapan farklı komut dizileri üretilir: araya etkisiz komutlar eklenir, yazmaçlar değiştirilir, komutların sırası karıştırılır. İki kopya arasında ortak bir bayt dizisi kalmadığı için imza yöntemi işe yaramaz.



Şekil 13.41: Çok şekilli virüs: çözücü her kopyada farklıdır, gövde şifrelidir; sabit bir imza kalmaz.

Bugünkü güvenlik yazılımları bu yüzden birkaç yöntemi birlikte kullanır.

İmza tarama
: Bilinen desenleri hızlı ve düşük yanlış alarm oranıyla yakalar; yeniyi kaçırır.

Bütünlük denetimi
: Sistem dosyalarının özetlerini saklar ve herhangi bir değişikliği fark eder.

Davranışsal ve sezgisel analiz
: Programın ne olduğuna değil ne yaptığına bakar. Kısa sürede yüzlerce dosyayı şifrelemek, başka süreçlere kod enjekte etmek ve gölge kopyaları silmek şüpheli eylemlerdir.

Korumalı alan ve bulut
: Şüpheli dosya yalıtılmış bir ortamda çalıştırılır ve davranışı gözlenir. Sonuç bulut üzerinden bütün müşterilerle paylaşılır.

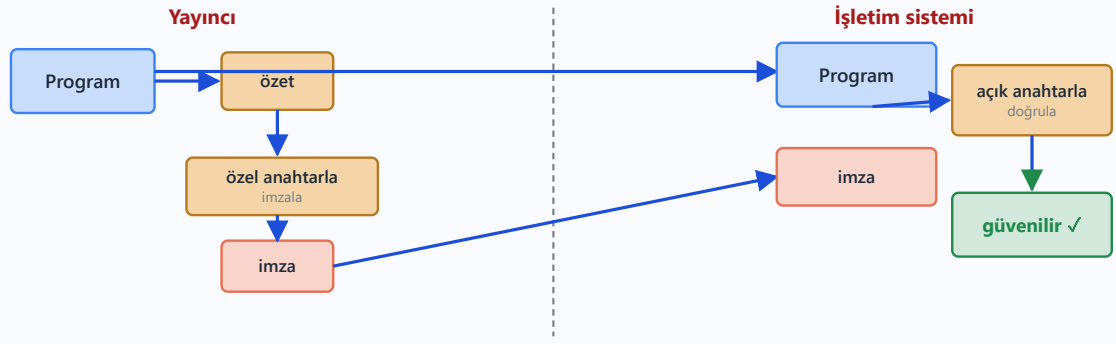
13.9 Savunma ve yalıtım

Hataların ve açıkların tamamen ortadan kalkmayacağı kabul edilirse savunmanın iki yolu kalır. Birincisi, sisteme yalnız güvenilir kaynaktan gelen kodun girmesini sağlamaktır. İkincisi, güvenilmeyen ya da ele geçirilmiş kodun verebileceği zararı sınırlamaktır. Bu kesim ilk yol için kod imzalamayı, ikinci yol için korumalı alanları, konteynerleri ve zorunlu erişim denetimini anlatır.

13.9.1 Kod imzalama

Kod imzalama, dijital imzanın programlara uygulanmasıdır (Şekil 13.42). Amaç, bir programın değiştirilmediğini ve iddia ettiği yayıncıdan geldiğini çalıştırmadan önce doğrulamaktır.

1. **Özet.** Yayıncı programın özetini hesaplar.
2. **İmza.** Özet, yayıncının özel anahtarıyla imzalanır. İmza programla birlikte dağıtılır.
3. **Dağıtım.** Kullanıcı programı ve imzayı alır. Yayıncının sertifikası açık anahtarını taşır ve bir sertifika otoritesine bağlanır.
4. **Doğrulama.** İşletim sistemi özeti yeniden hesaplar ve imzayı açık anahtarla doğrular. Eşleşirse program değiştirilmemiştir ve o yayıncıdandır.



Şekil 13.42: Kod imzalama: yayıncı özeti özel anahtarıyla imzalar, işletim sistemi açık anahtarla doğrular.

Bugünkü sistemlerin çoğu bu temele dayanır. Mobil uygulama mağazaları yalnız imzalı uygulamaları dağıtır. Windows, 64 bitlik sürümlerinde çekirdek sürücülerinin imzalı olmasını şart koşar. Linux dağıtımlarının paket yöneticileri (`apt` , `dnf`) her paketin imzasını dağıtımın anahtarıyla denetler. Bazı platformlar daha da ileri gider ve yalnız imzalı kodun çalışmasına izin verir.

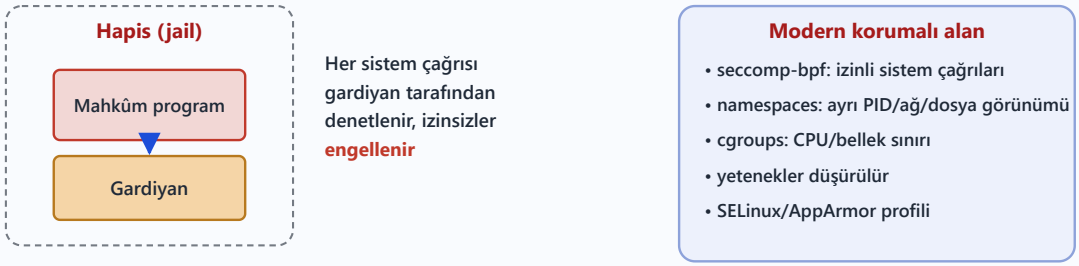
Kod imzalamanın neyi kanıtlamadığı da önemlidir. İmza, programın kimden geldiğini ve yolda değişmediğini gösterir; programın zararsız olduğunu göstermez. Yayıncının imza anahtarı çalınırsa ya da yayıncının derleme sunucusu ele geçirilirse zararlı kod da geçerli bir imza taşır. Tedarik zinciri saldırıları tam bu noktayı hedefler.

13.9.2 Korumalı alan

Tanım 13.20

Korumalı alan (*sandbox*), güvenilmeyen bir programı sınırlı haklarla, sistemin geri kalanından yalıtılmış bir ortamda çalıştırma yöntemidir. Program ele geçirilse bile verebileceği zarar alanın sınırlarıyla kısıtlı kalır.

Fikir, Lampson'ın hapsetme probleminin pratik bir cevabıdır. Şekil 13.43 eski bir benzetmeyi kullanır: program bir hapishanededir ve yaptığı her sistem çağrısını bir gardiyan denetler. İzin verilmeyen çağrılar engellenir. Erken UNIX'teki `chroot` hapisleri ve FreeBSD'nin `jail` mekanizması bu düşüncenin ilk uygulamalarıdır.



Şekil 13.43: Solda hapsedilmiş program ve gardiyanı; sağda Linux'taki modern korumalı alanın yapı taşları.

Bugün tarayıcı sekmeleri, mobil uygulamalar ve birçok sunucu hizmeti korumalı alanda çalışır. Linux'ta bir korumalı alan birkaç çekirdek mekanizmasının birleşimidir.

seccomp-bpf
: Sürecin yapabileceği sistem çağrılarını bir filtreyle sınırlar. Örneğin bir görüntü çözücü yalnız `read`, `write` ve `exit` çağırabilir. Başka bir çağrı yapmaya kalkarsa çekirdek süreci öldürür.

Ad alanları (*namespaces*)
: Sürece sistemin ayrı bir görünümünü verir. PID ad alanındaki süreç yalnız kendi alanındaki süreçleri görür; ağ ad alanının kendi ağ arayüzleri, bağlama ad alanının kendi dosya sistemi ağacı vardır.

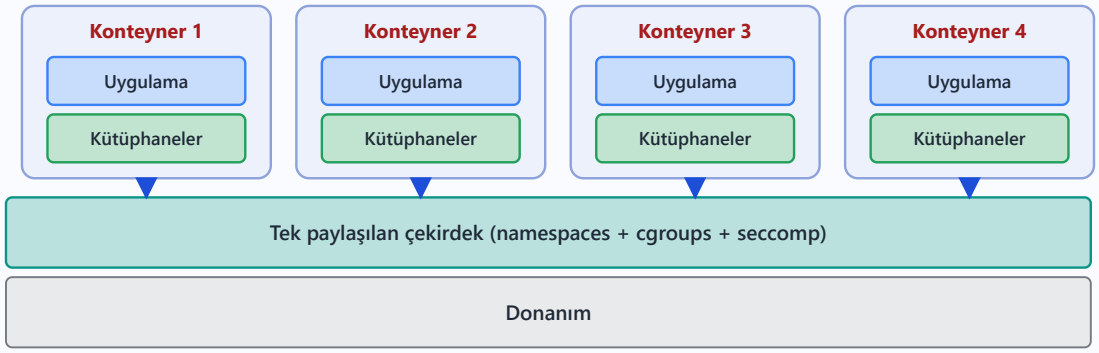
cgroups
: Bir süreç grubunun kullanabileceği CPU, bellek ve G/Ç miktarını sınırlar. Kaynak tüketme yoluyla hizmet reddini önler.

Yeteneklerin düşürülmesi
: Root olarak başlayan bir süreç bile ihtiyaç duymadığı bütün Linux yeteneklerini bırakır.

Zorunlu erişim denetimi
: SELinux ya da AppArmor profili, sürecin erişebileceği dosyaları ve kaynakları ayrıca sınırlar.

13.9.3 Konteynerler

Ad alanları ve cgroups bir araya geldiğinde ortaya çıkan yapıya **konteyner** denir (Şekil 13.44). Konteynerdeki uygulama kendine ait bir dosya sistemi, ağ yığını ve süreç ağacı görür; sanki tek başına bir makinedeymiş gibi çalışır. Ama gerçekte bütün konteynerler aynı çekirdeği paylaşır. Docker ve Kubernetes gibi araçlar bu çekirdek özellikleri üzerine kurulmuştur.



Şekil 13.44: Dört konteyner: her birinin kendi uygulaması ve kütüphaneleri var, ama hepsi tek bir çekirdeği paylaşır.

Konteynerin sanal makineyle karşılaştırılması güvenlik açısından öğreticidir (Tablo 13.20). Sanal makinede her konuk kendi çekirdeğini çalıştırır ve konukları birbirinden hipervizör ile donanım ayırır. Konteynerde yalıtım tamamen çekirdeğin kendi özelliklerine dayanır.

	Konteyner	Sanal makine
Yalıtım	Çekirdek özellikleri	Donanım + hipervizör
Ek yük	Düşük, saniyede başlar	Yüksek, ayrı çekirdek
Saldırı yüzeyi	Çekirdek açığı = kaçış riski	Daha güçlü sınır

Tablo 13.20: Konteyner ve sanal makine yalıtımı.

Bu farkın sonucu açıktır. Paylaşılan çekirdekte bir açık bulunursa, konteynerdeki bir saldırgan bu açığı kullanarak konteynerden kaçabilir ve aynı makinedeki bütün öteki konteynerlere erişebilir. Sanal makinede ise konuk çekirdekteki bir açık yalnız o konuğu etkiler; dışarı çıkmak için hipervizörde ikinci bir açık gerekir. Bulut sağlayıcıları bu yüzden farklı müşterilerin konteynerlerini çoğunlukla ayrı sanal makinelerde ya da hafif sanal makinelerde (örneğin Firecracker) çalıştırır.

Sık yapılan hata

Konteynerde root olarak çalışmayı zararsız saymak. Kullanıcı ad alanı kullanılmıyorsa konteynerdeki root, çekirdek açısından gerçek root'tur (uid 0). Bir yapılandırma hatası ya da çekirdek açığıyla konteynerden çıkan bu süreç bütün makineye hükmeder. Konteyner süreçleri de

en az ayrıcalıkla çalıştırılmalıdır.

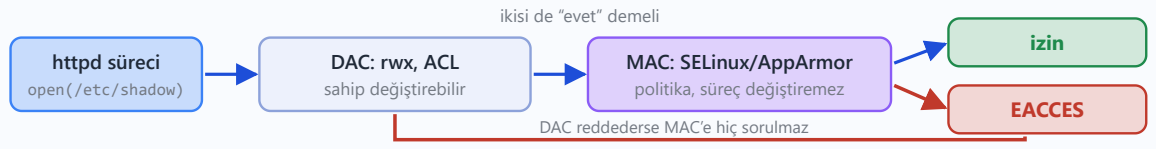
13.9.4 Zorunlu erişim denetimi: SELinux ve AppArmor

UNIX izinleri ve ACL'ler isteğe bağlı erişim denetimidir (DAC): bir dosyanın izinlerini sahibi belirler ve istediği zaman değiştirebilir. Bunun bir zaafı vardır. Ali'nin hesabıyla çalışan bir zararlı da Ali'nin dosyalarının izinlerini değiştirebilir, çünkü çekirdek açısından o da Ali'dir. Zorunlu erişim denetimi (MAC) bu zaafı kapatır: politikayı sistem yöneticisi belirler ve hiçbir süreç, root bile olsa, kendi kısıtını gevşetemez.

SELinux
: Her özneye ve nesneye bir güvenlik etiketi verir. Kurallar **tip zorlaması** (*type enforcement*) ile yazılır: "httpd_t tipindeki süreçler httpd_sys_content_t tipindeki dosyaları okuyabilir" gibi. Çok ayrıntılı ve güçlüdür, ama karmaşıktır. Ayrıca BLP tarzı çok düzeyli güvenliği (MLS) de destekler.

AppArmor
: Dosya yollarına dayanan, uygulama başına profiller kullanır: "/usr/sbin/nginx yalnız /var/www/ altını okuyabilir" gibi. Yazması ve anlaması daha kolaydır.

Linux'ta her iki sistem de çekirdeğin **Linux Güvenlik Modülleri** (LSM) altyapısını kullanır. Çekirdek, kritik işlemlerin önüne kancalar koyar. Bir süreç örneğin `open` çağırdığında önce olağan DAC denetimi yapılır. DAC izin verirse LSM kancası çağrılır ve MAC politikası sorulur (Şekil 13.45). İşlemin yapılabilmesi için ikisinin de "evet" demesi gerekir. DAC reddederse MAC'e hiç sorulmaz.



Şekil 13.45: Bir `open()` isteği önce DAC izin bitlerinden, sonra LSM kancasındaki MAC politikasından geçer.

Örnek 13.15

Bir web sunucusu, bir açık yüzünden ele geçirildi. Saldırgan sunucunun kullanıcı hesabıyla komut çalıştırabiliyor. İki gözlem yapılıyor: (1) saldırgan, sunucuya ait bir dosyanın iznini `chmod` ile rahatça değiştirebiliyor; (2) izin bitleri izin verse bile `/home` altındaki dosyalara erişemiyor. Bu davranışlar hangi denetimden kaynaklanır?

1. (1) DAC'tir. Dosyanın sahibi sunucu hesabıdır ve DAC'te sahip, zararlı da olsa, izinleri değiştirebilir.
2. (2) MAC'tir. SELinux ya da AppArmor profili sunucu sürecini yalnız web dosyalarıyla sınırlar. İzin bitleri "evet" dese de MAC "hayır" der ve sonuç EACCES olur.

MAC, ele geçirilen bir süreci bile profilinin izin verdiği işlemlerle sınırlar. Saldırgan yalnız o profilin tanımladığı dosyalara erişebilir. Android de uygulamaları birbirinden ayırmak için SELinux'u bütünüyle zorunlu kipte çalıştırır.

13.10 Donanım düzeyinde güvenlik

Önceki kesimlerin hepsi işletim sisteminin doğru çalıştığını varsaydı. Peki işletim sisteminin kendisi değiştirilmişse? Ya da donanım, yazılımın bütün denetimlerine rağmen bilgi sızdırıyorsa? Bu kesim güven zincirinin en altını ele alır.

13.10.1 Güvenli önyükleme ve TPM

Bilgisayar açıldığında sırasıyla bellenim, önyükleyici, çekirdek ve işletim sistemi çalışır. Bir saldırgan bu zincirin herhangi bir halkasını değiştirebilirse, kendi kodunu işletim sisteminden önce çalıştırır. İşletim sistemi açıldığında artık her şeyi denetleyen bu kod olur ve onu içeriden fark etmek imkânsızdır. Önyükleme sürecine yerleşen bu tür zararlılara **önyükleme kiti** (*bootkit*) denir.

Tanım 13.21

Güvenli önyükleme (*secure boot*), önyükleme zincirindeki her katmanın bir sonrakinin dijital imzasını doğrulamadan onu çalıştırmamasıdır. Zincir, donanımdaki değiştirilemez bir **kök güvene** dayanır.



Şekil 13.46: Güven zinciri: her aşama bir sonrakinin imzasını doğrular. İmzasız ya da değiştirilmiş bir bileşen çalıştırılmaz. TPM ölçümleri saklar.

Zincir şöyle işler (Şekil 13.46). Donanımdaki kök güven, UEFI belleniminin imzasını doğrular. UEFI, diskteki önyükleyicinin imzasını kendi veritabanındaki anahtarlarla denetler. Önyükleyici çekirdeğin, çekirdek de yüklediği sürücülerin imzasını denetler. Bir önyükleme kiti diskteki önyükleyiciyi değiştirmişse, UEFI imzayı doğrulayamaz ve önyükleyiciyi çalıştırmaz.

Güvenli önyüklemenin tamamlayıcısı **TPM**'dir (*Trusted Platform Module*, Güvenilir Platform Modülü). TPM, anakarta ya da işlemciye yerleşik küçük bir güvenlik yongasıdır. İçinde **platform yapılandırma yazmaçları** (PCR) vardır. **Ölçülü önyüklemede** (*measured boot*) her bileşen çalıştırılmadan önce özeti alınır ve bir PCR'ye eklenir. Ekleme işlemi yazmacın eski değeriyle yeni özetin birlikte özetlenmesidir. Bu yüzden bir PCR'nin son değeri, bütün zincirin sırasıyla birlikte bir parmak izidir ve geriye doğru değiştirilemez.

TPM bu ölçümleri iki biçimde kullanır. Birincisi **mühürlemedir**: disk şifreleme anahtarı (örneğin BitLocker ya da LUKS anahtarı) belirli PCR değerlerine mühürlenir. TPM anahtarı yalnız sistem beklenen durumda açıldıysa serbest bırakır. Önyükleyici değiştirilmişse ölçüm farklı çıkar ve anahtar açılmaz. İkincisi **uzaktan kanıtlamadır** (*attestation*): TPM ölçümleri kendi anahtarıyla imzalayıp uzaktaki bir sunucuya gönderir. Sunucu makinenin güvenilir bir yazılımla açıldığını doğrulayabilir.

Sık yapılan hata

Güvenli önyüklemeyi ve ölçülü önyüklemeyi aynı şey saymak. Güvenli önyükleme imzasız bileşeni *çalıştırmaz*. Ölçülü önyükleme ise her şeyi çalıştırır, ama ne çalıştığını *kaydeder* ve beklenmeyen bir ölçümde anahtarı açmaz. TPM hiçbir şeyi onarmaz ya da engellemez; yalnız ölçer, saklar ve buna göre anahtar verir.

13.10.2 Spectre, Meltdown ve yan kanallar

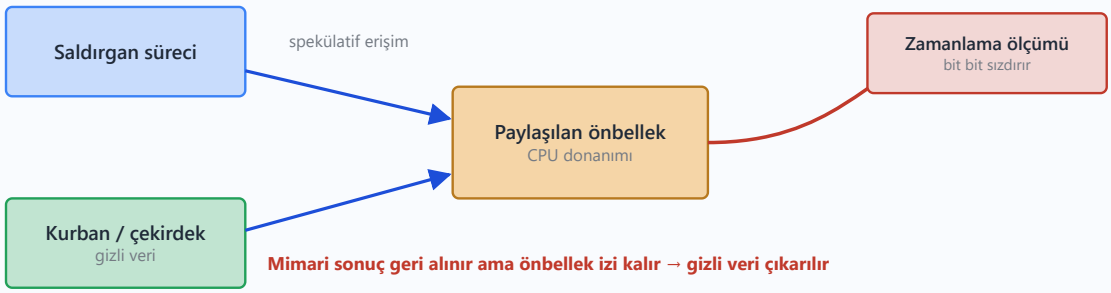
Gizli kanalda iki süreç bilerek işbirliği yapar. Yan kanalda ise kurban hiçbir şeyin farkında değildir; saldırgan onun davranışının dolaylı izlerini ölçer.

Tanım 13.22

Yan kanal (*side channel*), veriyi hesaplamanın dolaylı fiziksel etkilerinden sızdıran bir saldırı yoludur. Bu etkiler süre, güç tüketimi, önbellek durumu, elektromanyetik yayılım ya da ses olabilir.

Klasik bir örnek, parolayı karakter karakter karşılaştıran ve ilk farkta duran bir fonksiyondur. Doğru başlayan bir deneme biraz daha uzun sürer; saldırgan süreleri ölçerek parolayı karakter karakter bulur. Sabit zamanlı karşılaştırmanın nedeni budur. Önbellek de güçlü bir yan kanaldır. Bir verinin önbellekte olup olmadığı, ona erişim süresinden anlaşılır: önbellekten birkaç nanosaniyede, ana bellekten yüz nanosaniye civarında gelir.

2018'in başında duyurulan **Meltdown** ve **Spectre**, bu yan kanalı işlemcilerin **spekülatif yürütmesiyle** birleştirdi (Şekil 13.47). Bugünkü işlemciler hız kazanmak için, sonucu henüz kesinleşmemiş komutları önceden yürütür. Örneğin bir dal koşulu hesaplanmadan, tahmin edilen yöndeki komutlar çalıştırılır. Tahmin yanlış çıkarsa işlemci bu komutların mimari sonuçlarını, yani yazmaç değerlerini geri alır. Ama komutların önbelleğe getirdiği veriler önbellekte kalır.



Şekil 13.47: Spekülatif erişimin mimari sonucu geri alınır, ama paylaşılan önbellekte bıraktığı iz kalır ve zamanlama ölçümüyle okunur.

Saldırgan bunu şöyle kullanır. İşlemciyi, erişme hakkı olmayan bir gizli baytı spekülatif olarak okumaya yönlendirir. Spekülatif kod bu baytın değerini, büyük bir dizinin içinde bir indis olarak kullanır ve o indisteki satırı önbelleğe getirir. İşlemci hatayı fark eder ve sonucu geri alır; saldırgan gizli baytı hiç görmez. Ama sonra dizinin bütün satırlarına erişip süreleri ölçer. Hızlı gelen satırın indisi, gizli baytın değeridir. Meltdown bu yolla kullanıcı kipindeki bir sürecin çekirdek belleğini okumasına izin veriyordu. Spectre ise başka bir sürecin ya da aynı süreçteki korumalı bir alanın belleğini okumayı sağlıyordu.

Önlemler hem yazılımda hem donanımdadır. Meltdown'a karşı işletim sistemleri **çekirdek sayfa tablosu yalıtımını** (KPTI) getirdi: kullanıcı kipindeyken çekirdek belleğinin büyük kısmı sayfa tablosundan tamamen çıkarılır. Spectre'a karşı mikrokod güncellemeleri ve derleyicinin eklediği spekülasyon bariyerleri kullanılır. Bu önlemlerin çoğu bir performans bedeliyle gelir; sistem çağrısı yoğun iş yüklerinde yavaşlama belirgindir. Bu açıklar önemli bir dersi de gösterdi: koruma mekanizmalarının doğruluğu yalnız mimari belirtime değil, donanımın gerçekleme ayrıntılarına da bağlıdır.

13.11 Özet

- Güvenliğin üç temel hedefi **gizlilik**, **bütünlük** ve **erişilebilirliktir**; bunlara özgünlük ve hesap verebilirlik eklenir. **Tehdit** olası zarar, **zafiyet** zayıflık, **saldırı** kullanma girişimi, **istismar** da bunu yapan koddur.
- Ağda dinleme, tekrar, ortadaki adam ve oturum kaçırma saldırıları vardır. Çareleri sırasıyla şifreleme, tek kullanımlık değerler, sertifikayla kimlik doğrulama ve güvenli çerezlerdir.
- Savunma **derinlemesine** kurulur; güvenlik iddiaları küçük tutulması gereken **TCB**'ye ve her erişimi denetleyen **referans izleyiciye** dayanır.
- **Simetrik** şifreleme hızlıdır ama paylaşılan anahtar ister; **açık anahtarlı** şifreleme anahtar dağıtımını çözer ama yavaştır. Gerçek sistemler ikisini birlikte kullanır. Güvenlik anahtara dayanır, algoritmanın gizliliğine değil (Kerckhoffs).

- **Özet fonksiyonu** tek yönlü ve çakışmaya dayanıklıdır. **Dijital imzada** özet özel anahtarla imzalanır, açık anahtarla doğrulanır. **Sertifikalar** açık anahtarı kimliğe bağlar. **TLS 1.3** geçici Diffie-Hellman, sertifika ve kimlik doğrulamalı şifrelemeyi tek gidiş-dönüşte birleştirir.
- **Diffie-Hellman** örneğinde (p = 23, g = 5, a = 6, b = 15) A = 8, B = 19 ve ortak sır 2'dir. Kare al ve çarp yöntemi n bitlik üssü n turda hesaplar.
- **Koruma matrisi** seyrektir; sütunları **ACL**, satırları **yetenek listesi** olarak saklanır. ACL'de geri alma, yetenekte devretme kolaydır. UNIX izinleri dokuz bitlik bir ACL'dir; dosya tanımlayıcısı bir yetenektir.
- **En az ayrıcalık** ilkesi gereği setuid programlar ayrıcalığı işleri biter bitmez `setresuid` ile kalıcı olarak bırakır. Linux root'u yaklaşık 41 **yeteneğe** böler.
- **Bell-LaPadula** gizliliği korur: yukarı okuma ve aşağı yazma yoktur. **Biba** bütünlüğü korur: aşağı okuma ve yukarı yazma yoktur. İkisi de **gizli kanalları** kapatamaz; kanalların yalnız bant genişliği düşürülebilir.
- Parolalar **tuzlu** ve **yavaş** bir özetle (Argon2id) saklanır: tuz toplu saldırıyı, yavaşlık her denemeyi pahalılaştırır. **MFA** farklı kategorilerden iki etken ister. **TOTP** kodu HMAC(K, ⌊t / 30⌋)'tan dinamik kesmeyle türetilir. **FIDO2** imzası sitenin kökenine bağlı olduğu için oltalamaya dayanıklıdır.
- **Yığın tampon taşması** dönüş adresini bozar. Savunmalar katmanlıdır: **kanarya** bozulmayı dönüşte yakalar, **NX** enjekte kodu çalıştırmaz, **ASLR** adresleri gizler, **CFI** ve gölge yığın ROP'u zorlaştırır; en köklü çözüm bellek güvenli dillerdir.
- **TOCTOU** açığında kontrol ile kullanım arasında kaynak değişir; çözüm adı bir kez çözüp aynı tanımlayıcı üzerinde çalışmaktır.
- İçeriden saldırılar mantık bombası, arka kapı ve giriş sahtekârlığıdır. **Kötü amaçlı yazılımlar** virüs, solucan, Truva atı, fidye yazılımı, rootkit, casus yazılım ve botlardır; çok şekilli virüsler imza taramasını aşar.
- **Kod imzalama** kaynağı doğrular. **Korumalı alan** ve **konteyner** zararı sınırlar; konteyner çekirdeği paylaştığı için sanal makineden zayıf bir sınırdır. **MAC** (SELinux, AppArmor) ele geçirilen süreci bile profiliyle sınırlar.
- **Güvenli önyükleme** imzasız bileşeni çalıştırmaz; **TPM** ölçümleri saklar ve anahtarları bunlara mühürler. **Spectre** ve **Meltdown**, spekülatif yürütmenin önbellekte bıraktığı izi bir yan kanal olarak kullanır.

13.12 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

gizlilik / bütünlük / erişilebilirlik (CIA) *confidentiality / integrity / availability*

tehdit, zafiyet, istismar *threat, vulnerability, exploit*

hizmet reddi *denial of service*

ortadaki adam saldırısı *man-in-the-middle attack*

derinlemesine savunma *defense in depth*

güvenilir bilgi işleme tabanı (TCB) *trusted computing base* · referans izleyici *reference monitor* · şifreleme *encryption* · açık anahtarlı şifreleme *public-key cryptography* · özet fonksiyonu *hash function (cryptographic)* · dijital imza *digital signature* · sertifika, açık anahtar altyapısı *certificate, public key infrastructure* · ileriye dönük gizlilik *forward secrecy* · koruma etki alanı *protection domain* · koruma matrisi *access matrix* · erişim denetim listesi (ACL) *access control list* · yetenek *capability* · en az ayrıcalık ilkesi *principle of least privilege* · isteğe bağlı / zorunlu erişim denetimi *discretionary / mandatory access control* · gizli kanal *covert channel* · kimlik doğrulama *authentication* · tuz *salt* · çok etkenli kimlik doğrulama *multi-factor authentication* · tampon taşması *buffer overflow* · yığın kanaryası *stack canary* · adres alanı yerleşiminin rastgeleleştirilmesi (ASLR) *address space layout randomization* · kötü amaçlı yazılım *malware* · fidye yazılımı *ransomware* · korumalı alan *sandbox* · konteyner, ad alanı *container, namespace* · güvenli önyükleme *secure boot* · yan kanal *side channel*

13.13 Kendini Dene

1. Tablo 13.3'teki yerine koyma anahtarıyla GIRESUN sözcüğü nasıl şifrelenir? Şifreli ZTLZ neyi gizler?

   Cevap

   G → U, I → O, R → K, E → T, S → L, U → X, N → F: UOKTLXF. Ters yönde Z → T, T → E, L → S, Z → T: TEST. İki Z'nin ikisi de T'ye gider; sıklık yine korunur.

2. Diffie-Hellman'da p = 23 ve g = 5'tir. Alice a = 4, Bob b = 3 seçiyor. A, B ve ortak sır nedir?

   Cevap

   A = 5^4 mod 23 = 625 mod 23 = 4. B = 5^3 mod 23 = 125 mod 23 = 10. Alice: 10^4 mod 23 = 10000 mod 23 = 18. Bob: 4^3 mod 23 = 64 mod 23 = 18. Ortak sır 18'dir.

3. Bir arkadaş, “Alice belgeyi Bob'la paylaştığı gizli anahtarla HMAC'leyip gönderirse bu da bir imza-

dır" diyor. Bu düzen bütünlüğü ve inkâr edilemezliği sağlar mı?

Cevap

Bütünlüğü sağlar: anahtarı bilmeyen biri belgeyi değiştirip geçerli bir kod üretemez. İnkâr edilemezliği sağlamaz: anahtarı Bob da bildiği için aynı kodu Bob da üretebilirdi. Alice "bunu Bob üretti" diyebilir. İnkâr edilemezlik için yalnız Alice'in bildiği bir özel anahtar gerekir.

4. Kod izlemedeki ACL ile `erisim(G2, R|W)` ve `erisim(YON, W)` ne döndürür? Karar ifadesi `(h & istek) != 0` olsaydı ilk çağrının sonucu değişir miydi?

Cevap

G2'nin girdisi h = 2'dir; istek 6'dır. 2 & 6 = 2 ≠ 6, ret (0). YON için h = 7, 7 & 2 = 2, izin (1). Hatalı ifadeyle 2 & 6 = 2 ≠ 0 olur ve G2'ye okuma hakkı olmadığı hâlde izin verilirdi.

5. Bir dosyanın modu 754'tür. Grup üyeleri dosyayı değiştirebilir mi? umask 027 iken oluşturulan yeni bir dosya ve dizin hangi modları alır?

Cevap

754 = rwxr-xr--. Grubun hakkı r-x'tir; yazma yoktur, değiştiremez. umask 027 ile dosya 666 → 640, dizin 777 → 750 olur.

6. Gizli düzeydeki bir süreç için BLP hangi işleme izin verir: Tasnif Dışı bir duyuruyu okumak mı, Hizmete Özel bir dosyaya yazmak mı? Aynı süreç Biba'da Orta düzeyde olsaydı Düşük düzeydeki bir İnternet dosyasını okuyabilir miydi?

Cevap

BLP'de aşağı okuma serbesttir; Tasnif Dışı duyuruyu okumak izinlidir. Hizmete Özel dosyaya yazmak aşağı yazmadır ve yasaktır. Biba'da aşağı okuma yasaktır; Orta düzeydeki süreç Düşük dosyayı okuyamaz.

7. Dosya kilidiyle çalışan bir gizli kanalda aralık 5 ms'ye indiriliyor. Kanalın hızı nedir? 2048 bitlik bir RSA özel anahtarı ne kadar sürede sızar?

Cevap

Saniyede 1000 / 5 = 200 aralık, yani 200 bit/s. 2048 / 200 = 10,24 saniye.

8. Parolalar 12 küçük harften oluşuyor ve düz SHA-256 ile saklanıyor. Saniyede 10^{10} denemeyle tek bir kullanıcının parolası en kötü durumda ne kadar sürede bulunur? Tuz eklemek bu süreyi değiş-

tirir mi?

> **Cevap**
>
> $26^{12} \approx 9{,}54 \times 10^{16}$; 10^{10}'a bölününce $\approx 9{,}54 \times 10^{6}$ saniye, yaklaşık 110 gün. Tuz tek kullanıcının süresini değiştirmez; bütün kullanıcılar hedeflendiğinde işi kullanıcı sayısıyla çarpar.

9. t = 1.000.000.000 anında TOTP sayacı T kaçtır? Bu koda karşılık gelen 30 saniyelik pencere hangi aralıktır?

> **Cevap**
>
> 1.000.000.000 / 30 = 33.333.333,3; T = 33.333.333. Pencere T · 30 = 999.999.990'dan 1.000.000.019'a kadardır.

10. 64 bitlik bir makinede kanaryasız bir fonksiyon `char ad[16]` dizisine 30 karakterlik bir dizgiyi `strcpy` ile kopyalıyor. Kaç bayt taşar? Kayıtlı rbp ve dönüş adresinin kaçar baytı bozulur?

> **Cevap**
>
> Sonlandırıcıyla 31 bayt yazılır; 31 − 16 = 15 bayt taşar. İlk 8'i kayıtlı rbp'nin tamamını, kalan 7'si (son karakter ve `\0` dahil) dönüş adresinin 7 baytını bozar.

13.14 İleri okuma

- J. H. Saltzer ve M. D. Schroeder, "The Protection of Information in Computer Systems", 1975: en az ayrıcalık, güvenli varsayılan ve tam aracılık ilkelerini ilk kez bir arada sıralayan klasik makale. Bugün de günceldir.
- R. Anderson, *Security Engineering*, 3. baskı: güvenliği teknik olduğu kadar ekonomik ve insani bir sorun olarak ele alır. Parola, erişim denetimi ve biçimsel modeller bölümleri gerçek olay örnekleri içerir.
- Aleph One, "Smashing the Stack for Fun and Profit", Phrack 49, 1996: yığın tampon taşmasının istismarını adım adım gösteren tarihî metin. Bugünkü savunmaların neden gerektiğini anlamaya yardım eder.
- P. Kocher vd., "Spectre Attacks: Exploiting Speculative Execution" ve M. Lipp vd., "Meltdown", 2018: spekülatif yürütmenin önbellek yan kanalıyla nasıl birleştiğini özgün kaynağından anlatır.

13.15 Kaynaklar

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 9: Security.

2. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 16: Security; Bölüm 17: Protection.
3. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — güvenlik bölümleri.
4. R. Anderson, *Security Engineering*, 3. baskı, Wiley, 2020.
5. J. H. Saltzer, M. D. Schroeder, "The Protection of Information in Computer Systems", 1975.
6. D. E. Bell, L. J. LaPadula (1973); K. J. Biba (1977) — biçimsel güvenlik modelleri. B. W. Lampson, "A Note on the Confinement Problem", 1973.
7. RFC 8446 (TLS 1.3), RFC 6238 (TOTP), RFC 9106 (Argon2); FIDO2/WebAuthn (W3C); NIST FIPS 203 (ML-KEM).
8. Aleph One, "Smashing the Stack for Fun and Profit", Phrack 49, 1996; Kocher vd., "Spectre"; Lipp vd., "Meltdown", 2018.

KISIM IV · BÖLÜM 14

İşletim Sistemi Tasarımı

Süreçler, bellek, dosya sistemleri ve G/Ç tek tek ele alındığında her biri kendi algoritmalarıyla çözülmüş gibi görünür. Bir işletim sistemini baştan tasarlayan ekip ise başka türden sorularla karşılaşır. Hangi soyutlamalar sunulmalı? Sistem çağrıları kaç tane ve ne kadar genel olmalı? Sürücüler çekirdeğin içinde mi, dışında mı çalışmalı? Bir ad hangi anda nesneye bağlanmalı? Kod hangi noktada hızlandırılmalı, hangi noktada sade bırakılmalı? Bu sorulara kesin cevaplar yoktur; deneyimle biriken ilkeler ve ödünleşimler vardır. Not önce tasarımı zorlaştıran etkenleri ve bir işletim sisteminin dört temel hedefini anlatır. Ardından arayüz tasarımı, sistem yapısı seçenekleri ve mekanizma–politika ayrımı, diklik, adlandırma ve bağlama zamanı gibi tasarım ilkeleri gelir. Dolaylılık, yeniden girişlilik, uzay-zaman ödünleşimi, önbellek, ipucu ve yaygın durumu hızlandırma teknikleri kısa C kodları üzerinde adım adım izlenir. Son kısımlar büyük yazılım projelerinin yönetimini, çekirdek kodunun nasıl test edilip hata ayıklandığını ve işletim sistemlerindeki güncel eğilimleri ele alır. Bütün izlemeler ve sayılar Python ile yeniden hesaplanarak denetlenmiştir.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum14`

ÖĞRENME ÇIKTILARI

Zorluklar ve hedefler
İşletim sistemi tasarımını zorlaştıran etkenleri ve bir işletim sisteminin dört temel hedefini açıklamak.

Arayüz tasarımı
Basitlik, bütünlük ve verimlilik ilkelerini; yürütme ve veri paradigmalarını sistem çağrısı örnekleriyle tartışmak.

Sistem yapısı
Yapı seçeneklerini yalıtım–performans ödünleşimiyle değerlendirmek; katman sırasını, güvenli genişletmeyi ve çekirdek iş parçacıklarını tasarım kararı olarak tartışmak.

Tasarım ilkeleri
Mekanizma–politika ayrımını, dikliği, iki düzeyli adlandırmayı, bağlama zamanını ve statik–dinamik yapıları uygulamak.

Teknikler ve performans
Dolaylılık, yeniden girişlilik, uzay-zaman ödünleşimi, önbellek, ipucu ve yaygın durumu hızlandırma tekniklerini örneklerle kullanmak.

Proje, test ve eğilimler
Brooks yasasını yorumlamak; çekirdek test ve hata ayıklama araçlarını seçmek; sanallaştırma, çok çekirdek, Rust, eBPF ve unikernel eğilimlerini değerlendirmek.

14.1 Tasarımın zorlukları ve hedefleri

Bir derleyici ya da veritabanı yazan ekip ne yapacağını oldukça iyi bilir. Derleyicinin girdisi bir dil tanımıyla, çıktısı bir komut kümesiyle belirlidir. İşletim sisteminin işi ise bu kadar net çizilmiş değildir. Üzerinde hangi programların çalışacağı, hangi donanımda ve kaç yıl kullanılacağı önceden bilinmez. Bu yüzden işletim sistemi tasarımı bir algoritma seçmekten çok, birbiriyle çatışan isteklerin dengelenmesidir.

14.1.1 İşletim sistemi tasarlamak neden zor?

Tanenbaum, işletim sistemi tasarımını diğer yazılım projelerinden daha zor kılan sekiz etken sayar (Tablo 14.1). Bu etkenlerin bir kısmı yalnız büyüklükten, bir kısmı da işletim sisteminin konumundan kaynaklanır. İşletim sistemi donanım ile bütün uygulamalar arasında durur ve ikisinin de değişmesine dayanmak zorundadır.

Etken	Açıklama
1. Devasa boyut	Linux çekirdeği yaklaşık 40 milyon satırdır; bütününü kavrayan tek bir kişi yoktur.
2. Eş zamanlılık	Aygıtlar, kesmeler ve iş parçacıkları aynı anda çalışır; yarış koşulları ve kilitlenmeler doğar.
3. Düşmanca kullanıcılar	Bazı kullanıcılar sistemi bozmak ya da veri çalmak ister; her girdi şüphelidir.
4. Yalıtıma rağmen paylaşım	Kullanıcılar birbirinden ayrılmalı, ama seçtikleri bilgiyi paylaşabilmelidir.
5. Uzun ömür	Bir işletim sistemi 20–30 yıl yaşar; gelecekteki donanım ve uygulamalar öngörülmelidir.
6. Genellik	Sistemin nasıl kullanılacağı bilinmez: telefondan süper bilgisayara kadar.
7. Taşınabilirlik	Farklı işlemci ve platformlarda çalışmalıdır: x86-64, ARM64, RISC-V.
8. Geriye dönük uyumluluk	Eski programlar çalışmayı sürdürmelidir; yeni fikirler eski kısıtlarla sınırlanır.

Tablo 14.1: Tanenbaum'un işletim sistemi tasarımını zorlaştıran sekiz etkeni.

Boyut tek başına ciddi bir sorundur. Kırk milyon satırın büyük kısmı sürücü kodudur, ama geri kalanı da tek bir kişinin aklında tutabileceğinden çok büyüktür. Böyle bir sistemde bir değişikliğin nereleri etkileyeceğini kestirmek zordur. Bu yüzden iyi tasarlanmış arayüzler ve bağımlılıkları sınırlayan bir yapı, kodun kendisi kadar önemlidir.

Eş zamanlılık işletim sistemine özgü bir zorluktur. Uygulama programcısı çoğu zaman tek iş parçacıklı düşünebilir. Çekirdek ise kesmelerle, birden çok çekirdekle ve aynı anda gelen sistem çağrılarıyla baştan beri eş zamanlıdır. Bir hata ancak belirli bir zamanlama denk geldiğinde ortaya çıkabilir ve yeniden üretilmesi günler sürebilir.

Düşmanca kullanıcılar ile yalıtıma rağmen paylaşım birbirini dengeleyen iki istektir. Sistem her kullanıcıyı diğerlerinden korumalıdır, ama kullanıcılar birlikte çalışabilmek için dosya ve bellek de paylaşmak ister. Paylaşım için açılan her kapı aynı zamanda bir saldırı yüzeyidir.

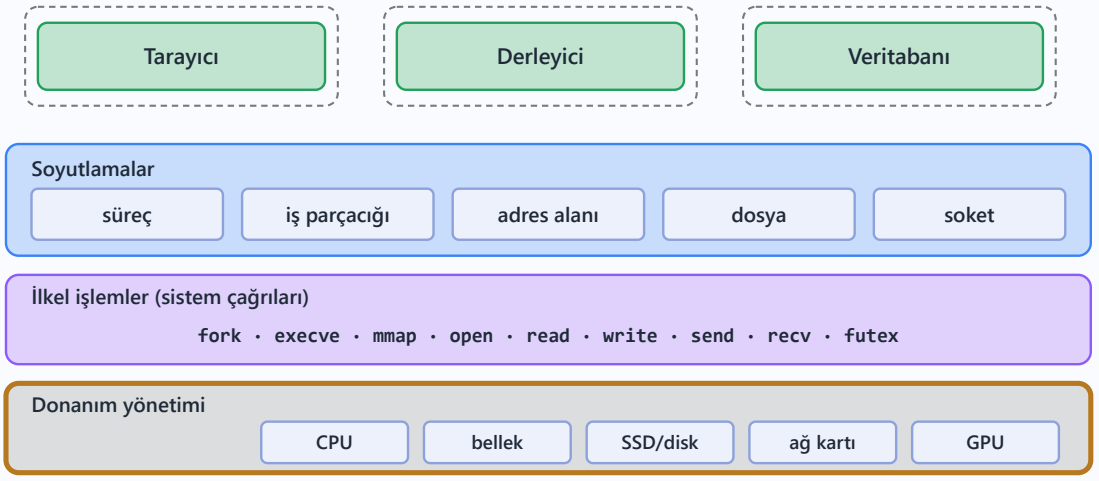
Son dört etken zamanla ilgilidir. Uzun ömür, sistemin bugün var olmayan donanıma uyum sağlamasını gerektirir. UNIX 1970'lerde tasarlandığında ne çok çekirdekli işlemciler ne de SSD'ler vardı. Genellik, tasarımcının tek bir kullanım biçimine göre iyileştirme yapmasını engeller. Taşınabilirlik donanıma bağımlı kodu sınırlamayı zorunlu kılar. Geriye dönük uyumluluk ise yayımlanmış her arayüzü kalıcı yapar. Uzun bakım ve onarım ihtiyacı, uzun ömür ile uyumluluk zorunluluğunun doğal sonucudur.

Sık yapılan hata

Zorlukların hepsini “büyük yazılım” sorununa indirgemek. Boyut her büyük projede vardır. İşletim sistemini ayıran şey, eş zamanlılık, düşmanca kullanıcılar ve geriye dönük uyumluluğun aynı kodda bir araya gelmesidir. Bir uygulamadaki hata o uygulamayı düşürür; çekirdekteki hata ise üzerinde çalışan her şeyi.

14.1.2 Dört temel hedef

Zorlukların karşısında işletim sisteminin ne için var olduğunu açıkça koymak gerekir. Tanenbaum dört temel hedef sayar. Bu hedefler bir katman resmi gibi düşünülebilir: uygulamalar üstte, donanım altta, işletim sistemi aradadır (Şekil 14.1).



Şekil 14.1: İşletim sisteminin dört hedefi. Uygulamalar (tarayıcı, derleyici, veritabanı) soyutlamaları kullanır; soyutlamalar üzerindeki ilkel işlemler sistem çağrılarıdır; yalıtım her süreci kendi adres alanında tutar; en altta donanım yönetilir.

Tanım 14.1

Soyutlama (*abstraction*), donanımın karmaşıklığını gizleyen ve programcıya daha kullanışlı bir model sunan kavramdır: süreç, iş parçacığı, adres alanı, dosya, soket. **İlkel işlem** (*primitive*) ise bir soyutlama üzerinde çalışan temel işlemdir; işletim sisteminde ilkel işlemler sistem çağrılarıdır.

1. **Soyutlamalar tanımlamak.** Disk aslında numaralı bloklardan oluşan bir dizidir. Dosya ise adı, boyutu ve izinleri olan, bayt bayt okunan bir nesnedir. Aradaki farkı işletim sistemi kapatır. Tasarımın en zor kısmı doğru soyutlamayı bulmaktır. Çok ayrıntılı bir soyutlama donanımı yeterince gizlemez; çok kaba bir soyutlama ise bazı uygulamalara dar gelir.
2. **İlkel işlemler sağlamak.** Soyutlama ancak üzerinde işlem yapılabildiğinde işe yarar. Dosya için `open`, `read`, `write` ve `close`; süreç için `fork` ve `execve`; adres alanı için `mmap`; senkronizasyon için `futex` bu işlemlerdir.
3. **Yalıtım sağlamak.** Kullanıcılar ve süreçler birbirine karışmamalıdır. Her süreç kendi adres alanında çalışır; biri çökerse diğerleri etkilenmez. Yine de dosya izinleri ve paylaşımlı bellek gibi denetimli paylaşım yolları açık kalmalıdır.
4. **Donanımı yönetmek.** Kesmeler, zamanlayıcılar, MMU ve aygıt denetleyicileri bu hedefin konusudur. Düşük düzeyli işlerin çoğu bu katmanda yapılır; üstteki hedeflerin hepsi buna dayanır.

Dört hedef birbirinden bağımsız değildir. Yalıtım ancak donanım yönetimiyle, yani MMU ve ayrıcalık düzeyleriyle sağlanabilir. Soyutlamalar ise ilkel işlemler olmadan yalnız birer addır. Bu notun geri kalanı büyük ölçüde şu sorunun cevabıdır: bu hedefler hangi yapı ve hangi ilkelerle, ne kadar maliyetle gerçeklenir?

Sık yapılan hata

Programları derlemeyi işletim sisteminin hedefleri arasında saymak. Derleme bir araç zincirinin, yani derleyicinin işidir. İşletim sistemi derlenmiş programı belleğe yükler, onun için süreç oluşturur ve çalıştırır. Derleyici de işletim sisteminin üzerinde çalışan sıradan bir uygulamadır.

14.2 Arayüz tasarımı

İşletim sisteminin dış dünyaya gösterdiği yüz, sistem çağrısı arayüzüdür. Uygulamalar bu arayüze göre yazılır ve arayüz bir kez yayımlandıktan sonra pratikte değiştirilemez. Bu yüzden arayüzün nasıl tasarlandığı, çekirdeğin içinin nasıl yazıldığından daha uzun ömürlü bir karardır.

14.2.1 Yol gösterici ilkeler

İyi bir arayüz için kesin bir tarif yoktur, ama üç ilke tasarımcıya yol gösterir: basitlik, bütünlük ve verimlilik.

Tanım 14.2

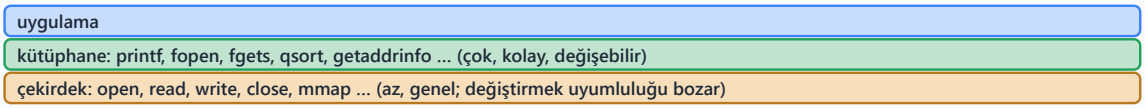
Basitlik (*simplicity*): arayüz az sayıda, iyi düşünülmüş kavramdan oluşmalıdır. **Bütünlük** (*completeness*): arayüz, kullanıcıların ihtiyaç duyduğu her şeyi yapabilmelidir. **Verimlilik** (*efficiency*): her özellik verimli gerçeklenebilmeli ve programcı bir çağrının yaklaşık maliyetini sezebilmelidir.

Basitlik için sıkça anılan söz Antoine de Saint-Exupéry'ye aittir: mükemmelliğe, eklenecek bir şey kalmadığında değil, çıkarılacak bir şey kalmadığında ulaşılır. Az sayıda kavramı öğrenmek, gerçeklemek ve doğrulamak kolaydır. Her yeni kavram, diğer bütün kavramlarla etkileşimi de beraberinde getirir.

Bütünlük basitliğin karşı ağırlığıdır. Einstein'a atfedilen söz bunu özetler: her şey olabildiğince basit olmalı, ama daha basit değil. Bir arayüz çok sade olduğu için gereken bir işi yapamıyorsa uygulamalar onu dolanmanın yollarını arar ve sonuç daha karmaşık olur. İdeal olan, az mekanizma ile çok iş yapabilmektir. MINIX 3 çekirdeği süreçler arası iletişim için yalnız birkaç ilkel sunar: `send`, `receive`, `sendrec` ve `notify`. Dosya sistemi, sürücüler ve bellek yönetimi bu birkaç ilkel üzerine kurulur.

Verimlilik ilkesi iki şey söyler. Birincisi, verimli gerçeklenemeyen bir özellik muhtemelen eklenmeye değmez. İkincisi, programcı bir çağrının yaklaşık maliyetini sezebilmelidir. `lseek` yalnız dosya bağıl konumunu değiştirir ve ucuzdur; `read` ise disk erişimi gerektirebilir ve pahalı olabilir. Ucuz görünen bir çağrı gizlice pahalıysa programcı yanlış kararlar verir.

Bu üç ilkeden pratik bir kural çıkar: her sistem çağrısı tek bir işi iyi yapmalıdır. Özellik eklemek kolaydır, kaldırmak ise neredeyse imkânsızdır. Geriye dönük uyumluluk yüzünden yanlış bir karar yıllarca taşınır.



Şekil 14.2: Kolaylık fonksiyonları (`printf`, `fopen`, `qsort`, `getaddrinfo`) kütüphanededir; çok sayıdadır ve değişebilir. Çekirdekteki çağrılar (`open`, `read`, `write`, `mmap`) az ve geneldir; değiştirmek uyumluluğu bozar.

Örnek 14.1

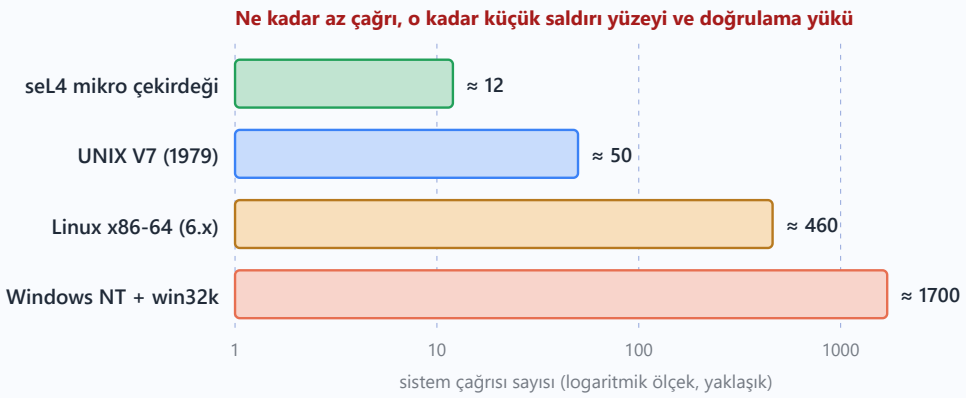
Her ilke için UNIX arayüzünden bir örnek verilebilir mi?

1. **Basitlik.** UNIX'te dosyalar, aygıtlar ve borular aynı dört çağrıyla kullanılır: `open`, `read`, `write`, `close`. Programcı tek bir model öğrenir.

2. **Bütünlük.** Altı farklı `exec` kütüphane fonksiyonu vardır, ama hepsi tek bir `execve` çağrısına dönüşür. `execve` yol, argüman dizisi ve ortam dizisi aldığı için her ihtiyacı karşılar.
3. **Verimlilik.** `lseek` yalnız bir sayıyı değiştirir, `read` ise disk okuyabilir. İki çağrının maliyet farkı adlarından ve tanımlarından sezilir.

14.2.2 Küçük güzeldir

Basitlik ilkesinin en somut ölçüsü sistem çağrısı sayısıdır. Az sayıda genel çağrı, çok sayıda özel çağrıdan iyidir. Gerçek sistemler bu ölçüde birbirinden büyük farklarla ayrılır (Şekil 14.3).



Şekil 14.3: Yaklaşık sistem çağrısı sayıları, logaritmik ölçekte: seL4 ≈ 12, UNIX V7 (1979) ≈ 50, Linux x86-64 (6.x) ≈ 460, Windows NT ve win32k birlikte ≈ 1700.

Ölçek logaritmik olduğu için çubuklar arasındaki fark gözü yanıltabilir. Windows'un çağrı sayısı seL4'ünkünün yaklaşık 140 katı, Linux'unki UNIX V7'ninkinin dokuz katından fazladır. Çağrı sayısı arttıkça saldırı yüzeyi de büyür: her çağrı, kötü niyetli bir girdinin çekirdeğe ulaşabileceği bir kapıdır.

seL4 mikro çekirdeği bu ilkenin uç örneğidir. Çekirdeği yaklaşık 10 bin satırdır ve biçimsel olarak doğrulanmıştır: C kodunun matematiksel bir belirtime uyduğu ispatlanmıştır. Milyonlarca satırlık bir çekirdekte böyle bir ispat bugünkü yöntemlerle mümkün değildir.

Linux'ta ise çağrı sayısı her yıl birkaç tane artar. Yakın zamanda eklenen `io_uring_*`, `landlock_*` ve `mseal` çağrıları buna örnektir. Windows farklı bir yol izler: belgelenmiş arayüz çekirdek çağrıları değil, Win32 API'dir. Çekirdek çağrıları sürümden sürüme değişebilir, çünkü uygulamalar onları doğrudan kullanmaz. Bu düzende araya giren kütüphane katmanı uyumluluğu korur.

Sık yapılan hata

Küçük arayüzü “az özellikli sistem” sanmak. UNIX V7'nin elli kadar çağrısı borular, yönlendirme ve süreç oluşturmayla zengin bir programlama ortamı kurmaya yetiyordu. Küçüklüğün amacı özelliği kısmak değil, özellikleri az sayıda genel mekanizmanın birleşimiyle elde etmek-

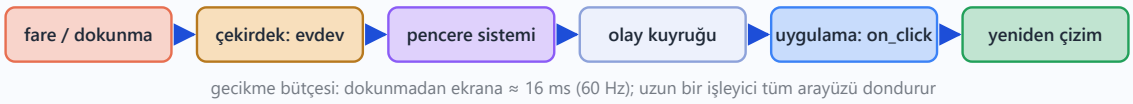
tir.

14.2.3 Kullanıcı arayüzü paradigmaları

Kullanıcı arayüzü çoğu zaman çekirdeğin parçası değildir. Yine de seçilen paradigma bütün sistemin tasarımını etkiler. Dört paradigma bugün yan yana yaşar.

- **Komut satırı** (CLI, *command-line interface*). Kabuk, metin komutları ve borular. Otomasyon ve uzaktan yönetim için hâlâ vazgeçilmezdir.
- **WIMP / GUI.** Pencere, simge, menü ve işaretçi (*windows, icons, menus, pointer*). Fare tuşlarının anlamı her uygulamada aynı olmalıdır.
- **Dokunmatik ve mobil.** Kaydırma ve yakınlaştırma gibi hareketler, tam ekran uygulamalar, bildirim merkezi.
- **Ses, sohbet ve ajanlar.** Sesli asistanlar ve doğal dil arayüzleri işletim sistemi hizmetlerine yeni bir giriş noktası açar.

Paradigmanın çekirdeğe etkisi en açık biçimde grafik arayüzde görülür. Bir GUI, olay güdümlü bir yürütme modeli, hızlı bir pencere sistemi ve tutarlı bir uygulama çerçevesi gerektirir. Bir tıklama, aygıttan uygulamaya uzanan bir yol izler (Şekil 14.4).



gecikme bütçesi: dokunmadan ekrana ≈ 16 ms (60 Hz); uzun bir işleyici tüm arayüzü dondurur

Şekil 14.4: Bir tıklamanın yolu: girdi aygıtı, çekirdekteki evdev sürücüsü, pencere sistemi, uygulamanın olay kuyruğu ve `on_click` işleyicisi; sonunda ekran yeniden çizilir. 60 Hz'lik bir ekranda dokunmadan ekrana bütçe yaklaşık 16 ms'dir.

16 ms'lik bütçe, 1000 ms / 60 ≈ 16,7 ms hesabından gelir. Bu süre içinde olay çekirdekten pencere sistemine, oradan uygulamaya geçmeli, işlenmeli ve yeni kare çizilmelidir. Uzun süren tek bir işleyici bütün arayüzü dondurur. Paradigmalar arasında ortak olan ilke tutarlılıktır: aynı eylem her yerde aynı sonucu vermelidir.

14.2.4 Yürütme paradigmaları

Bir programın akışını kimin belirlediği, işletim sisteminden ne isteyeceğini de belirler. İki temel paradigma vardır.

Tanım 14.3

Algoritmik kodda program akışını program belirler; işletim sistemine yalnız hizmet almak için çağrı yapar. **Olay güdümlü kodda** (*event-driven*) akışı dış olaylar belirler: tuş, tıklama, ağ paketi, zamanlayıcı. Program bir döngüde sıradaki olayı bekler ve ona uygun işleyiciyi çağırır.

Algoritmik kodun tipik biçimi aşağıdakine benzer. Program başlar, sırayla bir şeyler yapar, gerektiğinde okur ve yazar, sonunda biter. Derleyiciler, sıralama programları ve toplu işler bu yapıdadır.

```
int main(void)
{
    int x, y;
    init();
    do_something();
    read(fd, &x, sizeof x);
    do_something_else();
    write(fd, &y, sizeof y);
    keep_going();
    exit(0);
}
```

Olay güdümlü kod ise bir döngü etrafında kurulur. `get_message` sıradaki olayı getirir, `switch` onu türüne göre ilgili koda yönlendirir. GUI'ler, ağ sunucuları ve mikro çekirdek sunucuları bu yapıdadır.

```
int main(void)
{
    msg_t msg;
    init();
    while (get_message(&msg)) {
        switch (msg.type) {
        case 1: ... ; break;
        case 2: ... ; break;
        case 3: ... ; break;
        }
    }
}
```

Paradigma seçimi sistem çağrılarını da biçimlendirir. Algoritmik model engelleyen (*blocking*) bir `read` ister: veri gelene kadar beklenir, sonra akış kaldığı yerden sürer. Olay güdümlü model ise "sıradaki olayı ver" türünden bir çağrı ister: `get_message`, `epoll_wait` ya da `io_uring_enter`. Bir işletim sistemi iki paradigmayı da desteklemek zorundadır.

14.2.5 Kod izleme: olay döngüsü

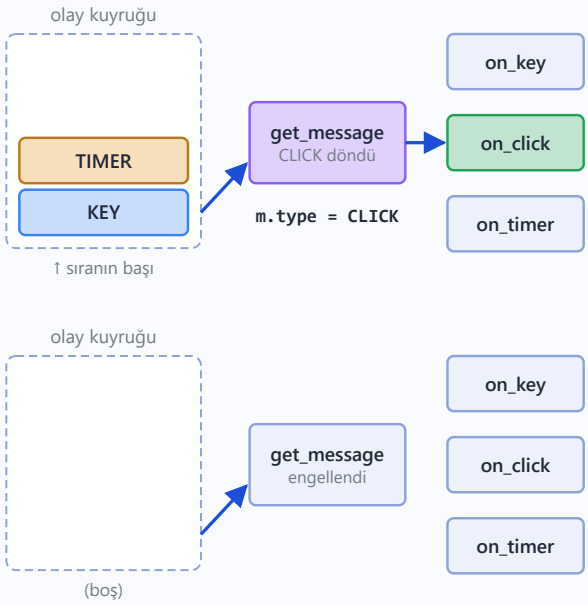
Olay döngüsünün nasıl çalıştığını somut bir kodla izlemek mümkündür. Aşağıdaki programda üç olay türü vardır: tuş (`KEY`), tıklama (`CLICK`) ve zamanlayıcı (`TIMER`). Her biri kendi işleyicisine gider.

```
int main(void) {
    msg_t m;
    init();
    while (get_message(&m)) {
        switch (m.type) {
        case KEY:   on_key(&m);   break;
        case CLICK: on_click(&m); break;
        case TIMER: on_timer(&m); break;
        }
    }
    return 0;
}
```

`init()` pencereyi ve kaynakları hazırlar. `while` koşulu `get_message(&m)` çağrısının dönüş değeridir. Çağrı bir olay döndürürse 1, programın kapanması gerektiğini bildiren `QUIT` olayında ise 0 döner. `switch` olay türüne göre `on_key` , `on_click` ya da `on_timer` işleyicisini çağırır. İşleyici bitince döngü başa döner. İzleme sekiz adımdır (Tablo 14.2).

Adım	Kuyruk	get_message	Çalışan kod
1	boş	henüz çağrılmadı	`init()`
2	CLICK, KEY, TIMER	olay bekliyor	pencere sistemi olayları ekler
3	KEY, TIMER	CLICK döndü	`on_click`
4	TIMER	KEY döndü	`on_key`
5	boş	TIMER döndü	`on_timer`
6	boş	engellendi	süreç uyur
7	QUIT	olay bekliyor	süreç uyanır
8	boş	0 döndü	döngü biter, `return 0`

Tablo 14.2: Olay döngüsünün izlemesi. Kuyruğa sırayla CLICK, KEY ve TIMER, en sonda QUIT gelir.



Şekil 14.5: (a) `get_message` sıranın başındaki CLICK'i döndürmüş, `switch` onu `on_click`'e yönlendirmiştir. (b) Kuyruk boşalınca `get_message` engellenir; süreç uyur ve CPU harcamaz.

İzlemenin iki noktası önemlidir. Birincisi altıncı adımdır. Kuyruk boşken `get_message` hemen dönmez, süreci uyutur. Süreç bu sırada CPU harcamaz; yeni bir olay geldiğinde çekirdek onu uyandırır. Döngü bir "meşgul bekleme" değil, engelleyen bir çağrı etrafında kurulmuş bir yapıdır.

İkincisi işleyicilerin süresidir. Döngü tek iş parçacıklıdır; bir işleyici çalışırken kuyruktaki diğer olaylar bekler. `on_timer` imleci yakıp söndürmek gibi kısa bir iş yapar. Ama bir işleyici büyük bir dosyayı okumaya kalkarsa arayüz o süre boyunca donar. Uzun işler bu yüzden ayrı bir iş parçacığına verilir. Aynı kalıp nginx ve Node.js gibi sunucularda `epoll` ile, mikro çekirdek sunucularında da ileti kuyruğuyla kullanılır.

Örnek 14.2

İşleyiciler bir sayaç tutsun: `KEY` sayacı 1, `CLICK` 10 artırsın; `TIMER` sayaç 10'dan büyükse onu sıfırlasın. Olaylar KEY CLICK KEY TIMER CLICK KEY KEY QUIT sırasıyla gelirse döngü sonunda sayaç kaçtır?

1. **KEY, CLICK, KEY.** 0 → 1 → 11 → 12.
2. **TIMER.** 12 > 10 olduğu için sayaç sıfırlanır: 0.
3. **CLICK, KEY, KEY.** 0 → 10 → 11 → 12.
4. **QUIT.** `get_message` 0 döndürür, döngü biter. Sonuç 12'dir.

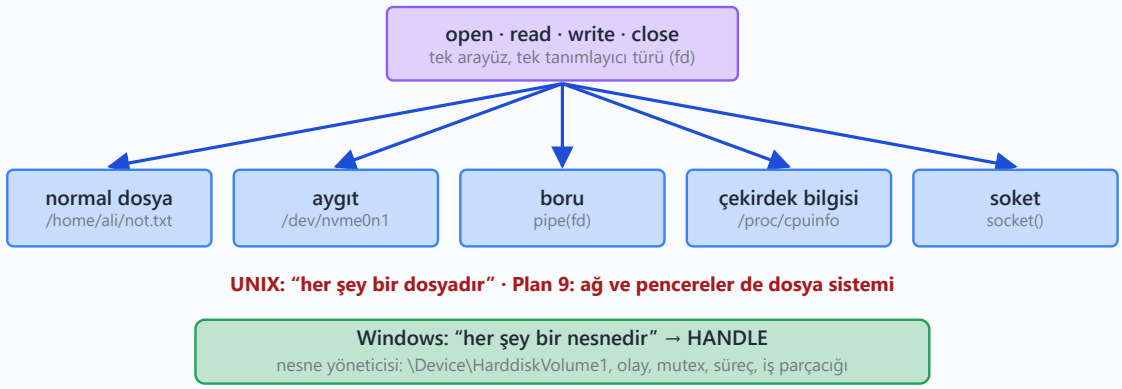
Sonucu programın metni değil, olayların sırası belirler. Aynı olaylar farklı sırayla gelseydi sayaç da farklı çıkardı.

Sık yapılan hata

Olay döngüsünün kuyruk boşken CPU'yu sürekli meşgul ettiğini sanmak. `get_message` engelleyen bir çağrıdır; olay yoksa süreç uyku durumuna geçer. Döngünün kendisi değil, uzun süren işleyiciler sorundur.

14.2.6 Veri paradigmaları

Yürütme paradigması akışı, veri paradigması ise kaynakların nasıl adlandırılıp kullanıldığını belirler. İki büyük gelenek vardır (Şekil 14.6).



Şekil 14.6: Veri paradigmaları. UNIX'te tek bir dosya arayüzü (`open` , `read` , `write` , `close`) normal dosyaya, aygıta, boruya, `/proc` bilgisine ve sokete uygulanır. Windows'ta her şey bir nesnedir ve bir HANDLE ile kullanılır.

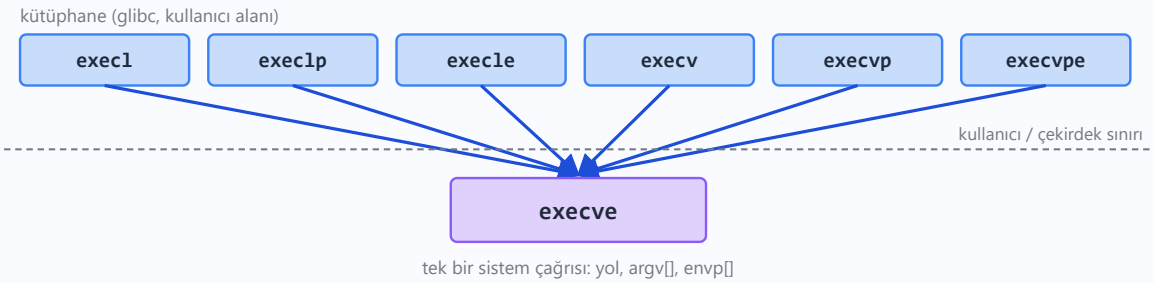
UNIX'in ilkesi "her şey bir dosyadır" biçiminde özetlenir. Normal dosya `/home/ali/not.txt` , NVMe diski `/dev/nvme0n1` , bir boru, çekirdek bilgisi `/proc/cpuinfo` ve bir soket aynı tanımlayıcı türüyle, yani dosya tanımlayıcısıyla (fd) kullanılır. Plan 9 bu fikri daha da ileri götürmüş, ağı ve pencereleri de dosya sistemi olarak sunmuştur. Windows'ta ise her şey bir nesnedir. Nesne yöneticisi dosyaları, olayları, mutex'leri, süreçleri ve iş parçacıklarını `\Device\HarddiskVolume1` gibi adlarla tek bir ad alanında tutar; program onlara bir HANDLE ile erişir.

Ortak bir veri paradigmasının kazancı, aynı araçların her kaynakta çalışmasıdır. `cat` , `dd` ve kabuk yönlendirmesi hem dosyada hem aygıtta hem de çekirdek bilgisinde iş görür. `cat /proc/loadavg` komutu sistemin yük ortalamasını, hiçbir özel araç gerekmeden gösterir.

Bedeli de vardır. Bazı aygıtlar dosya modeline sığmaz. Bir kameranın çözünürlüğünü ayarlamak ya da bir ağ kartının ayarlarını değiştirmek `read` ve `write` ile doğal biçimde ifade edilemez. O zaman her şey `ioctl` gibi "kaçış kapılarına" yığılır. `ioctl` aygıta göre anlamı değişen bir komut numarası alır ve tek bir çağrının arkasında yüzlerce farklı işlem saklanır. Karmaşıklık arayüzün yüzeyinden kalkmış, ama içeride kaybolmamıştır.

14.2.7 Sistem çağrısı arayüzü

Arayüz ilkelerinin en somut uygulaması sistem çağrılarının kütüphaneyle ilişkisidir. Çekirdekte az ve genel çağrı, kütüphanede ise kolaylık fonksiyonları bulunur (Şekil 14.7).



Şekil 14.7: Altı `exec` kütüphane fonksiyonu (`execl` , `execlp` , `execle` , `execv` , `execvp` , `execvpe`) kullanıcı alanındadır; hepsi tek bir `execve` sistem çağrısına dönüşür.

Altı fonksiyon argümanların nasıl verildiğine (liste ya da dizi), programın `PATH` içinde aranıp aranmadığına ve ortamın açıkça verilip verilmediğine göre ayrılır. Bu farkların hepsi kullanıcı alanında çözülür. Çekirdek yalnız bir yol, bir argüman dizisi ve bir ortam dizisi görür. Yeni bir kolaylık eklemek çekirdeği değiştirmez; yalnız kütüphane büyür.

İkinci ilke bağımsız bileşenlerdir. UNIX süreç oluşturmayı iki çağrıya böler: `fork` ve `execve` . Arada çocuk süreç kendi ortamını hazırlayabilir: girdi ve çıktıyı yönlendirebilir, çalışma dizinini değiştirebilir, gereksiz dosyaları kapatabilir. Windows'ta ise `CreateProcess` tek çağrıda on parametre alır. Bu çağrı güçlüdür, ama her yeni ihtiyaç yeni bir parametre ya da bayrak demektir.

Modern eğilimler iki yönde ilerler. `posix_spawn` , `fork` ile tüm adres alanını kopyalama maliyetinden kaçınmak için süreç oluşturmayı tek adımda yapar. `io_uring` ise toplu çağrı arayüzüdür: tek bir kullanıcı–çekirdek geçişinde çok sayıda istek gönderilir. Her iki eğilim de maliyetin sezilebilir olması gerektiği ilkesinden doğmuştur.

> **Sık yapılan hata**
>
> `printf` ya da `fopen` 'ı sistem çağrısı sanmak. Bunlar C kütüphanesi fonksiyonlarıdır; içlerinde gerektiğinde `write` ya da `openat` çağrısı yapılır. `printf` çoğu zaman yalnız kullanıcı alanındaki bir tampona yazar ve çekirdeğe hiç girmez.

14.3 Sistem yapısı

Arayüz dışarıdan görünen yüzdür; yapı ise çekirdeğin içinin nasıl bölündüğünü anlatır. Hangi kod hangi adres alanında ve hangi ayrıcalıkla çalışır? Bir bileşen diğerini nasıl çağırır? Bu soruların cevabı sistemin hem hızını hem de hatalara dayanıklılığını belirler.

14.3.1 Yapı seçenekleri: yalıtım ve hız

Yapı seçenekleri genellikle "monolitik", "katmanlı", "mikro çekirdek" gibi adlarla sayılır. Bu adların arkasında aslında birer tasarım kararı vardır. Her karar bir şey kazandırır ve bir bedel ister (Tablo 14.3).

Tasarım kararı	Kazanılan	Ödenen	Örnek
Her şeyi tek adres alanında doğrudan çağırmak	Hız: tek tuzak, doğrudan fonksiyon çağrıları	Hata yalıtımı yok: bir sürücü hatası sistemi düşürür	Linux, FreeBSD
Bağımlılığı tek yöne sınırlamak (katmanlar)	Katman katman test; değişiklik yerel kalır	Katman sırasını seçmek; her geçişin maliyeti	THE, MULTICS
Hizmetleri ayrı adres alanlarına koymak	Çöken sunucu yeniden başlatılır; küçük, doğrulanabilir çekirdek	Her istekte IPC, bağlam değiştirme, kopyalama	MINIX 3, QNX, seL4, Zircon
Korumayı yönetimden ayırmak	Uygulamaya özel soyutlama (libOS)	Uygulamalar arası paylaşım zor; libOS karmaşıklığı	Aegis/ExOS, unikernel'ler
Çalışırken kod yüklemek	Yeniden derlemeden uyarlama	Eklentinin güvenliği: doğrulayıcı ya da güvenli dil gerekir	SPIN, Linux eBPF
Saf modelden ödün vermek	Mikro çekirdek düzeni, monolitik hız	Yalıtımın bir kısmı kaybolur	Windows NT, macOS XNU

Tablo 14.3: Sistem yapısı seçenekleri tasarım kararı olarak.

Tablodaki satırların hepsi aynı ödünleşimin farklı noktalarıdır. Temel ödünleşim yalıtım ile performans arasındadır. Yalıtım sınırı her geçildiğinde bir bedel ödenir: bir tuzak, bir bağlam değiştirme ya da bir veri kopyası. Sınır ne kadar çoksa hatalar o kadar iyi hapsedilir, ama sistem o kadar yavaşlar.

Tanım 14.4

Monolitik çekirdekte (*monolithic kernel*) dosya sistemi, sürücüler ve bellek yönetimi dahil bütün hizmetler tek bir ayrıcalıklı adres alanında çalışır ve birbirini doğrudan çağırır. **Mikro çekirdekte** (*microkernel*) çekirdek yalnız IPC, çizelgeleme ve bellek eşleme gibi en temel işleri yapar. Diğer hizmetler kullanıcı alanında ayrı sunucu süreçleri olarak çalışır ve iletilerle haberleşir.

İki yapının farkı tek bir dosya okumasında açıkça görülür (Şekil 14.8). Monolitik çekirdekte uygulama bir tuzakla çekirdeğe girer; VFS, ext4 ve disk sürücüsü birbirini doğrudan fonksiyon çağrısıyla çağırır. Mikro çekirdekte uygulamanın isteği bir ileti olarak dosya sunucusuna, oradan başka bir iletiyle disk sürücüsüne gider; cevap da aynı yoldan döner. Her ileti en az bir çekirdek geçişi ve bağlam değiştirme demektir.



Şekil 14.8: Solda monolitik çekirdek: tek adres alanı, bir tuzak ve doğrudan çağrılar. Sağda mikro çekirdek: uygulama, dosya sunucusu ve disk sürücüsü ayrı süreçlerdir; çekirdek yalnız IPC, çizelgeleme ve bellek eşleme yapar.

Örnek 14.3

Bir ekip ağ sürücüsünü çekirdekten çıkarıp kullanıcı alanında ayrı bir sunucu süreci olarak çalıştırmayı düşünüyor. Sürücüde bir NULL işaretçi hatası olduğunda iki tasarımda ne olur ve kullanıcı alanı tasarımının bedeli nedir?

1. **Monolitik çekirdek.** Sürücü çekirdek adres alanındadır. Hatalı erişim bir çekirdek hatasıdır (oops); çoğu zaman sistem kararsız kalır ya da tamamen çöker.
2. **Kullanıcı alanı sürücüsü.** Hata yalnız sürücü sürecini öldürür. MINIX 3'te reenkarnasyon sunucusu bunu fark eder ve sürücüyü yeniden başlatır; diğer süreçler çalışmayı sürdürür.
3. **Bedel.** Her paket artık çekirdek ile sürücü süreci arasında iletiyle taşınır. Her ileti bir bağlam değiştirme ve çoğu zaman bir veri kopyası ekler; paket hızı düşer.

Seçenekler birbirini dışlamaz. Linux monolitiktir, ama yüklenebilir modüller ve eBPF ile çalışırken genişletilebilir; çekirdek iş parçacıkları da kullanır. Windows NT ve macOS XNU mikro çekirdek fikirleriyle başlamış, ama performans için hizmetlerin çoğunu çekirdek adres alanına geri almıştır. Bu karma yapılar, saf modellerin ikisinin de pratikte ödün vermek zorunda kaldığını gösterir.

Sık yapılan hata

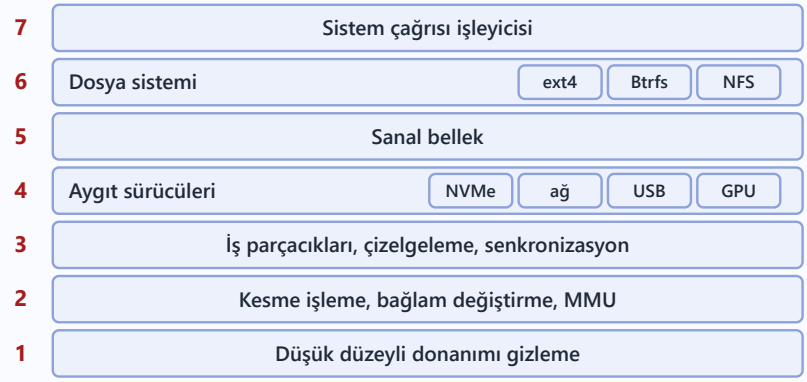
Mikro çekirdeğin her durumda daha yavaş olduğunu varsaymak. L4 ve seL4 gibi modern mikro çekirdeklerde IPC birkaç yüz döngüye kadar indirilmiştir. Asıl maliyet çoğu zaman tek bir iletide değil, bir istek için gereken ileti sayısında ve verinin kaç kez kopyalandığındadır.

14.3.2 Katmanlı sistemler

Katmanlı yapı, bağımlılığı tek yöne sınırlamanın en eski yoludur. Dijkstra'nın THE sistemi (1968) bu fikirle kurulmuştu. Her katman yalnız altındaki katmanın sunduğu soyutlamaları kullanır, üstündekini hiç bilmez. Tanenbaum modern bir işletim sistemi için yedi katmanlı bir tasarım önerir (Şekil 14.9).

Tanım 14.5

Katmanlı yapıda (*layered system*) sistem, her biri yalnız altındaki katmanın hizmetlerini kullanan sıralı katmanlara bölünür. Bir katman, alttaki katmanlar doğru çalışıyorsa tek başına test edilebilir.

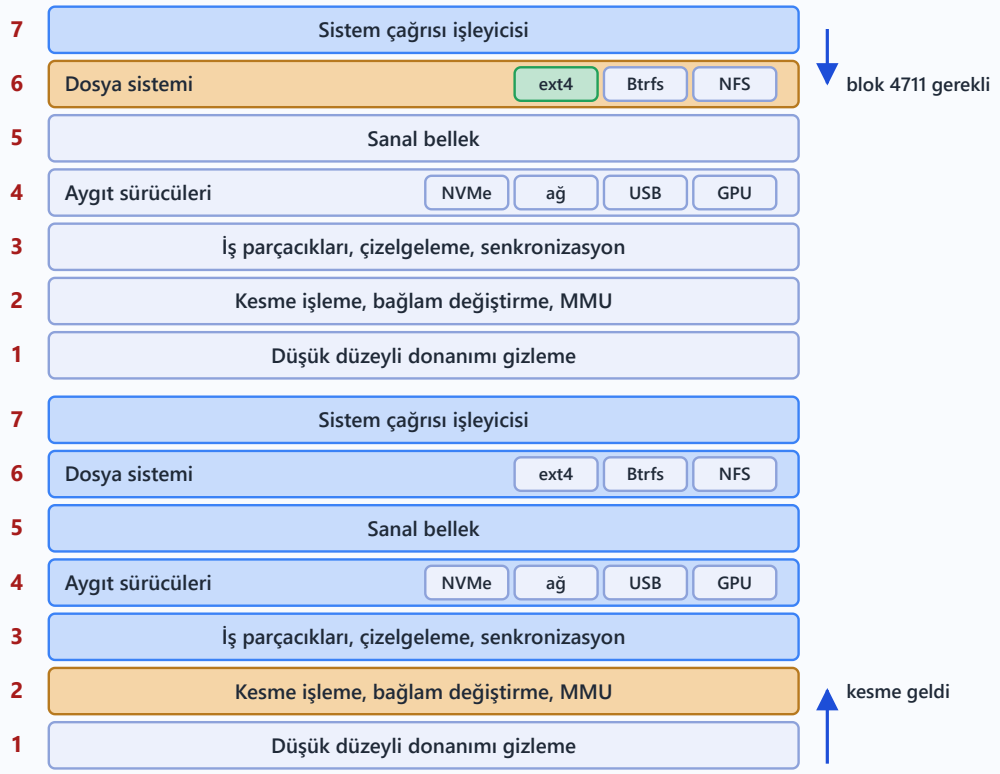


Şekil 14.9: Tanenbaum'un yedi katmanlı tasarımı. 1: düşük düzeyli donanımı gizleme; 2: kesme işleme, bağlam değiştirme, MMU; 3: iş parçacıkları, çizelgeleme, senkronizasyon; 4: aygıt sürücüleri; 5: sanal bellek; 6: dosya sistemleri; 7: sistem çağrısı işleyicisi.

Katmanların sırası keyfî değildir; bağımlılıklardan çıkar. Sanal bellek sayfaları diske yazdığı ve diskten okuduğu için aygıt sürücülerinin üstündedir. Dosya sistemi sayfa önbelleği için çerçeve ayırdığı için sanal belleğin üstündedir. İş parçacıkları kesme işlemeye ve bağlam değiştirmeye dayandığı için 2. katmanın üstündedir. Donanımı gizleyen 1. katman taşınabilirliği sağlar: yeni bir işlemciye taşırken öncelikle bu katman yeniden yazılır.

Bir `read` çağrısının katmanlardan geçişi yapıyı canlı olarak gösterir.

1. **7. katman.** Uygulama `read(fd, buf, n)` çağırır ve tuzak sistem çağrısı işleyicisine gelir. Parametreler burada denetlenir: `fd` geçerli mi, tampon kullanıcı alanında mı?
2. **6. katman.** Dosya sistemi (örneğin ext4) dosya bağıl konumunu disk bloğuna çevirir; okunacak blok 4711'dir. Blok önbellekte yoksa alt katmanlardan istenir.
3. **5. katman.** Sanal bellek, verinin konacağı sayfa önbelleği çerçevesini ayırır ve kullanıcı tamponunun eşlemesini doğrular.
4. **4. katman.** NVMe sürücüsü okuma komutunu aygıtın kuyruğuna yazar.
5. **3. katman.** Çağıran iş parçacığı G/Ç bitene kadar engellenir; çizelgeleyici başka bir iş parçacığını çalıştırır.
6. **2. katman.** Aygıt işi bitirince kesme üretir. Kesme işleyicisi uyuyan iş parçacığını hazır yapar. Sonuç katmanlar boyunca yukarı döner ve `read` okunan bayt sayısını döndürür.



Şekil 14.10: `read` çağrısının yolu. (a) Dosya sistemi katmanında bağıl konum blok 4711'e çevrilir. (b) Aygıt bitince 2. katmanda kesme gelir ve sonuç yukarı döner.

Katmanlı yapının bedeli her katman geçişinin maliyetidir. Saf katmanlı bir sistemde bir isteğin bütün katmanlardan tek tek geçmesi gerekir ve her geçiş bir fonksiyon çağrısı, bazen bir parametre kopyası ekler. Asıl zorluk ise katman sırasını seçmektir. Gerçek sistemlerde bağımlılıklar her zaman tek yönlü değildir. Sanal bellek sürücüleri kullanır, ama bir sürücü de kendi tamponları için bellek isteyebilir. Bu yüzden bugünkü sistemler katman fikrini bir düşünme aracı olarak kullanır, katı bir kural olarak değil.

> **Sık yapılan hata**
>
> Katman numarasını önem sırası sanmak. 7. katman 1. katmandan "daha önemli" değildir; yalnız daha fazla alt katmana dayanır. Sıra bağımlılığı gösterir: bir katman ancak altındakilerin hepsi hazırsa çalışır.

14.3.3 Genişletilebilir sistemler

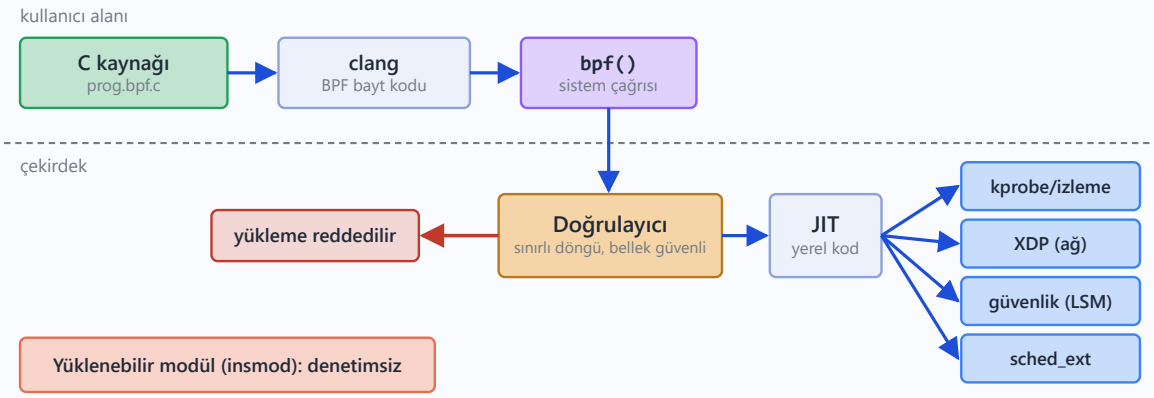
İhtiyaçlar değiştikçe çekirdeğe yeni yetenekler eklemek gerekir. Her seferinde çekirdeği yeniden derleyip makineyi yeniden başlatmak pratik değildir. Genişletilebilir sistemlerde çekirdek temel işlevleri sağlar, yeni bileşenler ise çalışırken eklenir.

Tanım 14.6

Genişletilebilir sistem (*extensible system*), çalışırken çekirdeğe yeni kod yüklenmesine izin veren sistemdir. **eBPF** (*extended Berkeley Packet Filter*), Linux'ta küçük ve doğrulanmış programların çekirdek kancalarına bağlanarak çalıştırılmasını sağlayan mekanizmadır.

Genişletmenin en büyük sorunu güvenliktir. Üçüncü taraflar sisteme özellik ekleyebildiğinde, eklenen kodun çekirdeği bozmaması da sağlanmalıdır. Klasik yüklenebilir modül (`insmod`) tam yetkilidir: modüldeki bir hata çekirdeğin kendisindeki bir hatadan farksızdır ve sistemi çökertebilir. 1995'teki SPIN sistemi bu sorunu güvenli bir dille çözmeye çalışmış, eklentileri Modula-3 ile yazdırarak bellek hatalarını derleme anında engellemişti.

Linux'taki eBPF bugünkü cevaptır (Şekil 14.11). Program C ile yazılır ve `clang` ile BPF bayt koduna derlenir. Kullanıcı alanı bu bayt kodunu `bpf()` sistem çağrısıyla çekirdeğe verir. Çekirdekteki **doğrulayıcı** (*verifier*) programın bütün yollarını inceler. Döngüler sınırlı mı, her bellek erişimi güvenli mi? Program bir koşulu sağlamazsa yükleme reddedilir. Geçen program JIT derleyicisiyle yerel makine koduna çevrilir ve bir kancaya bağlanır.



Şekil 14.11: eBPF programının yolu: C kaynağı `clang` ile bayt koduna derlenir, `bpf()` çağrısıyla çekirdeğe gelir, doğrulayıcıdan geçer ve JIT ile yerel koda çevrilip kancalara (izleme, XDP, LSM, sched_ext) bağlanır. Yüklenebilir modül ise denetimsizdir.

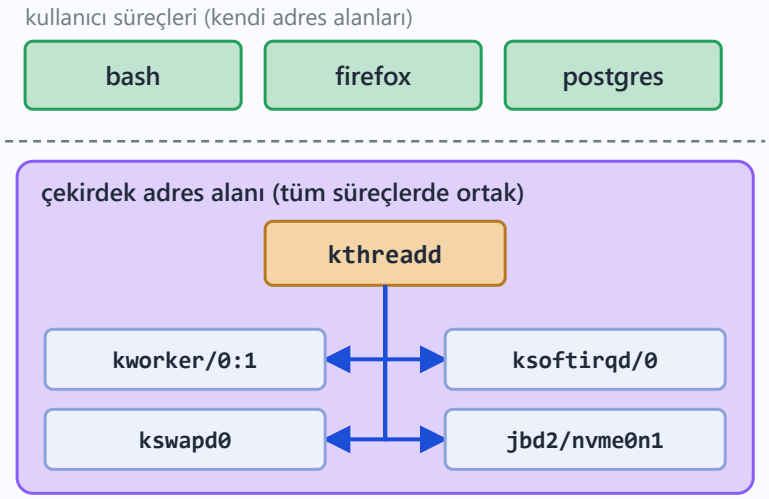
eBPF kancaları bugün dört alanda kullanılır: izleme (kprobe ve tracepoint'ler), ağ (XDP ile paketlerin sürücü düzeyinde işlenmesi), güvenlik (LSM kancaları) ve Linux 6.12'den beri çizelgeleme (sched_ext). eBPF'in önemi, yalıtım ile genişletilebilirliği aynı anda sağlamasıdır. Kod çekirdek adres alanında çalıştığı için hızlıdır, doğrulayıcı sayesinde de çekirdeği çökertemez.

14.3.4 Çekirdek iş parçacıkları

Çekirdeğin bazı işleri bir sistem çağrısına ya da kesmeye bağlı değildir. Bellek azaldığında sayfa geri kazanmak, günlüğü diske yazmak, kesmede yapılamayacak kadar uzun işleri bitirmek bunlara örnektir. Bu işler için çekirdek kendi iş parçacıklarını kullanır (Şekil 14.12).

Tanım 14.7

Çekirdek iş parçacığı (*kernel thread*), çekirdeğin oluşturup yönettiği ve yalnız çekirdek adres alanında çalışan iş parçacığıdır. Kullanıcı adres alanı yoktur. Linux'ta `kthread_create` ile oluşturulur.



Şekil 14.12: Kullanıcı süreçlerinin (bash, firefox, postgres) her birinin kendi adres alanı vardır. Çekirdek iş parçacıkları (kthreadd, kworker, ksoftirqd, kswapd, jbd2) ise bütün süreçlerde ortak olan çekirdek adres alanında çalışır.

Çekirdek iş parçacıkları normal iş parçacıkları gibi zamanlanır, engellenebilir ve uyuyabilir. Kesme işleyicisinde uyumak yasaktır; bu yüzden kesmede yapılamayan uzun işler bir çekirdek iş parçacığına ertelenir. Bağlam değiştirme de ucuzdur, çünkü adres alanı, yani sayfa tablosu değişmez. Linux'taki tipik örnekler şunlardır:

- `kworker` : ertelenmiş işleri (*workqueue*) yürütür.
- `ksoftirqd` : yumuşak kesmeleri işler; ağ paketlerinin işlenmesi gibi.
- `kswapd` : boş bellek azaldığında sayfa geri kazanır.
- `jbd2` : ext4'ün günlüğünü diske yazar.

Hepsinin atası `kthreadd`'dir. `ps -ef` çıktısında köşeli parantez içinde görünürler, örneğin `[kswapd0]`. Kullanıcı süreçleri çekirdek iş parçacığı oluşturamaz. Sistem çağrıları yalnız çekirdeğin zaten çalıştırdığı iş parçacıklarına iş verebilir; `io_uring`'in iş havuzu buna bir örnektir. Çekirdek iş parçacıkları yapıyı bir adım daha esnek hâle getirir: monolitik bir çekirdek de işini, ayrı ayrı zamanlanan bağımsız akışlara bölebilir.

14.4 Tasarım ilkeleri

Yapı seçildikten sonra tasarımcı yüzlerce küçük kararla karşılaşır. Bu kararlarda tekrar tekrar işe yarayan birkaç ilke vardır. Bu ilkeler tek bir algoritma değil, bir düşünme biçimidir; aynı ilke bellek yönetiminde de dosya sisteminde de karşımıza çıkar.

14.4.1 Mekanizma ve politikanın ayrılması

Tanım 14.8

Mekanizma (*mechanism*) bir işin nasıl yapılacağını, **politika** (*policy*) neyin yapılacağına karar vermeyi belirler. Mekanizma çekirdekte sabit kalır; politika değiştirilebilir ve çoğu zaman kullanıcı alanında belirlenir.

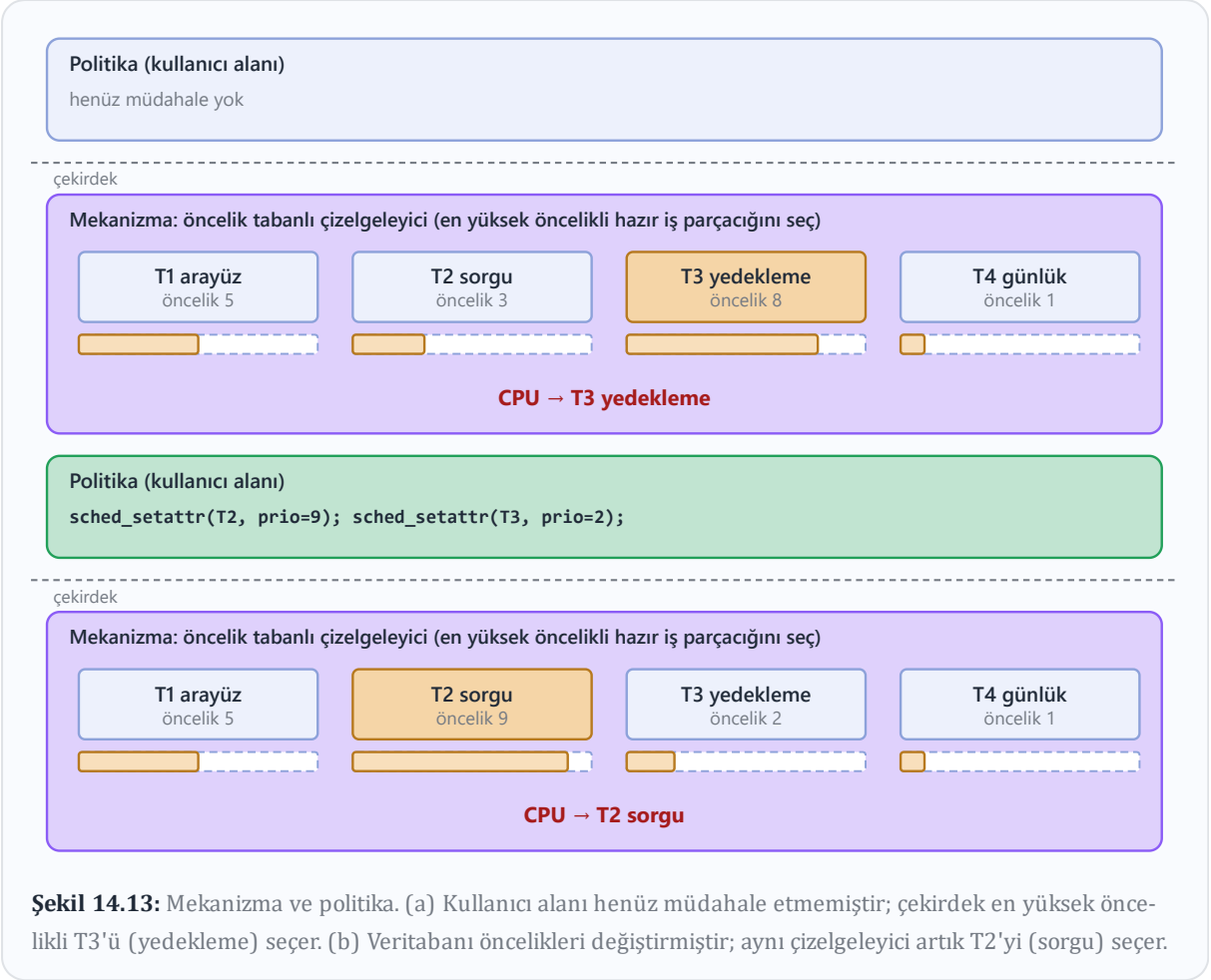
Bu ayrımın değeri, çekirdeği değiştirmeden farklı davranışlar elde edebilmektir. Çekirdek her durumu öngöremez. Ama iyi bir mekanizma sunarsa, durumu en iyi bilen taraf, yani uygulama ya da yönetici, politikayı kendisi seçer. Çizelgeleme bu ilkenin en açık örneğidir (Şekil 14.13).

Örnek 14.4

Bir veritabanı sürecinin dört iş parçacığı var: T1 arayüz (öncelik 5), T2 sorgu (3), T3 yedekleme (8), T4 günlük (1). Çekirdeğin mekanizması “en yüksek öncelikli hazır iş parçacığını çalıştır”dır. Hangi iş parçacığı çalışır ve politika değişince ne olur?

1. **Başlangıç.** En yüksek öncelik 8'dir; CPU T3'e, yani yedeklemeye verilir. Sorgular bekler.
2. **Politika.** Veritabanı kendi iş parçacıklarını en iyi bilen taraftır. Sorguların önemli olduğuna, yedeklemenin bekleyebileceğine karar verir ve bunu bir sistem çağrısıyla bildirir: `sched_setattr(T2, prio=9)`, `sched_setattr(T3, prio=2)`.
3. **Sonuç.** Öncelikler artık 5, 9, 2, 1'dir. Çekirdek kodu hiç değişmeden T2'yi seçer.
4. **Başka bir politika.** Etkileşimli bir uygulama arayüzü öne alır: öncelikler 9, 6, 2, 1 olur ve T1 çalışır.

Aynı mekanizma, farklı politikalarla farklı davranış üretmiştir.



Şekil 14.13: Mekanizma ve politika. (a) Kullanıcı alanı henüz müdahale etmemiştir; çekirdek en yüksek öncelikli T3'ü (yedekleme) seçer. (b) Veritabanı öncelikleri değiştirmiştir; aynı çizelgeleyici artık T2'yi (sorgu) seçer.

Linux 6.12'deki sched_ext bu ilkeyi bir adım öteye taşır. Yalnız öncelikler değil, çizelgeleme politikasının tamamı bir eBPF programı olarak yüklenebilir. Çekirdek görev kuyruklarını, bağlam değiştirmeyi ve CPU'lara dağıtımı sağlar; hangi görevin ne zaman çalışacağına ise yüklenen program karar verir.

Aynı ayrım işletim sisteminin her yerinde görülür:

- **Sayfalama.** Mekanizma MMU ve çerçeve tahsisidir; politika hangi sayfanın çıkarılacağına karar veren kuraldır (LRU, saat).
- **CPU frekansı.** Mekanizma cpufreq sürücüsüdür; politika `schedutil` ya da `powersave` yöneticisidir.
- **Güvenlik.** Mekanizma LSM kancalarıdır; politika SELinux kurallarıdır.

> **Sık yapılan hata**
>
> Her çekirdek kodunu mekanizma, her kullanıcı alanı kodunu politika saymak. Ayrım kodun yerine göre değil, işin türüne göre yapılır. Sayfa yer değiştirme kuralı çekirdekte çalışır, ama bir politikadır: LRU yerine saat algoritması seçilebilir. MMU'nun adres çevirisi ya da bağlam değiştirmede yazmaçların kaydedilmesi ise politikadan bağımsız mekanizmalardır.

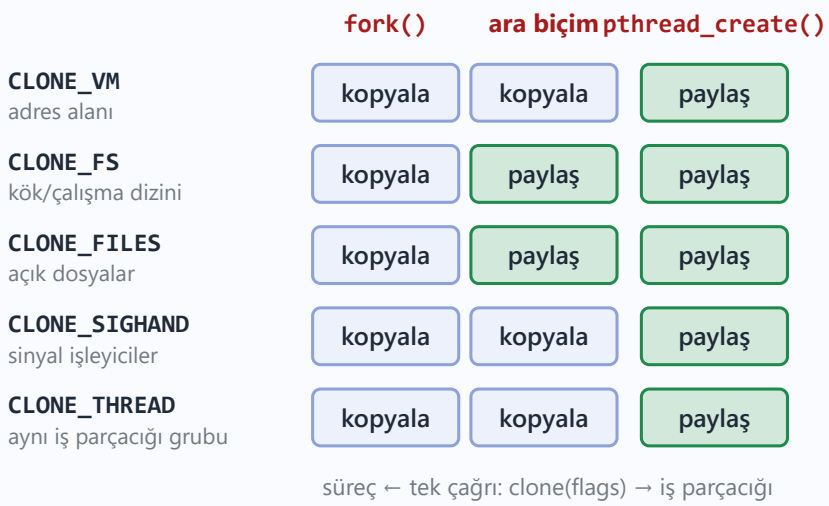
14.4.2 Diklik

Tanım 14.9

Diklik (*orthogonality*), ayrı kavramların birbirinden bağımsız olarak birleştirilebilmesidir. Dik bir tasarımda bir kavramı değiştirmek diğerini etkilemez.

Terim geometriden gelir: dik iki eksende bir yönde ilerlemek diğer eksendeki konumu değiştirmez. Tasarımda diklik, az sayıda kavramdan çok sayıda bileşim elde etmeyi sağlar. C dili basit bir örnektir. Bir değişkenin ömrü (`static`) ile türü birbirinden bağımsız seçilir: `static int`, `static char *`, `static struct proc` hepsi geçerlidir.

İşletim sistemlerinde en güzel örnek Linux'un `clone` çağrısıdır (Şekil 14.14). Yeni bir yürütme birimi oluşturulurken neyin paylaşılacağı ayrı ayrı bayraklarla seçilir. `CLONE_VM` adres alanını, `CLONE_FS` kök ve çalışma dizinini, `CLONE_FILES` açık dosya tablosunu, `CLONE_SIGHAND` sinyal işleyicilerini, `CLONE_THREAD` ise iş parçacığı grubunu paylaştırır.



Şekil 14.14: Linux `clone` çağrısında bağımsız bayraklar. `fork()` hiçbir şeyi paylaşmaz, `pthread_create()` hepsini paylaşır; arada her bileşim mümkündür.

Bayrakların hiçbiri verilmezse sonuç `fork`'tur: her şey kopyalanır. Hepsi verilirse sonuç `pthread_create`'tir: her şey paylaşılır. Süreç ve iş parçacığı böylece iki ayrı kavram değil, aynı mekanizmanın iki ucudur. Konteynerler de aynı çağrıyla, `CLONE_NEWPID` ve `CLONE_NEWNET` gibi `CLONE_NEW*` ad alanı bayraklarıyla kurulur.

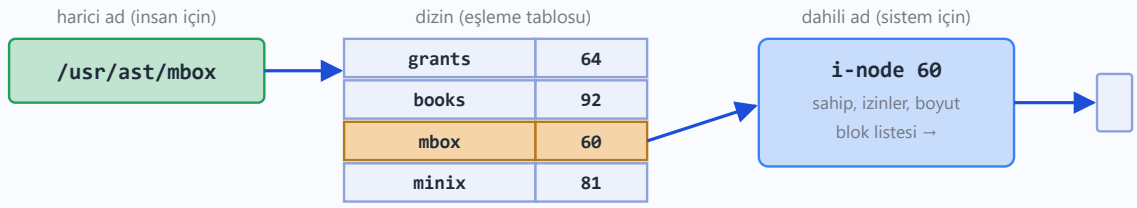
Windows aynı soruna farklı bir dik tasarımla yaklaşır. Süreç bir kaynak kabıdır: adres alanı, tanıtıcılar ve güvenlik bilgisi süreçtedir. İş parçacığı ise çizelgeleme birimidir. İki kavram baştan ayrıdır ve her süreç bir ya da daha çok iş parçacığı içerir.

14.4.3 Adlandırma: iki düzeyli adlar

Bir işletim sistemi binlerce nesneyi yönetir: dosyalar, süreçler, aygıtlar, bellek sayfaları, ağ uçları. İnsanlar bu nesnelere hatırlanabilir adlarla başvurmak ister. Sistemin içinde ise hızlı ve benzersiz bir sayı daha kullanışlıdır. Çözüm iki düzeyli adlandırmadır.

Tanım 14.10

Harici ad (*external name*) insanların kullandığı, genellikle bir ASCII ya da Unicode dizgisi olan addır. **Dahili ad** (*internal name*) sistemin içinde kullanılan, hızlı ve benzersiz bir sayıdır. İkisi arasındaki eşlemeyi bir tablo ya da hizmet yapar.



Şekil 14.15: İki düzeyli adlandırma. Harici ad `/usr/ast/mbox`, dizin (eşleme tablosu) aracılığıyla dahili ad olan i-node 60'a eşlenir; i-node sahibi, izinleri, boyutu ve blok listesini tutar.

Aynı kalıp işletim sisteminin birçok yerinde tekrar eder (Tablo 14.4).

Harici ad	Eşleyen	Dahili ad
`/usr/ast/mbox`	dizin	i-node numarası
`www.giresun.edu.tr`	DNS	IP adresi → (ARP) MAC adresi
dosya tanımlayıcısı 3	süreç dosya tablosu	çekirdekteki `struct file`
sanal adres	sayfa tablosu	fiziksel çerçeve
`\Device\Harddisk0`	nesne yöneticisi	çekirdek nesnesi (HANDLE)

Tablo 14.4: İki düzeyli adlandırma örnekleri.

Aradaki eşlemenin en büyük kazancı, adı değiştirmeden nesneyi taşıyabilmektir. Bir web sitesi başka bir sunucuya taşındığında yalnız DNS kaydı değişir; kullanıcılar aynı adı yazmayı sürdürür. Bir sayfa diske atılıp başka bir çerçeveye geri yüklendiğinde yalnız sayfa tablosu güncellenir; program aynı sanal adresi kullanır. Tablodaki DNS satırı ilginçtir: orada eşleme iki kez yapılır. Ad önce IP adresine, IP adresi de ARP ile MAC adresine çevrilir.

14.4.4 Ad çözümleme adım adım

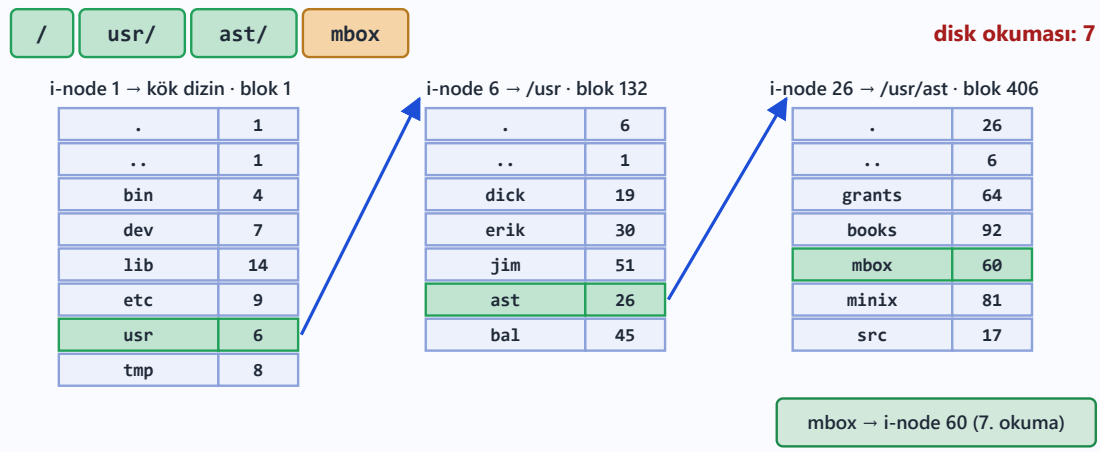
İki düzeyli adlandırmanın bir maliyeti vardır: her ad kullanıldığında eşleme yapılmalıdır. Bir dosya yolunda bu eşleme bileşen bileşen yürür ve her bileşen disk erişimi gerektirebilir.

Örnek 14.5

Çekirdek `open("/usr/ast/mbox")` çağrısını alıyor ve hiçbir şey önbellekte değil. Dosyanın i-node'una ulaşmak için kaç disk okuması gerekir?

1. **Kök dizin.** Kök dizinin i-node'u (i-node 1) okunur; dizin verisinin blok 1'de olduğu öğrenilir. Blok 1 okunur ve doğrusal aramayla `usr` → i-node 6 bulunur. Toplam 2 okuma.
2. **/usr.** i-node 6 okunur, ardından `/usr` dizininin bloğu 132. `ast` → i-node 26. Toplam 4 okuma.
3. **/usr/ast.** i-node 26 okunur, sonra dizin bloğu 406. `mbox` → i-node 60. Toplam 6 okuma.
4. **Dosya.** Son olarak dosyanın kendi i-node'u (60) okunur. Toplam 7 okuma.

Kural şudur: yoldaki her dizin için iki okuma (i-node ve dizin bloğu), sonda dosyanın i-node'u için bir okuma. Üç dizinli bu yol 2 × 3 + 1 = 7 okuma ister.



Şekil 14.16: `/usr/ast/mbox` yolunun bileşen bileşen çözümlenmesi. Kök dizin (i-node 1, blok 1), `/usr` (i-node 6, blok 132) ve `/usr/ast` (i-node 26, blok 406) okunarak i-node 60 bulunur.

Yalnız dosyayı bulmak için yedi disk erişimi gerekmiştir. Bir sabit diskte her erişim milisaniyeler sürdüğü için bu onlarca milisaniyelik bir gecikme demektir; dosyadan henüz tek bayt okunmamıştır. İki düzeyli adlandırmanın esnekliği ancak eşlemeler önbellekte tutulursa ucuz hâle gelir.

Sık yapılan hata

Yalnız dizin bloklarını saymak ya da dosyanın kendi i-node'unu unutmak. Bir dizinin içeriğini okumak için önce o dizinin i-node'u okunmalıdır, çünkü dizin verisinin hangi blokta olduğu oradadır. Dört dizinli `/home/ali/ders/notlar.txt` yolu bu yüzden 2 × 4 + 1 = 9 okuma ister.

14.4.5 Bağlama zamanı

Tanım 14.11

Bağlama zamanı (*binding time*), bir adın gerçek nesneye bağlandığı andır. **Erken bağlamada** (*early binding*) ilişki önceden sabitlenir; **geç bağlamada** (*late binding*) mümkün olan en son anda kurulur.

Bir ad birçok farklı anda bağlanabilir. Bellek adresleri bunun iyi bir örneğidir (Şekil 14.17). Program yazılırken koda gömülen mutlak bir adres en erken bağlamadır. Derleme anında sabit bir yükleme adresi seçilebilir. Bağlama anında statik kütüphaneler programa eklenir. Yükleme anında yer değiştirme ve ASLR adresleri belirler. Çalışma anında ise sanal bellek, `dlopen` ve DNS bağlamayı son ana bırakır.



Şekil 14.17: Bağlama zamanı çizgisi: yazılırken, derleme, bağlama, yükleme ve çalışma zamanı. Soldan sağa esneklik artar; erken bağlama basit ve hızlı, geç bağlama karmaşık ama esnektir.

Erken bağlama basit ve hızlıdır, ama az esnektir. Program yalnız belirli bir adreste çalışır; kütüphane güncellenince program yeniden derlenmelidir. Geç bağlama daha karmaşıktır, çünkü ek bir dolaylılık ve çalışma zamanı maliyeti getirir. Karşılığında esneklik kazanılır: sayfa belleğin neresine yüklenirse yüklensin program çalışır, kütüphane güncellenince program değişmez.

İşletim sistemleri genel olarak geç bağlamayı seçer. Sanal bellek bunun en uç örneğidir: bir sayfanın fiziksel yeri her erişimde yeniden çözülür. Bu maliyet TLB sayesinde katlanılabilir düzeydedir. Geç bağlamanın bedeli ödenir, ama bedeli küçültmek için donanım yardımı alınır.

14.4.6 Statik ve dinamik yapılar

Çekirdek veri yapılarının boyutu da bir tasarım kararıdır. Süreç tablosu, açık dosya tablosu ya da tampon havuzu derleme anında sabit boyutlu bir dizi olarak mı, yoksa çalışırken büyüyüp küçülen bir yapı olarak mı tutulmalı?

Tanım 14.12

Statik yapı, boyutu derleme ya da açılış anında sabitlenen yapıdır; örneğin sabit boyutlu bir dizi. **Dinamik yapı** ise çalışırken bellek ayırarak büyüyüp küçülür; örneğin bağlı liste ya da hash tablosu.

Eski UNIX çekirdeklerinde süreç tablosu sabit boyutlu bir diziydi ve bir süreç şöyle aranırdı:

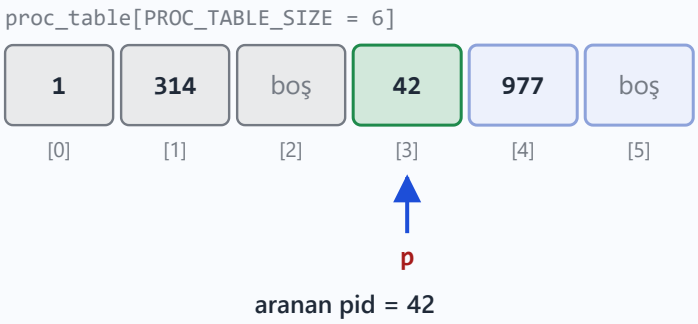
```
found = 0;
for (p = &proc_table[0];
     p < &proc_table[PROC_TABLE_SIZE];
     p++) {
    if (p->proc_pid == pid) {
        found = 1;
        break;
    }
}
```

`found` sıfırlanır ve `p` işaretçisi tablonun ilk elemanına konur. Döngü `p`, tablonun sonundaki `&proc_table[PROC_TABLE_SIZE]` adresine ulaşana kadar sürer. Her turda girdinin `proc_pid` alanı aranan `pid` ile karşılaştırılır. Eşleşme olunca `found = 1` yapılır ve `break` döngüden çıkar. `p++` işaretçiyi bir bayt değil, bir `struct proc` boyu ilerletir; C'de işaretçi aritmetiği gösterilen türün boyutuyla yapılır.

Altı girdilik bir tabloda pid 42 aranıyor. Tablodaki pid'ler sırasıyla 1, 314, 0, 42, 977, 0'dır; 0 boş girdi demektir (Tablo 14.5).

Tur	p	p->proc_pid	Karşılaştırma	found
1	`&proc_table[0]`	1	1 ≠ 42	0
2	`&proc_table[1]`	314	314 ≠ 42	0
3	`&proc_table[2]`	0 (boş)	0 ≠ 42	0
4	`&proc_table[3]`	42	eşit, `break`	1

Tablo 14.5: Statik süreç tablosunda pid 42'nin doğrusal aranması.



Şekil 14.18: Altı girdilik statik süreç tablosu (`PROC_TABLE_SIZE = 6`). İşaretçi `p` , dördüncü karşılaştırmada pid 42'yi bulmuştur.

Boş girdi de tek tek taranır; doğrusal aramanın bedeli budur. En kötü durumda tablonun tamamı taranır ve arama $O(n)$ sürer. Ama küçük ve sabit bir tablo için bu basit kod yeterlidir.

Statik tablonun artıları basitlik ve öngörülebilirliktir: bellek ayırıcı gerekmez, tablo hiçbir zaman parçalanmaz. Eksisi, tablo dolunca yeni süreç oluşturulamamasıdır. Eski UNIX'te bu sınır `NPROC` sabitiydi. Dinamik bir yapı, örneğin bağlı liste, hash tablosu ya da Linux'taki `xarray` , bu sınırı kaldırır; hash tablosu ve `xarray` aramayı da $O(1)$'e yaklaştırır. Bedeli bir bellek ayırıcı, parçalanma ve daha karmaşık kilitlemedir. Ara yol da vardır: boyutu ayarlanabilir tablolar. Linux'ta en büyük pid değeri `/proc/sys/kernel/pid_max` dosyasından okunur ve değiştirilebilir.

14.4.7 Yukarıdan aşağı ve aşağıdan yukarı gerçekleme

Bir sistem tasarlandıktan sonra yazılmaya hangi uçtan başlanmalıdır? İki yaklaşım vardır (Şekil 14.19).



Şekil 14.19: Yukarıdan aşağı gerçeklemede alt katmanlar koçan (stub) olarak bırakılır; aşağıdan yukarı gerçeklemede ise donanım soyutlamasından başlanır ve her katman bittiğinde hemen test edilir.

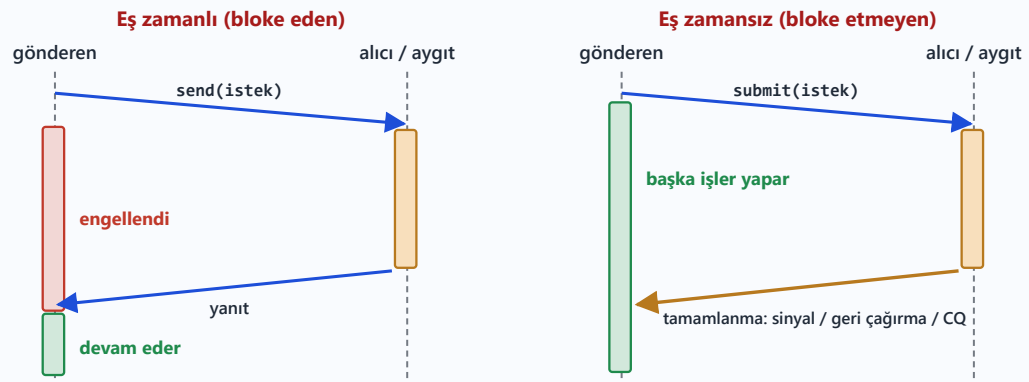
Yukarıdan aşağı (*top-down*) gerçeklemede önce sistem çağrısı işleyicileri yazılır. Alt katmanlar **koçan** (*stub*) olarak bırakılır. Koçan, doğru imzayla tanımlanmış ama içi boş olan ya da sabit bir değer döndüren fonksiyondur. Arayüz erken netleşir. Ama bütün sistem bitmeden gerçekçi bir test yapmak zordur, çünkü koçanlar gerçek davranışı taklit etmez.

Aşağıdan yukarı (*bottom-up*) gerçeklemede önce donanımı gizleyen katman, sonra kesmeler, zamanlayıcı, iş parçacıkları yazılır. Her katman bittiğinde test edilir, çünkü altındaki her şey gerçektir. Tanenbaum gerçeklemede bu yolu önerir. Tasarımın ise yukarıdan aşağı düşünülmesi gerekir: önce arayüz belirlenir, sonra o arayüzü sağlayacak katmanlar alttan üste kurulur.

14.4.8 Eş zamanlı ve eş zamansız iletişim

Tanım 14.13

Eş zamanlı (*synchronous*) iletişimde gönderen, isteği gönderdikten sonra yanıt gelene kadar engellenir. **Eş zamansız** (*asynchronous*) iletişimde gönderen çalışmayı sürdürür; tamamlanma ona sonradan bir sinyal, geri çağırma ya da tamamlanma kuyruğuyla bildirilir.



Şekil 14.20: Solda eş zamanlı iletişim: gönderen `send(istek)` sonrasında yanıt gelene kadar engellenir. Sağda eş zamansız iletişim: gönderen `submit(istek)` sonrasında başka işler yapar; tamamlanma sinyal, geri çağırma ya da CQ ile bildirilir.

Eş zamanlı iletişimin programlaması kolaydır. Akış sıralı okunur: istek, yanıt, sonraki satır. Bedeli, her bekleyen istek için bir iş parçacığının boşta durmasıdır. Mikro çekirdeklerde IPC genellikle eş zamanlıdır; L4 ve seL4 bu yolu izler. Gönderen yanıtı beklediği için çekirdeğin iletiyi tamponlamasına gerek kalmaz ve ileti doğrudan alıcıya aktarılabilir. Bu da IPC'yi çok hızlı yapar.

Eş zamansız iletişimde tek bir iş parçacığı binlerce isteği aynı anda yürütebilir. Ama yarış koşulları, hata yönetimi ve tamponlama zorlaşır. Bir isteğin hatası, onu gönderen kod çoktan başka bir işe geçmişken gelir. Linux'taki `io_uring` bu modelin güncel örneğidir. Gönderim (SQ, *submission queue*) ve

tamamlanma (CQ, *completion queue*) halkaları kullanıcı alanı ile çekirdek arasında paylaşılır. Uygun ayarla, çekirdekteki bir iş parçacığı gönderim halkasını izlediği için sistem çağrısı yapılmadan bile istek gönderilebilir.

Sık yapılan hata

Eş zamansız iletişimin her zaman daha hızlı olduğunu sanmak. Tek bir isteğin gecikmesi iki modelde aşağı yukarı aynıdır; eş zamansız model yalnız beklerken başka iş yapılmasını sağlar. Kazanç çok sayıda eş zamanlı istek olduğunda ortaya çıkar. Bedeli ise daha karmaşık ve hata ayıklaması zor bir koddur.

14.5 Yararlı teknikler

İlkeler neyin hedefleneceğini söyler; teknikler ise hedefe nasıl varılacağını. Bu kısımdaki teknikler işletim sistemi kodunun her yerinde, çoğu zaman adı konmadan kullanılır. Her birinin kazancı ve bedeli vardır.

14.5.1 Donanımı gizleme

Bir işletim sistemi birden çok işlemci mimarisinde çalışmak zorundadır. Her mimarinin başlatma kodu, yazmaç boyu ve kesme düzeni farklıdır. Bu farkların bütün koda dağılması taşınabilirliği öldürür. Çözüm, donanıma bağımlı kodu az sayıda dosyada toplamaktır. C'de bunun en eski aracı koşullu derlemedir.

Tanım 14.14

Koşullu derleme (*conditional compilation*), önişlemcinin `#if`, `#elif` ve `#endif` yönergeleriyle kaynak kodun yalnız belirli koşulları sağlayan kısımlarının derlenmesidir. Koşulu sağlamayan satırlar derleyiciye hiç ulaşmaz.

İlk örnek CPU türüne göre farklı başlatma kodu seçer. `config.h` dosyası `CPU` sabitini tanımlar ve önişlemci yalnız eşleşen bloğu derlemeye bırakır.

```
#include "config.h"
void init(void)
{
#if (CPU == X86_64)
    /* x86-64 başlatma kodu */
#endif
#if (CPU == ARM64)
    /* ARM64 başlatma kodu */
#endif
#if (CPU == RISCV64)
    /* RISC-V başlatma kodu */
#endif
}
```

İkinci örnek sözcük uzunluğuna göre bir yazmaç türü seçer. 32 bitlik makinede `Register` bir `int`, 64 bitlik makinede bir `long` olur. Son dal bugünkü taşınabilir çözümdür: `<stdint.h>` içindeki `uintptr_t` her platformda bir işaretçiyi tutacak kadar büyük bir tam sayı türüdür.

```
#include "config.h"
#include <stdint.h>
#if (WORD_LENGTH == 32)
typedef int Register;
#elif (WORD_LENGTH == 64)
typedef long Register;
#else  /* bugün: taşınabilir */
typedef uintptr_t Register;
#endif
Register R0, R1, R2, R3;
```

Kodlar Tanenbaum'daki Pentium–UltraSPARC örneğinin güncellenmiş hâlidir. Bugünkü sistemlerde aynı fikir dizin düzeyinde uygulanır. Windows'ta donanıma bağımlı kod HAL'de (*Hardware Abstraction Layer*) toplanır. Linux'ta ise `arch/x86`, `arch/arm64` ve `arch/riscv` gibi dizinlerdedir. Yeni bir işlemciye taşırken yalnız bu kısım yeniden yazılır.

Örnek 14.6

`config.h` içinde `CPU` sabiti `ARM64`, `WORD_LENGTH` 64 olarak tanımlı. Kod hangi başlatma bloğunu derler, `sizeof(Register)` kaçtır? Aynı kod 64 bit Windows'ta derlenirse ne değişir?

1. **Başlatma.** Yalnız `#if (CPU == ARM64)` bloğu derlenir. x86-64 ve RISC-V blokları önişlemcide silinir; derleyici onları hiç görmez.
2. **Register.** `WORD_LENGTH == 64` olduğu için `Register` bir `long` 'dur. 64 bit Linux LP64 modelini kullanır; bu modelde `long` 8 bayttır, yani `sizeof(Register)` 8'dir.

3. **Windows.** 64 bit Windows LLP64 modelini kullanır; orada `long` 4 bayttır. Aynı kaynak kod farklı boyda bir yazmaç türü üretir. Taşınabilir kodda bu yüzden `uintptr_t` tercih edilir.

Sık yapılan hata

Koşullu derlemeyi her yere dağıtmak. Her fonksiyonun içine serpiştirilmiş `#ifdef` blokları kodu okunmaz hâle getirir ve bazı bileşimler hiç derlenmediği için hatalar yıllarca fark edilmez. Doğrusu, donanıma bağımlı kodu birkaç dosyada toplamak ve geri kalan koda ortak bir arayüz sunmaktır.

14.5.2 Dolaylılık

Tanım 14.15

Dolaylılık (*indirection*), bir nesneye doğrudan değil, araya giren bir tablo, işaretçi ya da ad üzerinden erişmektir. Aradaki adım değiştirilerek, iki uçtaki taraflara dokunmadan davranış değiştirilebilir.

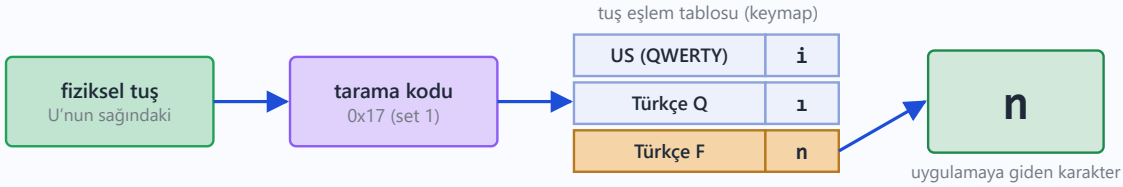
David Wheeler'a atfedilen söz tekniğin gücünü özetler: bilgisayar bilimindeki her sorun, bir dolaylılık düzeyi daha ile çözülebilir. Klavye girdisi iyi bir örnektir. Klavye bir tuşa basıldığında karakter göndermez; tuşun konumunu bildiren bir **tarama kodu** (*scan code*) gönderir. Bu kodu karaktere çeviren, işletim sistemindeki **tuş eşlem tablosudur** (*keymap*).

Örnek 14.7

Kullanıcı, ABD klavyesinde U'nun sağındaki tuşa basıyor. Üç farklı tuş eşlem tablosuyla hangi karakter üretilir?

1. **Donanım.** Klavye, set 1'de 0x17 tarama kodunu gönderir. USB klavyelerde gelen bir HID kullanım kodudur; sürücü onu ortak bir koda çevirir.
2. **ABD (QWERTY) eşlemi.** 0x17 → `i`.
3. **Türkçe Q eşlemi.** Yalnız tablo değiştirilir: aynı tuş artık `ı` üretir.
4. **Türkçe F eşlemi.** Aynı tuş `n` olur.

Üç durumda da donanım ve uygulama aynıdır; değişen yalnız aradaki tablodur.



Şekil 14.21: Klavye girdisinde dolaylılık. Fiziksel tuş 0x17 tarama kodunu üretir; tuş eşlem tablosu bunu ABD düzeninde `i`'ye, Türkçe Q'da `ı`'ya, Türkçe F'de `n`'ye çevirir.

Ek adımın kazancı esnekliktir, bedeli ise bir tablo okumasıdır. Bu bedel çoğu zaman önemsizdir, ama sıcak bir yolda birikebilir. İşletim sistemlerinde dolaylılık her yerdedir: sanal bellekte sayfa tablosu, dosya tanımlayıcılarında süreç dosya tablosu, sembolik bağlarda başka bir yol, dosya sistemlerinde i-node, sürücülerde aygıt sınıfları. İki düzeyli adlandırma ve geç bağlama da birer dolaylılık örneğidir.

14.5.3 Yeniden kullanım ve yeniden girişlilik

Büyük bir sistemde aynı türden sorunlar farklı yerlerde tekrar eder. Boş disk bloklarını, boş bellek çerçevelerini ve kullanılmayan PID'leri izlemek aynı sorundur: büyük bir kümede hangi elemanların boş olduğunu tutmak. Bu üçü için tek bir **biteşlem** (*bitmap*) modülü yazmak, üç ayrı kod yazmaktan iyidir. Linux bu ilkeyle ortak veri yapıları sunar: `list.h`'deki çift bağlı liste, kırmızı-siyah ağaç ve `xarray` çekirdeğin her yerinde kullanılır.

Yeniden kullanılacak kodun bir koşulu sağlaması gerekir: aynı anda birden çok yerden çağrılabilmelidir.

Tanım 14.16

Bir kod parçası, bir çağrısı sürerken aynı kod yeniden çağrıldığında da doğru çalışıyorsa **yeniden girişlidir** (*reentrant*). Kural: çağrılar arasında paylaşılan statik değişken yoktur; durum çağıranın verdiği yerde tutulur, paylaşılması gerekiyorsa kilitle korunur.

Çekirdekte yeniden girişlilik iki yoldan gerekir. Birincisi kesmelerdir: bir fonksiyon çalışırken kesme gelir ve kesme işleyicisi aynı fonksiyonu çağırabilir. İkincisi çok çekirdektir: başka bir CPU çekirdeği aynı kodu aynı anda çalıştırabilir. Standart C kütüphanesindeki `strtok` bu koşulu sağlamayan klasik bir örnektir.

```
char *strtok(char *s, const char *d)
{
    static char *last; /* paylaşılan durum! */
    ...
}

/* yeniden girişli sürüm: durum çağıranda */
char *strtok_r(char *s, const char *d,
               char **saveptr);
```

`strtok` bir dizgiyi ayraçlara göre parçalara böler. İlk çağrıda dizgi verilir; sonraki çağrılarda `NULL` verilir ve fonksiyon kaldığı yerden sürer. Kaldığı yeri içindeki `static char *last` değişkeninde saklar. Bu değişken yığında değil, bütün iş parçacıklarının paylaştığı veri kesimindedir. Yeniden girişli sürüm `strtok_r` ise aynı bilgiyi çağıranın verdiği `saveptr` işaretçisinde tutar.

14.5.4 Kod izleme: strtok

Sorunun nasıl ortaya çıktığını görmek için `strtok`'un sadeleştirilmiş bir sürümü izlenebilir. `my_strtok` tek karakterlik bir ayraç alır ve durumunu `static char *kalan` değişkeninde tutar.

```
char *my_strtok(char *s, char ayr) {
    static char *kalan;     /* herkes paylaşır! */
    if (s) kalan = s;
    if (!kalan || !*kalan) return NULL;
    char *bas = kalan;
    while (*kalan && *kalan != ayr) kalan++;
    if (*kalan) *kalan++ = '\0';
    return bas;
}
/* main:   my_strtok(a, ',')  a = "ali,veli" */
/* kesme:  my_strtok(b, ';')  b = "x;y"      */
/* main:   my_strtok(NULL, ',')              */
```

Fonksiyon satır satır şöyle çalışır. `s` verilmişse `kalan` onun başına konur. `kalan` `NULL` ise ya da dizginin sonunu gösteriyorsa parça kalmamıştır ve `NULL` döner. Aksi hâlde parçanın başı `bas`'ta saklanır. `while` döngüsü `kalan`'ı ayraca ya da dizgi sonuna kadar ilerletir. Ayraç bulunduysa yerine `'\0'` yazılır ve `kalan` bir sonraki karaktere geçer. Fonksiyon `bas`'ı döndürür; dönen dizgi yazılan `'\0'`'da biter.

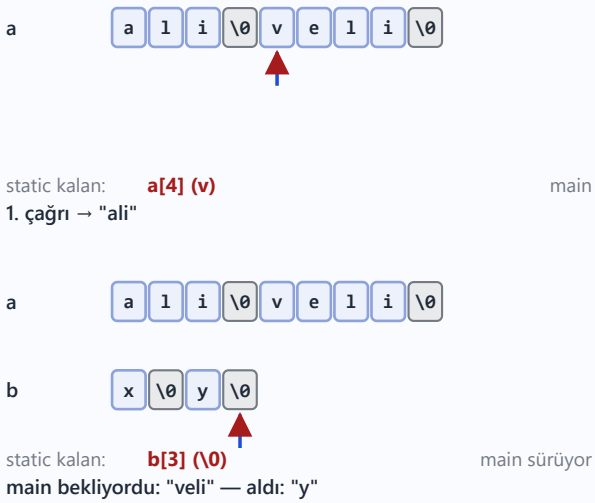
Senaryo şöyledir. `main`, `a = "ali,veli"` dizgisini virgülle ayıklıyor. İlk parçayı aldıktan hemen sonra bir kesme geliyor ve kesme işleyicisi aynı fonksiyonla `b = "x;y"` dizgisini noktalı virgülle ayıklıyor. Ardından `main` kaldığı yerden sürüyor (Tablo 14.6).

Çağıran	Çağrı	Dizgilerin durumu	kalan	Dönüş
main	`my_strtok(a, ',')`	a: `ali\0veli\0`	`&a[4]` (v)	`"ali"`

Çağıran	Çağrı	Dizgilerin durumu	kalan	Dönüş
kesme	`my_strtok(b, ';')`	b: `x\0y\0`	`&b[2]` (y)	`"x"`
main	`my_strtok(NULL, ',')`	değişmez	`&b[3]` (\0)	`"y"`

Tablo 14.6: `my_strtok` izlemesi. `kalan` sütunu çağrıdan sonraki değeri gösterir.

- **İlk çağrı.** `kalan = a` olur. Döngü `a[3]`'teki virgüle kadar ilerler; virgülün yerine `'\0'` yazılır ve `kalan` `a[4]`'e, yani "veli"nin başına geçer. Dönüş `"ali"`'dir.
- **Kesme.** İşleyicinin çağrısı `s = b` ile gelir ve paylaşılan `kalan`'ın üzerine yazar. Noktalı virgül `'\0'` olur, `kalan` `b[2]`'yi gösterir, dönüş `"x"`'tir. `main`'in kaldığı yer kaybolmuştur.
- **main sürer.** `main` `NULL` geçer ve fonksiyonun hâlâ `a` üzerinde olduğunu varsayar. Ama `kalan` `b[2]`'yi gösterir. Döngü dizgi sonuna kadar ilerler; ayraç yoktur, `kalan` `b[3]`'te kalır. Dönüş `"y"`'dir; beklenen ise `"veli"`'ydi.



Şekil 14.22: (a) İlk çağrıdan sonra `kalan`, `a[4]`'ü (v) gösterir. (b) Kesme işleyicisinin çağrısından sonra `main` sürdüğünde `kalan` `b`'nin içindedir; `main` "veli" beklerken "y" alır.

Hata nadiren ve zamanlamaya bağlı olarak görülür. Kesme tam iki çağrı arasına denk gelmezse program doğru çalışır. Bu tür hataları testle yakalamak bu yüzden zordur. Kök neden açıktır: çağrılar arasındaki durum paylaşılan bir `static` değişkende tutulur. Çözüm de açıktır: `strtok_r(s, d, &kalan)` gibi durumu çağırana taşımak. O zaman her çağıranın kendi `kalan`'ı olur ve kesme `main`'in durumuna dokunamaz.

Sık yapılan hata

`static` değişkenin her iş parçacığının kendi yığınında tutulduğunu sanmak. Yerel `static` değişkenler veri kesimindedir ve programdaki bütün iş parçacıkları onları paylaşır. İki iş parçacığı aynı anda `strtok` kullanırsa parçalar yanlış dizgiden gelebilir. Tek çekirdekli makine

de bu hatadan korunmuş değildir: kesme ya da ön alım aynı iç içe geçmeyi yaratır.

14.5.5 Kaba kuvvet ve önce hata denetimi

Tanım 14.17

Kaba kuvvet (*brute force*), en basit ve en açık çözümü, görünüşte yavaş da olsa tercih etmektir. Karmaşık bir algoritmaya ancak ölçüm gerekli gösterdiğinde geçilir.

Elli girişlik bir tabloda doğrusal arama, bir hash tablosundan hem daha hızlı hem de daha hatasız olabilir. Hash fonksiyonunun hesabı, çakışma zincirleri ve yeniden boyutlandırma küçük bir tabloda kazandırdığından fazlasını harcar. Karmaşık algoritmaya ancak n büyükse ya da kod sık çalışan bir yoldaysa gerek vardır. Sıralama da benzerdir: küçük dizilerde ekleme sıralaması, büyük dizilerde birleştirme ya da heap sort uygundur.

İkinci teknik hata denetiminin yeriyle ilgilidir. Bir sistem çağrısı işleyicisi kullanıcıdan bir boyut ve bir tampon adresi alıyor; bir kilit ve bellek de gerekiyor. Kötü sıralama şöyledir:

```
lock(&tbl_lock);
buf = kmalloc(n, GFP_KERNEL);
if (n > MAX || !access_ok(u, n)) {
    kfree(buf);
    unlock(&tbl_lock);
    return -EINVAL;
}
```

Kod önce kilidi alır, sonra belleği ayırır, ancak ondan sonra parametrelerin geçerli olup olmadığına bakar. Parametre hatalıysa aldığı her şeyi ters sırayla geri vermek zorundadır: `kfree`, `unlock`, sonra hata kodu. Bu kod bir de `kmalloc`'un başarısız olma durumunu hiç denetlemiyor. İyi sıralama ise şöyledir:

```
if (n > MAX || !access_ok(u, n))
    return -EINVAL; /* önce denetle */
buf = kmalloc(n, GFP_KERNEL);
if (!buf)
    return -ENOMEM;
lock(&tbl_lock);
...
```

Önce parametreler denetlenir; hatalıysa hiçbir kaynak alınmadan `-EINVAL` döner. Ardından bellek ayrılır ve ayırmanın başarısı denetlenir. Kilit en son alınır. Hata yolunda geri alınacak bir şey kalmaz. Gerçek çekirdeklerde en sık görülen hatalardan biri, bir hata yolunda `unlock`'un unutulmasıdır. Önce

hata denetimi bu hata sınıfını ortadan kaldırır.

Sık yapılan hata

Kaba kuvveti "özensiz kod" sanmak. İlke, basit çözümü bilerek ve ölçerek seçmektir. Kaba kuvvetle yazılmış bir tablo taraması, *n* küçük kaldığı sürece doğru bir tasarım kararıdır. Ancak *n* büyüyünce ya da profil bu kodu darboğaz gösterince yeniden düşünülmelidir.

14.6 Performans

Her işletim sistemi hızlı olmak ister. Ama hız ancak doğruluk sağlandıktan sonra anlamlıdır ve her hızlandırma kodu daha karmaşık yapar. Bu kısım önce neyin hızlandırılacağına nasıl karar verileceğini, sonra en sık kullanılan teknikleri ele alır.

14.6.1 Neden yavaş, neyi hızlandırmalı?

Donanım on yıllar içinde binlerce kat hızlandı, ama kullanıcının hissettiği hız aynı oranda artmadı. Artışın önemli bir kısmını yazılımın büyümesi tüketti. Bunun üç nedeni vardır. Birincisi özellik şişkinliğidir: her sürüm yeni özellik ekler, eskisini çıkaramaz. İkincisi geriye dönük uyumluluk katmanlarıdır. Üçüncüsü güvenlik önlemleridir. Spectre ve Meltdown açıklarına karşı eklenen yamalar, örneğin çekirdek ve kullanıcı sayfa tablolarının ayrılması, her sistem çağrısının maliyetini artırdı. Genellik uğruna eklenen dolaylılık da her katmanda biraz zaman alır.

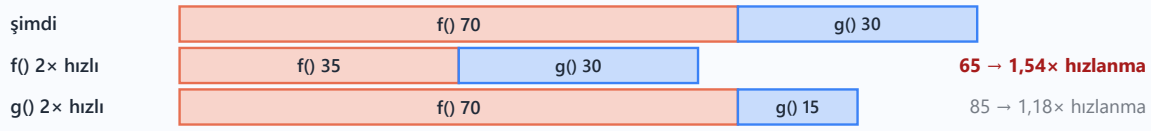
Neyin hızlandırılacağına ölçüm karar verir. Donald Knuth'un sözü bu konuda sıkça anılır: erken optimizasyon tüm kötülüklerin anasıdır. Önce doğru, sonra hızlı kod yazılır. Hızlandırma yalnız ölçümle darboğaz olduğu gösterilen kısımlara uygulanır. "İyi yeterlidir" ilkesi de buraya bağlanır: %5'lik bir kazanç için kodu iki kat karmaşıklaştırmak çoğu zaman değmez. Linux'ta ölçme araçları `perf`, eBPF tabanlı `bpftrace` ve alev grafikleridir.

Bir profilin nasıl okunacağını basit bir hesap gösterir (Şekil 14.23).

Örnek 14.8

Profil, çalışma süresinin %70'inin `f()` fonksiyonunda, %30'unun `g()`'de geçtiğini gösteriyor. Hangisini iki kat hızlandırmak daha çok kazandırır?

1. **Başlangıç.** Toplam süre 100 birim alınsın: `f` 70, `g` 30.
2. **f iki kat hızlı.** `f` 35 birime iner; toplam 35 + 30 = 65. Hızlanma 100 / 65 ≈ 1,54 kat.
3. **g iki kat hızlı.** `g` 15 birime iner; toplam 70 + 15 = 85. Hızlanma 100 / 85 ≈ 1,18 kat.
4. **Sonuç.** Aynı emekle `f`'yi hızlandırmak çok daha fazla kazandırır. `g` sonsuz hızlı olsa bile toplam süre 70'in altına inmez.



Şekil 14.23: Profil: zamanın %70'i `f()` 'de. `f()` 'yi iki kat hızlandırmak toplamı 65'e indirir (1,54 kat); `g()` 'yi iki kat hızlandırmak ise 85'e (1,18 kat).

Bu hesabın genel biçimine Amdahl yasası denir.

Teorem 14.1 (Amdahl yasası)

Çalışma süresinin p kesri k kat hızlandırılırsa toplam hızlanma

$$S = 1 / ((1 - p) + p / k)$$

olur. k ne kadar büyük olursa olsun hızlanma $1 / (1 - p)$'yi aşamaz.

Örnekteki değerler formüle uyar: $p = 0{,}7$ ve $k = 2$ için $S = 1 / (0{,}3 + 0{,}35) \approx 1{,}54$; $p = 0{,}3$ için $S = 1 / (0{,}7 + 0{,}15) \approx 1{,}18$. `f` sonsuz hızlı olsa bile hızlanma $1 / 0{,}3 \approx 3{,}33$ katı geçemez. Yasa, hızlandırma emeğinin yaygın durum üzerine harcanması gerektiğini sayısal olarak gösterir.

Ölçüm sonrasında başvurulan klasik teknikler beş tanedir: uzay-zaman ödünleşimi, önbellekleme, ipuçları, yerelliği kullanmak ve yaygın durumu hızlandırmak.

14.6.2 Uzay-zaman ödünleşimi: bit sayma

Tanım 14.18

Uzay-zaman ödünleşimi (*space-time trade-off*), daha fazla bellek kullanarak çalışma süresini kısaltmak ya da tersine, daha uzun süre çalışarak bellekten tasarruf etmektir.

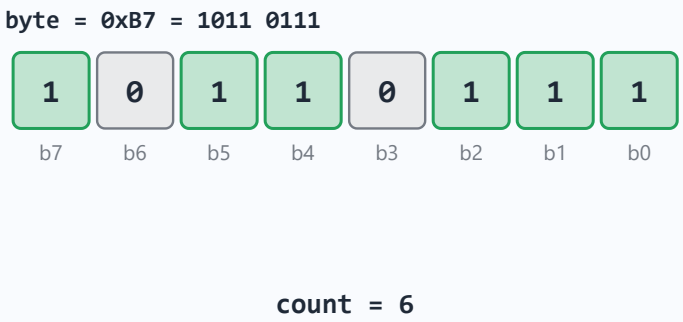
Klasik örnek bir bayttaki 1 bitlerinin sayılmasıdır. Bu işlem bellek ayırıcılarının biteşlemlerinde ve izin maskelerinde sık yapılır. İlk sürüm her biti bir döngüyle tek tek sınar:

```
#define BYTE_SIZE 8
int bit_count(int byte)
{
    int i, count = 0;
    for (i = 0; i < BYTE_SIZE; i++)
        if ((byte >> i) & 1)
            count++;
    return count;
}
```

Döngü `i`'yi 0'dan 7'ye kadar sayar. Her turda bayt `i` bit sağa kaydırılır ve en düşük bit 1 ile maskelenir: `(byte >> i) & 1`. Sonuç 1 ise sayaç artar. Örnek bayt 0xB7'dir; ondalık 183, ikili 1011 0111 (Tablo 14.7).

i	0	1	2	3	4	5	6	7
(byte >> i) & 1	1	1	1	0	1	1	0	1
count	1	2	3	3	4	5	5	6

Tablo 14.7: `bit_count(0xB7)` izlemesi. 0xB7 = 1011 0111.



Şekil 14.24: 0xB7'nin sekiz biti döngüyle tek tek sayılmıştır; sonuç 6'dır.

Bit 3 ve bit 6 sıfırdır; bu turlarda dal alınmaz ve sayaç değişmez. Döngü sekiz kez döner ve 6 döndürür. Bellek kullanımı yok denecek kadar azdır. Ama bayt başına sekiz tur, her turda bir kaydırma, bir maskeleme, bir karşılaştırma ve bir dal vardır. Dal tahmini başarısız olduğunda işlemcinin boru hattı boşaltılır ve bu ek maliyet getirir.

Daha hızlı iki sürüm yer harcayarak zaman kazanır:

```
/* (b) makro: döngü yok */
#define bit_count(b) ((b&1) + ((b>>1)&1) + \
    ((b>>2)&1) + ... + ((b>>7)&1))

/* (c) tablo: 256 bayt */
char bits[256] =
    {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...};
#define bit_count(b) (int) bits[b]
```

(b) makrosu döngüyü açar: sekiz terim doğrudan toplanır. Döngü ve dal yoktur, ama yaklaşık 24 komut gerekir. (c) ise 256 girişlik bir tablo kullanır. Tablonun *i*. girdisi, *i*'nin ikili gösterimindeki 1 sayısıdır. Bit sayma tek bir bellek okumasına iner (Şekil 14.25). Yöntemlerin karşılaştırması Tablo 14.8'de verilmiştir.



Şekil 14.25: 256 girişlik bit sayma tablosunun ilk 32 değeri ve `bits[0xB7] = bits[183] = 6`.

Yöntem	Bellek	Bayt başına iş
(a) döngü	~0	8 tur: kaydır, maskele, dallan
(b) makro	~0	~24 komut, dal yok
(c) 256'lık tablo	256 B	1 bellek okuması
16'lık tablo (yarım bayt)	16 B	2 okuma + toplama
POPCNT komutu	0	1 komut (`__builtin_popcount`)

Tablo 14.8: Bit sayma yöntemlerinin yer ve zaman maliyeti.

Tablonun son satırı bugünün gerçeğidir. x86-64 ve ARM64 işlemcilerinde 1 bitlerini sayan özel bir komut vardır ve derleyici `__builtin_popcount` çağrısını bu komuta çevirir. Yine de uzay-zaman ödünleşimi önemini korur, çünkü aynı fikir donanımın yardım etmediği yüzlerce başka hesapta kullanılır.

14.6.3 Kod izleme: yarım bayt tablosu

256 baytlık tablo ile döngü arasında bir orta nokta vardır. Bir bayt iki yarım bayta (*nibble*) bölünürse 16 girişlik bir tablo yeter. Tablo da elle yazılmak yerine bir döngüyle kurulabilir:

```
char yarim[16];          /* 16 bayt */
void tablo_kur(void) {
    yarim[0] = 0;
    for (int i = 1; i < 16; i++)
        yarim[i] = (i & 1) + yarim[i >> 1];
}
int bit_say(unsigned char b) {
    return yarim[b & 0x0F] + yarim[b >> 4];
}
```

`tablo_kur` fonksiyonunun dayandığı gözlem şudur. *i*'nin 1 bitleri sayısı, en düşük bitinin değeri ile *i* >> 1'in 1 bitleri sayısının toplamıdır. *i* >> 1 her zaman *i*'den küçüktür; bu yüzden döngü küçükten büyüğe ilerlerken o değer çoktan hesaplanmıştır. Bu bir dinamik programlama yinelemesidir.

Örnek 14.9

`tablo_kur()` tabloyu hangi değerlerle doldurur?

1. **Başlangıç.** `yarim[0] = 0`; 0'da hiç bit yoktur.
2. **i = 1 … 3.** `yarim[1] = 1 + yarim[0] = 1`, `yarim[2] = 0 + yarim[1] = 1`, `yarim[3] = 1 + yarim[1] = 2`.
3. **i = 4 … 7.** 1, 2, 2, 3. Örneğin `yarim[7] = 1 + yarim[3] = 3`.
4. **i = 8 … 15.** 1, 2, 2, 3, 2, 3, 3, 4. Örneğin `yarim[15] = 1 + yarim[7] = 4`.

Tamamı: 0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4.

Tablo hazır olunca `bit_say` baytı iki yarıya ayırır. `b & 0x0F` alt dört biti, `b >> 4` üst dört biti verir. İki tablo değeri toplanır. Döngü ve dal yoktur (Tablo 14.9).

b	İkili	Alt yarı	Üst yarı	Toplam
0xB7	1011 0111	`yarim[7]` = 3	`yarim[B]` = 3	6
0x5C	0101 1100	`yarim[C]` = 2	`yarim[5]` = 2	4

Tablo 14.9: `bit_say` çağrıları.



Şekil 14.26: 16 girişlik `yarim` tablosu doldurulmuştur. `bit_say(0xB7)`: alt yarı 7 ve üst yarı B için iki okuma, 3 + 3 = 6.

0xB7 için iki bellek okuması ve bir toplama yeter; döngülü sürüm aynı bayt için sekiz tur dönmüştü. Yarım bayt tablosu 16 bayt yer kaplar, 256'lık tablonun on altıda biri kadar. Karşılığında bir okuma ve bir toplama fazladır. Aynı soruya, ne kadar yer harcanacağına göre birkaç farklı cevap verilebildiği görülür.

Sık yapılan hata

Tablo yönteminin her zaman en hızlısı olduğunu sanmak. Tablo bellekte durur ve önbellekte değilse tek bir okuma bir döngünün tamamından uzun sürebilir. 256 baytlık bir tablo dört önbellek satırı kaplar. Sıcak bir yolda sürekli kullanıldığı sürece önbellekte kalır ve hızlıdır. Seyrek kullanılırsa her seferinde ıskalayabilir. Bu yüzden karar burada da ölçümle verilir.

14.6.4 Uzay-zaman ödünleşimi: renk paleti

Ödünleşim ters yönde de işler: biraz zaman harcayarak büyük ölçüde yer kazanılabilir. Renk paleti bunun klasik örneğidir (Şekil 14.27).



Şekil 14.27: (a) Piksel başına 24 bit (3 bayt RGB). (b) Piksel başına 8 bitlik palet indisi (1 bayt). (c) 256 girişlik

renk paleti; her giriş 3 bayttır.

Bir görüntüde her piksel için doğrudan kırmızı, yeşil ve mavi değerleri saklanabilir. Bu piksel başına 3 bayttır. Görüntüde en fazla 256 farklı renk kullanılıyorsa başka bir yol vardır. Kullanılan renkler bir palet tablosunda toplanır ve her piksel için yalnız 1 baytlık bir indis saklanır. Görüntü gösterilirken her indis paletten gerçek renge çevrilir.

Örnek 14.10

1024 × 768 bir görüntü iki biçimde saklanıyor. Boyutlar ve oran nedir?

1. **24 bit/piksel.** 1024 × 768 × 3 = 2.359.296 bayt; 2.359.296 / 2^{20} = 2,25 MiB.
2. **8 bit indis + palet.** İndisler 1024 × 768 × 1 = 786.432 bayt; palet 256 × 3 = 768 bayt. Toplam 787.200 bayt.
3. **Oran.** 2.359.296 / 787.200 ≈ 3,0. Yer yaklaşık üçte bire iner.

Biçim	Boyut (bayt)	Oran
(a) 24 bit/piksel	2.359.296 (2,25 MiB)	1
(b)+(c) 8 bit indis + palet	786.432 + 768 = 787.200	≈ 1/3

Tablo 14.10: 1024 × 768 bir görüntünün iki biçimdeki boyutu.

Kazanılan yerin bedeli iki türlüdür. Zaman bedeli, her piksel gösterilirken yapılan tablo bakışıdır. Kısıt bedeli ise en fazla 256 renk sınırıdır; fotoğraf gibi binlerce rengi olan bir görüntü bu biçime kayıpsız sığmaz. GIF ve PNG-8 bu yöntemi kullanır; GIF ayrıca LZW ile sıkıştırır. Paletin kendisi 768 bayttır ve görüntü büyüdükçe toplam içindeki payı önemsizleşir.

Sık yapılan hata

Paletin boyutunu hesaba katmamak ya da paleti piksel başına saymak. Palet görüntü başına bir kez saklanır. 800 × 600 bir görüntüde indisler 480.000, palet 768 bayttır; toplam 480.768 bayt eder. 24 bitlik biçim ise 1.440.000 bayttır ve oran yine yaklaşık 3'tür.

14.6.5 Önbelleğe almak

Tanım 14.19

Önbellek (*cache*), pahalı bir hesabın ya da yavaş bir erişimin sonucunu, aynı istek yeniden geldiğinde hızlıca verebilmek için saklayan yapıdır. Önbellekteki değer her zaman doğru olmalıdır; kaynak değişince kayıt geçersiz kılınır.

Genel kural basittir: aynı sonuç tekrar tekrar hesaplanıyorsa saklanır. Ad çözümleme bunun için biçilmiş kaftandır. `/usr/ast/mbox` yolunu önbelleksiz bulmak yedi disk okuması istiyordu. Çekirdek çözdüğü her yolun sonucunu bir tabloda tutarsa aynı yol ikinci kez sorulduğunda diske hiç gidilmez (Tablo 14.11).

Yol	i-node
`/usr`	6
`/usr/ast`	26
`/usr/ast/mbox`	60
`/usr/ast/books`	92
`/usr/bal`	45
`/usr/bal/paper.ps`	85

Tablo 14.11: Yol → i-node önbelleği.

Üç durum ayırt edilir. Önbellek yokken `/usr/ast/mbox`'ı bulmak için kök i-node'u, kök dizini, `/usr` i-node'u, `/usr` dizini, `/usr/ast` i-node'u ve `/usr/ast` dizini okunur. Bu altı disk okumasıdır; dosyanın kendi i-node'u ile yedi olur. Önbellekte tam yol varsa tek bir bellek araması yeter ve disk erişimi sıfırdır. Önbellekte yalnız `/usr/ast` varsa arama oradan sürer: kısmi isabet de kazançtır.

Linux'ta bu iş birkaç önbellekle yapılır. **Dizin girdisi önbelleği** (dcache, *dentry cache*) yol bileşenlerini i-node'lara eşler. i-node önbelleği i-node'ların kendisini, sayfa önbelleği ise dosya içeriğini tutar. Linux'ta boş RAM'in çoğu zaman “dolu” görünmesinin nedeni budur: kullanılmayan bellek bu önbelleklere verilir ve gerektiğinde geri alınır.

Önbelleğin bedeli **tutarlılıktır**. Bir dosya silinir ya da yeniden adlandırılırsa ona ait kayıt geçersiz kılınmalıdır; yoksa önbellek var olmayan bir dosyayı gösterir. Bu yük, önbellekteki değerin her zaman doğru olması şartından gelir. İşletim sisteminde önbellekler her katmandadır: TLB, CPU önbellekleri, sayfa önbelleği, dcache, ARP ve DNS önbellekleri.

14.6.6 Kod izleme: yol önbelleği

Kısmi isabeti de kullanan bir yol çözümleyici birkaç satırla yazılabilir. `bil[]` yolun bileşenlerini, `n` bileşen sayısını tutar. Fonksiyon önbellekteki en uzun öneki bulur ve yalnız kalan bileşenleri diskten çözer.

```
/* bil[]: "usr","bal","tez.tex"; n = 3 */
int cozumle(char *bil[], int n) {
    int k = n, ino = KOK, okuma = 0;
    while (k > 0 && !onbellekte(bil, k, &ino))
        k--;              /* en uzun önek */
    for (int j = k; j < n; j++) {
        ino = dizinde_ara(ino, bil[j]);
        okuma += 2;       /* i-node + blok */
        onbellege_ekle(bil, j + 1, ino);
    }
    return ino;
}
```

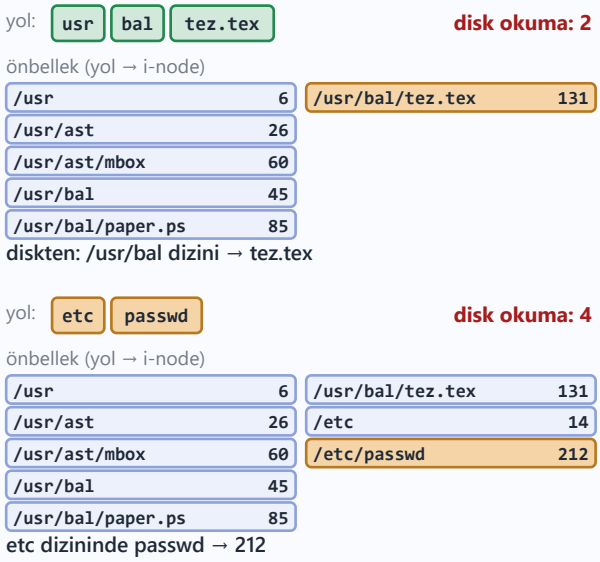
Fonksiyonun iki aşaması vardır. Birinci aşamada `k`, `n`'den başlayıp azalır. `onbellekte(bil, k, &ino)`, ilk `k` bileşenden oluşan yolun önbellekte olup olmadığına bakar; varsa i-node numarasını `ino`'ya yazar. Döngü ilk isabette ya da `k` 0'a inince durur. `k` 0'a inerse `ino` başlangıç değeri olan kök i-node'unda (`KOK` = 1) kalır. İkinci aşamada kalan her bileşen için `dizinde_ara` çağrılır. Bu çağrı dizinin i-node'unu ve dizin bloğunu okur; bu yüzden `okuma` 2 artar. Bulunan sonuç `onbellege_ekle` ile önbelleğe eklenir.

Önbellekte başlangıçta beş yol vardır: `/usr` (6), `/usr/ast` (26), `/usr/ast/mbox` (60), `/usr/bal` (45) ve `/usr/bal/paper.ps` (85). Sırayla üç arama yapılıyor (Tablo 14.12).

Yol	k (önek)	Başlangıç ino	Diskten çözülen	okuma	Sonuç
`/usr/bal/tez.tex`	2 (`/usr/bal`)	45	`tez.tex` → 131	2	131
`/usr/bal/tez.tex`	3 (tam yol)	131	yok	0	131
`/etc/passwd`	0	1 (kök)	`etc` → 14, `passwd` → 212	4	212

Tablo 14.12: `cozumle()` izlemesi.

- **İlk arama.** Tam yol `/usr/bal/tez.tex` önbellekte yoktur, `k` 2'ye iner. `/usr/bal` önbellektedir: `ino = 45`. Kökten başlamaya gerek kalmamış, iki bileşen bedavaya gelmiştir. Kalan tek bileşen için 45'in i-node'u ve dizin bloğu okunur: `tez.tex` → 131. Sonuç önbelleğe eklenir.
- **Aynı yol yeniden.** Tam yol artık önbellektedir. Birinci döngü hemen durur, ikinci döngü hiç dönmez. Disk okuması 0'dır.
- **`/etc/passwd`.** Ne tam yol ne de `/etc` önbellektedir; `k` 0'a iner ve çözüm kök i-node'undan başlar. Kök i-node'u ve kök dizin bloğu okunur: `etc` → 14; `/etc` önbelleğe girer. Ardından `/etc` i-node'u ve dizin bloğu: `passwd` → 212. Toplam 4 okuma.



Şekil 14.28: (a) `/usr/bal` önbellekten gelir; yalnız `tez.tex` diskten çözülür (2 okuma) ve önbelleğe eklenir. (b) `/etc/passwd` kökten çözülür (4 okuma); sonunda önbellekte sekiz yol vardır.

Önbelleksiz yolda bileşen başına iki okuma gerekir; en uzun önek önbellekteyse yalnız kalan bileşenler için ödeme yapılır. En kötü durum, hiçbir önekin önbellekte olmadığı durumdur ve maliyet bileşen sayısıyla doğru orantılıdır. Linux'ta bu önbellek dcache'tir ve bileşenleri ayrı ayrı tutar; mantık ise aynıdır. Kodun saydığı okumalar dosyanın kendi i-node'unu içermez; fonksiyon yalnız i-node numarasını döndürür.

Sık yapılan hata

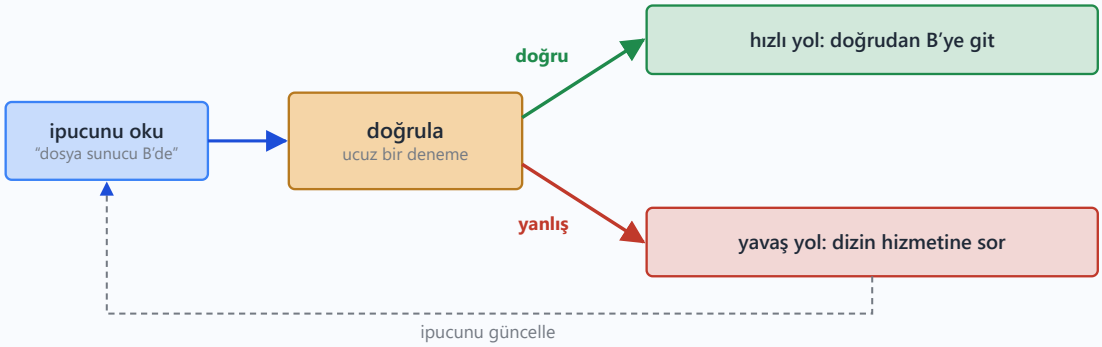
Önbelleği yalnız tam isabet için kullanmak. Tam yol önbellekte yoksa aramaya kökten başlamak, önbellekteki `/usr/bal` bilgisini çöpe atmak demektir. En uzun önek kuralı, kısmi isabetten de yararlanır.

14.6.7 İpuçları

Tanım 14.20

İpucu (*hint*), doğru olması muhtemel, ama yanlış da olabilen bir bilgidir. İpucu geçersiz kılınmaz; bunun yerine kullanılmadan önce ucuz bir denetimle doğrulanır.

İpucu önbelleğe benzer, ama aradaki fark önemlidir. Önbellekteki değer her zaman doğru olmalıdır; bu yüzden kaynak değiştiğinde önbellek güncellenmek zorundadır ve bu tutarlılık yükü getirir. İpucunda böyle bir yük yoktur. İpucu yanlış çıkarsa sistem bunu kullanım anında fark eder ve yavaş yola döner (Şekil 14.29).



Şekil 14.29: İpucu kullanımı. İpucu okunur ("dosya sunucu B'de") ve ucuz bir denemeyle doğrulanır. Doğruysa hızlı yoldan doğrudan B'ye gidilir; yanlışsa yavaş yoldan dizin hizmetine sorulur ve ipucu güncellenir.

Dağıtık bir dosya sistemi örneği tekniği açıklar. İstemci, bir dosyanın hangi sunucuda olduğunu bir ipucu olarak tutar. Dosya bir sunucudan diğerine taşınabilir ve istemcinin bundan haberi olmayabilir. İstemci önce ipucundaki B sunucusuna sorar. B dosyayı gerçekten tutuyorsa iş bitmiştir. Tutmuyorsa istemci dizin hizmetine başvurur, dosyanın yeni yerini öğrenir ve ipucunu günceller. Dosya taşındığında hiçbir istemciye haber vermek gerekmez.

Diğer örnekler:

- **Web'de yönlendirme önbelleği.** Bir adresin taşındığı yer hatırlanır; yanlışsa sunucu yeniden yönlendirir.
- **Dal tahmini.** İşlemci dalın yönünü tahmin eder ve o yoldan çalışmaya başlar; tahmin yanlışsa sonuçları atar.
- `madvise(MADV_SEQUENTIAL)`. Uygulama çekirdeğe bir bellek bölgesini sırayla okuyacağını bildirir. Çekirdek daha agresif önden okuma yapar. İpucu yanlışsa yalnız performans düşer, doğruluk bozulmaz.

Sık yapılan hata

İpucunun yanlış olabilmesini bir kusur saymak. İpucunun bütün değeri buradadır: yanlış olmasına izin verildiği için geçersiz kılma mesajlarına, kilitlere ve tutarlılık protokollerine gerek kalmaz. Şart tek bir şeydir: yanlış bir ipucu yanlış bir sonuca yol açmamalı, yalnız yavaş yola düşürmelidir.

14.6.8 Yerelliği kullanmak

Tanım 14.21

Zamansal yerellik (*temporal locality*): yakın zamanda kullanılan veri yakında yeniden kullanılır. **Uzamsal yerellik** (*spatial locality*): bir veri kullanılınca yakınındaki veri de kullanılır.

Zamansal yerellik önbelleğin temelidir: yakın zamanda kullanılan şey tutulursa bir sonraki istek büyük olasılıkla isabet eder. Uzamsal yerellik ise önden okumanın ve büyük sayfaların temelidir: bir blok istendiğinde ardındaki birkaç blok da getirilir.

Yerelliği kullanmanın bir yolu, birlikte kullanılan verileri yan yana koymaktır (Şekil 14.30). BSD'nin hızlı dosya sistemi (FFS) diski silindir gruplarına, ext4 ise blok gruplarına böler. Bir dosyanın i-node'u ile veri blokları aynı gruba konur; böylece i-node'u okuduktan sonra veriye ulaşmak için diskin kolu uzağa gitmez. Aynı ilke işlemcide de geçerlidir. Bir iş parçacığı hep aynı CPU çekirdeğinde zamanlanırsa o çekirdeğin L1 ve L2 önbellekleri "sıcak" kalır. NUMA makinelerinde bellek, iş parçacığının çalıştığı düğümden ayrılır.



Şekil 14.30: Yerelliği kullanmak. Üstte ext4 blok grupları: i-node ile verisi aynı gruptadır, arama kısadır. Altta CPU yakınlığı (affinity): iş parçacığı aynı çekirdekte kalırsa L1/L2 önbelleği sıcak kalır.

14.6.9 Yaygın durumu hızlandırmak: futex

Butler Lampson'ın sistem tasarımı üzerine verdiği öğütlerden biri şudur: yaygın durum hızlı, nadir durum doğru olmalıdır. Amdahl yasası bunun nedenini gösterir; kazanç en çok zaman alan kısmı hızlandırmaktan gelir. Kilitler bu ilkenin en güzel örneğidir.

Bir kilidi almak için her seferinde çekirdeğe girmek pahalıdır: bir sistem çağrısı, mod geçişi ve kuyruk işlemleri. Oysa ölçümler, kilitlerin büyük çoğunluğunun çekişmesiz alındığını gösterir. Kilidi isteyen iş parçacığı çoğu zaman tektir. Yaygın durum buysa hızlandırılması gereken de odur.

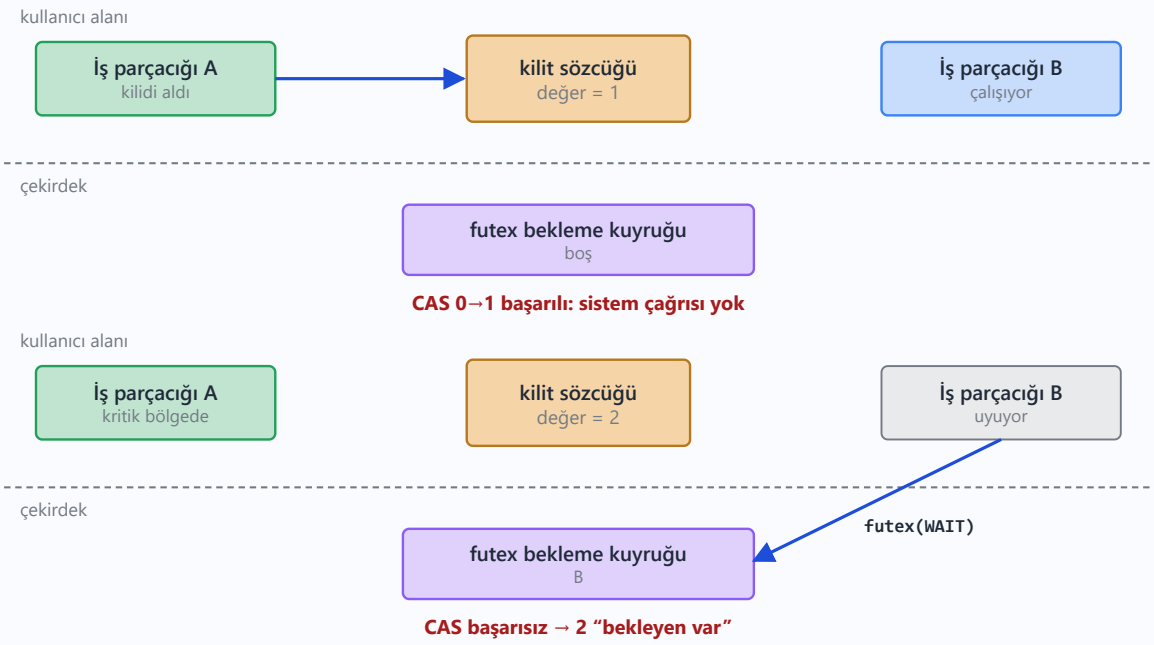
Tanım 14.22

futex (*fast userspace mutex*), Linux'ta kilidin durumunun kullanıcı alanındaki bir tam sayıda tutulduğu, çekirdeğe yalnız bekleme ve uyandırma için girilen senkronizasyon mekanizmasıdır. Kilit sözcüğünün değeri 0 (boş), 1 (alınmış) ya da 2 (alınmış, bekleyen var) olur.

Kilit sözcüğü üzerinde atomik **karşılaştır-değiştir** (CAS, *compare-and-swap*) işlemi kullanılır. CAS bir bellek sözcüğünü, beklenen değere eşitse yeni değerle tek bir bölünmez adımda değiştirir ve başarılı olup olmadığını bildirir. İki iş parçacığının kilidi sırayla istediği bir senaryo dört adımda izlenir (Tablo 14.13).

Adım	Olay	Kilit sözcüğü	Bekleme kuyruğu	Sistem çağrısı
1	Kilit boş, A kilidi istiyor	0	boş	—
2	A: CAS 0 → 1 başarılı	1	boş	yok
3	B: CAS başarısız, değeri 2 yapar	2	B	`futex(FUTEX_WAIT)`
4	A bırakır; eski değer 2	0	boş	`futex(FUTEX_WAKE)`

Tablo 14.13: futex tabanlı kilidin izlemesi.



Şekil 14.31: futex tabanlı kilit. (a) Çekişme yoktur: A, CAS 0 → 1 ile kilidi kullanıcı alanında alır; sistem çağrısı yapılmaz. (b) Nadir durum: B'nin CAS'ı başarısız olur, değeri 2 yapar ve `futex(WAIT)` ile çekirdekte uyur.

- **Hızlı yol.** A, CAS ile 0'ı 1 yapar. Çekirdeğe hiç girilmez; işlem birkaç nanosaniye sürer.

- **Yavaş yol.** B de kilidi ister; kilit sözcüğü 0 olmadığı için CAS başarısız olur. B değeri 2 yapar, yani "bekleyen var" işaretini koyar, ve `futex(FUTEX_WAIT)` ile çekirdekte uyur. Çekirdeğe yalnız burada girilir.
- **Bırakma.** A kilidi bırakırken sözcüğü 0 yapar ve eski değere bakar. Eski değer 2 ise `futex(FUTEX_WAKE)` çağırır ve B uyanıp kilidi alır. Eski değer 1 olsaydı hiç çağrı yapmazdı.

Çekişmesiz bir kilit alma ve bırakma işlemi böylece hiç sistem çağrısı yapmadan biter. Çekirdek yalnız gerçekten bekleme gerektiğinde devreye girer. 1, 2 değerlerinin ayrımı da yaygın durum ilkesinin parçasıdır: kimse beklemiyorsa bırakırken bile çekirdeğe gidilmez.

Aynı ilkenin başka örnekleri de vardır. Sistem çağrılarının sık kullanılanları için çekirdekte kısa bir hızlı yol bulunur. `clock_gettime` gibi bazı çağrılar vDSO sayesinde çekirdeğe hiç girmeden, kullanıcı alanına eşlenmiş bir sayfadan okunur. TLB isabeti de aynı mantıktadır: adres çevirisinin yaygın durumu tablo yürüyüşü olmadan biter.

Sık yapılan hata

futex'i başlı başına bir kilit sanmak. futex bir kilit değil, bir bekleme–uyandırma mekanizmasıdır. Kilidin kendisi kullanıcı alanındaki tam sayı ve ona uygulanan CAS işlemleridir. `pthread_mutex` gibi kütüphane kilitleri bu iki parçayı birleştirir.

14.7 Proje yönetimi

Bir işletim sistemi yüzlerce, bazen binlerce kişinin yıllarca süren emeğidir. Bu ölçekte teknik kararlar kadar insanların nasıl birlikte çalıştığı da sonucu belirler. Fred Brooks'un IBM OS/360 projesinden çıkardığı dersler bugün de geçerlidir.

14.7.1 Efsanevi adam-ay ve Brooks yasası

Tanım 14.23

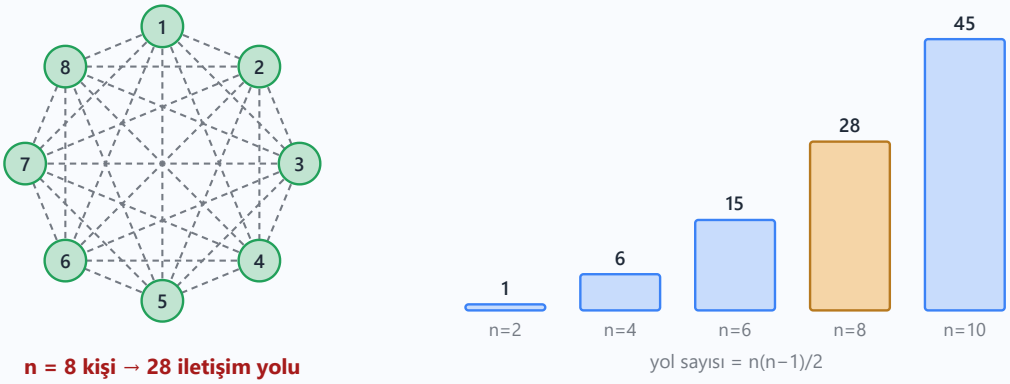
Adam-ay (*man-month*), bir kişinin bir aylık çalışmasıdır ve proje büyüklüğünü ölçmek için kullanılır. **Brooks yasası**: gecikmiş bir yazılım projesine eleman eklemek onu daha da geciktirir.

Adam-ay yanıltıcı bir birimdir, çünkü işin kişiler arasında serbestçe bölünebildiğini varsayar. 12 adam-aylık bir iş, 12 kişiyle 1 ayda bitmez. Kişiler birbirleriyle eşgüdüm kurmak zorundadır ve iletişim yolu sayısı ekip büyüdükçe karesel artar.

Örnek 14.11

Her çift arasında bir iletişim yolu varsa n kişilik bir ekipte kaç yol vardır? 2, 4, 6, 8 ve 10 kişi için hesaplanırsa ne görülür?

1. **Formül.** n kişiden her biri diğer $n - 1$ kişiyle bağlıdır; her yol iki kez sayıldığı için toplam $n(n - 1) / 2$'dir.
2. **Değerler.** 2 kişide 1, 4 kişide 6, 6 kişide 15, 8 kişide 28, 10 kişide 45 yol.
3. **Yorum.** Kişi sayısı 2'den 10'a beş kat artarken yol sayısı 1'den 45'e çıkar. 5 kişilik bir ekibe 3 kişi eklemek kişi sayısını 1,6 kat, yol sayısını ise 10'dan 28'e, yani 2,8 kat artırır.



Şekil 14.32: 8 kişilik ekipte 28 iletişim yolu. Sağdaki çubuklar yol sayısının $n(n - 1)/2$ olarak karesel artışını gösterir: 1, 6, 15, 28, 45.

Her yeni kişi herkesle eşgüdüm kurmak zorundadır: toplantılar, belgeler ve arayüz anlaşmaları artar. Gecikmiş bir projeye eklenen kişinin bir de eğitilmesi gerekir. Onu eğitecek olanlar da projeyi zaten bilen, en değerli kişilerdir. Kısa vadede eklenen her kişi ekibin toplam verimini düşürür. Brooks yasası bu iki etkinin birleşimidir.

Brooks, OS/360 deneyimine dayanarak bir zaman dağılımı da önerir: zamanın 1/3'ü planlamaya, 1/6'sı kodlamaya, 1/4'ü modül testine, 1/4'ü sistem testine. 12 aylık bir projede bu planlamaya 4, kodlamaya 2, modül testine 3 ve sistem testine 3 ay demektir. Kod yazmak işin yalnız altıda biridir; zamanın yarısı teste gider.

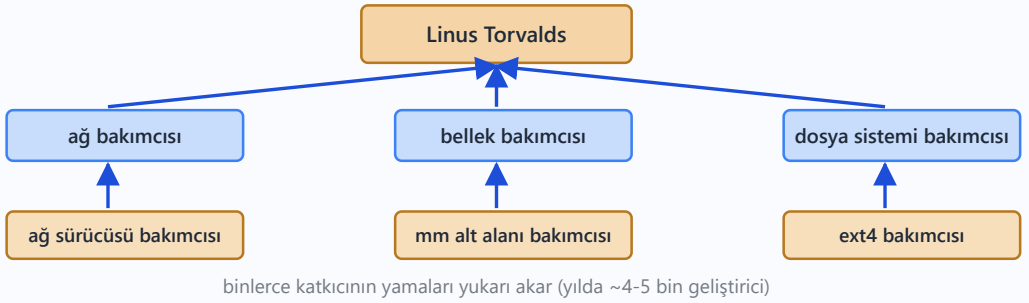
Sık yapılan hata

Brooks yasasını "ekip hiçbir zaman büyütülmemeli" diye okumak. Yasa gecikmiş projeye geç eklenen kişiler içindir. Planlama aşamasında, işin bağımsız parçalara bölünebildiği bir projede ekip büyütmek işe yarayabilir. Sorun, işi bölünemeyen ve öğrenme maliyeti yüksek bir anda kişi eklemektir.

14.7.2 Takım yapısı, deneyim ve gümüş kurşun

İletişim yükünü azaltmanın yolu, ekibi küçük ve güçlü takımlara bölmektir. Harlan Mills'in önerdiği ve Brooks'un benimsediği **cerrah ekibi** modelinde baş programcı tasarlar ve kodlar; diğerleri ona destek olur: biri yardımcıdır, biri belgeleri yönetir, biri araçları hazırlar. Kavramsal bütünlük birkaç kişide kalır. Brooks'a göre bir sistemin kavramsal bütünlüğü, tasarımının en önemli niteliğidir.

Linux'un geliştirme düzeni bu fikrin bugünkü, büyük ölçekli bir biçimidir (Şekil 14.33). Binlerce katkıcının yamaları önce sürücü ve alt alan bakımcılarına, oradan alt sistem bakımcılarına, en sonunda Linus Torvalds'a akar. Her düzeyde kavramsal bütünlüğü koruyan, sayıca az bir grup vardır.



Şekil 14.33: Linux geliştirme hiyerarşisi: katkıcıların yamaları sürücü ve alt alan bakımcılarına (ağ sürücüsü, mm, ext4), oradan alt sistem bakımcılarına (ağ, bellek, dosya sistemi) ve Linus Torvalds'a akar. Yılda yaklaşık 4–5 bin geliştirici katkı yapar.

Deneyimin rolü de büyüktür. İyi bir programcı ortalamanın kat kat üzerinde verimlidir; bu fark ekipteki kişi sayısından daha belirleyici olabilir. Brooks'un iki uyarısı buraya bağlanır. Birincisi "birini atmayı planlayın" sözüdür: ilk sürüm büyük olasılıkla atılacaktır, bu yüzden baştan bir prototip olarak düşünülmelidir. İkincisi **ikinci sistem etkisidir**: ilk sistemini başarıyla bitiren bir tasarımcı, ikincisine ilkinde bıraktığı bütün süsleri ekler ve sistem şişer.

Brooks 1986'daki "No Silver Bullet" makalesinde daha genel bir sonuca varır: yazılım geliştirmede verimi tek başına on kat artıracak bir "gümüş kurşun" yoktur. Karmaşıklığı ikiye ayırır. **Arızi karmaşıklık** (*accidental complexity*) araçlardan ve dillerden gelir; daha iyi derleyiciler, editörler ve diller onu azaltır. **Özsel karmaşıklık** (*essential complexity*) ise sorunun kendisindedir ve hiçbir araç onu ortadan kaldıramaz. Bir işletim sisteminin eş zamanlılığı, güvenlik gereksinimleri ve uyumluluk yükü özseldir. Yapay zekâ destekli kodlama araçları da bu tabloyu değiştirmez: kod yazmayı hızlandırırlar, ama tasarım sorumluluğunu kaldırmazlar.

14.8 Test ve hata ayıklama

Brooks'un zaman dağılımında testin payı yarıdır. İşletim sistemlerinde bu pay belki daha da büyüktür, çünkü çekirdekteki bir hata bütün sistemi etkiler. Bu kısım çekirdek kodunun neden zor test edildiğini, hangi test türlerinin ve araçların kullanıldığını ve bir hatanın hangi yollarla bulunduğunu anlatır.

14.8.1 İşletim sistemi kodunu test etmek neden zor?

Bir uygulamada başarısız olan test çoğu zaman yalnız bir hata mesajı üretir. Çekirdekte ise beş etken işi zorlaştırır.

- **Ayrıcalıklı mod.** Hatayı yakalayacak bir üst katman yoktur. Çekirdekte tek bir hatalı işaretçi bir oops ya da panic üretir ve bütün sistemi, test düzeneğiyle birlikte düşürür.
- **Eş zamanlılık.** Kesmeler, ön alım ve çok çekirdek sayısız iç içe geçme üretir. Bir hata ancak nadir bir zamanlamada ortaya çıkar.
- **Belirlenimsizlik.** Bazı hatalar her çalıştırmada tekrarlanmaz. Bunlara şakayla **heisenbug** denir: `printk` eklemek bile zamanlamayı değiştirip hatayı saklayabilir.
- **Çeşitlilik.** Binlerce sürücü, birçok mimari (x86-64, arm64, RISC-V) ve on binlerce Kconfig seçeneği vardır. Bütün bileşimleri denemek mümkün değildir.
- **Uzun ömürlü durum.** Bellek sızıntıları ve sayaç taşmaları ancak sistem günlerce çalıştıktan sonra görünür.

Test türleri bu zorluklara göre katmanlanır (Tablo 14.14).

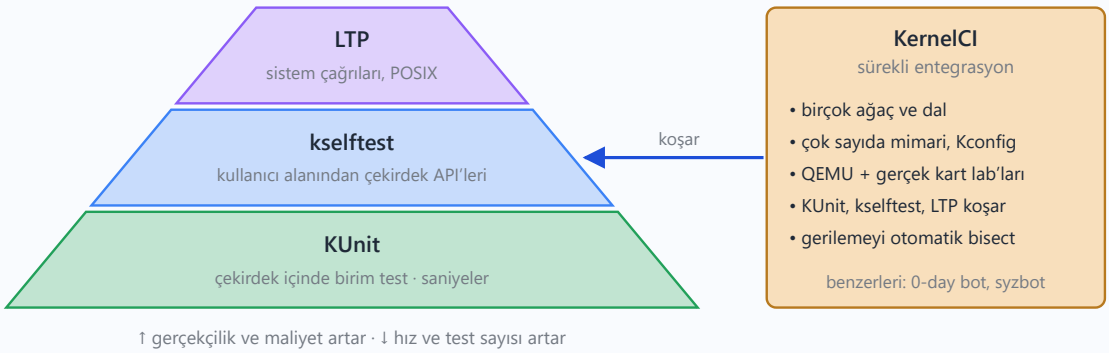
Tür	Ne sınar? (çekirdek örneği)
Birim	Tek işlev ya da modül, yalıtılmış hâlde (KUnit)
Entegrasyon	Alt sistemlerin birlikte çalışması, kullanıcı alanı arayüzü (kselftest)
Sistem	Bütün işletim sistemi, gerçek ya da sanal makinede (LTP, açılış testleri)
Regresyon	Düzeltilen hata geri geldi mi? Her değişiklikte yeniden koşulur (KernelCI)
Stres / yük	Aşırı yük, uzun süre, kaynak kıtlığı (stress-ng, rcutorture)

Tablo 14.14: Test türleri ve çekirdekteki karşılıkları.

Çekirdekte başarısız bir test çoğu zaman makinenin kendisini kaybetmek demektir. Bu yüzden testler sanal makinelerde koşulur; her testin bir zaman aşımı vardır ve makine çökerse otomatik olarak yeniden başlatılır.

14.8.2 Çekirdek test piramidi

Linux'taki test araçları bir piramit oluşturur (Şekil 14.34). Tabanda çok sayıda hızlı birim testi, tepede az sayıda yavaş ama gerçekçi sistem testi vardır.



Şekil 14.34: Çekirdek test piramidi. Tabanda çekirdek içinde koşan KUnit birim testleri, ortada kullanıcı alanından çekirdek API'lerini sınayan kselftest, tepede LTP sistem testleri. Yukarı çıktıkça gerçekçilik ve maliyet, aşağı indikçe hız ve test sayısı artar. KernelCI hepsini sürekli koşar.

- **KUnit.** Testler çekirdekle birlikte derlenir ve iç işlevleri doğrudan çağırır. UML (*User-Mode Linux*) ya da QEMU'da saniyeler içinde koşar; `kunit.py run` komutuyla başlatılır. Çıktı KTAP biçimindedir.
- **kselftest.** `tools/testing/selftests` altındadır. Çalışan çekirdeğin sistem çağrılarını ve cgroup, bpf, ağ gibi arayüzlerini dışarıdan, kullanıcı alanından sınar; `make kselftest` ile koşulur.
- **LTP.** Linux Test Project binlerce testten oluşur: sistem çağrılarının anlamı, POSIX uyumu, dosya sistemi ve stres senaryoları. Dağıtımlar yeni sürümden önce LTP koşar.

KernelCI bu testleri sürekli entegrasyon içinde birçok ağaç ve dalda, çok sayıda mimari ve Kconfig bileşiminde koşar. QEMU'nun yanında gerçek kartlardan oluşan laboratuvarlar da kullanılır. Bir gerileme bulunduğunda KernelCI onu otomatik olarak ikili aramayla bir commit'e kadar daraltır. 0-day bot ve syzbot benzer sürekli test hizmetleridir.

14.8.3 Dinamik denetleyiciler

Bir test ancak hatanın bir belirtisini görebiliyorsa onu yakalar. Bellek hatalarının çoğu ise belirti vermez: dizinin bir bayt ötesine yazmak çoğu zaman başka bir nesneyi sessizce bozar ve sistem çok sonra, ilgisiz bir yerde çöker. Dinamik denetleyiciler bu sessiz hataları ilk geçersiz erişim anında görünür kılar (Tablo 14.15).

Araç	Yakaladığı hata	Nasıl çalışır?
KASAN `CONFIG_KASAN`	Serbest bırakma sonrası kullanım; heap, yığın ve global dizi taşması; çift serbest bırakma	Derleyici her bellek erişiminin önüne denetim ekler. Gölge bellek (her 8 bayta 1 bayt) hangi baytların geçerli olduğunu tutar. Rapor ayırma ve serbest bırakma yığın izlerini de verir.
KCSAN `CONFIG_KCSAN`	Veri yarışı: en az biri yazma olan, eş zamanlı ve kilitsiz erişim	Rastgele erişimlerde bir gözlem noktası kurup kısa süre bekler; bu arada başka bir çekirdek aynı adrese dokunursa raporlar.
UBSAN `CONFIG_UBSAN`	Tanımsız davranış: işaretli taşma, sıfıra bölme, geçersiz kaydırma, dizi sınırı	Derleyicinin eklediği çalışma zamanı denetimleri.

Araç	Yakaladığı hata	Nasıl çalışır?
lockdep `CONFIG_PROVE_LOCKING`	Kilit sırası ihlali (AB-BA), kesme bağlamında hatalı kilit kullanımı	Kilit sınıfları arasındaki "önce–sonra" çizgesini tutar; çevrim görünce kilitlenme gerçekleşmeden uyarır.
kmemleak `CONFIG_DEBUG_KMEMLEAK`	Bellek sızıntısı	Çöp toplayıcı gibi belleği tarar; hiçbir işaretçinin göstermediği ayrılmış blokları raporlar.

Tablo 14.15: Çekirdeğin dinamik denetleyicileri.

KASAN'ın tipik bir raporu iki satırla başlar:

```
BUG: KASAN: slab-use-after-free in mydrv_release+0x54/0x90
Read of size 8 at addr ffff888012a4c010 by task cat/1432
```

İlk satır hatanın türünü (*slab-use-after-free*: serbest bırakılmış bir slab nesnesine erişim) ve yerini verir: `mydrv_release` fonksiyonunun 0x54 baytındaki komut. İkinci satır erişimin boyunu, adresini ve hangi görevin yaptığını söyler: `cat` komutunun 1432 numaralı süreci 8 baytlık bir okuma yapmıştır. Raporun devamında nesnenin nerede ayrıldığını ve nerede serbest bırakıldığını gösteren iki yığın izi gelir; nesnenin ömrü bunlardan çıkarılır.

Denetleyicilerin bedeli yüksektir. KASAN çekirdeği birkaç kat yavaşlatır ve belleğin yaklaşık sekizde birini gölgeye ayırır. KCSAN ve lockdep de belirgin yük getirir. Bu yüzden geliştirme, test ve sürekli entegrasyon çekirdeklerinde açılırlar. Üretim sistemlerinde ise örneklemeli ve düşük maliyetli KFENCE kullanılabilir: yalnız ara sıra seçilen ayırmaları denetler.

14.8.4 Kod izleme: KASAN gölge belleği

Tanım 14.24

Gölge bellek (*shadow memory*), KASAN'ın her 8 baytlık bellek bölgesi için tuttuğu 1 baytlık durum bilgisidir. Değer 0 ise bölgenin 8 baytı da geçerlidir. 1…7 arası bir k değeri yalnız ilk k baytın geçerli olduğunu gösterir. Negatif değerler kırmızı bölge ya da serbest bırakılmış bellek demektir.

KASAN'ın her erişimden önce yaptığı denetim, sadeleştirilmiş hâliyle şöyledir:

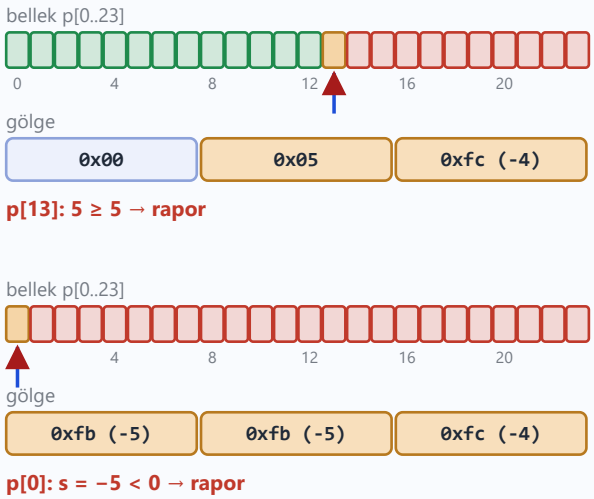
```
/* 8 baytlık bölge başına 1 gölge bayt:
   0: hepsi geçerli, 1..7: ilk k bayt,
   < 0: kırmızı bölge ya da serbest */
void kasan_denetle(void *a, int boy) {
    int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
                           + GOLGE_OFS);
    if (s == 0) return;            /* hızlı yol */
    if ((int)((uintptr_t)a & 7) + boy - 1 >= s)
        rapor(a, s);
}
/* p = kmalloc(13); p[3]; p[12]; p[13]; */
/* kfree(p); p[0]; */
```

Fonksiyon önce gölge baytı bulur. Adres 3 bit sağa kaydırılır, yani 8'e bölünür; sonuca sabit bir bağıl konum eklenir. Gölge baytı 0 ise bölgenin tamamı geçerlidir ve denetim tek bir karşılaştırmayla biter. Bu hızlı yoldur ve erişimlerin büyük çoğunluğu buradan geçer. Değer sıfır değilse erişimin bölge içindeki son baytının konumu hesaplanır: `(a & 7) + boy - 1`. Bu konum gölge değerine eşit ya da büyükse erişim geçerli baytların dışına taşıyordur ve rapor üretilir. Karşılaştırmanın işaretli yapılması önemlidir: konum `(int)` ile tam sayıya çevrildiği için `s` de işaretli olarak karşılaştırılır. Böylece negatif bir gölge değeri her konumda bu koşulu sağlar; Linux KASAN da bu denetimi işaretli bir baytla yapar. Çevirme yapılmasaydı `uintptr_t` yüzünden karşılaştırma işaretsiz olur, −5 çok büyük bir sayıya dönüşür ve serbest bırakılmış bellek hiç raporlanmazdı.

Senaryoda `p = kmalloc(13)` çağrılıyor. Çekirdeğin ayırıcısı 13 baytlık istek için 16 baytlık bir yuva verir. Gölge şöyle kurulur: ilk 8 bayt tamamen geçerlidir (0), ikinci bölgenin ilk 13 − 8 = 5 baytı geçerlidir (5), arkasındaki bölge kırmızı bölgedir (0xfc, yani −4). Ardından sırayla `p[3]`, `p[12]` ve `p[13]` okunuyor; sonra `kfree(p)` ve `p[0]` geliyor (Tablo 14.16).

Erişim	Bölge (a >> 3)	Gölge s	(a & 7) + boy − 1	Sonuç
`p[3]`	0	0	—	hızlı yol, geçerli
`p[12]`	1	5	4	4 ≥ 5 değil, geçerli
`p[13]`	1	5	5	5 ≥ 5, rapor
`kfree(p)`	0, 1	0xfb (−5)	—	gölge “serbest” olarak işaretlenir
`p[0]`	0	−5	0	0 ≥ −5, rapor (serbest bırakma sonrası kullanım)

Tablo 14.16: KASAN denetimi. Tek baytlık okumalar (`boy` = 1).



Şekil 14.35: (a) `p[13]` : dizinin bir bayt ötesi; 5 ≥ 5 olduğu için rapor üretilir. (b) `kfree(p)` sonrası nesnenin gölgesi 0xfb'dir; `p[0]` okuması serbest bırakma sonrası kullanım olarak raporlanır.

`p[12]`, 13 baytlık nesnenin son geçerli baytıdır ve sorunsuz geçer. `p[13]` ise dizinin bir bayt ötesidir. Gerçek bir çekirdekte bu bayt yuvanın dolgusuna ya da başka bir nesneye ait olabilir; KASAN olmadan hata sessiz kalırdı. KASAN onu ilk geçersiz erişimde raporlar.

`kfree(p)` nesnenin gölgesini 0xfb (serbest) olarak işaretler. Yuva bir süre **karantinada** bekletilir; hemen yeniden kullanılırsa yeni nesnenin gölgesi geçerli olur ve eski işaretçiyle yapılan erişim gizlenir. `p[0]` okunduğunda gölge −5'tir ve rapor, nesnenin ayrıldığı ve serbest bırakıldığı yerlerin yığın izleriyle birlikte gelir.

Sık yapılan hata

KASAN'ın yalnız çöken kodu yakaladığını sanmak. KASAN'ın değeri tam tersindedir: `p[13]` okuması KASAN'sız bir çekirdekte hiçbir belirti vermez. Denetleyici, sessiz kalacak bir hatayı ilk anda yakalar. Bir başka hata da çok baytlı erişimi unutmaktır: 4 baytlık bir okuma, başlangıç adresi geçerli olsa bile geçersiz baytlara taşabilir. Formüldeki `boy - 1` terimi bu yüzden vardır.

14.8.5 Kod izleme: lockdep kilit sırası çizgesi

Kilitlenmelerin en sık nedeni, iki kilidin farklı yerlerde ters sırayla alınmasıdır. Bir iş parçacığı A'yı tutarken B'yi, diğeri B'yi tutarken A'yı isterse ikisi sonsuza kadar bekleyebilir. lockdep bu durumu kilitlenme yaşanmadan önce bulur.

Tanım 14.25

lockdep, Linux çekirdeğinin kilit doğrulayıcısıdır. Kilit örneklerini değil kilit sınıflarını izler ve "H tutulurken L alındı" ilişkisini bir yönlü çizgede H → L kenarı olarak biriktirir. Yeni bir kenar çizgede bir çevrim kapatırsa olası bir kilitlenme olarak uyarır.

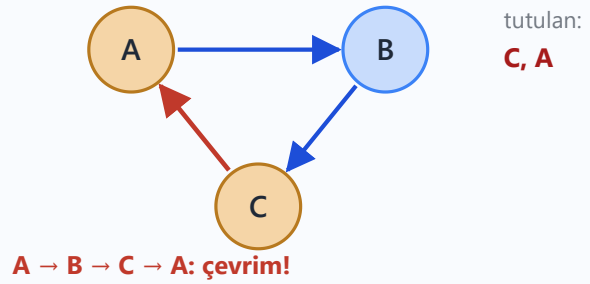
```
void lock_acquire(task *t, int L) {
    for (int i = 0; i < t->n; i++) {
        int H = t->tutulan[i];
        if (yol_var(L, H))   /* L → … → H */
            uyar("olası kilitlenme", H, L);
        kenar_ekle(H, L);    /* H, L'den önce */
    }
    t->tutulan[t->n++] = L;
}
void lock_release(task *t) { t->n--; }
/* T1: A,B   T2: B,C   T3: C,A */
```

`lock_acquire`, görev `L` kilidini alırken çağrılır. Görevin o an tuttuğu her `H` kilidi için iki şey yapılır. Önce çizgede `L`'den `H`'ye bir yol olup olmadığına bakılır. Böyle bir yol varsa bir yerde `L` tutulurken (dolaylı yoldan) `H` alınmıştır ve şimdi tersi yapılıyor demektir: uyarı verilir. Sonra `H → L` kenarı eklenir. En sonda `L` tutulan kilitler listesine konur. `lock_release` listeden son kilidi çıkarır.

Üç görev farklı zamanlarda çalışıyor: T1 önce A'yı sonra B'yi, T2 önce B'yi sonra C'yi, T3 önce C'yi sonra A'yı alıyor (Tablo 14.17).

Olay	Tutulan	Yol denetimi	Eklenen kenar	Çizge
T1: A'yı alır	—	döngü dönmez	—	boş
T1: B'yi alır	A	B → … → A? yok	A → B	A → B
T2: C'yi alır (B tutulurken)	B	C → … → B? yok	B → C	A → B → C
T3: C'yi alır	—	döngü dönmez	—	aynı
T3: A'yı alır (C tutulurken)	C	A → B → C var: **uyarı**	C → A	A → B → C → A (çevrim)

Tablo 14.17: lockdep izlemesi.



Şekil 14.36: T3, C'yi tutarken A'yı ister. Çizgede A → B → C yolu vardır; yeni C → A kenarı bir çevrim kapatır ve lockdep uyarır.

Üç görev hiç aynı anda çalışmamıştır ve gerçek bir kilitlenme olmamıştır. Ama bir gün T1 A'yı, T2 B'yi, T3 C'yi aynı anda tutarsa her biri bir sonrakinin kilidini bekler: T1 B'yi, T2 C'yi, T3 A'yı. Bu dairesel bekleme bir kilitlenmedir. lockdep bunu tek bir çalıştırmada, olmadan önce gösterir. Gücü buradan gelir: zamanlamaya bağlı, nadir bir hatayı belirlenimci bir çizge denetimine çevirir.

Sık yapılan hata

Yalnız iki kilitli AB-BA sırasını aramak. Örnekteki çevrim üç kilitten oluşur ve hiçbir görev ters sırada iki kilit almaz: T1 A–B, T2 B–C, T3 C–A alır. Sorun ancak üç ilişki birleştirildiğinde görülür. lockdep'in çizgede yol araması bu yüzden gereklidir.

14.8.6 Fuzzing: syzkaller döngüsü

Tanım 14.26

Fuzzing, bir programı çok sayıda rastgele ya da yarı rastgele girdiyle çalıştırarak çökmeler ve hatalar aramaktır. **Kapsam güdümlü** (*coverage-guided*) fuzzing'de yeni kod yolu açan girdiler saklanır ve sonraki denemelerin tohumu olur.

Linux çekirdeğinin en verimli hata bulucusu syzkaller'dır (Şekil 14.37). Döngü altı adımda işler.

1. **Açıklamalar.** syzkaller sistem çağrılarının imzalarını syzlang açıklamalarından bilir: hangi argüman dosya tanımlayıcısı, hangisi bayrak, hangisi işaretçidir. Korpus başta boştur ya da tohum programlardan oluşur.
2. **Üretim ve mutasyon.** Üretici korpustan bir program seçip değiştirir ya da yenisini üretir; örneğin `openat` → `ioctl` → `mmap` → `close`. Argümanlar türlerine uygun seçildiği için çağrılar hemen `EINVAL` ile reddedilmez.
3. **Çalıştırma.** Program, KASAN, KCSAN ve lockdep açık bir test çekirdeğiyle çalışan QEMU sanal makinelerinde koşar. VM çökse bile ana makine etkilenmez; syz-manager VM'i yeniden başlatır.

4. **Kapsam.** KCOV, çekirdekte geçilen temel blokları (PC değerlerini) kaydeder. Bu geri bildirim rastgele denemeyi yönlendirilmiş bir aramaya çevirir.
5. **Korpus.** Yeni bir kod yolu açan program korpusa eklenir ve sonraki mutasyonların tohumu olur. Döngü her VM'de saniyede yüzlerce programla durmadan döner.
6. **Rapor.** Bir çökme ya da KASAN raporu gelirse syzkaller programı yeniden üretir, en küçük hâline indirger ve çoğu zaman bir C yeniden üreticisi çıkarır. syzbot bunu otomatik olarak geliştiricilere bildirir.



Şekil 14.37: syzkaller döngüsü: korpus, üretim ve mutasyon (syzlang çağrı açıklamaları), QEMU VM'de KASAN açık test çekirdeğinde çalıştırma, KCOV kapsam geri bildirimi. Çökme olursa yeniden üretilip küçültülerek raporlanır.

Fuzzer ile denetleyici birbirini tamamlar. Fuzzer tek başına yalnız çökmeleri görür. Bellek bozan ama çökmeyen bir hata, fuzzer'ın gözünde başarılı bir çalıştırmadır. KASAN gibi denetleyiciler bu sessiz bozulmaları ilk geçersiz erişimde görünür kılar ve onları çökmeye çevirir. syzkaller'ın KASAN açık bir çekirdekle çalıştırılmasının nedeni budur.

14.8.7 Gözlem ve iz sürme araçları

Bir hata bulunduktan sonra nedenini anlamak için sistemin içini görmek gerekir. Linux bunun için katman katman araçlar sunar (Tablo 14.18).

Araç	Ne sağlar?	Örnek
`printk`	Log düzeyleri 0 (EMERG) … 7 (DEBUG); halka tampona yazılır, eşiğin üstündekiler konsola basılır	`pr_warn("kuyruk dolu: %d\n", n);`
`dmesg`	Halka tamponu okur; `kernel.dmesg_restrict=1` ise yalnız `CAP_SYSLOG` yetkisi olan okur (adres sızıntısını önler)	`dmesg -w --level=err,warn`
dynamic debug	`pr_debug` satırlarını yeniden derlemeden, çalışırken açar	`echo 'module e1000e +p' > …/dynamic_debug/control`
ftrace	Çekirdek içi izleyici: fonksiyon ve fonksiyon çizgesi izleme, tracepoint olayları (tracefs)	`trace-cmd record -p function_graph -g vfs_read`
perf	Donanım sayaçları ve örneklemeli profil: sıcak noktalar, önbellek ıskaları, çizelgeleyici gecikmesi	`perf record -g -- ./uyg`, `perf report`

Araç	Ne sağlar?	Örnek
eBPF / bpftrace	Doğrulayıcıdan geçen küçük programlar kprobe ve tracepoint'lere bağlanır; üretimde güvenli gözlem	tek satırlık örnek metinde
kdump / crash	Panikte kexec ile yedek çekirdeğe geçip bellek dökümünü (`/proc/vmcore`) kaydeder; döküm sonra çevrim dışı incelenir	`crash vmlinux vmcore`

Tablo 14.18: Gözlem ve iz sürme araçları.

bpftrace ile tek satırlık bir program, sistemde hangi sürecin hangi dosyayı açtığını canlı olarak gösterir:

```
# hangi süreç hangi dosyayı açıyor?
bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm,
str(args.filename)); }'
```

Program `sys_enter_openat` tracepoint'ine bağlanır. `openat` sistem çağrısı her başladığında çalışır ve sürecin adını (`comm`) ile açılmak istenen dosya adını yazdırır. Çekirdeği yeniden derlemek ya da makineyi yeniden başlatmak gerekmez; program doğrulayıcıdan geçtiği için çalışan bir üretim sisteminde bile güvenle kullanılabilir.

Araç seçiminde genel kural, önce gözlem sonra tahmindir: log ve izler hipotez kurmayı sağlar. Ama ağır bir `printk` zamanlamayı değiştirip bir yarış hatasını gizleyebilir. Bu yüzden düşük yüklü izleyiciler, ftrace ve eBPF, tercih edilir.

14.8.8 QEMU ve gdb ile çekirdek hata ayıklama

Bazen izler yetmez ve çekirdeği adım adım çalıştırmak gerekir. Bunun en kolay yolu çekirdeği QEMU'da çalıştırıp dışarıdan gdb ile bağlanmaktır.

```
# 1. terminal: -s → gdb sunucusu :1234
#              -S → CPU başta durur
qemu-system-x86_64 -kernel bzImage \
  -append "console=ttyS0 nokaslr" \
  -nographic -s -S
# 2. terminal
gdb vmlinux
(gdb) target remote :1234
(gdb) hbreak start_kernel
(gdb) continue
(gdb) bt
(gdb) lx-dmesg  # çekirdek gdb betiği
```

İlk terminalde QEMU çekirdeği başlatır. `-s` seçeneği 1234 numaralı portta bir gdb sunucusu açar; `-S` ise CPU'yu en baştan durdurur, böylece daha ilk komuttan önce bağlanılabilir. `console=ttyS0` çekirdeğin çıktısını seri hatta, yani terminale yönlendirir. `nokaslr` çekirdek adres alanı rastgeleleştir-

mesini kapatır; böylece gdb'deki semboller gerçek adreslerle eşleşir. İkinci terminalde gdb, sembol bilgisi içeren sıkıştırılmamış `vmlinux` dosyasıyla açılır (`CONFIG_DEBUG_INFO`). `target remote :1234` QEMU'ya bağlanır. `hbreak start_kernel` çekirdeğin C'deki giriş noktasına bir donanım kesme noktası koyar. `continue` ile oraya kadar çalışılır, `bt` çağrı yığınını gösterir. `lx-dmesg`, çekirdeğin kendi gdb betiklerinden biridir ve log tamponunu yazdırır. Aynı oturum gerçek donanımda seri hat üzerinden kgdb ile de kurulabilir (`kgdboc=ttyS0,115200`).

Gerçek bir hatada ilk bakılan şey oops mesajıdır:

```
BUG: kernel NULL pointer dereference,
     address: 0000000000000010
Oops: 0000 [#1] PREEMPT SMP NOPTI
CPU: 2 PID: 1432 Comm: cat
Tainted: G           O  6.12.0
RIP: 0010:mydrv_read+0x2a/0x80 [mydrv]
Call Trace:
 vfs_read+0xa4/0x2f0
 ksys_read+0x63/0xe0
 do_syscall_64+0x82/0x170
```

Mesaj satır satır okunur. İlk satır hatanın türünü verir: 0x10 adresinde bir NULL işaretçi erişimi. Bu tipik olarak NULL bir yapı işaretçisi üzerinden, yapının başından 16 bayt ötedeki bir alana erişmektir. `CPU: 2 PID: 1432 Comm: cat` hatayı hangi çekirdekte, hangi sürecin tetiklediğini söyler. `Tainted: G O` çekirdeğin "lekeli" olduğunu gösterir; `O` ağaç dışı bir modülün yüklü olduğu anlamına gelir. `RIP: mydrv_read+0x2a/0x80` hatalı komutun yeridir: `mydrv_read` fonksiyonunun başından 0x2a bayt ötesi; fonksiyonun boyu 0x80'dir. `scripts/faddr2line` bu bilgiden kaynak dosyadaki satırı bulur. Çağrı izi aşağıdan yukarı okunur: `do_syscall_64` → `ksys_read` → `vfs_read` → sürücü. Yani bir `read` sistem çağrısı VFS üzerinden sürücüye gelmiş ve sürücüde çökmüştür.

Oops yalnız o süreci öldürür; sistem çalışmayı sürdürebilir, ama artık güvenilir değildir. `panic_on_oops=1` ayarı oops'u doğrudan panic'e çevirir. Windows dünyasında bunun karşılığı WinDbg'dir: çekirdek modunda ağ, USB ya da sanal makine üzerinden hata ayıklanır ve mavi ekran dökümleri `!analyze -v` ile incelenir. Driver Verifier (`verifier.exe`) sürücüleri özel havuz, IRQL ve kaynak sızıntısı denetimleriyle zorlayarak hataları erkenden yakalar.

14.8.9 Regresyon avı: git bisect

Bir hata v6.11'de yokken v6.12'de ortaya çıkmışsa, onu getiren değişikliği bulmanın sistemli bir yolu vardır. İki sürüm arasındaki commit'ler bir dizi oluşturur ve dizinin bir noktasından sonra hata vardır. Bu noktayı bulmak bir ikili arama sorusudur.

Tanım 14.27

Regresyon (*regression*), daha önce doğru çalışan bir özelliğin yeni bir değişiklikten sonra bozulmasıdır. `git bisect`, bilinen iyi ve kötü iki commit arasında ikili arama yaparak hatayı getiren ilk commit'i bulan komuttur.

```
git bisect start
git bisect bad v6.12
git bisect good v6.11
# #500: derle, başlat, sına
git bisect bad
git bisect good   # #250
git bisect good   # #375
git bisect bad    # #437
git bisect run ./sina.sh
git bisect reset
```

Komutlar şu işi yapar. `git bisect start` aramayı başlatır. `bad v6.12` ve `good v6.11` iki ucu bildirir. Git aradaki commit'lerin ortasındakini çalışma dizinine çıkarır. Her adımda çekirdek derlenir, başlatılır ve sınanır; sonuca göre `git bisect bad` ya da `git bisect good` yazılır. Bir noktadan sonra sınama bir betiğe bırakılabilir: `git bisect run ./sina.sh` kalan adımları kendisi yürütür. Bitince `git bisect reset` başlangıç dalına döner.

Örnek 14.12

v6.11 (iyi) ile v6.12 (kötü) arasında 1000 commit var ve ilk kötü commit #412. Git her adımda aralığın ortasını, ⌊(iyi + kötü) / 2⌋'yi sınatıyor. Hangi commit'ler sınanır?

1. **#500.** Hata var: bad. İlk kötü commit #500 ya da öncesidir; aday 500.
2. **#250.** Hata yok: good. İlk kötü commit 251–500 arasındadır; aday 250.
3. **#375.** İyi; aday 376–500, yani 125.
4. **#437.** Kötü; aday 376–437, yani 62.
5. **git bisect run.** Betik kalanı yürütür: 406 iyi, 421 kötü, 413 kötü, 409 iyi, 411 iyi, 412 kötü.
6. **Sonuç.** “#412 is the first bad commit”. Toplam 10 test.



Şekil 14.38: git bisect: v6.11 (iyi) ile v6.12 (kötü) arasındaki 1000 commit. Her testte aday aralık yarıya iner; ilk kötü commit #412, 10 testte bulunur.

Her yanıt aday sayısını yarıya indirir: 1000 → 500 → 250 → 125 → 62 → … Bu yüzden gereken test sayısı commit sayısının logaritmasıyla büyür.

Teorem 14.2 (İkili aramanın test sınırı)

n aday commit arasında ilk kötü commit, ikili aramayla en çok $\lceil \log_2 n \rceil$ testte bulunur.

1000 commit için $\lceil \log_2 1000 \rceil = 10$'dur, çünkü $2^9 = 512 < 1000 \le 1024 = 2^{10}$. Bir milyon commit olsaydı bile yalnız $\lceil \log_2 10^6 \rceil = 20$ test gerekirdi. 12.000 commit için sınır 14'tür: $2^{13} = 8192 < 12.000 \le 16.384 = 2^{14}$.

`git bisect run` betiğin çıkış koduna bakar. Betik 0 ile biterse commit iyi, 1 ile 127 arasında (125 hariç) bir kodla biterse kötü sayılır. 125 özel bir anlama gelir: “bu commit sınanamaz, atla”. Örneğin o commit'te çekirdek derlenmiyorsa betik 125 döndürür ve git komşu bir commit'i dener. Performans gerilemelerinde ölçüm gürültülü olduğu için betik birkaç ölçümün ortalamasını almalıdır.

Sık yapılan hata

Sınanamayan bir commit için betikten 1 döndürmek. Commit derlenmediği için “kötü” sayılırsa ikili arama yanlış yarıya gider ve sonuçta yanlış commit suçlanır. Derlenmeyen ya da başlamayan bir sürüm için doğru cevap 125'tir.

14.9 İşletim sistemlerinde eğilimler

İşletim sistemleri uzun ömürlü olduğu için bugünkü kararlar on yıllar sonraki donanımı ve uygulamaları da karşılamalıdır. Bu kısım, tasarımı bugün en çok etkileyen eğilimleri toplar.

14.9.1 Genel tablo

Eğilim	İşletim sistemine etkisi
Sanallaştırma ve bulut	Misafir çekirdek virtio ile hipervizörle işbirliği yapar; çok kiracılı bulutta yalıtım sınırı hipervizöre taşınır. Konteyner hafif ama yalıtımı zayıftır; mikro-VM (Firecracker) VM yalıtımı ve milisaniyede açılış sunar.
Çok çekirdekli yongalar	Ölçeklenebilir kilitler, RCU, NUMA farkındalığı, heterojen çekirdekler
Geniş adres alanı	64 bit, 5 düzeyli sayfa tabloları; tek adres alanlı sistem fikirleri
Kesintisiz veri erişimi	Ağ, bulut depolama ve yerel disk arasında şeffaf erişim; NVMe, CXL bellek
Pille çalışan sistemler	DVFS, "hızla bitir ve uyu" (*race-to-idle*), Android Doze, enerji farkında çizelgeleme (EAS)
Gömülü ve sensör düğümleri	Gerçek zamanlı küçük çekirdekler: Zephyr, FreeRTOS, Contiki-NG; kilobaytlarca bellek
Güvenli diller ve programlanabilir çekirdek	Linux'ta Rust (6.1'den beri), eBPF ve sched_ext, unikernel'ler

Tablo 14.19: Eğilimler ve işletim sistemine etkileri.

Tanenbaum'un eski baskılarında ayrı birer eğilim sayılan ağ işlemleri, paralel ve dağıtık sistemler ve çoklu ortam artık her sistemin olağan parçasıdır. Bu yüzden ayrı birer eğilim olarak değil, tablodaki ilgili satırların içinde düşünülür.

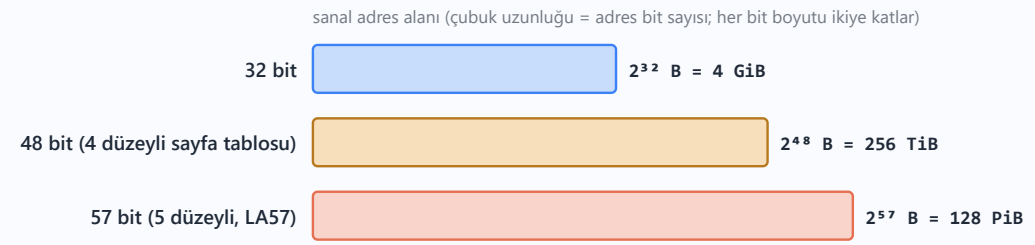
Sanallaştırma satırı yalıtım–performans ödünleşiminin yeniden çizilmesidir. Çok kiracılı bir bulutta farklı müşterilerin kodu aynı makinede çalışır ve aralarındaki asıl sınır artık işletim sistemi çekirdeği değil, hipervizördür. Konteynerler aynı çekirdeği paylaştığı için hafif ve hızlıdır, ama çekirdekteki bir açık bütün konteynerleri etkiler. Firecracker gibi mikro-VM'ler her iş yüküne ayrı ve çok küçük bir sanal makine vererek iki dünyanın arasında bir nokta seçer.

Pille çalışan sistemlerde amaç enerjidir. DVFS (*dynamic voltage and frequency scaling*) işlemcinin gerilimini ve frekansını yüke göre değiştirir. "Hızla bitir ve uyu" stratejisinde iş en yüksek hızda bitirilir ve işlemci hemen derin uyku durumuna geçirilir. Android Doze, ekran kapalıyken uygulamaların arka plan işlerini toplu pencerelere erteler.

14.9.2 Çok çekirdek ve geniş adres alanları

Bugün sunucularda yuva başına 128–192 çekirdek, telefonlarda ise büyük ve küçük çekirdeklerin karıştığı (big.LITTLE) yongalar vardır. Tek bir büyük çekirdek kilidi bu ölçekte işlemez: bütün çekirdekler aynı kilidi beklerse eklenen her çekirdek boşa gider. Gereken araçlar ince taneli kilitler, okuyucuların hiç kilit almadığı RCU (*read-copy-update*) ve her çekirdeğe ayrı veri yapılarıdır. Heterojen çekirdeklerde çizelgeleyici bir de hangi işin hangi tür çekirdeğe gideceğine karar verir.

64 bitlik adresler de yeni olanaklar açar (Şekil 14.39).



Şekil 14.39: Sanal adres alanı boyutları: 32 bit 4 GiB, 48 bit (4 düzeyli sayfa tablosu) 256 TiB, 57 bit (5 düzeyli, LA57) 128 PiB. Her bit boyutu ikiye katlar.

Örnek 14.13

48 ve 57 bitlik sanal adres alanları ne kadar büyüktür ve bu bit sayıları sayfa tablosu düzeylerinden nasıl çıkar?

1. **48 bit.** 2^{48} B = $2^{8} \times 2^{40}$ B = 256 TiB.
2. **57 bit.** 2^{57} B = $2^{7} \times 2^{50}$ B = 128 PiB.
3. **Düzeyler.** 4 KB'lık sayfada bağıl konum 12 bittir. Her tablo düzeyi 512 girdilidir ve 9 bit çözer. 4 düzey: 4 × 9 + 12 = 48 bit; 5 düzey: 5 × 9 + 12 = 57 bit.

Bu genişlikte bir adres alanında bütün dosyaları belleğe eşlemek (`mmap`) ya da bütün süreçlerin tek bir adres alanını paylaştığı sistemler kurmak mümkün olur. Bedeli daha derin bir sayfa tablosu yürüyüşüdür: 5 düzeyli tabloda bir TLB ıskası beş bellek okuması ister. Geniş adres alanı TLB üzerindeki baskıyı da artırır.

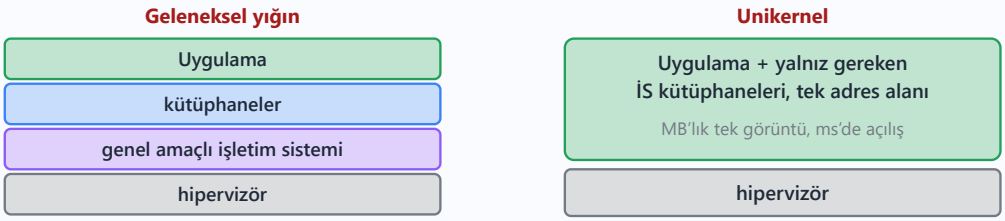
14.9.3 Rust, eBPF ve unikernel'ler

Son eğilim, çekirdeğin hangi dille ve hangi biçimde yazıldığıyla ilgilidir. Üç gelişme öne çıkar.

Rust ile çekirdek. Linux 6.1 (Aralık 2022) ile Rust desteği ana dala girdi; ardından ilk Rust sürücüleri geldi. Amaç, serbest bırakma sonrası kullanım ve taşma gibi bellek hatalarını derleme anında önlemektir. Çekirdek güvenlik açıklarının önemli bir kısmı bu türden hatalardır. Rust'ın sahiplik ve ödünç alma kuralları bir nesnenin ömrünü derleyicinin denetlemesini sağlar. Bu, KASAN'ın çalışma anında yakaladığı hataları daha kod derlenirken engellemek demektir. Windows çekirdeğinde de Rust ile yazılmış bileşenler vardır.

eBPF. Doğrulanmış küçük programlarla çekirdeği yeniden derlemeden genişletmek; izleme, ağ (XDP), güvenlik ve çizelgeleme (sched_ext) alanlarında kullanılır. Genişletilebilir sistem fikri, eBPF ile ilk kez geniş ölçekte üretime girmiştir.

Unikernel. Exokernel'in kütüphane işletim sistemi (libOS) fikrinin bulut sürümüdür. Tek bir uygulama, ona gereken işletim sistemi kütüphaneleriyle birlikte tek bir görüntü olarak derlenir ve hipervizör üzerinde tek adres alanında çalışır (Şekil 14.40). MirageOS ve Unikraft bu yaklaşımın örnekleridir.



Şekil 14.40: Geleneksel yığında uygulama, kütüphaneler ve genel amaçlı işletim sistemi hipervizörün üzerindedir. Unikernel uygulamayı yalnız gereken işletim sistemi kütüphaneleriyle tek adres alanlı, megabaytlık tek bir görüntüde birleştirir; açılış milisaniyeler sürer.

Unikernel'in kazancı küçüklük ve hızdır. Görüntü yalnız gereken kodu içerdiği için saldırı yüzeyi küçüktür ve açılış milisaniyeler sürer. Kullanıcı–çekirdek geçişi olmadığı için sistem çağrıları sıradan fonksiyon çağrılarına dönüşür. Tek uygulamalı mikro hizmetler için uygundur. Bedeli, hata ayıklamanın zorlaşması ve birden çok uygulamanın aynı görüntüde çalıştırılamamasıdır. Yalıtımı ise artık hipervizör sağlar.

14.10 Özet

- İşletim sistemi tasarımını sekiz etken zorlaştırır: devasa boyut, eş zamanlılık, düşmanca kullanıcılar, yalıtıma rağmen paylaşım, uzun ömür, genellik, taşınabilirlik ve geriye dönük uyumluluk.
- Bir işletim sisteminin dört temel hedefi soyutlamalar tanımlamak, bu soyutlamalar üzerinde ilkel işlemler (sistem çağrıları) sağlamak, yalıtım sağlamak ve donanımı yönetmektir.
- İyi bir arayüz basit, bütün ve verimlidir. Çekirdekte az ve genel çağrı, kütüphanede kolaylık fonksiyonları bulunur: altı `exec` fonksiyonu tek bir `execve`'ye dönüşür. Sistem çağrısı sayısı seL4'te ≈ 12, Linux'ta ≈ 460'tır.
- Algoritmik kodda akışı program, olay güdümlü kodda dış olaylar belirler. Olay döngüsü engelleyen bir `get_message` etrafında kurulur; uzun bir işleyici bütün arayüzü dondurur. UNIX'te her şey bir dosya, Windows'ta her şey bir nesnedir.
- Sistem yapısındaki temel ödünleşim yalıtım ile performans arasındadır. Monolitik çekirdek hızlıdır ama hata yalıtımı yoktur; mikro çekirdek hataları hapseder ama her istekte IPC öder. Katman sırası bağımlılıklardan çıkar. eBPF, doğrulayıcı sayesinde güvenli genişletme sağlar.
- Mekanizma bir işin nasıl yapılacağını, politika neyin yapılacağını belirler; sched_ext politikanın tamamını eBPF programı olarak yükler. Diklik ayrı kavramları bağımsız birleştirir: `clone` bayrakları `fork` ile `pthread_create` arasındaki her bileşimi verir.

- İki düzeyli adlandırma harici adı dahili sayıya eşler ve nesneyi adını değiştirmeden taşımayı sağlar. Önbelleksiz `/usr/ast/mbox` çözümlemesi 7 disk okuması ister. İşletim sistemleri genellikle geç bağlamayı seçer.
- Statik tablolar basittir ama dolunca büyüyemez; dinamik yapılar esnektir ama ayırıcı ve kilitleme ister. Gerçekleme aşağıdan yukarı, tasarım yukarıdan aşağı yapılır. Eş zamanlı iletişim kolay, eş zamansız iletişim ölçeklenebilirdir.
- Teknikler: donanıma bağımlı kodu az sayıda dosyada toplamak, dolaylılık, yeniden kullanım, yeniden girişlilik (paylaşılan `static` durum yok; `strtok_r`), kaba kuvvet ve önce hata denetimi.
- Hızlandırılacak kısmı ölçüm belirler; Amdahl yasası kazancın sınırını verir. Uzay-zaman ödünleşimi (bit sayma tablosu, renk paleti), önbellek (her zaman doğru), ipucu (yanlış olabilir, kullanmadan önce doğrulanır), yerellik ve yaygın durumu hızlandırma (futex) klasik tekniklerdir.
- Brooks yasası: gecikmiş projeye eleman eklemek onu daha da geciktirir; iletişim yolu sayısı $n(n-1)/2$ ile karesel artar. Kod yazmak işin yalnız altıda biridir ve gümüş kurşun yoktur.
- Çekirdek testi KUnit, kselftest ve LTP ile katmanlanır. KASAN, KCSAN, UBSAN, lockdep ve kmemleak sessiz hataları görünür kılar; syzkaller bunları kapsam güdümlü fuzzing ile tetikler. ftrace, perf ve eBPF gözlem, QEMU ve gdb adım adım hata ayıklama, `git bisect` ise en çok $\lceil \log_2 n \rceil$ testle regresyon avı sağlar.
- Eğilimler: sanallaştırma ve bulut, çok çekirdek, geniş adres alanları, enerji verimliliği, gömülü sistemler, Rust, eBPF ve unikernel'ler.

14.11 Kavram listesi

soyutlama *abstraction* · ilkel işlem *primitive* · yalıtım *isolation* · sistem çağrısı *system call* · basitlik, bütünlük, verimlilik *simplicity, completeness, efficiency* · olay güdümlü kod *event-driven code* · olay döngüsü *event loop* · monolitik çekirdek *monolithic kernel* · mikro çekirdek *microkernel* · katmanlı yapı *layered system* · exokernel *exokernel* · genişletilebilir sistem *extensible system* · eBPF *extended Berkeley Packet Filter* · çekirdek iş parçacığı *kernel thread* · politika / mekanizma *policy / mechanism* · diklik *orthogonality* · harici ad, dahili ad *external name, internal name* · bağlama zamanı *binding time* · koçan *stub* · eş zamanlı / eş zamansız iletişim *synchronous / asynchronous communication* · koşullu derleme *conditional compilation* · dolaylılık *indirection* · yeniden girişlilik *reentrancy* · kaba kuvvet *brute force* · uzay-zaman ödünleşimi *space-time trade-off* · önbellek *cache* · ipucu *hint* · yerellik *locality* · futex *fast userspace mutex* · Brooks yasası *Brooks's law* · gölge bellek *shadow memory* · fuzzing *fuzzing* · regresyon *regression* · unikernel *unikernel*

14.12 Kendini Dene

1. Olay döngüsündeki işleyiciler bir sayaç tutuyor: `KEY` 1, `CLICK` 10 ekliyor; `TIMER` sayaç 10'dan büyükse onu sıfırlıyor. Olaylar CLICK CLICK TIMER KEY KEY KEY TIMER QUIT sırasıyla gelirse döngü bitince sayaç kaçtır?

   Cevap

   3. Sayaç sırasıyla 10, 20, 0 (20 > 10), 1, 2, 3 olur. Son `TIMER` 'da 3 > 10 sağlanmadığı için sayaç değişmez. `QUIT` 'te `get_message` 0 döndürür ve döngü biter.

2. Yol önbelleğinde `/var` → 30 kaydı var, başka ilgili kayıt yok. `cozumle()` fonksiyonu `/var/log/syslog` için kaç disk okuması sayar? Önbellek tamamen boş olsaydı kaç sayardı? Dosyanın kendi i-node'u da okunursa toplamlar nedir?

   Cevap

   `/var` önbellekteyken `k` = 1'de durulur; `log` ve `syslog` diskten çözülür: 2 + 2 = 4 okuma. Önbellek boşken `k` 0'a iner ve üç bileşen kökten çözülür: 6 okuma. Dosyanın i-node'u da sayılırsa 5 ve 7 olur.

3. `yarim` tablosuyla `bit_say(0xE3)` hangi iki girdiyi okur ve ne döndürür?

   Cevap

   0xE3 = 1110 0011. Alt yarı 3, üst yarı E'dir. `yarim[3]` = 2 ve `yarim[E]` = 3; sonuç 5. Döngülü sürüm aynı sonuca sekiz turda ulaşır.

4. 1920 × 1080 bir görüntü (a) piksel başına 24 bit RGB ile, (b) piksel başına 4 bitlik indis ve 16 renklik palet (renk başına 3 bayt) ile saklanıyor. Boyutlar ve oran nedir?

   Cevap

   (a) 1920 × 1080 × 3 = 6.220.800 bayt. (b) İki piksel bir bayta sığar: 1920 × 1080 / 2 = 1.036.800 bayt, palet 16 × 3 = 48 bayt; toplam 1.036.848 bayt. Oran yaklaşık 6'dır. Bedeli, her piksel için tablo bakışı ve 16 renk sınırıdır.

5. Bir programın süresinin %40'ı `sikistir()` fonksiyonunda geçiyor. Bir seçenek bu fonksiyonu 4

kat, diğer seçenek programın geri kalanını 2 kat hızlandırmak. Hangisi daha çok kazandırır?

Cevap

İkisi de eşittir. Birincide süre 60 + 40/4 = 70, ikincide 40 + 60/2 = 70 olur; hızlanma iki durumda da 100 / 70 ≈ 1,43 kat. Amdahl yasasıyla: 1 / (0,6 + 0,1) = 1 / (0,4 + 0,3) ≈ 1,43. Hangi kısmın ne kadar hızlanabildiği, payı kadar önemlidir.

6. 12 kişilik ekip gecikmiş bir projeye 4 kişi daha ekliyor. İletişim yolu sayısı kaçtan kaça çıkar? Kişi sayısındaki artışla karşılaştırıldığında ne görülür?

Cevap

12 · 11 / 2 = 66 yoldan 16 · 15 / 2 = 120 yola çıkar: yaklaşık 1,82 kat. Kişi sayısı ise yalnız 16 / 12 ≈ 1,33 kat artmıştır. Yeni gelenlerin eğitimi de mevcut ekibin zamanından yer; Brooks yasasının iki nedeni budur.

7. `git bisect` için #0 iyi, #64 kötü; ilk kötü commit #45. Git her adımda ⌊(iyi + kötü) / 2⌋'yi sınatıyor. Hangi commit'ler sınanır? 5000 commit'lik bir aralıkta en çok kaç test gerekir?

Cevap

#32 iyi, #48 kötü, #40 iyi, #44 iyi, #46 kötü, #45 kötü: 6 test, $\lceil \log_2 64 \rceil = 6$. 5000 commit için $\lceil \log_2 5000 \rceil = 13$, çünkü $2^{12} = 4096 < 5000 \le 8192 = 2^{13}$.

8. `p = kmalloc(20)` 32 baytlık bir yuva veriyor. KASAN gölge baytları ne olur? 4 baytlık `p[8]`, `p[16]` ve `p[18]` okumalarından hangisi raporlanır?

Cevap

Gölge: 0, 0, 4, 0xfc. İlk iki bölge tamamen geçerli, üçüncü bölgenin ilk 20 − 16 = 4 baytı geçerli, dördüncü kırmızı bölgedir. `p[8]` : gölge 0, hızlı yol. `p[16]` : (16 & 7) + 4 − 1 = 3, 3 ≥ 4 değil, geçerli. `p[18]` : 2 + 4 − 1 = 5 ≥ 4, rapor: okuma 18–21. baytları kapsar ve 20 ile 21 nesnenin dışındadır.

9. Dört görev farklı zamanlarda çalışıyor: T1 önce A sonra B'yi, T2 önce C sonra D'yi, T3 önce B sonra C'yi, T4 önce D sonra A'yı alıyor. lockdep hangi anda uyarır? T4 önce A sonra D'yi alsaydı ne olurdu?

Cevap

Kenarlar sırasıyla A → B, C → D, B → C olur. T4 D'yi tutarken A'yı istediğinde A → B → C → D yolu vardır ve yeni D → A kenarı bir çevrim kapatır: uyarı T4'ün A'yı alışındadır. T4 önce A sonra D'yi alsaydı A → D kenarı eklenirdi; çevrim oluşmaz ve uyarı verilmezdi.

10. Aşağıdakilerden hangileri mekanizma, hangileri politikadır? (a) cpufreq sürücüsünün işlemci frekansını değiştirmesi, (b) `schedutil` yöneticisinin frekansı yüke göre seçmesi, (c) LRU yer değiştirme kuralı, (d) MMU'nun TLB ıskasında sayfa tablosunu yürümesi, (e) SELinux kuralları.

    Cevap

    Mekanizma: (a) ve (d); işin nasıl yapılacağını belirlerler ve hangi kararın verileceğinden bağımsızdırlar. Politika: (b), (c) ve (e); neyin yapılacağına karar verirler ve aynı mekanizma üzerinde değiştirilebilirler.

14.13 İleri okuma

- B. W. Lampson, "Hints for Computer System Design", ACM SOSP, 1983: arayüz, gerçekleme ve performans üzerine kısa öğütlerin klasik derlemesidir; "yaygın durumu hızlı yap" ve ipucu fikri buradan gelir.
- F. P. Brooks Jr., *The Mythical Man-Month*, 20. yıl baskısı, 1995: Brooks yasası, cerrah ekibi, ikinci sistem etkisi ve "No Silver Bullet" makalesi tek kitapta toplanmıştır.
- U. Drepper, "Futexes Are Tricky", 2011: futex tabanlı kilidin adım adım nasıl doğru yazılacağını, 0–1–2 durumlarının nedenini ve sık yapılan yanlışları anlatır.
- Linux çekirdek belgeleri, `docs.kernel.org/dev-tools/`: KASAN, KCSAN, UBSAN, kmemleak, KUnit ve kselftest'in nasıl açılıp kullanıldığını çekirdeğin kendi belgesinde verir.

14.14 Kaynaklar

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 12: Operating System Design.
2. B. W. Lampson, "Hints for Computer System Design", ACM SOSP, 1983 (ve 2020 güncellemesi "Hints and Principles for Computer System Design").
3. F. P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering*, 20. yıl baskısı, Addison-Wesley, 1995 — "No Silver Bullet" (1986) dahil.
4. J. H. Saltzer, M. F. Kaashoek, *Principles of Computer System Design: An Introduction*, Morgan Kaufmann, 2009.
5. D. R. Engler, M. F. Kaashoek, J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management", SOSP 1995.
6. G. Klein vd., "seL4: Formal Verification of an OS Kernel", SOSP 2009; J. N. Herder vd., MINIX 3 güvenilirlik çalışmaları.
7. Linux çekirdeği belgeleri: Rust (docs.kernel.org/rust), BPF ve sched_ext; H. Franke vd., "Fuss, Futexes and Furwocks", 2002.

KISIM IV · BÖLÜM 15

Linux

Dersin genel kavramları, yani süreçler, iş parçacıkları, çizelgeleme, sanal bellek, dosya sistemleri ve koruma, gerçek bir sistemde nasıl görünür? Bu not Linux'u bir örnek incelemesi olarak ele alır. Kavramları yeniden tanımlamaz; Linux'a özgü yapılara ve kararlara odaklanır. UNIX'ten Linux'a uzanan tarihle ve kabuğun boru hatlarını nasıl kurduğuyla başlar. Ardından çekirdeğin içine iner: `task_struct`, `clone()` bayrakları, sinyal teslimi, CFS'ten EEVDF'e adil çizelgeleme, dört ve beş düzeyli sayfa tabloları, buddy ve SLUB ayırıcıları, sayfa önbelleği ve OOM. G/Ç ve dosya sistemlerinde aygıt dosyalarını, blk-mq'yu, VFS'i ve ext4 günlüklemesini izler. Güvenlik kesimi izin hesaplarını, yetenekleri ve konteynerlerin temeli olan ad alanları ile cgroups'u anlatır. Not Android ile kapanır.

Kod (C): `github.com/sercankulcu/operating-systems-labs/Bolum15`

ÖĞRENME ÇIKTILARI

Tarih ve aile
UNIX'ten BSD, MINIX, GNU ve Linux'a uzanan çizgiyi; dağıtımları, sürüm modelini ve Android'i konumlandırmak.

Arayüzler ve kabuk
Kütüphane, sistem çağrısı ve çekirdek katmanlarını ayırmak; yönlendirmelerin ve boru hatlarının `pipe`, `fork`, `dup2` ve `exec` ile kurulduğunu izlemek.

Süreçler ve çizelgeleme
`fork` ve `execve`'nin çekirdekteki adımlarını, sinyal teslimini, `task_struct`'ı, `clone()` bayraklarını ve NPTL'i açıklamak; EEVDF'te uygunluk ve sanal son tarihi hesaplamak.

Bellek
Adres alanı düzenini ve VMA'ları, dört düzeyli sayfa tablosu yürüyüşünü, bölgeleri, buddy ve SLUB ayırıcılarını, sayfa önbelleğini ve OOM'u yorumlamak.

G/Ç ve dosyalar
Aygıt dosyalarını, blk-mq'yu ve soketleri tanımak; VFS nesnelerini, ext4'ün extent ve günlükleme düzenini, bağ sayısını ve `/proc`'u açıklamak.

Güvenlik
rwx, setuid ve umask hesaplamak; ilk eşleşen sınıf kuralını uygulamak; yetenekleri, ad alanlarını, cgroups'u, seccomp ve LSM'i ve Android korumalı alanını tanımak.

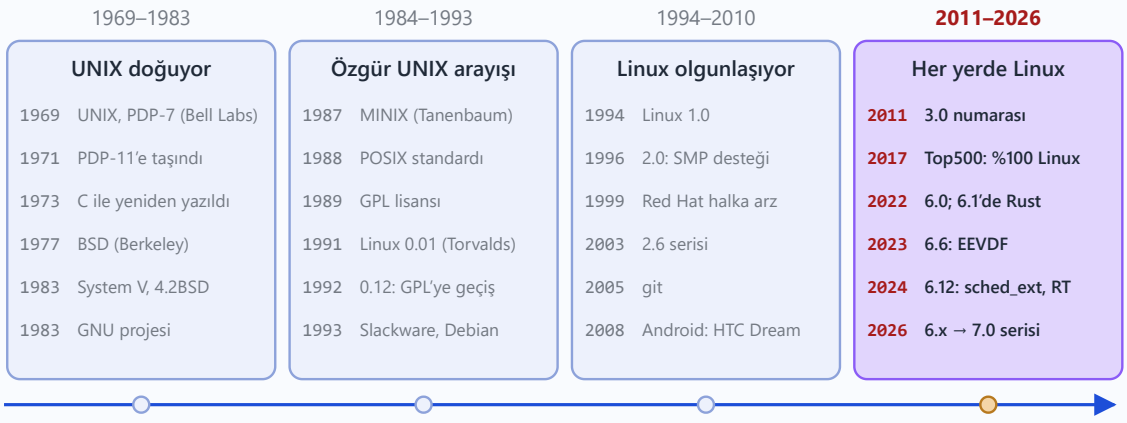
15.1 Linux'un kökleri

Ders boyunca süreç, iş parçacığı, çizelgeleme, sanal bellek, dosya sistemi ve koruma kavramları genel olarak anlatıldı. Bu son bölüm bir **örnek incelemesidir**: aynı kavramların yaygın bir gerçek sistemde nasıl gerçeklendiğine bakar. Seçilen sistem Linux'tur. Sunuculardan telefonlara, süper bilgisayarlardan gömülü aygıtlara kadar en çok çalıştırılan çekirdek odur. Kaynak kodu açıktır; her yapının adı ve alanı doğrudan okunabilir. Not bu yüzden kavramları yeniden tanımlamaz. Linux'a özgü ayrıntıya odaklanır: `task_struct`, `clone()` bayrakları, EEVDF çizelgeleyicisi, SLUB ayırıcısı, ad alanları ve cgroups.

15.1.1 UNIX'ten Linux'a

Linux'un hikâyesi UNIX'le başlar (Şekil 15.1). Ken Thompson ve Dennis Ritchie, Bell Laboratuvarları'nda 1969'da bir PDP-7 üzerinde küçük bir işletim sistemi yazdı. Sistem 1971'de PDP-11'e taşındı. 1973'te çekirdek C ile yeniden yazıldı. Bu adım o güne kadar görülmemiş bir şeydi: işletim sistemleri hep makine diliyle yazılırdı. C ile yazılan UNIX, yeni bir makineye derleyici taşınarak aktarılabildi. Taşınabilirlik UNIX'in yayılmasının asıl nedenidir.

AT&T bir antitröst kararı yüzünden bilgisayar satamıyordu. Kaynak kodu üniversitelere düşük ücretle verdi. Berkeley'deki California Üniversitesi bu kodu geliştirip **BSD** (*Berkeley Software Distribution*) adıyla dağıttı. Sanal bellek ve TCP/IP yığını UNIX'e BSD üzerinden girdi. AT&T'nin kendi çizgisi ise System V oldu. Seksenlerde her üretici kendi UNIX'ini çıkardı: SunOS, AIX, HP-UX, Xenix. Bu sürümler birbirinden ayrışınca bir programı hepsinde çalıştırmak zorlaştı. IEEE 1988'de **POSIX** standardını yayımladı. POSIX bir gerçekleme değildir; sistem çağrılarının ve kütüphane işlevlerinin ortak arayüzünü tanımlar.



Şekil 15.1: UNIX'ten Linux'a dört dönem: UNIX'in doğuşu, özgür UNIX arayışı, Linux'un olgunlaşması ve her yerde Linux.

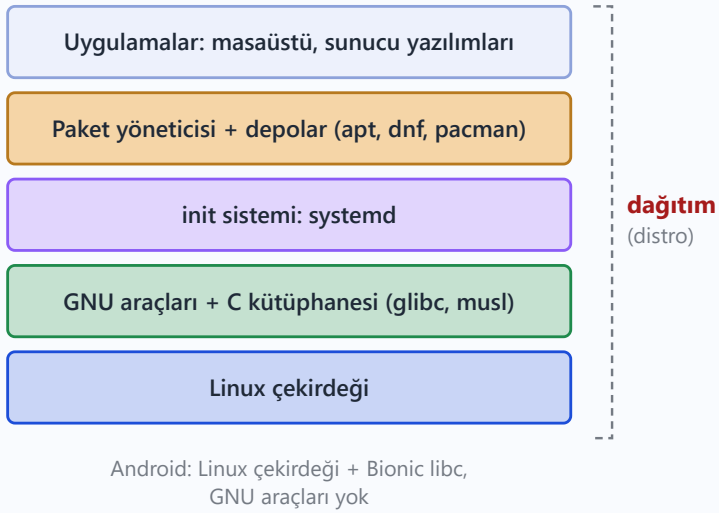
Seksenlerin ikinci yarısı özgür bir UNIX arayışıyla geçti. Richard Stallman 1983'te **GNU** projesini başlattı. Amaç tümüyle özgür, UNIX uyumlu bir sistemdi. Proje derleyiciyi (GCC), kabuğu (bash), C kütüphanesini ve yüzlerce aracı üretti, ama çekirdeği bitiremedi. Andrew Tanenbaum 1987'de öğretim için **MINIX**'i yazdı. MINIX küçük bir mikro çekirdekti ve kodu bir ders kitabıyla birlikte yayımlandı. Helsinki Üniversitesi öğrencisi Linus Torvalds 1991'de MINIX üzerinde kendi çekirdeğini geliştirmeye başladı. Linux 0.01 aynı yıl yayımlandı. Torvalds çekirdeği 1992'de, 0.12 sürümüyle GPL lisansına geçirdi. GPL, değiştirilmiş kodun da aynı özgürlükle dağıtılmasını şart koşar. GNU'nun araçları ile Linux çekirdeği birleşince kullanılabilir, tümüyle özgür bir sistem ortaya çıktı. Slackware ve Debian gibi ilk dağıtımlar 1993'te doğdu.

Linux UNIX kodu içermez. POSIX arayüzünü baştan yazılmış kodla gerçekler. Tasarımı ise MINIX'in tersidir. Torvalds mikro çekirdek yerine **monolitik** bir çekirdek seçti. Bu seçim 1992'de Tanenbaum ile Torvalds arasında ünlü bir e-posta tartışmasına yol açtı. Tanenbaum monolitik tasarımı eskimiş buluyordu. Torvalds ise basitlik ve hızı savunuyordu. Bugünkü Linux monolitiktir, ama modüler bir yapıyla esneklik kazanmıştır.

Sonraki on beş yılda çekirdek olgunlaştı. Linux 1.0 1994'te çıktı. 2.0 sürümü (1996) çok işlemcili makineleri, yani simetrik çok işlemeyi (SMP) destekledi. 2.6 serisi (2003) kesilebilir çekirdeği ve yeni çizelgeleyicileri getirdi. 2005'te çekirdek geliştiricilerinin kullandığı ticari sürüm denetim aracı BitKeeper'ın lisansı sorun çıkardı. Torvalds birkaç haftada `git`'i yazdı; bugün yazılım dünyasının çoğu onu kullanır. İlk Android telefonu, HTC Dream, 2008'de satışa çıktı. 2011'deki 3.0 sürümünden bu yana numaralar teknik bir anlam taşımaz; yalnızca büyür. 6.19'un ardından gelen sürüm, Nisan 2026'da yayımlanan 7.0'dır. Kasım 2017'den bu yana dünyanın en hızlı 500 süper bilgisayarının tamamı Linux çalıştırır. 6.x serisi Rust desteğini (6.1), EEVDF çizelgeleyicisini (6.6) ve genişletilebilir çizelgeleyici `sched_ext` ile gerçek zaman yamasını (6.12) getirdi.

15.1.2 GNU, dağıtımlar ve sürüm modeli

"Linux" sözcüğü, kesin anlamıyla yalnızca çekirdeğin adıdır. Kullanıcının bilgisayarına kurduğu şey bir **dağıtımdır** (Şekil 15.2). Dağıtım, çekirdeğin üstüne katmanlar ekler: GNU araçları ve C kütüphanesi (çoğunlukla glibc), sistemi başlatan init programı (bugün çoğunlukla systemd), paket yöneticisi ve uygulamalar. Bu yüzden tam sistem için "GNU/Linux" adı da kullanılır. Android aynı çekirdeği kullanır ama GNU araçlarını kullanmaz; C kütüphanesi Bionic'tir.



Şekil 15.2: Bir dağıtımın katmanları: çekirdek, GNU araçları ve C kütüphanesi, init sistemi, paket yöneticisi ve uygulamalar.

Dağıtımlar paket biçimine göre ailelere ayrılır (Tablo 15.1). Aynı ailenin üyeleri aynı paket yöneticisini kullanır. Türkiye'de geliştirilen Pardus Debian ailesindendir. Alpine ise glibc yerine küçük musl kütüphanesini ve tek bir ikili dosyada toplanmış BusyBox araçlarını kullanır; bu yüzden konteyner imajlarında sevilir.

Aile	Örnekler	Paket biçimi / araç
Debian	Debian, Ubuntu, Mint, Pardus	.deb / apt
Red Hat	Fedora, RHEL, Rocky, Alma	.rpm / dnf
Arch	Arch, Manjaro	pacman
SUSE	openSUSE, SLES	zypper
Hafif	Alpine (musl + BusyBox)	apk

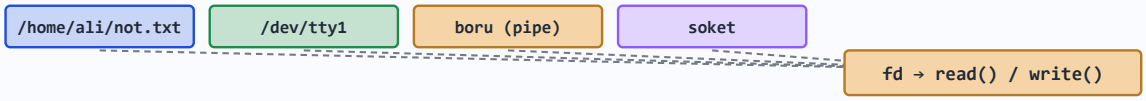
Tablo 15.1: Başlıca dağıtım aileleri.

Çekirdeğin kendi sürüm modeli düzenlidir. Bir sürüm yayımlanınca iki haftalık **birleştirme penceresi** açılır. Bu sürede alt sistem sorumlularının biriktirdiği yeni özellikler ana ağaca alınır. Ardından haftalık sürüm adayları gelir: rc1, rc2 … çoğunlukla rc7. Bu haftalarda yalnız hata düzeltmesi kabul edilir.

Böylece yaklaşık 9–10 haftada bir yeni ana sürüm çıkar. Her yılın son ana sürümü uzun süreli destek (*long-term support*, LTS) alır: 6.6 (2023), 6.12 (2024) ve 6.18 (2025) böyledir. Dağıtımlar çoğunlukla bir LTS çekirdeği seçer ve yıllarca onun güvenlik yamalarını izler.

15.1.3 Tasarım ilkeleri

UNIX'i programcılar kendileri için tasarladı. Bu kökenden gelen ilkeler Linux'ta da sürer. Sistem **basittir**: az sayıda, iyi tanımlı kavram üzerine kuruludur; süreç, dosya ve boru. **Tutarlıdır**: disk dosyası, terminal, boru ve ağ soketi aynı `read` ve `write` çağrılarıyla kullanılır (Şekil 15.3). "Her şey dosyadır" sözü bu tutarlılığı anlatır. Sistem **esnektir**: araçlar birbirini tanımadan, metin akışları üzerinden birleşir. Programcıya az "sihir" gösterir; başarılı bir komut çoğunlukla hiçbir şey yazmaz.



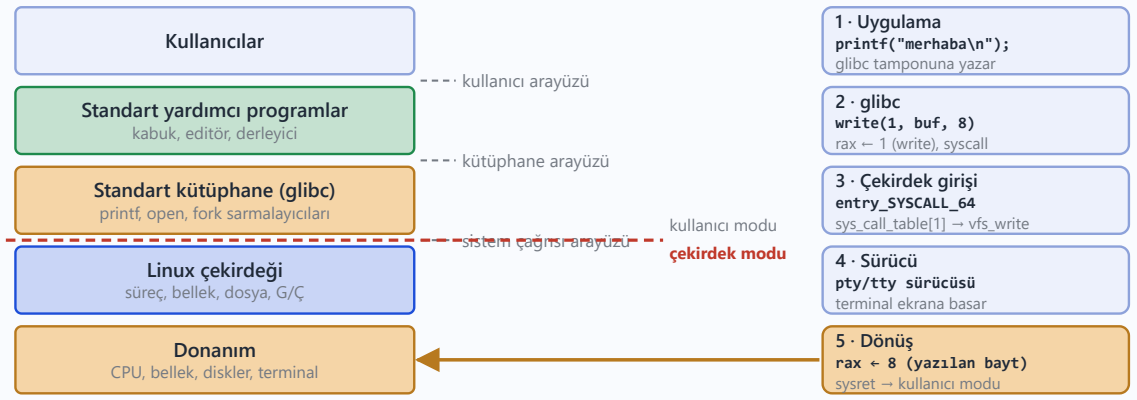
Şekil 15.3: Her şey dosyadır: disk dosyası, terminal, boru ve soket aynı dosya tanımlayıcısı arayüzüyle kullanılır.

Boru hatlarını UNIX'e kazandıran Doug McIlroy bu düşünceyi üç kurala bağladı. Her program tek bir işi iyi yapmalıdır. Programlar birlikte çalışacak biçimde yazılmalıdır. Programlar metin akışlarını işlemelidir, çünkü metin evrensel arayüzdür. POSIX bu arayüzü standartlaştırır. Aynı kaynak kod Linux'ta, BSD'de ve macOS'ta derlenip çalışabilir.

15.2 Arayüzler ve kabuk

15.2.1 Üç arayüz: bir `printf`'in yolculuğu

Bir Linux sistemi katmanlar hâlinde düşünülebilir (Şekil 15.4). En altta donanım, onun üstünde çekirdek vardır. Çekirdeğin üstünde standart C kütüphanesi, onun üstünde kabuk, düzenleyici, derleyici gibi standart yardımcı programlar durur. En üstte kullanıcılar bulunur. Katmanlar arasında üç arayüz vardır. **Sistem çağrısı arayüzü** kütüphane ile çekirdek arasındadır. **Kütüphane arayüzü** programların çağırdığı `printf`, `open`, `fork` gibi işlevlerdir. **Kullanıcı arayüzü** ise kabuk ve programların kendisidir. POSIX kütüphane arayüzünü tanımlar, sistem çağrılarını tanımlamaz. Hangi işlevin sistem çağrısı olduğu sisteme göre değişebilir.



Şekil 15.4: UNIX/Linux katmanları ve `printf("merhaba\n")` çağrısının beş adımlık yolculuğu.

Bu katmanları tek bir satırın izini sürerek görmek öğreticidir. Program `printf("merhaba\n")` çağırsın.

1. **Uygulama.** `printf` bir sistem çağrısı değildir. glibc biçimlendirilmiş metni kullanıcı alanındaki bir tampona yazar. Çıktı terminale bağlıysa tampon satır sonunda boşaltılır.
2. **glibc.** Tampon boşaltılırken glibc'nin `write(1, buf, 8)` sarmalayıcısı çalışır. x86-64'te sistem çağrısı numarası `rax` yazmacına konur; `write`'ın numarası 1'dir. Argümanlar `rdi`, `rsi`, `rdx` yazmaçlarına yerleşir. Sonra `syscall` komutu yürütülür.
3. **Çekirdek girişi.** CPU çekirdek moduna geçer ve `entry_SYSCALL_64` giriş koduna atlar. Giriş kodu numarayı `sys_call_table` dizisinde indis olarak kullanır ve `write`'ın işleyicisini çağırır. İşleyici VFS katmanına geçer: dosya tanımlayıcısı 1'in arkasındaki nesneyi bulur.
4. **Sürücü.** Etkileşimli bir oturumda tanımlayıcı 1 bir sözde terminale (pty) bağlıdır. tty katmanı baytları terminal öykünücüsüne iletir. Çıktı bir diske yönlendirilmiş olsaydı yol sayfa önbelleğine ve blok katmanına giderdi.
5. **Dönüş.** Yazılan bayt sayısı (8) `rax`'ta döner. `sysret` komutu kullanıcı moduna geri geçer. Hata olsaydı çekirdek negatif bir hata kodu döndürürdü; glibc bunu `errno`'ya yazar ve çağırana −1 verir.

Linux x86-64'te 460 dolayında sistem çağrısı vardır. Hepsi aynı kapıdan girer: numara `rax`'ta, argümanlar yazmaçlarda, giriş `syscall` ile. Bir programın hangi çağrıları yaptığı `strace` ile izlenebilir.

> **Sık yapılan hata**
>
> Her kütüphane işlevini bir sistem çağrısı sanmak. `printf`, `malloc`, `opendir` ve `readdir` kütüphane işlevleridir. Kimi hiç çağrı yapmaz, kimi arada bir alttaki çağrıya (`write`, `brk` / `mmap`, `getdents64`) başvurur. Bir döngüde 1000 kez `printf` çağırmak, tampon sayesinde yalnızca birkaç `write` çağrısı üretebilir.

15.2.2 Komutlar ve yönlendirme

Linux'ta komutların çoğu küçük, tek işlik programlardır (Tablo 15.2). Filtre adı verilen programlar standart girdiden okur, standart çıktıya yazar. Bu yüzden boru hattında birbirine eklenebilirler. Bazı adlar yanıltıcıdır. `touch` dosyanın zaman damgasını günceller; dosya yoksa boş bir dosya oluşturur. `sync` çekirdekteki kirli tamponları diske yazar; iki makine arasında eşitleme yapan araç `rsync`'tir. `/proc` bir komut değil, sözde bir dosya sistemidir: `cat /proc/meminfo` çekirdeğin bellek sayaçlarını okur. Ağ araçları da yenilendi: `ip addr`, `ip route`, `ss` ve `nft`, eski `ifconfig`, `route`, `netstat` ve `iptables`'ın yerini aldı.

Kategori	Komutlar
Dosya / dizin	`ls cd pwd mkdir rm -r cp mv touch find tar`
Metin (filtre)	`cat less head tail grep sort uniq wc echo`
Süreç / sistem	`ps top kill chmod chown df du sync`
Ağ	`ssh scp ping curl wget traceroute ip ss`
Bilgi	`man uname free lsblk`

Tablo 15.2: Sık kullanılan komutlar.

Her süreç doğduğunda üç açık dosya tanımlayıcısı bulur: 0 standart girdi, 1 standart çıktı, 2 standart hata. Kabuk bu tanımlayıcıları programı çalıştırmadan önce başka dosyalara yönlendirebilir:

```
# yönlendirme: 0 = stdin, 1 = stdout, 2 = stderr
sort < girdi.txt > cikti.txt 2> hatalar.log   # < okur, > yazar (üzerine)
echo "satır" >> gunluk.txt                    # >> sona ekler
make 2>&1 | tee derleme.log &                 # stderr→stdout, boru, arka plan
```

`sort` satırında kabuk `sort`'un tanımlayıcı tablosunu kurar (Şekil 15.5). 0'ı `girdi.txt`'ye, 1'i `cikti.txt`'ye, 2'yi `hatalar.log`'a bağlar. Sonra `sort`'u çalıştırır. `sort` yönlendirmeden habersizdir; yalnız 0'dan okur, 1'e yazar. `>` dosyanın üzerine yazar, `>>` sonuna ekler. `2>&1` "2'yi, 1'in şu anki hedefine kopyala" demektir. Sondaki `&` komutu arka planda başlatır.



Şekil 15.5: `sort < girdi.txt > cikti.txt 2> hatalar.log` için kabuğun kurduğu tanımlayıcı tablosu.

Yönlendirmeler soldan sağa uygulanır. Kabuk her birini bir `dup2` çağrısına çevirir. Sıra bu yüzden sonucu değiştirir.

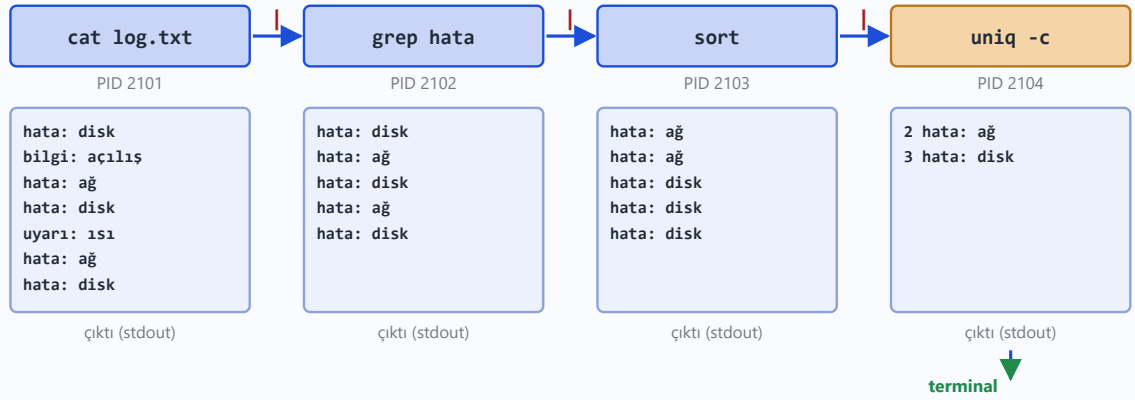
Örnek 15.1

`/yok` dizini yoktur. `ls /yok /tmp > c.txt 2>&1` ile `ls /yok /tmp 2>&1 > c.txt` komutlarında hata iletisi nereye gider?

1. Birinci komut. `> c.txt` : 1 → c.txt. `2>&1` : 2, 1'in o anki hedefine, yani c.txt'ye kopyalanır. Hem liste hem hata dosyaya gider.
2. İkinci komut. `2>&1` : 1 o anda terminaldir; 2 de terminale bağlanır. `> c.txt` : yalnız 1 değişir. Liste dosyaya, hata ekrana gider.
3. `2>&1` bir "bağ" kurmaz, o anki hedefi kopyalar. 1 sonradan değişse de 2 eski hedefte kalır.

15.2.3 Boru hattı

Boru hattı (*pipeline*), bir programın çıktısını bir sonrakinin girdisine bağlayan komut zinciridir. `cat log.txt | grep hata | sort | uniq -c` satırını ele alalım (Şekil 15.6). Kabuk satırı ayrıştırır, 3 boru ve 4 süreç oluşturur. Dört süreç aynı anda çalışır. Her biri yalnız kendi standart girdisini ve çıktısını bilir.



Şekil 15.6: `cat log.txt | grep hata | sort | uniq -c` : yedi satır, beş eşleşme, sıralı beş satır ve iki sayım.

`cat` dosyanın 7 satırını boruya yazar. Bir borunun çekirdekteki tamponu sınırlıdır; Linux'ta varsayılan 64 KiB'tır. Tampon dolarsa yazan süreç bloke olur, okuyan süreç boşalttıkça devam eder. Bu, üretici–tüketici sorununun çekirdekte hazır bir çözümüdür. `grep` satır satır okur ve eşleşen 5 satırı hemen iletir. Filtre durumsuzdur, akışı bekletmez. `sort` ise farklıdır. Girdinin sonunu (EOF) görmeden tek satır bile yazamaz, çünkü son satır sıralamada ilk sıraya geçebilir. `uniq -c` yalnızca *ardışık* tekrarları sayar. Bu yüzden önüne `sort` gerekir. Sonuç iki satırdır: 2 × "hata: ağ" ve 3 × "hata: disk".

Sık yapılan hata

`uniq`'in bütün tekrarları birleştirdiğini sanmak. Sıralanmamış beş satırda (disk, ağ, disk, ağ, disk) hiçbir satır kendinden öncekinin aynısı değildir. `uniq -c` beş ayrı satır, her birinin önünde 1 yazar. Python benzetimi de aynı sonucu verir.

15.2.4 Kod izleme: kabuk bir boru hattını nasıl kurar?

Kabuk `ls | wc -l` satırını dört sistem çağrısıyla kurar: `pipe`, `fork`, `dup2` ve `exec`. Bu işi yapan kabuğun basitleştirilmiş hâli şöyledir:

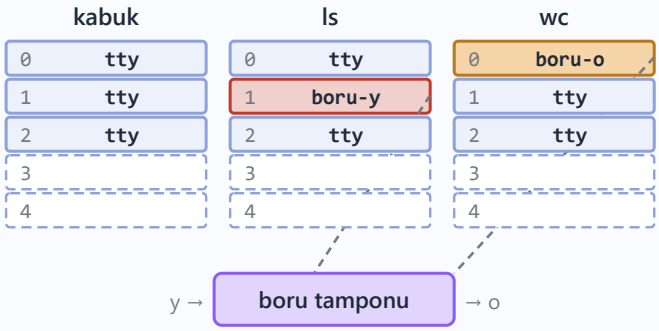
```
int fd[2];
pipe(fd);              /* 3: okuma, 4: yazma */
if (fork() == 0) {     /* çocuk 1 */
    dup2(fd[1], 1);
    close(fd[0]); close(fd[1]);
    execlp("ls", "ls", NULL);
}
if (fork() == 0) {     /* çocuk 2 */
    dup2(fd[0], 0);
    close(fd[0]); close(fd[1]);
    execlp("wc", "wc", "-l", NULL);
}
/* kabuk: iki ucu da kapat */
close(fd[0]); close(fd[1]);
wait(NULL); wait(NULL);
```

Kodun ilk satırı iki elemanlı bir dizi ayırır. `pipe(fd)` çekirdekte bir tampon ve iki uç oluşturur. Çekirdek her zaman en küçük boş tanımlayıcı numarasını verir. 0, 1 ve 2 dolu olduğu için okuma ucu 3, yazma ucu 4 olur. Birinci `fork` çocuk süreci doğurur. Çocuk, ebeveynin tanımlayıcı tablosunun bir kopyasını alır; iki süreç aynı boruyu gösterir. Çocuk `dup2(fd[1], 1)` ile 1 numaralı tanımlayıcıyı borunun yazma ucuna yöneltir. Artık gereksiz olan 3 ve 4'ü kapatır. `execlp` sürecin görüntüsünü `ls` ile değiştirir. Tanımlayıcı tablosu `exec`'ten sonra da korunur. `ls` hiçbir şeyden habersiz 1'e yazar ve veri boruya akar. İkinci çocuk aynı işi okuma ucu için yapar: `dup2(fd[0], 0)`, iki ucu kapatır ve `wc -l` olur. Son olarak kabuk iki ucu da kapatır ve iki çocuğu bekler.

Adım	Kabuk 0 1 2 3 4	Çocuk 1	Çocuk 2
`pipe(fd)`	tty tty tty o y	—	—
1. `fork`	tty tty tty o y	tty tty tty o y	—
`dup2(4, 1)` + iki `close`	tty tty tty o y	tty **y** tty – –	—
`execlp("ls")`	aynı	tty y tty – – (artık `ls`)	—
2. `fork`, `dup2(3, 0)`, `exec("wc")`	tty tty tty o y	aynı	**o** tty tty – –

Adım	Kabuk 0 1 2 3 4	Çocuk 1	Çocuk 2
Kabuk iki ucu kapatır	tty tty tty – –	aynı	aynı

Tablo 15.3: `ls | wc -l` kurulurken tanımlayıcı tabloları (tty: terminal; y: borunun yazma ucu; o: okuma ucu).



Şekil 15.7: Son durum: borunun yazma ucu yalnız `ls`'te, okuma ucu yalnız `wc`'de açık.

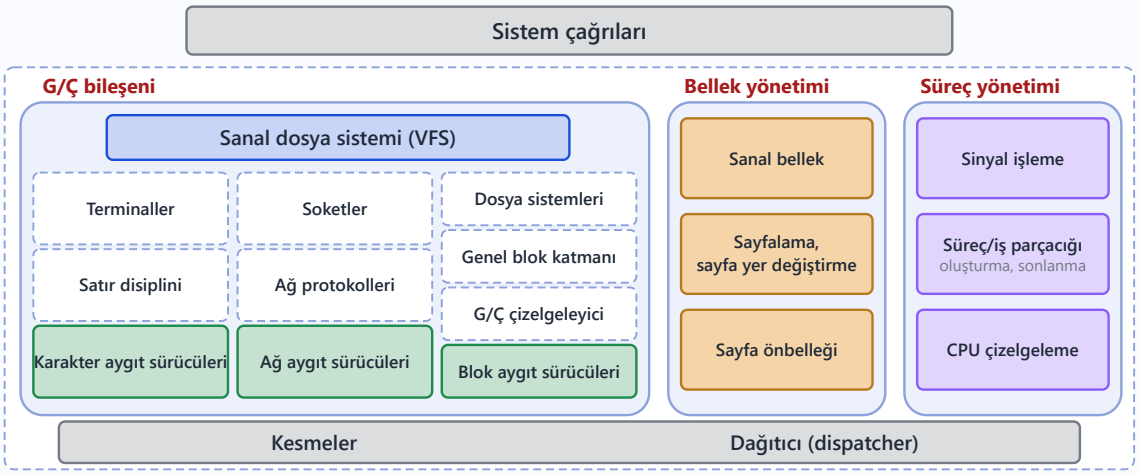
Sondan bir önceki satırdaki iki `close` çağrısı önemsiz görünür ama sistemin doğru çalışması onlara bağlıdır. Bir borunun okuyucusu ancak *bütün* yazma uçları kapanınca EOF görür. Kabuk 4'ü kapatmasaydı yazma ucu açık kalırdı. `ls` bitip kendi ucunu kapatsa bile `wc` daha veri gelebileceğini varsayıp sonsuza dek beklerdi. Kabuk da `wc`'yi beklediği için ikisi birlikte asılı kalırdı. Ters durumda, bütün okuma uçları kapanmışken boruya yazan süreç `SIGPIPE` sinyali alır ve varsayılan olarak sonlanır.

> **Sık yapılan hata**
>
> Çocuklarda fazla uçları kapatmayı unutmak. `wc`'yi çalıştıran çocuk 4'ü kapatmazsa, kendi elinde tuttuğu yazma ucu yüzünden kendi EOF'unu hiç göremez. Kural kısadır: her süreç kullanmadığı her boru ucunu kapatır.

15.3 Çekirdeğin yapısı

15.3.1 Monolitik çekirdeğin bileşenleri

Linux çekirdeği üç ana bileşene ayrılır (Şekil 15.8). **G/Ç bileşeni** terminalleri, ağı ve dosya sistemlerini yönetir. **Bellek yönetimi** sanal belleği, sayfalamayı ve sayfa önbelleğini yürütür. **Süreç yönetimi** süreçleri ve iş parçacıklarını oluşturur, sinyalleri işler ve CPU'yu çizelgeler. Hepsinin üstünde sistem çağrısı arayüzü, altında kesmeler ve dağıtıcı (*dispatcher*) bulunur. Dağıtıcı bağlam değiştirmeyi yapan koddur. Klasik 4.4BSD çekirdeği de aynı üç bileşene ayrılırdı.



Şekil 15.8: Linux çekirdeğinin yapısı: G/Ç, bellek ve süreç yönetimi; üstte sistem çağrıları, altta kesmeler ve dağıtıcı.

G/Ç bileşeni kendi içinde üç sütuna ayrılır. Karakter aygıtları terminaller ve satır disiplininden geçer. Ağ aygıtları soket katmanından ve protokollerden geçer. Blok aygıtları dosya sistemlerinden, genel blok katmanından ve G/Ç çizelgeleyicisinden geçer. Üç sütun tepede **sanal dosya sisteminde** (VFS) buluşur. Şekildeki kutuların hepsi tek bir adres alanında, çekirdek modunda çalışır. Bir bileşen ötekinin işlevini doğrudan çağırabilir; mesaj geçirmeye ya da bağlam değiştirmeye gerek yoktur. Linux'u **monolitik** yapan budur. Bedeli de açıktır: herhangi bir bileşendeki hata, örneğin bir sürücüdeki geçersiz işaretçi, bütün çekirdeği çökertebilir.

15.3.2 Yüklenebilir çekirdek modülleri

Monolitik bir çekirdek, akla gelebilecek her aygıtın sürücüsünü baştan içerseydi çok büyük olurdu. Linux bu sorunu modüllerle çözer.

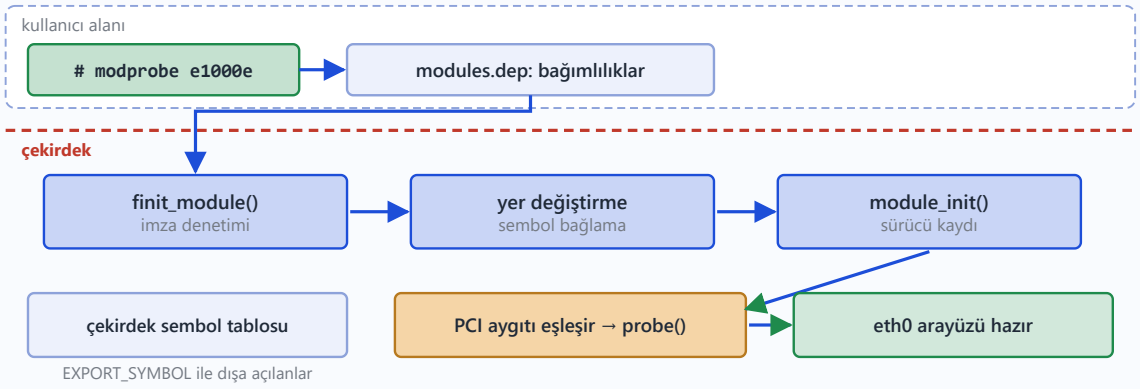
Tanım 15.1

Yüklenebilir çekirdek modülü (*loadable kernel module*, LKM), sistem çalışırken çekirdeğe eklenip çıkarılabilen, `.ko` uzantılı nesne dosyasıdır. Yüklenen modül çekirdeğin adres alanında, çekirdek ayrıcalığıyla çalışır.

Sürücülerin çoğu, dosya sistemlerinin ve ağ protokollerinin bir kısmı modüldür. Bir modülün yüklenmesi altı adımda izlenebilir (Şekil 15.9). Örnek olarak Intel ağ kartı sürücüsü `e1000e` yüklensin.

1. **İstek.** Yönetici `modprobe e1000e` yazar. Daha sık görülen yol otomatik yüklemedir: aygıt sisteme takılınca çekirdek bir olay üretir, kullanıcı alanındaki `udev` aygıtın kimliğine uyan modülü bulup yükler.

2. **Bağımlılıklar.** `modprobe`, `/lib/modules/$(uname -r)/modules.dep` dosyasından modülün hangi modüllere ihtiyaç duyduğunu okur ve önce onları yükler. Daha ilkel `insmod` bunu yapmaz; verilen tek dosyayı yüklemeye çalışır.
3. **Kopyalama ve imza.** `finit_module()` sistem çağrısı modül dosyasını çekirdek belleğine kopyalar. Çağıranın `CAP_SYS_MODULE` yeteneği olmalıdır. Secure Boot açıkken modülün imzası doğrulanır; imzasız modül reddedilir.
4. **Bağlama.** Çekirdek modülün yer değiştirmelerini yapar. Modülün kullandığı her dış sembolü çekirdeğin sembol tablosunda arar. Yalnız `EXPORT_SYMBOL` ile dışa açılmış çekirdek işlevleri modüle görünür.
5. **Başlatma.** Modülün `module_init` ile işaretlenmiş işlevi çalışır. Sürücü kendini ilgili alt sisteme kaydeder; bir PCI kartı için bu `pci_register_driver` çağrısıdır.
6. **Eşleşme.** PCI alt sistemi sürücünün desteklediği kimliklerle sistemdeki aygıtları karşılaştırır. Eşleşen her aygıt için sürücünün `probe()` işlevi çağrılır ve `eth0` benzeri bir ağ arayüzü oluşur.



Şekil 15.9: Bir modülün yüklenmesi: `modprobe`, `finit_module()`, imza denetimi, yer değiştirme ve sembol bağlama, `module_init()` ve `probe()`.

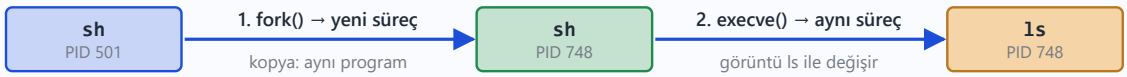
Yüklü modüller `lsmod` ile listelenir, `rmmod` ile kaldırılır. Modül sistemi çekirdeği esnek kılar ama koruma getirmez. Modül, çekirdeğin kendisi kadar ayrıcalıklıdır. Bu yüzden yükleme yetkisi sıkı denetlenir.

Çekirdeğin kaynak ağacı 2025 başında 40 milyon satırı geçti. Bu satırların büyük kısmı sürücülerdir. Son yıllarda iki yenilik çekirdeğin güvenliğini artırmaya yöneldi. Birincisi Rust'tır. Rust desteği 6.1'de çekirdeğe girdi; ilk Rust sürücüsü, bir ağ PHY sürücüsü, 6.8'de eklendi. Rust'ın tür sistemi, C sürücülerinde sık görülen bellek hatalarının bir bölümünü derleme anında yakalar. İkincisi **eBPF**'tir. eBPF, kullanıcının yazdığı küçük programları çekirdekte güvenle çalıştırmaya yarar. Program yüklenmeden önce bir doğrulayıcıdan geçer. Doğrulayıcı programın sonlanacağını ve yalnız izin verilen belleğe eriştiğini kanıtlamadan yüklemeye izin vermez. eBPF ile ağ paketleri süzülür, çekirdek olayları izlenir; bir modül yazmadan çekirdeğin davranışı genişletilir.

15.4 Süreçler ve iş parçacıkları

15.4.1 `fork` ve `execve` çekirdekte

Linux'ta yeni bir program iki adımda başlar (Şekil 15.10). Kabuk (PID 501) önce `fork()` ile kendini kopyalar; kopya PID 748 alır ve yine kabuk kodunu çalıştırır. Kopya sonra `execve()` ile kendi görüntüsünü `ls` ile değiştirir. PID değişmez; değişen, sürecin çalıştırdığı programdır.



Şekil 15.10: `fork()` yeni bir süreç (aynı program), `execve()` aynı süreçte yeni bir program üretir.

`fork()` çekirdekte şu işleri yapar:

1. Yeni bir `task_struct` ayırır, ebeveyninkinden doldurur ve yeni bir PID atar.
2. Çocuğa ayrı bir çekirdek yığını ayırır. Ebeveynin kaydedilmiş yazmaçlarını kopyalar; yalnız dönüş değeri yazmacını çocukta 0 yapar. `fork`'un "iki kez dönmesi" buradan gelir.
3. Sayfa tablolarını kopyalar ve yazılabilir sayfaları iki tarafta da salt okunur işaretler. Bu **yazınca kopyalamadır** (COW): sayfa ancak biri ona yazmaya kalkınca kopyalanır.
4. Açık dosyaları paylaştırır. Çocuğun tanımlayıcı tablosu ebeveyninkinin kopyasıdır; her girdi aynı `struct file` nesnesini gösterir ve nesnenin referans sayısı artar.
5. Sinyal işleyicilerini, çalışma dizinini ve kimlik bilgilerini kopyalar.

`execve()` ise eski görüntüyü atar. Önce dosyanın ELF başlığını okur. Sürecin eski adres alanını tanımlayan `mm_struct`'ı bırakır ve yenisini kurar. Programın kod ve veri kesimlerini dosyadan `mmap` ile eşler; hiçbir sayfayı hemen okumaz. Dosyanın setuid ya da setgid biti açıksa sürecin yeni kimlik bilgilerini (`cred`) hazırlar. Son olarak program sayacını ayarlar. Dinamik bağlanmış bir programda ilk çalışan kod programın kendisi değil, dinamik bağlayıcı `ld.so`'dur; o, paylaşımlı kütüphaneleri eşleyip programın giriş noktasına atlar. Başarılı bir `execve` asla geri dönmez, çünkü onu çağıran kod artık yoktur. Başarısız olursa −1 döner. Bu durumda çocuk geleneksel olarak `_exit(127)` ile çıkar; kabuğun "komut bulunamadı" çıkış kodu 127 buradan gelir.

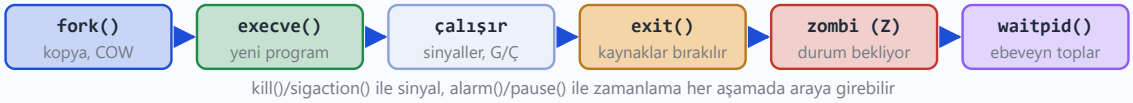
Eski UNIX'ler `fork`'ta sürecin veri ve yığın kesimlerini gerçekten kopyalardı. Linux yalnız sayfa tablolarını kopyalar. `fork`'ların çoğunu hemen bir `exec` izlediği için kopyalanan sayfa sayısı çok azdır. Kabuğun adres alanı binlerce sayfa tutsa bile `fork` mikrosaniyeler sürer.

15.4.2 Süreç yönetimi çağrıları

Bir sürecin yaşamı birkaç sistem çağrısıyla yönetilir (Tablo 15.4). `clone` ve `clone3` genel yaratma çağrılarıdır; `fork` onların özel bir hâlidir. `exit` ile `_exit` arasındaki fark küçük ama önemlidir: `exit` bir kütüphane işlevidir, stdio tamponlarını boşaltır ve `atexit` işlevlerini çağırır; `_exit` doğrudan çekirdeğe gider.

Çağrı	İşlev
`fork()`	Süreci kopyalayarak yeni süreç oluşturur (COW)
`clone()`, `clone3()`	Neyin paylaşılacağı bayraklarla seçilen yeni görev (iş parçacığı, konteyner)
`execve()`	Süreç görüntüsünü yeni programla değiştirir
`waitpid()`	Çocuğun bitmesini bekler, çıkış durumunu alır, zombiyi toplar
`exit()` / `_exit()`	Süreci sonlandırır (`exit` stdio tamponlarını boşaltır)
`kill()`	Sürece sinyal gönderir; sinyal sonlandırma olmak zorunda değil
`sigaction()`, `sigprocmask()`	İşleyici atar; engellenen sinyal kümesini okur ve değiştirir
`sigpending()`, `sigsuspend()`	Bekleyen sinyalleri döndürür; maskeyi geçici değiştirip sinyal bekler
`rt_sigreturn()`	İşleyiciden dönüşte kaydedilmiş bağlamı geri yükler
`alarm()` / `pause()`	n saniye sonra `SIGALRM` üretir / bir sinyal gelene kadar uyur

Tablo 15.4: Süreç yönetimi sistem çağrıları.

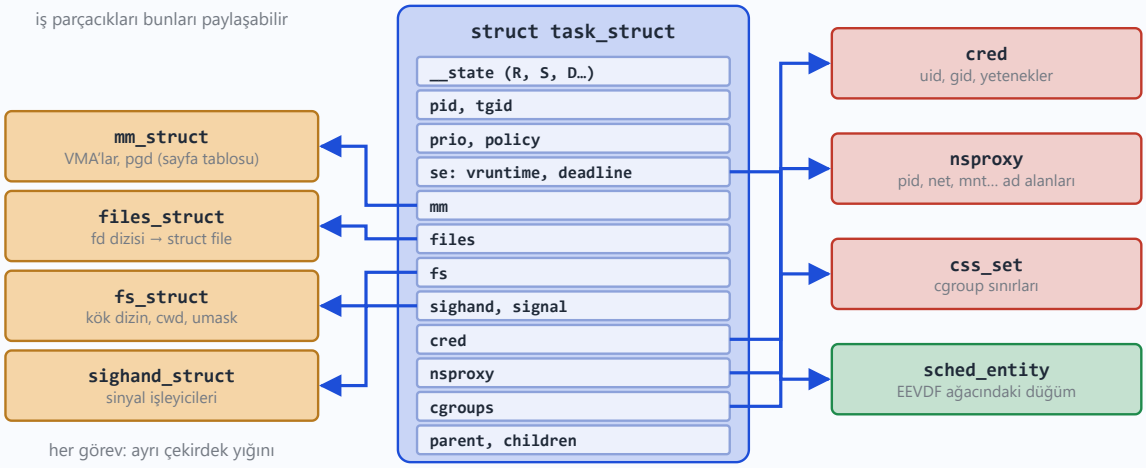


Şekil 15.11: Bir sürecin yaşamı: `fork`, `execve`, çalışma, `exit`, zombi ve `waitpid` ile toplanma.

Biten bir süreç hemen yok olmaz (Şekil 15.11). Belleğini ve dosyalarını bırakır, ama `task_struct`'ı çıkış durumunu taşıyarak kalır. Bu hâldeki sürece **zombi** denir; `ps` onu Z harfiyle gösterir. Ebeveyn `waitpid` ile durumu okuyunca zombi tamamen silinir. Ebeveyn önce ölürse çocuklar PID 1'e (ya da kendini "alt süreç toplayıcı" ilan etmiş bir sürece) bağlanır; o da onları toplar. Süreç ağacı `pstree -p` ya da `ps -ef --forest` ile görülür.

15.4.3 Süreç tanımlayıcısı: `task_struct`

Ders kitaplarındaki süreç kontrol bloğunun (PCB) Linux'taki karşılığı `struct task_struct`'tır (Şekil 15.12). Yapı büyüktür; x86-64'te birkaç kilobayt tutar ve sürüme göre değişir. Ama içindekilerin çoğu doğrudan veri değil, başka yapılara işaretçidir.



Şekil 15.12: `task_struct` ve gösterdiği yapılar. Soldakiler iş parçacıkları arasında paylaşılabilir; her görevin kendi çekirdek yığını vardır.

Yapının kendi içinde tuttuğu alanlar görevin kimliğini ve çizelgeleme durumunu anlatır. `__state` görevin hâlidir. `pid` ve `tgid` kimliklerdir. `prio` ve `policy` önceliği ve çizelgeleme politikasını verir. `se` alanı gömülü bir `sched_entity` yapısıdır; adil çizelgeleyicinin kullandığı `vruntime` ve `deadline` değerleri buradadır. `parent` ve `children` süreç ağacını kurar. Geri kalan alanlar işaretçidir:

- `mm` → `mm_struct`: adres alanı. Bellek bölgelerinin (VMA) listesi ve en üst sayfa tablosunun adresi (`pgd`) buradadır. Çekirdek iş parçacıklarının `mm`'si boştur.
- `files` → `files_struct`: dosya tanımlayıcısı tablosu. Her girdi bir `struct file`'ı gösterir.
- `fs` → `fs_struct`: kök dizin, çalışma dizini ve umask.
- `sighand` ve `signal`: sinyal işleyicileri ve süreç düzeyindeki sinyal durumu.
- `cred`: uid, gid ve yetenekler.
- `nsproxy`: görevin içinde yaşadığı ad alanları (PID, ağ, bağlama …).
- `cgroups` → `css_set`: görevin bağlı olduğu kontrol grupları ve sınırları.

Bu düzenin sonucu, Linux'un en şaşırtıcı tasarım kararıdır: **süreç ile iş parçacığı arasında yapısal bir fark yoktur**. İkisi de bir görevdir, yani bir `task_struct`'tır. Fark, alt yapıların paylaşılıp paylaşılmadığıdır. İki ayrı sürecin `mm` işaretçileri farklı `mm_struct`'ları gösterir. Aynı sürecin iki iş parçacığının `mm` işaretçileri ise aynı nesneyi gösterir.

Tanım 15.2

Linux'ta her görevin çekirdek içi benzersiz kimliği `pid`'dir. Aynı süreçteki iş parçacıklarının ortak kimliği **iş parçacığı grubu kimliğidir** (*thread group ID*, `tgid`). Kullanıcının "PID" diye gördüğü değer `tgid`'dir: `getpid()` `tgid`'i, `gettid()` görevin kendi `pid`'ini döndürür.

Örnek 15.2

PID'i 900 olan tek iş parçacıklı bir süreç iki kez `pthread_create` çağırıyor. Çekirdek yeni görevlere sırayla 901 ve 902 kimliklerini veriyor. Her görevde `getpid()` ve `gettid()` ne döndürür?

1. Ana iş parçacığı: `pid` = 900, `tgid` = 900. `getpid()` = 900, `gettid()` = 900. Grubun lideri, kimliği grup kimliğine eşit olan görevdir.
2. Yeni iş parçacıkları `CLONE_THREAD` ile doğduğu için grubun `tgid`'ini miras alır: `getpid()` ikisinde de 900'dür. `gettid()` 901 ve 902'dir.
3. `ps` tek bir 900 süreci gösterir. `ps -L` ya da `/proc/900/task/` dizini üç görevi ayrı ayrı listeler.

`__state` alanının değerleri `ps`'in STAT sütununda harflerle görünür. R çalışıyor ya da çalışmaya hazırdır; Linux bu iki durumu ayırmaz. S kesilebilir uykudur: görev bir olayı bekler ve bir sinyalle uyandırılabilir. D **kesilemez uykudur**: görev çoğunlukla bir disk işleminin bitmesini bekler ve sinyaller onu uyandırmaz. T durdurulmuş, Z zombi demektir. D durumundaki bir görev `SIGKILL` ile bile hemen ölmez. Yanıt vermeyen bir ağ dosya sistemi bu yüzden `kill -9`'a direnen süreçler üretebilirdi. Çekirdek bu sorunu azaltmak için bekleyişlerin çoğunda artık `TASK_KILLABLE` kullanır: görev yalnız ölümcül bir sinyalle uyandırılabilir.

15.4.4 `clone()` bayrakları ve NPTL

Süreç ile iş parçacığı aynı yapıyla temsil edildiğine göre, ikisini aynı çağrı yaratabilir. O çağrı `clone()`'dur.

Tanım 15.3

`clone(bayraklar, …)` yeni bir görev oluşturan Linux sistem çağrısıdır. Bayrakların her biri, ebeveynin bir alt yapısının çocukla *paylaşılacağını* ya da çocuğa *yeni* bir kopyasının verileceğini seçer. Bayrak verilmeyen yapı kopyalanır.

Paylaşım bayrakları `task_struct`'ın işaretçileriyle birebir eşleşir (Tablo 15.5). `CLONE_VM` verilirse çocuğun `mm`'si ebeveyninkiyle aynı nesneyi gösterir; verilmezse adres alanı COW ile kopyalanır. `CLONE_FILES` tanımlayıcı tablosunu, `CLONE_FS` kök ve çalışma dizinini, `CLONE_SIGHAND` sinyal işleyicilerini paylaştırır. `CLONE_THREAD` çocuğu ebeveynin iş parçacığı grubuna sokar: çocuk aynı `tgid`'i alır ve dışarıdan aynı süreç görünür.

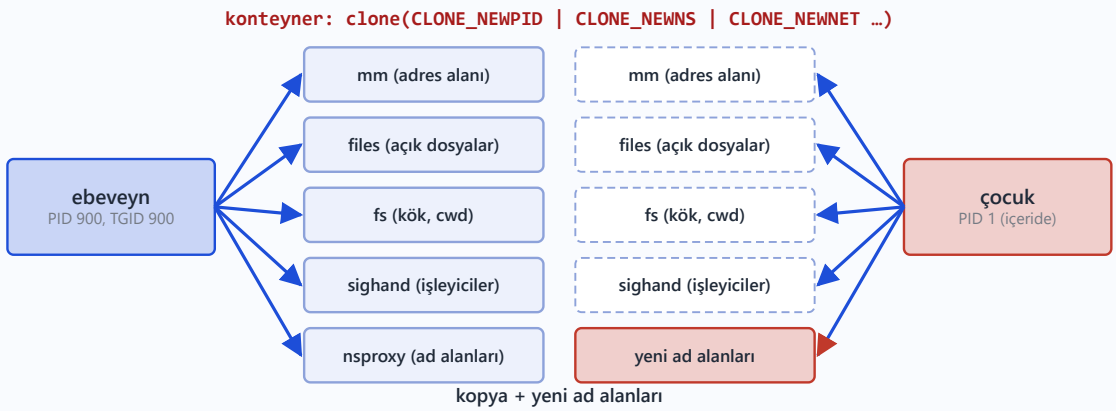
Bayrak	Verilirse	Verilmezse
`CLONE_VM`	adres alanı (`mm`) paylaşılır	COW ile kopyalanır

Bayrak	Verilirse	Verilmezse
`CLONE_FILES`	tanımlayıcı tablosu paylaşılır	tablo kopyalanır (aynı `struct file` 'lar)
`CLONE_FS`	kök, çalışma dizini, umask paylaşılır	kopyalanır
`CLONE_SIGHAND`	sinyal işleyicileri paylaşılır (`CLONE_VM` gerektirir)	kopyalanır
`CLONE_THREAD`	aynı iş parçacığı grubu, aynı `tgid` (`CLONE_SIGHAND` gerektirir)	yeni süreç, yeni `tgid`
`CLONE_NEWPID`, `CLONE_NEWNS`, `CLONE_NEWNET`, `CLONE_NEWUTS`, `CLONE_NEWIPC`, `CLONE_NEWUSER`, `CLONE_NEWCGROUP`	çocuk yeni bir ad alanında başlar	ebeveynin ad alanında kalır

Tablo 15.5: Başlıca `clone` bayrakları ve etkilediği yapılar.

Üç tipik çağrı bayrakların gücünü gösterir (Şekil 15.13).

1. `fork()`. glibc'de `fork`, yalnız `SIGCHLD` bayrağıyla çağrılan bir `clone`'dur. Hiçbir paylaşım bayrağı yoktur; çocuk her yapının kopyasını alır, bellek COW ile paylaşılır. Ayrı PID, ayrı TGID: yeni bir süreç. `SIGCHLD` burada bir paylaşım bayrağı değil, çocuk bitince ebeveyne gönderilecek sinyalin numarasıdır.
2. `pthread_create()`. Linux'un iş parçacığı kütüphanesi NPTL (*Native POSIX Thread Library*) şu bayrakları verir: `CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SYSVSEM | CLONE_SIGHAND | CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID`. Her şey paylaşılır ve yeni görev aynı TGID'e girer. Son üç bayrak iş parçacığının yerel deposunu kurar ve bitişini `pthread_join`'e bildirir.
3. **Konteyner.** Ad alanı bayrakları çocuğa dünyanın yeni bir görünümünü verir. `CLONE_NEWPID` ile doğan çocuk kendi PID ad alanının ilk sürecidir: içeride kendini PID 1 sanır, dışarıda sıradan bir PID taşır. Konteynerlerin temeli budur.



Şekil 15.13: Ad alanı bayraklarıyla `clone`: çocuk yapıların kopyasını ve yeni ad alanlarını alır; içeride PID 1'dir.

NPTL'in kurduğu model **1:1**'dir: her kullanıcı iş parçacığı bir çekirdek görevine karşılık gelir. Çekirdek iş parçacıklarını doğrudan çizelgeler; biri bloke olunca ötekiler çalışmayı sürdürür, çok çekirdekli makinede gerçekten paralel çalışırlar. NPTL 2003'te, 2.6 serisiyle geldi. Ondan önceki LinuxThreads da 1:1'di ama her iş parçacığı ayrı bir PID taşıyordu ve sinyal işleme POSIX'e uymuyordu. `CLONE_THREAD` ve TGID kavramı bu sorunları çözmek için eklendi.

Bayraklar bağımsız olarak birleştirilebildiği için "yarım süreç, yarım iş parçacığı" görevler de mümkündür. `CLONE_VM | CLONE_FILES` ile, ama `CLONE_THREAD` olmadan doğan bir görev ayrı bir süreç olarak görünür; kendi TGID'i vardır. Yine de ebeveyniyle aynı belleği ve aynı tanımlayıcı tablosunu kullanır. Birinin açtığı dosya ötekinde de açıktır. Klasik UNIX'te böyle bir ara biçim yoktur. `clone3()` (5.3) aynı işi daha geniş bir argüman yapısıyla yapar; yeni bayraklar artık ona eklenir.

Sık yapılan hata

`ps`'teki PID'in bir iş parçacığını gösterdiğini sanmak. Çok iş parçacıklı bir sunucu `ps`'te tek satırdır, çünkü `ps` TGID'leri listeler. Bir iş parçacığına sinyal göndermek için `tgkill` ve görev kimliği gerekir; `kill` sinyali bütün gruba gönderir ve grup içinden uygun bir iş parçacığı onu alır.

15.5 Sinyaller

15.5.1 POSIX sinyalleri

Tanım 15.4

Sinyal (*signal*), bir sürece bir olayı bildiren yazılım kesmesidir. Her sinyalin bir numarası ve varsayılan bir eylemi vardır. Süreç her sinyal için varsayılan eylemi, yok saymayı ya da kendi işleyicisini seçebilir.

Sinyaller iki türlü doğar. **Senkron** sinyaller sürecin kendi komutunun sonucudur: geçersiz bellek erişimi `SIGSEGV`, tam sayıyı sıfıra bölme `SIGFPE` üretir. **Eş zamansız** sinyaller dışarıdan gelir: klavyeden Ctrl-C `SIGINT`, `kill` komutu varsayılan olarak `SIGTERM` üretir. Bir işleyicinin olağan işleri kaynakları temizlemek, günlük yazmak ya da yapılandırmayı yeniden okumaktır.

Sinyal	No	Ne zaman üretilir?	Varsayılan
`SIGHUP`	1	Terminal bağlantısı koptu; cinlerde "yapılandırmayı yeniden oku"	sonlandır
`SIGINT`	2	Ctrl-C (ön plan süreç grubuna)	sonlandır
`SIGQUIT`	3	Ctrl-\	çekirdek dökümü
`SIGABRT`	6	`abort()`	çekirdek dökümü

Sinyal	No	Ne zaman üretilir?	Varsayılan
`SIGFPE`	8	Aritmetik hata	çekirdek dökümü
`SIGKILL`	9	`kill -9` ; yakalanamaz, yok sayılamaz	sonlandır
`SIGUSR1` / `SIGUSR2`	10 / 12	Kullanıcı tanımlı	sonlandır
`SIGSEGV`	11	Geçersiz bellek erişimi	çekirdek dökümü
`SIGPIPE`	13	Okuyanı kalmamış boruya yazma	sonlandır
`SIGALRM`	14	`alarm()` süresi doldu	sonlandır
`SIGTERM`	15	`kill` 'in varsayılanı; nazik sonlandırma isteği	sonlandır
`SIGCHLD`	17	Çocuk bitti ya da durdu	yok say
`SIGCONT` / `SIGSTOP`	18 / 19	Sürdür / durdur (yakalanamaz); Ctrl-Z `SIGTSTP` (20) üretir	sürdür / durdur

Tablo 15.6: Sık karşılaşılan sinyaller (numaralar x86 ve ARM Linux içindir).

`SIGKILL` ile `SIGTERM` 'in ikisi de `kill` komutuyla gönderilir. Fark yakalanabilirliktedir. `SIGTERM` 'i süreç yakalayabilir; açık dosyalarını kapatır, geçici dosyalarını siler ve nazikçe çıkar. `SIGKILL` ve `SIGSTOP` ise yakalanamaz, engellenemez ve yok sayılamaz. Yönetici, kontrolden çıkmış bir süreci her durumda durdurabilsin diye böyle tasarlanmışlardır. Doğru sıra önce `SIGTERM` , gerekirse `SIGKILL` 'dir. `SIGRTMIN` ile `SIGRTMAX` arasındaki numaralar (glibc'de 34–64) gerçek zamanlı sinyallere ayrılmıştır. Klasik sinyallerden farklı olarak bunlar kuyruğa girer: aynı sinyal üç kez gelirse üç kez teslim edilir.

15.5.2 Sinyalin teslim yolu

Bir sinyalin üretilmesi ile işlenmesi ayrı anlardır (Şekil 15.14). Arada sinyal **bekler**. Ctrl-C örneği bu yolu baştan sona gösterir.

1. **Üretim.** Kullanıcı Ctrl-C'ye basar. tty sürücüsünün satır disiplini bu karakteri tanır ve ön plan süreç grubundaki her sürece `SIGINT` üretir. Çekirdeğin yaptığı tek iş, hedef görevin bekleyen sinyal kümesinde 2 numaralı sinyalin bitini açmaktır.
2. **Denetim.** Sinyal hemen işlenmez. Süreç bir sistem çağrısından ya da bir kesmeden kullanıcı moduna dönerken çekirdek “bekleyen ve engellenmemiş bir sinyal var mı?” diye bakar. Uyuyan bir süreç ise sinyalle uyandırılır ve o da dönüş yolunda bu denetimden geçer.
3. **Çerçeve.** Süreç `sigaction` ile bir işleyici kurmuşsa çekirdek sürecin kullanıcı yığınına bir **sinyal çerçevesi** yazar. Çerçevede kesintiye uğrayan anın yazmaçları ve bir dönüş trampolini bulunur. Çekirdek program sayacını işleyicinin adresine ayarlar.
4. **İşleyici.** İşleyici kullanıcı modunda, sürecin kendi koduyla çalışır.
5. **Dönüş.** İşleyici dönünce trampolin `rt_sigreturn` sistem çağrısını yapar. Çekirdek çerçevedeki yazmaçları geri yükler.
6. **Devam.** Süreç kesildiği komuttan (şekilde PC = 0x401a2c) devam eder. İşleyici kurulmamış olsaydı varsayılan eylem uygulanırdı; `SIGINT` için bu, sürecin sonlanmasıdır.



Şekil 15.14: Bir sinyalin üretilmesi, beklemesi ve işleyiciye teslimi. Dönüş `rt_sigreturn` çağrısıyla olur.

İşleyici, kesilen kodun ortasında, onunla aynı iş parçacığında çalışır. Süreç tam o anda `malloc`'un içinde, heap'in kilidini tutarken kesilmiş olabilir. İşleyici de `malloc` ya da `printf` çağırırsa aynı kilidi ikinci kez almaya çalışır ve süreç kilitlenir. Bu yüzden işleyicide yalnız **eş zamansız sinyal güvenli** (*async-signal-safe*) işlevler kullanılabilir: `write`, `_exit` gibi. En sağlam kalıp, işleyicide yalnız `volatile sig_atomic_t` türünden bir bayrağı 1 yapmak ve asıl işi ana döngüde bayrağa bakarak yürütmektir.

> **Sık yapılan hata**
>
> `sigreturn`'ü işleyiciye giriş çağrısı sanmak. İşleyiciye giriş bir sistem çağrısı değildir; çekirdek program sayacını değiştirerek süreci işleyiciye "yollar". `rt_sigreturn` işleyiciden *dönüşü* yapar ve onu programcı değil, trampolin çağırır.

15.5.3 Kod izleme: bekleyen sinyalleri seçme

Çekirdek her görev için iki bit kümesi tutar: **bekleyen** sinyaller ve **engellenen** sinyaller. n numaralı sinyal (n − 1). bittedir. Kullanıcı moduna dönüşteki denetimin sadeleştirilmiş hâli şöyledir:

```
/* kullanıcı moduna dönerken (çekirdek) */
void sinyal_bak(task *t) {
    unsigned long h = t->bekl & ~t->engl;
    while (h) {
        int sig = __builtin_ctzl(h) + 1;
        t->bekl &= ~(1UL << (sig - 1));
        teslim_et(t, sig); /* işleyici/varsayılan */
        h = t->bekl & ~t->engl;
    }
}
/* önce:  sigprocmask(SIG_BLOCK, {SIGINT}) */
/* sonra: sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```

Üçüncü satır teslime hazır kümeyi hesaplar: hazır = bekleyen & ~engelli. Döngü bu küme boşalana kadar döner. `__builtin_ctzl(h)` sayının sağdaki sıfır bitlerini sayar, yani en düşük açık bitin konumunu verir; +1 bunu sinyal numarasına çevirir. Böylece en küçük numaralı hazır sinyal önce seçilir. Seçilen sinyalin biti bekleyen kümesinden silinir ve sinyal teslim edilir. Teslimden sonra küme yeniden hesaplanır, çünkü işleyicinin çalışması kümeleri değiştirebilir.

Senaryo şöyledir. Süreç kritik bir bölüme girmeden önce `sigprocmask(SIG_BLOCK, {SIGINT})` ile `SIGINT`'i engeller. Uyurken üç sinyal gelir: Ctrl-C (2), `kill -USR1` (10) ve bir çocuğun bitişi (17). `SIGUSR1` için bir işleyici kuruludur.

Olay	bekleyen	engelli	hazır	Sonuç
`SIG_BLOCK {SIGINT}`	{}	{2}	{}	—
2, 10, 17 gelir	{2, 10, 17}	{2}	—	yalnız bitler açılır
Kullanıcı moduna dönüş	{2, 10, 17}	{2}	{10, 17}	en küçük: 10
10 teslim	{2, 17}	{2}	{17}	`SIGUSR1` işleyicisi çalışır
17 teslim	{2}	{2}	{}	`SIGCHLD`: varsayılan, yok say
`SIG_UNBLOCK {SIGINT}`	{2}	{}	{2}	dönüşte aynı denetim
2 teslim	{}	{}	{}	işleyici yok: süreç sonlanır

Tablo 15.7: `sinyal_bak` izlemesi (bit işlemleri Python ile yeniden çalıştırıldı).

Bit gösterimiyle, üç sinyal geldikten sonra bekleyen kümesi 0x10202'dir: 2. sinyal 1. bit (0x2), 10. sinyal 9. bit (0x200), 17. sinyal 16. bit (0x10000). Engelli küme 0x2'dir. Hazır küme 0x10202 & ~0x2 = 0x10200 olur. En düşük açık bit 9'dur, yani sinyal 10.



Şekil 15.15: İzlemenin sonu: engel kalkınca hazır kümesine giren `SIGINT` varsayılan eylemle süreci sonlandırır.

İzlemeden iki ders çıkar. Birincisi, **engellemek ertelemektir, yok saymak değildir**. `SIGINT` kritik bölüm boyunca kaybolmadı; bekleyen kümesinde durdu ve engel kalkar kalkmaz teslim edildi. Yok saymak (`SIG_IGN`) ise sinyali geldiği anda atar. İkincisi, klasik sinyaller sayılmaz. Bekleyen kümesi bir bit kümesidir. Ctrl-C kritik bölüm sırasında üç kez basılsaydı bit yine tek kez açık olurdu ve sinyal tek kez teslim edilirdi.

15.6 Çizelgeleme

15.6.1 Çizelgeleme sınıfları

Linux çizelgeleyicisi tek bir algoritma değil, öncelik sırasına dizilmiş **çizelgeleme sınıflarıdır**. Her görev bir sınıfa aittir. Bir CPU boşalınca çekirdek sınıflara sırayla sorar ve çalıştırılacak görevi veren ilk sınıf kazanır. Sıra şöyledir:

1. **stop**: çekirdeğin bir CPU'yu durdurmak ya da görev taşımak için kullandığı iç sınıf.
2. **deadline**: `SCHED_DEADLINE` politikası. Görev bir çalışma süresi, bir son tarih ve bir dönem bildirir; çizelgeleyici en erken son tarihi önce çalıştırır.
3. **rt**: `SCHED_FIFO` ve `SCHED_RR`, 1–99 arası sabit öncelikler. FIFO görevi kendi isteğiyle bırakana kadar çalışır; RR aynı öncelikteki görevler arasında zaman dilimiyle döner.
4. **fair**: `SCHED_NORMAL` (adı eskiden `SCHED_OTHER`), `SCHED_BATCH` ve `SCHED_IDLE`. Sıradan süreçlerin hepsi buradadır. 6.6'dan beri bu sınıfı EEVDF yönetir.
5. **idle**: CPU'da başka iş yokken çalışan boşta görevi.

6.12 ile gelen `sched_ext`, çizelgeleme politikasının eBPF programı olarak yüklenmesine izin verir. Bir şirket kendi iş yüküne uygun çizelgeleyiciyi çekirdeği yeniden derlemeden deneyebilir. Program hata yaparsa çekirdek onu kaldırıp görevleri adil sınıfa geri verir. Mekanizma çekirdekte, politika kullanıcıdadır.

Her CPU'nun kendi **çalışma kuyruğu** (`struct rq`) vardır. Her sınıf bu kuyrukta kendi veri yapısını tutar. Görevler CPU'lar arasında arada bir dengelenir. Yük dengeleyici CPU'ları çizelgeleme alanları (*scheduling domains*) hiyerarşisinde görür: aynı çekirdeğin donanım iş parçacıkları, aynı yongadaki çekirdekler, aynı NUMA düğümündeki yongalar. Görev önce yakın düzeyde taşınır, çünkü önbelleği ve belleği orada daha sıcaktır.

15.6.2 O(n)'den CFS'e

Adil sınıfın seçim kuralı yirmi yılda üç kez değişti (Tablo 15.8). 2.4 çekirdeğindeki çizelgeleyici her seçimde bütün hazır görevleri tarardı. Görev sayısıyla doğrusal büyüyen bu maliyet, yüzlerce görevli sunucularda sorun oldu. 2.6.0 ile gelen **O(1) çizelgeleyicisi** her öncelik düzeyi için ayrı bir liste tuttu: 140 listelik bir dizi ve hangi listenin boş olmadığını gösteren bir biteşlem. En yüksek öncelikli görev, biteşlemde ilk açık biti bulan tek bir komutla seçiliyordu. Zaman dilimi biten görev ikinci bir "süresi dolmuş" diziye geçerdi, etkin dizi boşalınca iki dizinin rolü değişirdi. Hız sorunu çözüldü, ama etkileşimli görevleri tanımak için kullanılan sezgisel kurallar karmaşıklaştı ve öngörülemez davranışlar üretti.

Sürüm	Seçim kuralı	Veri yapısı
2.4 (2001)	O(n): her seçimde bütün hazır görevler taranır	tek liste

Sürüm	Seçim kuralı	Veri yapısı
2.6.0–2.6.22 (2003)	O(1): öncelik başına liste, etkin ve süresi dolmuş iki dizi	140 listelik dizi + biteşlem
2.6.23–6.5 (2007)	CFS: en küçük `vruntime`	kırmızı-siyah ağaç
6.6+ (2023)	EEVDF: uygun görevler (v ≤ V) içinde en erken sanal son tarih	aynı ağaç, alt ağaç bilgisiyle zenginleştirilmiş

Tablo 15.8: Adil sınıfın seçim kuralının evrimi.

2.6.23'te Ingo Molnár'ın **tamamen adil çizelgeleyicisi** (*Completely Fair Scheduler*, CFS) geldi. CFS'in fikri ideal bir çok görevli işlemciye öykünmektir. İdeal işlemcide n görev aynı anda, her biri 1/n hızla çalışırdı. Gerçek işlemci bunu yapamaz, ama hangi görevin kendi payının en çok gerisinde kaldığını izleyebilir. Bu izleme her görevin **sanal çalışma süresiyle** yapılır.

15.6.3 Ağırlıklar ve sanal çalışma süresi

Tanım 15.5

Her görevin nice değerinden türeyen bir **ağırlığı** w vardır; nice 0'ın ağırlığı 1024'tür. Görev gerçek zamanda Δt süre çalışınca **sanal çalışma süresi** (`vruntime`) şu kadar artar:

$$\Delta v = \Delta t \times 1024 / w$$

nice değeri −20 ile +19 arasındadır; küçük değer yüksek öncelik demektir. Ağırlıklar çekirdekteki `sched_prio_to_weight` tablosundan okunur. Tablo her nice adımında ağırlığı yaklaşık 1,25 kat değiştirir: nice 1 → 820, nice 0 → 1024, nice −1 → 1277. Sonuç, nice değeri bir artan görevin CPU payının kabaca %10 azalmasıdır. Uçlar nice −20 için 88.761, nice 19 için 15'tir.

Formül iki uçta sezgiyle örtüşür. Ağırlığı büyük görevin sanal saati yavaş işler; gerçek zamanda çok çalışsa da sanal süresi az artar. Ağırlığı küçük görevin sanal saati hızlı işler. CFS'in kuralı, en küçük `vruntime` 'lı görevi çalıştırmaktır. Görevler kırmızı-siyah ağaçta `vruntime` 'a göre sıralanır; en soldaki düğüm seçilir. Eklemek ve silmek O(log n)'dir. En soldaki düğüm önbellekte tutulduğu için seçim O(1)'dir.

Örnek 15.3

nice 0 (w = 1024), nice 5 (w = 335) ve nice −5 (w = 3121) olan üç görev 6 ms çalışıyor. Her birinin `vruntime` 'ı ne kadar ilerler? Sürekli hazır bir nice 0 ile bir nice 5 görevi tek CPU'yu hangi oranda paylaşır?

1. nice 0: 6 × 1024 / 1024 = 6 ms.
2. nice 5: 6 × 1024 / 335 ≈ 18,34 ms. Sanal saati üç kat hızlı işler.

3. nice −5: 6 × 1024 / 3121 ≈ 1,97 ms.
4. Uzun vadede iki görevin `vruntime`'ı birbirine yakın kalır. Bunun için gerçek çalışma süreleri ağırlıklarla orantılı olmalıdır: nice 0 için 1024 / (1024 + 335) ≈ %75,3, nice 5 için 335 / 1359 ≈ %24,7.

Yeni uyanan bir görev ne olacak? Saatlerce uyumuş bir görevin `vruntime`'ı çok geride kalmıştır. Olduğu gibi kuyruğa girerse uzun süre CPU'yu tekeline alır. CFS bu yüzden uyanan görevin `vruntime`'ını kuyruğun en küçük değerinin biraz gerisine çekerdi. Bu tür ayarlar zamanla çoğaldı: gecikme hedefi (`sched_latency`), en küçük dilim, uyanma ayrıntı düzeyi. Her biri bir sorunu çözerken başka bir iş yükünde yeni bir sorun doğurdu. CFS'in asıl eksiği ise kuramsaldı. CFS bir görevin *ne kadar* CPU alacağını iyi belirliyordu, ama *ne zaman* alacağı konusunda bir güvence vermiyordu. Ses ya da oyun gibi gecikmeye duyarlı bir görev, payı küçük olsa bile kısa aralıklarla CPU'ya ihtiyaç duyar.

15.6.4 EEVDF: uygunluk ve sanal son tarih

Linux 6.6 (2023) adil sınıfın seçim kuralını **EEVDF** (*Earliest Eligible Virtual Deadline First*) ile değiştirdi. Algoritma Ion Stoica ve Hussein Abdel-Wahab'ın 1995 tarihli çalışmasına dayanır; çekirdeğe Peter Zijlstra uyarladı. EEVDF iki kavram ekler: uygunluk ve sanal son tarih.

Tanım 15.6

Kuyruğun **ortalama sanal zamanı**, görevlerin `vruntime`'larının ağırlıklı ortalamasıdır: $V = \Sigma\, w_i v_i \,/\, \Sigma\, w_i$. Bir görevin **alacağı** (*lag*) $\mathrm{lag}_i = w_i\,(V - v_i)$'dir. Alacağı sıfır ya da pozitif olan, yani $v_i \le V$ koşulunu sağlayan görev **uygundur** (*eligible*). Her görevin bir **sanal son tarihi** vardır: $d_i = v_i + \text{dilim} \times 1024 \,/\, w_i$. EEVDF, uygun görevler arasından sanal son tarihi en erken olanı seçer.

Alacak, görevin ideal işlemcide alacağı hizmet ile gerçekte aldığı hizmet arasındaki farktır. Pozitif alacak, görevin payının gerisinde kaldığını gösterir; çizelgeleyici ona borçludur. Negatif alacak, görevin payından fazlasını almış olduğunu gösterir; görev uygun değildir ve sırası gelene kadar bekler. Bütün görevlerin alacakları toplamı her an sıfırdır. Uygunluk koşulu, bir görevin payını aşarak CPU'yu tekeline almasını engeller. Sanal son tarih ise *ne zaman* sorusunu yanıtlar. Dilimi kısa olan görevin son tarihi yakındır ve uygun olduğu anda öne geçer.

CFS'ten EEVDF'e geçişte değişmeyenler de az değildir. `sched_entity` içindeki `vruntime`, nice → ağırlık tablosu ve çalışma kuyruğundaki kırmızı-siyah ağaç aynen kaldı. Ağaç hâlâ `vruntime`'a göre sıralıdır. Her düğüm ayrıca kendi alt ağacındaki en erken son tarihi tutar. Çizelgeleyici bu ek bilgiyle "uygun ve en erken son tarihli" görevi ağaçta tek bir inişle, O(log n) adımda bulur. Değişen, CFS'in ayar düğmeleridir. Gecikme hedefi ve uyanma ayrıntı düzeyi kalktı; yerine tek bir **temel dilim** geldi. Değeri `/sys/kernel/debug/sched/base_slice_ns` dosyasında görülür. Varsayılan, milisaniyenin kesri olan bir taban değerin CPU sayısıyla ölçeklenmiş hâlidir; sekiz ve daha çok CPU'lu makinelerde birkaç milisaniyedir. Bu nottaki örneklerde dilim 3 ms alınmıştır.

Örnek 15.4

Tek CPU'da iki sürekli hazır görev var: A nice 0 (w = 1024), B nice 5 (w = 335). İkisi de v = 0'dan başlıyor, dilim 3 ms. Her görev seçilince dilimini bitirene kadar çalışıyor. İlk 24 ms'de seçimler nasıl olur?

1. t = 0. V = 0; ikisi de uygun. Son tarihler: $d_A = 0 + 3 = 3$, $d_B = 0 + 3 \times 1024 / 335 \approx 9{,}17$. A seçilir.
2. t = 3. $v_A = 3$, $d_A = 6$. $V = (3 \times 1024 + 0 \times 335) / 1359 \approx 2{,}26$. A uygun değil ($3 > 2{,}26$). Tek uygun görev B'dir; seçilir.
3. t = 6. $v_B \approx 9{,}17$, $d_B \approx 18{,}34$. $V \approx 4{,}52$. A uygun ($3 \le 4{,}52$), B değil. A seçilir.
4. t = 9 ve t = 12. V sırasıyla 6,78 ve 9,04; v_A 6 ve 9. A iki kez daha uygundur ve seçilir. B'nin v'si 9,17 olduğundan V onu henüz geçmemiştir.
5. t = 15. $v_A = 12$, $V \approx 11{,}30$. A uygun değil, B uygun ($9{,}17 \le 11{,}30$). B seçilir. Ardından desen tekrarlanır.

24 ms'de A altı dilim (18 ms), B iki dilim (6 ms) çalışır: %75 ile %25. Bu, ağırlıkların oranıdır (1024 : 335 ≈ 3 : 1).

t (ms)	v_A / d_A	v_B / d_B	V	Uygun	Seçilen
0	0 / 3	0 / 9,17	0	A, B	A
3	3 / 6	0 / 9,17	2,26	B	B
6	3 / 6	9,17 / 18,34	4,52	A	A
9	6 / 9	9,17 / 18,34	6,78	A	A
12	9 / 12	9,17 / 18,34	9,04	A	A
15	12 / 15	9,17 / 18,34	11,30	B	B
18	12 / 15	18,34 / 27,51	13,56	A	A
21	15 / 18	18,34 / 27,51	15,82	A	A

Tablo 15.9: Karar anlarında v, d ve V (A nice 0, B nice 5, dilim 3 ms; Python benzetimiyle üretildi).

Pay, CFS'te olduğu gibi ağırlıklarla belirlenir. EEVDF'in farkı, iki görevin dilimleri farklı olduğunda ortaya çıkar. Gecikmeye duyarlı bir görev `sched_setattr()` çağrısıyla daha kısa bir dilim isteyebilir; çağrının `sched_runtime` alanı adil sınıfta dilim isteği olarak yorumlanır. Bu olanak 6.12'de tamamlandı. Kısa dilim daha erken son tarih, dolayısıyla daha sık seçilme demektir. Pay ise değişmez.

Örnek 15.5

A ve B ikisi de nice 0. A'nın dilimi 3 ms, B 1 ms istemiş. İlk 12 ms'de seçim sırası nedir?

1. t = 0: ikisi de uygun; $d_A = 3$, $d_B = 1$. B seçilir ve 1 ms çalışır.

2. t = 1: $v_B = 1$, V = 0,5. Yalnız A uygundur; A 3 ms çalışır.
3. t = 4: $v_A = 3$, V = 2. B uygun, $d_B = 2$. B seçilir. t = 5'te (V = 2,5, $d_B = 3$) ve t = 6'da (V = 3, ikisi de uygun; $d_B = 4 < d_A = 6$) yine B seçilir.
4. t = 7: $v_B = 4 > V = 3{,}5$. A çalışır (7–10). Sonra B iki dilim daha alır.

Sıra B AAA BBB AAA BB'dir. 12 ms'de iki görev de 6 ms çalıştı; paylar eşittir. Ama B ilk dilimini hemen aldı ve hiçbir zaman 3 ms'den uzun beklemedi.

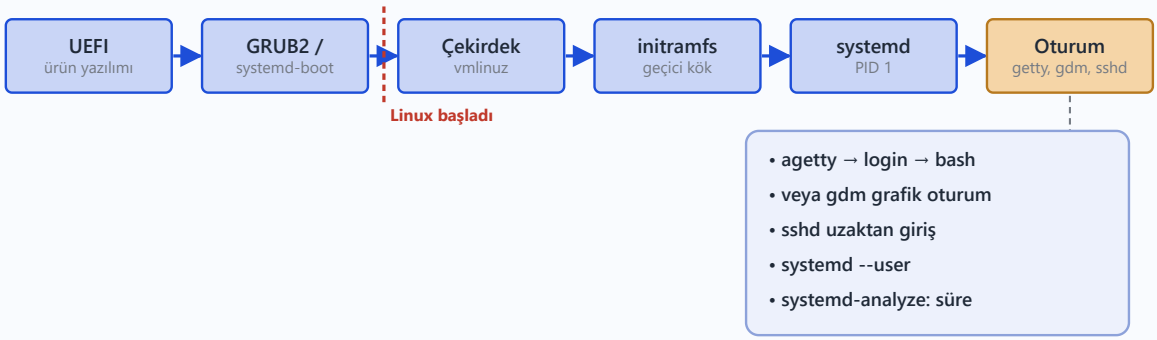
Sık yapılan hata

EEVDF'i "en küçük `vruntime`'ı seçen CFS"le karıştırmak. En küçük `vruntime`'lı görev her zaman uygundur, ama seçilen o olmayabilir. Örneğin V = 12 iken A (v = 10, d = 14) ve B (v = 12, d = 13) uygundur. EEVDF son tarihi erken olan B'yi seçer; CFS A'yı seçerdi.

15.7 Önyükleme

Güç düğmesine basıldığı andan oturum açma ekranına kadar bir dizi program birbirine devreder (Şekil 15.16).

1. **UEFI.** Ana karttaki ürün yazılımı donanımı sınar. Diskteki EFI sistem bölümünü (ESP) bulur ve oradaki EFI uygulamasını yükler. Secure Boot açıksa her aşama bir sonrakinin imzasını doğrular.
2. **Önyükleyici.** GRUB2 ya da systemd-boot çekirdek görüntüsünü (`vmlinuz`) ve `initramfs`'i belleğe yükler, çekirdeğe komut satırı parametrelerini verir. Çekirdek bir EFI uygulaması olarak doğrudan da başlatılabilir. Çekirdek, initramfs ve komut satırı tek bir imzalı dosyada birleşirse buna birleşik çekirdek görüntüsü (UKI) denir.
3. **Çekirdek.** Sıkıştırılmış görüntü kendini açar. `start_kernel()` bellek yönetimini, kesmeleri ve çizelgeleyiciyi kurar. Bu kodu çalıştıran ilk görev PID 0 olur ve sonunda boşta görevine dönüşür. Çekirdek `kernel_init`'i (PID 1) ve çekirdek iş parçacıklarının atası `kthreadd`'yi (PID 2) oluşturur.
4. **initramfs.** Kök dosya sistemini bağlamak için gereken sürücüler çekirdekte olmayabilir: NVMe, RAID, şifreli disk. initramfs RAM'de geçici bir kök sağlar. İçindeki betikler gerekli modülleri yükler, gerçek kökü bağlar ve `switch_root` ile ona geçer.
5. **systemd.** PID 1 artık systemd'dir. Birim (*unit*) dosyalarındaki bağımlılıklara göre hizmetleri paralel başlatır ve bir hedefe ulaşır: `multi-user.target` ya da `graphical.target`.
6. **Oturum.** Konsolda `agetty` → `login` → kabuk, masaüstünde gdm, uzaktan sshd kullanıcıyı karşılar. Kullanıcı oturumu için ayrıca bir `systemd --user` örneği başlar.



Şekil 15.16: Linux önyükleme aşamaları: UEFI, önyükleyici, çekirdek, initramfs, systemd ve oturum.

Klasik UNIX'teki `init → getty → login → sh` zinciri bugün de tanınabilir. Fark, PID 1'in artık hizmetleri bağımlılık çizgesine göre paralel başlatan, onları cgroup'larda izleyen ve çöken hizmeti yeniden başlatan büyük bir program olmasıdır. Açılışın hangi aşamada ne kadar sürdüğü `systemd-analyze` ile ölçülür.

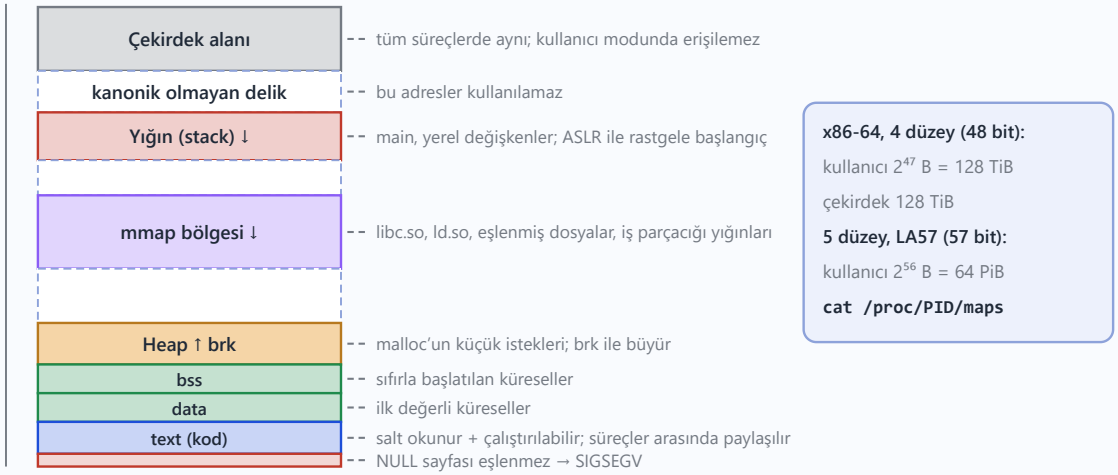
Sık yapılan hata

PID 1'i sıradan bir süreç sanmak. PID 1 sonlanırsa çekirdek panikler. Öksüz kalan süreçler ona bağlanır ve zombilerini o toplar. Konteynerde de içerideki PID 1 bu görevleri üstlenir; zombileri toplamayan bir uygulama PID 1 olarak çalıştırılırsa konteynerde zombiler birikir.

15.8 Bellek yönetimi

15.8.1 Sürecin adres alanı ve VMA'lar

x86-64'te dört düzeyli sayfalamayla sanal adresler 48 bittir. Linux bu alanı ikiye böler (Şekil 15.17). Alt yarı, 2^{47} B = 128 TiB, kullanıcıya aittir. Üst yarı, yine 128 TiB, çekirdeğindir. Çekirdek yarısı bütün süreçlerde aynıdır ve kullanıcı modunda erişilemez. İki yarının arasında kullanılamayan dev bir delik vardır. Beş düzeyli sayfalamada (LA57) kullanıcı alanı 2^{56} B = 64 PiB'a çıkar.



Şekil 15.17: x86-64 Linux'ta bir sürecin sanal adres alanı: kod, veri, bss, heap, mmap bölgesi, yığın ve çekirdek alanı.

Kullanıcı yarısının düzeni alttan üste şöyledir. En alttaki sayfa hiç eşlenmez; NULL işaretçiye erişen program `SIGSEGV` alır. Sonra program kodu (text) gelir: salt okunur ve çalıştırılabilir, aynı programı çalıştıran bütün süreçler arasında paylaşılır. Ardından ilk değerli küresel değişkenler (data) ve sıfırla başlatılan küreseller (bss) durur. Heap bss'in hemen üstünden başlar ve `brk` ile yukarı büyür. Adres alanının üst ucunda yığın vardır ve aşağı büyür. Yığının altında, yine aşağı doğru büyüyen mmap bölgesi bulunur: paylaşımlı kütüphaneler (`libc.so` , `ld.so`), eşlenmiş dosyalar ve iş parçacıklarının yığınları buraya yerleşir. ASLR (adres alanı düzeninin rastgeleleştirilmesi) yığının, mmap bölgesinin ve heap'in başlangıcını her çalıştırmada değiştirir; saldırganın adresleri tahmin etmesi zorlaşır. Heap ile mmap bölgesi arasındaki boşluk yalnız sanaldır, fiziksel bellek harcamaz.

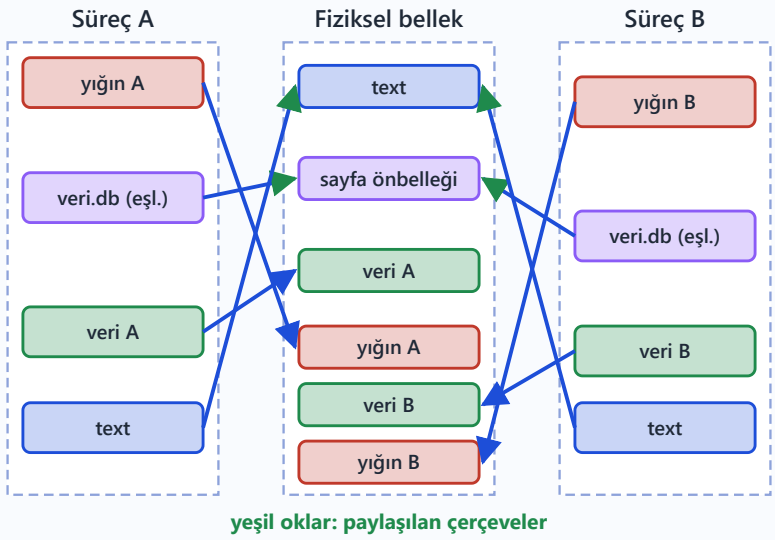
Tanım 15.7

Sanal bellek alanı (VMA, `struct vm_area_struct`), adres alanının aynı izinleri ve aynı kaynağı taşıyan bitişik bir bölgesidir. Her VMA başlangıç ve bitiş adresini, izinlerini (r, w, x, paylaşımlı ya da özel) ve varsa eşlendiği dosyayı tutar.

Bir sürecin VMA'ları `mm_struct` içinde tutulur. 6.1'den beri bunlar bir **maple ağacında** durur. Bu ağaç, aralık aramasına uygun, önbellek dostu bir B-ağacı türüdür; daha önceki kırmızı-siyah ağaç ile bağlı listenin yerini aldı. Sayfa hatası olunca çekirdeğin ilk sorusu “bu adres hangi VMA'da?” olur. Adres hiçbir VMA'da değilse ya da VMA erişime izin vermiyorsa sürece `SIGSEGV` gönderilir. Adres geçerli bir VMA'daysa sayfa ya dosyadan okunur ya da sıfırlanmış yeni bir çerçeve verilir. Bir sürecin VMA'ları `cat /proc/PID/maps` ile satır satır görülür.

15.8.2 Paylaşım, eşlenmiş dosyalar ve bellek çağrıları

İki süreç aynı programı çalıştırınca text sayfalarını fiziksel bellekte bir kez tutar (Şekil 15.18). İki sürecin sayfa tabloları aynı çerçeveleri gösterir. Aynı dosyayı paylaşımlı eşleyen iki süreç de aynı sayfa önbelleği çerçevelerini görür. Her sürecin veri ve yığın sayfaları ise kendine aittir.



Şekil 15.18: A ve B süreçleri program metnini ve eşlenmiş `veri.db` dosyasını aynı fiziksel çerçevelerde paylaşır.

Çağrı	İşlev
`brk` / `sbrk`	Heap'in sonunu taşıyarak büyütür ya da küçültür
`mmap`	Dosyayı, aygıtı ya da anonim belleği adres alanına eşler
`munmap`	Eşlemeyi kaldırır, belleği sisteme geri verir
`mprotect`	Bölgenin izinlerini (r/w/x) değiştirir
`madvise`	Çekirdeğe kullanım ipucu verir (ör. `MADV_DONTNEED`)
`msync`	Paylaşımlı eşlemenin kirli sayfalarını dosyaya yazar

Tablo 15.10: Bellek yönetimi sistem çağrıları.

`mmap`'in iki eşleme türü vardır. `MAP_SHARED` ile yazılanlar dosyaya ve aynı dosyayı eşleyen öteki süreçlere görünür. `MAP_PRIVATE` ile ilk yazmada sayfa kopyalanır ve dosya değişmez; programın veri kesimi böyle eşlenir. Dosyası olmayan eşlemeye anonim eşleme denir; sayfaları ilk dokunuşta sıfırlanmış çerçevelerden gelir. glibc'nin `malloc`'u küçük istekleri heap'ten karşılar ve heap'i gerektikçe `brk` ile büyütür. 128 KiB ve üstündeki istekleri ise doğrudan anonim `mmap` ile karşılar. Bu eşik varsayılandır ve glibc onu çalışma sırasında uyarlayabilir. Büyük bloklar böylece `free` anında `munmap` ile hemen sisteme döner.

15.8.3 Sayfa tablosu yürüyüşü: dört ve beş düzey

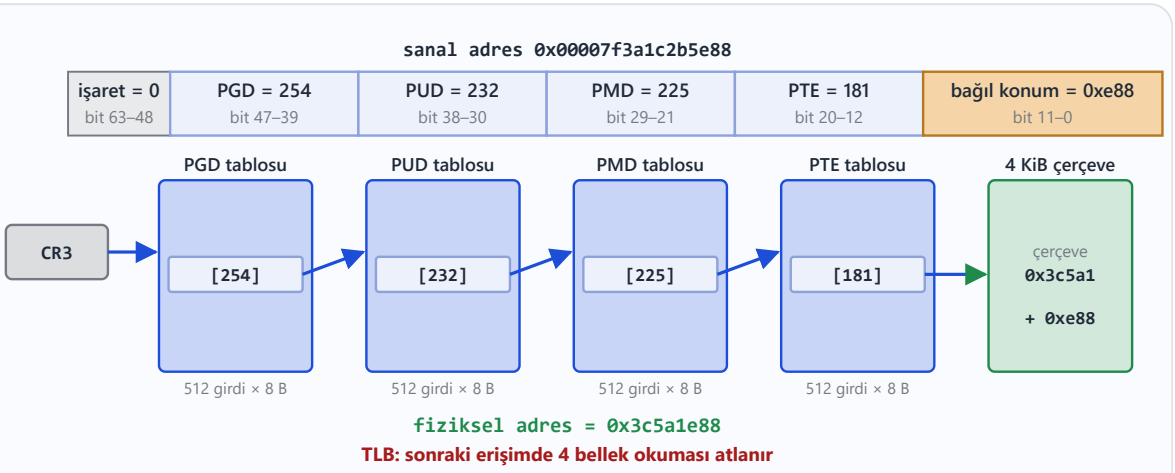
x86-64'te 4 KiB'lık sayfa 12 bitlik bağıl konum ister. 48 bitlik adresin kalan 36 biti dört tabloya bölünür. Her tablo 4 KiB'lık bir sayfadır ve 8 baytlık 512 girdi taşır; bir tablo indisi 9 bittir. Linux'un tablo adları şöyledir: PGD (bit 47–39), PUD (38–30), PMD (29–21) ve PTE (20–12). 63–48. bitler 47. bitin kopyası olmalıdır. Bu koşulu sağlayan adrese **kanonik** adres denir. Kullanıcı adreslerinde bu bitler 0, çekirdek adreslerinde 1'dir.

Örnek 15.6

Sanal adres 0x00007f3a1c2b5e88. Dört indis ve bağıl konum nedir? PTE'deki çerçeve numarası 0x3c5a1 ise fiziksel adres nedir?

1. Bağıl konum en düşük 12 bittir, yani son üç onaltılık hane: 0xe88.
2. PGD = adres >> 39 = 0xfe = 254. 63–48. bitler 0, 47. bit de 0'dır; adres kanoniktir.
3. PUD = (adres >> 30) & 0x1ff = 232. PMD = (adres >> 21) & 0x1ff = 225. PTE = (adres >> 12) & 0x1ff = 181.
4. Fiziksel adres = çerçeve × 4096 + bağıl konum = 0x3c5a1 × 0x1000 + 0xe88 = 0x3c5a1e88. Çarpma, onaltılıkta çerçeve numarasının sonuna üç sıfır eklemektir.

Yürüyüşün kendisi şöyledir (Şekil 15.19). `CR3` yazmacı sürecin PGD tablosunun fiziksel adresini tutar; bağlam değiştirmede çekirdek `CR3`'ü yeni sürecin `mm->pgd`'siyle yükler. PGD[254] bir PUD tablosunu gösterir. PUD[232] bir PMD tablosunu gösterir; 1 GiB'lık dev sayfa bu düzeyde de bitebilir. PMD[225] bir PTE tablosunu gösterir; 2 MiB'lık dev sayfa ve saydam dev sayfalar (THP) burada biter. PTE[181] asıl girdidir: geçerli (*present*) biti, çerçeve numarası, okuma/yazma (R/W), kullanıcı/yönetici (U/S), çalıştırma yasağı (NX), kirli (D) ve erişildi (A) bitleri. Geçerli biti 0 ise donanım sayfa hatası üretir.



Şekil 15.19: 0x00007f3a1c2b5e88 için dört düzeyli yürüyüş: PGD 254, PUD 232, PMD 225, PTE 181 ve bağıl konum 0xe88.

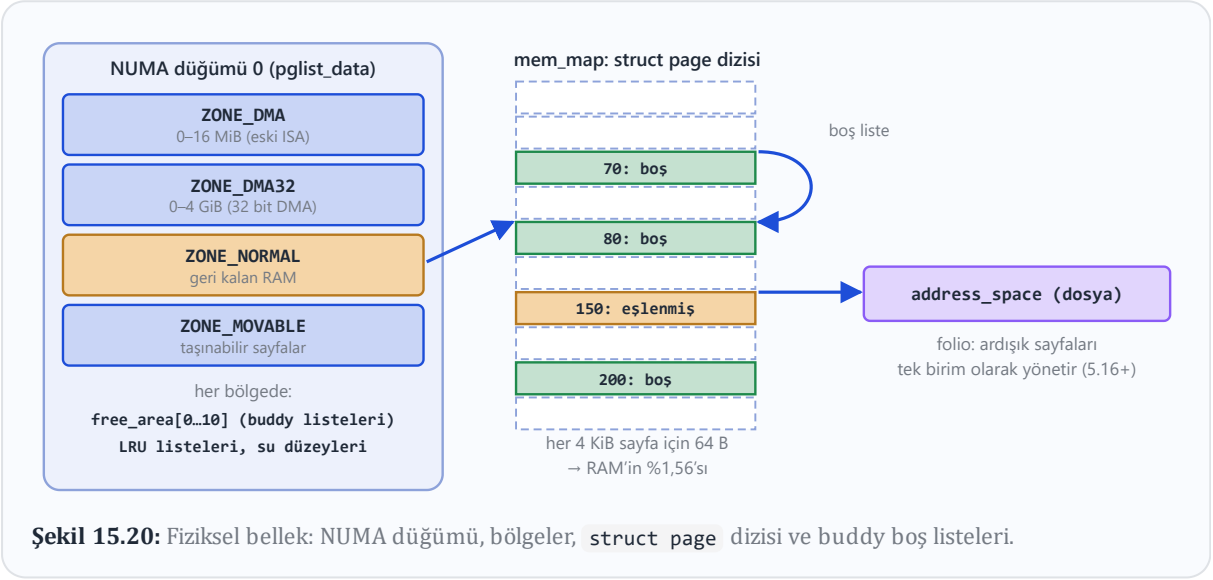
Her bellek erişiminde dört ek okuma yapmak pahalıdır. TLB çeviriyi önbelleğe alır; bir sonraki erişimde dört okuma atlanır. TLB ıskasında ise yürüyüşü donanım yapar; çekirdek yalnız tabloları kurar. Dev sayfalar yürüyüşü bir ya da iki düzey kısaltır ve tek bir TLB girdisiyle çok daha büyük bir alanı kapsar.

Beş düzeyli sayfalama (LA57) Linux 4.14 ile desteklenmeye başladı. PGD ile PUD arasına P4D adlı bir düzey ekler ve sanal adresi 57 bite çıkarır. Linux'un sayfa tablosu kodu genel olarak beş düzeylidir. Donanım daha azını kullanıyorsa aradaki düzeyler "katlanır": dört düzeyli bir makinede P4D, tek girdili ve PGD'ye gömülü sayılır. Aynı kod böylece iki düzeyli 32 bit ARM'dan beş düzeyli x86-64'e kadar çalışır.

> **Sık yapılan hata**
>
> İndisleri adresin onaltılık hanelerinden doğrudan okumaya çalışmak. 9 bitlik alanlar 4 bitlik hanelerle hizalı değildir. Doğru yol kaydırıp maskelemektir: (adres >> k) & 0x1ff, k = 39, 30, 21, 12. Yalnız 12 bitlik bağıl konum son üç haneye denk gelir.

15.8.4 Fiziksel belleğin gösterimi

Linux fiziksel belleği üç düzeyli bir hiyerarşiyle tanımlar: düğüm → bölge → sayfa (Şekil 15.20). Çok soketli bir sunucuda her işlemci soketinin kendine yakın belleği vardır. Bu düzene NUMA (tekdüze olmayan bellek erişimi) denir. Her NUMA düğümü bir `pglist_data` yapısıyla tanımlanır. Bir CPU belleği öncelikle kendi düğümünden ayırır, çünkü uzak düğüme erişim daha yavaştır.



Şekil 15.20: Fiziksel bellek: NUMA düğümü, bölgeler, `struct page` dizisi ve buddy boş listeleri.

Her düğüm bölgelere (*zone*) ayrılır. `ZONE_DMA` ilk 16 MiB'tır; yalnız 24 bit adresleyebilen eski ISA aygıtları içindir. `ZONE_DMA32` ilk 4 GiB'tır; 32 bit DMA yapabilen aygıtlar içindir. `ZONE_NORMAL` geri kalan RAM'dir. `ZONE_MOVABLE` yalnız taşınabilir sayfaları tutar; bellek modülü sökülmeden önce bo-

şaltılabilir ve dev sayfa oluşturmayı kolaylaştırır. 32 bit çekirdeklerde çekirdeğin doğrudan eşleyemediği yüksek bellek için `ZONE_HIGHMEM` de vardı. 64 bit sistemlerde bu bölge yoktur, çünkü çekirdek adres alanı bütün RAM'i doğrudan eşleyecek kadar büyüktür.

Her fiziksel sayfa çerçevesi için çekirdekte bir `struct page` bulunur. Yapının boyu 64 bayttır. Bunlar `mem_map` adlı büyük bir dizide durur. 4 KiB'lık her sayfa için 64 B, RAM'in 64 / 4096 ≈ %1,56'sı demektir. 16 GiB'lık bir makinede bu dizi 256 MiB tutar. Ardışık sayfaları tek bir birim olarak yönetmek için 5.16 ile **folio** kavramı geldi. Bir folio birden çok sayfayı kapsayabilir; sayfa önbelleği ve geri kazanım kodu sayfa sayfa yerine folio folio çalışır.

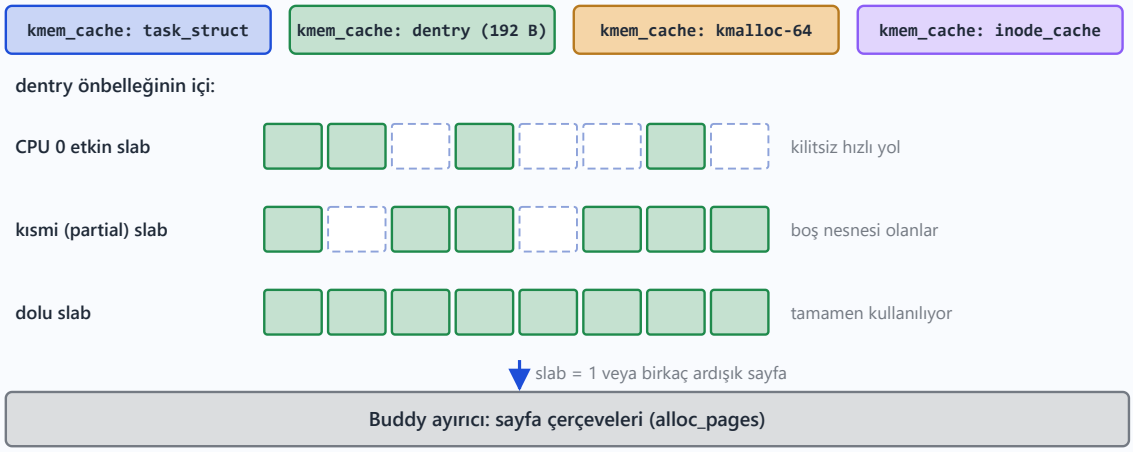
Her bölgenin `free_area` dizisi buddy ayırıcının boş listelerini tutar. Liste k, 2^k sayfalık boş blokları içerir. k 0'dan 10'a kadardır: 4 KiB'tan 4 MiB'a. İstek en küçük uygun bloktan karşılanır; büyük blok ikiye bölünür, serbest kalan blok "kardeşi" de boşsa birleşir. Bölgelerdeki boş blok sayıları `/proc/buddyinfo` dosyasından okunur. Her bölgenin ayrıca üç su düzeyi (min, düşük, yüksek) vardır; geri kazanım bunlara göre çalışır.

15.8.5 SLUB nesne ayırıcısı

Çekirdek çok sayıda küçük ve aynı boyda nesne ayırır: her süreç için bir `task_struct`, her açık dosya için bir `struct file`, her ad için bir dentry (192 B), her dosya için bir i-node. Bunların her biri için bir sayfa ayırmak hem yavaş hem israftır.

> **Tanım 15.8**
>
> **SLUB**, Linux'un nesne ayırıcısıdır. Her nesne türü için bir **nesne önbelleği** (`kmem_cache`) tutar. Önbellek, buddy ayırıcıdan aldığı bir ya da birkaç ardışık sayfayı (bir **slab**) aynı boyda nesnelere böler. Boş nesneler bir serbest listeye bağlanır.

Önbelleklerin bir kısmı belirli türlere aittir: `task_struct`, `dentry`, `inode_cache` (Şekil 15.21). `kmalloc` ise genel amaçlıdır ve boy sınıflarına ayrılmış önbellekler kullanır: `kmalloc-8`, `kmalloc-16` … `kmalloc-8k`. 100 baytlık bir istek `kmalloc-128`'den karşılanır. Bundan büyük istekler doğrudan buddy ayırıcıya gider. Fiziksel olarak bitişik olması gerekmeyen büyük alanlar için `vmalloc` kullanılır; o, dağınık çerçeveleri sanal olarak bitişik bir bölgeye eşler.



Şekil 15.21: SLUB: nesne önbellekleri buddy ayırıcıdan aldığı sayfaları sabit boyda nesnelere böler. Her CPU'nun etkin bir slab'ı, önbelleğin kısmi ve dolu slab'ları vardır.

Bir önbellekte üç tür slab bulunur. Her CPU'nun bir **etkin slab**'ı vardır; ayırma ve bırakmanın çoğu kilit almadan burada olur. **Kısmi** (*partial*) slab'larda hem dolu hem boş nesne vardır. **Dolu** slab'ların bütün nesneleri kullanımdadır. Etkin slab'da boş nesne kalmayınca CPU kısmi listeden yeni bir slab alır; o da yoksa buddy'den sayfa ister. Önbelleklerin durumu `/proc/slabinfo` dosyasından ya da `slabtop` komutuyla izlenir.

SLUB'ın adı tarihten gelir. Linux'ta uzun yıllar üç ayırıcı birlikte yaşadı. SLAB, Bonwick'in Solaris için önerdiği tasarımın ilk uyarlamasıydı ve karmaşık kuyruklar tutuyordu. SLOB küçük gömülü sistemler için yazılmıştı. SLUB, 2.6.22'de SLAB'ın sadeleştirilmiş hâli olarak geldi ve zamanla varsayılan oldu. SLOB 6.4'te, SLAB 6.8'de kaldırıldı. Bugün tek ayırıcı SLUB'dır.

Örnek 15.7

dentry nesnesi 192 bayttır. Tek sayfalık (4 KiB) bir slab'a kaç dentry sığar ve kaç bayt boş kalır?

1. 4096 / 192 = 21,33. Tam sayı kısmı 21 nesnedir.
2. 21 × 192 = 4032. Kalan 4096 − 4032 = 64 bayt.
3. Kayıp sayfanın %1,6'sıdır. Her dentry için ayrı bir 4 KiB'lık sayfa ayrılsaydı kayıp %95 olurdu.

15.8.6 Kod izleme: SLUB serbest listesi

SLUB'ın hızının sırrı, serbest listeyi tutmak için hiç ek bellek kullanmamasıdır. Boş bir nesnenin içi zaten kullanılmıyordur. SLUB “sonraki boş nesne” işaretçisini nesnenin kendi içine yazar. Hızlı yolun sadeleştirilmiş hâli şöyledir:

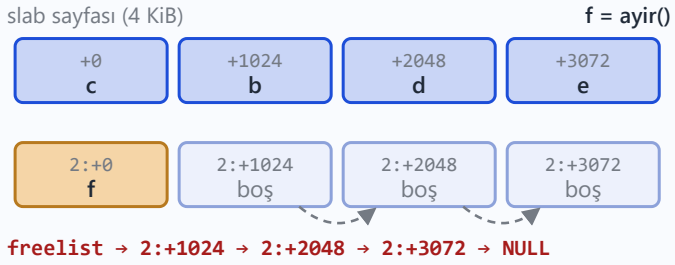
```
void *slab_ayir(kmem_cache *c) {
    void *nesne = c->freelist;    /* listenin başı */
    if (!nesne)
        nesne = yeni_slab(c);     /* yavaş yol */
    c->freelist = *(void **)nesne;  /* sonraki */
    return nesne;
}
void slab_birak(kmem_cache *c, void *nesne) {
    *(void **)nesne = c->freelist;  /* nesnenin içine */
    c->freelist = nesne;
}
/* a,b = ayir; birak(a); c,d,e,f = ayir */
```

`slab_ayir` listenin başını alır. Liste boşsa yavaş yola girer: `yeni_slab` buddy'den bir sayfa ister, onu nesnelere böler ve ilk nesneyi verir. Sonra `c->freelist`'i, alınan nesnenin ilk 8 baytında yazan işaretçiye ilerletir. `*(void **)nesne` ifadesi "nesnenin başındaki 8 baytı bir işaretçi olarak oku" demektir. `slab_birak` bunun tersini yapar: listenin eski başını bırakılan nesnenin içine yazar ve nesneyi listenin başı yapar. Liste böylece bir yığın gibi çalışır: son bırakılan ilk verilir.

İzleme `kmalloc-1k` önbelleğinde yapılır. Bir slab 4 KiB'lık tek sayfadır ve 1 KiB'lık dört nesne taşır: +0, +1024, +2048, +3072. Başta liste +0 → +1024 → +2048 → +3072 → NULL'dır. Program önce a ve b'yi ayırır, a'yı bırakır, sonra c, d, e ve f'yi ayırır.

İşlem	Verilen nesne	freelist sonra	Açıklama
a = ayir	+0	+1024	listenin başı; içindeki işaretçi +1024
b = ayir	+1024	+2048	hızlı yol: kilit yok, sayfa ayırma yok
birak(a)	—	+0	+0'ın içine eski baş (+2048) yazılır
c = ayir	+0	+2048	a'nın az önce bıraktığı "sıcak" nesne
d = ayir	+2048	+3072	
e = ayir	+3072	NULL	slab doldu
f = ayir	2:+0	2:+1024	yavaş yol: yeni sayfa, dört nesne

Tablo 15.11: Serbest liste izlemesi (Python ile birebir benzetildi; "2:" ikinci slab sayfasıdır).



Şekil 15.22: İzlemenin sonu: birinci slab dolu (c, b, d, e), f ikinci slab'ın ilk nesnesi.

İzlemedeki üç gözlem SLUB'ın tasarımını özetler. c'nin a'nın yerine düşmesi rastlantı değildir. Son bırakılan nesnenin baytları büyük olasılıkla hâlâ CPU önbelleğindedir; onu hemen yeniden vermek önbellek ıskasını azaltır. İkincisi, yaygın durum birkaç komuttur: bir yükleme, bir yazma, bir dönüş. Gerçek SLUB bu adımı CPU başına yapılarla ve kilit yerine atomik bir karşılaştır-değiştir komutuyla yapar. Üçüncüsü, nadir durum, yani yeni sayfa gereken f, yavaştır ama doğrudur. Bütün nesneler aynı boyda olduğu için slab içinde dış parçalanma oluşmaz; bırakılan her boşluk bir sonraki istek için tam uygundur.

> **Sık yapılan hata**
>
> Bırakılmış bir nesneyi kullanmaya devam etmek. Bırakılan nesnenin ilk baytları artık serbest listenin işaretçisidir. Nesneye yazan hatalı kod listeyi bozar; bir sonraki ayırma rastgele bir adres döndürür. Bu tür hatalar çekirdek açıklarının sık kaynağıdır. Çekirdek bu yüzden işaretçiyi gizli bir değerle karıştırarak saklayabilir ve hata ayıklama seçenekleriyle bırakılan nesneleri bir desenle doldurup denetleyebilir.

15.8.7 Sayfa önbelleği, geri kazanım ve OOM

Linux'un bellek felsefesi kısa bir cümleyle özetlenir: boş bellek israftır. Kullanılmayan RAM **sayfa önbelleği** olarak dosya verisi için kullanılır. Bir dosya bir kez okununca sayfaları önbellekte kalır; ikinci okuma diske gitmez. `free -h` çıktısındaki “available” sütunu bu yüzden “free”den büyüktür: önbelleğin geri alınabilir kısmını da sayar. 16 GiB'lık bir makinede başta 5 GiB uygulama belleği, 8 GiB sayfa önbelleği ve 2 GiB boş bellek bulunsun (Tablo 15.12). Çekirdeğin kendisi ve tamponları kalan 1 GiB'ı tutar.

Aşama	Anonim	Önbellek	Boş	Takas	Çekirdeğin tepkisi
Başlangıç	5	8	2	0	—
Uygulama büyüdü	8,3	6	0,7	0	düşük düzey aşıldı: kswapd uyanır
Soğuk sayfalar gider	9	4,7	1,3	1	MGLRU, zswap → takas
Baskı sürüyor	11,5	3,2	0,3	2	min altı: doğrudan geri kazanım
OOM	4,6	2	8,4	2	`bellek_yiyen` (PID 7311) öldürüldü

Tablo 15.12: Bellek baskısı altında 16 GiB'lık bir makine (GiB).

Geri kazanım su düzeylerine göre kademeli çalışır (Şekil 15.23).

1. **Arka plan.** Boş bellek düşük su düzeyinin altına inince her düğümün `kswapd` çekirdek iş parçacığı uyanır. Temiz önbellek sayfalarını hemen bırakır; kirli olanları önce diske yazar. Boş bellek yüksek düzeye çıkınca yeniden uyur. Uygulama bu sırada fark edilir biçimde yavaşlamaz.
2. **Kurban seçimi.** Hangi sayfaların gideceğini 6.1 ile gelen **çok nesilli LRU** (MGLRU) belirler. Sayfalar erişim nesillerine ayrılır; en soğuk nesil önce gider. Önbellek sayfası atılabilir, çünkü verisi diskteki dosyada vardır. Soğuk anonim sayfaların ise dosyası yoktur. zswap onları önce RAM'de sı-

kıştırır, gerekirse takas alanına yazar.

3. **Doğrudan geri kazanım.** Ayırma hızı geri kazanımı geçer ve boş bellek min düzeyinin altına inerse, bellek isteyen süreç sayfa bekleyemez. Geri kazanımı kendisi yapar ve bu sırada takılır. Uygulamanın gecikmesi belirgin biçimde artar.
4. **OOM.** Hiçbir şey yetmezse OOM killer (bellek tükenince süreç öldüren mekanizma) devreye girer. En yüksek `oom_score`'lu süreci `SIGKILL` ile öldürür. Puan çoğunlukla sürecin bellek kullanımıyla orantılıdır. Yönetici `/proc/PID/oom_score_adj` dosyasına −1000 ile +1000 arası bir değer yazarak bir süreci korur ya da öne çıkarır. Bir cgroup'un bellek sınırı aşılırsa kurban yalnız o gruptan seçilir.



Şekil 15.23: Bellek baskısının son aşaması: OOM killer `bellek_yiyen`'i öldürdü, boş bellek 8,4 GiB'a çıktı.

Takas alanının ne kadar istekli kullanılacağı `vm.swappiness` ayarıyla belirlenir; varsayılan 60'tır. Değer düştükçe çekirdek anonim sayfaları takasa göndermek yerine önbellek sayfalarını bırakmayı yeğler.

Sık yapılan hata

`free` çıktısındaki küçük "free" değerini bellek sıkıntısı sanmak. Uzun süre çalışmış bir Linux makinesinde boş bellek hep azdır, çünkü RAM önbellekle doludur. Asıl gösterge "available" değeri, takas etkinliği ve `/proc/pressure/memory` dosyasındaki bellek baskısı (PSI) sayaçlarıdır.

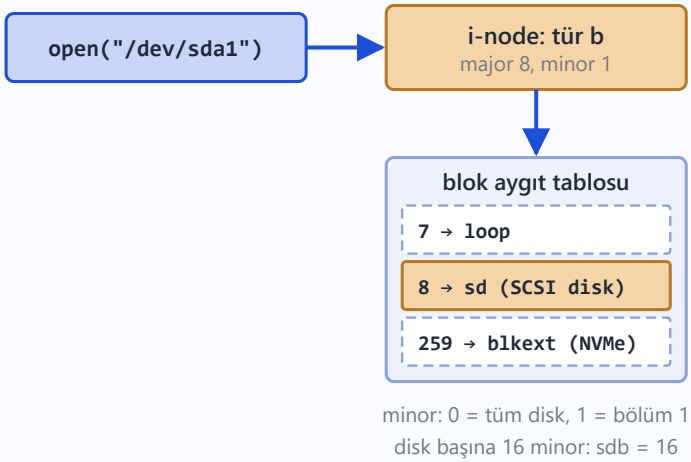
15.9 Giriş/çıkış

15.9.1 Aygıtlar dosyadır: major ve minor numaralar

Linux'ta disklere ve terminallere `/dev` dizinindeki **aygıt dosyaları** üzerinden erişilir. Uygulama bir diske de bir terminale de aynı `open`, `read` ve `write` çağrılarıyla ulaşır. `ls -l` bu dosyalar için boyut yerine iki sayı gösterir:

```
$ ls -l /dev     # seçilmiş satırlar
brw-rw---- 1 root disk   8, 0 sda
brw-rw---- 1 root disk   8, 1 sda1
brw-rw---- 1 root disk 259, 0 nvme0n1
crw-rw-rw- 1 root root   1, 3 null
crw-rw-rw- 1 root tty    5, 0 tty
```

Satırın ilk harfi aygıt türüdür. `c` **karakter aygıtıdır**: bayt akışı sunar, çoğunlukla önbelleksizdir; terminal, `/dev/null` ve fare böyledir. `b` **blok aygıtıdır**: sabit boylu bloklara rastgele erişim sunar, sayfa önbelleğini ve G/Ç çizelgeleyicisini kullanır. Boyut sütunundaki iki sayı major ve minor numaralardır (Şekil 15.24). **Major** numara sürücüyü seçer: 8, SCSI disk sürücüsü `sd`'dir; SATA ve USB diskler de ona bağlanır. **Minor** numara o sürücünün yönettiği birimi seçer. `sd` sürücüsü her diske 16 minor ayırır: 0 tüm disk, 1–15 bölümler. Bu yüzden `sda1` 8,1'dir; ikinci disk `sdb` 8,16'dan başlar, `sdb3` 8,19'dur. NVMe diskler 259 numaralı genişletilmiş blok majoruna (`blkext`) bağlanır.



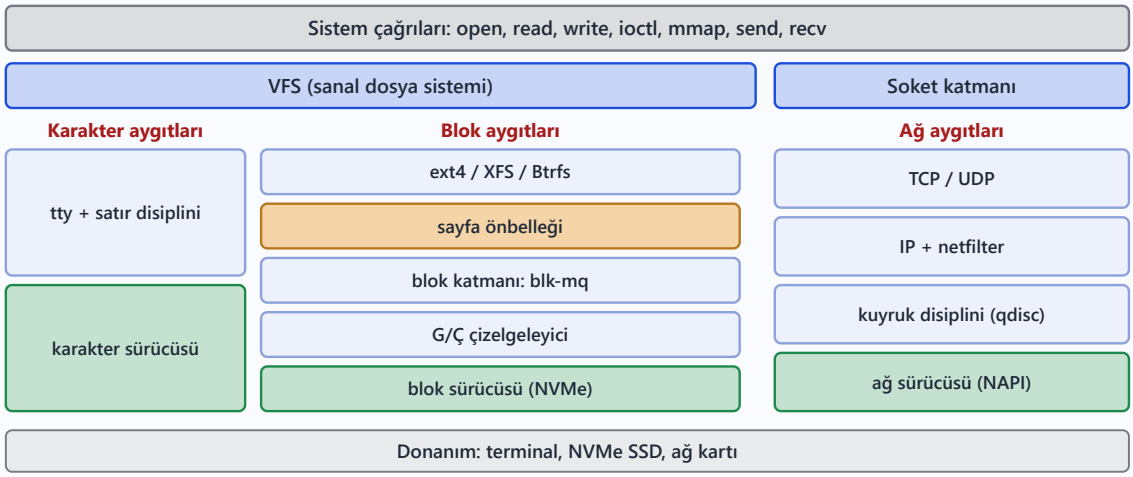
Şekil 15.24: `open("/dev/sda1")` : i-node'daki tür ve major numara sürücüyü, minor numara bölümü seçer.

Aygıt dosyası diskte veri tutmaz. i-node'u yalnız türü ve iki numarayı saklar. `open` bu numaralarla çekirdeğin aygıt tablosuna bakar ve sürücüyü bulur. Bugün `/dev`'deki düğümleri elle oluşturmak gerekmez. Çekirdek bir aygıt bulunca `devtmpfs` sözde dosya sisteminde düğümü kendisi açar; `udev`

izinleri ayarlar ve kalıcı adlar ekler (`/dev/disk/by-uuid/…`). Ağ arayüzleri (`eth0`, `wlan0`) ise `/dev`'de yoktur. Onlara soketlerle erişilir. "Her şey dosyadır" ilkesinin en bilinen istisnası budur.

15.9.2 G/Ç sisteminin yapısı

Üç aygıt türü çekirdekte üç ayrı yoldan iner (Şekil 15.25). Karakter aygıtları VFS'ten sonra tty katmanına ve satır disiplinine, oradan karakter sürücüsüne gider. Blok aygıtlarına giden yol dosya sisteminden (ext4, XFS, Btrfs), sayfa önbelleğinden, blok katmanından ve G/Ç çizelgeleyicisinden geçer. Ağ yolu soket katmanından başlar: TCP ya da UDP, IP ve paket süzgeci netfilter, kuyruk disiplini (qdisc) ve ağ sürücüsü. Ağ sürücüleri NAPI arayüzünü kullanır: yoğun trafikte her paket için kesme üretmek yerine kesmeleri kapatıp paketleri toplu yoklar.



Şekil 15.25: Linux G/Ç sistemi: karakter, blok ve ağ aygıtları için ayrı katmanlar.

Ağ yolunda veri `sk_buff` yapılarında taşınır. Bir `sk_buff` paketin verisini ve başlıklar için önden ayrılmış boşluğu tutar; her katman başlığını eklerken veriyi kopyalamaz, yalnız işaretçiyi kaydırır. XDP ve eBPF ile paketler daha sürücü düzeyinde, yığına girmeden süzülebilir ya da yönlendirilebilir. Klasik BSD'de de benzer bir üç yol ayrımı vardı. System V'in STREAMS adlı modül yığını ise Linux ana çekirdeğine hiç girmedi.

15.9.3 Sürücü arayüzü: `file_operations` ve terminal

Eski UNIX ders kitapları her sürücü için ayrı giriş noktalarını tablo hâlinde listeler: disk için `k_open`, `k_read`, terminal için `tty_open`, `tty_read` ve benzerleri. Linux bu giriş noktalarını tek bir fonksiyon işaretçisi yapısında toplar:

```
static const struct file_operations my_fops = {
    .owner          = THIS_MODULE,
    .open           = my_open,
    .release        = my_release,     /* close */
    .read           = my_read,
    .write          = my_write,
    .unlocked_ioctl = my_ioctl,
    .poll           = my_poll,
    .mmap           = my_mmap,
};
cdev_init(&cdev, &my_fops);  /* major:minor → fops */
```

Bir karakter sürücüsü bu yapıyı doldurur ve `cdev_init` ile bir major:minor aralığına bağlar. Uygulama aygıt dosyasında `read()` çağırınca VFS çağrıyı ilgili `struct file`'ın `f_op->read` işaretçisine yönlendirir. `close`'un karşılığı `.release`'tir ve yalnız son tanımlayıcı kapanınca çağrılır. Aygıta özgü komutlar `.unlocked_ioctl` ile gelir. `.poll`, aygıtın `select`, `poll` ve `epoll` ile beklenebilmesini sağlar. `.mmap` aygıt belleğini doğrudan sürecin adres alanına eşler. Bu, C dilinde nesne yönelimli tasarımdır: fonksiyon işaretçisi tablosu bir sanal yöntem tablosu gibi çalışır. Aynı desen VFS'in her yerinde tekrarlanır.

Terminal G/Ç'si **termios** arayüzüyle ayarlanır. `tcgetattr` ve `tcsetattr` terminal parametrelerini okur ve yazar. **Kanonik modda** (`ICANON`) satır disiplini satır düzenlemesini kendisi yapar: geri silme işlenir ve program satırı ancak Enter'dan sonra alır. Ham modda her tuş hemen iletilir; metin düzenleyiciler ve parola soran programlar bu modu kullanır. `ECHO` bayrağı yazılanın ekrana yansıtılıp yansıtılmayacağını belirler. `cfsetispeed` ve `cfsetospeed` giriş ve çıkış baud hızını ayarlar. Bu ayar bugün çoğunlukla gerçek seri portlar için anlamlıdır; terminal pencereleri sözde terminaldir (pty).

15.9.4 Blok katmanı: blk-mq

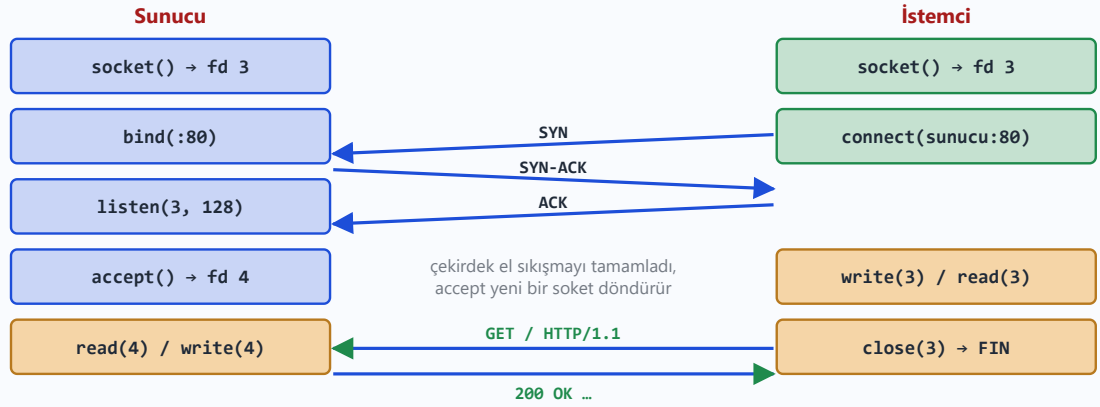
Tek kuyruklu eski blok katmanı, saniyede yüzlerce işlem yapan döner diskler için tasarlanmıştı. Her istek tek bir kuyruğa ve tek bir kilide uğruyordu. Saniyede milyonlarca işlem yapabilen NVMe SSD'lerde bu kilit darboğaz oldu. Bugünkü blok katmanı **blk-mq** çok kuyrukludur. Her CPU isteklerini kendi yazılım kuyruğuna yazar. İstekler isteğe bağlı bir G/Ç çizelgeleyiciden geçip aygıtın donanım kuyruklarına dağıtılır. NVMe denetleyicileri çok sayıda donanım kuyruğu sunar; çoğunlukla her CPU'ya bir kuyruk düşer ve CPU'lar arasında kilit paylaşımı kalmaz.

Dört çizelgeleyici seçeneği vardır. `none` hiç sıralama yapmaz; hızlı NVMe aygıtlarında çoğunlukla en iyisidir. `mq-deadline` okumaları ve yazmaları son tarihlerle sıralar, hiçbir isteğin aç kalmamasını sağlar. `bfq` (bütçe adil kuyruklama) süreçler arasında adil bant genişliği paylaştırır; masaüstündeki yavaş disklerde etkileşimi korur. `kyber` hedef gecikmelere göre kuyruk derinliğini ayarlar. Etkin çizelgeleyici aygıt başına `/sys/block/<aygıt>/queue/scheduler` dosyasından okunur; köşeli parantez içindeki seçilidir. Dosyaya bir ad yazmak çizelgeleyiciyi değiştirir. Asenkron G/Ç'nin yeni arayüzü `io_uring` de bu katmanın üstüne kurulur. Uygulama ile çekirdek, paylaşılan iki halka tampon üzerinden istek ve sonuç alışverişi yapar. Sistem çağrısı sayısı böylece azalır.

15.9.5 Ağ: soketler ve bağlantı kurulumu

Soket, ağ iletişiminin uç noktasıdır ve bir dosya tanımlayıcısıyla temsil edilir. Bir TCP bağlantısının kuruluşu iki taraftaki çağrılarla izlenebilir (Şekil 15.26).

1. **Sunucu hazırlanır.** `socket()` bir soket oluşturur; en küçük boş tanımlayıcı olan 3'ü alır. `bind` soketi 80 numaralı porta bağlar. `listen(3, 128)` soketi dinleme durumuna geçirir ve bekleyen bağlantılar için bir kuyruk açar.
2. **Bekleme.** Sunucu `accept`'te bloke olur. İstemci kendi soketini oluşturur.
3. **El sıkışma.** İstemci `connect` çağırır. İstemci çekirdeği SYN gönderir, sunucu çekirdeği SYN-ACK ile yanıtlar, istemci ACK ile tamamlar. Bu üçlü el sıkışmayı iki çekirdek yürütür; uygulamalar paketleri hiç görmez.
4. **Yeni soket.** Bağlantı kurulunca `accept` yeni bir soket döndürür (fd 4). fd 3 dinlemeyi sürdürür. Böylece tek bir dinleyen soket binlerce bağlantı soketi doğurabilir.
5. **Veri ve kapanış.** Veri `write` ve `read` ile akar. Çekirdek veriyi `sk_buff`'lara koyar, TCP/IP başlıklarını ekler; sürücü paketi kartın halka tamponuna yazar. `close` karşı tarafa FIN gönderir.



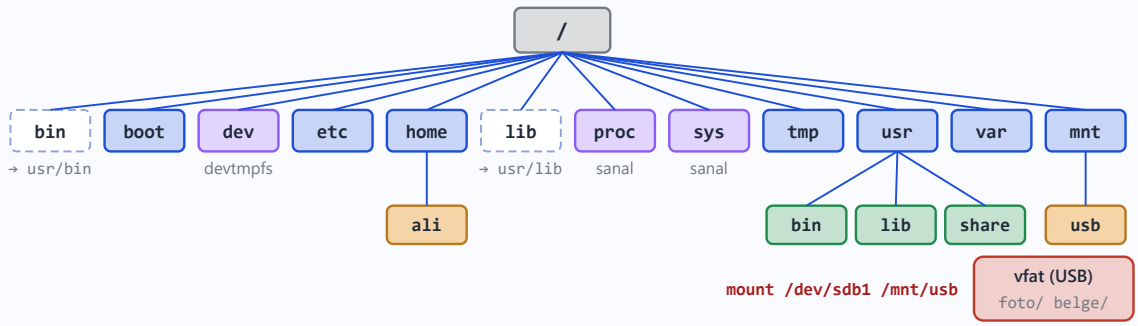
Şekil 15.26: TCP soketleriyle istemci–sunucu bağlantısı: `socket`, `bind`, `listen`, `accept`, `connect`, üçlü el sıkışma, veri ve FIN.

Çok istemcili bir sunucu her bağlantı için bir iş parçacığı açabilir. Binlerce bağlantıda bu pahalılaşır. Linux'un yanıtı `epoll`'dur. Sunucu ilgilendiği soketleri bir kez kaydeder. Sonra tek bir `epoll_wait` çağrısıyla yalnız hazır olan soketlerin listesini alır. Maliyet toplam bağlantı sayısıyla değil, hazır soket sayısıyla orantılıdır.

15.10 Dosya sistemleri

15.10.1 Dizin ağacı ve bağlama

Windows'taki sürücü harflerinin aksine Linux'ta tek bir dizin ağacı vardır (Şekil 15.27). Başka bir dosya sistemi ağacın herhangi bir dizinine **bağlanır** (*mount*). `mount /dev/sdb1 /mnt/usb` komutundan sonra USB belleğin kök dizini `/mnt/usb`'de görünür. Bağlama noktasının eski içeriği bağlama süresince gizlenir.



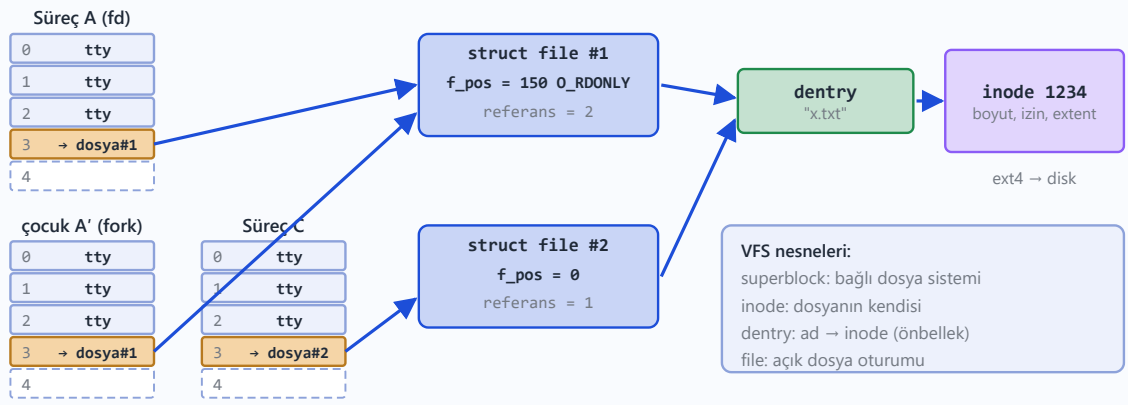
Şekil 15.27: Linux dizin ağacı ve `/mnt/usb` noktasına bağlanmış bir vfat dosya sistemi.

Dizinlerin rolleri Dosya Sistemi Hiyerarşisi Standardı (FHS) ile belirlenir. `/usr` salt okunur, paylaşılabilir yazılımı tutar: komutlar `/usr/bin`'de, paylaşımlı kütüphaneler `/usr/lib`'de, çekirdek modülleri `/usr/lib/modules`'te. `/etc` metin biçimindeki yapılandırma dosyalarıdır. `/var` değişken veriyi (günlükler, kuyruklar, önbellekler), `/home` kullanıcı dizinlerini, `/root` yöneticinin dizinini tutar. `/boot`'ta çekirdek ve initramfs durur. `/tmp` geçici dosyalar içindir ve çoğu dağıtımda RAM'deki tmpfs'tir. `/dev`, `/proc` ve `/sys` diskte değil, çekirdeğin bellekte ürettiği sözde dosya sistemleridir. Çoğu dağıtım "usrmerge" ile `/bin`, `/sbin` ve `/lib`'i `/usr` altına taşıdı. Eski yollar, uyumluluk için sembolik bağ olarak kaldı.

15.10.2 VFS nesneleri

Linux onlarca dosya sistemini aynı çağrılarla kullanır: ext4, XFS, Btrfs, FAT, NFS, procfs. Bunu **sanal dosya sistemi** (VFS) sağlar. VFS dört nesne türü tanımlar (Şekil 15.28):

- **superblock**: bağlanmış bir dosya sistemi. Blok boyu, kök i-node ve dosya sistemi işlemleri buradadır.
- **inode**: dosyanın kendisi. Boyut, izinler, zamanlar ve verinin yeri. Bellekteki `struct inode`, diskteki i-node'un bir kopyasıdır.
- **dentry**: bir ad ile bir i-node arasındaki bağ. `"x.txt" → inode 1234`. Yol çözümlemesini hızlandırmak için önbellekte tutulur.
- **file**: açık bir dosya oturumu. Dosya konumu (`f_pos`), açılış bayrakları ve referans sayısı.



Şekil 15.28: Dosya tanımlayıcı tablosu, `struct file`, dentry ve inode ilişkisi. A ile çocuğu A′ aynı `struct file`'ı paylaşır; C aynı dosyayı ayrıca açmıştır.

Şekildeki durum üç açık dosya tablosu düzeyini somutlaştırır. A süreci `x.txt`'yi açtı ve 150 bayt okudu. Sonra `fork` etti. Çocuk A′'nün 3 numaralı tanımlayıcısı aynı `struct file #1`'i gösterir; referans sayısı 2'dir ve konum ortaktır. A′ 10 bayt daha okursa A'nın konumu da 160 olur. C süreci aynı dosyayı kendisi açtı. Onun `struct file #2`'si ayrıdır ve konumu 0'dır. İki `struct file` da aynı dentry'yi, dentry de aynı inode'u gösterir. Sürecin tanımlayıcı tablosu `task_struct → files_struct` içindedir.

Her nesnenin bir işlem tablosu vardır: `super_operations`, `inode_operations`, `file_operations`, `dentry_operations`. VFS bir işi yaparken bu tablolardaki fonksiyon işaretçisini çağırır. ext4'te `read` ext4'ün okuma işlevine, NFS'te ağ üzerinden bir RPC'ye, procfs'te çekirdek verisini o anda metne çeviren bir işleve gider. Çağıran program farkı görmez.

15.10.3 Dosya ve dizin çağrıları, `stat`

Dosya sisteminin sistem çağrısı arayüzü küçüktür (Tablo 15.13). Birkaç ayrıntı sık karıştırılır. `rmdir` yalnız boş dizini siler. `rename` atomiktir: hedef ad varsa tek adımda yer değiştirir; bir dosyayı güvenle güncellemenin yolu, yeni içeriği geçici dosyaya yazıp `rename` etmektir. `opendir`, `readdir` ve `closedir` sistem çağrısı değildir; `getdents64` üzerine kurulu kütüphane işlevleridir.

Çağrı	İşlev
`open`, `creat`, `close`	Aç / oluştur, fd döndür; tanımlayıcıyı kapat
`read`, `write`, `lseek`	Konumdan oku / yaz; konumu taşı
`stat`, `fstat`, `lstat`	Bilgi al (`lstat`: bağın kendisi)
`pipe`, `dup2`, `fcntl`	Boru oluştur; fd kopyala; kilit ve bayraklar
`mkdir`, `rmdir`	Dizin oluştur / sil (boşsa)
`link`, `unlink`, `symlink`	Ad ekle / sil; sembolik bağ oluştur
`rename`, `chdir`	Taşı (atomik); çalışma dizinini değiştir

Çağrı	İşlev
`getdents64`	Dizin girdilerini oku

Tablo 15.13: Dosya ve dizin sistem çağrıları.

`stat` bir yolu çözer, i-node'u okur ve alanlarını `struct stat`'a kopyalar (Şekil 15.29). `st_ino` i-node numarası, `st_mode` tür ve izin bitleri, `st_nlink` katı bağ sayısı, `st_size` bayt cinsinden boyuttur. `st_blocks` diskte ayrılan alanı 512 baytlık birimlerle verir; seyrek bir dosyada `st_size`'tan çok küçük olabilir. Üç zaman damgası vardır: `st_atime` son erişim, `st_mtime` son içerik değişikliği, `st_ctime` son i-node değişikliğidir. `st_ctime` oluşturma zamanı *değildir*; `chmod` ya da `link` da onu günceller. Oluşturma zamanı yalnız daha yeni `statx()` çağrısıyla, `stx_btime` alanından okunabilir.



Şekil 15.29: `stat`: yol çözümlenir, i-node okunur, alanlar `struct stat`'a kopyalanır.

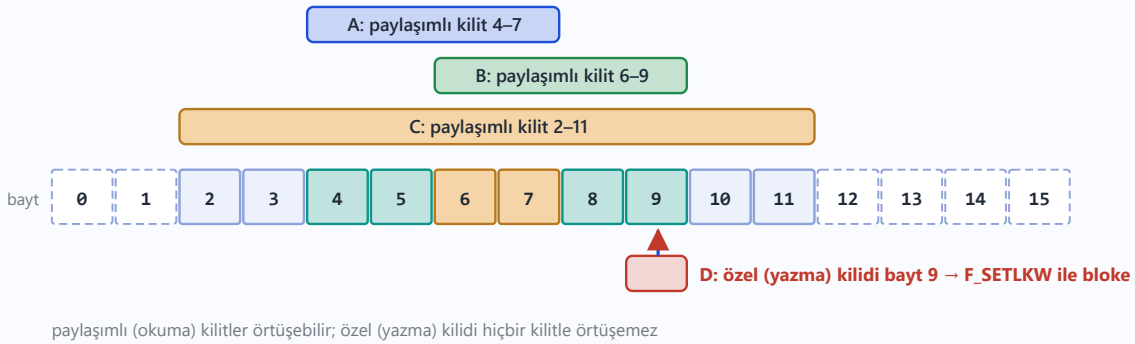
15.10.4 Dosya kilitleme

Aynı dosyayı paylaşan süreçler `fcntl` ile bayt aralıklarını kilitleyebilir. Kilit bir `struct flock` ile tanımlanır: tür (`F_RDLCK` paylaşımlı, `F_WRLCK` özel), başlangıç (`l_start`) ve uzunluk (`l_len`). Paylaşımlı kilitler birbiriyle örtüşebilir; özel kilit hiçbir kilitle örtüşemez. Bu, okuyucu–yazar kilidinin bayt düzeyindeki karşılığıdır.

Örnek 15.8

16 baytlık bir dosyada sırayla A 4–7'ye, B 6–9'a, C 2–11'e paylaşımlı kilit koyuyor. Sonra D yalnız 9. bayta özel kilit istiyor. Hangi istekler kabul edilir?

1. A: `F_RDLCK`, `l_start` = 4, `l_len` = 4. Dosyada kilit yok; kabul.
2. B: 6–9 aralığı A ile 6–7'de örtüşür. İkisi de paylaşımlı; kabul.
3. C: 2–11, A ve B'nin tamamını kapsar. Yine paylaşımlı; kabul. 6 ve 7. baytlar artık üç kilitle kaplıdır.
4. D: 9. bayt B ve C'nin aralığındadır. Özel kilit paylaşımlı kilitlerle çakışır. `F_SETLK` ile istenirse çağrı beklemeden hata döndürür (`EAGAIN` ya da `EACCES`). `F_SETLKW` ile istenirse D, B ve C kilitlerini kaldırana dek bekler.

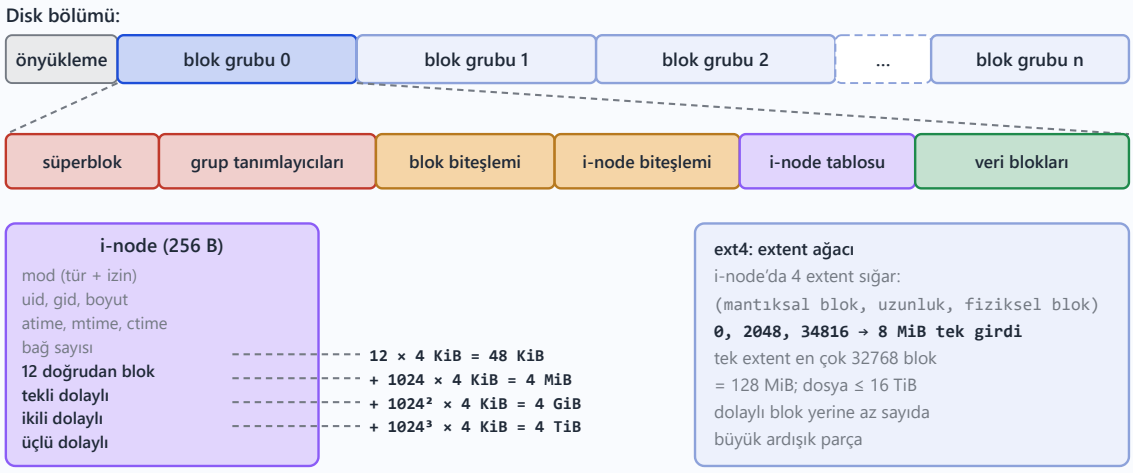


Şekil 15.30: fcntl bayt aralığı kilitleri: üç paylaşımlı kilit ve bayt 9 için bekleyen özel kilit isteği.

`F_SETLKW` ile bekleyen iki süreç birbirinin kilidini bekleyebilir. Çekirdek bekleme ilişkilerini izler; bir döngü fark ederse bekleyen çağrıya `EDEADLK` döndürür. Bu, kilitlenme tespitinin gerçek bir örneğidir. Bu kilitler **tavsiye niteliğindedir** (*advisory*): kilidi sormayan bir süreç dosyaya yine yazabilir. Bütün dosya için daha basit `flock()` vardır. `fcntl` kilitlerinin önemli bir kusuru, sürece bağlı olmalarıdır. Süreç dosyanın herhangi bir tanımlayıcısını kapatınca bütün kilitleri düşer. Aynı süreçteki iş parçacıkları da kilitleri birbirinden ayırt edemez. Açık dosya tanımına bağlı **OFD kilitleri** (`F_OFD_SETLK`) bu kusuru giderir.

15.10.5 ext2 ve ext4'ün disk düzeni

Klasik UNIX dosya sisteminin düzeni basitti: önyükleme bloğu, süperblok, i-node tablosu ve veri blokları. Bu düzende bir dosyanın i-node'u diskin başında, verisi ise herhangi bir yerde olabilirdi; disk kolu ikisi arasında sürekli gidip gelirdi. ext2 diski **blok gruplarına** böldü (Şekil 15.31). Her grubun kendi blok biteşlemi, i-node biteşlemi, i-node tablosu ve veri blokları vardır. Çekirdek bir dosyanın i-node'u ile verisini aynı gruba, bir dizinin dosyalarını da dizinin grubuna yakın koymaya çalışır. Süperblok ve grup tanımlayıcıları bazı gruplarda yedeklenir; birinci kopya bozulursa yedekten kurtarılır.



Şekil 15.31: ext2/ext4 disk düzeni, blok grupları, i-node'un blok işaretçileri ve ext4 extent'leri.

ext2'nin i-node'u 12 doğrudan blok işaretçisi ile tekli, ikili ve üçlü dolaylı işaretçi taşır. 4 KiB'lık bloklarla bir dolaylı blok 1024 işaretçi tutar. Erişilebilen alan katlanarak büyür: 12 × 4 KiB = 48 KiB, + 1024 × 4 KiB = 4 MiB, + 1024^2 × 4 KiB = 4 GiB, + 1024^3 × 4 KiB = 4 TiB. Dolaylı bloklarla teorik sınır böylece yaklaşık 4 TiB'tır. ext2'de gerçek sınır ise 2 TiB'tır: i-node'daki 32 bitlik `i_blocks` alanı ayrılan alanı 512 baytlık birimlerle sayar ve 2^{32} × 512 B = 2 TiB'ta tıkanır. Kök dizinin i-node numarası her zaman 2'dir; 1 eskiden bozuk blokların listesine ayrılmıştı.

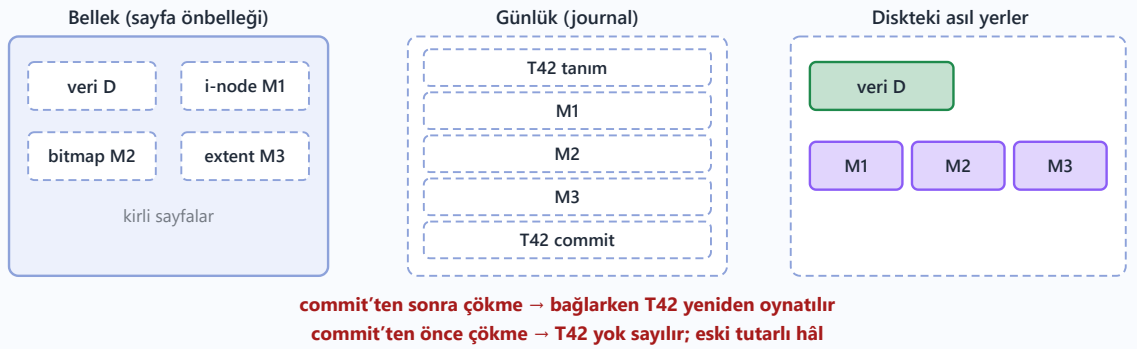
ext4 büyük dosyalar için dolaylı bloklar yerine **extent** kullanır. Bir extent ardışık bir blok aralığını üç sayıyla tanımlar: mantıksal blok, uzunluk ve fiziksel blok. `(0, 2048, 34816)` extent'i dosyanın ilk 2048 bloğunun, yani 8 MiB'ının diskte 34816. bloktan başlayıp ardışık durduğunu söyler. Dolaylı bloklarla aynı bilgi 2048 işaretçi, iki dolaylı blok demekti. i-node'a dört extent sığar; daha çoğu gerekirse extent'ler i-node'dan başlayan bir ağaca yerleşir. Tek bir extent en çok 32.768 blok, yani 4 KiB'lık bloklarla 128 MiB kapsar. 4 KiB blokla bir dosya en çok 16 TiB olabilir. ext4 ayrıca gecikmeli ayırma kullanır: veri bloklarının yerine, veri diske yazılacağı ana kadar karar vermez; böylece büyük ardışık extent'ler seçebilir.

15.10.6 ext4'te günlükleme

Bir `write()` ve ardından `fsync()` birkaç bloğu değiştirir. Verinin kendisi (D) yazılır; yanında üst veriler değişir: i-node (M1), blok biteşlemi (M2) ve extent (M3). Bu blokların bir kısmı yazılıp öbürleri yazılmadan elektrik kesilirse dosya sistemi tutarsız kalır. Örneğin biteşlem bir bloğu kullanımda gösterir ama hiçbir i-node o bloğu göstermez. ext4 bu soruna **günlükleme** ile çözüm bulur. Günlüğü JBD2 katmanı yönetir (Şekil 15.32).

1. **Veri önce.** Varsayılan `data=ordered` modunda D önce asıl yerine yazılır. Böylece üst veri hiçbir zaman çöp içeren bir bloğu göstermez.
2. **Günlük işlemi.** M1, M2 ve M3 günlüğe tek bir işlem (T42) olarak yazılır: bir tanım bloğu, üst veri bloklarının kopyaları ve en sonda bir **commit** kaydı. Commit kaydı diske ulaştığı an işlem kalıcıdır.

3. **Checkpoint.** Üst veri blokları sonra, acele etmeden asıl yerlerine yazılır. Günlükteki alan yeniden kullanılır; günlük dairesel bir tampondur.
4. **Kurtarma.** Çökmeden sonra bağlama sırasında tam bir fsck taraması yapılmaz. Günlük okunur. Commit kaydı olan işlemler yeniden oynatılır. Commit'i olmayan yarım işlemler atılır ve dosya sistemi eski tutarlı hâlinde kalır. Bu iş saniyeler sürer.



Şekil 15.32: ext4 sıralı günlükleme: veri, günlük işlemi T42, commit ve checkpoint; çökme commit'ten önce mi sonra mı olduğuna göre iki sonuç.

Öteki iki mod farklı ödünleşimler sunar. `data=journal` veriyi de günlüğe yazar: en güvenlisidir ama her veri bloğu iki kez yazıldığı için en yavaşıdır. `data=writeback` veri ile üst veri arasında sıra güvencesi vermez: hızlıdır ama çökmeden sonra bir dosya eski ya da çöp içerikli bloklar gösterebilir. ext4 varsayılan olarak yaklaşık beş saniyede bir commit yapar. `fsync` çağrılmayan bir ekleme, commit'ten önce elektrik kesilirse tümüyle kaybolabilir; ama `data=ordered` sayesinde dosya hiçbir zaman çöp göstermez. XFS de üst veri günlüğü kullanır. Btrfs ise günlük yerine yazınca kopyalamayla tutarlı kalır.

Sık yapılan hata

Günlüklemenin veriyi koruduğunu sanmak. Varsayılan modda günlük yalnız üst veriyi korur ve yalnız *tutarlılığı* garanti eder. Uygulamanın son yazdıkları `fsync` olmadan kaybolabilir. Verinin diske ulaştığından emin olmak isteyen program `fsync` çağırmalıdır.

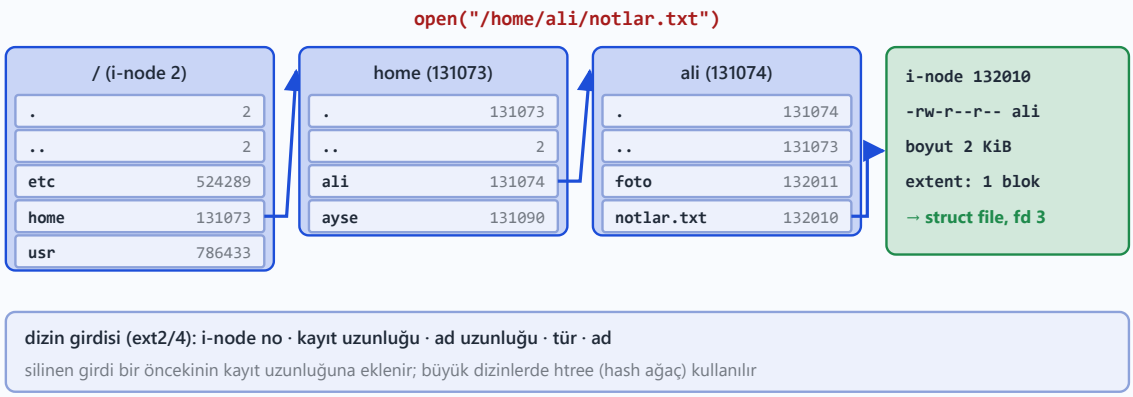
15.10.7 Dizinler ve yol çözümleme

ext2/ext4'te bir dizin, girdilerden oluşan sıradan bir dosyadır. Her girdi i-node numarasını, kayıt uzunluğunu, ad uzunluğunu, dosya türünü ve adın kendisini tutar. Silinen bir girdi yok edilmez; alanı bir önceki girdinin kayıt uzunluğuna eklenir. Binlerce girdili dizinlerde doğrusal arama yavaşlar. ext4 bu dizinlerde adların hash değerine göre sıralanmış bir ağaç (htree) kullanır.

Örnek 15.9

`open("/home/ali/notlar.txt")` nasıl çözülür? (Şekil 15.33)

1. Yol mutlaktır; arama kökten başlar. Kök dizinin i-node'u (2) okunur. Veri bloklarındaki girdilerde `home` aranır: 131073. Kök için x (arama) izni denetlenir.
2. `home` dizininin (131073) verisinde `ali` aranır: 131074.
3. `ali` dizininde (131074) `notlar.txt` aranır: 132010. Göreli bir yol olsaydı arama sürecin çalışma dizininden başlardı.
4. i-node 132010 okunur. İzinler denetlenir; okuma izni varsa bir `struct file` oluşturulur ve en küçük boş tanımlayıcı, örneğin 3, döner.



Şekil 15.33: `/home/ali/notlar.txt` yolunun çözümlenmesi: i-node 2 → 131073 → 131074 → 132010.

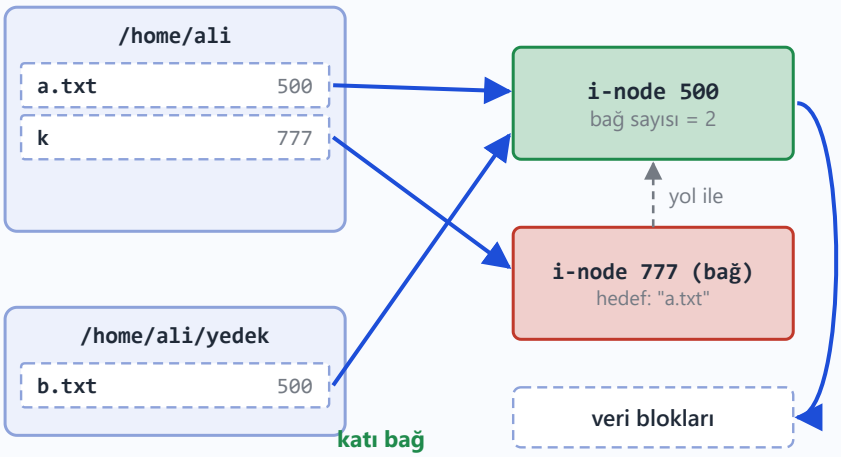
i-node numaraları rastgele değildir. 131073 ile 132010'un yakınlığı, ext4'ün bir dizinin dosyalarını aynı blok grubuna koyma çabasını yansıtır. Her adım bir disk okuması gerektirseydi yol çözümleme çok yavaş olurdu. **dentry önbelleği** (dcache) ad → i-node eşlemelerini bellekte tutar. Sık kullanılan yollar diske hiç gitmeden çözülür. dcache olumsuz sonuçları da saklar: "bu dizinde böyle bir ad yok" bilgisi de önbelleğe girer. Var olmayan bir dosyayı her açılışta arayan bir program böylece diski yormaz.

15.10.8 Kod izleme: bağ sayısı ve `unlink`

Linux'ta bir dosyanın adı ile kendisi ayrı şeylerdir. Ad bir dizin girdisidir; dosya ise i-node'dur. İki tür bağ bu ayrımı kullanır (Şekil 15.34):

```
$ ln a.txt yedek/b.txt     # katı
$ ln -s a.txt k            # sembolik
$ ls -li
500 -rw-r--r-- 2 ali ali 9 a.txt
777 lrwxrwxrwx 1 ali ali 5 k -> a.txt
```

Katı bağ aynı i-node'a verilmiş ikinci bir addır. `a.txt` ile `yedek/b.txt` aynı i-node'u (500) gösterir ve bağ sayısı 2'dir. **Sembolik bağ** ise ayrı bir i-node'dur (777); içeriği yalnız hedefin yoludur. `ls -l` onu `l` türüyle ve `k -> a.txt` biçiminde gösterir. Boyutu 5'tir, çünkü `"a.txt"` beş karakterdir. Katı bağ dosya sistemi sınırını aşamaz, çünkü i-node numarası yalnız kendi dosya sisteminde anlamlıdır. Dizinlere de konamaz, çünkü ağaçta döngü oluşturabilir. Sembolik bağ başka bir dosya sistemine ve dizinlere işaret edebilir. Bedeli, hedef silinince bağın **sarkmasıdır**.



Şekil 15.34: Katı bağ aynı i-node'u gösteren ikinci bir addır; sembolik bağ hedefin yolunu içeren ayrı bir i-node'dur.

Bir i-node iki sayaç taşır. **Bağ sayısı** diskteki dizin girdilerini, **açık dosya sayısı** bellekteki referansları sayar. Veri ancak ikisi de sıfıra inince serbest kalır. Şu program bu iki sayacı adım adım değiştirir:

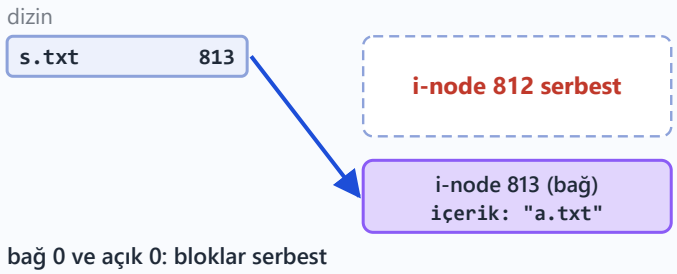
```
int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
write(fd, "veri", 4);
link("a.txt", "b.txt");      /* katı bağ */
symlink("a.txt", "s.txt");   /* sembolik */
unlink("a.txt");
int r1 = open("b.txt", O_RDONLY);
int r2 = open("s.txt", O_RDONLY);
unlink("b.txt");
close(r1);
close(fd);
```

Satır	Dizin girdileri	812 bağ	812 açık	Sonuç
1–2 `open`, `write`	a.txt → 812	1	1	fd = 3
3 `link`	a.txt, b.txt → 812	2	1	veri kopyalanmaz
4 `symlink`	+ s.txt → 813 ("a.txt")	2	1	812 değişmez
5 `unlink("a.txt")`	b.txt, s.txt	1	1	veri b.txt'den erişilebilir
6 `open("b.txt")`	b.txt, s.txt	1	2	r1 = 4

Satır	Dizin girdileri	812 bağ	812 açık	Sonuç
7 `open("s.txt")`	b.txt, s.txt	1	2	r2 = −1, `ENOENT`
8 `unlink("b.txt")`	s.txt	0	2	ad yok ama veri duruyor
9 `close(r1)`	s.txt	0	1	fd 3 hâlâ çalışır
10 `close(fd)`	s.txt	0	0	i-node ve bloklar serbest

Tablo 15.14: Bağ ve açık dosya sayaçları (Python'da dizin ve i-node benzetimiyle doğrulandı).

Satırların anlamı şöyledir. `open` yeni bir i-node (812) ayırır ve dizine `a.txt → 812` girdisini ekler. `link` yalnız yeni bir dizin girdisi yazar; a.txt ile b.txt eşittir ve hangisinin "asıl" olduğu sorusunun yanıtı yoktur. `symlink` ayrı bir i-node (813) oluşturur ve 812'ye dokunmaz. `unlink("a.txt")` bir adı siler ve bağ sayısını 1'e indirir; adı "silmek" dosyayı silmek değildir. b.txt sorunsuz açılır ve en küçük boş tanımlayıcı 4 döner. s.txt açılınca çekirdek içindeki yolu, `"a.txt"` 'yi izler. O ad artık yoktur: sembolik bağ sarkmıştır ve `open` −1 döndürür. `unlink("b.txt")` son adı da siler. Bağ sayısı 0'dır, ama iki tanımlayıcı açık olduğu için i-node ve veri henüz serbest bırakılmaz. `ls` dosyayı artık göstermez, yine de fd 3 üzerinden okuma ve yazma çalışır. Son tanımlayıcı kapanınca iki sayaç da sıfıra iner ve bloklar serbest kalır.



Şekil 15.35: İzlemenin sonu: i-node 812 serbest, yalnız sarkan sembolik bağ s.txt (813) kaldı.

Geçici dosyaların bilinen kalıbı bu davranışa dayanır. Program dosyayı açar ve hemen `unlink` eder. Dosya hiçbir dizinde görünmez, başka bir süreç onu adıyla bulamaz. Program çöktüğünde bile çekirdek tanımlayıcıları kapatır ve alan kendiliğinden geri döner. Linux bu kalıp için `O_TMPFILE` bayrağını da sunar: dosya baştan adsız oluşur.

Sık yapılan hata

Diskte yer açmak için büyük bir günlük dosyasını silip alanın hemen geri gelmesini beklemek. Dosyaya yazan süreç tanımlayıcıyı açık tuttuğu sürece bloklar serbest kalmaz; `df` değişmez. Böyle silinmiş ama açık dosyalar `lsof +L1` ile bulunur. Doğru yol, süreci yeniden başlatmak ya da dosyayı silmek yerine boşaltmaktır.

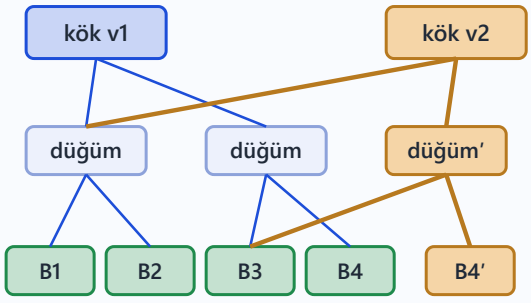
15.10.9 ext4'ün ötesi

Linux'un desteklediği dosya sistemleri farklı ihtiyaçlara yanıt verir (Tablo 15.15). ext4 olgun, hızlı ve onarım aracı güvenilir olduğu için çoğu dağıtımda varsayılandır. XFS paralel G/Ç'de ve çok büyük dosyalarda güçlüdür; RHEL'in varsayılanıdır. F2FS flash belleğe uygun, günlük yapılı (*log-structured*) bir tasarımdır ve Android telefonlarda yaygındır.

Dosya sistemi	Öne çıkan özellik	Tipik kullanım
ext4	Günlükleme, extent, olgun ve hızlı fsck	Çoğu dağıtımda varsayılan
XFS	Paralel G/Ç, büyük dosyalar, gecikmeli ayırma	RHEL varsayılanı, büyük sunucular
Btrfs	Yazınca kopyalama, anlık görüntü, sağlama toplamı, sıkıştırma, alt birimler	Fedora ve openSUSE varsayılanı
F2FS	Flash'a uygun günlük yapılı tasarım	Android telefonlar
tmpfs	Tamamen RAM'de (sayfa önbelleği + takas)	`/tmp`, `/run`, `/dev/shm`
NFS, SMB	Ağ dosya sistemleri	Paylaşılan ev dizinleri

Tablo 15.15: Linux'ta sık kullanılan dosya sistemleri.

Btrfs'in tutarlılık yöntemi günlükten farklıdır (Şekil 15.36). Hiçbir bloğun üzerine yazılmaz. B4 bloğu değişince yeni içerik boş bir yere, B4′'ye yazılır. B4'ü gösteren düğüm de değişmek zorundadır; onun da yeni bir kopyası (düğüm′) yazılır. Bu zincir köke kadar sürer ve yeni bir kök (v2) oluşur. Son adım, süperbloğun yeni kökü göstermesidir; bu tek bir atomik yazmadır. Çökme bu yazmadan önce olursa eski kök (v1) bütünüyle geçerlidir. Eski kökü saklamak bedavaya bir **anlık görüntü** (*snapshot*) verir: v1 ile ulaşılan ağaç, değişiklikten önceki dosya sistemidir.



Şekil 15.36: Btrfs'te yazınca kopyalama: B4 değişince yalnız kökten B4'e giden yol kopyalanır; eski kök anlık görüntü olur.

VFS sayesinde bütün bu dosya sistemleri aynı sistem çağrılarıyla kullanılır. Hangi dizinde hangi dosya sisteminin bağlı olduğunu `findmnt` ya da `df -T` gösterir.

15.10.10 `/proc` ve `/sys`

Linux çekirdeğin iç durumunu da dosya olarak sunar. Bu dosyalar diskte yoktur; okunduğu anda çekirdek tarafında üretilir.

```
$ cat /proc/loadavg
0.42 0.37 0.30 2/1234 98765
$ grep -E "MemTotal|MemAvailable" /proc/meminfo
MemTotal:       16303452 kB
MemAvailable:    9876544 kB
$ ls /proc/self
cmdline  cwd  environ  exe  fd  maps  status  …
$ cat /sys/block/nvme0n1/queue/scheduler
[none] mq-deadline kyber bfq
$ sysctl vm.swappiness        # = /proc/sys/vm/swappiness
vm.swappiness = 60
```

`/proc` (procfs) iki tür bilgi taşır. Her süreç için bir dizin vardır: `/proc/PID/status` durumu ve bellek kullanımını, `maps` VMA'ları, `fd/` açık tanımlayıcıları gösterir. `/proc/self` , okuyan sürecin kendi dizinidir. Çekirdek istatistikleri de buradadır. `/proc/loadavg` çıktısındaki ilk üç sayı son 1, 5 ve 15 dakikanın yük ortalamalarıdır. `2/1234` çalışan ve toplam görev sayısını verir; son sayı en son atanan PID'dir. Örnekteki `/proc/meminfo` satırları, 15,55 GiB'lık toplam belleğin 9,42 GiB'ının (%60,6) kullanılabilir olduğunu söyler. `ps` , `top` ve `free` verilerini buradan okur. `/proc/sys` altındaki dosyalar ayarlanabilir çekirdek parametreleridir. `sysctl vm.swappiness` aslında `/proc/sys/vm/swappiness` dosyasını okur.

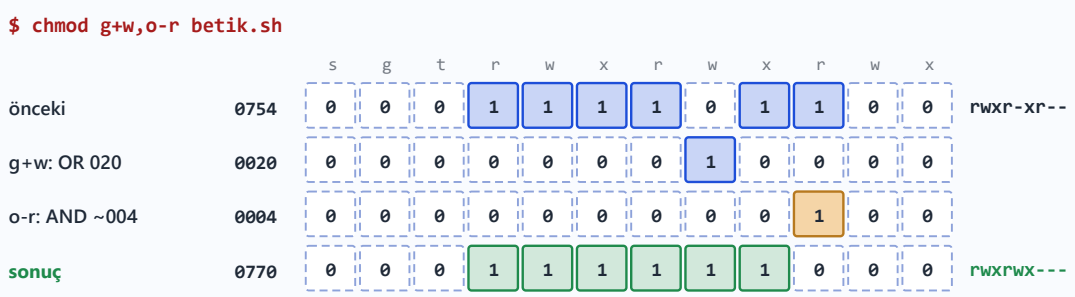
`/sys` (sysfs) çekirdeğin aygıt modelini gösterir. Her aygıt, sürücü ve veri yolu bir dizindir. Her dosya tek bir değer tutar. `/sys/block/nvme0n1/queue/scheduler` dosyası blk-mq çizelgeleyicisini gösterir: köşeli parantez içindeki `none` seçilidir. `/sys/fs/cgroup` ise cgroup v2'nin arayüzüdür; kaynak sınırları bu dizindeki dosyalara yazılarak konur. Süreç ağacını, tanımlayıcıları ve bellek haritalarını etkileşimli olarak incelemek için Linux Süreç Gezgini kullanılabilir.

15.11 Güvenlik

15.11.1 `chmod` ve `umask` hesapları

Linux'ta her dosyanın sahip, grup ve diğerleri için üçer izin biti vardır: r okuma, w yazma, x çalıştırma. Bunların önünde üç özel bit durur: setuid, setgid ve yapışkan bit. On iki bit dört sekizli rakamla yazılır. Her sekizli rakam üç bittir: 7 = 111 = rwx, 5 = 101 = r-x, 4 = 100 = r--. `chmod 754 betik.sh` böylece `rwxr-xr--` verir.

`chmod` 'un sembolik biçimi mevcut bitleri değiştirir. `g+w` bit düzeyinde VEYA (OR 020), `o-r` bit düzeyinde maskelemedir (AND ~004). 0754'e `chmod g+w,o-r` uygulanınca grup yazma biti açılır, diğerlerinin okuma biti kapanır: sonuç 0770, `rwxrwx---` 'dur (Şekil 15.37).



Şekil 15.37: `chmod g+w,o-r` : 0754 OR 020 AND ~004 = 0770.

Tanım 15.9

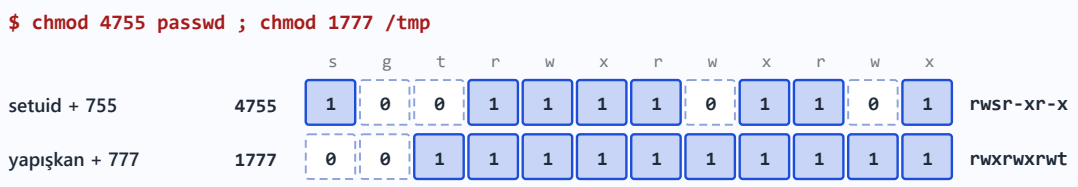
umask, sürecin yeni oluşturduğu dosyalardan silinecek izin bitlerinin maskesidir. Yeni dosya 666, yeni dizin 777 izin ister. Gerçek izin, istek & ~umask'tir.

Örnek 15.10

umask 022 ve 027 iken `touch a` ile `mkdir d` hangi izinleri üretir?

1. umask 022: dosya 666 & ~022 = 0644 (`rw-r--r--`), dizin 777 & ~022 = 0755 (`rwxr-xr-x`). Çoğu dağıtımın varsayılanı budur.
2. umask 027: dosya 666 & ~027 = 0640 (`rw-r-----`), dizin 777 & ~027 = 0750 (`rwxr-x---`). Diğerleri hiçbir şey göremez.
3. Sekizli çıkarma ilk durumda doğru sonucu tesadüfen verir; ikincide 666 − 027 = 637 (`rw--wxrwx`) gibi yanlış bitleri açan bir değer üretir. umask bit temizler, çıkarma yapmaz.

Üç özel bitin her biri farklı bir sorunu çözer (Şekil 15.38). **setuid** (4xxx) çalıştırılabilir dosyada, çalışan sürecin etkin kullanıcı kimliğini dosya sahibininki yapar. `passwd` komutu 4755 (`rwsr-xr-x`) izinlidir: sıradan bir kullanıcı onu çalıştırınca süreç root yetkisiyle `/etc/shadow`'a yazabilir. **setgid** (2xxx) aynısını grup için yapar; dizinde ise yeni dosyaların dizinin grubunu miras almasını sağlar. **Yapışkan bit** (1xxx) dizinde anlamlıdır. `/tmp` 1777 (`rwxrwxrwt`) izinlidir: herkes yazabilir, ama herkes yalnız kendi dosyasını silebilir. Sembolik gösterimde x'in yerine geçen s ya da t, alttaki x bitinin de açık olduğunu gösterir. x kapalıysa büyük harf (S, T) yazılır.



Şekil 15.38: setuid + 755 = 4755 (`rwsr-xr-x`) ve yapışkan + 777 = 1777 (`rwxrwxrwt`).

Sonuçlar `stat -c "%a %A" dosya` ile denetlenebilir; komut sekizli ve sembolik izni birlikte yazar. `umask -S` maskeyi sembolik gösterir. İzin hesapları İşletim Sistemi Koruma ve İzinler Laboratuvarı simülatöründe etkileşimli olarak denenebilir.

15.11.2 Erişim denetimi, root ve yetenekler

Bir süreç bir dosyaya eriştiğinde çekirdek sürecin etkin kimliklerine bakar ve tek bir izin sınıfı seçer:

1. Etkin uid 0 (root) ise izin verilir. Yalnız çalıştırma isteğinde dosyada en az bir x biti aranır.
2. Etkin uid dosyanın sahibiyse yalnız sahip bitlerine bakılır.
3. Etkin gid ya da ek gruplardan biri dosyanın grubuysa yalnız grup bitlerine bakılır.
4. Aksi hâlde diğerleri bitlerine bakılır.

İlk eşleşen sınıf kazanır; sonraki sınıflara hiç bakılmaz. Bu yüzden `----rw-rw-` izinli bir dosyayı sahibi okuyamaz, ama grup üyeleri ve başkaları okuyabilir. Sahip, isterse `chmod` ile izni kendine geri verebilir.

Bir sürecin iki tür kimliği vardır. **Gerçek** uid süreci başlatan kullanıcıdır. **Etkin** uid izin denetiminde kullanılan kimliktir; setuid programlarda ikisi ayrışır. `access()` çağrısı erişimi bilerek gerçek uid ile sınar. setuid bir programın “bu kullanıcı bu dosyayı gerçekten okuyabilir mi?” sorusunu yanıtlamasına yarar. `setuid` ve `setresuid` ise kimlik değiştirir; iyi yazılmış bir hizmet ayrıcalıklı işini bitirince bu çağrılarla yetkisini bırakır. Kullanıcı ve grup kimlikleri `/etc/passwd` ve `/etc/group` dosyalarında tutulur. Parola özetleri yalnız root'un okuyabildiği `/etc/shadow`'dadır. `sudo`, yetkilendirilmiş bir kullanıcıya komut başına root yetkisi verir.

Klasik modelde root her şeyi yapabilir, ötekiler hiçbir ayrıcalıklı işi yapamaz. Bu “ya hep ya hiç” yaklaşımı tehlikelidir. `ping` yalnız ham soket açabilmek için setuid root olmak zorundaydı. İçindeki bir hata saldırgana bütün sistemi açıyordu.

Tanım 15.10

Yetenek (*capability*), root'un ayrıcalıklarının, bağımsız olarak verilip alınabilen bir parçasıdır. Linux root yetkisini 40'tan fazla yeteneğe böler. Örnekler: `CAP_NET_RAW` ham soket açmak, `CAP_NET_BIND_SERVICE` 1024'ün altındaki portlara bağlanmak, `CAP_SYS_MODULE` modül yüklemek, `CAP_DAC_OVERRIDE` dosya izinlerini aşmak içindir.

`setcap cap_net_raw+ep /usr/bin/ping` komutundan sonra `ping` setuid root olmadan ham soket açabilir. Bir açık bulunsa bile saldırgan yalnız ham soket açabilir; dosyaları okuyamaz, modül yükleyemez. Yetenekler süreç başına kümeler hâlinde tutulur ve `task_struct`'ın `cred` yapısında saklanır.

15.11.3 Kod izleme: dosya izin denetimi

Çekirdeğin izin denetimi kabaca şu işlevle yapılır. `istek` bir bit maskesidir: okuma 4, yazma 2, çalıştırma 1.

```
int izin_var(cred *c, inode *f, int istek) {
    int bit, m = f->mode;       /* r4 w2 x1 */
    if (c->euid == 0)           /* root */
        return !(istek & 1) || (m & 0111);
    if (c->euid == f->uid)
        bit = (m >> 6) & 7;     /* sahip */
    else if (grupta(c, f->gid))
        bit = (m >> 3) & 7;     /* grup */
    else
        bit = m & 7;            /* diğer */
    return (bit & istek) == istek;
}
```

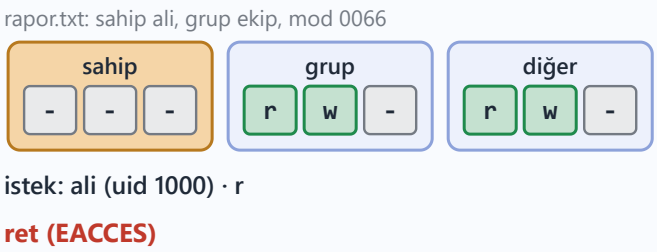
Root dalında dönüş ifadesi biraz şaşırtıcıdır. `!(istek & 1)`, istekte çalıştırma yoksa doğrudur; okuma ve yazma için izin bitlerine hiç bakılmaz. Çalıştırma isteniyorsa `m & 0111` sahip, grup ya da diğerlerinden en az birinde x biti arar. Root bile hiçbir x biti olmayan bir dosyayı çalıştıramaz, çünkü öyle bir dosya büyük olasılıkla bir program değildir. Sonraki `if` zinciri tek bir sınıf seçer. Seçilen üç bit kaydırma ve maskeyle alınır: sahip için `(m >> 6) & 7`, grup için `(m >> 3) & 7`, diğerleri için `m & 7`. Son satır, istenen bütün bitlerin açık olup olmadığını sınar.

Dosya `rapor.txt`'nin sahibi ali (uid 1000), grubu ekip, modu 0640 (`rw-r-----`) olsun. veli (1001) ekip grubundadır, can (1002) değildir.

Kim, istek	Mod	Sınıf	bit	Sınama	Sonuç
ali, w (2)	0640	sahip	6 (rw-)	6 & 2 == 2	izin
veli, w (2)	0640	grup	4 (r--)	4 & 2 = 0 ≠ 2	ret
veli, r (4)	0640	grup	4 (r--)	4 & 4 == 4	izin

Kim, istek	Mod	Sınıf	bit	Sınama	Sonuç
can, r (4)	0640	diğer	0 (---)	0 & 4 = 0 ≠ 4	ret
root, r (4)	0640	root	—	istekte x yok	izin
ali, r (4)	0066	sahip	0 (---)	0 & 4 = 0 ≠ 4	ret

Tablo 15.16: `izin_var` izlemesi (işlev Python'a birebir çevrilip çalıştırıldı).



Şekil 15.39: Mod 0066'da ali sahip sınıfına düşer ve orada hiç bit yoktur: ret (`EACCES`).

Son satır izlemenin en öğretici durumudur. Mod 0066'da (`----rw-rw-`) can ve veli dosyayı okuyabilir. ali ise dosyanın sahibi olduğu için sahip sınıfına düşer; grup üyeliğine hiç bakılmaz. Sahip bitleri boş olduğundan istek reddedilir. Gerçek çekirdekte bu denetimin önünde ve arkasında başka katmanlar da vardır: erişim denetim listeleri (ACL), yetenekler ve güvenlik modülleri.

15.11.4 Konteynerler: ad alanları ve cgroups

Konteyner, Linux çekirdeğinde ayrı bir nesne değildir. Sıradan süreçlerin, iki çekirdek özelliğiyle yalıtılmış hâlidir. Bu özellikler ad alanları ve cgroups'tur.

Tanım 15.11

Ad alanı (*namespace*), bir süreç grubuna sistem kaynaklarının ayrı bir görünümünü veren çekirdek özelliğidir. Ad alanındaki süreçler yalnız o görünümdeki PID'leri, bağlama noktalarını, ağ arayüzlerini ya da kullanıcı kimliklerini görür. **Kontrol grubu** (*control group*, cgroup), bir süreç grubunun CPU, bellek, G/Ç ve süreç sayısı gibi kaynak kullanımını sınırlayan ve ölçen çekirdek özelliğidir.

Kısa bir kural ikisini ayırır: **ad alanı ne görüleceğini, cgroup ne kadar kullanılacağını belirler**. Linux'ta sekiz ad alanı türü vardır (Tablo 15.17). Her biri `clone` , `unshare` ya da `setns` çağrısında bir bayrakla seçilir. Bir sürecin hangi ad alanlarında olduğu `/proc/PID/ns/` dizinindeki bağlardan okunur; iki süreç aynı ad alanındaysa bağlar aynı numarayı gösterir.

Ad alanı	Bayrak	Yalıttığı kaynak
PID	`CLONE_NEWPID`	Süreç kimlikleri; içerideki ilk süreç PID 1

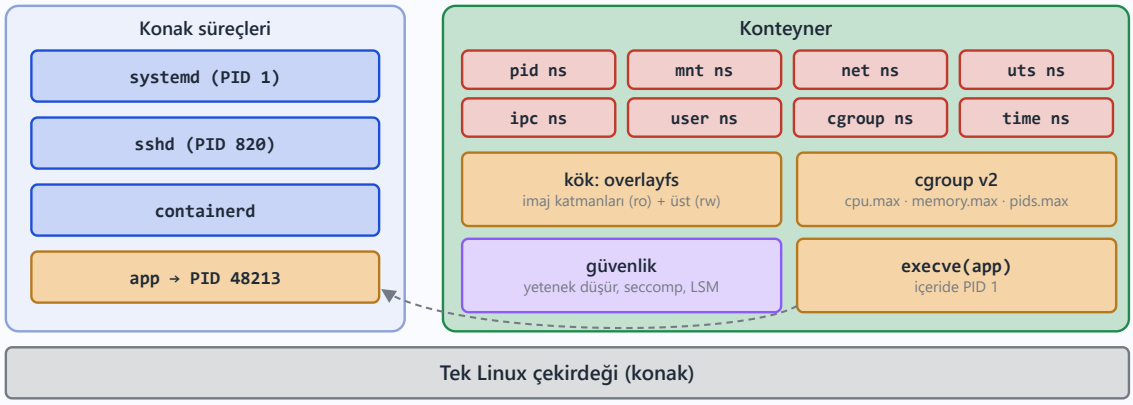
Ad alanı	Bayrak	Yalıttığı kaynak
Bağlama	`CLONE_NEWNS`	Bağlama noktaları, dosya sistemi ağacı
Ağ	`CLONE_NEWNET`	Ağ arayüzleri, yönlendirme tablosu, portlar, güvenlik duvarı
UTS	`CLONE_NEWUTS`	Ana makine adı ve alan adı
IPC	`CLONE_NEWIPC`	System V IPC, POSIX ileti kuyrukları
Kullanıcı	`CLONE_NEWUSER`	uid/gid eşlemesi: içerideki root dışarıda sıradan bir uid
cgroup	`CLONE_NEWCGROUP`	cgroup ağacının görünen kökü
Zaman	`CLONE_NEWTIME`	Açılıştan beri geçen ve monoton saatler (5.6+)

Tablo 15.17: Linux ad alanları.

Bağlama ad alanı tarihte ilk gelendir. O zaman başka türlerinin olacağı düşünülmediği için bayrağın adı yalnız `CLONE_NEWNS` 'dir. Kullanıcı ad alanı özeldir: ayrıcalıksız bir süreç bile onu oluşturabilir ve içinde root olabilir. İçerideki uid 0, dışarıda örneğin uid 100000'e eşlenir. İçerideki root yalnız kendi ad alanındaki kaynaklar üzerinde yetkilidir. “Köksüz” (*rootless*) konteynerler bu özelliğe dayanır.

Bir konteyner çalışma zamanının (runc, crun) yaptığı iş beş adımda izlenebilir (Şekil 15.40).

1. **Ad alanları.** `clone()` ya da `unshare()` ile yeni ad alanları oluşturulur: kendi PID numaraları, bağlama ağacı, ağ yığını, ana makine adı, IPC, kullanıcı eşlemesi, cgroup ve saat görünümü.
2. **Kök dosya sistemi.** Bağlama ad alanında `pivot_root` ile imajın kök dosya sistemine geçilir. Kök çoğunlukla overlayfs'tir: imajın salt okunur katmanları üst üste konur, en üste konteynere özel yazılabilir bir katman eklenir. Aynı imajdan çalışan yüz konteyner alt katmanları paylaşır.
3. **Kaynak sınırları.** cgroup v2 dizinine sınırlar yazılır: `cpu.max` = `"50000 100000"`, `memory.max` = 512M, `pids.max` = 100.
4. **Saldırı yüzeyi.** Gereksiz yetenekler düşürülür. `no_new_privs` bayrağı açılır; böylece setuid programlar bile yetki kazandıramaz. Onlarca riskli sistem çağrısını engelleyen bir seccomp süzgeci ve bir AppArmor ya da SELinux profili uygulanır.
5. **Başlatma.** Son olarak `execve` uygulamayı çalıştırır. Uygulama içeride PID 1'dir, konakta PID 48213'tür.



Şekil 15.40: Bir konteynerin kuruluşu: sekiz ad alanı, overlayfs kök, cgroup v2 sınırları ve güvenlik ayarları; hepsi tek bir konak çekirdeği üzerinde.

Örnek 15.11

`cpu.max` dosyasındaki `"50000 100000"` ne anlama gelir? Konteynere tam iki CPU vermek için ne yazılır?

1. Biçim `"kota dönem"` 'dir, ikisi de mikrosaniye. Grup her 100.000 µs'lik (100 ms) dönemde toplam 50.000 µs CPU zamanı kullanabilir.
2. 50.000 / 100.000 = 0,5: grup yarım CPU'luk zaman alır. Kotayı doldurursa dönemin geri kalanında, makinede boşta CPU'lar olsa bile çalıştırılmaz.
3. İki CPU için kota dönemin iki katı olmalıdır: `"200000 100000"` . Kota birden çok CPU'ya yayılabilir; grubun iki iş parçacığı iki çekirdekte aynı anda çalışabilir.

cgroup v2'de bütün denetleyiciler (`cpu` , `memory` , `io` , `pids`) tek bir hiyerarşiyi paylaşır. Eski v1'de her denetleyicinin ayrı ağacı vardı ve bir süreç farklı ağaçlarda tutarsız gruplara düşebiliyordu. Bellek sınırı aşılırsa OOM killer yalnız o gruptan bir kurban seçer. systemd de her hizmeti kendi cgroup'una koyar. `systemd-cgls` ağacı, `systemd-cgtop` grupların anlık kullanımını gösterir.

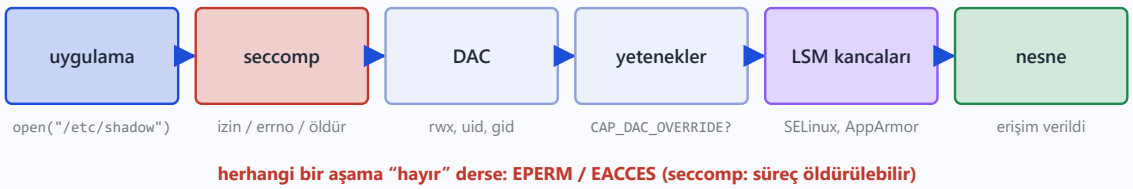
Konteynerlerin hepsi aynı çekirdeği paylaşır. Bu, onları sanal makineden çok daha hafif kılar: konteyner milisaniyeler içinde başlar ve kendi çekirdeği için bellek harcamaz. Bedeli yalıtımın zayıflığıdır. Çekirdekteki bir açık bütün konteynerleri birden etkileyebilir. Sanal makinede ise saldırganın önce konuk çekirdeği, sonra hiper yöneticiyi aşması gerekir.

Sık yapılan hata

Ad alanını güvenlik sınırı, cgroup'u görünürlük aracı sanmak. Bir PID ad alanı süreçleri gizler ama bellek kullanımını sınırlamaz; sınırsız bir konteyner konağın belleğini tüketebilir. Tersine, cgroup belleği sınırlar ama konaktaki süreçleri gizlemez. Konteyner için ikisi birlikte gerekir.

15.11.5 seccomp ve Linux güvenlik modülleri

Bir sistem çağrısı nesnesine ulaşmadan önce birkaç denetimden geçer (Şekil 15.41). Örnek olarak bir uygulama `open("/etc/shadow")` çağırsın.



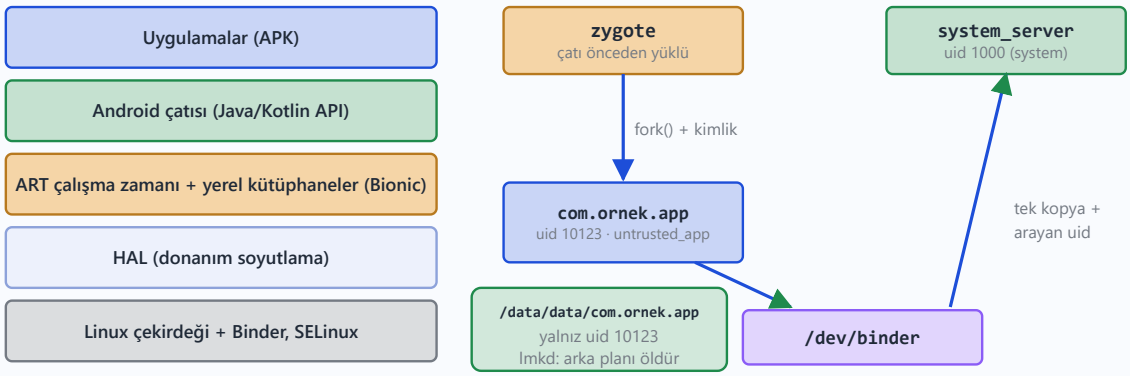
Şekil 15.41: Bir sistem çağrısının seccomp, DAC, yetenek ve LSM denetimlerinden geçmesi. Herhangi bir aşama "hayır" derse çağrı `EPERM` ya da `EACCES` ile döner.

1. **seccomp.** Süreç kendine bir seccomp-BPF süzgeci kurmuşsa çağrı önce ondan geçer. Süzgeç çağrının numarasına ve argümanlarına bakan küçük bir BPF programıdır. İzin verebilir, bir hata kodu döndürebilir ya da süreci öldürebilir. Chrome'un oluşturucu süreçleri, systemd hizmetleri, Docker ve Android uygulamaları bu süzgeçleri kullanır.
2. **DAC.** İsteğe bağlı erişim denetimi (*discretionary access control*) rwx bitlerine, uid ve gid'e bakar. Buna isteğe bağlı denir, çünkü izinleri dosyanın sahibi kendi isteğiyle değiştirebilir.
3. **Yetenekler.** DAC reddederse çekirdek sürecin bu reddi aşan bir yeteneği olup olmadığına bakar; örneğin `CAP_DAC_OVERRIDE`.
4. **LSM.** Son olarak Linux güvenlik modüllerinin kancaları çalışır. Bunlar zorunlu erişim denetimi uygular: kuralları yönetici koyar ve root bile onları aşamaz.

Başlıca güvenlik modülleri üç tanedir. **SELinux** etiket tabanlıdır: her sürecin ve dosyanın bir tipi vardır; politika hangi tipin hangi tipe ne yapabileceğini söyler. RHEL, Fedora ve Android'de kullanılır. **AppArmor** yol tabanlı profiller kullanır: "bu program yalnız şu dosyalara erişebilir". Yazması daha kolaydır; Ubuntu, SUSE ve Debian'da yaygındır. **Landlock** (5.13) ayrıcalıksız bir sürecin kendini korumalı alana almasını sağlar: "bundan sonra yalnız `./veri` altına yazabilirim". Bir süreç bu kısıtı bir kez koyunca kaldıramaz; çocukları da miras alır.

15.12 Android

Android, Linux çekirdeği üzerinde çalışan ama GNU kullanıcı alanını kullanmayan bir sistemdir (Şekil 15.42). C kütüphanesi Google'ın yazdığı Bionic'tir. Uygulamalar Java ya da Kotlin ile yazılır ve Android Çalışma Zamanında (ART) çalışır. Donanım soyutlama katmanı (HAL) üreticilerin sürücülerini çatıdan ayırır. Android çekirdeğe üç önemli ek getirdi: Binder IPC mekanizması, enerji yönetimi için uyanıklık kilitleri (*wakelock*) ve zorunlu SELinux. Binder zamanla ana çekirdeğe de girdi.



Şekil 15.42: Android mimarisi, zygote ile uygulama başlatma, Binder IPC ve uygulama korumalı alanı.

Bir uygulamanın başlaması ilginç bir `fork` kullanımıdır. Önyüklemede `init`, **zygote** adlı süreci başlatır. zygote ART'ı ve çatının binlerce sınıfını ile kaynaklarını bir kez yükler ve bekler. Bir uygulama açılırken zygote `fork` eder. Yüklü sayfalar COW ile paylaşılır; her uygulama sınıfları yeniden yüklemek zorunda kalmaz ve başlatma hızlanır. Bellek de tasarruf edilir, çünkü çatının sayfaları bütün uygulamalarda tek kopyadır. Çocuk süreç sonra uygulamanın benzersiz uid'ine, SELinux bağlamına (`untrusted_app`) ve seccomp süzgecine geçer. Burada `fork` sonrası `exec` yapılmaz; zaten hazır olan görüntü kullanılır.

Uygulama konum ya da kamera gibi bir sistem hizmetini **Binder** ile çağırır. Binder, `/dev/binder` aygıtı üzerinden çalışan bir uzak yordam çağrısı mekanizmasıdır. Çekirdek veriyi alıcının önceden eşlenmiş tamponuna tek bir kopyayla taşır. Ayrıca arayanın uid'ini de iletir. Bu uid'i uygulama uyduramaz, çünkü onu çekirdek ekler. Hizmet izni buna göre denetler.

Android'in korumalı alanı Linux'un en eski koruma mekanizmasına dayanır: her uygulama ayrı bir Linux kullanıcısıdır. Şekildeki uygulama uid 10123 ile çalışır ve `/data/data/com.ornek.app` dizini yalnız bu uid'e açıktır. Bir uygulama başka bir uygulamanın dosyalarını okumaya kalkarsa sıradan dosya izinleri onu durdurur. Bellek azalınca çekirdeğin OOM killer'ı yerine kullanıcı alanındaki `lmkd` devreye girer. lmkd bellek baskısını PSI sayaçlarıyla izler ve önemsiz arka plan uygulamalarını öldürür. Kullanıcı o uygulamaya döndüğünde Android onu kaydedilmiş durumundan yeniden başlatır.

15.13 Dersin kavramları Linux'ta

Bu örnek inceleme, dersin genel kavramlarının her birine Linux'ta somut bir karşılık buldu (Tablo 15.18). Tablonun son satırı, Linux'un bugünkü yönünü özetler: mekanizma çekirdekte kalırken politika giderek kullanıcıya bırakılıyor.

Konu	Kavram	Linux'taki karşılığı
Yapılar	Sistem çağrısı, monolitik çekirdek	`syscall`, LKM, eBPF
Süreçler	PCB, oluşturma, IPC	`task_struct`, fork/exec/wait, sinyal, boru

Konu	Kavram	Linux'taki karşılığı
İş parçacıkları	Çekirdek iş parçacığı modeli	`clone()`, NPTL 1:1
Çizelgeleme	Adil paylaşım, gerçek zaman	EEVDF, FIFO/RR/DEADLINE, `sched_ext`
Senkronizasyon	Kilit, koşul değişkeni	futex, döner kilit, RCU
Kilitlenme	Tespit ve önleme	lockdep, fcntl `EDEADLK`
Bellek	Ayırma, parçalanma	bölgeler, buddy listeleri, SLUB
Sanal bellek	Sayfalama, yer değiştirme	4/5 düzey tablo, COW, MGLRU, zswap, OOM
Dosya sistemleri	i-node, dizin, günlük	VFS, ext4, XFS, Btrfs
G/Ç	Sürücü, çizelgeleme	aygıt dosyaları, blk-mq, io_uring
Çok işlemcili	Yük dengeleme, NUMA	CPU başına kuyruklar, çizelgeleme alanları
Güvenlik	Erişim denetimi, yalıtım	rwx, yetenekler, ad alanları, SELinux, seccomp
Tasarım	Mekanizma ve politika ayrımı	cgroup, `sched_ext`: politika kullanıcıda

Tablo 15.18: Dersin kavramları ve Linux'taki karşılıkları.

Tabloda adı geçen ama bu notta ayrıntılanmayan iki mekanizma kısa bir açıklamayı hak eder. **futex** (hızlı kullanıcı alanı muteksi), pthread kilitlerinin temelidir. Kilit boştayken alma ve bırakma tümüyle kullanıcı alanında, tek bir atomik komutla olur. Çekirdeğe yalnız çekişme varken, yani bir iş parçacığının uyuması gerektiğinde gidilir. **RCU** (oku–kopyala–güncelle), okuyucuların hiç kilit almadan veri yapısını okuduğu bir senkronizasyon yöntemidir. Yazar yapının yeni bir kopyasını hazırlar ve işaretçiyi atomik olarak değiştirir. Eski kopya, onu okuyor olabilecek bütün okuyucular bitene kadar silinmez. `lockdep` ise çekirdeğin kendi kilitlenme denetleyicisidir: kilitlerin hangi sırayla alındığını izler ve bir sıra döngüsü gördüğü anda, kilitlenme henüz yaşanmadan uyarı verir.

15.14 Özet

- Linux, UNIX'in POSIX arayüzünü baştan yazılmış kodla gerçekleyen **monolitik** bir çekirdektir. GNU araçlarıyla birleşerek dağıtımları oluşturur. Yaklaşık 9–10 haftada bir sürüm çıkar; her yılın son sürümü LTS olur.
- Katmanlar arasında üç arayüz vardır: kullanıcı, kütüphane ve sistem çağrısı. `printf` bir tampona yazar; asıl sistem çağrısı `write`'tır ve `syscall` komutuyla çekirdeğe girer.
- Kabuk yönlendirmeleri soldan sağa `dup2` ile, boru hatlarını `pipe`, `fork`, `dup2` ve `exec` ile kurar. Kullanılmayan yazma ucu kapatılmazsa okuyucu EOF göremez.
- **Yüklenebilir modüller** monolitik çekirdeğe esneklik verir ama çekirdek ayrıcalığıyla çalışır. eBPF doğrulanmış küçük programları çekirdekte güvenle çalıştırır.
- Linux'ta süreç ve iş parçacığı aynı yapıdır: `task_struct`. Fark `mm`, `files`, `fs`, `sighand` gibi alt yapıların paylaşılıp paylaşılmadığıdır. `clone()` bayrakları bunu seçer; NPTL 1:1 modelini kurar. Kullanıcının gördüğü PID, `tgid`'dir.

- Sinyaller bekleyen kümesinde bir bit olarak bekler ve kullanıcı moduna dönüşte, hazır = bekleyen & ~engelli kümesinin en küçük numarasından başlayarak teslim edilir. Engellemek ertelemektir; klasik sinyaller sayılmaz.
- Adil sınıf 6.6'dan beri **EEVDF** ile çalışır: uygun görevler ($v \le V$) arasından sanal son tarihi en erken olan seçilir. Pay ağırlıklara (nice 0 → 1024) bağlıdır; kısa dilim daha sık seçilme sağlar.
- Adres alanı VMA'lardan oluşur ve maple ağacında tutulur. x86-64'te 48 bitlik adres 9+9+9+9+12 bitlik dört düzeyli yürüyüşle çevrilir; LA57 beşinci düzeyi ekler.
- Fiziksel bellek düğüm → bölge → sayfa hiyerarşisindedir; buddy ayırıcı 2^0–2^{10} sayfalık blokları yönetir. **SLUB** küçük nesneleri slab'larda tutar; serbest liste işaretçisi boş nesnenin içindedir ve hızlı yol kilitsizdir.
- Boş RAM sayfa önbelleğine gider. Baskı altında kswapd, MGLRU, zswap/takas, doğrudan geri kazanım ve son çare olarak OOM killer devreye girer.
- Aygıt dosyalarında major numara sürücüyü, minor numara birimi seçer. Blok katmanı blk-mq ile çok kuyrukludur. Sürücüler `file_operations` tablosuyla VFS'e bağlanır.
- VFS'in dört nesnesi superblock, inode, dentry ve file'dır. ext4 blok grupları, extent'ler ve `data=ordered` günlükleme kullanır. Veri, bağ sayısı ile açık dosya sayısının ikisi de sıfır olunca serbest kalır.
- İzin denetiminde ilk eşleşen sınıf kazanır; umask bit temizler (istek & ~umask). Yetenekler root'u parçalara böler. Konteyner = **ad alanları** (ne görülür) + **cgroups** (ne kadar kullanılır) + seccomp ve LSM.
- Android aynı çekirdeği Bionic, ART, Binder, zygote ve uygulama başına uid ile kullanır.

15.15 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

monolitik çekirdek *monolithic kernel* · yüklenebilir çekirdek modülü *loadable kernel module* · sistem çağrısı *system call* · kabuk *shell* · boru, boru hattı *pipe, pipeline* · dosya tanımlayıcısı *file descriptor* · süreç kontrol bloğu (`task_struct`) *process control block* · iş parçacığı grubu kimliği *thread group ID* · sinyal *signal* · yazınca kopyalama *copy-on-write* · sanal çalışma süresi *virtual runtime* · uygunluk, sanal son tarih *eligibility, virtual deadline* · önyükleyici *bootloader* · sanal bellek alanı (VMA) *virtual memory area* · buddy (kardeş) sistemi *buddy system* · slab ayırıcı *slab allocator* · sayfa önbelleği *page cache* · sanal dosya sistemi (VFS) *virtual file system* · dosya kilitleme *file locking* · günlükleme *journaling* · katı bağ / sembolik bağ *hard link / symbolic link* · yetenek *capability* · ad alanı *namespace* · kontrol grubu (cgroup) *control group* · konteyner *container*

15.16 Kendini Dene

1. x86-64'te 4 KiB sayfa ve dört düzeyli tablo kullanılıyor. 0x00007ffd4a3b2010 adresinin dört indisi ve bağıl konumu nedir? PTE'deki çerçeve numarası 0x2f00a ise fiziksel adres nedir?

   Cevap

   PGD = adres >> 39 = 255, PUD = 501, PMD = 81, PTE = 434, bağıl konum = 0x010. Fiziksel adres 0x2f00a × 4096 + 0x010 = 0x2f00a010. Bu adres bir yığın adresidir: PGD 255, kullanıcı yarısının son girdisidir.

2. nice 0 (w = 1024) ve nice 1 (w = 820) iki sürekli hazır görev tek CPU'yu paylaşıyor. Uzun vadede CPU nasıl paylaşılır? nice 1 görevi 4 ms çalışınca `vruntime`'ı ne kadar artar?

   Cevap

   Pay ağırlıkla orantılıdır: 1024 / 1844 ≈ %55,5 ve 820 / 1844 ≈ %44,5. Bir nice adımı payı yaklaşık 10 puan kaydırır. Δv = 4 × 1024 / 820 ≈ 4,995 ms.

3. Bir kuyrukta V = 12 ms. A (v = 10, d = 14), B (v = 12, d = 13) ve C (v = 13, d = 12) görevleri var. EEVDF hangisini seçer? CFS hangisini seçerdi?

   Cevap

   Uygun görevler v ≤ V olanlardır: A ve B. C'nin son tarihi en erken olsa da uygun değildir, çünkü payından fazlasını almıştır. A ile B arasında son tarihi erken olan B seçilir. CFS en küçük `vruntime`'ı, yani A'yı seçerdi.

4. Bir süreç `SIGUSR1`'i engellemiş ve `SIGINT` için bir işleyici kurmuş. Uyurken sırayla 17, 10 ve 2 numaralı sinyaller geliyor. Kullanıcı moduna dönüşte hangi sinyaller hangi sırayla teslim edilir? Hangisi bekler?

   Cevap

   Bekleyen {2, 10, 17}, engelli {10}; hazır {2, 17}. En küçük numara önce: 2 (işleyici çalışır), sonra 17 (`SIGCHLD`, varsayılan: yok say). 10 bekleyen kümesinde kalır ve engel kalkınca teslim edilir. Geliş sırası seçimde rol oynamaz.

5. Yeni bir `kmalloc-512` slab'ı (4 KiB, sekiz nesne: +0, +512, …) ile şu işlemler yapılıyor: a, b, c = ayir;

birak(b); birak(a); d, e, f = ayir. d, e ve f hangi adresleri alır?

Cevap

a = +0, b = +512, c = +1024; freelist +1536. birak(b): +512 → +1536. birak(a): +0 → +512 → +1536. Son bırakılan ilk verilir: d = +0, e = +512, f = +1536.

6. `clone(CLONE_VM | CLONE_FILES | SIGCHLD, …)` ile, `CLONE_THREAD` verilmeden oluşturulan görev `ps`'te nasıl görünür? Çocuk yeni bir dosya açarsa ebeveyn onu kullanabilir mi?

Cevap

`CLONE_THREAD` olmadığı için çocuğun kendi TGID'i vardır; `ps`'te ayrı bir süreç olarak görünür ve bitince ebeveyne `SIGCHLD` gönderir. Yine de adres alanını ve tanımlayıcı tablosunu ebeveynle paylaşır. Çocuğun açtığı dosyanın numarası ebeveynde de geçerlidir; ebeveyn aynı numarayla okuyabilir.

7. umask 077 ve 002 iken `touch a` ve `mkdir d` hangi izinleri üretir?

Cevap

umask 077: dosya 0600 (`rw-------`), dizin 0700 (`rwx------`). umask 002: dosya 0664 (`rw-rw-r--`), dizin 0775 (`rwxrwxr-x`). 002, grup çalışması için ortak dizinlerde kullanılır.

8. `prog` dosyasının sahibi ali, grubu ekip, modu 0750. veli ekip grubunda, can değil. veli `prog`'u çalıştırabilir mi? can okuyabilir mi? root, modu 0644 olan bir betiği çalıştırabilir mi?

Cevap

veli grup sınıfına düşer: bit 5 (r-x), 5 & 1 = 1, izin. can diğerleri sınıfına düşer: bit 0, ret. root için okuma ve yazmada bitlere bakılmaz, ama çalıştırmada en az bir x biti gerekir; 0644'te hiç yoktur, ret (`EACCES`).

9. `echo x > f`, `ln f g`, `ln -s f h` çalıştırılıyor. Bir süreç `g`'yi açıp tanımlayıcıyı tutuyor. Sonra `rm f` ve `rm g` yapılıyor. Veri blokları ne zaman serbest kalır? `cat h` ne yazar?

Cevap

f ve g aynı i-node'dur; bağ sayısı 2'dir. İki `rm`'den sonra bağ sayısı 0 olur, ama açık bir tanımlayıcı bulunduğu için bloklar ancak süreç tanımlayıcıyı kapatınca serbest kalır. h, `"f"` yolunu tutar; o ad artık yoktur. `cat h` “No such file or directory” (`ENOENT`) hatası verir.

10. Bir konteynerin `cpu.max` dosyasında `"25000 100000"` yazıyor. Konteyner ne kadar CPU kullana-

bilir? Bu sınır içerideki `ps` çıktısını etkiler mi?

Cevap

Her 100 ms'lik dönemde 25 ms, yani çeyrek CPU. Sınır bir cgroup ayarıdır; yalnız ne kadar kullanılacağını belirler. `ps` 'in neyi gösterdiğini PID ad alanı belirler, cgroup değil.

15.17 İleri okuma

- R. Love, *Linux Kernel Development*, 3. baskı, Addison-Wesley, 2010: süreç yönetimi, çizelgeleme, bellek ve VFS bölümleri `task_struct` 'tan slab ayırıcısına kadar bu nottaki yapıların çekirdek kodundaki karşılığını sade bir dille gösterir. CFS dönemine aittir.
- M. Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010: sinyaller, süreç oluşturma, borular, dosya kilitleme ve yetenekler için en kapsamlı programcı kaynağı; yazarın man7.org'daki kılavuz sayfaları güncel tutulur.
- Linux çekirdek belgeleri, docs.kernel.org, "EEVDF Scheduler" ve "Control Group v2" sayfaları: EEVDF'in kavramlarını ve cgroup v2 denetleyicilerinin bütün dosyalarını birinci elden anlatır.
- I. Stoica, H. Abdel-Wahab, "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation", Old Dominion Üniversitesi teknik raporu, 1995: alacak, uygunluk ve sanal son tarih kavramlarının kuramsal temeli.

15.18 Kaynaklar

1. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 10: Case Study 1: Unix, Linux and Android.
2. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 20: The Linux System.
3. R. Love, *Linux Kernel Development*, 3. baskı, Addison-Wesley, 2010.
4. M. Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010; man7.org kılavuz sayfaları (namespaces(7), capabilities(7), signal(7), fcntl(2), clone(2)).
5. The Linux Kernel Documentation, docs.kernel.org — scheduler (EEVDF, sched-ext), admin-guide/mm (MGLRU, zswap), block (blk-mq), filesystems (vfs, ext4), userspace-api (seccomp, landlock), admin-guide/cgroup-v2.
6. Android Open Source Project, source.android.com — mimari, Binder, uygulama korumalı alanı.
7. D. M. Ritchie, K. Thompson, "The UNIX Time-Sharing System", *Communications of the ACM*, 17(7), 1974.
8. I. Stoica, H. Abdel-Wahab, "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation", teknik rapor, Old Dominion University, 1995.

EK A

Terim Sözlüğü

Kitapta geçen İngilizce terimler ve her biri için kullanılan tek Türkçe karşılık. Sağdaki sayı, terimin ayrıntılı anlatıldığı bölümdür. Sözlüğün aranabilir hâli ders sayfasındadır.

A

abstraction → soyutlama 1
Donanımın karmaşık ayrıntılarını gizleyip süreç, dosya, adres alanı gibi kavramlar sunmak.

access control list (ACL) → erişim denetim listesi (ACL) 13
Bir nesneye kimin hangi hakla erişebileceğini sıralayan liste.

access matrix → koruma matrisi 13
Satırları alanlar, sütunları nesneler olan ve her hücresinde izinlerin tutulduğu koruma modeli.

address space → adres alanı 2
Bir sürecin kullanabildiği bellek adreslerinin kümesi.

address space layout randomization (ASLR) → adres alanı yerleşiminin rastgeleleştirilmesi 13
Kod, yığın ve kütüphanelerin adreslerini her çalıştırmada rastgele seçerek saldırıyı zorlaştırma.

aging → yaşlandırma 5
Bekleyen sürecin önceliğini zamanla artırarak açlığı önleme; sayfa yer değiştirmede sayaç tekniği.

Amdahl's law → Amdahl yasası 4
Seri bölümün, paralel hızlanmanın üst sınırını belirlediğini söyleyen yasa.

application programming interface (API) → API (uygulama programlama arayüzü) 2
Programcının kullandığı kütüphane işlevleri; altta sistem çağrılarını kullanır.

atomic instruction → atomik komut 6
Okuma ve yazmayı bölünmeden yapan donanım komutu (TSL, XCHG, CAS).

authentication → kimlik doğrulama 13
Kullanıcının iddia ettiği kişi olduğunu doğrulama.

authorization → yetkilendirme 13
Kimliği doğrulanmış bir kullanıcının hangi kaynaklara hangi hakla erişebileceğine karar verme adımı.

B

backing store (swap space) → yedekleme deposu (takas alanı) 9
Bellekte olmayan sayfaların diskte tutulduğu alan.

banker's algorithm → bankacı algoritması 7
Azami istekleri bilerek yalnızca güvenli durumlara izin veren kaçınma algoritması.

barrier → senkronizasyon bariyeri 6
Bir süreç ya da iş parçacığı grubunu belli bir noktada buluşturan senkronizasyon aracı; hepsi gelmeden kimse geçemez.

base and limit registers → taban ve limit yazmaçları 8
Sürecin bellek bölgesinin başlangıcını ve boyutunu tutan yazmaçlar.

batch system → toplu iş sistemi 1
İşlerin kullanıcı etkileşimi olmadan sırayla yürütüldüğü erken sistem türü.

Belady's anomaly → Belady anomalisi 9
FIFO'da çerçeve sayısı artınca sayfa hatası sayısının da artabilmesi.

binding time → bağlama zamanı 14
Bir adın bir nesneye bağlandığı an: derleme, yükleme ya da çalışma zamanı.

bitmap → biteşlem 8
Her birim için bir bitle boş/dolu bilgisini tutan yapı.

block / character device → blok / karakter aygıtı 11
Veriyi adreslenebilir bloklarla ya da bayt akışı olarak aktaran aygıt sınıfları.

blocking / nonblocking / asynchronous I/O → bloke eden / bloke etmeyen / eş zamansız G/Ç 11
Çağıranı bekleten, hemen dönen ya da bitince bildiren G/Ç arayüzleri.

booting → önyükleme 2
Açılışta firmware'den başlayıp çekirdeği belleğe yükleyen süreç.

bootloader → önyükleyici 15
Firmware'den sonra çalışıp çekirdeği yükleyen küçük program (GRUB).

bounded waiting → sınırlı bekleme 6
Bir sürecin kritik bölgeye girmek için sınırlı sayıda sıra beklemesi koşulu.

Brooks's law → Brooks yasası 14
Geciken bir yazılım projesine insan eklemek onu daha da geciktirir.

buddy system → buddy (kardeş) sistemi 9
Belleği ikinin kuvvetleri boyutunda bölüp kardeş blokları birleştiren ayırıcı.

buffer / buffering → tampon / tamponlama 11
Aktarılan veriyi geçici olarak bellekte tutmak.

buffer cache → tampon önbelleği 10
Disk bloklarını bellekte tutan önbellek.

buffer overflow → tampon taşması 13
Tampon sınırını aşan yazmayla bitişik belleğin (dönüş adresi) bozulması.

bus → veri yolu 11
Bileşenleri birbirine bağlayan ortak iletişim hattı (PCIe, USB).

busy waiting → meşgul bekleme 6
Bir koşulu döngüde sürekli sınayarak beklemek; CPU harcar.

C

cache → önbellek 8
Sık kullanılan veriyi daha hızlı bir düzeyde tutan bellek.

cache coherence → önbellek tutarlılığı 12
Aynı verinin farklı önbelleklerdeki kopyalarının tutarlı tutulması.

cache hit / miss → isabet / ıska 8
Aranan veri önbellekte bulunur (isabet) ya da bulunmaz (ıska).

cache line → önbellek satırı 8
Önbellek ile bellek arasında taşınan en küçük blok (genellikle 64 bayt).

capability → yetenek 13
Bir nesneye belirli haklarla erişim veren, sürece ait değiştirilemez belirteç.

cascading termination → basamaklı sonlandırma 3
Ebeveyn süreç sonlanınca bütün çocuklarının da sonlandırılması; UNIX'te çocuklar ise yeni bir ebeveyne bağlanır.

certificate → sertifika 13
Bir açık anahtarın kime ait olduğunu güvenilir bir otoritenin imzasıyla doğrulayan belge.

certificate authority (CA) → sertifika otoritesi 13
Sertifikadaki açık anahtarın sahibine ait olduğunu imzasıyla onaylayan kurum.

checkpoint → denetim noktası 7
Sürecin durumunun diske kaydedildiği an; kilitlenmeden sonra süreç bu noktaya geri alınabilir.

circular wait → döngüsel bekleme 7
Süreçlerin bir döngü oluşturacak biçimde birbirini beklemesi.

client–server model → istemci–sunucu modeli 2
Hizmetlerin sunucu süreçlerde çalışıp istemcilerle ileti alışverişi yaptığı yapı.

clock algorithm → saat algoritması 9
İkinci şansı dairesel liste ve bir elle (işaretçi) gerçekleştiren algoritma.

clock cycle → saat turu (tur) 8
İşlemci saatinin bir periyodu; boru hattında her aşama bir tur sürer.

Coffman conditions → kilitlenmenin dört koşulu 7
Karşılıklı dışlama, tut ve bekle, kesme yok, döngüsel bekleme.

compaction → sıkıştırma 8
Kullanılan bölgeleri bir araya kaydırıp boş alanı birleştirme.

compare-and-swap (CAS) → karşılaştır ve değiştir (CAS) 6
Adresteki değer beklenene eşitse yeni değeri yazan ve her durumda eski değeri döndüren atomik komut.

Completely Fair Scheduler (CFS) → tamamen adil çizelgeleyici (CFS) 15
Linux'un ideal çok görevli işlemciye öykünen, her zaman sanal çalışma süresi en küçük görevi seçen çizelgeleyicisi.

compression (lossy / lossless) → sıkıştırma (kayıplı / kayıpsız) 9
Veriyi daha az bitle gösterme; bilgi kaybıyla ya da kaybı olmadan.

concurrency → eş zamanlılık 3
Birden çok etkinliğin zaman içinde iç içe ilerlemesi; aynı anda çalışmaları gerekmez.

condition variable → koşul değişkeni 6
Bir koşul gerçekleşene dek wait ile bekleyip signal ile uyandırılan nesne.

conditional compilation → koşullu derleme 14
Kodun bir bölümünün önişlemci koşullarına göre derlemeye alınması ya da dışarıda bırakılması.

confidentiality / integrity / availability (CIA) → gizlilik / bütünlük / erişilebilirlik 13
Güvenliğin üç temel hedefi.

container → konteyner 12
Aynı çekirdeği paylaşan, ad alanları ve cgroups ile yalıtılmış süreç grubu.

context switch → bağlam değiştirme 3
CPU'yu bir süreçten diğerine geçirmek için durumu kaydedip yenisini yüklemek.

contiguous allocation → bitişik tahsis 8
Her sürece (ya da dosyaya) tek parça, ardışık bir bölge verme.

control group (cgroup) → kontrol grubu 15
Süreç gruplarının CPU, bellek ve G/Ç kullanımını sınırlayan ve ölçen Linux mekanizması.

convoy effect → konvoy etkisi 5
Uzun bir işin arkasında kısa işlerin birikip beklemesi; FCFS çizelgelemede ortalama bekleme süresini büyütür.

cooperative multitasking → işbirlikçi çok görevlilik 3
İşletim sisteminin işlemciyi ancak program kendisi bıraktığında ya da sistem çağrısı yaptığında geri alabildiği düzen.

copy-on-write (COW) → yazınca kopyalama 9
Paylaşılan sayfanın yalnızca biri yazınca kopyalanması.

core dump → çekirdek dökümü 3
Bir sinyalle ölen sürecin bellek görüntüsünün, sistem ayarına göre yazıldığı dosya.

covert channel → gizli kanal 13
Güvenlik politikasının öngörmediği yoldan bilgi sızdıran kanal.

CPU burst → CPU patlaması 5
Sürecin iki G/Ç arasında kesintisiz CPU kullandığı süre.

CPU scheduling → CPU çizelgeleme 5
Hazır süreçlerden hangisinin CPU'yu alacağına karar verme.

CPU-bound / I/O-bound → CPU'ya bağlı / G/Ç'ye bağlı 3
Zamanının çoğunu hesaplamada ya da G/Ç beklemede geçiren süreç.

critical section (region) → kritik bölge 6
Paylaşılan veriye erişen ve aynı anda tek bir sürecin girmesi gereken kod parçası.

crossbar switch → çapraz çubuk anahtar 12
Her işlemciyi her bellek modülüne ayrı bir kesişim noktasıyla bağlayan, çakışmasız ama pahalı ara bağlantı.

C-SCAN → C-SCAN (dairesel SCAN) 11
Yalnız bir yönde hizmet veren SCAN türü: kafa uca varınca hizmet vermeden diskin başına döner.

D

data / task parallelism → veri / görev paralelliği 4
Veri paralelliğinde veri bölünür ve her çekirdek kendi parçasında aynı işlemi yapar; görev paralelliğinde her çekirdek farklı bir görev yürütür.

deadlock → kilitlenme 7
Her sürecin kümedeki başka bir sürecin tuttuğu kaynağı sonsuza dek beklemesi.

deadlock avoidance → kilitlenmeden kaçınma 7
Her isteği, sistemi güvensiz duruma sokmayacaksa kabul etme.

deadlock detection → kilitlenme tespiti 7
Sistemde kilitlenme olup olmadığını çizge ya da matrislerle bulma.

deadlock prevention → kilitlenmeyi önleme 7
Dört koşuldan birini yapısal olarak imkânsız kılma.

deadlock recovery → kilitlenmeden kurtarma 7
Kaynak geri alma, geri sarma ya da süreç sonlandırmayla kilitlenmeyi çözme.

defense in depth → derinlemesine savunma 13
Birbirinden bağımsız birkaç güvenlik katmanı kullanma ilkesi; biri aşılsa da ötekiler korur.

defragmentation → birleştirme 10
Diske dağılmış dosya bloklarını yeniden bitişik hâle getirme işlemi.

demand paging → talebe bağlı sayfalama 9
Sayfaları yalnızca ilk erişildiklerinde belleğe getirme.

denial of service → hizmet reddi 13
Sistemi meşru kullanıcılara hizmet veremez hâle getiren saldırı.

device controller → aygıt denetleyicisi 11
Aygıtı doğrudan süren, yazmaçları ve tamponu olan donanım birimi.

device driver → aygıt sürücüsü 11
Belirli bir denetleyiciyle konuşan, çekirdekte çalışan yazılım.

digital signature → dijital imza 13
Özel anahtarla imzalanan, açık anahtarla doğrulanan özet.

dining philosophers problem → yemek yiyen filozoflar problemi 6
Paylaşılan kaynaklarla kilitlenme ve açlığı gösteren klasik senkronizasyon problemi.

direct memory access (DMA) → doğrudan bellek erişimi (DMA) 11
Denetleyicinin CPU'yu meşgul etmeden veriyi doğrudan belleğe aktarması.

directory → dizin 10
Dosya adlarını dosyalara eşleyen özel dosya.

directory entry → dizin girdisi 10
Dizinde bir adı i-node numarasıyla eşleyen kayıt.

discretionary access control (DAC) → isteğe bağlı erişim denetimi 13
Nesnenin sahibinin kime hangi izni vereceğine kendisinin karar verdiği erişim denetimi.

disk scheduling → disk kolu çizelgeleme 11
Bekleyen disk isteklerinin sırasını seçme (SSTF, SCAN, C-LOOK).

dispatch latency → dağıtım gecikmesi 5
Dağıtıcının bir süreci durdurup ötekini başlatması için geçen süre.

dispatcher → görev dağıtıcı 5
Seçilen sürece CPU'yu veren modül: bağlam değiştirme ve mod geçişi.

distributed shared memory → dağıtık paylaşımlı bellek 12
Ayrı makinelerin belleklerini sayfa düzeyinde tek bir paylaşımlı adres alanı gibi gösteren yöntem.

dual-mode operation → çift modlu çalışma 2
Donanımın çekirdek ve kullanıcı modunu ayırarak İS'yi koruması.

dynamic loading → dinamik yükleme 8
Bir rutinin çağrılana kadar belleğe alınmaması; hiç çağrılmayan kod hiç yüklenmez.

E

earliest deadline first (EDF) → en erken son tarih önce (EDF) 5
Son tarihi en yakın görevi çalıştıran dinamik öncelikli algoritma.

Earliest Eligible Virtual Deadline First (EEVDF) → EEVDF 15
Linux 6.6'dan beri adil sınıfın seçim kuralı: uygun görevler arasından sanal son tarihi en erken olanı çalıştırır.

eBPF (extended Berkeley Packet Filter) → eBPF 14
Denetlenmiş küçük programların Linux çekirdeğinde güvenle çalıştırıldığı sanal makine.

effective access time (EAT) → etkin erişim süresi 8
TLB ya da sayfa hatası olasılıkları hesaba katılarak ortalama bellek erişim süresi.

eligibility → uygunluk 15
EEVDF'te bir görevin hak ettiğinden az CPU almış olması; yalnız uygun görevler seçilebilir.

encryption → şifreleme 13
Veriyi anahtar olmadan okunamaz hâle getirme.

entry / exit / remainder section → giriş / çıkış / kalan bölüm 6
Kritik bölgeye girmeden önce izin isteyen kod, çıkarken bunu duyuran kod ve kodun geri kalanı.

event loop → olay döngüsü 14
Olayları bekleyip sırayla ilgili işleyiciye gönderen döngü.

event-driven code → olay güdümlü kod 14
Akışı, gelen olaylara bağlanan işleyicilerin belirlediği program.

exception → istisna 11
Komutun yürütülmesi sırasında oluşan olağan dışı durum (sıfıra bölme, sayfa hatası).

executable file → çalıştırılabilir dosya 1
Yükleyicinin belleğe alıp çalıştırabileceği biçimdeki program dosyası (ELF, PE).

exokernel → exokernel 2
Yalnızca kaynakları korumalı biçimde dağıtan, soyutlamayı kütüphaneye bırakan çekirdek.

exploit → istismar 13
Bir zafiyetten yararlanarak saldırganın amacına ulaşmasını sağlayan kod ya da yöntem.

exponential average → üstel ortalama 5
Bir sonraki CPU patlamasını geçmiş patlamaların ağırlığı giderek azalan ortalamasıyla tahmin etme yöntemi.

exponential backoff → üstel geri çekilme 12
Başarısız bir kilit denemesinden sonra beklemek ve bekleme süresini her başarısızlıkta ikiye katlamak.

extensible system → genişletilebilir sistem 14
Çekirdeğe çalışma anında yeni işlev eklenebilen işletim sistemi.

external name → harici ad 14
Kullanıcının bir nesneye verdiği, insanın okuyabileceği ad (ör. dosya yolu).

F

fair-share scheduling → adil paylaşım çizelgelemesi 5
İşlemci payının süreçlere değil, kullanıcılara ya da gruplara verildiği çizelgeleme.

false sharing → yanlış paylaşım 12
Farklı değişkenlerin aynı önbellek satırında olması yüzünden oluşan gereksiz tutarlılık trafiği.

fetch–decode–execute cycle → getir–çöz–yürüt döngüsü 1
CPU'nun her komut için tekrarladığı temel çalışma döngüsü.

FIFO page replacement → FIFO algoritması 9
Sayfaların yüklenme sırasıyla bir kuyrukta tutulduğu ve yer gerekince en önce yüklenmiş sayfanın çıkarıldığı yer değiştirme algoritması.

file allocation table (FAT) → dosya tahsis tablosu (FAT) 10
Bağlı tahsiste blok zincirlerini diskin başındaki tek bir tabloda tutan yapı; i. girdi, i. bloktan sonra gelen bloğun numarasıdır.

file attributes (metadata) → öznitelik (üst veri) 10
Dosyanın içeriği dışındaki bilgileri: adı, boyutu, sahibi, izinleri, zaman damgaları gibi.

file descriptor → dosya tanımlayıcısı 10
Açık dosyayı temsil eden, süreç tablosundaki küçük tam sayı.

file locking → dosya kilitleme 15
Bir dosyanın ya da bölgesinin eş zamanlı erişime karşı kilitlenmesi.

file system → dosya sistemi 10
Diskte dosyaları ve dizinleri düzenleyen, adlandıran ve koruyan yazılım.

firewall → güvenlik duvarı 13
Ağ katmanında yalnız izinli bağlantıları geçiren savunma bileşeni.

first-come, first-served (FCFS) → ilk gelen ilk hizmet alır (FCFS) 5
Süreçleri geliş sırasıyla çalıştıran kesmesiz algoritma.

flash translation layer (FTL) → flaş çeviri katmanı (FTL) 10
SSD denetleyicisinde her yazmayı yeni, boş bir sayfaya yönlendirip mantıksal blok adresini o sayfaya eşleyen katman.

forward secrecy → ileriye dönük gizlilik 13
Uzun ömürlü anahtar sonradan ele geçse bile eski oturumların çözülememesi özelliği.

frame (page frame) → çerçeve 8
Fiziksel belleğin sayfayla aynı boyuttaki bloğu.

futex (fast userspace mutex) → futex 14
Çakışma yokken kullanıcı alanında, varsa çekirdekte bekleyen hızlı kilit.

fuzzing → bulanıklaştırma (fuzzing) 14
Programa rastgele ya da yarı rastgele girdiler vererek çökme ve hata arama tekniği.

G

gang scheduling → çete çizelgeleme 12
Birbirine bağlı iş parçacıklarını aynı zaman diliminde birlikte çalıştırma.

global / local replacement → genel / yerel yer değiştirme 9
Genelde kurban bütün çerçeveler arasından seçilir; yerelde süreç kurbanı yalnız kendi çerçevelerinden seçer.

graphical user interface (GUI) → GUI (grafik kullanıcı arayüzü) 1
Pencere, simge, menü ve işaretçiyle görsel etkileşim sağlayan arayüz.

guest operating system → misafir işletim sistemi 2
Sanal makinenin içinde çalışan işletim sistemi.

GUID partition table (GPT) → GPT 10
UEFI standardının parçası olan, 64 bitlik LBA kullanan ve her bölümü bir GUID ile tanımlayan bölümleme biçimi.

H

happens-before → önce-olur ilişkisi 4
Bir iş parçacığının yazmalarının başka bir iş parçacığına görünür olacağını güvenceye alan sıralama ilişkisi; örneğin join bunu sağlar.

hard / soft real-time → katı / yumuşak gerçek zamanlı 5
Katı gerçek zamanlı sistemde son tarihin kaçırılması başarısızlıktır; yumuşakta kritik görevler önceliklidir ama garanti yoktur.

hard disk drive (HDD) → sabit disk (HDD) 11
Dönen manyetik plakalarda veri saklayan ikincil depolama aygıtı.

hard link / symbolic link → katı bağ / sembolik bağ 10
Aynı i-node'u gösteren ikinci ad ya da başka bir yolu içeren özel dosya.

hash function (cryptographic) → özet fonksiyonu 13
Veriden sabit boyutlu, geri döndürülemez bir özet üreten fonksiyon.

hashed page table → hash sayfa tablosu 8
Sanal sayfa numarasının hash değeriyle aranan sayfa tablosu.

hierarchical page table → hiyerarşik sayfa tablosu 8
Sayfa tablosunu sayfalayarak çok düzeyli ağaç biçiminde tutmak.

hint → ipucu 14
Doğru olması gerekmeyen, yanlışsa ucuzca denetlenip düzeltilebilen önbelleklenmiş bilgi.

hit (cache) → isabet 8
Aranan verinin önbellekte bulunması.

hold and wait → tut ve bekle 7
Kaynak tutan sürecin yenisini beklemesi koşulu.

hole → delik 8
Değişken bölümlemede süreçlerin arasında kalan boş bellek bloğu.

hybrid kernel → hibrit çekirdek 2
Mikro çekirdek fikirlerini monolitik çekirdekle birleştiren yapı (Windows, macOS).

hypercall → hiperçağrı 12
Yarı sanallaştırmada misafir işletim sisteminin hassas komutlar yerine hipervizöre yaptığı doğrudan çağrı.

hypercube → hiperküp 12
n boyutlu küp biçiminde ara bağlantı; 2^n düğümün her biri n komşuya bağlıdır.

hypervisor → hipervizör 12
Sanal makineleri oluşturup çalıştıran yazılım katmanı.

I

i-node → i-node 10
Bir dosyanın özniteliklerini ve blok adreslerini tutan yapı.

indexed allocation → indisli tahsis 10
Dosyanın tüm blok adreslerini bir indis bloğunda (i-node) tutma.

indirection → dolaylılık 14
Bir nesneye doğrudan değil, araya konan bir ad ya da tablo üzerinden erişme; esneklik kazandırır.

internal / external fragmentation → iç / dış parçalanma 8
Tahsis edilen bölge içinde ya da bölgeler arasında kullanılamayan boş alan.

internal fragmentation → iç parçalanma 9
Ayrılan bloğun (ör. sayfanın) kullanılmayan iç kısmında boşa giden bellek.

internal name → dahili ad 14
Sistemin nesneyi içeride tanıdığı ad (ör. i-node numarası).

interprocess communication (IPC) → süreçler arası iletişim (IPC) 3
Süreçlerin veri alışverişi ve eşgüdümü için kullandığı yöntemler.

interrupt → kesme 11
Aygıtın ya da olayın CPU'nun akışını durdurup işleyiciye geçirmesi.

interrupt handler → kesme işleyicisi 11
Bir kesme geldiğinde çekirdekte çalışan yordam.

interrupt vector → kesme vektörü 11
Kesme numarasına göre işleyici adresini veren tablo girdisi.

interrupt-driven I/O → kesme güdümlü G/Ç 11
Aygıt hazır olunca kesmeyle CPU'yu uyaran G/Ç yöntemi.

inverted page table → ters sayfa tablosu 8
Fiziksel çerçeve başına bir girdi tutan tek, sistem çapında tablo.

isolation → yalıtım 14
Bir bileşenin hatasının ya da verisinin ötekileri etkilememesi.

J

journaling → günlükleme 10
Değişiklikleri önce günlüğe yazıp sonra yerine uygulayarak çökmeye dayanıklılık.

K

kernel → çekirdek 1
İşletim sisteminin çekirdek modunda çalışan, donanıma doğrudan erişen merkezî bölümü.

kernel mode → çekirdek modu 1
Tüm komutların ve donanımın kullanılabildiği ayrıcalıklı çalışma modu.

kernel thread → çekirdek iş parçacığı 14
Yalnız çekirdek modunda çalışan, çekirdeğin kendi işleri için açtığı iş parçacığı.

L

layered system → katmanlı yapı 2
Her katmanın yalnızca alt katmanın hizmetlerini kullandığı İS yapısı.

least recently used (LRU) → en uzun süredir kullanılmayan (LRU) 9
En uzun süredir erişilmemiş sayfayı çıkaran algoritma.

lightweight process (LWP) → hafif süreç 4
İş parçacıklarının eski adı; oluşturulması yeni bir süreçten çok daha ucuz olduğu için “hafif” denir.

limited direct execution → sınırlı doğrudan yürütme 3
Programın işlemcide tam hızla doğrudan çalışması; ama ayrıcalıklı işleri çekirdekten istemesi ve zamanlayıcıyla işlemciyi geri vermesi.

linked allocation → bağlı liste ile tahsis 10
Dosya bloklarını birbirine işaretçilerle bağlama.

linker → bağlayıcı 2
Nesne dosyalarını ve statik kütüphaneleri birleştirip sembolleri çözen ve çalıştırılabilir dosya üreten araç.

livelock → canlı kilit 7
Süreçlerin durumunu sürekli değiştirdiği ama ilerleyemediği durum.

load balancing → yük dengeleme 5
İşleri işlemciler arasında eşit dağıtmak (itme/çekme göçü).

loadable kernel module (LKM) → yüklenebilir çekirdek modülü 15
Çalışırken çekirdeğe eklenip çıkarılabilen kod parçası (sürücüler).

loader → yükleyici 2
Çalıştırılabilir dosyanın başlığını okuyup kod ve veri kesimlerini sürecin adres alanına eşleyen çekirdek bileşeni.

locality → yerellik 9
Programların kısa sürede küçük bir adres bölgesine erişme eğilimi.

lock-free → kilitsiz 6
Kilit yerine CAS gibi atomik komutlarla yazılan ve sistemin bütün olarak her zaman ilerlediği yapılar.

logical / physical address → mantıksal / fiziksel adres 8
CPU'nun ürettiği adres ile bellek biriminin gördüğü adres.

log-structured file system (LFS) → günlük yapılı dosya sistemi 10
Bütün diski tek bir günlük sayan, yazmaları bellekte biriktirip tek bir uzun sıralı yazmayla diske ekleyen dosya sistemi tasarımı.

lottery scheduling → piyango çizelgeleme 5
Süreçlere bilet verip rastgele çekilişle CPU dağıtan orantılı çizelgeleme.

M

mailbox → posta kutusu 3
Dolaylı ileti geçirmede iletilerin gönderildiği ve alındığı, tekil kimliği olan nesne (port).

main memory (RAM) → ana bellek 8
CPU'nun doğrudan eriştiği, programların ve verilerin çalışırken tutulduğu bellek.

malware → kötü amaçlı yazılım 13
Virüs, solucan, truva atı, fidye yazılımı gibi zararlı yazılımlar.

man-in-the-middle attack → ortadaki adam saldırısı 13
Saldırganın iki taraf arasına girip iletileri okuduğu ya da değiştirdiği saldırı.

mandatory access control (MAC) → zorunlu erişim denetimi 13
Sistem politikasının sahibin isteğinden bağımsız uyguladığı erişim denetimi (SELinux).

marshalling → sıralama (paketleme) 12
RPC'de koçanın prosedür parametrelerini ağ üzerinden gönderilecek bir iletiye paketlemesi.

master boot record (MBR) → MBR 10
Diskin ilk sektöründe önyükleyici kodu ve dört bölüm girdisi taşıyan eski bölümleme biçimi; 2 TiB'ın ötesini adresleyemez.

master-slave model → usta-köle modeli 12
İşletim sisteminin yalnız bir işlemcide çalıştığı, ötekilerin yalnız kullanıcı programlarını çalıştırdığı çok işlemcili düzen.

measured boot → ölçülü önyükleme 13
Her önyükleme bileşeninin çalıştırılmadan önce özetinin alınıp bir PCR'ye eklenmesi; son değer bütün zinciri özetler.

memory barrier (fence) → bellek bariyeri 6
Önceki bellek işlemlerinin sonrakilerden önce görünmesini zorlayan komut.

memory consistency model → bellek tutarlılık modeli 12
Farklı adreslere yapılan yazmaların başka çekirdeklerce hangi sırada görüleceğini belirleyen kurallar (ör. x86-TSO).

memory hierarchy → bellek hiyerarşisi 1
Hızlı-küçük-pahalı düzeylerden yavaş-büyük-ucuz düzeylere bellek katmanları.

memory management → bellek yönetimi 8
Belleği süreçlere tahsis etme, koruma ve geri alma işleri.

memory management unit (MMU) → bellek yönetim birimi (MMU) 8
Mantıksal adresleri çalışma anında fiziksel adrese çeviren donanım.

memory-mapped file → bellek eşlemeli dosya 9
Dosyanın sayfalarının adres alanına eşlenip bellek gibi okunup yazılması (mmap).

memory-mapped I/O → bellek eşlemeli G/Ç 11
Aygıt yazmaçlarının bellek adres alanına eşlenmesi.

MESI protocol → MESI protokolü 12
Önbellek satırının Değiştirilmiş, Özel, Paylaşılan ya da Geçersiz durumlarından birinde tutulduğu tutarlılık protokolü.

message → ileti 3
Süreçler ya da makineler arasında gönderilen veri birimi.

message passing → ileti geçirme 3
Süreçlerin send/receive ile çekirdek üzerinden ileti alışverişi yapması.

message-signaled interrupts (MSI) → MSI 11
Kesmenin bir kablo yerine, vektör numarasını içeren bir bellek yazmasıyla iletilmesi (genişletilmiş biçimi MSI-X).

microkernel → mikro çekirdek 2
Çekirdekte yalnızca IPC, çizelgeleme ve adres alanı yönetimini bırakan yapı.

middleware → ara yazılım 12
Dağıtık sistemde uygulamalara ortak bir model sunan yazılım katmanı.

monitor → monitör 6
Paylaşılan veriyi ve onu kullanan yordamları otomatik karşılıklı dışlamayla saran yapı.

monolithic kernel → monolitik çekirdek 2
Tüm İS hizmetlerinin tek adres alanında, çekirdek modunda çalıştığı yapı.

mount → bağlama 10
Bir dosya sistemini dizin ağacındaki bir noktaya eklemek.

multi-factor authentication → çok etkenli kimlik doğrulama 13
Bilinen, sahip olunan ve olunan şeylerden en az ikisiyle yapılan kimlik doğrulama.

multicomputer → çoklu bilgisayar 12
Her düğümün kendi belleği olan ve düğümlerin ileti ile haberleştiği sıkı bağlı bilgisayar topluluğu.

multicore → çok çekirdekli 1
Tek yongada birden çok bağımsız işlemci çekirdeği.

multilevel feedback queue (MLFQ) → çok düzeyli geri beslemeli kuyruk 5
Davranışına göre süreçleri kuyruklar arasında taşıyan çizelgeleme.

multilevel queue (MLQ) → çok düzeyli kuyruk 5
Süreçleri sabit sınıflara ve ayrı kuyruklara bölen çizelgeleme.

multiprogramming → çoklu programlama 1
Bellekte birden çok iş tutup biri G/Ç beklerken diğerini çalıştırma.

multithreaded process → çok iş parçacıklı süreç 4
Aynı adres alanını paylaşan birden çok iş parçacığı çalıştıran süreç.

mutex → mutex 6
Karşılıklı dışlama için kullanılan, sahibi olan ikili kilit.

mutual exclusion → karşılıklı dışlama 6
Kritik bölgede aynı anda en çok bir sürecin bulunması koşulu.

N

namespace → ad alanı 15
Süreç grubuna PID, ağ, bağlama gibi kaynakların ayrı bir görünümünü veren Linux özelliği.

network file system (NFS) → ağ dosya sistemi (NFS) 10
Uzak makinedeki dosyalara yerel gibi erişim sağlayan protokol.

no preemption → kesme yok 7
Kilitlenmenin dört koşulundan biri: bir kaynak, tutan süreç bırakmadıkça elinden alınamaz.

nonpreemptable resource → kesilemez kaynak 7
Sahibinden geri alınırsa işin bozulacağı kaynak (ör. yazıcı, kilit).

not recently used (NRU) → NRU (yakın zamanda kullanılmayan) 9
Sayfaları R ve M bitlerine göre dört sınıfa ayırıp hata anında boş olmayan en düşük sınıftan bir sayfa atan algoritma.

NUMA (non-uniform memory access) → NUMA 12
Her işlemcinin yerel belleğe uzak bellekten daha hızlı eriştiği çok işlemcili düzen.

O

omega network → omega ağı 12
Az sayıda 2×2 anahtardan kurulan, aşamalı ve daha ucuz bir işlemci–bellek ara bağlantısı.

OOM killer (out-of-memory killer) → OOM killer 3
Linux'ta bellek tükenince en çok bellek tutan süreçlerden birini seçip SIGKILL ile sonlandıran çekirdek mekanizması.

open-file table → açık dosya tablosu 10
Açık dosyaların konum ve durum bilgisini tutan çekirdek tablosu.

operating system (OS) → işletim sistemi (İS) 1
Donanımı yöneten ve uygulamalara soyutlamalar sunan temel yazılım.

operating system services → işletim sistemi hizmetleri 2
İS'nin kullanıcıya ve programlara sunduğu program yürütme, G/Ç, dosya, iletişim gibi işlevler.

optimal page replacement (OPT) → optimal algoritma (OPT) 9
En uzun süre kullanılmayacak sayfayı çıkaran, gerçeklenemeyen ölçüt algoritma.

orphan process → yetim süreç 3
Ebeveyni kendisinden önce sonlanmış süreç.

orthogonality → diklik 14
Birbirinden bağımsız kavramların serbestçe ve yan etkisiz birleştirilebilmesi.

P

page → sayfa 8
Mantıksal adres alanının sabit boyutlu bloğu.

page cache → sayfa önbelleği 10
Dosya verisini sayfa birimiyle bellekte tutan önbellek.

page fault → sayfa hatası 9
Bellekte olmayan bir sayfaya erişilince oluşan tuzak.

page locking (pinning) → sayfa sabitleme (kilitleme) 9
Bir çerçeveye kilit biti koyarak onu yer değiştirme adaylarının dışında tutmak; örneğin DMA süren tampon için.

page offset → sayfa içi konum 8
Mantıksal adresin, sayfa içindeki bayt konumunu veren alt n biti (d).

page replacement → sayfa yer değiştirme 9
Boş çerçeve yokken bellekten çıkarılacak sayfayı seçme.

page table → sayfa tablosu 8
Her sayfanın hangi çerçevede olduğunu tutan tablo.

page table entry (PTE) → sayfa tablosu girdisi 8
Çerçeve numarası ile geçerli, koruma, kirli ve referans bitlerini tutan girdi.

page-table base register (PTBR) → sayfa tablosu taban yazmacı (PTBR) 8
Ana bellekteki sayfa tablosunun başlangıç adresini tutan yazmaç (x86'da CR3); bağlam değiştirmede yalnız bu yazmaç değişir.

paging → sayfalama 8
Mantıksal belleği sayfalara, fiziksel belleği çerçevelere bölüp eşleme.

paging daemon → sayfalama daemonu 9
Periyodik uyanan, boş çerçeve azsa sayfaları önceden serbest bırakan ve kirli sayfaları diske yazan çekirdek iş parçacığı.

parallelism → paralellik 3
Birden çok işin gerçekten aynı anda, ayrı işlemcilerde çalışması.

paravirtualization → yarı sanallaştırma 12
Misafir İS'nin hipervizörle doğrudan çağrılarla (hiperçağrı) işbirliği yapması.

parent / child process → ebeveyn / çocuk süreç 3
fork ile süreç oluşturan süreç ebeveyn, oluşturulan süreç çocuktur.

parity → eşlik 11
Verilerin XOR'u; bir disk bozulunca kayıp bloğu yeniden kurmaya yarar.

partition → bölüm 10
Diskin ayrı bir dosya sistemi, takas alanı ya da şifreli kap taşıyabilen parçası; listesi diskin başındaki bir tabloda durur.

path name (absolute / relative) → yol adı (mutlak / göreli) 10
Bir dosyayı kökten ya da çalışma dizininden başlayarak gösteren ad.

permission bits (rwx) → izin bitleri 10
Sahip, grup ve diğerleri için okuma, yazma, çalıştırma hakları.

Peterson's algorithm → Peterson algoritması 6
İki süreç için yalnızca okuma/yazmayla karşılıklı dışlama sağlayan çözüm.

pipe → boru 3
İki süreci bağlayan, çekirdekte tamponlanan tek yönlü bayt akışı.

pipeline (CPU) → boru hattı 11
Komutların aşamalarını üst üste bindirerek her turda bir komut bitirme tekniği.

pipeline (shell) → boru hattı 15
Bir komutun çıktısını sonrakinin girdisine bağlayan komut zinciri.

plaintext / ciphertext → düz metin / şifreli metin 13
Şifrelemede okunabilir ileti (P) ve onun anahtarla dönüştürülmüş biçimi (C).

policy / mechanism → politika / mekanizma 1
Mekanizma bir işin nasıl yapılacağını, politika neyin yapılacağına karar verir.

polling → yoklama 11
CPU'nun aygıtın durum yazmacını hazır olana dek sürekli okuması.

port-mapped I/O → kapı eşlemeli G/Ç 11
Aygıt yazmaçlarının bellekten ayrı bir adres uzayına (G/Ç kapılarına) konduğu ve özel IN/OUT komutlarıyla erişildiği düzen.

position-independent code (PIC) → konumdan bağımsız kod 8
Mutlak adres kullanmayan, PC'ye göreli atlayan ve global veriye GOT üzerinden erişen kod; paylaşımlı kütüphaneler böyle derlenir.

preemptable resource → kesilebilir kaynak 7
Sahibinden zarar vermeden geri alınabilen kaynak (ör. CPU, bellek).

preemptive / nonpreemptive scheduling → kesmeli / kesmesiz çizelgeleme 5
Çalışan süreç CPU'dan zorla alınabilir (kesmeli) ya da kendisi bırakana dek çalışır (kesmesiz).

prepaging → ön sayfalama 9
Gerekeceği tahmin edilen sayfaları hata beklemeden, tek bir toplu okumayla önceden getirmek.

primitive → ilkel işlem 14
İşletim sisteminin sunduğu, başka işlemlerin üzerine kurulduğu temel işlem.

principle of least privilege → en az ayrıcalık ilkesi 13
Her bileşene yalnızca işi için gereken en az hakkı verme.

priority ceiling → öncelik tavanı 6
Öncelik terslemesine çözüm: kilidi alan görev, o kilidi kullanabilecek en yüksek öncelikli görevin önceliğine hemen yükseltilir.

priority inheritance → öncelik kalıtımı 6
Kilidi tutan sürecin geçici olarak bekleyenin önceliğini alması.

priority inversion → öncelik terslemesi 6
Yüksek öncelikli sürecin, düşük öncelikli sürecin tuttuğu kilidi beklemesi.

priority scheduling → öncelik çizelgeleme 5
En yüksek öncelikli hazır süreci çalıştıran algoritma.

privilege escalation → ayrıcalık yükseltme 13
Sıradan bir kullanıcının ya da sürecin bir hatadan yararlanarak yönetici haklarına ulaşması.

process → süreç 3
Yürütülmekte olan program; kendi adres alanı ve kaynakları vardır.

process control block (PCB) → süreç kontrol bloğu (PCB) 3
Bir sürecin durumu, yazmaçları ve kaynakları hakkında çekirdeğin tuttuğu kayıt.

process state → süreç durumu 3
Sürecin yaşam döngüsündeki konumu: yeni, hazır, çalışıyor, bekliyor, sonlandı.

processor affinity → işlemci afinitesi 5
Sürecin önbelleği sıcak kalsın diye aynı işlemcide çalıştırılması eğilimi.

producer–consumer problem → üretici–tüketici problemi 6
Sınırlı bir tamponu paylaşan üretici ve tüketicinin eşgüdümü problemi.

program counter (PC) → program sayacı 1
Sıradaki komutun adresini tutan yazmaç.

program status word (PSW) → program durum sözcüğü (PSW) 1
Koşul bitlerini, işlemcinin modunu (çekirdek ya da kullanıcı) ve kesmelerin açık olup olmadığını tutan yazmaç.

programmed I/O → programlanmış G/Ç 11
CPU'nun veriyi aygıta bayt bayt kendisinin aktardığı G/Ç.

progress → ilerleme 6
Kritik bölge çözümü koşulu: bölge boşsa kimin gireceğine yalnız girmek isteyen süreçler karar verir ve karar süresiz ertelenemez.

protection domain → koruma etki alanı 13
Bir sürecin erişebildiği (nesne, hak) çiftleri kümesi.

public key infrastructure → açık anahtar altyapısı 13
Sertifikaları veren, dağıtan ve iptal eden otoriteler ve kurallar bütünü.

public-key cryptography → açık anahtarlı şifreleme 13
Açık anahtarla şifrelenen veriyi yalnız özel anahtarın çözebildiği yöntem.

publish-subscribe → yayınla-abone ol 12
Gönderenin iletiyi konuya yayınladığı, o konuya abone olanların aldığı haberleşme modeli.

push / pull migration → itme / çekme göçü 5
Çok işlemcide yük dengeleme: itmede aşırı yüklü çekirdekten işler itilir, çekmede boşta kalan çekirdek yüklü olandan iş çeker.

Q

quota → kota 10
Her kullanıcının kullanabileceği blok ve dosya sayısına konan sınır.

R

race condition → yarış durumu 6
Sonucun paylaşılan veriye erişimin zamanlamasına bağlı olduğu hatalı durum.

ransomware → fidye yazılımı 13
Verileri şifreleyip fidye isteyen kötü amaçlı yazılım.

rate-monotonic scheduling (RM) → hız monoton çizelgeleme 5
Periyodu kısa olan göreve yüksek sabit öncelik veren gerçek zamanlı algoritma.

RAID → RAID 11
Birden çok diski performans ve dayanıklılık için birleştiren düzenler.

readers–writers problem → okur-yazar problemi 6
Çok okura birlikte, yazara tek başına izin veren erişim problemi.

ready queue → hazır kuyruğu 3
CPU'yu bekleyen hazır süreçlerin kuyruğu.

real-time operating system (RTOS) → gerçek zamanlı işletim sistemi 1
Sonucun belirli bir süre içinde üretilmesini garanti eden işletim sistemi.

reentrancy → yeniden girişlilik 14
Kodun kendisi yürürken yeniden çağrılabilmesi.

reentrant / thread-safe → yeniden girilir / iş parçacığı güvenli 4
Aynı anda birden çok çağrı ya da iş parçacığı tarafından güvenle yürütülebilen kod.

reference bit / dirty bit → referans biti / kirli bit 9
Sayfaya erişildiğini (R) ve sayfanın değiştirildiğini (M) gösteren bitler.

reference monitor → referans izleyici 13
Her erişim isteğini denetleyip kurala göre izin veren ya da reddeden bileşen.

reference string → referans dizisi 9
Algoritmaları karşılaştırmak için kullanılan sayfa erişim sırası.

register → yazmaç 1
İşlemci içindeki çok hızlı küçük depolama birimi.

regression → regresyon 14
Önceden çalışan bir özelliğin yeni bir değişiklikten sonra bozulması.

relocation → yer değiştirme 8
Programın bellekte farklı bir yere yüklenebilmesi için adreslerin kaydırılması.

relocation register → yer değiştirme yazmacı 8
Mantıksal adrese eklenerek fiziksel adresi veren yazmaç; Tanenbaum'un taban yazmacıyla aynıdır.

remote direct memory access (RDMA) → RDMA 12
NIC'in, çekirdek devreye girmeden karşı makinedeki bir sürecin belleğine doğrudan yazdığı ağ erişimi.

remote procedure call (RPC) → uzak prosedür çağrısı (RPC) 12
Başka makinedeki bir yordamı yerel çağrı gibi çağırma.

resource allocation graph → kaynak tahsis çizgesi 7
Süreç ve kaynak düğümleriyle istek ve tahsisleri gösteren yönlü çizge.

response time → tepki süresi 5
İstekten ilk yanıtın (ilk çalışmanın) başlamasına kadar geçen süre.

return-oriented programming (ROP) → dönüş yönelimli programlama 13
Programın ve kütüphanelerin içinde ret ile biten küçük kod parçalarının (gadget) adreslerini yığına dizerek saldırı kodu oluşturma tekniği.

round robin (RR) → döner sıralı çizelgeleme (round robin, RR) 5
Hazır süreçleri sırayla birer zaman dilimi çalıştıran kesmeli algoritma.

runqueue → çalışma kuyruğu 3
Linux'ta her işlemci çekirdeğine ayrı atanan hazır süreç kuyruğu; çekirdekler seçim yaparken birbirini beklemez.

S

safe / unsafe state → güvenli / güvensiz durum 7
Tüm süreçlerin bitebileceği bir sıranın var olduğu ya da olmadığı durum.

salt → tuz 13
Parola özetinden önce parolaya eklenen rastgele değer.

sandbox → korumalı alan 13
Programı sınırlı haklarla yalıtılmış bir ortamda çalıştırma.

SCAN (elevator algorithm) → SCAN (asansör algoritması) 11
Kafanın bir yönde diskin ucuna kadar gidip yolundaki istekleri karşıladığı, uca varınca yön değiştirdiği algoritma.

scatter-gather → dağıt-topla 11
DMA'da aygıta (adres, uzunluk) çiftleri listesi verilmesi; aygıt bitişik olmayan sayfalara tek işlemde yazar ya da onlardan okur.

scheduler → çizelgeleyici 5
Çizelgeleme kararını veren çekirdek bileşeni.

second chance → ikinci şans 9
Referans biti 1 olan sayfaya bir şans daha veren FIFO türevi.

secure boot → güvenli önyükleme 13
Açılış zincirindeki her yazılımın yüklenmeden önce imzasının doğrulanması.

seek time / rotational latency → arama süresi / dönme gecikmesi 11
Kolun ize gitme süresi ve sektörün kafanın altına gelme süresi.

segmentation → kesimleme 8
Adres alanını kod, veri, yığın gibi değişken boyutlu mantıksal kesimlere bölme.

semaphore → semafor 6
down/up (P/V) işlemleriyle kullanılan tam sayı sayaçlı senkronizasyon nesnesi.

sequential consistency (SC) → sıralı tutarlılık 6
Bütün iş parçacıklarının bellek işlemlerinin, her birinin program sırasını koruyan tek bir ortak sıraya dizilebilmesi.

shadow memory → gölge bellek 14
Her bellek baytının erişilebilir olup olmadığını kaydeden ayrı bellek bölgesi; KASAN bunu denetler.

shadow page table → gölge sayfa tablosu 12
Hipervizörün her misafir süreci için misafir sanal adresini doğrudan ana makine fiziksel adresine eşleyen tablosu.

shared library → paylaşımlı kütüphane 8
Belleğe bir kez yüklenip birçok süreçte kullanılan kütüphane (.so, .dll).

shared memory → paylaşımlı bellek 3
Birden çok sürecin adres alanına eşlenen ortak bellek bölgesi.

shell → kabuk 1
Kullanıcının komut yazarak işletim sistemiyle etkileştiği komut yorumlayıcısı.

shortest job first (SJF) → en kısa iş önce (SJF) 5
Sonraki CPU patlaması en kısa olan süreci seçen algoritma.

shortest remaining time first (SRTF) → en kısa kalan süre önce (SRTF) 5
SJF'nin kesmeli biçimi.

shortest seek time first (SSTF) → en kısa arama süresi önce (SSTF) 11
Her adımda kafaya en yakın bekleyen isteği seçen disk kolu çizelgeleme algoritması.

side channel → yan kanal 13
Süre, güç ya da önbellek davranışı gibi dolaylı bir yoldan gizli bilginin sızması.

signal → sinyal 15
Bir sürece bir olayı bildiren asenkron yazılım bildirimi (SIGKILL, SIGINT).

simultaneous multithreading (SMT) → eş zamanlı çoklu iş parçacığı (SMT) 12
Bir çekirdeğin iki donanım iş parçacığını aynı anda yürütmesi.

slab allocator → slab ayırıcı 9
Sık kullanılan çekirdek nesneleri için önceden hazırlanmış önbellekler sunan ayırıcı.

snooping (cache coherence) → dinleme 12
Her önbelleğin ortak veri yolundaki yazmaları izleyip kendi kopyasını güncellediği ya da geçersiz kıldığı tutarlılık yöntemi.

socket → soket 3
Ağ üzerinden ya da yerel olarak iki uç arasındaki iletişim uç noktası.

solid-state drive (SSD) → SSD 11
Hareketli parçası olmayan, NAND flash tabanlı depolama aygıtı.

space sharing → alan paylaşımı 12
Birbirine bağlı iş parçacıklarının aynı anda ayrı işlemcilere yerleştirildiği çizelgeleme.

space–time trade-off → uzay-zaman ödünleşimi 14
Bellek harcayarak zaman kazanma ya da tersi.

speedup → hızlanma 4
Eski çalışma süresinin yeni (paralel) çalışma süresine oranı.

spinlock → spinlock 6
Kilit açılana dek döngüde bekleyen kilit; kısa kritik bölgeler için.

spooling → biriktirme 1
Yavaş aygıta gidecek işleri diskte sıraya alıp sırayla gönderme (ör. yazdırma).

spurious wakeup → sahte uyanma 6
Bir koşul değişkeninde bekleyen iş parçacığının hiç sinyal gelmeden uyanması; bu yüzden koşul while ile denetlenir.

stable storage → kararlı depolama 11
Bir yazmanın ya doğru biçimde tamamlandığını ya da eski verinin yerinde kaldığını garanti eden depolama düzeni.

stack canary → yığın kanaryası 13
Dönüş adresinin önüne konan rastgele değer; değişmişse taşma saptanır ve program durdurulur.

stack pointer (SP) → yığın işaretçisi (SP) 1
Fonksiyon çağrılarında yerel değişkenlerin ve dönüş adreslerinin tutulduğu yığının üstünü gösteren yazmaç.

starvation → açlık 5
Bir sürecin sürekli başkalarına yenik düşüp hiç ilerleyememesi.

striping / mirroring → şeritleme / aynalama 11
RAID'de şeritleme ardışık blokları farklı disklere dağıtarak paralellik sağlar; aynalama her bloğun kopyasını başka bir diskte tutar.

stub → koçan 14
Henüz yazılmamış bir bileşenin yerine konan, yalnız arayüzünü taklit eden geçici kod.

superblock → süperblok 10
Dosya sisteminin boyut, blok sayısı gibi temel bilgilerini tutan blok.

swapping → takas 8
Sürecin ya da sayfaların bellek ile disk arasında taşınması.

symmetric multiprocessor (SMP) → simetrik çok işlemci (SMP) 12
Tek bir İS kopyasının tüm işlemcilerde eşit haklarla çalıştığı model.

synchronization → senkronizasyon 6
Eş zamanlı süreçlerin paylaşılan veriye erişimini düzenlemek.

synchronous / asynchronous communication → eş zamanlı / eş zamansız iletişim 14
Gönderenin yanıtı bekleyerek ya da beklemeden sürdüğü iletişim.

system call → sistem çağrısı 2
Bir programın çekirdekten hizmet istemek için kullandığı arayüz.

T

test and set lock (TSL) → TSL 12
Bir bellek sözcüğünü okuyup eski değerini yazmaca koyan ve sözcüğe 1 yazan bölünmez komut; çok işlemcide veri yolunu kilitler.

thrashing → thrashing (boşa çabalama) 9
Sistemin zamanının çoğunu sayfa taşımakla geçirmesi.

thread → iş parçacığı 4
Süreç içindeki bağımsız yürütme akışı; kendi PC'si, yazmaçları ve yığını vardır.

thread control block (TCB) → iş parçacığı kontrol bloğu (TCB) 4
İş parçacığının kimliğini, durumunu, kaydedilmiş PC ve yazmaçlarını, yığın işaretçisini ve önceliğini tutan yapı.

thread group ID (TGID) → iş parçacığı grubu kimliği 15
Linux'ta bir sürecin bütün iş parçacıklarının paylaştığı kimlik; kullanıcının gördüğü PID'dir.

thread pool → iş parçacığı havuzu 4
Önceden oluşturulmuş, kuyruktaki görevleri alan iş parçacıkları.

thread-local storage (TLS) → iş parçacığına yerel depolama (TLS) 4
Her iş parçacığının kendi kopyasını tuttuğu değişkenler.

threat → tehdit 13
Sistemin güvenlik hedeflerinden birini bozabilecek olası olay ya da saldırgan.

throughput → verim 5
Birim zamanda tamamlanan süreç sayısı.

ticket lock → bilet kilidi 12
Bekleyenlere sıra numarası veren ve kilidi bu sırayla dağıtan adil spinlock.

time of check to time of use (TOCTOU) → TOCTOU (kontrol zamanı ile kullanım zamanı) 13
Kaynağın denetlendiği an ile kullanıldığı an arasında saldırganın kaynağı değiştirebildiği yarış durumu açığı.

time sharing → zaman paylaşımı 12
CPU'yu kısa dilimlerle kullanıcılar arasında paylaştırarak etkileşimli kullanım sağlamak.

time slice (quantum) → zaman dilimi 5
Bir sürecin kesilmeden önce kullanabileceği en uzun CPU süresi.

timer → zamanlayıcı 2
Belirli süre sonunda kesme üreten donanım sayacı; CPU'nun İS'ye geri dönmesini sağlar.

TLB reach → TLB erişim alanı 8
TLB'nin kapsadığı bellek miktarı: girdi sayısı × sayfa boyu.

TLB shootdown → TLB vurusu 9
Bir sayfa eşlemesi değişince öteki çekirdeklerin TLB'lerindeki eski girdilerin de temizlenmesi.

top half / bottom half → üst yarı / alt yarı 11
Kesme işlemenin ikiye bölünmesi: kısa tutulan asıl işleyici (üst yarı) ve kesmeler açıkken çalışan ertelenmiş iş (alt yarı).

trade-off → ödünleşim 5
Bir özelliği iyileştirirken başka bir özellikten vazgeçmek.

translation lookaside buffer (TLB) → TLB 8
Son kullanılan sayfa–çerçeve eşlemelerini tutan küçük, hızlı önbellek.

trap → tuzak 2
Programın kendi isteğiyle (sistem çağrısı) ya da hatayla çekirdeğe geçmesi; yazılım kesmesi.

Trojan horse → Truva atı 13
Kullanıcının kendi isteğiyle kurduğu, yararlı görünen ama gizlice zararlı iş yapan yazılım.

trusted computing base → güvenilir bilgi işleme tabanı (TCB) 13
Güvenlik kurallarını uygulamakla görevli ve doğru çalıştığına güvenilmesi gereken donanım ve yazılımın bütünü.

Trusted Platform Module (TPM) → TPM (Güvenilir Platform Modülü) 13
Anakarta ya da işlemciye yerleşik, platform yapılandırma yazmaçları (PCR) içeren küçük güvenlik yongası.

tuple space → demet uzayı 12
Süreçlerin demet bırakıp eşleştirerek aldığı ortak, ilişkisel bellek (Linda).

turnaround time → dönüş süresi 5
Sürecin gelişinden bitişine kadar geçen süre.

two-phase locking (2PL) → iki aşamalı kilitleme 7
Önce bütün kilitlerin alındığı, sonra bırakıldığı protokol; bir kilit bırakıldıktan sonra yenisi alınmaz.

U

UMA (uniform memory access) → UMA 12
Her işlemcinin bellekteki her adrese aynı sürede eriştiği çok işlemcili düzen.

UMA / NUMA → UMA / NUMA 12
Bellek erişim süresinin tüm işlemciler için aynı olduğu ya da yakınlığa göre değiştiği mimari.

unikernel → unikernel 14
Uygulama ile gereken çekirdek işlevlerinin tek adres alanında, tek imaj olarak derlendiği yapı.

use-after-free → serbest sonrası kullanım 13
Programın serbest bıraktığı bir bloğa eski işaretçi üzerinden erişmeyi sürdürmesiyle oluşan açık.

user mode → kullanıcı modu 1
Ayrıcalıklı komutların yasak olduğu, uygulamaların çalıştığı mod.

user-level / kernel-level thread → kullanıcı düzeyi / çekirdek düzeyi iş parçacığı 4
Kütüphanece ya da çekirdekçe yönetilen iş parçacıkları.

V

valid–invalid bit → geçerli / geçersiz biti 8
Sayfanın adres alanında ve bellekte olup olmadığını gösteren bit.

virtual address → sanal adres 9
Sürecin gördüğü, MMU'nun fiziksel adrese çevirdiği adres.

virtual deadline → sanal son tarih 15
EEVDF'te uygun görevler arasından seçim için kullanılan, istek boyuna göre hesaplanan son tarih.

virtual file system (VFS) → sanal dosya sistemi (VFS) 10
Farklı dosya sistemlerini ortak bir arayüzün arkasında birleştiren çekirdek katmanı.

virtual machine (VM) → sanal makine 2
: Gerçek bir bilgisayar gibi davranan, yazılımla oluşturulmuş makine.

virtual memory → sanal bellek 9
: Süreçlere fiziksel bellekten büyük, kesintisiz bir adres alanı sunan teknik.

virtual memory area (VMA) → sanal bellek alanı 15
: Linux'ta bir sürecin adres alanının aynı izinleri ve kaynağı paylaşan ardışık bölgesi.

virtual runtime → sanal çalışma süresi 15
: CFS ve EEVDF'te bir görevin önceliğine göre ölçeklenmiş birikmiş CPU süresi.

virtualization → sanallaştırma 1
: Tek bir fiziksel kaynağı birden çok sanal kaynak gibi göstermek.

vulnerability → zafiyet 13
: Bir saldırının yararlanabileceği tasarım ya da gerçekleme kusuru.

W

wait queue → bekleme kuyruğu 3
: Belirli bir olayı (G/Ç'nin bitmesi, bir çocuğun sonlanması gibi) bekleyen süreçlerin tutulduğu kuyruk.

wait-for graph → bekleme çizgesi 7
: Kaynak düğümleri kaldırılmış kaynak tahsis çizgesi; bir döngü kilitlenme demektir.

waiting time → bekleme süresi 5
: Sürecin hazır kuyruğunda geçirdiği toplam süre.

wear leveling → aşınma dengeleme 11
: Yazmaları flash blokları arasında eşit dağıtarak ömrü uzatma.

work stealing → iş çalma 4
: Boşta kalan iş parçacığının başka kuyruktan görev alması.

working set → çalışma kümesi 9
: Sürecin son Δ erişimde kullandığı sayfalar kümesi.

worm → solucan 13
: Ağdaki yamalanmamış bir hizmeti kullanarak kullanıcıdan bağımsız biçimde makineden makineye kendini kopyalayan program.

write-invalidate / write-update → yazınca geçersiz kıl / yazınca güncelle 12
: Önbellek tutarlılığı politikaları: yazmadan önce öteki kopyalar geçersiz kılınır ya da yeni değer bütün kopyalara gönderilir.

write-through / write-back → anında yazma / geri yazma 10
: Anında yazma her değişikliği hemen diske gönderir; geri yazma kirli blokları bellekte biriktirip sonra yazar.

Z

zero trust → sıfır güven 13
: Güvenlik duvarının içini de güvenilir saymayan; ağın içinden gelen isteği de kimlik doğrulamasından ve yetki denetiminden geçiren yaklaşım.

zero-fill-on-demand → sıfırla doldurulan sayfa 9
: BSS ve yeni heap sayfalarına ilk erişimin sıfırlanmış bir çerçeveyle karşılanması; sıfırlama güvenlik gereğidir.

zombie process → zombi süreç 3
: Bitmiş ama çıkış durumu ebeveyn tarafından alınmamış süreç.

I

I/O scheduler → G/Ç çizelgeleyicisi 11
: Blok isteklerini sıralayıp birleştiren çekirdek katmanı (mq-deadline, bfq).

EK B

İşletim Sistemleri İçin Ön Bilgiler

İŞLETİM SİSTEMLERİ · ÖN BİLGİLER

İşletim sistemleri yazılımı, donanımı, algoritmaları ve alt düzey programlamayı bir araya getirir. Aşağıdaki alanlarda temel bir birikim; çekirdeğin davranışını, süreçlerin yürütülmesini, bellek yönetimini, dosya sistemlerini ve aygıtlarla etkileşimi anlamayı çok kolaylaştırır.

Temel hazırlık
En yararlı hazırlık; programlama, veri yapıları, bilgisayar mimarisi, bellek, komut satırı araçları ve temel ağ bilgisi üzerine pratik deneyimdir.

TEMELLER · PROGRAMLAMA · DONANIM

Temel Bilgi Birikimi

Bu konular süreçlerin, çizelgelemenin, senkronizasyonun, sanal belleğin, dosya sistemlerinin, G/Ç'nin, ağın ve korumanın teknik temelini oluşturur.

Algoritmalar

Problem çözme · verimlilik · çizelgeleme

İşletim sistemi CPU zamanı, bellek, G/Ç (giriş/çıkış) ve kaynak dağıtımı konusunda sürekli karar verir.

BİLİNMESİ GEREKENLER

Veri Yapıları

Gösterim · arama · kaynak izleme

Çekirdek; süreçleri, bellek bölgelerini, açık dosyaları, aygıtları, zamanlayıcıları ve ağ durumunu veri yapılarıyla tutar.

BİLİNMESİ GEREKENLER

Karmaşıklık, çizelgelemedeki ödünleşimler, arama, sıralama, özyineleme ve durum izleme yararlı temellerdir.

Karmaşıklık **Çizelgeleme** **Arama**

Özetle: Karmaşıklık, çizelgelemedeki ödünleşimler, arama, sıralama, özyineleme ve durum izleme yararlı temellerdir.

Diziler, bağlı listeler, yığınlar, kuyruklar, ağaçlar, hash tabloları ve biteşlemler (*bitmap*) rahatça kullanılabilmelidir.

Listeler **Kuyruklar** **Ağaçlar**

Hash tabloları

Özetle: Diziler, bağlı listeler, yığınlar, kuyruklar, ağaçlar, hash tabloları ve biteşlemler rahatça kullanılabilmelidir.

Matematik Temelleri

Sayı sistemleri · çizgeler · olasılık

Sayfa tablosu boyutu, adresin sayfa numarası ve konuma bölünmesi, kilitlenme çizgeleri, bankacı algoritmasının matrisleri, etkin erişim süresi ve anahtar değişimi hep temel ayrık matematik ve olasılık üzerine kuruludur.

BİLİNMESİ GEREKENLER

İkinin kuvvetleri ve KiB/MiB/GiB birimleri, ikili ve onaltılık (*hexadecimal*) gösterim, logaritma, modüler aritmetik, kümeler, matrisler, yönlü çizgeler ve döngüler, ağırlıklı ortalama ve basit olasılık hesabı tanıdık olmalıdır.

İkinin kuvvetleri **Onaltılık** **Çizgeler**

Olasılık

Özetle: İkinin kuvvetleri ve KiB/MiB/GiB birimleri, ikili ve onaltılık gösterim, logaritma, modüler aritmetik, kümeler, matrisler, yönlü çizgeler ve döngüler, ağırlıklı

Bilgisayar Mimarisi

CPU · yazmaçlar · ayrıcalık · bellek

İşletim sistemi doğrudan donanım üzerinde çalışır. Koruma ve denetim için mimarinin sunduğu düzeneklere dayanır.

BİLİNMESİ GEREKENLER

Yazmaçlar (*register*), komut yürütme, ayrıcalık düzeyleri, kesmeler (*interrupt*), istisnalar, önbellekler ve adres çevirisi bilinmelidir.

Yazmaçlar **Kesmeler** **MMU**

Ayrıcalık

Özetle: Yazmaçlar, komut yürütme, ayrıcalık düzeyleri, kesmeler, istisnalar, önbellekler ve adres çevirisi bilinmelidir.

ortalama ve basit olasılık hesabı tanıdık olmalıdır.

C ile Programlama

İşaretçiler · bellek · sistem programlama

Klasik çekirdek arayüzlerinin çoğu ve birçok işletim sistemi gerçeklemesi, C tarzı sistem programlamaya sıkı sıkıya bağlıdır.

BİLİNMESİ GEREKENLER

İşaretçiler (*pointer*), diziler, yapılar (*struct*), bit işlemleri, elle bellek yönetimi ve fonksiyon çağrıları anlaşılmış olmalıdır.

İşaretçiler **Yapılar** **Bellek** **Bit işlemleri**

Özetle: İşaretçiler, diziler, yapılar, bit işlemleri, elle bellek yönetimi ve fonksiyon çağrıları anlaşılmış olmalıdır.

Süreç ve İş Parçacığı Temelleri

Yürütme · eş zamanlılık · durum

Çizelgeleme ve senkronizasyondan önce programların nasıl yürütüldüğünü ve iş parçacıklarının süreç kaynaklarını nasıl

Derleme ve Assembly Okuma

Derleyici · bağlayıcı · makine komutları

Ders örnekleri C kodunu gcc ile derler, kütüphanelerle bağlar ve sistem çağrılarını, tuzakları ve atomik işlemleri birkaç satırlık assembly ile gösterir.

BİLİNMESİ GEREKENLER

Kaynak dosyadan nesne dosyasına ve çalıştırılabilir dosyaya giden yol, statik ve dinamik kütüphaneler, derleyici iyileştirmelerinin kodu nasıl değiştirebildiği ve kısa bir assembly listesinin C koduna nasıl karşılık geldiği temel düzeyde bilinmelidir.

gcc **Bağlama** **Çalıştırılabilir dosya** **Assembly**

Özetle: Kaynak dosyadan nesne dosyasına ve çalıştırılabilir dosyaya giden yol, statik ve dinamik kütüphaneler, derleyici iyileştirmelerinin kodu nasıl değiştirebildiği ve kısa bir assembly listesinin C koduna nasıl karşılık geldiği temel düzeyde bilinmelidir.

Bellek Temelleri

Adresler · yığın · heap · sanal bellek

İşletim sistemi fiziksel belleği yönetir ve her sürece denetimli bir adres alanı (*address space*) sunar.

BİLİNMESİ GEREKENLER

paylaştığını bilmek işe yarar.

BİLİNMESİ GEREKENLER

Program, süreç (*process*) ve iş parçacığı (*thread*) arasındaki fark; yığın (*stack*), heap ve yürütme durumu temel düzeyde bilinmelidir.

Süreçler **İş parçacıkları** **Yığın** **Heap**

Özetle: Program, süreç ve iş parçacığı arasındaki fark; yığın, heap ve yürütme durumu temel düzeyde bilinmelidir.

Adresler, baytlar, yığın ile heap arasındaki fark, yerellik (*locality*) ve sanal adreslerin çevrilmesi gerektiği fikri tanıdık olmalıdır.

Adresler **Yığın** **Heap** **Yerellik**

Özetle: Adresler, baytlar, yığın ile heap arasındaki fark, yerellik ve sanal adreslerin çevrilmesi gerektiği fikri tanıdık olmalıdır.

Sayısal Mantık ve Donanım

Bitler · aygıtlar · veri yolları · kesmeler

İşletim sistemi işlemcilerle, bellekle ve aygıtlarla donanımın tanımladığı arayüzler üzerinden konuşur.

BİLİNMESİ GEREKENLER

İkili sayılar, veri yolları, aygıt yazmaçları, kesmeler ve doğrudan bellek erişimi (DMA) hakkında temel bilgi, alt düzey G/Ç'yi anlamayı çok kolaylaştırır.

İkili sayılar **Aygıtlar** **Kesmeler** **DMA**

Özetle: İkili sayılar, veri yolları, aygıt yazmaçları, kesmeler ve DMA hakkında temel bilgi, alt düzey G/Ç'yi anlamayı çok kolaylaştırır.

Komut Satırı ve Linux Araçları

Kabuk · dosyalar · süreçler · tanılama

Birçok işletim sistemi kavramı, komut satırında incelendiğinde somutlaşır.

BİLİNMESİ GEREKENLER

Dosyalar, izinler, süreçler, bellek, diskler ve günlükler için ps, top, chmod, df ve dmesg gibi komutlarla pratik yapmak yararlıdır.

Kabuk **Süreçler** **İzinler** **Günlükler**

Özetle: Dosyalar, izinler, süreçler, bellek, diskler ve günlükler için ps, top, chmod, df ve dmesg gibi komutlarla pratik yapmak yararlıdır.

Ağ Temelleri

Soketler · TCP/IP · portlar

Güvenlik Temelleri

Koruma · kimlik doğrulama · izinler

Modern işletim sistemleri bir ağ yığını içerir ve iletişimi soket (*socket*) API'leriyle sunar.

BİLİNMESİ GEREKENLER

IP adresleri, portlar, TCP ile UDP arasındaki fark, istemci/sunucu davranışı ve temel soket kavramları bilinmelidir.

TCP/IP **Soketler** **Portlar**

Özetle: IP adresleri, portlar, TCP ile UDP arasındaki fark, istemci/sunucu davranışı ve temel soket kavramları bilinmelidir.

İşletim sistemi kullanıcılar, süreçler, dosyalar ve aygıtlar arasındaki sınırları korur.

BİLİNMESİ GEREKENLER

Kimlik doğrulama (*authentication*), yetkilendirme (*authorization*), izinler, en az ayrıcalık ilkesi ve temel tehdit modellerinin ardındaki fikirler bilinmelidir.

İzinler **Kimlik doğrulama** **Koruma**

Özetle: Kimlik doğrulama, yetkilendirme, izinler, en az ayrıcalık ilkesi ve temel tehdit modellerinin ardındaki fikirler bilinmelidir.

Dersten önce en önemlileri

C ile programlama, işaretçiler, veri yapıları, süreçler, bellek ve temel bilgisayar mimarisi.

Dersle birlikte öğrenilebilecekler

Kabuk araçları, Linux'ta süreçleri inceleme, ağ komutları, izinler ve sistem tanılama.

Çekirdek deneyimi gerekmez

Sistem çağrıları, çizelgeleme, senkronizasyon, sanal bellek, dosya sistemleri ve G/Ç derste temelden ele alınır.

Hazırlık Kontrol Listesi

Aşağıdaki ifadelerin çoğu tanıdık geliyorsa hazırlık yeterli demektir.

✓ Temel C programlarını okuyup yazabiliyorum, gcc ile derleyip çalıştırabiliyorum.

✓ İşaretçileri, dizileri, yapıları, yığını ve heap'i anlıyorum.

✓ Kuyrukları, bağlı listeleri, ağaçları ve hash tablolarını biliyorum.

✓ İkili ve onaltılık sayılarla, ikinin kuvvetleriyle ve basit olasılıkla hesap yapabiliyorum.

✓ CPU yazmaçlarının ve kesmelerin ne olduğunu biliyorum.

✓ Terminali ve temel Linux komutlarını kullanabiliyorum.

✓ Dosyaları, süreçleri, izinleri ve temel ağ kavramlarını anlıyorum.

EK C

İşletim Sistemlerinin Öncüleri

İŞLETİM SİSTEMLERİ · TARİHSEL BAKIŞ

İşletim sistemleri onlarca yıllık bir birikimin ürünüdür. Zaman paylaşımı (*time sharing*), eş zamanlılık (*concurrency*), bellek yönetimi, etkileşimli kullanım, taşınabilirlik, ağ, sanallaştırma, güvenlik ve açık sistemler üzerine yapılan çalışmalar bugünkü yapıyı biçimlendirdi.

İlk bilgisayarlardan modern işletim sistemlerine
İşletim sistemlerinin ve modern bilgisayar biliminin gelişimini etkileyen kişiler, sistemler ve fikirler.

KİŞİLER · SİSTEMLER · MÜHENDİSLİK

İşletim Sistemlerinde Önemli İsimler

İşletim sistemi mimarisine, eş zamanlılığa, bellek yönetimine, dosya sistemlerine, sanallaştırmaya, güvenliğe, ağa ve sistem yazılımına büyük katkı yapan isimler.

John von Neumann

Depolanmış program modeli · 1940'lar

Komutların ve verinin aynı bellekte durduğu depolanmış program (*stored-program*) modelinin biçimlenmesine katkı yaptı.

John McCarthy

Zaman paylaşımı fikri · 1950'ler–1960'lar

Güçlü bir bilgisayarı birçok kullanıcının zaman paylaşımıyla aynı anda kullanması fikrini savundu. Bu öneriyi 1959'da MIT'de yazdığı bir notla ortaya koydu.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Mimari, bellek, yürütme, ayrıcalıklı denetim ve program yükleme bu modele dayanır.

Mimari **Bellek** **Yürütme**

Neden önemli: Mimari, bellek, yürütme, ayrıcalıklı denetim ve program yükleme bu modele dayanır.

1940'lar

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Zaman paylaşımı; çizelgeleme (*scheduling*), zamanlayıcı kesmeleri, bağlam değiştirme (*context switch*) ve çok kullanıcılı koruma ihtiyacını doğurdu.

Zaman paylaşımı **Çizelgeleme** **Çok kullanıcılı**

Neden önemli: Zaman paylaşımı; çizelgeleme, zamanlayıcı kesmeleri, bağlam değiştirme ve çok kullanıcılı koruma ihtiyacını doğurdu.

1950'ler–60'lar

Fernando Corbató

CTSS ve Multics · 1960'lar

MIT'de CTSS ve Multics gibi etkili zaman paylaşımlı sistem projelerini yönetti. Bu çalışmalarla 1990'da Turing Ödülü'nü (*Turing Award*) aldı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Multics; koruma halkalarını, hiyerarşik dosya sistemini, dinamik bağlamayı ve çok kullanıcılı kullanımı ileri taşıdı.

CTSS **Multics** **Koruma**

Neden önemli: Multics; koruma halkalarını, hiyerarşik dosya sistemini, dinamik bağlamayı ve çok kullanıcılı kullanımı ileri taşıdı.

1960'lar

Edsger W. Dijkstra

Senkronizasyon ve işletim sistemi tasarımı · 1960'lar

Semaforu (*semaphore*) tanıttı ve katmanlı THE sistemini tasarladı. Senkronizasyonun ve eş zamanlı programlamanın temellerini attı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Kritik bölge (*critical section*), karşılıklı dışlama (*mutual exclusion*), kilitlenme (*deadlock*) ve açlık (*starvation*) bugün de işletim sistemlerinin temel sorunlarıdır.

Semafor **Senkronizasyon** **Kilitlenme**

Neden önemli: Kritik bölge, karşılıklı dışlama, kilitlenme ve açlık bugün de işletim sistemlerinin temel sorunlarıdır.

1960'lar

Peter J. Denning

Sanal bellek ve çalışma kümesi · 1960'lar

Yerelliği (*locality*) ve sanal bellek başarımını açıklamak için çalışma kümesi (*working set*) modelini geliştirdi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Bu model sayfa hatalarını (*page fault*), bellek baskısını ve boşa çabalamayı (*thrashing*) açıklar.

Sanal bellek **Çalışma kümesi** **Boşa çabalama**

Neden önemli: Bu model sayfa hatalarını, bellek baskısını ve boşa çabalamayı açıklar.

1960'lar

Dennis Ritchie

C ve UNIX · Bell Labs · 1970'ler

C dilini yarattı ve UNIX'i Ken Thompson ile birlikte geliştirdi. Böylece taşınabilir sistem yazılımı mümkün oldu.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

UNIX'in 1973'te büyük ölçüde C ile yeniden yazılması, bir işletim sisteminin farklı donanımlara taşınabileceğini gösterdi.

C **UNIX** **Taşınabilirlik**

Neden önemli: UNIX'in 1973'te büyük ölçüde C ile yeniden yazılması, bir işletim

1960'lar

Ken Thompson

UNIX · Bell Labs · 1969'dan itibaren

UNIX'i Dennis Ritchie ile birlikte geliştirdi. Basit ve birbirine eklenebilen araçlara dayanan yalın bir tasarım anlayışı kurdu.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

UNIX; süreç (*process*), dosya tanımlayıcısı (*file descriptor*), boru (*pipe*), kabuk (*shell*) ve hiyerarşik dosya sistemi kavramlarını biçimlendirdi.

UNIX **Süreçler** **Dosya sistemleri**

Neden önemli: UNIX; süreç, dosya tanımlayıcısı, boru, kabuk ve hiyerarşik dosya sistemi kavramlarını biçimlendirdi.

1969+

Per Brinch Hansen

Eş zamanlı programlama · 1970'ler

Eş zamanlı programlama ve işletim sistemi yapısı için etkili soyutlamalar geliştirdi. RC 4000 çekirdeği ve Concurrent Pascal dili bu çalışmaların ürünüdür.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Monitöre (*monitor*) dayalı senkronizasyon, paylaşılan veriye erişimi daha düzenli hâle getirdi.

Monitörler **Eş zamanlılık** **Sistem yapısı**

sisteminin farklı donanımlara taşınabileceğini gösterdi.

1970'ler

Neden önemli: Monitöre dayalı senkronizasyon, paylaşılan veriye erişimi daha düzenli hâle getirdi.

1970'ler

C. A. R. Hoare

Monitörler ve CSP · 1970'ler

Eş zamanlılık çalışmalarına monitörleri ve Communicating Sequential Processes (CSP, iletişen ardışık süreçler) modelini kazandırdı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Bu çalışmalar senkronizasyon, kanallar ve ileti geçirme (*message passing*) için açık modeller sunar.

CSP **Monitörler** **IPC**

Neden önemli: Bu çalışmalar senkronizasyon, kanallar ve ileti geçirme için açık modeller sunar.

Butler Lampson

Sistem tasarımı · Xerox PARC · 1970'ler

Alto gibi etkili kişisel bilgisayar projelerine ve dağıtık sistem çalışmalarına katkı yaptı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Sistem çalışmaları koruma, adlandırma, ağ ve güvenilir arayüz tasarımını bir araya getirir.

Koruma **Dağıtık sistemler** **Alto**

Neden önemli: Sistem çalışmaları koruma, adlandırma, ağ ve güvenilir arayüz tasarımını bir araya getirir.

Richard Stallman

1970'ler **GNU Projesi · 1983'ten itibaren**

UNIX uyumlu, özgür bir yazılım sistemi kurmak için GNU Projesi'ni başlattı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

GNU derleyicileri, kütüphaneleri, kabukları ve araçları UNIX benzeri sistemlerin merkezine yerleşti.

GNU **Özgür yazılım** **Araç zinciri**

Andy Hertzfeld

1970'ler+ **Macintosh sistem yazılımı · 1980'ler**

İlk Macintosh'un sistem yazılımı ve Macintosh Toolbox üzerinde çalıştı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

İlk Macintosh yazılımı, masaüstü grafik kullanıcı arayüzünden (*GUI*) beklentileri ve sıkı bütünleşik sistem hizmetlerini etkiledi.

Macintosh **GUI** **Sistem yazılımı**

Neden önemli: GNU derleyicileri, kütüphaneleri, kabukları ve araçları UNIX benzeri sistemlerin merkezine yerleşti.

1983+

Neden önemli: İlk Macintosh yazılımı, masaüstü grafik kullanıcı arayüzünden beklentileri ve sıkı bütünleşik sistem hizmetlerini etkiledi.

1980'ler

Andrew S. Tanenbaum

MINIX ve işletim sistemi eğitimi · 1980'lerden itibaren

MINIX'i geliştirdi ve etkili işletim sistemi ders kitapları yazdı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

MINIX; mikro çekirdek (*microkernel*), süreçler arası iletişim (IPC), süreç ve dosya sistemi kavramlarını bütün bir sistem üzerinde incelenebilir kılar.

MINIX **Mikro çekirdek** **Eğitim**

Neden önemli: MINIX; mikro çekirdek, IPC, süreç ve dosya sistemi kavramlarını bütün bir sistem üzerinde incelenebilir kılar.

Ken Sakamura

TRON ve gömülü sistemler · 1980'lerden itibaren

1987+

Gömülü ve gerçek zamanlı sistemlerde etkili olan TRON projesini 1984'te başlattı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

ITRON belirlenimci çizelgelemeyi, kesme gecikmesini ve son tarihe göre tasarımı öne çıkarır.

David Cutler

VMS ve Windows NT · 1970'ler–1990'lar

DEC'te RSX-11M ve VMS gibi büyük işletim sistemi projelerini, ardından Microsoft'ta Windows NT'yi yönetti.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Windows NT; korumalı bellek, güvenlik, eş zamansız G/Ç (giriş/çıkış) ve taşınabilirlik için güçlü soyutlamalar kurdu.

VMS **Windows NT**

Çekirdek mimarisi

Neden önemli: Windows NT; korumalı bellek, güvenlik, eş zamansız G/Ç ve taşınabilirlik için güçlü soyutlamalar kurdu.

Linus Torvalds

Linux çekirdeği · 1991'den itibaren

1970'ler–90'lar

Linux çekirdeğini yarattı. Bu çekirdek bugün sunucularda, bulut sistemlerinde, Android'de, gömülü aygıtlarda ve süper bilgisayarlarda çalışır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Linux, büyük bir monolitik çekirdeğin (*monolithic kernel*) içinde modüler alt sistemlerin nasıl kurulabileceğini gösterir.

TRON **RTOS** **Gömülü**

Neden önemli: ITRON belirlenimci çizelgelemeyi, kesme gecikmesini ve son tarihe göre tasarımı öne çıkarır.

1980'ler+

Linux **Çekirdek** **Açık geliştirme**

Neden önemli: Linux, büyük bir monolitik çekirdeğin içinde modüler alt sistemlerin nasıl kurulabileceğini gösterir.

1991+

J. C. R. Licklider

Etkileşimli bilgisayar vizyonu · 1960'lar

Tek başına çalışan toplu iş makinelerinin ötesinde, etkileşimli ve ağa bağlı bir bilgisayar kullanımını savundu.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Etkileşimli kullanım; düşük gecikmeyi, zaman paylaşımını ve uzaktan erişimi önemli hâle getirdi.

Etkileşimli kullanım **Zaman paylaşımı** **Ağ**

Neden önemli: Etkileşimli kullanım; düşük gecikmeyi, zaman paylaşımını ve uzaktan erişimi önemli hâle getirdi.

Douglas Engelbart

Etkileşimli sistemler · 1960'lar

NLS projesini yönetti. Etkileşimli kullanımı, hiper metni, ortak çalışma araçlarını ve fareyi 1968'deki ünlü gösterimde tanıttı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Bu çalışmalar olay güdümlü G/Ç'ye, ekranlara ve hızlı tepki veren grafik ortamlara olan ihtiyacı artırdı.

Etkileşim **Giriş aygıtları** **GUI tarihi**

Neden önemli: Bu çalışmalar olay güdümlü G/Ç'ye, ekranlara ve hızlı tepki veren grafik ortamlara olan ihtiyacı artırdı.

Douglas McIlroy

1960'lar **UNIX boruları ve yazılım araçları · 1970'ler**

UNIX borularını önerdi. Borular 1973'te UNIX'e eklendi. McIlroy "yazılım araçları" anlayışını da derinden etkiledi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Borular, bağımsız programların standart giriş ve çıkış akışları üzerinden birbirine bağlanmasını sağlar.

Brian Kernighan

1960'lar **UNIX araçları ve sistem programlama · 1970'lerden itibaren**

UNIX yazılım ortamına katkı yaptı. AWK gibi etkili araçları ve programlama kitaplarını başkalarıyla birlikte yazdı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Bu çalışmalar komut satırında araçları birleştirmeyi, betik yazmayı ve taşınabilir sistem programlamayı biçimlendirdi.

Borular **UNIX** **Yazılım araçları**

Neden önemli: Borular, bağımsız programların standart giriş ve çıkış akışları üzerinden birbirine bağlanmasını sağlar.

1970'ler

UNIX **AWK** **Araçlar**

Neden önemli: Bu çalışmalar komut satırında araçları birleştirmeyi, betik yazmayı ve taşınabilir sistem programlamayı biçimlendirdi.

1970'ler+

John Lions

UNIX kaynak koduyla eğitim · 1970'ler

UNIX Version 6 çekirdeğinin kaynak kodunu satır satır açıklayan etkili bir çalışma hazırladı: *Lions' Commentary on UNIX 6th Edition*.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Süreç tabloları, tuzaklar (*trap*), çizelgeleme, i-node'lar ve sistem çağrısı (*system call*) yolları bu çalışma sayesinde öğrencilere açıldı.

UNIX V6 **Çekirdeğin iç yapısı**

Eğitim

Neden önemli: Süreç tabloları, tuzaklar, çizelgeleme, i-node'lar ve sistem çağrısı yolları bu çalışma sayesinde öğrencilere açıldı.

1970'ler

Bill Joy

BSD UNIX · 1970'ler–1980'ler

BSD UNIX'in başlıca geliştiricilerinden biridir. vi düzenleyicisini ve csh kabuğunu yazdı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

BSD; ağ, iş istasyonları ve sonraki UNIX türevleri üzerinde büyük etki bıraktı.

BSD **UNIX** **Ağ**

Neden önemli: BSD; ağ, iş istasyonları ve sonraki UNIX türevleri üzerinde büyük etki bıraktı.

1970'ler–80'ler

Marshall Kirk McKusick

BSD ve Fast File System · 1980'lerden itibaren

BSD'nin önde gelen geliştiricilerindendir. Özellikle Fast File System (FFS, hızlı dosya sistemi) ve FreeBSD çalışmalarıyla tanınır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Gerald J. Popek

Sanallaştırma kuramı · 1970'ler

Bir mimarinin sanal makineleri hangi koşullarda verimli biçimde destekleyebileceğini belirleyen biçimsel ölçütleri 1974'te Robert P. Goldberg ile birlikte ortaya koydu.

FFS, dosya sisteminde yerelliği, blok tahsisini ve disk başarımını iyileştirdi.

FFS **BSD** **Dosya sistemleri**

Neden önemli: FFS, dosya sisteminde yerelliği, blok tahsisini ve disk başarımını iyileştirdi.

1980'ler+

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Popek–Goldberg ölçütleri, hipervizörlerin (*hypervisor*) ve tuzakla öykünmeye (*trap-and-emulate*) dayalı sanallaştırmanın temeli oldu.

Sanallaştırma **Hipervizörler**

Yalıtım

Neden önemli: Popek–Goldberg ölçütleri, hipervizörlerin ve tuzakla öykünmeye dayalı sanallaştırmanın temeli oldu.

1970'ler

Robert P. Goldberg

Sanal makine araştırmaları · 1970'ler

Sanal makine izleyicilerinin (*virtual machine monitor, VMM*) biçimsel olarak incelenmesinin temellerini attı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Bu çalışmalar eşdeğerliği, kaynak denetimini ve verimliliği bir VMM'nin temel özellikleri olarak belirledi.

VMM **Sanal makineler**

Kaynak denetimi

Neden önemli: Bu çalışmalar eşdeğerliği, kaynak denetimini ve verimliliği bir VMM'nin temel özellikleri olarak belirledi.

Leslie Lamport

Dağıtık sistemler · 1970'lerden itibaren

Dağıtık sistemlerde olayların sırası, saatler ve doğruluk üzerine akıl yürütmenin temel yöntemlerini geliştirdi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Mantıksal saatler ve "önce gelir" (*happens-before*) ilişkisi, dağıtık eşgüdümün ve çoğaltmanın temelini oluşturur.

Mantıksal saatler **Dağıtık sistemler**

Sıralama

Neden önemli: Mantıksal saatler ve "önce gelir" ilişkisi, dağıtık eşgüdümün ve çoğaltmanın temelini oluşturur.

Barbara Liskov

1970'ler

Soyutlama ve dağıtık sistemler · 1970'lerden itibaren

Tom Kilburn

1970'ler+

Atlas ve sanal bellek · 1950'ler–1960'lar

Programlama dillerinde soyutlamaya (*abstraction*) ve hataya dayanıklı dağıtık sistem tasarımına öncülük etti.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Bu çalışmalar modüler arayüzleri, atomik işlemleri ve güvenilir hizmet tasarımını güçlendirdi.

Soyutlama **Dağıtık sistemler**

Hata toleransı

Neden önemli: Bu çalışmalar modüler arayüzleri, atomik işlemleri ve güvenilir hizmet tasarımını güçlendirdi.

1970'ler+

Manchester'daki bilgisayar projelerini yönetti. Bunlardan Atlas (1962), sanal bellek tarihinde bir dönüm noktasıdır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Atlas; sayfalamanın (*paging*), bellek ile depolama arasında kendiliğinden veri taşımanın ve adres çevirisinin uygulamada işe yaradığını gösterdi.

Atlas **Sayfalama** **Sanal bellek**

Neden önemli: Atlas; sayfalamanın, bellek ile depolama arasında kendiliğinden veri taşımanın ve adres çevirisinin uygulamada işe yaradığını gösterdi.

1950'ler–60'lar

Maurice Wilkes

EDSAC ve mikroprogramlama · 1940'lar–1950'ler

EDSAC'ı (1949) geliştiren ekibi yönetti. İşlemci denetimi için mikroprogramlamayı 1951'de önerdi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Bu çalışmalar donanım–yazılım arayüzünü ve yeniden kullanılabilir sistem yazılımı uygulamalarını biçimlendirdi.

EDSAC **Mikroprogramlama**

Sistem yazılımı

Neden önemli: Bu çalışmalar donanım–yazılım arayüzünü ve yeniden kullanılabilir sistem yazılımı uygulamalarını biçimlendirdi.

David Wheeler

Alt programlar ve ilk sistem yazılımları · 1940'lar–1950'ler

EDSAC üzerinde yeniden kullanılabilir alt programlara ve ilk programlama yöntemlerine önemli katkılar yaptı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Alt program çağrıları zamanla çağrı kurallarına, yığınlara (*stack*), kütüphanelere ve çalışma zamanı desteğine dönüştü.

Alt programlar **EDSAC**

Sistem yazılımı

Neden önemli: Alt program çağrıları zamanla çağrı kurallarına, yığınlara, kütüphanelere ve çalışma zamanı desteğine dönüştü.

1940'lar–50'ler

Niklaus Wirth

Oberon sistemi · 1980'lerden itibaren

Pascal, Modula-2 ve Oberon gibi etkili programlama dilleri tasarladı. Jürg Gutknecht ile birlikte küçük ve derli toplu Oberon işletim sistemini geliştirdi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Oberon; dilin ve işletim sisteminin birlikte tasarlanabileceğini, yalınlığı ve tutarlı bir etkileşimli sistem kurmayı gösterdi.

Oberon **Sistem tasarımı** **Diller**

Neden önemli: Oberon; dilin ve işletim sisteminin birlikte tasarlanabileceğini, yalınlığı ve tutarlı bir etkileşimli sistem kurmayı gösterdi.

1980'ler+

Auguste Kerckhoffs

Kerckhoffs ilkesi · 1883

1883'te yayımlanan *La cryptographie militaire* makalesinde askerî şifreler için altı kural sıraladı. Bunların en bilineni, sistemin düşman eline geçse bile güvenli kalması gerektiğini söyler: güvenlik algoritmanın gizliliğine değil, yalnız anahtarın gizliliğine dayanmalıdır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

1940'lar–50'ler

Robert Morris Sr.

UNIX güvenliği · 1970'lerden itibaren

UNIX üzerinde çalıştı. Parola güvenliği ve sistem koruması alanında etkili katkılar yaptı; Ken Thompson ile yazdığı *Password Security: A Case History* (1979) bunların en bilinenidir.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Bu çalışmalar kimlik doğrulamayı (*authentication*), tuzlu (*salted*) parola özetlemeyi, ayrıcalık ayrımını ve tehdit modellemeyi birbirine bağlar.

Güvenlik **UNIX** **Parolalar**

Neden önemli: Bu çalışmalar kimlik doğrulamayı, tuzlu parola özetlemeyi, ayrıcalık ayrımını ve tehdit modellemeyi birbirine bağlar.

1970'ler+

Fred Brooks

OS/360 ve *The Mythical Man-Month* · 1960'lar–1970'ler

IBM'de System/360 donanımının ardından 1964'te gecikmiş OS/360 projesinin yönetimini üstlendi. Deneyimlerini 1975'te yayımlanan *The Mythical Man-Month* kitabında anlattı; kitaptaki en ünlü gözlem, geciken bir yazılım projesine yeni insan eklemenin projeyi daha da geciktirdiğidir (Brooks yasası). 1999'da Turing Ödülü'nü aldı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

İlke, açık tasarımın ve geniş anahtar uzayının neden önemli olduğunu açıklar; işletim sistemlerinin kullandığı şifreleme ve kimlik doğrulama düzenekleri bu varsayımla tasarlanır.

Kerckhoffs ilkesi **Şifreleme**

Anahtar

Neden önemli: İlke, açık tasarımın ve geniş anahtar uzayının neden önemli olduğunu açıklar; işletim sistemlerinin kullandığı şifreleme ve kimlik doğrulama düzenekleri bu varsayımla tasarlanır.

1883

OS/360, büyük bir işletim sisteminin karmaşıklığını yöneten ilk geniş çaplı denemelerden biriydi; Brooks'un dersleri işletim sistemi tasarımında ekip, arayüz ve kavramsal bütünlük tartışmalarının hâlâ çıkış noktasıdır.

OS/360 **Yazılım mühendisliği**

Brooks yasası

Neden önemli: OS/360, büyük bir işletim sisteminin karmaşıklığını yöneten ilk geniş çaplı denemelerden biriydi; Brooks'un dersleri işletim sistemi tasarımında ekip, arayüz ve kavramsal bütünlük tartışmalarının hâlâ çıkış noktasıdır.

1964–1975

László Bélády

OPT algoritması ve Bélády anomalisi · 1960'lar

IBM Watson Araştırma Merkezi'nde çalıştı. 1966'da *IBM Systems Journal*'da yayımlanan çalışmasında, gelecekte en geç kullanılacak sayfayı çıkaran algoritmanın (OPT ya da MIN) en az sayfa hatasını verdiğini gösterdi. 1969'da Nelson ve Shedler ile birlikte, FIFO'da çerçeve sayısı artınca sayfa hatası sayısının da artabildiğini buldu; bu durum Bélády anomalisi diye anılır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

OPT gerçeklenemez, ama öteki yer değiştirme algoritmalarının ölçüldüğü alt sınırdır. Anomali de LRU gibi yığın algoritmalarını FIFO'dan ayıran özelliği gösterir.

Gene Amdahl

Amdahl yasası · 1967

IBM System/360 mimarisinin baş tasarımcılarındandır. 1967'deki AFIPS konferansında, bir programın paralelleştirilemeyen seri kısmının hızlanmaya kesin bir üst sınır koyduğunu savundu. Bu gözlem sonradan Amdahl yasası adıyla formüle edildi: seri pay s ise, çekirdek sayısı ne kadar artarsa artsın hızlanma 1/s'yi geçemez.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Yasa, çok çekirdekli işlemcilerde iş parçacığı sayısını artırmanın neden bir noktadan sonra az kazandırdığını açıklar ve kilitlerle seri kesimleri kısaltmanın önemini gösterir.

Sayfa yer değiştirme **OPT**

Bélády anomalisi

Neden önemli: OPT gerçeklenemez, ama öteki yer değiştirme algoritmalarının ölçüldüğü alt sınırdır. Anomali de LRU gibi yığın algoritmalarını FIFO'dan ayıran özelliği gösterir.

1966–1969

Donald Knuth

Dinamik bellek ayırma ve %50 kuralı · 1968

1968'de yayımlanan *The Art of Computer Programming* serisinin ilk cildinde dinamik bellek ayırmayı (ilk uyan, en iyi uyan, buddy (kardeş) sistemi) ayrıntılı biçimde çözümledi. %50 kuralı bu çözümlemeden çıkar: ilk uyanla çalışan bir sistemde N tahsisli blok varken ortalamada yaklaşık 0,5 N boş delik bulunur, yani belleğin yaklaşık üçte biri parçalanma yüzünden kullanılamaz hâle gelebilir.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Kural, dış parçalanmanın ne kadar büyük olabileceğini tek bir sayıyla özetler ve sayfalama gibi parçalanmayı ortadan kaldıran yöntemlere neden ihtiyaç duyulduğunu açıklar.

Dış parçalanma **İlk uyan**

%50 kuralı

Neden önemli: Kural, dış parçalanmanın ne kadar büyük olabileceğini tek bir sa-

Amdahl yasası **Hızlanma** **Paralellik**

Neden önemli: Yasa, çok çekirdekli işlemcilerde iş parçacığı sayısını artırmanın neden bir noktadan sonra az kazandırdığını açıklar ve kilitlerle seri kesimleri kısaltmanın önemini gösterir.

1967

Edward G. Coffman Jr.

Kilitlenmenin dört koşulu · 1971

M. J. Elphick ve A. Shoshani ile 1971'de *ACM Computing Surveys*'de yayımladığı "System Deadlocks" makalesinde kilitlenmenin dört gerekli koşulunu sıraladı: karşılıklı dışlama, tut ve bekle, kesme yok ve döngüsel bekleme. Bu koşullar onun adıyla Coffman koşulları diye anılır. Makale, her türden birden çok örneği olan kaynaklar için bir kilitlenme tespit algoritması da verir.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Kilitlenmeyi önleme yöntemlerinin hepsi bu dört koşuldan birini bozmaya dayanır; tespit algoritması da bugün ders kitaplarında anlatılan biçimin temelidir.

Kilitlenme **Coffman koşulları**

Tespit

Neden önemli: Kilitlenmeyi önleme yöntemlerinin hepsi bu dört koşuldan birini

yıyla özetler ve sayfalama gibi parçalanmayı ortadan kaldıran yöntemlere neden ihtiyaç duyulduğunu açıklar.

1968

Richard C. Holt

Kaynak tahsis çizgesi · 1971–1972

"Some Deadlock Properties of Computer Systems" çalışmasını 1971'de SOSP'ta sundu, genişletilmiş biçimini 1972'de *ACM Computing Surveys*'de yayımladı. Süreçleri ve kaynakları iki tür düğüm, istekleri ve atamaları da kenar olarak gösteren çizge modeli bugün kaynak tahsis çizgesi adıyla bilinir.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Çizge, kilitlenmenin nasıl oluştuğunu görünür kılar ve tespit algoritmalarının üzerinde çalıştığı veri yapısıdır: her kaynaktan tek örnek varsa çizgedeki döngü kilitlenme demektir.

Kaynak tahsis çizgesi **Kilitlenme**

Çizge modeli

Neden önemli: Çizge, kilitlenmenin nasıl oluştuğunu görünür kılar ve tespit algoritmalarının üzerinde çalıştığı veri yapısıdır: her kaynaktan tek örnek varsa çizgedeki döngü kilitlenme demektir.

1972

bozmaya dayanır; tespit algoritması da bugün ders kitaplarında anlatılan biçimin temelidir.

1971

C. L. Liu ve James W. Layland

Hız monoton çizelgeleme ve EDF · 1973

MIT Project MAC'ten Liu ile Jet Propulsion Laboratory'den Layland, 1973'te *Journal of the ACM*'de yayımladıkları makalede periyodik görevlerin katı gerçek zamanlı çizelgelenmesini incelediler. Hız monoton (RM) önceliklerin sabit öncelikli yöntemler arasında en iyisi olduğunu kanıtladılar ve n görev için $n(2^{1/n} - 1)$ kullanım sınırını verdiler. Aynı makale, toplam kullanımı 1'i aşmayan her görev kümesinin en erken son tarih önce (EDF) ile çizelgelenebildiğini gösterir.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Liu–Layland testi, bir görev kümesinin son tarihlerini kaçırmadan çalışıp çalışamayacağını çizelgeyi kurmadan söyler; gerçek zamanlı işletim sistemlerindeki öncelik atamalarının kuramsal dayanağıdır.

Gerçek zamanlı çizelgeleme **RM**

EDF

Neden önemli: Liu–Layland testi, bir görev kümesinin son tarihlerini kaçırmadan çalışıp çalışamayacağını çizelgeyi kurmadan söyler; gerçek zamanlı işletim

sistemlerindeki öncelik atamalarının kuramsal dayanağıdır.

1973

David E. Bell ve Leonard J. LaPadula

Bell-LaPadula modeli · 1973

MITRE Corporation'da 1973'te çok düzeyli güvenlik için biçimsel bir model geliştirdiler. Modelin iki temel kuralı vardır: basit güvenlik özelliğine göre bir süreç kendi düzeyinin üstündeki nesneleri okuyamaz (yukarı okuma yok); * özelliğine göre de kendi düzeyinin altındaki nesnelere yazamaz (aşağı yazma yok).

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Model, bilginin yüksek gizlilik düzeyinden alçak düzeye sızmasını kurallarla engeller; zorunlu erişim denetiminin (MAC) ve askerî sınıflandırma düzenlerinin bilgisayardaki karşılığıdır.

Çok düzeyli güvenlik **Gizlilik**

Zorunlu erişim denetimi

Neden önemli: Model, bilginin yüksek gizlilik düzeyinden alçak düzeye sızmasını kurallarla engeller; zorunlu erişim denetiminin (MAC) ve askerî sınıflandırma düzenlerinin bilgisayardaki karşılığıdır.

1973

Whitfield Diffie ve Martin Hellman

Açık anahtarlı şifreleme · 1976

Jerome Saltzer ve Michael Schroeder

Koruma ve güvenli tasarım ilkeleri · 1975

1975'te *Proceedings of the IEEE*'de yayımladıkları "The Protection of Information in Computer Systems" makalesinde güvenli sistem tasarımı için sekiz ilke sıraladılar. Düzeneğin sade tutulması, varsayılanın reddetmek olması, her erişimin denetlenmesi, açık tasarım, ayrıcalık ayrımı ve en az ayrıcalık bu ilkeler arasındadır. Saltzer, Multics projesinde de çalışmıştı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

İlkeler, erişim denetimi, setuid programlar ve ayrıcalık ayrımı gibi işletim sistemi düzeneklerini değerlendirmek için bugün de kullanılan ortak bir ölçüt sunar.

Koruma **En az ayrıcalık**

Tasarım ilkeleri

Neden önemli: İlkeler, erişim denetimi, setuid programlar ve ayrıcalık ayrımı gibi işletim sistemi düzeneklerini değerlendirmek için bugün de kullanılan ortak bir ölçüt sunar.

1975

Kenneth J. Biba

Biba bütünlük modeli · 1977

MITRE Corporation'daki "Integrity Considerations for Secure Computer Systems" raporunda (1977) bütünlüğü

1976'da *IEEE Transactions on Information Theory*'de yayımladıkları "New Directions in Cryptography" makalesinde açık anahtarlı şifreleme fikrini ve adlarıyla anılan anahtar değişimini ortaya koydular. İki taraf, güvensiz bir kanal üzerinden yalnız açık değerler göndererek ortak bir gizli anahtar üretebilir. 2015 Turing Ödülü'nü birlikte aldılar.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Önceden paylaşılmış bir anahtar gerektirmeyen bu değişim, TLS el sıkışmasının ve dijital imzaların temelidir; işletim sistemlerinin ağ üzerinden kimlik doğrulama ve güvenli iletişim hizmetleri buna dayanır.

Açık anahtar **Anahtar değişimi**

TLS

Neden önemli: Önceden paylaşılmış bir anahtar gerektirmeyen bu değişim, TLS el sıkışmasının ve dijital imzaların temelidir; işletim sistemlerinin ağ üzerinden kimlik doğrulama ve güvenli iletişim hizmetleri buna dayanır.

1976

Gary L. Peterson

Peterson algoritması · 1981

1981'de *Information Processing Letters*'ta yayımlanan "Myths About the Mutual Exclusion Problem" makalesinde, iki süreç için yalnız okuma ve yazma işlemleriyle doğru çalışan kısa bir karşılıklı dışlama çözümü verdi. Algoritma, her sürecin "girkorumak için Bell-LaPadula modelinin tersi kurallara dayanan bir model tanımladı. Bir süreç kendinden düşük bütünlük düzeyindeki nesneleri okuyamaz (aşağı okuma yok) ve kendinden yüksek düzeydeki nesnelere yazamaz (yukarı yazma yok).

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Model, güvenilmeyen verinin güvenilir bileşenleri bozmasını engeller; Windows'taki bütünlük düzeyleri gibi zorunlu bütünlük denetimleri bu fikrin uygulamalarıdır.

Bütünlük **Çok düzeyli güvenlik**

Zorunlu erişim denetimi

Neden önemli: Model, güvenilmeyen verinin güvenilir bileşenleri bozmasını engeller; Windows'taki bütünlük düzeyleri gibi zorunlu bütünlük denetimleri bu fikrin uygulamalarıdır.

1977

Richard W. Carr ve John L. Hennessy

WSClock · 1981

Stanford Üniversitesi'nde geliştirdikleri WSClock algoritmasını 1981'de SOSP'ta sundular. Algoritma, çalışma kümesi fikrini saat algoritmasının dairesel çerçeve lis-

mek istiyorum" bayrağını sıranın kimde olduğunu tutan tek bir değişkenle birleştirir.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Algoritma, kritik bölge probleminin üç koşulunu (karşılıklı dışlama, ilerleme, sınırlı bekleme) sağlayan çözümün klasik örneğidir; modern işlemcilerde bellek bariyeri olmadan neden bozulduğu da bellek sıralamasının önemini gösterir.

Kritik bölge **Karşılıklı dışlama**

Bellek sıralaması

Neden önemli: Algoritma, kritik bölge probleminin üç koşulunu (karşılıklı dışlama, ilerleme, sınırlı bekleme) sağlayan çözümün klasik örneğidir; modern işlemcilerde bellek bariyeri olmadan neden bozulduğu da bellek sıralamasının önemini gösterir.

1981

Andrew D. Birrell ve Bruce J. Nelson

Uzak prosedür çağrısı (RPC) · 1984

Xerox PARC'ta Cedar ortamı için geliştirdikleri RPC paketini 1984'te *ACM Transactions on Computer Systems*'te yayımlanan "Implementing Remote Procedure Calls" makalesinde anlattılar. Başka bir makinedeki bir prosedür, otomatik üretilen istemci ve sunucu koçanları sayesinde sıradan bir yerel çağrı sözdizimiyle çağrılabilir.

tesiyle birleştirir: el çerçeveler üzerinde sırayla gezinir ve tüm tabloyu taramadan ilk uygun kurbanda durur.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

WSClock, çalışma kümesi modelinin pratikte uygulanabilir bir biçimidir; sayfa yer değiştirmede yerellik bilgisini düşük maliyetle kullanır.

WSClock **Çalışma kümesi**

Sayfa yer değiştirme

Neden önemli: WSClock, çalışma kümesi modelinin pratikte uygulanabilir bir biçimidir; sayfa yer değiştirmede yerellik bilgisini düşük maliyetle kullanır.

1981

John Gustafson

Gustafson yasası · 1988

Sandia Ulusal Laboratuvarları'nda 1024 işlemcili bir sistem üzerindeki çalışmalarına dayanarak 1988'de *Communications of the ACM*'de "Reevaluating Amdahl's Law" notunu yayımladı. Çekirdek sayısı artınca genellikle problemin de büyüdüğünü, bu durumda ölçeklenmiş hızlanmanın çekirdek sayısıyla neredeyse doğrusal arttığını gösterdi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

MÜHENDİSLİK AÇISINDAN ÖNEMİ

RPC, ileti geçirmenin ayrıntılarını programcıdan gizleyen temel dağıtık sistem soyutlamasıdır; NFS gibi dağıtık dosya sistemleri ve bugünkü gRPC benzeri çerçeveler bu modele dayanır.

RPC **Koçanlar** **Dağıtık sistemler**

Neden önemli: RPC, ileti geçirmenin ayrıntılarını programcıdan gizleyen temel dağıtık sistem soyutlamasıdır; NFS gibi dağıtık dosya sistemleri ve bugünkü gRPC benzeri çerçeveler bu modele dayanır.

1984

Amdahl "bu iş ne kadar hızlı biter?", Gustafson ise "aynı sürede ne kadar büyük bir iş yapılabilir?" sorusunu cevaplar; ikisi birlikte paralel sistemlerin ölçeklenmesini değerlendirmenin iki yüzüdür.

Ölçeklenmiş hızlanma **Paralellik** **Ölçeklenebilirlik**

Neden önemli: Amdahl "bu iş ne kadar hızlı biter?", Gustafson ise "aynı sürede ne kadar büyük bir iş yapılabilir?" sorusunu cevaplar; ikisi birlikte paralel sistemlerin ölçeklenmesini değerlendirmenin iki yüzüdür.

1988

David Patterson, Garth Gibson ve Randy Katz

RAID · 1988

UC Berkeley'de 1988'de SIGMOD'da sundukları "A Case for Redundant Arrays of Inexpensive Disks (RAID)" makalesinde, pahalı tek büyük disk yerine ucuz küçük disklerden oluşan ve fazlalıkla korunan diziler kurmayı önerdiler. Makale beş RAID düzeyi tanımlayıp maliyet ve başarımlarını karşılaştırdı. Kısaltmadaki "I" sonradan *independent* olarak yeniden tanımlandı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Şeritleme, aynalama ve eşlik bilgisi bugün işletim sistemlerinin yazılım RAID katmanlarında ve depolama denetleyicilerin-

Mendel Rosenblum ve John Ousterhout

Günlük yapılı dosya sistemi (LFS) · 1991

Sprite işletim sistemi için geliştirdikleri günlük yapılı dosya sistemini (LFS) 1991'de SOSP'ta sundular. Tasarım, okumaların çoğunun önbellekten karşılanacağını ve diske giden trafiğin ağırlıkla yazma olacağını varsayar; bu yüzden bütün değişiklikler diske tek bir uzun sıralı günlük olarak yazılır. Günlük kesimlere bölünür ve bir temizleyici dağınık kesimlerdeki canlı veriyi sıkıştırarak boş alan açar.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

LFS, yazmaları sıralı hâle getirerek küçük dosya yükünde büyük başarım kazancı sağladı ve çökme sonrası kurtarmayı hız-

de kullanılan temel güvenilirlik ve başarım yöntemleridir.

RAID **Fazlalık** **Depolama**

Neden önemli: Şeritleme, aynalama ve eşlik bilgisi bugün işletim sistemlerinin yazılım RAID katmanlarında ve depolama denetleyicilerinde kullanılan temel güvenilirlik ve başarım yöntemleridir.

1988

landırdı; fikirleri günlüklemeli ve yazınca kopyalayan dosya sistemlerinde, SSD'lerin çeviri katmanlarında yaşamaya devam eder.

LFS **Günlük yapılı**

Çökme kurtarma

Neden önemli: LFS, yazmaları sıralı hâle getirerek küçük dosya yükünde büyük başarım kazancı sağladı ve çökme sonrası kurtarmayı hızlandırdı; fikirleri günlüklemeli ve yazınca kopyalayan dosya sistemlerinde, SSD'lerin çeviri katmanlarında yaşamaya devam eder.

1991

John Mellor-Crummey ve Michael L. Scott

MCS kuyruk kilidi · 1991

1991'de *ACM Transactions on Computer Systems*'te yayımlanan makalelerinde, adlarının baş harfleriyle anılan MCS kuyruk kilidini önerdiler. Bekleyenler bir bağlı liste oluşturur; her bekleyen ayrı bir önbellek satırındaki kendi bayrağında döner ve kilidi bırakmak tek bir uzak yazmayla olur. Bu çalışma 2006 Dijkstra Ödülü'nü aldı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Tek bir paylaşılan değişkende dönen kilitler çekirdek sayısı arttıkça önbellek tutarlılık trafiğinde boğulur; MCS bu sorunu

Carl Waldspurger ve William Weihl

Piyango ve stride çizelgeleme · 1994–1995

MIT'de 1994'te OSDI'da sundukları piyango çizelgelemede (*lottery scheduling*) her sürece bilet verilir ve her dilimde rastgele çekilen biletin sahibi çalışır; daha çok bilet, işlemcide daha büyük pay demektir. Waldspurger ile Weihl 1995'te aynı payları rastgelelik olmadan sağlayan stride çizelgelemeyi önerdi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Bu yöntemler orantılı pay (*proportional share*) fikrini basit ve anlaşılır biçimde gerçekledi; Linux'un CFS ve EEVDF çizelgeleyicilerindeki ağırlıklı adil paylaşım aynı soyağacından gelir.

çözen ölçeklenebilir senkronizasyonun temel örneğidir ve Linux'un qspinlock'u bu fikre dayanır.

Spinlock

Ölçeklenebilir senkronizasyon

Önbellek tutarlılığı

Neden önemli: Tek bir paylaşılan değişkende dönen kilitler çekirdek sayısı arttıkça önbellek tutarlılık trafiğinde boğulur; MCS bu sorunu çözen ölçeklenebilir senkronizasyonun temel örneğidir ve Linux'un qspinlock'u bu fikre dayanır.

1991

Orantılı pay **Piyango çizelgeleme**

Stride çizelgeleme

Neden önemli: Bu yöntemler orantılı pay (proportional share) fikrini basit ve anlaşılır biçimde gerçekledi; Linux'un CFS ve EEVDF çizelgeleyicilerindeki ağırlıklı adil paylaşım aynı soyağacından gelir.

1994–1995

Jeff Bonwick

Slab ayırıcı · 1994

Sun Microsystems'te SunOS 5.4 (Solaris) çekirdeği için tasarladığı slab ayırıcıyı 1994'te USENIX Summer konferansında yayımladı. Ayırıcı, sık kullanılan çekirdek nesnelerini önceden hazırlanmış önbelleklerde tutar; nesneler tam boyutludur ve sayfa içinde yan yana paketlenir.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Slab ayırıcı, çekirdek nesneleri için iç parçalanmayı ve ilk değer verme maliyetini azaltır; Linux'un SLAB ve SLUB ayırıcıları bu tasarımdan türemiştir.

Slab ayırıcı **Nesne önbelleği**

Çekirdek belleği

Dawson Engler ve Frans Kaashoek

Exokernel · 1995

MIT'de James O'Toole ile birlikte 1995'te SOSP'ta sundukları exokernel mimarisinde çekirdek yalnız donanım kaynaklarını korumalı biçimde dağıtır ve sahipliği izler. Sanal bellek ve dosya gibi soyutlamaları kullanıcı modunda çalışan kütüphane işletim sistemleri sağlar. Prototipte çekirdeğin adı Aegis, kütüphane işletim sisteminin adı ExOS'tur.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

“Yönetimi korumadan ayırmak” sloganı, çekirdekte ne kadar işlev bulunması gerektiği tartışmasının uç noktasıdır; fikir bugün unikernel'lerde ve DPDK, SPDK gibi çekirdeği atlayan G/Ç kütüphanelerinde yaşar.

Neden önemli: Slab ayırıcı, çekirdek nesneleri için iç parçalanmayı ve ilk değer verme maliyetini azaltır; Linux'un SLAB ve SLUB ayırıcıları bu tasarımdan türemiştir.

1994

Exokernel **Kütüphane işletim sistemi**

Çekirdek mimarisi

Neden önemli: "Yönetimi korumadan ayırmak" sloganı, çekirdekte ne kadar işlev bulunması gerektiği tartışmasının uç noktasıdır; fikir bugün unikernel'lerde ve DPDK, SPDK gibi çekirdeği atlayan G/Ç kütüphanelerinde yaşar.

1995

Ion Stoica ve Hussein Abdel-Wahab

EEVDF · 1995

1995'te önerdikleri "Earliest Eligible Virtual Deadline First" (EEVDF) algoritması orantılı pay çizelgelemeyi iki net ölçüte dayandırır: bir görev ancak payının gerisinde kaldığında uygundur ve uygun görevler arasından sanal son tarihi en erken olan seçilir. Algoritmayı Linux çekirdeğine Peter Zijlstra uyarladı; EEVDF, 6.6 sürümünden itibaren CFS'in yerini aldı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

EEVDF, adaleti korurken gecikmeye duyarlı görevlere daha kısa dilimler verebilir; bugün Linux'ta sıradan görevlerin varsayılan çizelgeleyicisidir.

EEVDF **Orantılı pay**

Linux çizelgeleyicisi

Neden önemli: EEVDF, adaleti korurken gecikmeye duyarlı görevlere daha kısa di-

Ingo Molnár

O(1) çizelgeleyici ve CFS · 2000'ler

Linux çekirdeğine 2.6.0 sürümüyle gelen O(1) çizelgeleyiciyi yazdı. 2007'de 2.6.23 sürümüyle O(1)'in yerini alan tamamen adil çizelgeleyiciyi (*Completely Fair Scheduler*, CFS) geliştirdi. CFS ideal bir çok görevli işlemciye öykünür: her görevin sanal çalışma süresini (vruntime) tutar ve kırmızı-siyah ağaçta en geride kalan görevi çalıştırır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

CFS, Linux'ta on beş yılı aşkın süre sıradan görevlerin varsayılan çizelgeleyicisi oldu ve ağırlıklı adil paylaşımı geniş ölçekte uygulanabilir kıldı.

CFS **vruntime** **Linux çizelgeleyicisi**

Neden önemli: CFS, Linux'ta on beş yılı aşkın süre sıradan görevlerin varsayılan çizelgeleyicisi oldu ve ağırlıklı adil paylaşımı geniş ölçekte uygulanabilir kıldı.

limler verebilir; bugün Linux'ta sıradan görevlerin varsayılan çizelgeleyicisidir.

1995

2002–2007