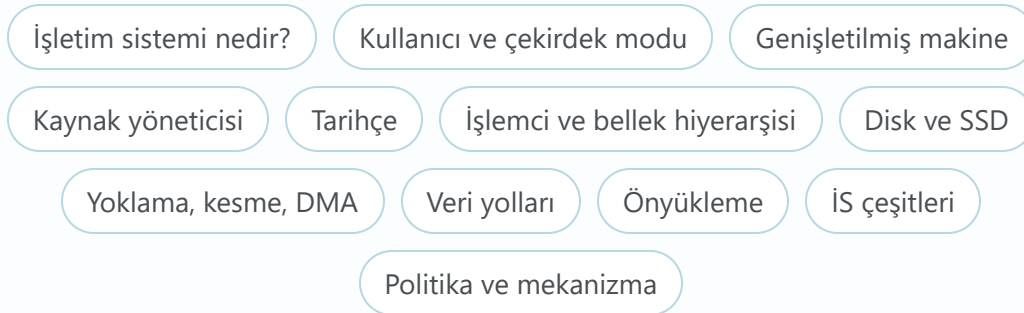


Bölüm 1: Giriş

İşletim Sistemleri



GİRİŞ

Öğrenme Çıktıları

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 kontrollü geçişi (sistem çağırısı, kesme, istisna) açıklamak.

Tarihsel gelişim

Toplu iş, çoklu programlama ve zaman paylaşımı fikirlerinin hangi sorunları çözdüğünü anlatmak.

Donanım temelleri

İşlemci, bellek hiyerarşisi, önbellek, disk/SSD ve veri yollarının İS'yi nasıl etkilediğini yorumlamak.

G/Ç yöntemleri

Yoklama, kesme ve DMA'yı karşılaştırmak; açılış (önyükleme) sırasını izlemek.

Kavramlar

İS çeşitlerini, İS hizmetlerini ve politika–mekanizma ayrımını örneklerle açıklamak.

GİRİŞ

İşletim Sistemi Nedir?

- Modern bir bilgisayar; çok çekirdekli işlemci, GB'larca bellek, SSD, ağ kartı, GPU ve onlarca ayardan oluşur.
- Uygulama geliştiricinin bu donanımın **her ayrıntısını** bilmesi ve doğrudan programlaması imkânsızdır.
- **İşletim sistemi (İS)**: donanımı yöneten, uygulamalara **basit, güvenli ve ortak** bir arayüz sunan yazılım katmanı.
- **Çekirdek (kernel)**: her zaman bellekte duran, ayrıcalıklı kısım; dar anlamda "işletim sistemi" budur.
- Kullanıcılar bilgisayarla **kabuk** (shell, komut satırı) ya da **GUI** (grafik kullanıcı arayüzü, GUI) üzerinden etkileşir.

Masaüstü

Windows 11, macOS, Linux dağıtımları (Ubuntu, Fedora), ChromeOS

Mobil

Android, iOS / iPadOS, HarmonyOS

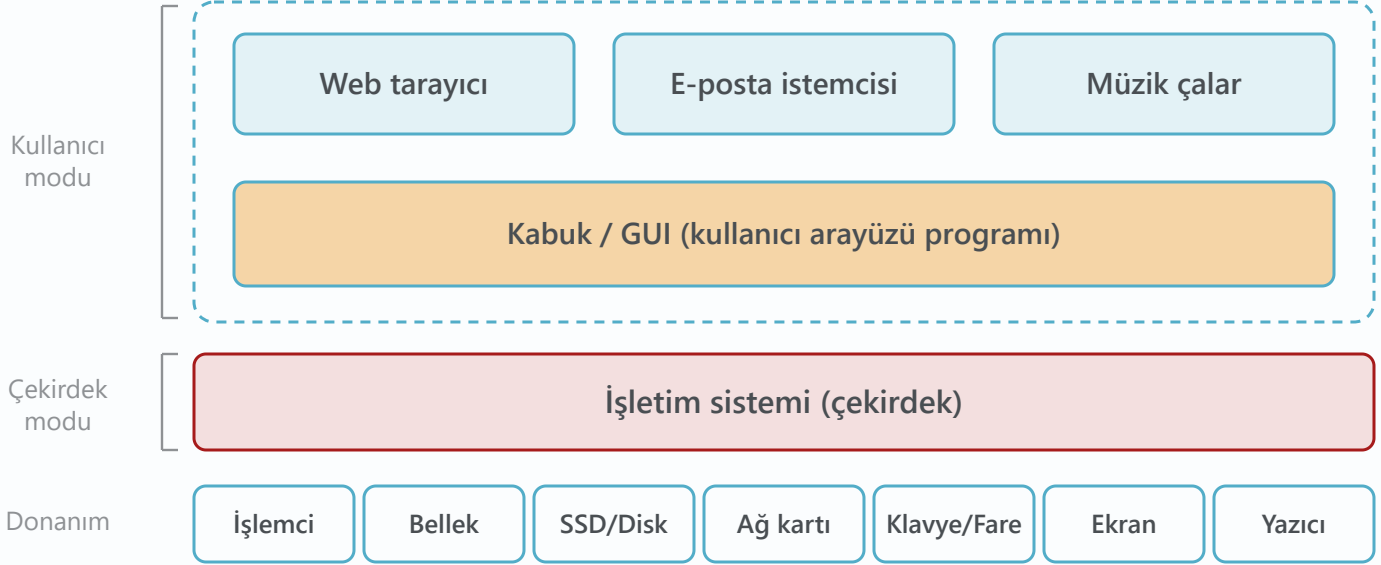
Sunucu ve bulut

Linux, Windows Server, FreeBSD

Gömülü ve IoT

Zephyr, FreeRTOS, Contiki-NG, VxWorks, QNX

Bilgisayar Sisteminin Bileşenleri



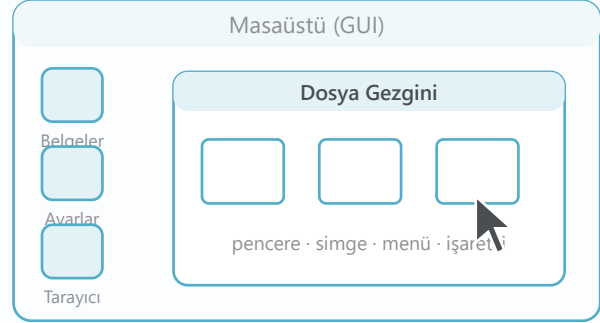
Donanım: işlemci, bellek, depolama ve G/Ç aygıtları. **Yazılım:** çekirdek modunda çalışan işletim sistemi ve kullanıcı modunda çalışan her şey; kabuk/GUI bile sıradan bir kullanıcı programıdır.

GİRİŞ

Kabuk ve GUI

```
Terminal: bash

$ ls ~/notlar
bolum1.md bolum2.md
$ ps aux | grep firefox
sercan 4211 2.1 firefox
$ uname -sr
Linux 6.12.8
$ _
```



- **Kabuk (shell):** komut satırı arayüzü (CLI) · bash, zsh, fish, PowerShell, cmd.exe.
- **GUI (GUI):** pencere, simge, menü ve işaretçi (WIMP) ile görsel etkileşim · Windows Gezgini, macOS Finder, GNOME, KDE Plasma.
- İki de **çekirdeğin parçası değildir**: kullanıcı modunda çalışan programlardır ve çekirdek hizmetlerini **sistem çağrılarıyla** isterler. Bu yüzden değiştirilebilir, kişiselleştirilebilir; aynı anda birden fazlası çalışabilir.

Kabuk mu, GUI mi?

Kabuk

- + Betiklerle (script) **otomasyon**: aynı işi binlerce kez tekrarlamak kolay.
- + Boru hattı: `ps aux | grep ssh | wc -l` gibi küçük araçları birleştirme.
- + Az kaynak; SSH ile uzaktan sunucu yönetimi.
- + Belirli görevleri hızlı ve kesin biçimde yapar.

GUI

- + **Sezgisel** ve keşfedilebilir: komut ezberlemek gerekmez.
- + Görsel içerik (fotoğraf, harita, tasarım) için doğal.
- + Geniş bir kullanıcı kitlesine ulaşır; erişilebilirlik araçları.
- + Dokunmatik ve sesli arayüzler de bu ailededir.

İkisi birbirinin rakibi değil, **tamamlayıcıdır**: günlük işler için GUI, tekrarlı ve uzaktan işler için kabuk. Her ikisi de aynı çekirdek hizmetlerini kullanır; çekirdek, isteğin hangi arayüzden geldiğini bilmez.

Aygıt Yöneticisi

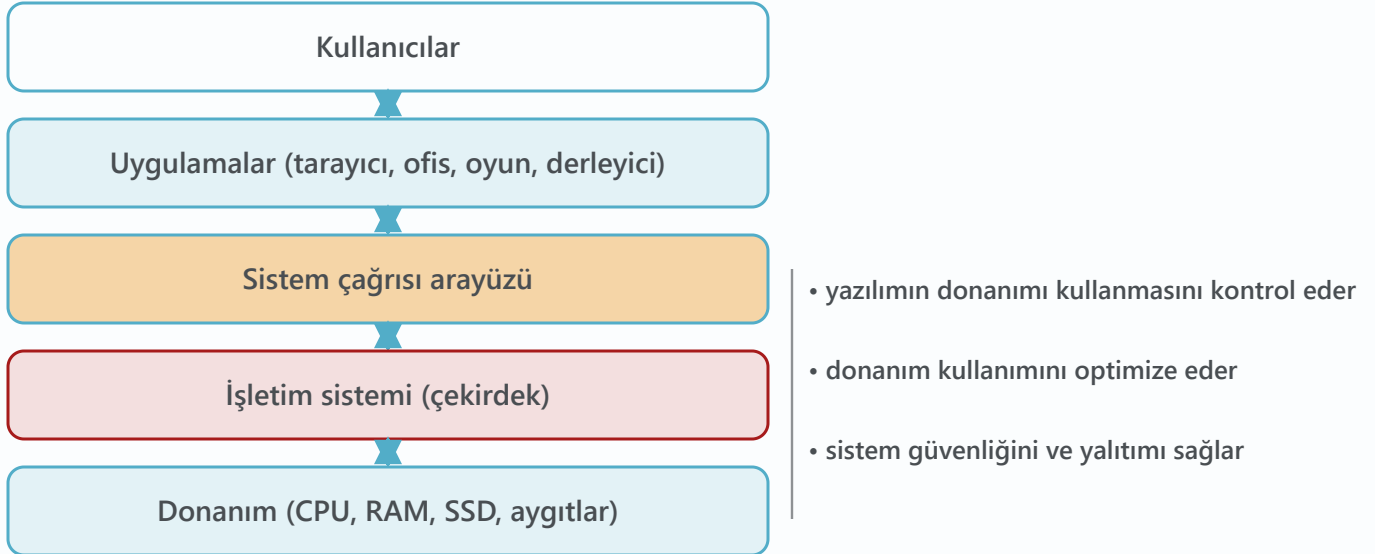
- Windows'ta bilgisayara bağlı donanım aygıtlarını ve **sürücülerini** listeleyen yönetim aracıdır (`devmgmt.msc` , bir MMC eklentisi).
- Aygıtın türünü, üreticisini, sürücü sürümünü ve kullandığı kaynakları (kesme, bellek aralığı) gösterir.
- Sürücüyü güncelleme / geri alma, aygıtı devre dışı bırakma; sorunlu aygıtlar uyarı simgesiyle işaretlenir.
- Asıl işi **çekirdekteki** Tak-Çalıştır (PnP) yöneticisi ve aygıt sürücülerini yapar; araç onların bir *görünümüdür*.
- Linux karşılıkları: `lspci` , `lsusb` , `lsblk` , `lsmod` , `/sys` ve `udev` .

☐ BILGISAYAR-01

- ☐ Disk sürücülerini
 - ☐ NVMe SSD 1 TB
- ☐ Ekran bağdaştırıcıları
 - ☐ GPU (ayrık)
- ☐ Ağ bağdaştırıcıları
 - ☐ Wi-Fi 7
 - ☐ Ethernet 2.5 GbE
- ☐ USB denetleyicileri
 - ☐ USB kök hub (USB 3.2)
- ☐ Diğer aygıtlar
 - ☒ **Bilinmeyen aygıt**

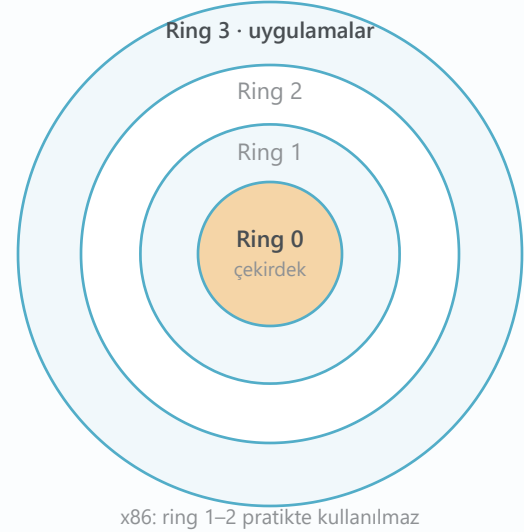
İşletim Sisteminin Konumu

- İS, **donanım** (fiziksel bileşenler: CPU, RAM, disk) ile **yazılım** (bilgisayarın yapacağı işleri tarif eden kod) arasındaki arayüzdür.
- Uygulamalar donanıma doğrudan değil, İS'nin sunduğu **sistem çağrıları** üzerinden erişir.

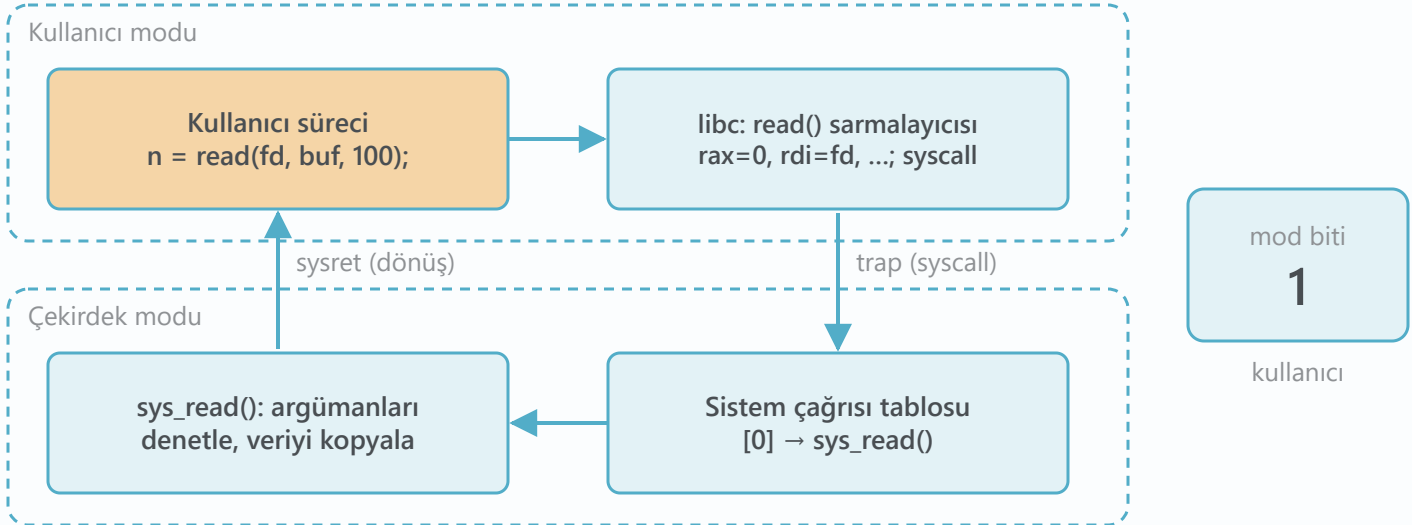


Çekirdek Modu ve Kullanıcı Modu

- İşlemcide bir **mod biti** / ayrıcalık seviyesi vardır: x86 ring 0–3, ARMv8 EL0–EL3, RISC-V U/S/M.
- Çekirdek modu:** tüm komutlar ve tüm bellek kullanılabilir; G/Ç, kesmeleri kapatma, sayfa tablosunu değiştirme gibi *ayrıcalıklı* komutlar serbesttir.
- Kullanıcı modu:** ayrıcalıklı komutlar yasaktır; denenirse işlemci bir **istisna (trap)** üretir ve çekirdek süreci sonlandırabilir.
- Kabuk ve GUI, kullanıcı modu yazılımının en alt seviyesidir.
- Kullanıcıdan çekirdeğe yalnızca **kontrollü kapılardan** geçilir: sistem çağırısı, kesme, istisna.

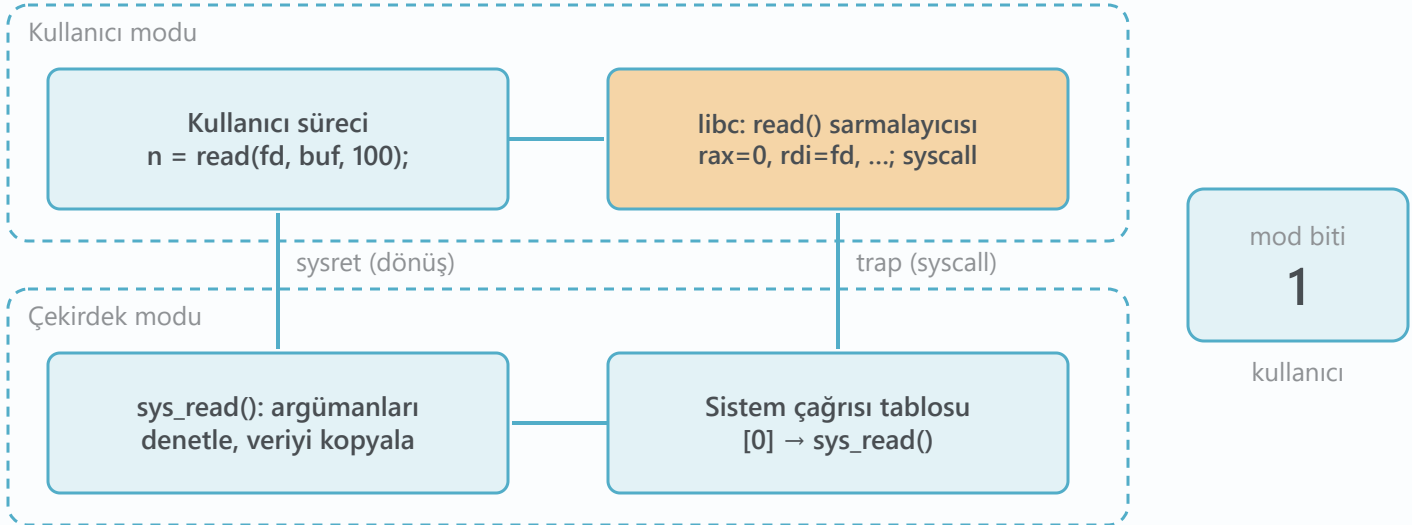


Mod Geçişi Adım Adım: `read()` Sistem Çağrısı



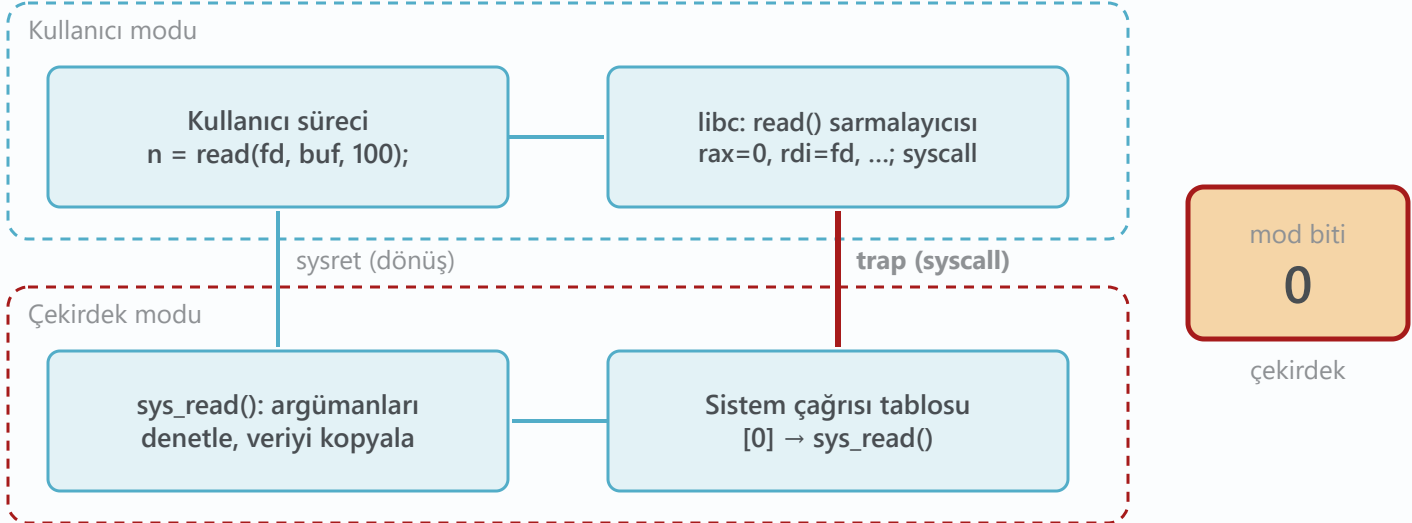
Süreç **kullanıcı modunda** çalışıyor (mod biti = 1). Bir dosyadan okumak istiyor; diske kendisi erişemez.

Mod Geçişi Adım Adım: `read()` Sistem Çağrısı



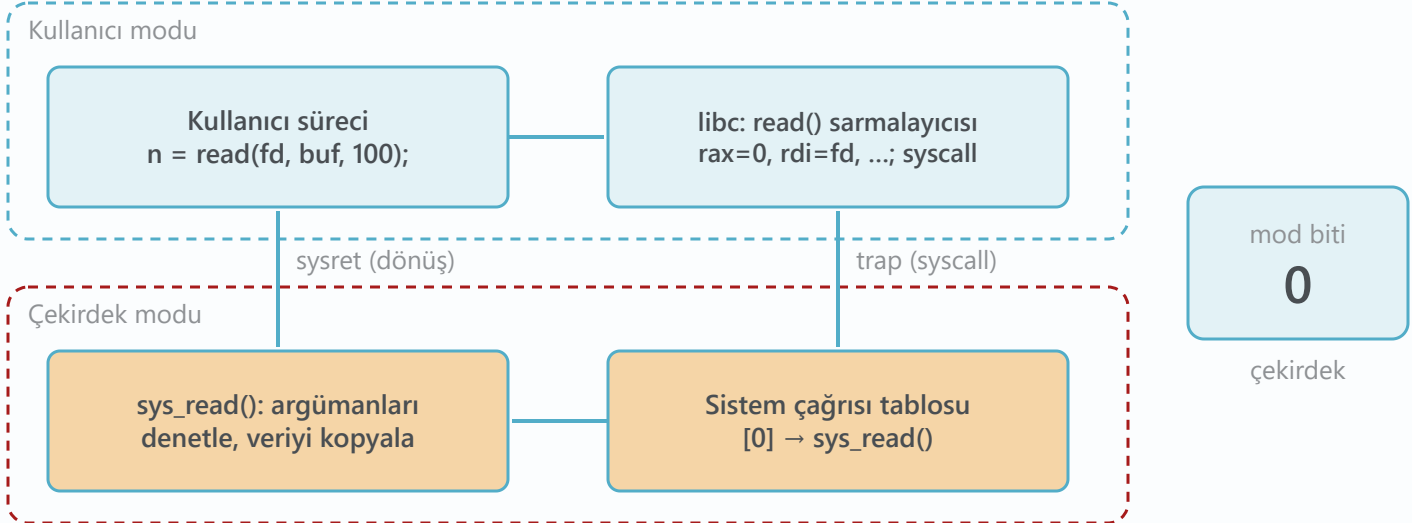
C kütüphanesindeki `read()` sarmalayıcısı çağrı numarasını (x86-64 Linux: `read = 0`, `rax`) ve argümanları yazmaçlara koyar, `syscall` komutunu çalıştırır.

Mod Geçişi Adım Adım: `read()` Sistem Çağrısı



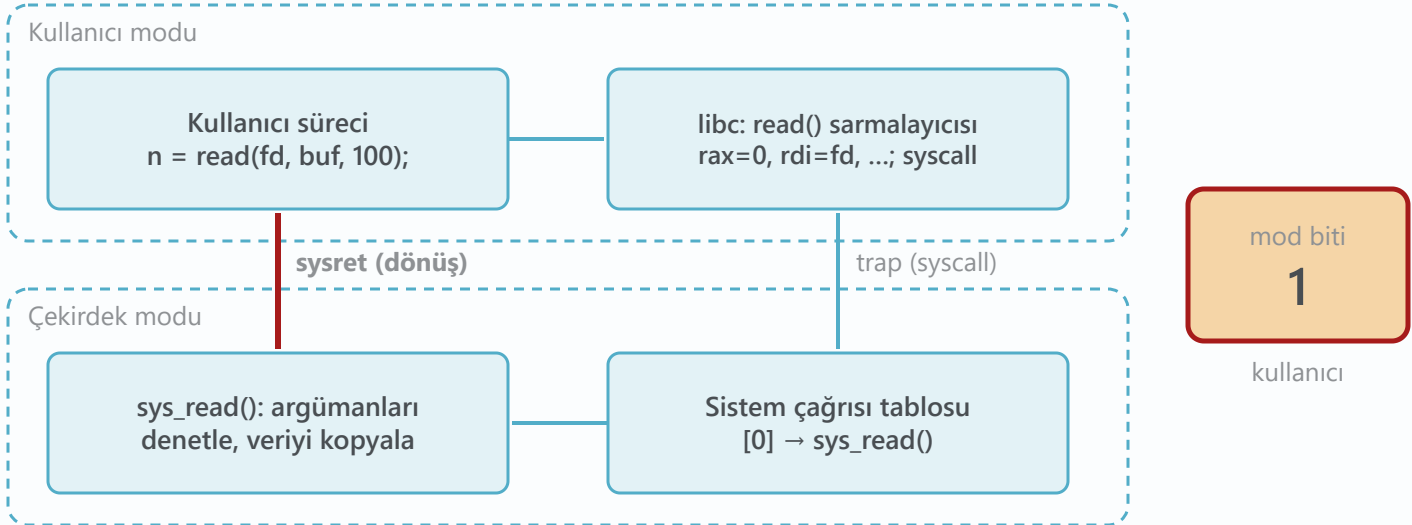
Trap: donanım mod bitini **0** yapar, dönüş adresini kaydeder ve çekirdeğin **sabit** giriş noktasına atlar. Süreç nereye atlanacağını seçemez.

Mod Geçişi Adım Adım: `read()` Sistem Çağrısı



Çekirdek, numarayla **sistem çağrısı tablosundan** `sys_read()` 'i bulur; argümanları doğrular (buf gerçekten sürecin mi?) ve işi yapar.

Mod Geçişi Adım Adım: `read()` Sistem Çağrısı



Sonuç (okunan bayt sayısı) `rax` 'a yazılır; `sysret` mod bitini **1** yapar ve süreç `read()` 'ten sonraki komuttan devam eder.

Fiziksel Aygıtlar, Mikro Mimari, Makine Dili



Fiziksel aygıtlar

Yongalar, bellek modülleri, SSD, G/Ç aygıtları.

Mikro mimari

İşlemcinin iç tasarımı: veri yolu, yazmaçlar, ALU, önbellekler, denetim birimi.

Makine dili (ISA)

Donanımın doğrudan yürüttüğü ikili komutlar: x86-64, ARM64, RISC-V.

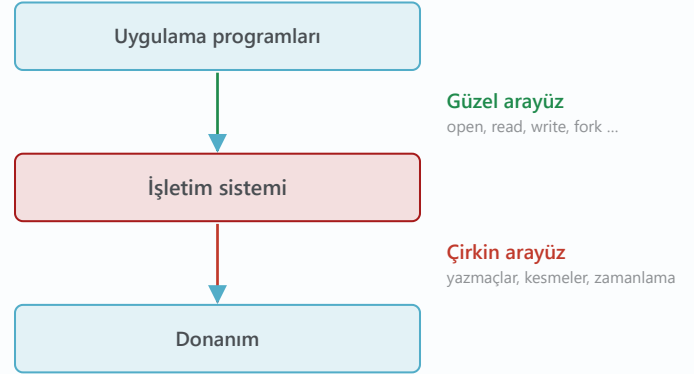
İS'nin işi

Sürücüler, üst katmanın isteklerini aygıtın anlayacağı düşük düzeyli komutlara çevirir.

Programı makine diline *derleyici* çevirir; işletim sistemi bunu yapmaz. İşletim sistemi, derlenmiş programı yükler, çalıştırır ve donanıma erişimine aracılık eder.

Geniřletilmiř Makine Olarak İS

- Donanımın üstüne inşa edilmiř bir yazılımdır; donanımı **kullanmayı kolaylařtırır**.
- Donanımın yeteneklerini kullanılabilir hâle getirir, karmařık ayrıntılarını **gizler**.
- Uygulamaların donanıma **doğrudan** erişmesini engeller: hem kolaylık hem koruma için.
- Örnek: SSD'ye blok yazmak için NVMe kuyrukları, LBA, hata kodları... Uygulama ise yalnızca `write(fd, buf, n)` der.
- Tanenbaum'un ifadesiyle: İS, **çirkin donanımı güzel soyutlamalara** dönüřtürür.



Soyutlamalar ve Roller

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

Her soyutlama, donanımın sınırlı ve paylaşılan kaynağını her programa "kendine ait" gibi gösterir.

Donanım tasarımcısı

ISA'yı, aygıtları ve arayüzlerini tasarlar.

Çekirdek geliştirici

Soyutlamaları donanım üzerinde gerçekleştirir.

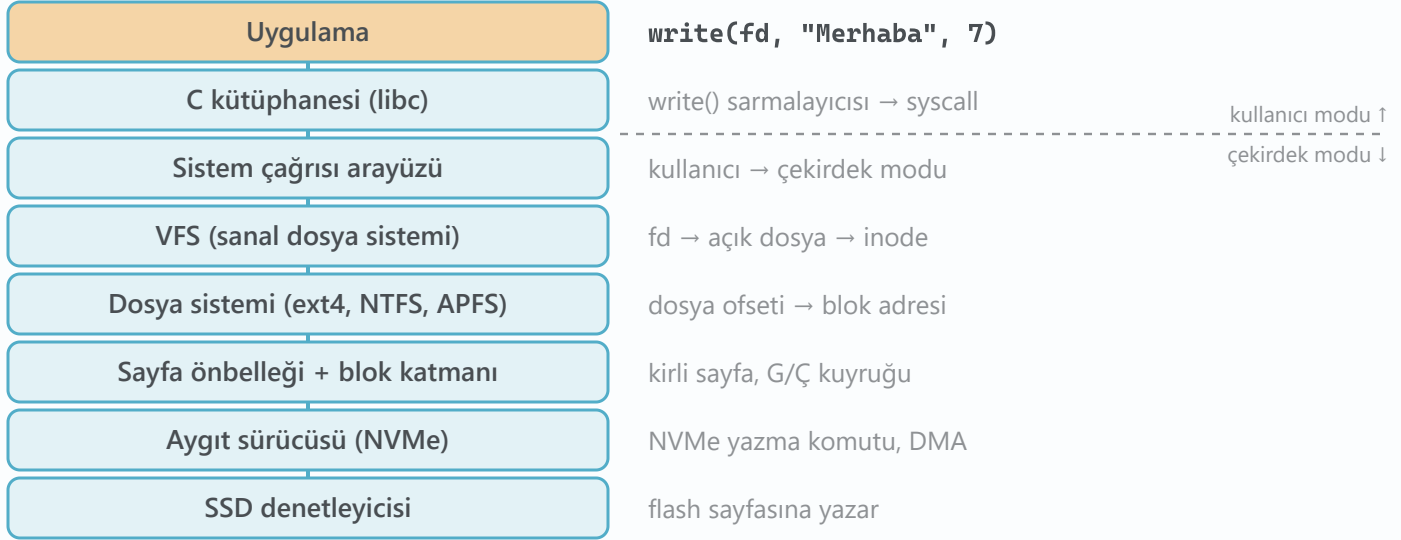
Uygulama geliştirici

Soyutlamaları sistem çağrıları ve API'lerle kullanır.

Son kullanıcı

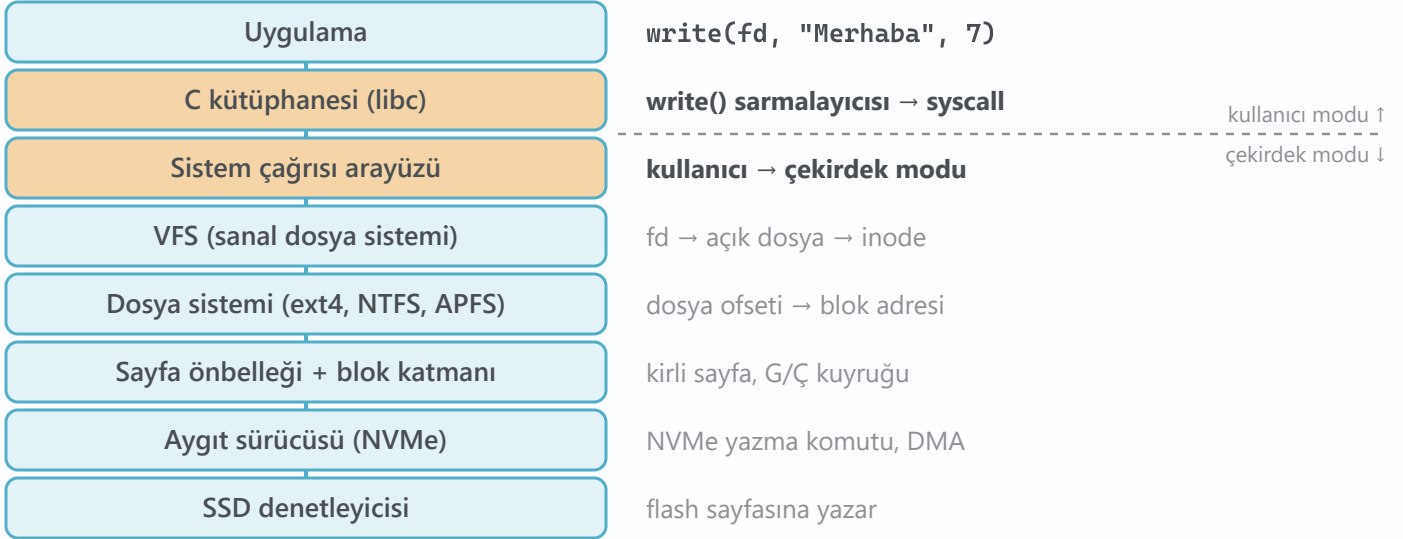
Yalnızca uygulamaları ve arayüzü görür.

Bir `write()` Çağrısının Yolculuğu



Uygulama yalnızca **dosya tanımlayıcısı** (fd), bir tampon ve uzunluk verir; diskin türünü bile bilmez.

Bir `write()` Çağrısının Yolculuğu



libc sarmalayıcısı argümanları yazmaçlara koyar ve **syscall** ile çekirdeğe geçer.

Bir `write()` Çağrısının Yolculuğu



VFS: fd, sürecin açık dosya tablosundan bir dosya nesnesine ve **inode**'a çözülür. Tüm dosya sistemleri için ortak katman.

Bir `write()` Çağrısının Yolculuğu



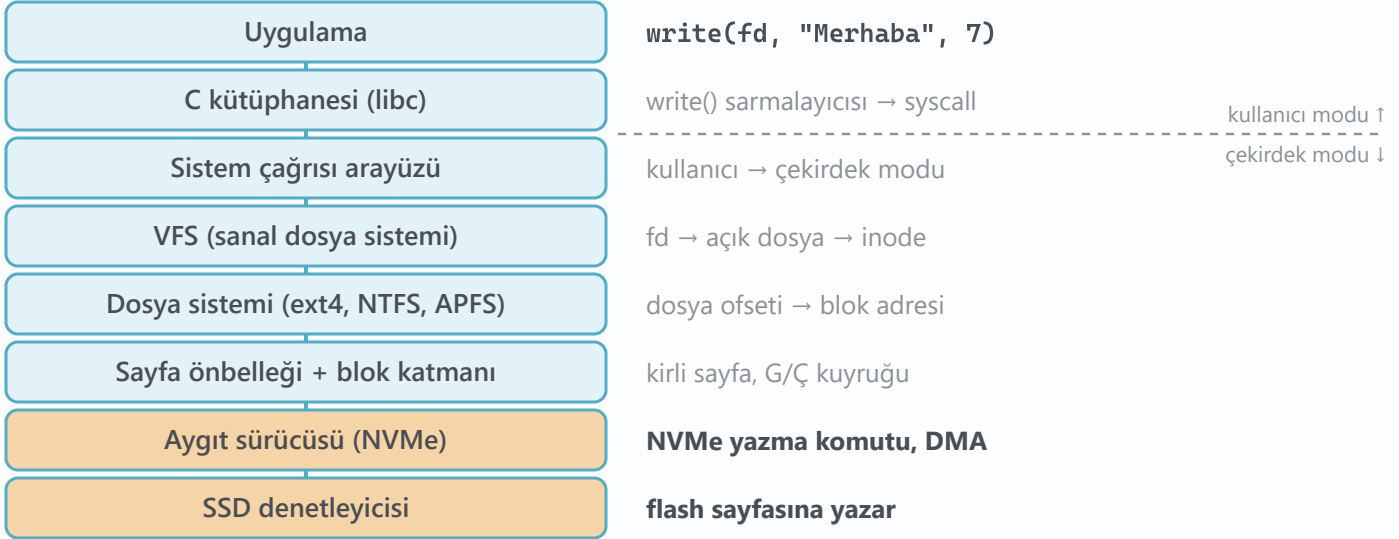
Dosya sistemi, dosyadaki konumu (offset) diskteki **blok adresine** çevirir; gerekirse yeni blok ayırır.

Bir `write()` Çağrısının Yolculuğu



Veri genelde önce **sayfa önbelleğine** yazılır ve `write()` hemen döner; diske yazma sonra yapılır (kalıcılık için `fsync()`).

Bir `write()` Çağrısının Yolculuğu



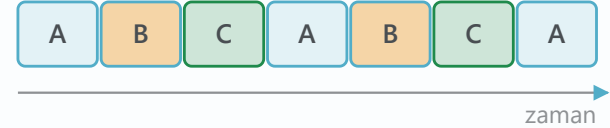
Sürücü bir **NVMe komutu** hazırlar; SSD veriyi DMA ile bellekten alıp flash'a yazar ve bitince kesme üretir.

Kaynak Yöneticisi Olarak İS

- Birden çok programın **aynı anda** çalışmasına izin verir; CPU, bellek, G/Ç aygıtları gibi kaynakları **dağıtır ve korur**.
- Hedef: kaynakları **verimli** kullanmak ve programlar arasında **adil** paylaşmak.
- Yukarıdan aşağı** bakış: uygulamalara soyutlamalar sağlar. **Aşağıdan yukarı** bakış: karmaşık sistemin parçalarını yönetir.
- Kaynaklar iki biçimde paylaşılır: **zamanda** (sırayla) ve **alanda** (parçalara bölerek).

Sorular: Birçok program aynı anda yazdırmak isterse? Her sürecin kaynak kullanımı nasıl ölçülür? Kaynaklar çoğaldığında adalet ve verimlilik nasıl sağlanır?

Zamanda çoklama: tek CPU

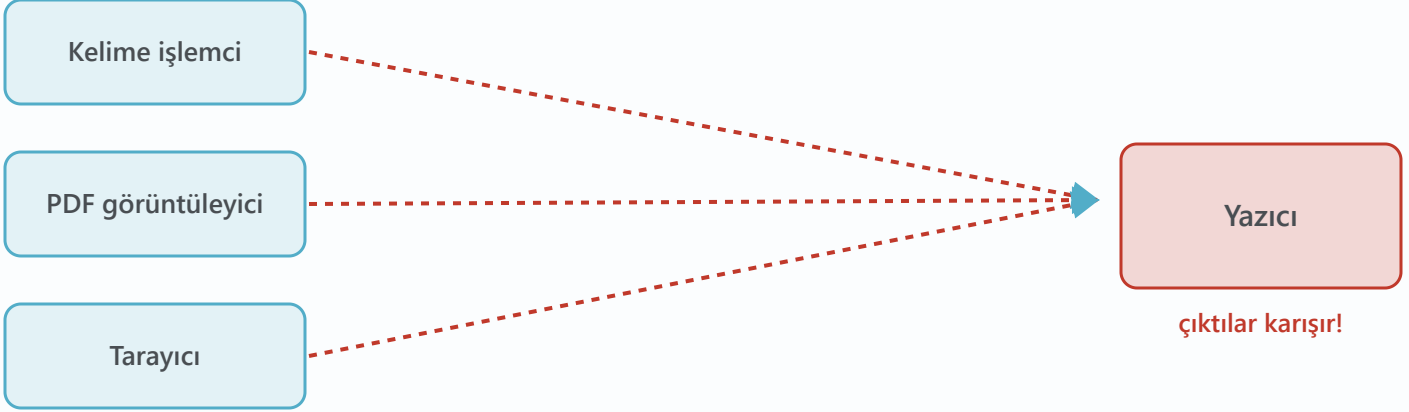


Alanda çoklama: bellek



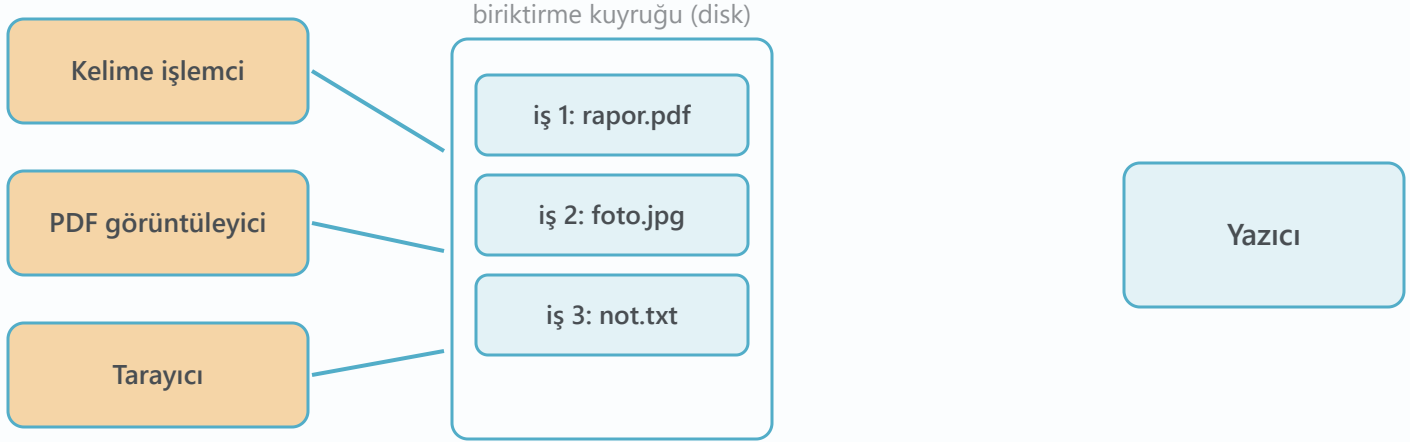
Yazıcı → zamanda · disk blokları → alanda

Birçok Program Aynı Anda Yazdırmak İsterse



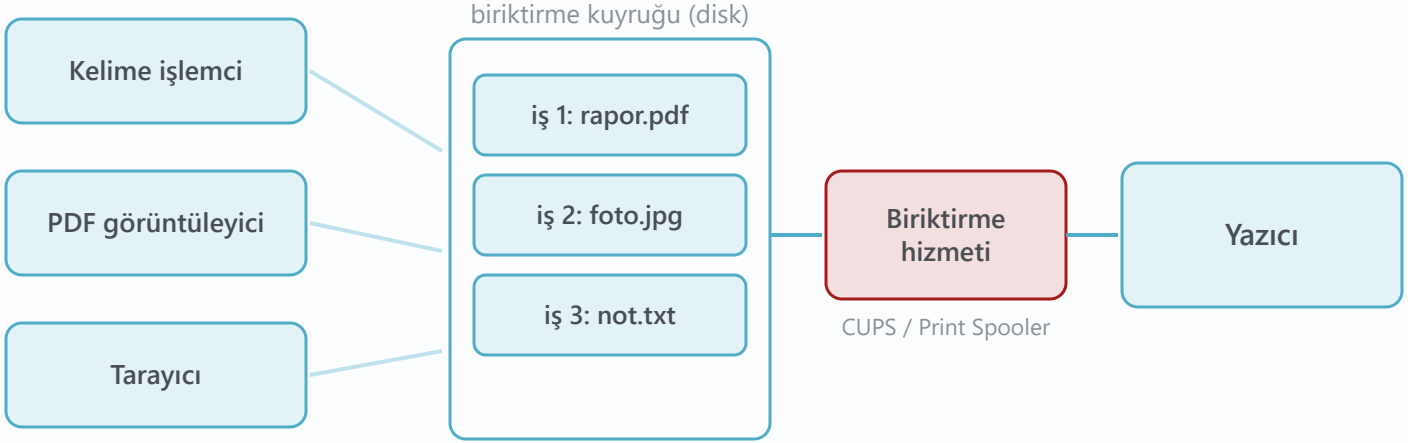
Üç program neredeyse aynı anda yazdırmak istiyor; yazıcı tek. Hepsi yazıcıya **doğrudan** yazabilseydi sayfalar birbirine karışırdı.

Birçok Program Aynı Anda Yazdırmak İsterse



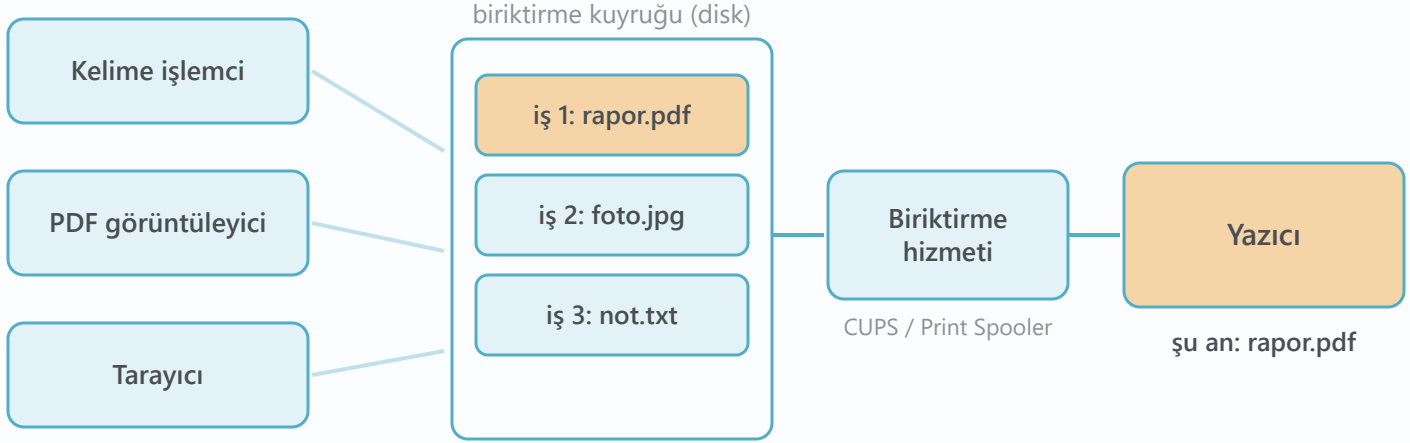
Programlar işlerini yazıcıya değil, İS'nin diskteki **biriktirme (spool) kuyruğuna** yazar ve hemen kendi işlerine döner.

Birçok Program Aynı Anda Yazdırmak İsterse



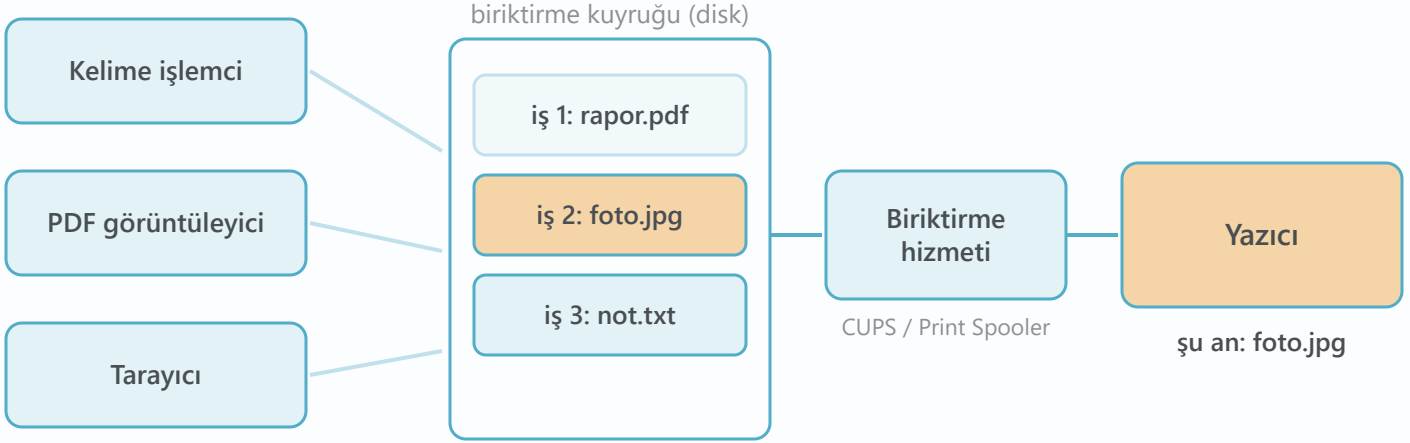
Yazıcıyı kullanan **tek süreç** biriktirme hizmetidir (Linux: CUPS, Windows: Print Spooler); karışıklık bu tekellikle önlenir.

Birçok Program Aynı Anda Yazdırmak İsterse



Hizmet işleri genellikle **geliş sırasıyla** (FCFS) gönderir; yönetici öncelik ya da duraklatma uygulayabilir. Şimdi iş 1 basılıyor.

Birçok Program Aynı Anda Yazdırmak İsterse

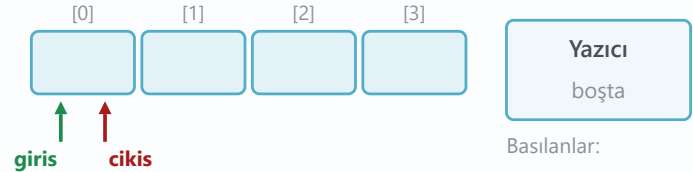


İş 1 bitti, sıradaki iş 2. Yavaş yazıcı beklenirken programlar ve CPU başka işler yapabildi: **G/Ç ile hesaplama örtüşür.**

Kod İzleme: Biriktirme Kuyruğu

```
► char *kuyruk[4];
► int giris = 0, cikis = 0, adet = 0;
3 void ekle(char *dosya) { /* programlar */
4   kuyruk[giris] = dosya;
5   giris = (giris + 1) % 4;
6   adet++;
7 }
8 void biriktirme_hizmeti(void) {
9   while (adet > 0) {
10    yazici_gonder(kuyruk[cikis]);
11    cikis = (cikis + 1) % 4;
12    adet--;
13  }
14 }
```

Biriktirme kuyruğu (diskte)



adet = 0

Kuyruk 4 yuvalı ve **dairesel**: **giris** sıradaki boş yuvayı, **cikis** basılacak ilk işi gösterir. Kuyruk boş: adet = 0.

Adım 1 / 11

DEĞİŞKENLER

giris = 0

cikis = 0

adet = 0

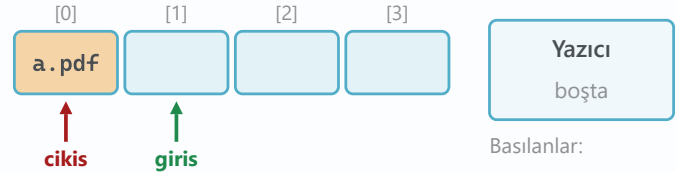
ÇAĞRI YİĞİNİ

boş

Kod İzleme: Biriktirme Kuyruğu

```
1 char *kuyruk[4];
2 int giris = 0, cikis = 0, adet = 0;
3 void ekle(char *dosya) { /* programlar */
4     ▶ kuyruk[giris] = dosya;
5     ▶ giris = (giris + 1) % 4;
6     ▶ adet++;
7 }
8 void biriktirme_hizmeti(void) {
9     while (adet > 0) {
10         yazici_gonder(kuyruk[cikis]);
11         cikis = (cikis + 1) % 4;
12         adet--;
13     }
14 }
```

Biriktirme kuyruğu (diskte)



adet = 1

Program 1 `ekle("a.pdf")` çağırır: dosya [0] yuvasına yazılır, giris = 1, adet = 1. Program yazıcıyı beklemeden işine döner.

Adım 2 / 11

DEĞİŞKENLER

giris = 1

cikis = 0

adet = 1

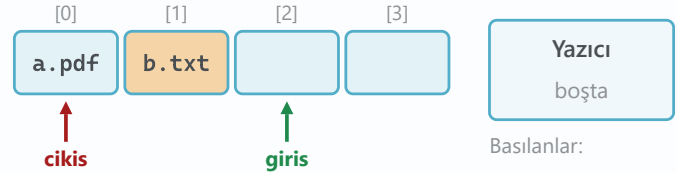
ÇAĞRI YİĞİNİ

ekle("a.pdf")

Kod İzleme: Biriktirme Kuyruğu

```
1 char *kuyruk[4];
2 int giris = 0, cikis = 0, adet = 0;
3 void ekle(char *dosya) { /* programlar */
4     ▶ kuyruk[giris] = dosya;
5     ▶ giris = (giris + 1) % 4;
6     ▶ adet++;
7 }
8 void biriktirme_hizmeti(void) {
9     while (adet > 0) {
10         yazici_gonder(kuyruk[cikis]);
11         cikis = (cikis + 1) % 4;
12         adet--;
13     }
14 }
```

Biriktirme kuyruğu (diskte)



adet = 2

ekle("b.txt") → [1], giris = 2, adet = 2.

Adım 3 / 11

DEĞİŞKENLER

giris = 2

cikis = 0

adet = 2

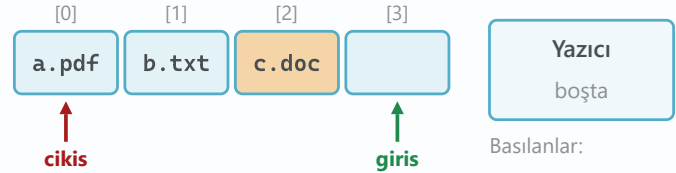
ÇAĞRI YIĞINI

ekle("b.txt")

Kod İzleme: Biriktirme Kuyruğu

```
1 char *kuyruk[4];
2 int giris = 0, cikis = 0, adet = 0;
3 void ekle(char *dosya) { /* programlar */
4     ▶ kuyruk[giris] = dosya;
5     ▶ giris = (giris + 1) % 4;
6     ▶ adet++;
7 }
8 void biriktirme_hizmeti(void) {
9     while (adet > 0) {
10         yazici_gonder(kuyruk[cikis]);
11         cikis = (cikis + 1) % 4;
12         adet--;
13     }
14 }
```

Biriktirme kuyruğu (diskte)



adet = 3

ekle("c.doc") → [2], giris = 3, adet = 3.

Adım 4 / 11

DEĞİŞKENLER

giris = 3

cikis = 0

adet = 3

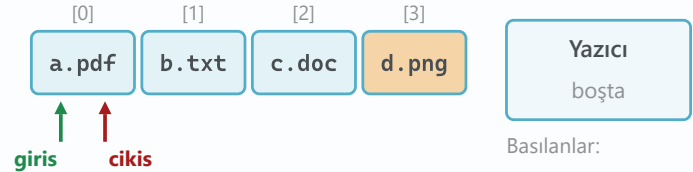
ÇAĞRI YIĞINI

ekle("c.doc")

Kod İzleme: Biriktirme Kuyruğu

```
1 char *kuyruk[4];
2 int giris = 0, cikis = 0, adet = 0;
3 void ekle(char *dosya) { /* programlar */
4     ▶ kuyruk[giris] = dosya;
5     ▶ giris = (giris + 1) % 4;
6     ▶ adet++;
7 }
8 void biriktirme_hizmeti(void) {
9     while (adet > 0) {
10         yazici_gonder(kuyruk[cikis]);
11         cikis = (cikis + 1) % 4;
12         adet--;
13     }
14 }
```

Biriktirme kuyruğu (diskte)



adet = 4

ekle("d.png") → [3]; giris = (3 + 1) %% 4 = 0: dizinin başına **sarar**. Kuyruk dolu: adet = 4, giris = cikis.

Adım 5 / 11

DEĞİŞKENLER

giris = 0

cikis = 0

adet = 4

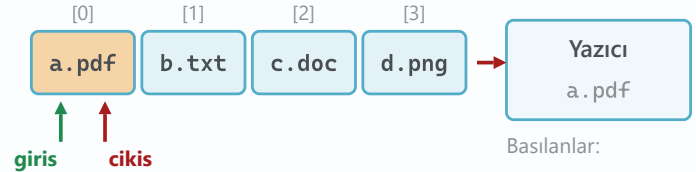
ÇAĞRI YIĞINI

ekle("d.png")

Kod İzleme: Biriktirme Kuyruğu

```
1 char *kuyruk[4];
2 int giris = 0, cikis = 0, adet = 0;
3 void ekle(char *dosya) { /* programlar */
4     kuyruk[giris] = dosya;
5     giris = (giris + 1) % 4;
6     adet++;
7 }
8 void biriktirme_hizmeti(void) {
9     while (adet > 0) {
10        yazici_gonder(kuyruk[cikis]);
11        cikis = (cikis + 1) % 4;
12        adet--;
13    }
14 }
```

Biriktirme kuyruğu (diskte)



adet = 4

Biriktirme hizmeti çalışır: $adet = 4 > 0$. Yazıcıya **en eski** iş, $kuyruk[cikis] = a.pdf$ gönderilir (FCFS).

Adım 6 / 11

DEĞİŞKENLER

giris = 0

cikis = 0

adet = 4

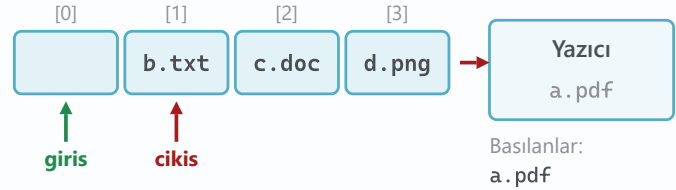
ÇAĞRI YIĞINI

biriktirme_hizmeti()

Kod İzleme: Biriktirme Kuyruğu

```
1 char *kuyruk[4];
2 int giris = 0, cikis = 0, adet = 0;
3 void ekle(char *dosya) { /* programlar */
4     kuyruk[giris] = dosya;
5     giris = (giris + 1) % 4;
6     adet++;
7 }
8 void biriktirme_hizmeti(void) {
9     while (adet > 0) {
10        yazici_gonder(kuyruk[cikis]);
11        cikis = (cikis + 1) % 4;
12        adet--;
13    }
14 }
```

Biriktirme kuyruğu (diskte)



adet = 3

Yuva boşaltılır: cikis = 1, adet = 3. Yazıcı yavaştır; programlar bu sırada başka iş yapar.

Adım 7 / 11

DEĞİŞKENLER

giris = 0

cikis = 1

adet = 3

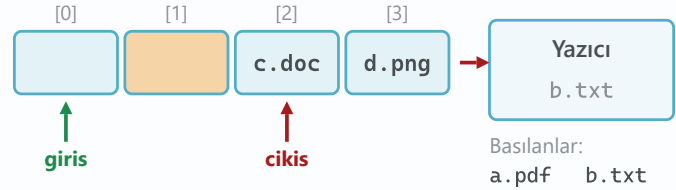
ÇAĞRI YIĞINI

biriktirme_hizmeti()

Kod İzleme: Biriktirme Kuyruğu

```
1 char *kuyruk[4];
2 int giris = 0, cikis = 0, adet = 0;
3 void ekle(char *dosya) { /* programlar */
4     kuyruk[giris] = dosya;
5     giris = (giris + 1) % 4;
6     adet++;
7 }
8 void biriktirme_hizmeti(void) {
9     while (adet > 0) {
10         yazici_gonder(kuyruk[cikis]);
11         cikis = (cikis + 1) % 4;
12         adet--;
13     }
14 }
```

Biriktirme kuyruğu (diskte)



adet = 2

adet = 3 > 0: **b.txt** gönderilir; cikis = 2, adet = 2.

Adım 8 / 11

DEĞİŞKENLER

giris = 0

cikis = 2

adet = 2

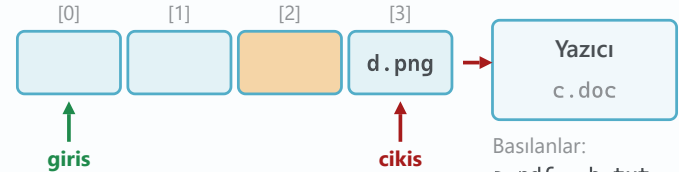
ÇAĞRI YIĞINI

biriktirme_hizmeti()

Kod İzleme: Biriktirme Kuyruğu

```
1 char *kuyruk[4];
2 int giris = 0, cikis = 0, adet = 0;
3 void ekle(char *dosya) { /* programlar */
4     kuyruk[giris] = dosya;
5     giris = (giris + 1) % 4;
6     adet++;
7 }
8 void biriktirme_hizmeti(void) {
9     while (adet > 0) {
10         yazici_gonder(kuyruk[cikis]);
11         cikis = (cikis + 1) % 4;
12         adet--;
13     }
14 }
```

Biriktirme kuyruğu (diskte)



adet = 1

adet = 2 > 0: **c.doc** gönderilir; cikis = 3, adet = 1.

Adım 9 / 11

DEĞİŞKENLER

giris = 0

cikis = 3

adet = 1

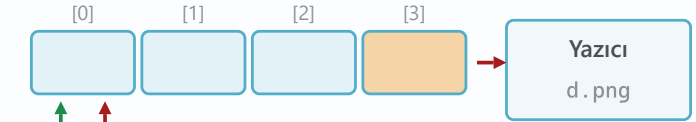
ÇAĞRI YIĞINI

biriktirme_hizmeti()

Kod İzleme: Biriktirme Kuyruğu

```
1 char *kuyruk[4];
2 int giris = 0, cikis = 0, adet = 0;
3 void ekle(char *dosya) { /* programlar */
4     kuyruk[giris] = dosya;
5     giris = (giris + 1) % 4;
6     adet++;
7 }
8 void biriktirme_hizmeti(void) {
9     while (adet > 0) {
10         yazici_gonder(kuyruk[cikis]);
11         cikis = (cikis + 1) % 4;
12         adet--;
13     }
14 }
```

Biriktirme kuyruğu (diskte)



Basılanlar:
a.pdf b.txt
c.doc d.png

adet = 0

adet = 1 > 0: **d.png** gönderilir; cikis = 0, adet = 0.

Adım 10 / 11

DEĞİŞKENLER

giris = 0

cikis = 0

adet = 0

ÇAĞRI YIĞINI

biriktirme_hizmeti()

Kod İzleme: Biriktirme Kuyruğu

```
1 char *kuyruk[4];
2 int giris = 0, cikis = 0, adet = 0;
3 void ekle(char *dosya) { /* programlar */
4     kuyruk[giris] = dosya;
5     giris = (giris + 1) % 4;
6     adet++;
7 }
8 void biriktirme_hizmeti(void) {
9     while (adet > 0) {
10         yazici_gonder(kuyruk[cikis]);
11         cikis = (cikis + 1) % 4;
12         adet--;
13     }
14 }
```

Biriktirme kuyruğu (diskte)



adet = 0

cikis da 0'a sardı; adet = 0 → döngü biter. Yazıcıyı yalnızca hizmet kullandığı için çıktılar **karışmadı**. Eş zamanlı `adet++` / `adet--` yarışa açık olduğundan gerçek kuyruklar kilitlerle korunur.

Basılanlar:

a.pdf b.txt
c.doc d.png

Adım 11 / 11

DEĞİŞKENLER

giris = 0

cikis = 0

adet = 0

ÇAĞRI YİĞİNİ

biriktirme_hizmeti()

Kaynak Kullanımını Ölçmek

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

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

Ölçümün kaynağı çekirdektir: her bağlam değiştirmede, sistem çağrısında ve sayfa hatasında süreç başına sayaçları günceller.

Adalet ve Verimlilik

Kaynakların çoğalması (çok çekirdek, büyük bellek) performans ve tepki süresi kazandırır; ama onları **adil ve verimli** paylaştırmayı zorlaştırır.

Çizelgeleme

Hangi süreç ne zaman, hangi çekirdekte çalışacak? Linux 6.6 (2023) ile CFS yerine **EEVDF** çizelgeleyicisi geldi.

Önceliklendirme

nice değerleri, Windows öncelik sınıfları, gerçek zamanlı sınıflar.

Kotalar ve sınırlar

Linux **cgroups v2** (konteynerler), **ulimit**, disk kotaları, Windows Job nesneleri.

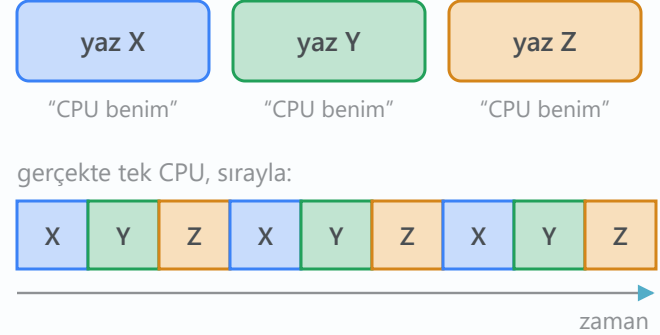
Ödünleşim

Toplam iş hacmi ↔ tepki süresi ↔ adalet: biri iyileşirken diğeri kötüleşebilir.

Doğru yaklaşım **iş yüküne** bağlıdır: sunucuda iş hacmi, masaüstünde tepki süresi, gerçek zamanlı sistemde son tarihlere uymak önceliklidir.

Üç Büyük Fikir: Dört Küçük Deney

```
$ ./yaz X & ./yaz Y & ./yaz Z &  
[1] 3101 [2] 3102 [3] 3103  
X  
Y  
Z  
Z  
X  
Y  
...
```



dilim başına birkaç ms; ne zaman kimin çalışacağına İS karar verir

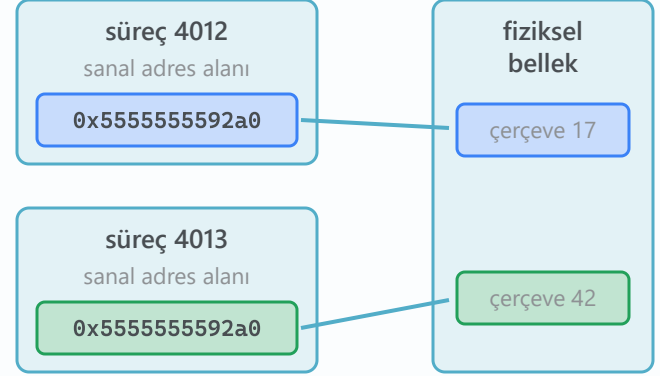
CPU'yu sanallaştırma: tek çekirdekte üç program aynı anda çalışıyor gibi görünür. İS zaman paylaşımıyla her birine kendi CPU'su varmış yanılsaması verir; sıranın kime geleceği **politika** (çizelgeleme), geçişin nasıl yapılacağı **mekanizmadır** (bağlam değiştirme).

Adım 1 / 4

Dersin haritası: sanallaştırma (Bölüm 3–5, 8–9), eş zamanlılık (Bölüm 4, 6–7), kalıcılık (Bölüm 10–11).

Üç Büyük Fikir: Dört Küçük Deney

```
$ setarch -R ./sayac & setarch -R ./sayac &
(4012) p = 0x5555555592a0
(4013) p = 0x5555555592a0
(4012) *p = 1
(4013) *p = 1
(4013) *p = 2
(4012) *p = 2
...
```



aynı sanal adres, farklı fiziksel çerçeve

Belleği sanallaştırma: iki süreç aynı adresi yazdırır ama değerleri birbirinden bağımsız artar. Her süreç kendi adres alanını görür; İS onları farklı fiziksel çerçevelere eşler. Aynı adresi görmek için ASLR kapatılır (`setarch -R`), yoksa adresler her çalıştırmada değişir.

Adım 2 / 4

Dersin haritası: sanallaştırma (Bölüm 3–5, 8–9), eş zamanlılık (Bölüm 4, 6–7), kalıcılık (Bölüm 10–11).

Üç Büyük Fikir: Dört Küçük Deney

```
$ ./artir 1000  
sonuç: 2000  
$ ./artir 1000000  
sonuç: 1318442  
$ ./artir 1000000  
sonuç: 1407715  
(beklenen: 2000000)
```



sayac++ aslında üç komut:



araya diğer iş parçacığı girerse bir artış kaybolur: sonuç 2N'den küçük olur

Eş zamanlılık: iki iş parçacığı ortak sayacı N'er kez artırır. Küçük N'de sonuç doğru görünür; büyük N'de 2N'den küçük ve her çalıştırmada farklıdır, çünkü **sayac++** atomik değildir (örnek çıktı).

Adım 3 / 4

Dersin haritası: sanallaştırma (Bölüm 3–5, 8–9), eş zamanlılık (Bölüm 4, 6–7), kalıcılık (Bölüm 10–11).

Üç Büyük Fikir: Dört Küçük Deney

```
$ ./kaydet
$ cat not.txt
merhaba disk
$ sudo reboot
... (sistem kapanıp yeniden açılır)
$ cat not.txt
merhaba disk
```

program: open, write, close

İS: dosya sistemi

aygıt sürücüsü: G/Ç isteği

SSD / disk: kalıcı

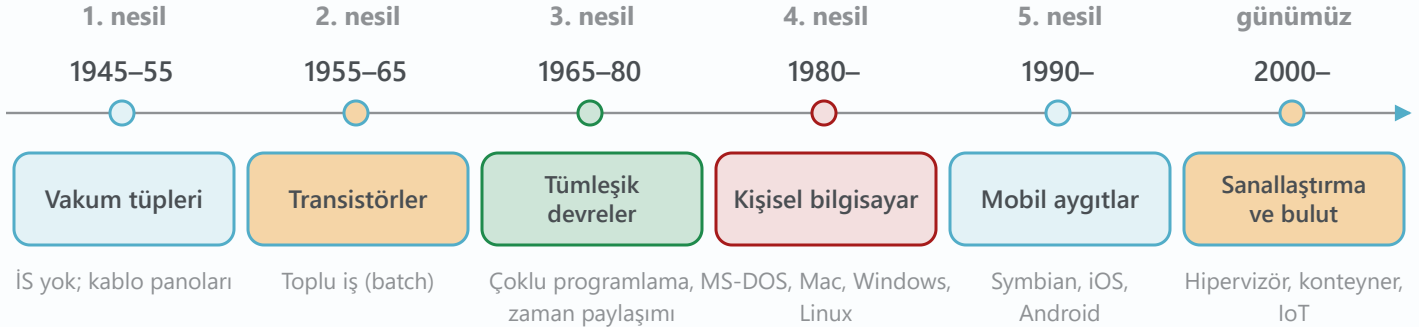
blok seçimi, meta veri, sıralama: hepsi İS'de

Kalıcılık: DRAM uçucudur; veri yeniden başlatmadan sonra da kalsın istiyorsak dosya sistemiyle diske yazarız. Program yalnızca `open`, `write`, `close` çağırır; hangi bloğa yazılacağı, meta veri ve aygıtlarla konuşma İS'nin işidir.

Adım 4 / 4

Dersin haritası: sanallaştırma (Bölüm 3–5, 8–9), eş zamanlılık (Bölüm 4, 6–7), kalıcılık (Bölüm 10–11).

İşletim Sistemlerinin Tarihi



- Nesiller, donanım teknolojisindeki sıçramalarla (tüp → transistör → tümleşik devre → mikroişlemci) birlikte değişir.
- Her nesilde aynı soru yeniden sorulur: **pahalı kaynağı (CPU, bellek, insan zamanı) nasıl daha iyi kullanırız?**
- Son satır Tanenbaum'un nesillerinden biri değildir; bugünkü sistemleri anlamak için eklenmiştir.

1. ve 2. Nesil

1. nesil (1945–55): vakum tüpleri

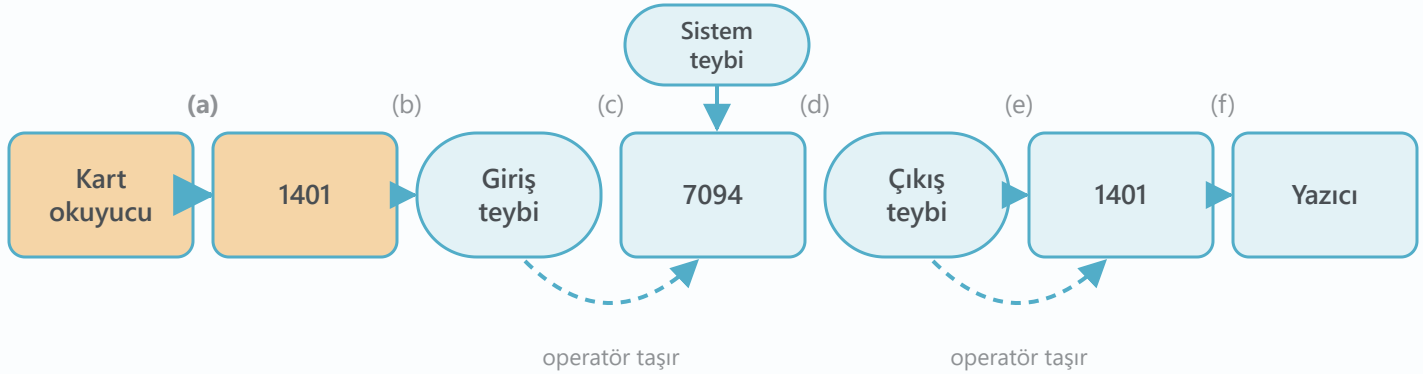
- Oda büyüklüğünde, yavaş, sık arızalanan makineler (ENIAC, 1945).
- Aynı mühendis grubu tasarlar, kurar, programlar, bakımını yapar.
- Makine diliyle ya da **kablo panolarıyla** programlama; 1950'lerin başında **delikli kartlar**.
- Programlama dili ve **işletim sistemi yok**; çoğunlukla sayısal tablolar hesaplanır.

2. nesil (1955–65): transistör ve toplu iş

- Transistörler tüplerin yerini aldı: daha küçük, güvenilir, enerji verimli.
- Ana bilgisayarlar klimalı odalarda; programcı ile makine arasında **operatör**.
- **Toplu iş**: işler teybe toplanır, küçük bir monitör programı onları **sırayla, otomatik** çalıştırır; operatör yalnızca teypleri taşır.
- Bellekte tek iş: G/Ç sırasında CPU boşta; etkileşim yok.

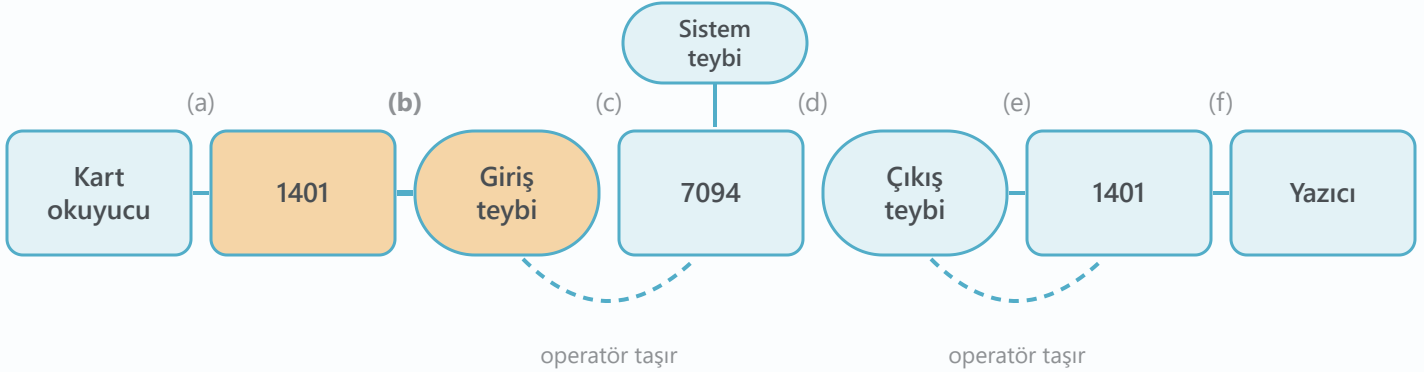
Toplu işin amacı işler arasındaki **el ile geçiş** süresini ortadan kaldırmaktı. Etkileşimli ve gerçek zamanlı kullanıma ise uygun değildi.

Erken Bir Toplu İş Sistemi



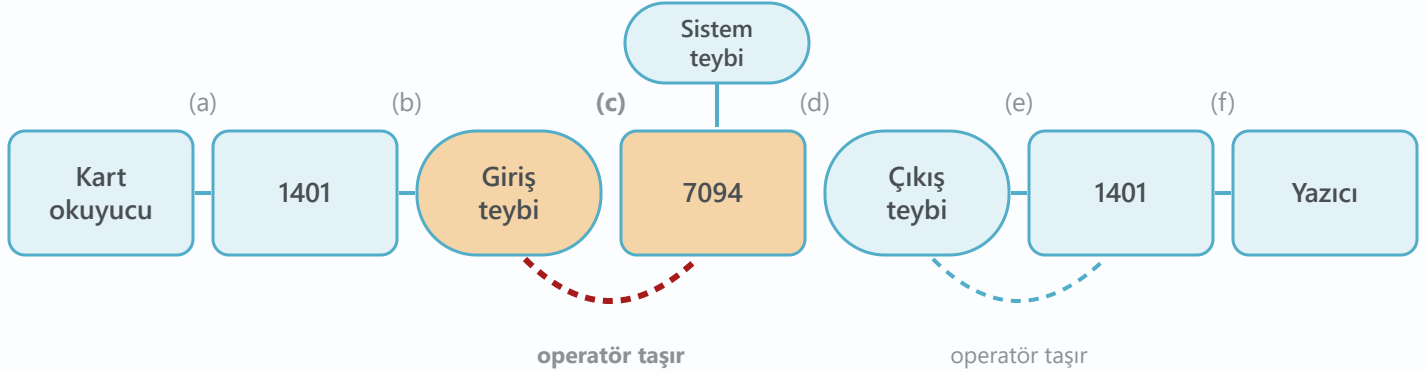
(a) Programcılar delikli kart destelerini 1401'in kart okuyucusuna getirir.

Erken Bir Toplu İş Sistemi



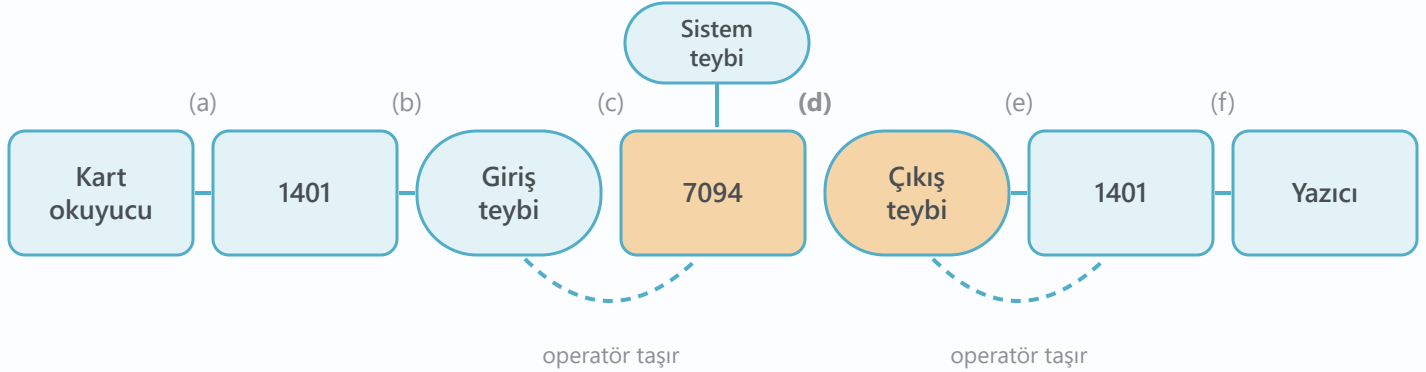
(b) Ucuz 1401, bir grup işin kartlarını okuyup **giriş teybine** yazar (yavaş kart okumayı pahalı makineden ayırır).

Erken Bir Toplu İş Sistemi



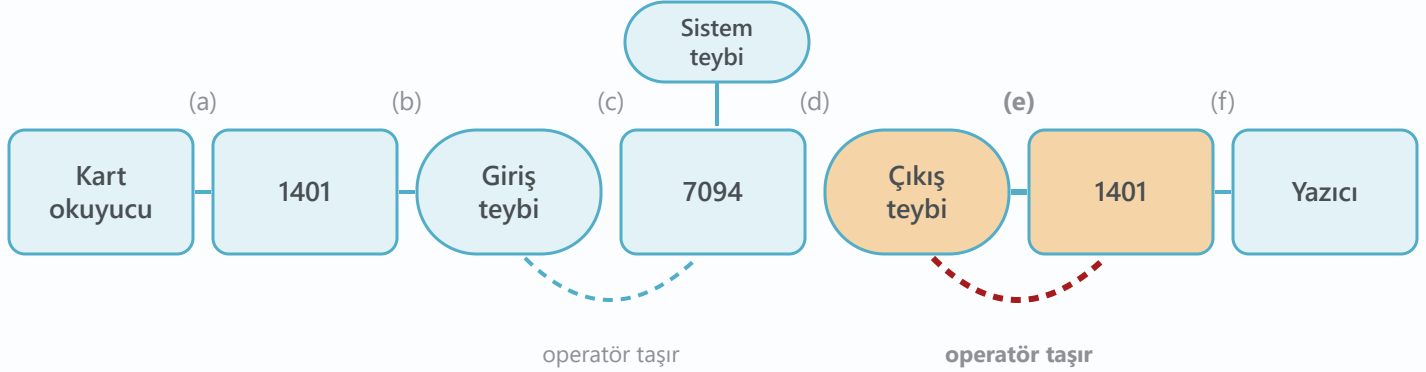
(c) Operatör giriş teybini pahalı **7094**'e taşır.

Erken Bir Toplu İş Sistemi



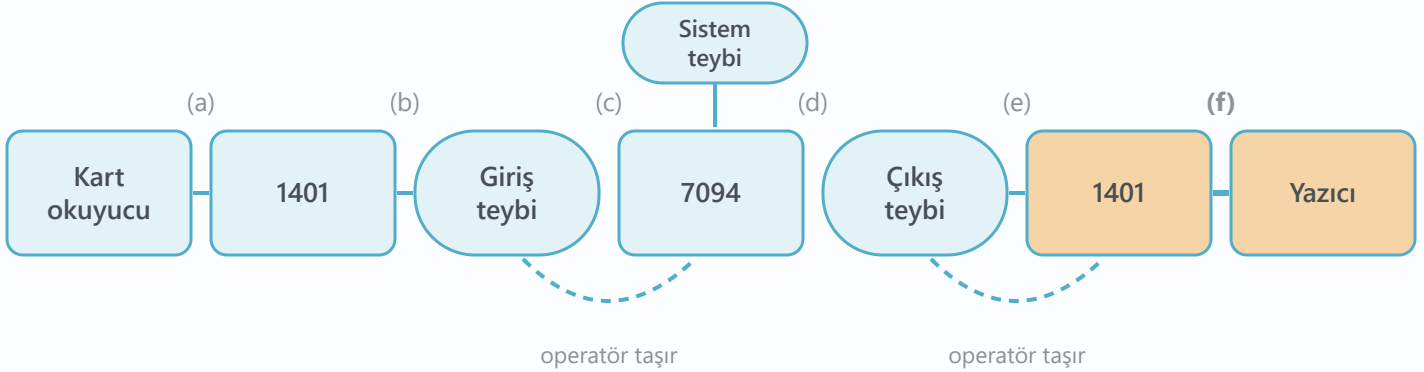
(d) 7094, sistem teybindeki monitör programıyla işleri **sırayla** hesaplar; çıktıları çıkış teybine yazar.

Erken Bir Toplu İş Sistemi



(e) Operatör çıkış teybini yeniden 1401'e taşır.

Erken Bir Toplu İş Sistemi



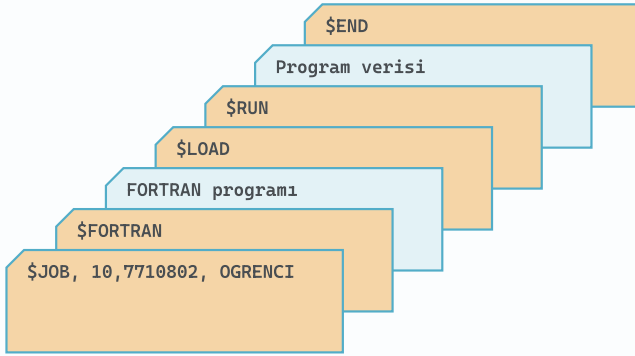
(f) 1401 çıktıları yazıcıdan basar (çevrim dışı yazdırma). Pahalı 7094 hiç kart okumak ya da yazdırmakla uğraşmadı.

IBM 1401, 7094 ve System/360

Makine	Yıl	Özellik
IBM 1401	1959	Ticari veri işleme; karakter tabanlı, görece ucuz. Kart okuma, teybe kopyalama, yazdırma gibi G/Ç işlerinde de kullanıldı. 1960'ların en yaygın bilgisayarlarından.
IBM 7094	1962	Bilimsel hesaplama; 36 bit sözcüklü, 1401'den tamamen farklı bir mimari. IBSYS / 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 aile. İşletim sistemi OS/360 : çoklu programlama, biriktirme; milyonlarca satırlık, hatalarla dolu dev bir proje (Brooks, <i>The Mythical Man-Month</i>).

1401 ve 7094, OS/360 altında çalışmazdı; OS/360, 1964'te duyurulan System/360 ailesi için yazıldı. 7094 de 1401'in güçlü bir sürümü değil, onu tamamlayan ayrı bir makineydi.

Tipik Bir FMS İşi

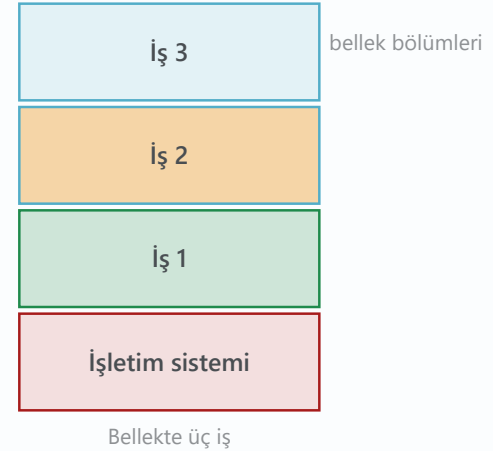


- **FMS = Fortran Monitor System:** IBM 709/7094 için bir toplu iş izleme (monitör) programı.
- '\$' ile başlayan **denetim kartları** monitöre ne yapacağını söyler: işi başlat, derleyiciyi yükle, programı bağla, çalıştır, bitir.
- Bu kartlar bugünkü **kabuk betiklerinin** ve IBM JCL'nin atasıdır.

Buradaki FMS (FORTRAN Monitor System), fabrika otomasyonundaki "Esnek Üretim Sistemi" (Flexible Manufacturing System) ile karıştırılmamalıdır; iki kavram arasında bağlantı yoktur.

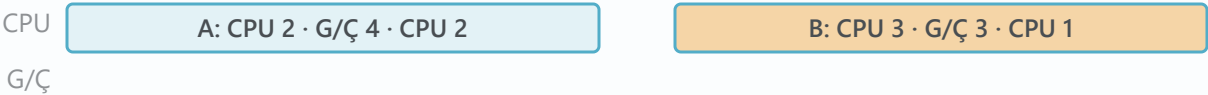
3. Nesil: Tümüleşik Devreler ve Çoklu Programlama

- **Tümüleşik devreler** (1965–80): System/360 gibi uyumlu aileler; daha küçük, ucuz, güvenilir.
- **Çoklu programlama**: bellek bölümlere ayrılır, her bölümde bir iş. Bir iş G/Ç beklerken CPU **başka işe** geçer.
- Tek CPU'da işler **iç içe** ilerler: gerçek paralellik değil, **eş zamanlılıktır** (concurrency).
- İşlerin birbirinin belleğine dokunmaması için **koruma donanımı** gerekir.
- **Biriktirme**: kartlar doğrudan diske; bir iş bitince sıradaki diskten yüklenir.
- **Zaman paylaşımı**: CTSS (MIT, 1961) → MULTICS → **UNIX** (Bell Labs, 1969) → BSD, POSIX, MINIX, Linux (1991).



Çoklu Programlama: CPU Neden Boş Kalmaz?

Tek programlı



Çoklu programlı

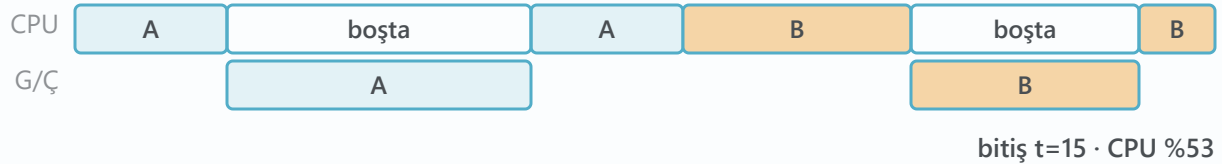
CPU
G/Ç (A)
G/Ç (B)



İki iş (birim zaman): **A** = CPU 2, G/Ç 4, CPU 2 · **B** = CPU 3, G/Ç 3, CPU 1. Toplam CPU ihtiyacı 8 birim.

Çoklu Programlama: CPU Neden Boş Kalmaz?

Tek programlı



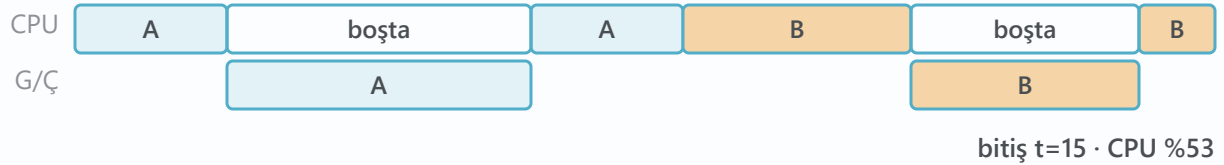
Çoklu programlı



Tek programlı: önce A ($t = 0-8$), sonra B ($t = 8-15$). G/Ç sırasında CPU **boşta**: kullanım $8 / 15 \approx \text{\%53}$.

Çoklu Programlama: CPU Neden Boş Kalmaz?

Tek programlı



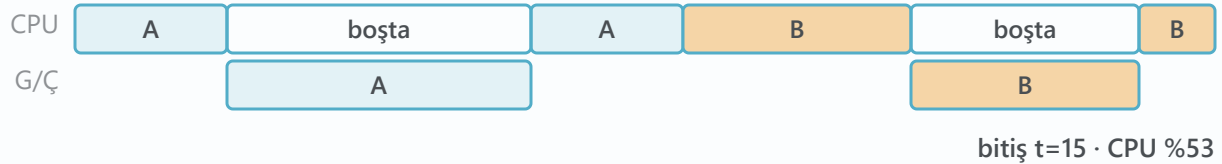
Çoklu programlı



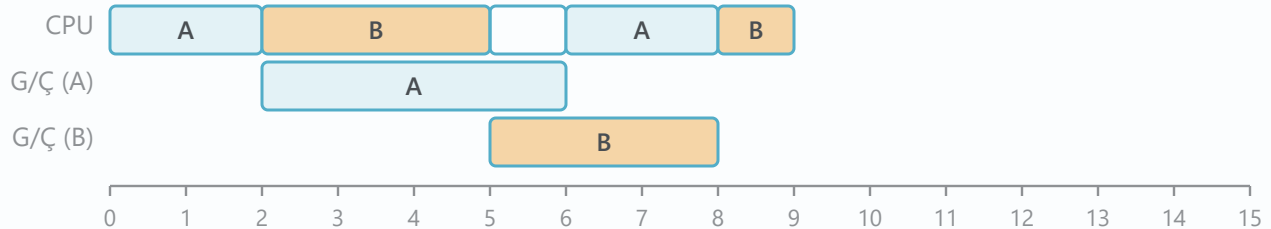
Çoklu programlı: A $t = 2$ 'de G/Ç'ye gidince CPU hemen **B**'ye geçer ($t = 2-5$). A'nın G/Ç'si arka planda sürer.

Çoklu Programlama: CPU Neden Boş Kalmaz?

Tek programlı



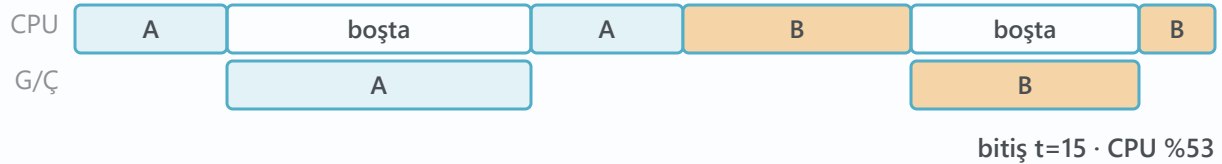
Çoklu programlı



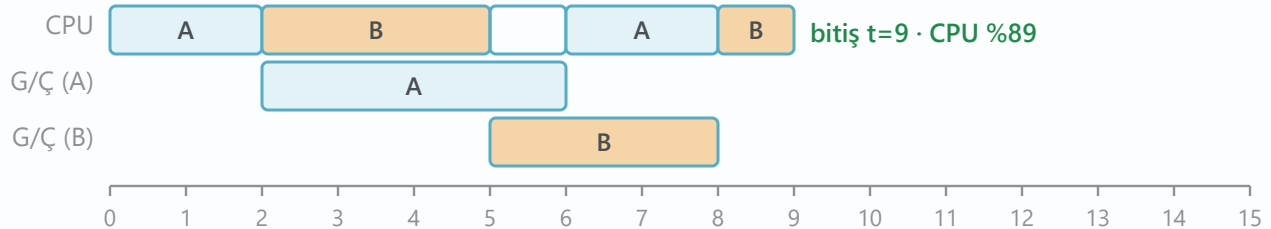
B $t = 5$ 'te G/Ç'ye gider (5–8). A'nın G/Ç'si $t = 6$ 'da biter: CPU yalnız 5–6 arasında boşta; A 6–8, B 8–9 çalışır.

Çoklu Programlama: CPU Neden Boş Kalmaz?

Tek programlı



Çoklu programlı



Aynı iki iş **15 yerine 9** birimde bitti; CPU kullanımı $8 / 9 \approx \textbf{\%89}$. Çoklu programlamanın kazancı, CPU ile G/Ç'nin örtüşmesidir.

4. Nesil: Kişisel Bilgisayarlar

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

Fiyatların hızla düşmesi bilgisayarı herkese açtı; İS'nin hedefi "makineyi verimli kullan"dan "kullanıcıya kolaylık"a kaydı.

5. Nesil: Mobil ve Günümüz

Mobil aygıtlar (1990–)

- 1990'lar: PDA'lar, Palm OS; 2000'ler: Symbian, BlackBerry OS.
- 2007 **iOS** (iPhone), 2008 **Android** (Linux çekirdeği üzerinde).
- Windows Phone 2010'da çıktı, 2017'de geliştirmesi durduruldu.
- Bugün pazarı Android ve iOS paylaşır; dokunmatik arayüz, pil yönetimi, izin modeli öne çıkar.

Sanallaştırma ve bulut

Hipervizörler (KVM, Hyper-V, Xen); tek sunucuda çok sayıda sanal makine.

Konteynerler

Docker, Kubernetes: aynı çekirdeği paylaşan yalıtılmış ortamlar (namespaces, cgroups).

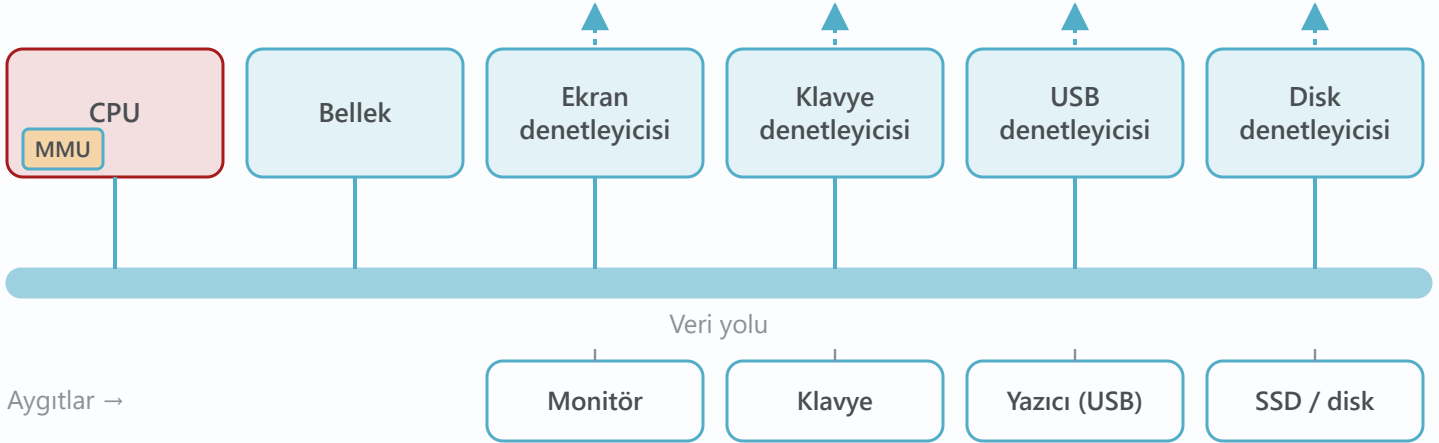
Heterojen donanım

CPU + GPU + NPU; büyük ve küçük çekirdekli hibrit işlemciler.

Güvenlik

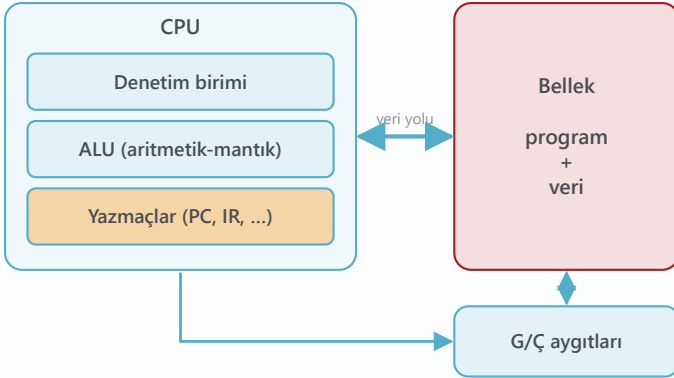
Bellek güvenli diller: Linux 6.1'den beri çekirdekte Rust desteği.

Kişisel Bilgisayarın Bileşenleri



- CPU, bellek ve G/Ç aygıtları bir **veri yolu** üzerinden haberleşir; her aygıtın bir **denetleyicisi** vardır.
- **MMU** (bellek yönetim birimi) CPU içindedir ve sanal adresleri fiziksel adrese çevirir.
- Gerçek sistemlerde tek ortak hat yerine noktadan noktaya bağlantılar (PCIe) kullanılır; ama kavramsal model aynıdır.

Von Neumann Mimarisi

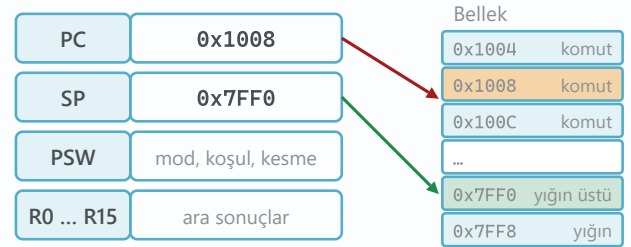


- **Program ve veri aynı bellekte** durur; komutlar da veri gibi okunur (saklı program fikri, 1945).
- Komutlar bellekten **sırayla** getirilir ve yürütülür.
- **Von Neumann darboğazı**: CPU, bellekten çok daha hızlıdır; aradaki dar yol performansı sınırlar → önbellekler.
- **Harvard** mimarisi komut ve veri için ayrı yollar kullanır. Modern CPU'lar "değiştirilmiş Harvard"dır: L1 önbellek komut (L1i) ve veri (L1d) olarak ayrılır.

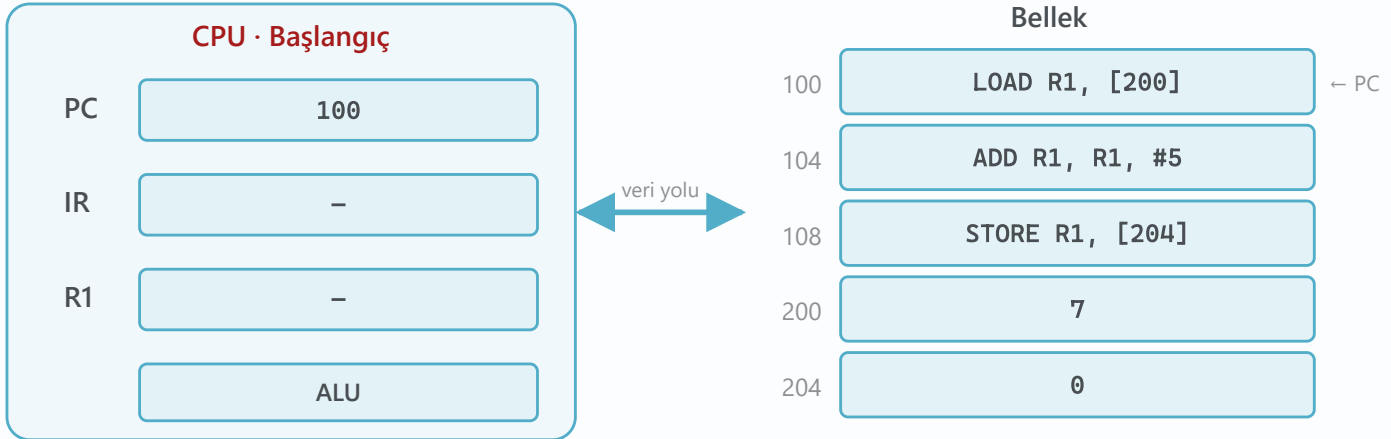
İşlemci ve Yazmaçlar

- Bilgisayarın "beyni": tüm komutları işlemci (CPU) yürütür.
- Temel döngü: komutu **getir** (fetch), **çöz** (decode), **yürüt** (execute).
- Bellek yavaş olduğu için değişkenler ve ara sonuçlar **yazmaçlarda** tutulur: *yükle* (bellek → yazmaç), *sakla* (yazmaç → bellek).
- Her işlemci ailesinin kendi **komut kümesi** vardır: x86-64 programı ARM'da doğrudan çalışmaz.
- Performans: saat hızı, çekirdek sayısı, önbellek boyutu, boru hattı derinliği, bellek bant genişliği.
- İS, bağlam değiştirirken (süreç değişimi) tüm yazmaçları **kaydeder ve geri yükler**.

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

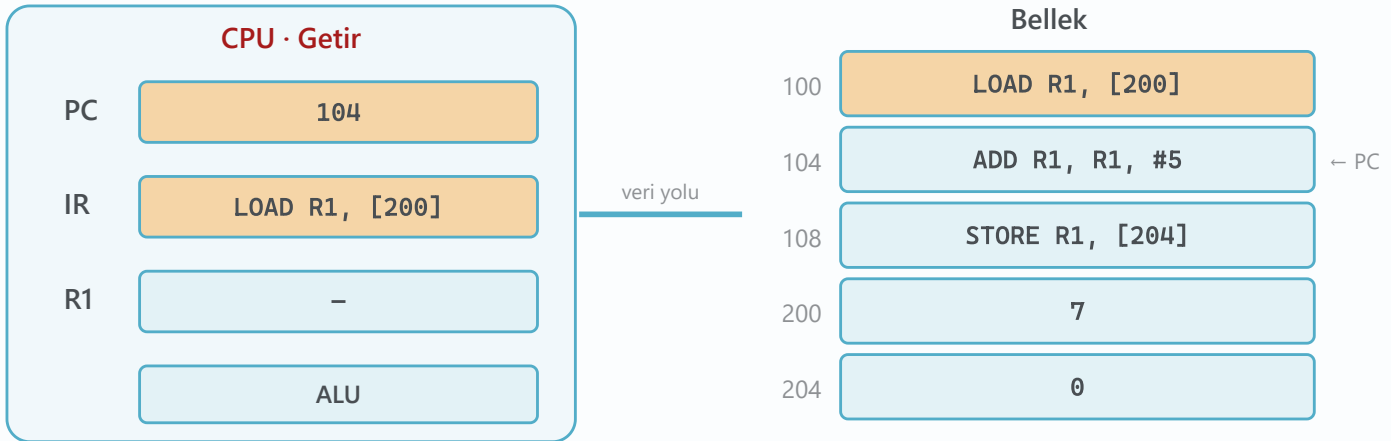


Getir-Çöz-Yürüt Döngüsü



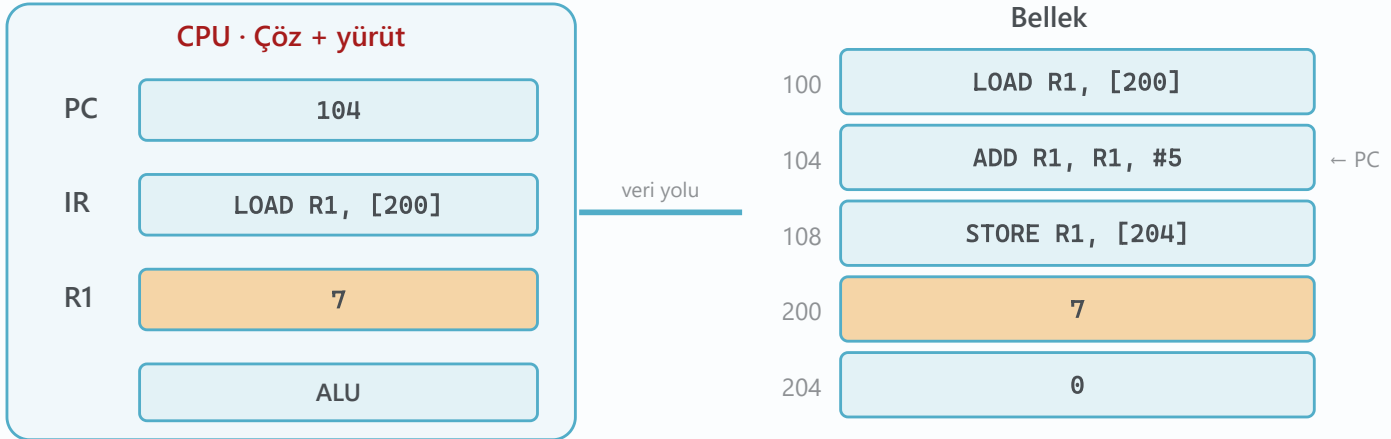
Program bellekte 100. adresten başlıyor; **PC = 100**. Bellek[200] = 7, bellek[204] = 0.

Getir-Çöz-Yürüt Döngüsü



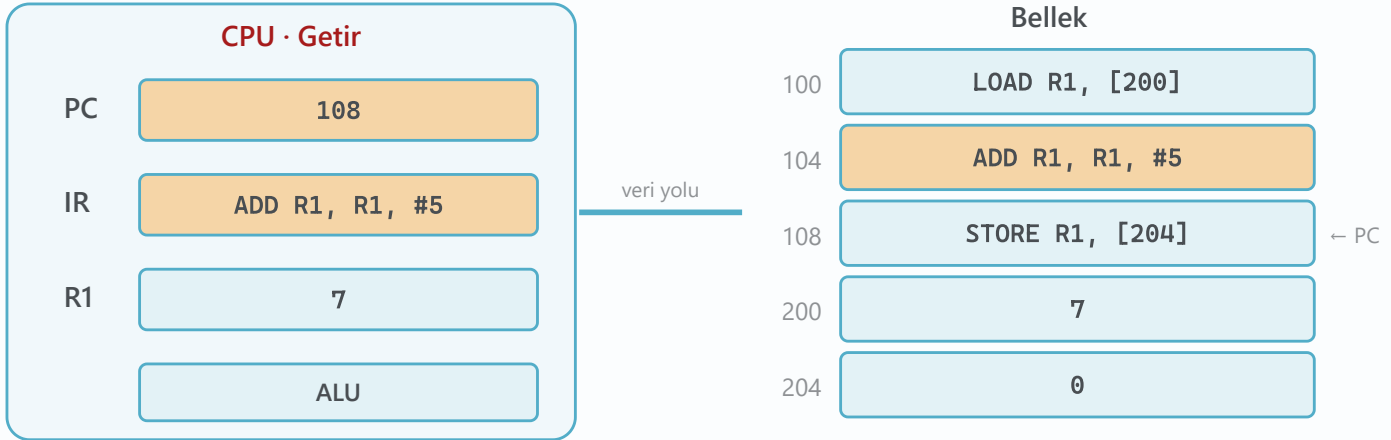
Getir: PC'nin gösterdiği komut IR'ye okunur: `LOAD R1, [200]`. PC bir sonraki komuta ilerler: **104** (komutlar 4 bayt).

Getir-Çöz-Yürüt Döngüsü



Çöz + yürüt: bellek[200] okunur, **R1 = 7**.

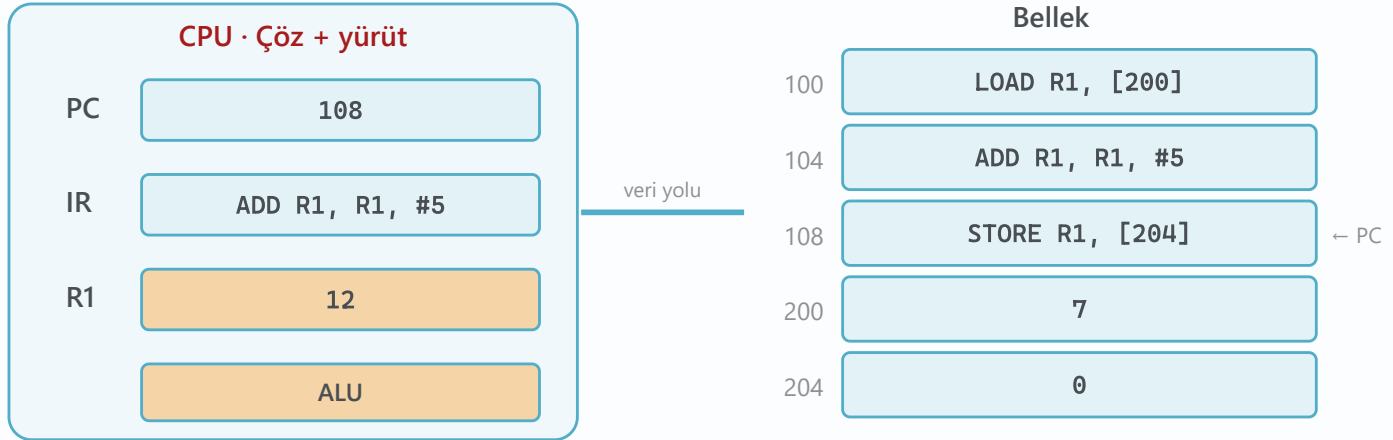
Getir-Çöz-Yürüt Döngüsü



Getir: IR = `ADD R1, R1, #5`, PC = 108.

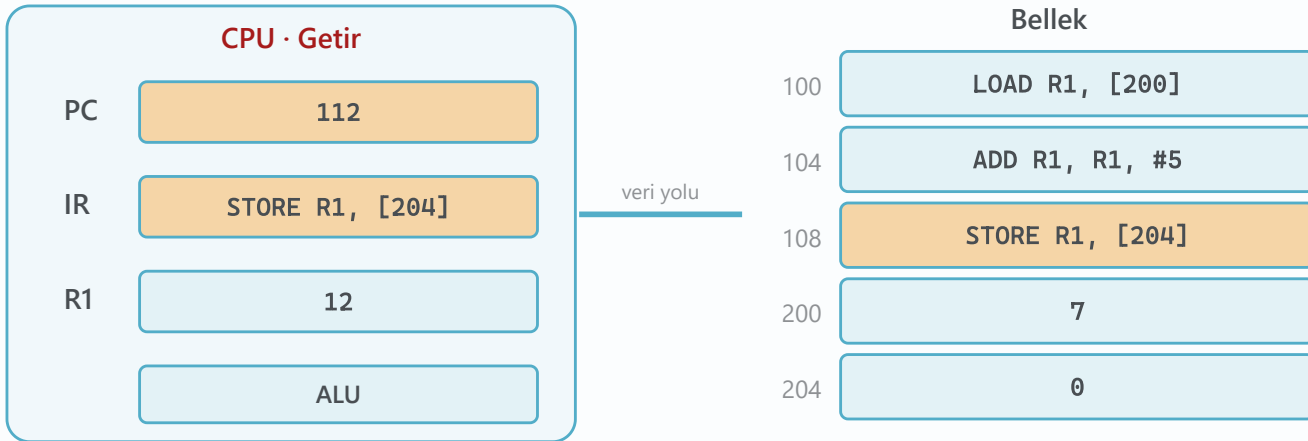
Adım 4 / 7

Getir-Çöz-Yürüt Döngüsü



Yürüt: ALU toplar: $R1 = 7 + 5 = 12$. Belleğe hiç gidilmedi; yazmaç işlemi hızlıdır.

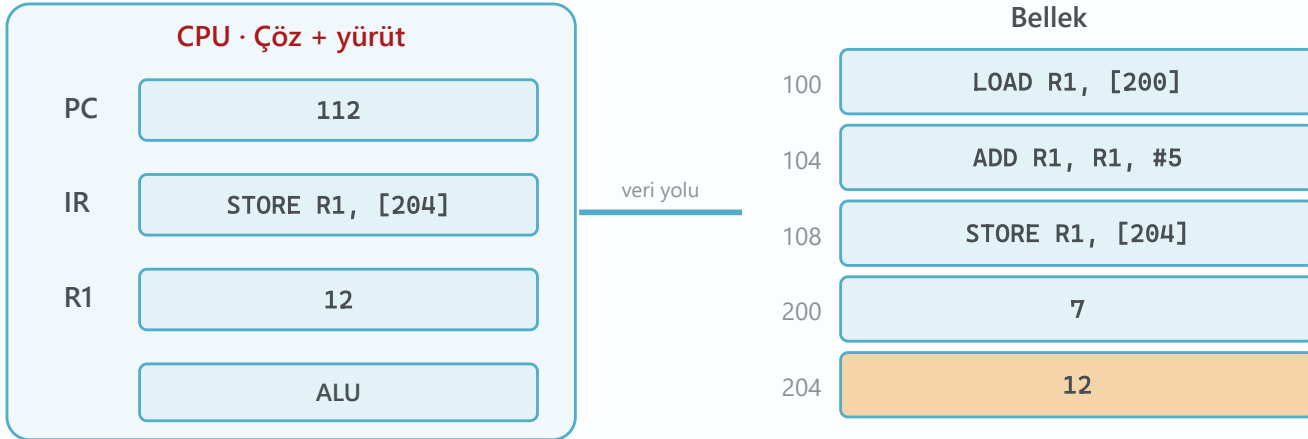
Getir-Çöz-Yürüt Döngüsü



Getir: IR = STORE R1, [204] , PC = 112.

Adım 6 / 7

Getir-Çöz-Yürüt Döngüsü



Yürüt: R1 belleğe saklanır: **bellek[204] = 12**. Döngü PC = 112'den sürer.

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

```

▶ int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
▶ int pc = 0, acc = 0, ir, op, adr;

3 while (1) {
4   ir = M[pc];          /* getir */
5   pc = pc + 1;
6   op = ir / 100;       /* çöz */
7   adr = ir % 100;
8   if (op == 1) acc = M[adr]; /* LOAD */
9   else if (op == 2) acc += M[adr];
10  else if (op == 3) M[adr] = acc;
11  else break;          /* HALT */
12 }

```

Bellek M[]

başlangıç

105	206	307	0	0	6	9	0
0	1	2	3	4	5	6	7
LOAD 5	ADD 6	STORE 7	HALT	veri	veri	veri	veri
PC		IR		op		adr	
0		-		-		-	
						ACC	
						0	

Program ve veri **aynı bellekte**: M[0..3] komutlar (yüzler basamağı işlem kodu, kalan iki basamak adres), M[5], M[6] veri. **pc = 0, acc = 0.**

Adım 1 / 12

DEĞİŞKENLER

pc = 0

ir = ?

op = ?

adr = ?

acc = 0

M[7] = 0

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

```

1 int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
2 int pc = 0, acc = 0, ir, op, adr;
3 while (1) {
4     ▶ ir = M[pc];           /* getir */
5     ▶ pc = pc + 1;
6     op = ir / 100;         /* çöz */
7     adr = ir % 100;
8     if (op == 1) acc = M[adr]; /* LOAD */
9     else if (op == 2) acc += M[adr];
10    else if (op == 3) M[adr] = acc;
11    else break;           /* HALT */
12 }

```

Bellek M[]

105	206	307	0	0	6	9	0
0	1	2	3	4	5	6	7
LOAD 5	ADD 6	STORE 7	HALT	veri	veri	veri	veri

getir

PC	IR	op	adr	ACC
1	105	—	—	0

Getir: PC'nin gösterdiği M[0] = 105, IR'ye okunur; PC hemen bir sonraki komuta ilerler: **pc = 1**.

Adım 2 / 12

DEĞİŞKENLER

pc = 1

ir = 105

op = ?

adr = ?

acc = 0

M[7] = 0

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

```

1 int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
2 int pc = 0, acc = 0, ir, op, adr;
3 while (1) {
4     ir = M[pc];          /* getir */
5     pc = pc + 1;
6     ▶ op = ir / 100;      /* çöz */
7     ▶ adr = ir % 100;
8     if (op == 1) acc = M[adr]; /* LOAD */
9     else if (op == 2) acc += M[adr];
10    else if (op == 3) M[adr] = acc;
11    else break;          /* HALT */
12 }

```

Bellek M[]

çöz

105	206	307	0	0	6	9	0
0	1	2	3	4	5	6	7
LOAD 5	ADD 6	STORE 7	HALT	veri	veri	veri	veri

PC	IR	op	adr	ACC
1	105	1	5	0
		LOAD 5		

Çöz: $105 / 100 = 1$ (LOAD), $105 \% 100 = 5$ (adres).

Adım 3 / 12

DEĞİŞKENLER

pc = 1

ir = 105

op = 1

adr = 5

acc = 0

M[7] = 0

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

```

1 int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
2 int pc = 0, acc = 0, ir, op, adr;
3 while (1) {
4   ir = M[pc];          /* getir */
5   pc = pc + 1;
6   op = ir / 100;       /* çöz */
7   adr = ir % 100;
8   ▶ if (op == 1) acc = M[adr]; /* LOAD */
9   else if (op == 2) acc += M[adr];
10  else if (op == 3) M[adr] = acc;
11  else break;          /* HALT */
12 }

```

Bellek M[]

yürüt: LOAD

105	206	307	0	0	6	9	0
0	1	2	3	4	5	6	7
LOAD 5	ADD 6	STORE 7	HALT	veri	veri	veri	veri

PC	IR	op	adr	ACC
1	105	1	5	6
LOAD 5				

Yürüt: LOAD, bellekten okur: $acc = M[5] = 6$.

Adım 4 / 12

DEĞİŞKENLER

pc = 1

ir = 105

op = 1

adr = 5

acc = 6

M[7] = 0

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

```

1 int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
2 int pc = 0, acc = 0, ir, op, adr;
3 while (1) {
4     ▶ ir = M[pc];          /* getir */
5     ▶ pc = pc + 1;
6     op = ir / 100;        /* çöz */
7     adr = ir % 100;
8     if (op == 1) acc = M[adr]; /* LOAD */
9     else if (op == 2) acc += M[adr];
10    else if (op == 3) M[adr] = acc;
11    else break;           /* HALT */
12 }

```

Bellek M[]

105	206	307	0	0	6	9	0
0	1	2	3	4	5	6	7
LOAD 5	ADD 6	STORE 7	HALT	veri	veri	veri	veri

getir

PC	IR	op	adr	ACC
2	206	1	5	6
LOAD 5				

Getir: IR = M[1] = 206, **pc** = 2.

Adım 5 / 12

DEĞİŞKENLER

pc = 2

ir = 206

op = 1

adr = 5

acc = 6

M[7] = 0

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

```

1 int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
2 int pc = 0, acc = 0, ir, op, adr;
3 while (1) {
4   ir = M[pc];          /* getir */
5   pc = pc + 1;
6   ▶ op = ir / 100;      /* çöz */
7   ▶ adr = ir % 100;
8   if (op == 1) acc = M[adr]; /* LOAD */
9   else if (op == 2) acc += M[adr];
10  else if (op == 3) M[adr] = acc;
11  else break;          /* HALT */
12 }

```

Bellek M[]

çöz

105	206	307	0	0	6	9	0
0	1	2	3	4	5	6	7
LOAD 5	ADD 6	STORE 7	HALT	veri	veri	veri	veri

PC	IR	op	adr	ACC
2	206	2	6	6
		ADD 6		

Çöz: $206 / 100 = 2$ (ADD), $206 \% 100 = 6$ (adres).

Adım 6 / 12

DEĞİŞKENLER

pc = 2

ir = 206

op = 2

adr = 6

acc = 6

M[7] = 0

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

```

1 int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
2 int pc = 0, acc = 0, ir, op, adr;
3 while (1) {
4   ir = M[pc];          /* getir */
5   pc = pc + 1;
6   op = ir / 100;       /* çöz */
7   adr = ir % 100;
8   if (op == 1) acc = M[adr]; /* LOAD */
9   ▶ else if (op == 2) acc += M[adr];
10  else if (op == 3) M[adr] = acc;
11  else break;          /* HALT */
12 }

```

Bellek M[]

105	206	307	0	0	6	9	0
0	1	2	3	4	5	6	7
LOAD 5	ADD 6	STORE 7	HALT	veri	veri	veri	veri

yürüt: ADD

PC	IR	op	adr	ACC
2	206	2	6	15
ADD 6				

Yürüt: ADD, ALU toplar: $acc = 6 + M[6] = 6 + 9 = 15$.

Adım 7 / 12

DEĞİŞKENLER

pc = 2

ir = 206

op = 2

adr = 6

acc = 15

M[7] = 0

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

```

1 int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
2 int pc = 0, acc = 0, ir, op, adr;
3 while (1) {
4     ▶ ir = M[pc];          /* getir */
5     ▶ pc = pc + 1;
6     op = ir / 100;        /* çöz */
7     adr = ir % 100;
8     if (op == 1) acc = M[adr]; /* LOAD */
9     else if (op == 2) acc += M[adr];
10    else if (op == 3) M[adr] = acc;
11    else break;           /* HALT */
12 }

```

Bellek M[]

105	206	307	0	0	6	9	0
0	1	2	3	4	5	6	7
LOAD 5	ADD 6	STORE 7	HALT	veri	veri	veri	veri

getir

PC	IR	op	adr	ACC
3	307	2	6	15
		ADD 6		

Getir: IR = M[2] = 307, pc = 3.

Adım 8 / 12

DEĞİŞKENLER

pc = 3

ir = 307

op = 2

adr = 6

acc = 15

M[7] = 0

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

```

1 int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
2 int pc = 0, acc = 0, ir, op, adr;
3 while (1) {
4   ir = M[pc];          /* getir */
5   pc = pc + 1;
6   ▶ op = ir / 100;      /* çöz */
7   ▶ adr = ir % 100;
8   if (op == 1) acc = M[adr]; /* LOAD */
9   else if (op == 2) acc += M[adr];
10  else if (op == 3) M[adr] = acc;
11  else break;          /* HALT */
12 }

```

Bellek M[]

çöz



Çöz: $307 / 100 = 3$ (STORE), $307 \% 100 = 7$ (adres).

Adım 9 / 12

DEĞİŞKENLER

pc = 3

ir = 307

op = 3

adr = 7

acc = 15

M[7] = 0

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

```

1 int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
2 int pc = 0, acc = 0, ir, op, adr;
3 while (1) {
4   ir = M[pc];          /* getir */
5   pc = pc + 1;
6   op = ir / 100;       /* çöz */
7   adr = ir % 100;
8   if (op == 1) acc = M[adr]; /* LOAD */
9   else if (op == 2) acc += M[adr];
10  ▶ else if (op == 3) M[adr] = acc;
11  else break;          /* HALT */
12 }

```

Bellek M[]

yürüt: STORE

105	206	307	0	0	6	9	15
0	1	2	3	4	5	6	7
LOAD 5	ADD 6	STORE 7	HALT	veri	veri	veri	veri
PC		IR		op		adr	
3		307		3		7	
				STORE 7		ACC	
						15	

Yürüt: STORE, yazmacı belleğe yazar: **M[7] = 15.**

Adım 10 / 12

DEĞİŞKENLER

pc = 3

ir = 307

op = 3

adr = 7

acc = 15

M[7] = 15

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

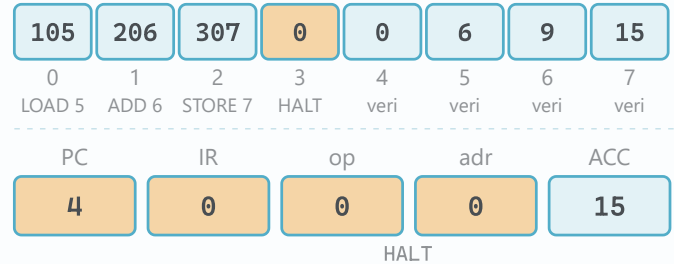
```

1 int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
2 int pc = 0, acc = 0, ir, op, adr;
3 while (1) {
    ▶ ir = M[pc];           /* getir */
    ▶ pc = pc + 1;
    ▶ op = ir / 100;        /* çöz */
    ▶ adr = ir % 100;
    8 if (op == 1) acc = M[adr]; /* LOAD */
    9 else if (op == 2) acc += M[adr];
   10 else if (op == 3) M[adr] = acc;
   11 else break;           /* HALT */
   12 }

```

Bellek M[]

getir + çöz



M[3] = 0 getirilir ve çözülür: **op = 0**, yani HALT.

Adım 11 / 12

DEĞİŞKENLER

pc = 4

ir = 0

op = 0

adr = 0

acc = 15

M[7] = 15

Kod İzleme: Getir-Çöz-Yürüt Benzetimi

```

1 int M[8] = {105, 206, 307, 0, 0, 6, 9, 0};
2 int pc = 0, acc = 0, ir, op, adr;
3 while (1) {
4     ir = M[pc];          /* getir */
5     pc = pc + 1;
6     op = ir / 100;       /* çöz */
7     adr = ir % 100;
8     if (op == 1) acc = M[adr]; /* LOAD */
9     else if (op == 2) acc += M[adr];
10    else if (op == 3) M[adr] = acc;
11    ▶ else break;        /* HALT */
12 }

```

Bellek M[]

yürüt: HALT

105	206	307	0	0	6	9	15	
0	1	2	3	4	5	6	7	
LOAD 5	ADD 6	STORE 7	HALT	veri	veri	veri	veri	
PC		IR		op		adr		ACC
4		0		0		0		15
HALT								

Döngü biter. Sonuç belleğe yazılmış: **M[7] = 15** (6 + 9). Dört komut için döngü dört kez döndü; her turda önce getir, sonra çöz, sonra yürüt.

Adım 12 / 12

DEĞİŞKENLER

pc = 4

ir = 0

op = 0

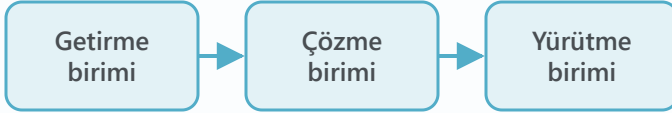
adr = 0

acc = 15

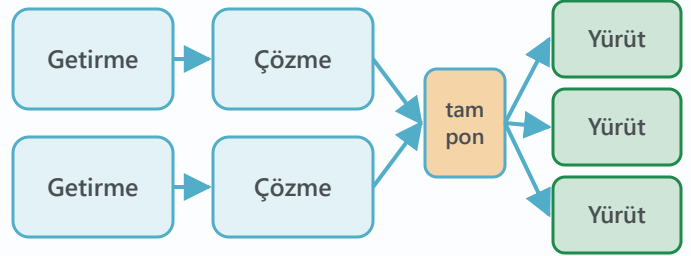
M[7] = 15

Boru Hattı ve Süperskalar

(a) Üç aşamalı boru hattı



(b) Süperskalar işlemci



Boru hattı

Komutların aşamaları üst üste bindirilir: biri yürütülürken sonraki çözülür, ondan sonraki getirilir.

Süperskalar

Birden çok yürütme birimi; tampondaki komutlar sırası dışında (out-of-order) çalıştırılabilir.

İS'ye etkisi

Dallanmada boru hattı boşaltılır; kesmede "hangi komut bitti?" sorusu karmaşıklaşır. Spectre gibi açıklar bu tahminlerden doğdu.

Boru Hattı Adım Adım

	tur 1	tur 2	tur 3	tur 4	tur 5	tur 6
Komut 1	Getir					
Komut 2						
Komut 3						
Komut 4						

Tur 1: Komut 1 **getiriliyor**. Diğer birimler boş.

Boru Hattı Adım Adım

	tur 1	tur 2	tur 3	tur 4	tur 5	tur 6
Komut 1	Getir	Çöz				
Komut 2		Getir				
Komut 3						
Komut 4						

Tur 2: Komut 1 **çözülüyor**, aynı anda Komut 2 getiriliyor.

Boru Hattı Adım Adım

	tur 1	tur 2	tur 3	tur 4	tur 5	tur 6
Komut 1	Getir	Çöz	Yürüt			
Komut 2		Getir	Çöz			
Komut 3			Getir			
Komut 4						

Tur 3: boru hattı **dolu**: üç komut üç farklı aşamada. Bundan sonra her turda bir komut tamamlanır.

Boru Hattı Adım Adım

	tur 1	tur 2	tur 3	tur 4	tur 5	tur 6
Komut 1	Getir	Çöz	Yürüt			
Komut 2		Getir	Çöz	Yürüt		
Komut 3			Getir	Çöz		
Komut 4				Getir		

Tur 4: Komut 1 bitti; Komut 2 yürütülüyor, 3 çözülüyor, 4 getiriliyor.

Boru Hattı Adım Adım

	tur 1	tur 2	tur 3	tur 4	tur 5	tur 6
Komut 1	Getir	Çöz	Yürüt			
Komut 2		Getir	Çöz	Yürüt		
Komut 3			Getir	Çöz	Yürüt	
Komut 4				Getir	Çöz	

Tur 5: Komut 3 yürütülüyor, Komut 4 çözülüyor.

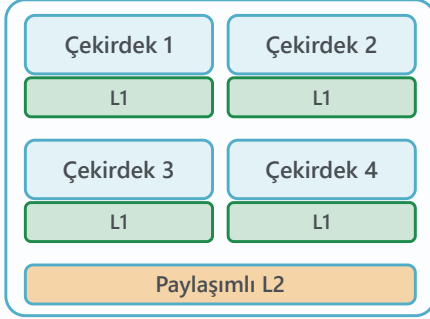
Boru Hattı Adım Adım

	tur 1	tur 2	tur 3	tur 4	tur 5	tur 6
Komut 1	Getir	Çöz	Yürüt			
Komut 2		Getir	Çöz	Yürüt		
Komut 3			Getir	Çöz	Yürüt	
Komut 4				Getir	Çöz	Yürüt

Tur 6: 4 komut **6 turda** bitti; boru hattı olmasaydı $4 \times 3 = 12$ tur sürerdi. Bir dallanma yanlış tahmin edilirse sonraki aşamalar boşaltılır.

Çok Çekirdekli Yongalar

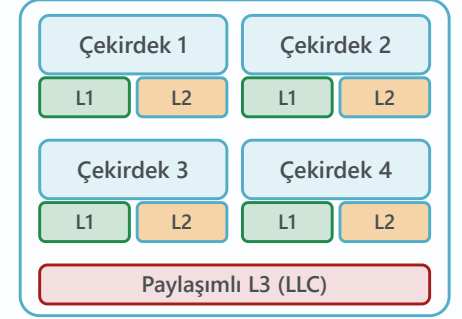
(a) Paylaşımlı L2



(b) Çekirdeğe özel L2



(c) Günümüz: özel L1+L2, ortak L3



Çok çekirdek

Bir yongada birden çok bağımsız CPU; İS her birini ayrı işlemci olarak görür ve çizelgeler.

SMT / Hyper-Threading

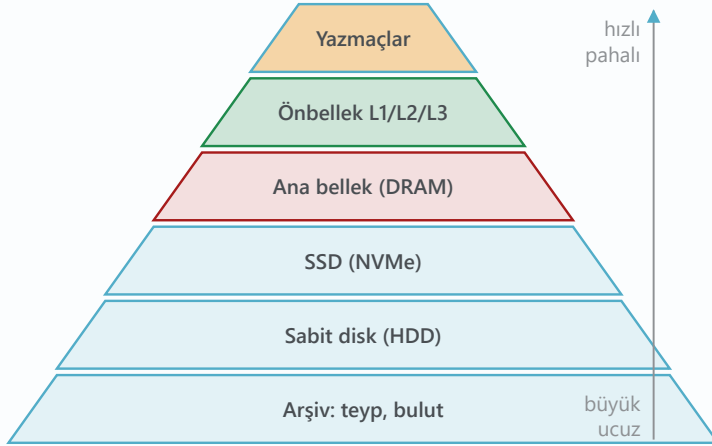
Bir çekirdek iki donanım iş parçacığı çalıştırır; İS'ye iki "mantıksal CPU" görünür ama kaynakları paylaşırlar.

Hibrit çekirdekler

Performans (P) ve verimlilik (E) çekirdekleri (ARM big.LITTLE, Intel 2021+): çizelgeleyici hangisini seçeceğini bilmelidir.

İS açısından sonuç: paralellik, senkronizasyon ve önbellek tutarlılığı sorunları (Bölüm 6 ve 12).

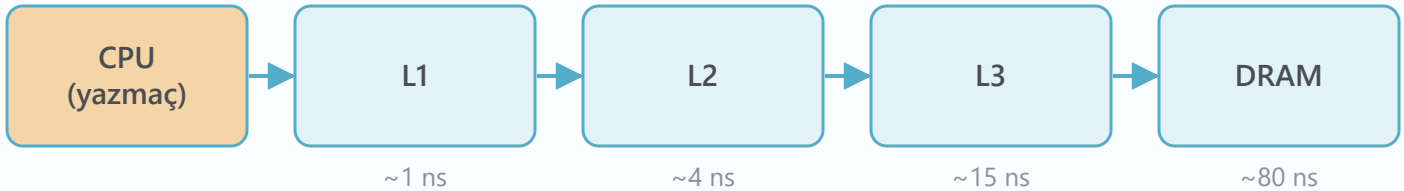
Bellek Hiyerarşisi



Düzye	Erişim (yakl.)	Kapasite (tipik)
Yazmaçlar	< 1 ns	< 1 KB
L1 ön bellek	~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

DRAM ve üstü **uçucudur** (güç gidince silinir); SSD ve HDD kalıcıdır. Değerler 2026 itibarıyla yaklaşık büyüklüklerdir.

Bir Okuma Hiyerarşide Nasıl İlerler?

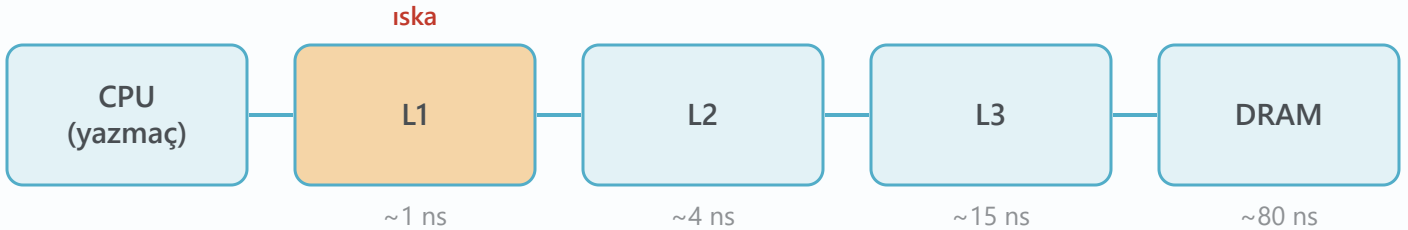


toplam gecikme: 0 ns

CPU `x` değişkenini okumak istiyor (`LOAD R1, [x]`). Önce en yakın ve en hızlı düzeye bakılır.

Adım 1 / 5

Bir Okuma Hiyerarşide Nasıl İlerler?



toplam gecikme: ~1 ns

L1'de yok: **ıska** (miss). ~1 ns kaybedildi; aramaya bir alt düzeyde devam edilir.

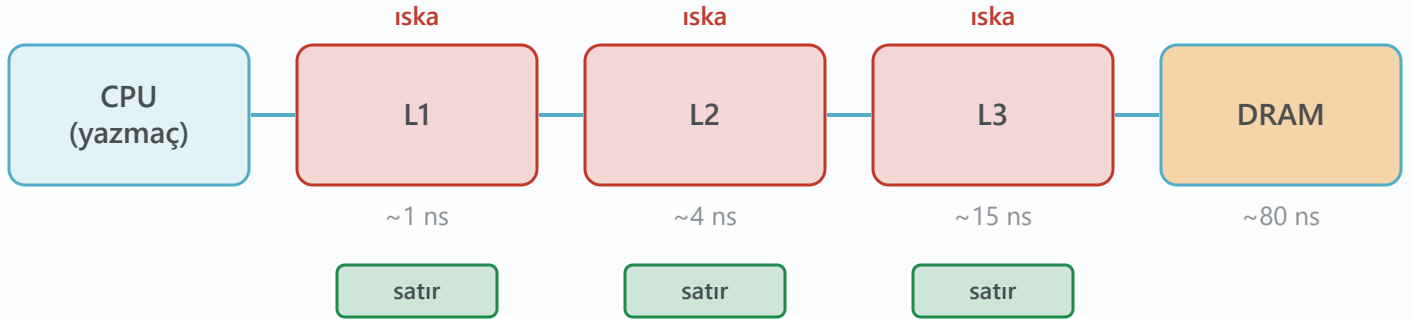
Bir Okuma Hiyerarşide Nasıl İlerler?



toplam gecikme: ~20 ns

L2 ve L3'te de ıska. Her düzey biraz daha büyük ama daha yavaştır; toplam ~20 ns.

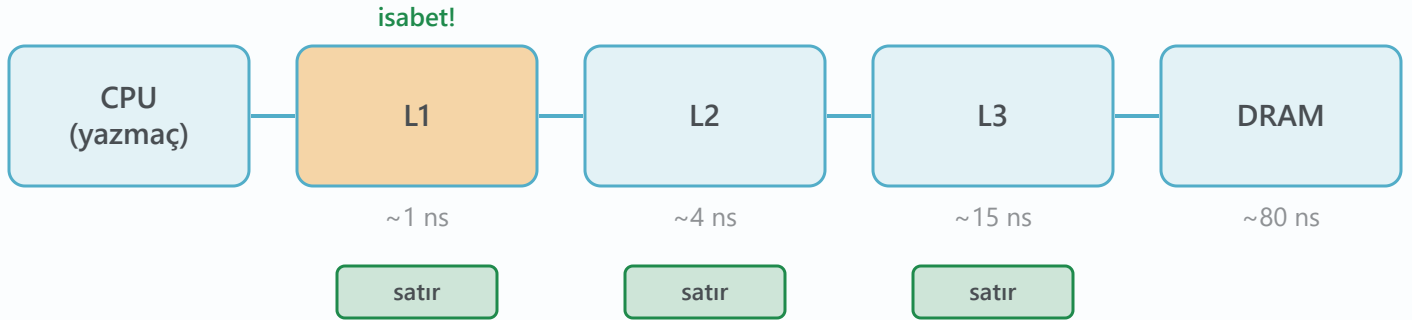
Bir Okuma Hiyerarşide Nasıl İlerler?



toplam gecikme: ~ 100 ns

Veri DRAM'den gelir (~ 80 ns). Tek bir sözcük değil, **64 baytlık satırın tamamı** L3, L2 ve L1'e yerleştirilir.

Bir Okuma Hiyerarşide Nasıl İlerler?

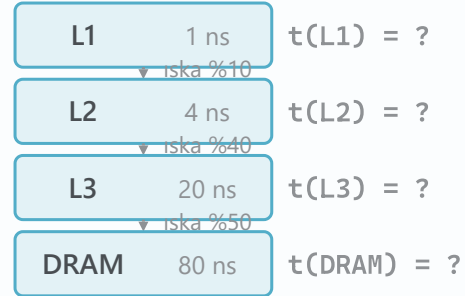


toplam gecikme: ~1 ns

Aynı satırdaki bir veriye ikinci erişim artık **L1 isabeti** (hit): ~1 ns. Önbellek, **yerellik** sayesinde işe yarar.

Kod İzleme: Ortalama Bellek Erişim Süresi

```
▶ /* L1, L2, L3, DRAM erişim süresi (ns) */  
▶ double sure[4] = {1, 4, 20, 80};  
▶ double iska[3] = {0.10, 0.40, 0.50};  
4 double t = sure[3];           /* DRAM */  
5 for (int k = 2; k >= 0; k--)  
6   t = sure[k] + iska[k] * t;  
7 printf("ortalama = %.1f ns\n", t);
```



Dört düzey: erişim süreleri ve her önbellek düzeyinin **iska oranı**. Bir düzeyde iska olursa istek bir alttakine iner.

Adım 1 / 10

DEĞİŞKENLER

t = ?

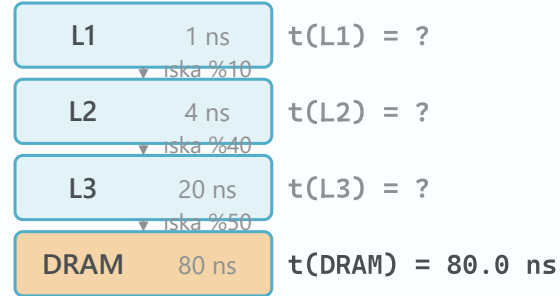
k = ?

ÇIKTI



Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 ► double t = sure[3]; /* DRAM */
5 for (int k = 2; k >= 0; k--)
6     t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



t = 80

En alt düzey DRAM her isteği karşılar: oradan başlayınca **t = 80 ns**.

Adım 2 / 10

DEĞİŞKENLER

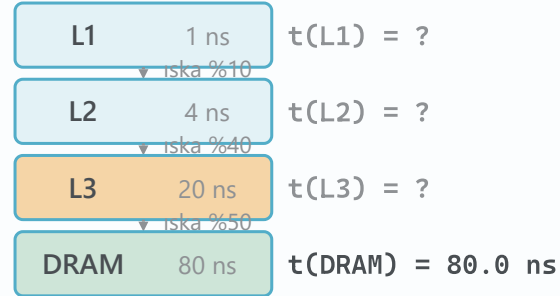
t = 80.0

k = ?

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5 ► for (int k = 2; k >= 0; k--)
6     t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



$k = 2 \geq 0$: bir üst düzeye (L3) çıkılır.

Adım 3 / 10

DEĞİŞKENLER

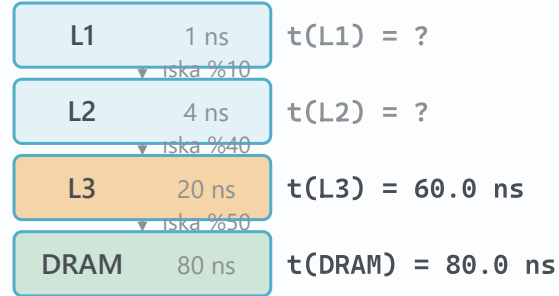
t = 80.0

k = 2

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
6     t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



$$t = 20 + 0.50 \times 80.0 = 60.0$$

L3 her istekte 20 ns harcar; isteklerin %50'sinde alttaki düzeyin 80.0 ns'si eklenir: **t = 60.0 ns**.

Adım 4 / 10

DEĞİŞKENLER

t = 60.0

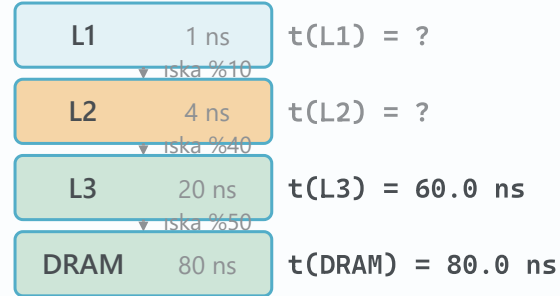
k = 2

ÇIKTI



Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5 ► for (int k = 2; k >= 0; k--)
6     t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



$k = 1 \geq 0$: bir üst düzeye (L2) çıkılır.

Adım 5 / 10

DEĞİŞKENLER

t = 60.0

k = 1

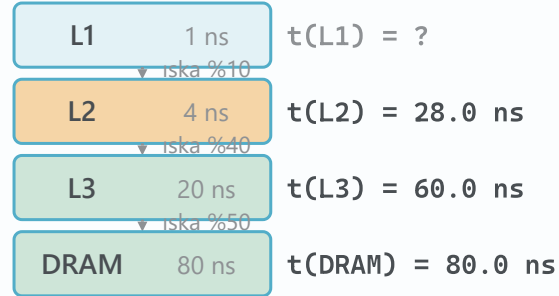
ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```

1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
6     t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);

```



$$t = 4 + 0.40 \times 60.0 = 28.0$$

L2 her istekte 4 ns harcar; isteklerin %40'sinde alttaki düzeyin 60.0 ns'si eklenir: **t = 28.0 ns**.

Adım 6 / 10

DEĞİŞKENLER

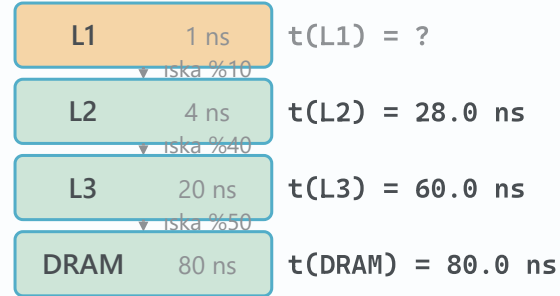
t = 28.0 k = 1

ÇIKTI



Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5
6 ► for (int k = 2; k >= 0; k--)
7     t = sure[k] + iska[k] * t;
8
9 printf("ortalama = %.1f ns\n", t);
```



$k = 0 \geq 0$: bir üst düzeye (L1) çıkılır.

Adım 7 / 10

DEĞİŞKENLER

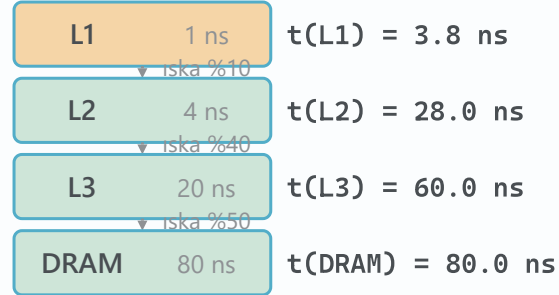
$t = 28.0$

$k = 0$

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
6     t = sure[k] + iska[k] * t;
7 printf("ortalama = %.1f ns\n", t);
```



$$t = 1 + 0.10 \times 28.0 = 3.8$$

L1 her istekte 1 ns harcar; isteklerin %10'sinde alttaki düzeyin 28.0 ns'si eklenir: **t = 3.8 ns.**

Adım 8 / 10

DEĞİŞKENLER

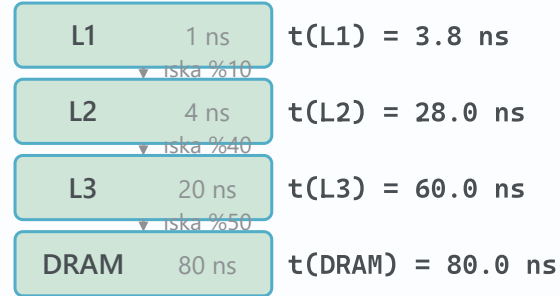
t = 3.8

k = 0

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5
6 ► for (int k = 2; k >= 0; k--)
7     t = sure[k] + iska[k] * t;
8
9 printf("ortalama = %.1f ns\n", t);
```



k = -1: döngü biter.

Adım 9 / 10

DEĞİŞKENLER

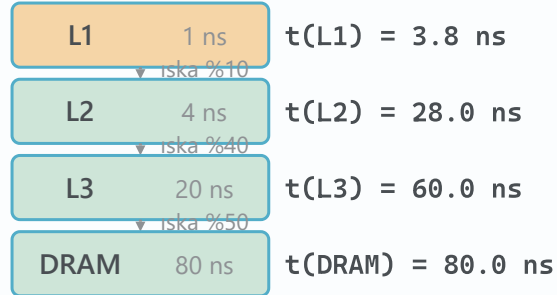
t = 3.8

k = -1

ÇIKTI

Kod İzleme: Ortalama Bellek Erişim Süresi

```
1 /* L1, L2, L3, DRAM erişim süresi (ns) */
2 double sure[4] = {1, 4, 20, 80};
3 double iska[3] = {0.10, 0.40, 0.50};
4 double t = sure[3];          /* DRAM */
5 for (int k = 2; k >= 0; k--)
6   t = sure[k] + iska[k] * t;
7 ▶ printf("ortalama = %.1f ns\n", t);
```



ortalama = 3.8 ns

Her erişimin %10'u L1'i iskalasa da ortalama **3.8 ns**:
DRAM'ın 80 ns'sinden yaklaşık 21 kat hızlı. Hiyerarşiyi işe yarar kılan yüksek isabet oranıdır.

Adım 10 / 10

DEĞİŞKENLER

t = 3.8 k = -1

ÇIKTI

ortalama = 3.8 ns

Önbellek Satırları

- Bellek, **önbellek satırlarına** (cache line, genelde **64 bayt**) bölünür: satır 0 = 0–63, satır 1 = 64–127, ...
- Adres a için satır numarası $\lfloor a / 64 \rfloor$, satır içi konum $a \bmod 64$.
- CPU bir sözcük okurken donanım önce önbelleğe bakar: **isabet** (hit) → L1'de ~4–5 tur; **ıska** (miss) → alt düzey ya da DRAM, yüzlerce tur.
- Önbellek pahalı olduğundan küçüktür; bu yüzden düzeyler (L1, L2, L3) hâlinde kurulur.
- Neden işe yarar? **Zamansal yerellik** (az önce kullanılan yine kullanılır) ve **uzamsal yerellik** (komşu adresler birlikte kullanılır).

Adres 200 (64 B satır):



satır no = 3

satır içi ofset = 8 (6 bit)



Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası

0	8	64	256	0	72
---	---	----	-----	---	----

Önbellek (4 küme)

küme blok (etiket)

0	—
1	—
2	—
3	—

Başlangıçta önbellek boş (–). 64 B satır, **4 küme**, doğrudan eşleme: her blok tek bir kümeyle gidebilir.

Adım 1 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
▶ int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     int blok = adres[i] / 64;
6     int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

etiket = [-1, -1, -1, -1]

isabet = 0 iska = 0

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



Önbellek (4 küme)

küme blok (etiket)

0	-
1	-
2	-
3	-

$\text{adres}[0] = 0 \rightarrow \text{blok} = 0 / 64 = 0$, küme
 $= 0 \bmod 4 = 0$.

Adım 2 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     ▶ int blok = adres[i] / 64;
6     ▶ int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

i = 0 **blok = 0** **kume = 0**

etiket = [-1, -1, -1, -1]

isabet = 0 iska = 0

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



M

Önbellek (4 küme)

küme blok (etiket)

Küme 0 boş → **iska**; blok 0 yüklenir.

Adım 3 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     int blok = adres[i] / 64;
6     int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     ▶ else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

i = 0 blok = 0 kume = 0

etiket = [0, -1, -1, -1]

isabet = 0 iska = 1

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



M

Önbellek (4 küme)

küme blok (etiket)

0	blok 0
1	-
2	-
3	-

$\text{adres}[1] = 8 \rightarrow \text{blok} = 8 / 64 = \mathbf{0}$, küme
 $= 0 \bmod 4 = \mathbf{0}$.

Adım 4 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     ▶ int blok = adres[i] / 64;
6     ▶ int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

i = 1 blok = 0 kume = 0

etiket = [0, -1, -1, -1]

isabet = 0 iska = 1

Kod İzleme: Doğrudan Eşlemeli Ön bellek

Erişim sırası



Ön bellek (4 küme)

küme	blok (etiket)
0	blok 0
1	-
2	-
3	-

etiket[0] = 0 = blok → **isabet**. 8, 0 ile aynı 64 baytlık satırda: **uzamsal yerellik**.

Adım 5 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     int blok = adres[i] / 64;
6     int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

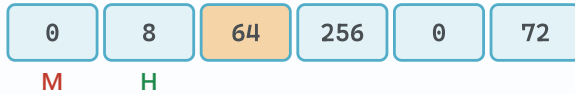
i = 1 blok = 0 kume = 0

etiket = [0, -1, -1, -1]

isabet = 1 iska = 1

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



Önbellek (4 küme)

küme blok (etiket)

0	blok 0
1	-
2	-
3	-

$\text{adres}[2] = 64 \rightarrow \text{blok} = 64 / 64 = 1$,
 $\text{küme} = 1 \bmod 4 = 1$.

Adım 6 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     ▶ int blok = adres[i] / 64;
6     ▶ int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

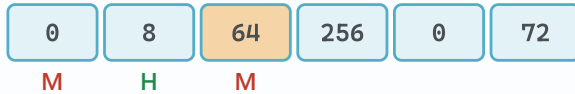
i = 2 **blok = 1** **kume = 1**

etiket = [0, -1, -1, -1]

isabet = 1 iska = 1

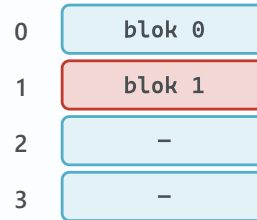
Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



Önbellek (4 küme)

küme blok (etiket)

Küme 1 boş → **iska**; blok 1 yüklenir.

Adım 7 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     int blok = adres[i] / 64;
6     int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

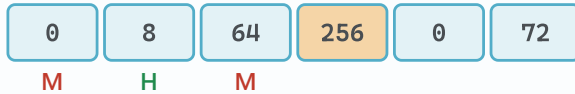
i = 2 blok = 1 kume = 1

etiket = [0, 1, -1, -1]

isabet = 1 iska = 2

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	blok 1
2	-
3	-

$\text{adres}[3] = 256 \rightarrow \text{blok} = 256 / 64 = 4$,
 $\text{küme} = 4 \bmod 4 = 0$.

Adım 8 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     ▶ int blok = adres[i] / 64;
6     ▶ int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

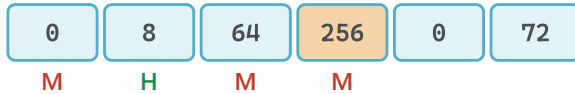
i = 3 **blok = 4** **kume = 0**

etiket = [0, 1, -1, -1]

isabet = 1 iska = 2

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



Önbellek (4 küme)

küme blok (etiket)

0	blok 4
1	blok 1
2	-
3	-

Küme 0'da blok 0 var → **iska**; blok 4 onu **dışarı atar** (çakışma iskası).

Adım 9 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     int blok = adres[i] / 64;
6     int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

i = 3 blok = 4 kume = 0

etiket = [4, 1, -1, -1]

isabet = 1 iska = 3

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



Önbellek (4 küme)



$\text{adres}[4] = 0 \rightarrow \text{blok} = 0 / 64 = 0$, küme
 $= 0 \bmod 4 = 0$.

Adım 10 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     ▶ int blok = adres[i] / 64;
6     ▶ int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

i = 4 **blok = 0** kume = 0

etiket = [4, 1, -1, -1]

isabet = 1 iska = 3

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	blok 1
2	-
3	-

Küme 0'da blok 4 var → **ıska**; blok 0 onu **dışarı atar** (çakışma ıskası).

Adım 11 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     int blok = adres[i] / 64;
6     int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     ▶ else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

i = 4

blok = 0

kume = 0

etiket = [0, 1, -1, -1]

isabet = 1

ıska = 4

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



Önbellek (4 küme)

küme blok (etiket)

0	blok 0
1	blok 1
2	-
3	-

$\text{adres}[5] = 72 \rightarrow \text{blok} = 72 / 64 = 1$,
 $\text{küme} = 1 \bmod 4 = 1$.

Adım 12 / 14

```

1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     ▶ int blok = adres[i] / 64;
6     ▶ int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }

```

DEĞİŞKENLER

i = 5 **blok = 1** **kume = 1**

etiket = [0, 1, -1, -1]

isabet = 1 iska = 4

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



Önbellek (4 küme)

küme	blok (etiket)
0	blok 0
1	blok 1
2	-
3	-

etiket[1] = 1 = blok → **isabet**. 72, 64 ile aynı satırda.

Adım 13 / 14

```
1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     int blok = adres[i] / 64;
6     int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }
```

DEĞİŞKENLER

i = 5 blok = 1 kume = 1

etiket = [0, 1, -1, -1]

isabet = 2 iska = 4

Kod İzleme: Doğrudan Eşlemeli Önbellek

Erişim sırası



Önbellek (4 küme)

küme blok (etiket)

0	blok 0
1	blok 1
2	-
3	-

Döngü bitti: **2 isabet, 4 iska** (isabet oranı %33). 0 ve 256 aynı kümeye düştüğü için birbirini kovdu; 2 yollu küme ilişkili önbellek bunu önlerdi. (Burada etiket yerine blok numarasının tamamı saklandı.)

Adım 14 / 14

```

1 int adres[6] = {0, 8, 64, 256, 0, 72};
2 int etiket[4] = {-1, -1, -1, -1};
3 int isabet = 0, iska = 0;
4 for (int i = 0; i < 6; i++) {
5     int blok = adres[i] / 64;
6     int kume = blok % 4;
7     if (etiket[kume] == blok) isabet++;
8     else { etiket[kume] = blok; iska++; }
9 }

```

DEĞİŞKENLER

etiket = [0, 1, -1, -1]

isabet = 2 iska = 4

Önbellek Tasarım Soruları

1. Ne zaman yerleştirilir?

Genelde ıskada; donanım ayrıca sıradaki satırları önceden getirir (prefetch).

2. Hangi satıra konur?

Eşleme: doğrudan eşlemeli, k yollu küme ilişkili ya da tam ilişkili.

3. Yer gerekince hangisi çıkar?

Değiştirme politikası: LRU'ya yakın yaklaşımlar, rastgele, FIFO.

4. Çıkan nereye gider?

Değiştirilmişse ("kirli") alt düzeye yazılır (write-back); ya da her yazma hemen aşağı iletilir (write-through).

Boyut

Büyük önbellek daha çok isabet ama daha yavaş ve pahalı.

Değiştirme

Performans ile donanım karmaşıklığı arasında denge.

Tutarlılık

Çok çekirdekte aynı satırın kopyaları güncel kalmalı (MESI gibi protokoller).

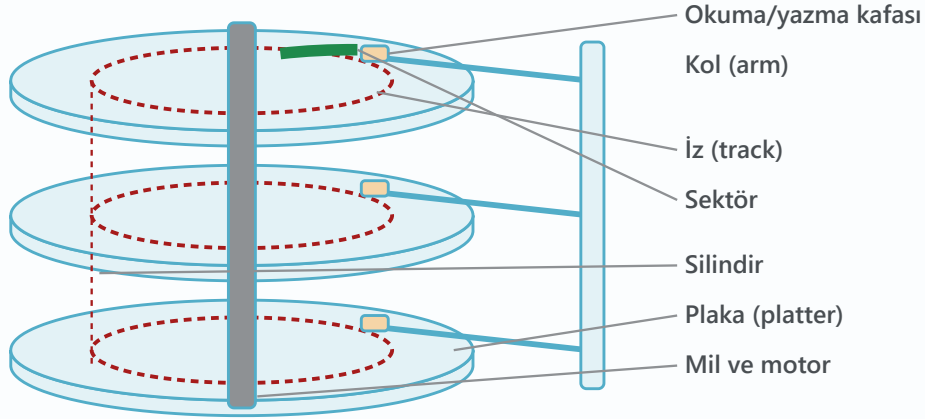
Aynı dört soru İS'nin yönettiği "önbelleklerde" de karşımıza çıkar: sayfa önbelleği, sanal bellek, TLB.

Ana Bellek Türleri

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 ve yazma ömrü sınırlı.	UEFI/BIOS donanım yazılımı (SPI flash), SSD, USB bellek, telefon ve müzik çalar depolaması (NAND flash)
CMOS / NVRAM	Pil ya da flash ile korunan küçük alan.	Saat, önyükleme ayarları

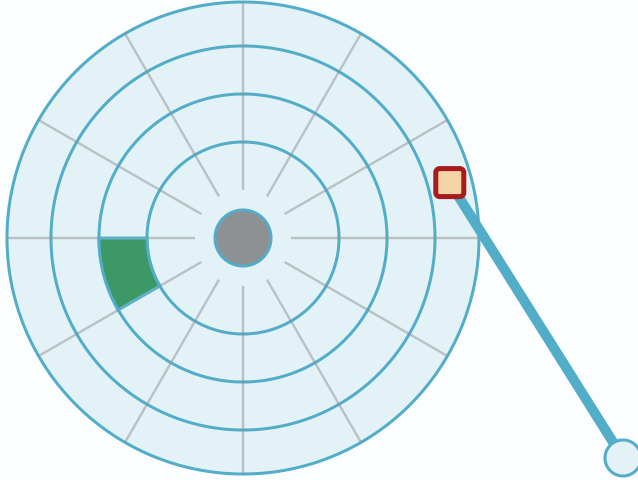
Bugünkü bilgisayarlarda BIOS/UEFI kodu ROM'da değil **flash bellekte** durur; bu yüzden güncellenebilir. Taşınabilir müzik çalarlar da disk değil NAND flash kullanır.

Disk (HDD) Yapısı



- Veriyi **kalıcı** saklar; ucuz ve büyük ama **mekanik** olduğu için yavaştır.
- Her plaka yüzeyinde eş merkezli **izler**, izler de **sektörlere** (512 B / 4 KB) bölünür; aynı konumdaki izlerin kümesi **silindirdir**.
- Tüm kafalar tek bir kola bağlıdır; motor plakaları 5400–7200 (sunucuda 10 000+) devir/dakika döndürür, denetleyici işlemleri yönetir.

Diskten Bir Sektör Okumak



Komut

LBA 1234 oku

Arama

~5 ms

Dönme

ort. 4,17 ms

Aktarım + ECC

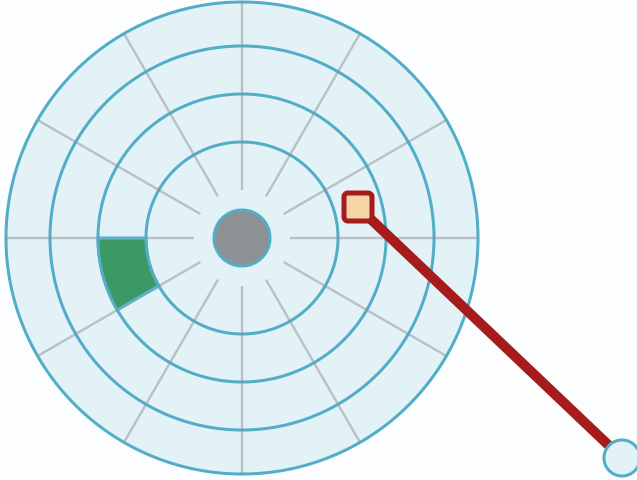
~0,01 ms

DMA + kesme

bitti

Sürücü denetleyiciye "**LBA 1234'ü oku**" der. Bugün diskler mantıksal blok adresi (LBA) kullanır; denetleyici bunu kendi iç (silindir, kafa, sektör) geometrisine çevirir.

Diskten Bir Sektör Okumak



Komut

LBA 1234 oku

Arama

~5 ms

Dönme

ort. 4,17 ms

Aktarım + ECC

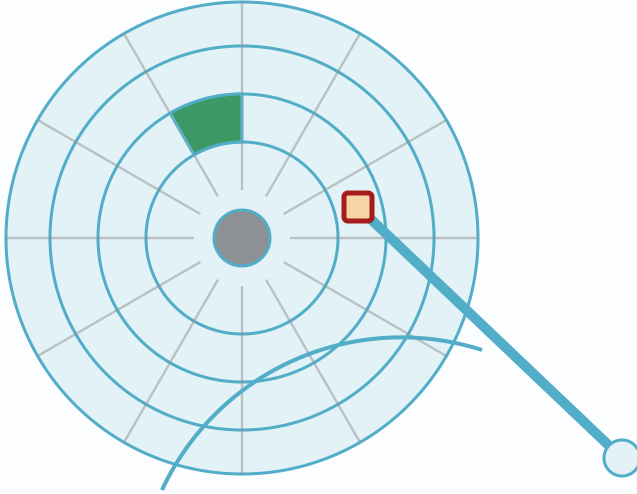
~0,01 ms

DMA + kesme

bitti

Arama (seek): kol, kafaları doğru silindire taşır. Mekanik hareket: birkaç milisaniye.

Diskten Bir Sektör Okumak



Komut

LBA 1234 oku

Arama

~5 ms

Dönme

ort. 4,17 ms

Aktarım + ECC

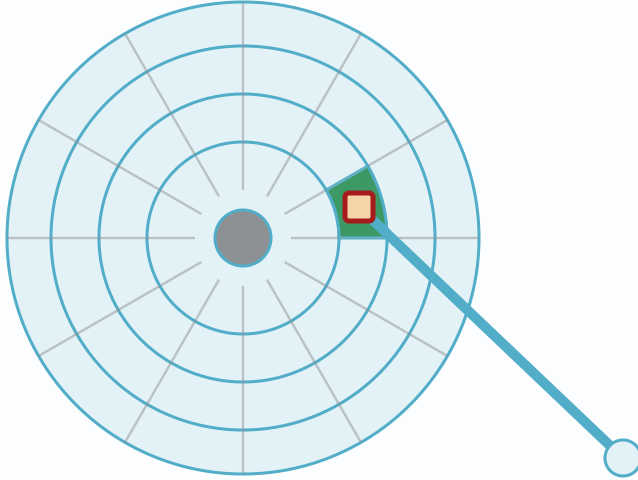
~0,01 ms

DMA + kesme

bitti

Dönme gecikmesi: hedef sektör kafanın altına gelene kadar beklenir. 7200 dev/dk → tur 8,33 ms, ortalama yarım tur **4,17 ms**.

Diskten Bir Sektör Okumak



Komut

LBA 1234 oku

Arama

~5 ms

Dönme

ort. 4,17 ms

Aktarım + ECC

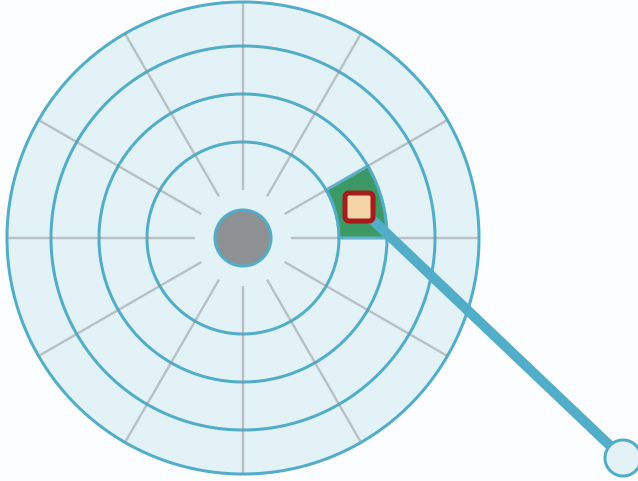
~0,01 ms

DMA + kesme

bitti

Aktarım: sektör kafanın altından geçerken bitler okunur; **ECC** (hata düzeltme kodu) ile denetlenir, gerekirse düzeltilir.

Diskten Bir Sektör Okumak



Komut	LBA 1234 oku
-------	--------------

Arama	~5 ms
-------	-------

Dönme	ort. 4,17 ms
-------	--------------

Aktarım + ECC	~0,01 ms
---------------	----------

DMA + kesme	bitti
-------------	-------

Veri **DMA** ile belleğe yazılır, denetleyici **kesme** üretir. Toplam ~9 ms: CPU bu sürede milyonlarca komut çalıştırabilir, bu yüzden İS beklemez, başka süreç çalıştırır.

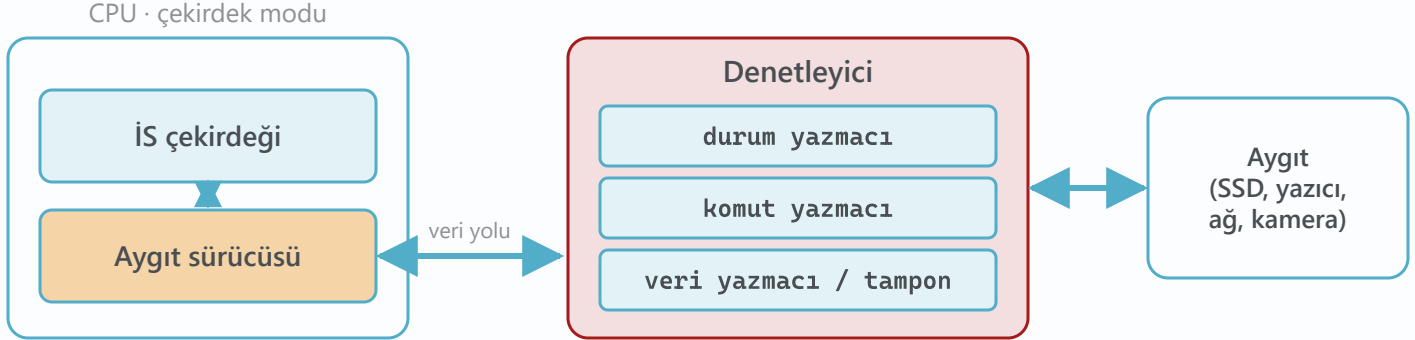
HDD, SSD ve Sanal Bellek

	HDD	SATA SSD	NVMe SSD
Rastgele erişim	5–10 ms	~0,1 ms	0,02–0,1 ms
Rastgele okuma (IOPS)	100–200	~90 000	500 000–1 000 000+
Sıralı okuma	~250 MB/s	~550 MB/s	3–14 GB/s
Arayüz	SATA / SAS	SATA	PCIe (×4)
Hareketli parça	var	yok	yok

SSD: arama ve dönme gecikmesi yoktur; ama flash yerinde yazılamaz, blok blok silinir. Denetleyicideki FTL katmanını ve İS'nin TRIM komutu bunu yönetir.

Sanal bellek: RAM yetmediğinde az kullanılan sayfalar diske/SSD'ye (takas alanı, `pagefile.sys`) yazılır; MMU sanal adresleri fiziksel adrese çevirir (Bölüm 8–9).

G/Ç Aygıtları: Denetleyici ve Sürücü



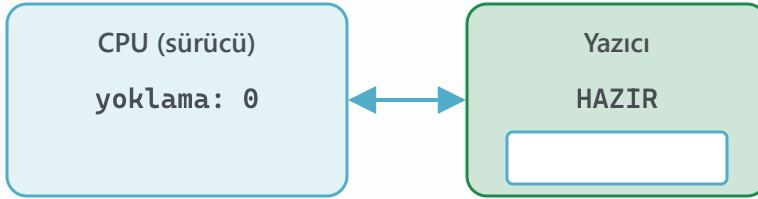
- G/Ç aygıtı iki parçadır: **aygıtın kendisi** ve ona İS'ye basit bir arayüz sunan **denetleyicisi**.
- **Aygıt sürücüsü** denetleyicinin **yazmaçlarına** komut yazar, durumunu okur. Yazmaçlara G/Ç portlarıyla (x86 `in/out`) ya da **bellek eşlemeli G/Ç** (MMIO) ile erişilir.
- Sürücüler genellikle üretici her İS için yazar; çoğu **çekirdek modunda** çalışır (Windows `.sys` , Linux modülleri). Kullanıcı modu sürücüler de vardır: Windows UMDF, Linux VFIO/UIO.
- Sürücü çekirdeğe derlenebilir, açılışta ya da aygıt takıldığında (tak-çalıştır) yüklenebilir.

Üç G/Ç Yöntemi

Yöntem	Nasıl çalışır?	Artı	Eksi
Yoklama (polling, meşgul bekleme)	CPU, aygıt hazır olana kadar durum yazmacını döngüde okur.	Basit; çok kısa beklemelerde en düşük gecikme.	Beklerken CPU başka iş yapamaz.
Kesme (interrupt)	Aygıt işi bitince CPU'ya sinyal gönderir; CPU o arada başka iş yapar.	CPU serbest; duyarlı.	Her kesmenin bağlam kaydetme maliyeti; çok sık kesme yük olur.
DMA (doğrudan bellek erişimi)	Özel denetleyici bloğu aygıt ile bellek arasında taşır; bitince tek kesme.	Büyük veride çok verimli.	Ek donanım; küçük aktarımda kurulum maliyeti.

Yoklama bugün de kullanılır. Çok hızlı aygıtlarda kesme yükü pahalı olduğundan bilerek tercih edilir: Linux ağ yığınınındaki **NAPI** (kesme + yoklama karışımı), NVMe/ `io_uring` yoklama modu, DPDK; ayrıca basit gömülü sistemler.

Kod İzleme: Programlı G/Ç (Yoklama)



Çekirdek, kullanıcının "OS" dizisini çekirdek tamponuna kopyalar ($n = 2$).

Adım 1 / 11

```
► copy_from_user(buf, p, n);
2 for (i = 0; i < n; i++) {
3   while (*durum_yaz != HAZIR)
4     ; /* meşgul bekle */
5   *veri_yaz = buf[i];
6 }
7 return_to_user();
```

DEĞİŞKENLER

i = -

n = 2

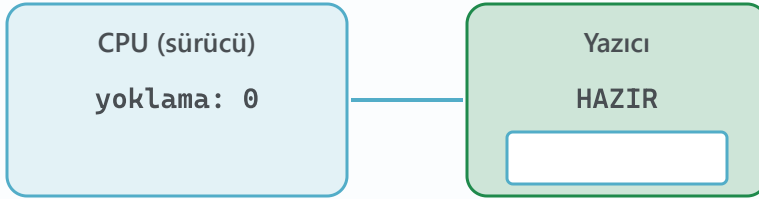
buf = "OS"

durum = HAZIR

ÇIKTI



Kod İzleme: Programlı G/Ç (Yoklama)



Döngü başlar: $i = 0, 0 < 2$.

Adım 2 / 11

```
1 copy_from_user(buf, p, n);
2 ► for (i = 0; i < n; i++) {
3     while (*durum_yaz != HAZIR)
4         ; /* meşgul bekle */
5     *veri_yaz = buf[i];
6 }
7 return_to_user();
```

DEĞİŞKENLER

$i = 0$

$n = 2$

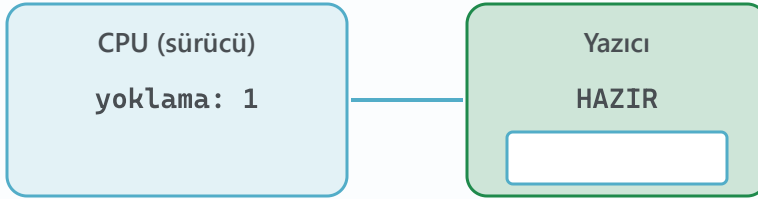
$buf = "OS"$

$durum = HAZIR$

ÇIKTI



Kod İzleme: Programlı G/Ç (Yoklama)



Durum yazmacı okunur: **HAZIR** →
bekleme döngüsünden hemen çıkılır.

Adım 3 / 11

```
1 copy_from_user(buf, p, n);
2 for (i = 0; i < n; i++) {
3     ► while (*durum_yaz != HAZIR)
4         ; /* meşgul bekle */
5     *veri_yaz = buf[i];
6 }
7 return_to_user();
```

DEĞİŞKENLER

i = 0

n = 2

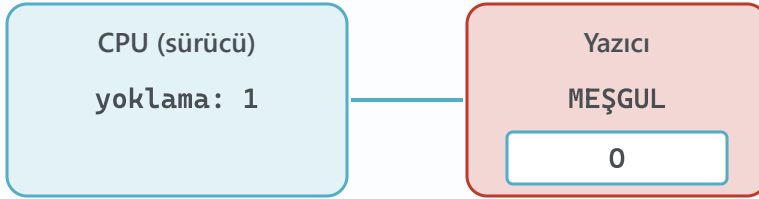
buf = "OS"

durum = **HAZIR**

ÇIKTI



Kod İzleme: Programlı G/Ç (Yoklama)



`buf[0] = '0'` veri yazmacına yazılır;
yazıcı karakteri basarken **MEŞGUL** olur.

Adım 4 / 11

```
1 copy_from_user(buf, p, n);
2 for (i = 0; i < n; i++) {
3     while (*durum_yaz != HAZIR)
4         ; /* meşgul bekle */
5     ► *veri_yaz = buf[i];
6 }
7 return_to_user();
```

DEĞİŞKENLER

`i = 0`

`n = 2`

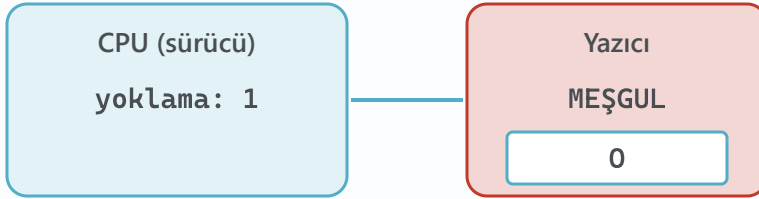
`buf = "OS"`

`durum = MEŞGUL`

ÇIKTI

0

Kod İzleme: Programlı G/Ç (Yoklama)



$i = 1$, $1 < 2$: sıradaki karakter.

Adım 5 / 11

```
1 copy_from_user(buf, p, n);  
▶ for (i = 0; i < n; i++) {  
3   while (*durum_yaz != HAZIR)  
4       ;           /* meşgul bekle */  
5   *veri_yaz = buf[i];  
6 }  
7 return_to_user();
```

DEĞİŞKENLER

$i = 1$

$n = 2$

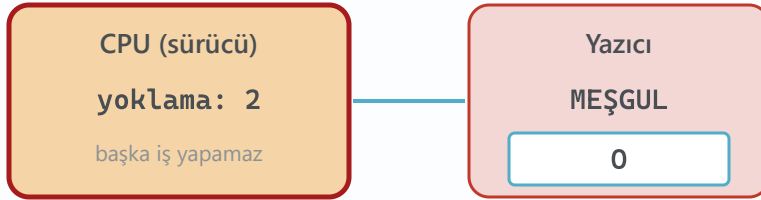
buf = "OS"

durum = MEŞGUL

ÇIKTI

0

Kod İzleme: Programlı G/Ç (Yoklama)



Durum **MEŞGUL** → boş döngü. CPU yalnızca yazmacı yeniden okur.

Adım 6 / 11

```
1 copy_from_user(buf, p, n);
2 for (i = 0; i < n; i++) {
3     ► while (*durum_yaz != HAZIR)
4     ► ;          /* meşgul bekle */
5     *veri_yaz = buf[i];
6 }
7 return_to_user();
```

DEĞİŞKENLER

i = 1

n = 2

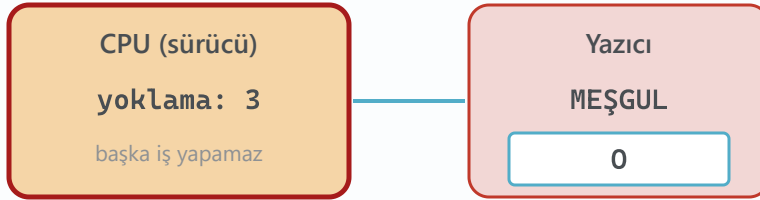
buf = "OS"

durum = **MEŞGUL**

ÇIKTI

0

Kod İzleme: Programlı G/Ç (Yoklama)



Yine MEŞGUL. Gerçekte bu döngü binlerce kez dönebilir: **CPU zamanı boşa gider.**

Adım 7 / 11

```
1 copy_from_user(buf, p, n);
2 for (i = 0; i < n; i++) {
3     while (*durum_yaz != HAZIR)
4         ; /* meşgul bekle */
5     *veri_yaz = buf[i];
6 }
7 return_to_user();
```

DEĞİŞKENLER

i = 1

n = 2

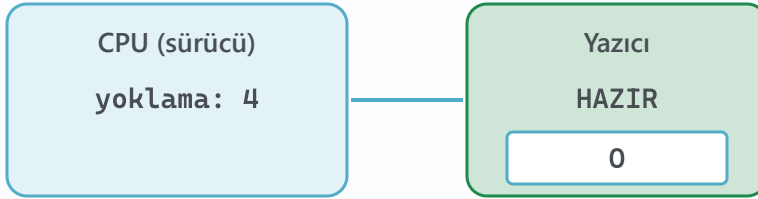
buf = "OS"

durum = MEŞGUL

ÇIKTI

0

Kod İzleme: Programlı G/Ç (Yoklama)



Yazıcı hazır: **HAZIR** → döngüden çıkılır.

Adım 8 / 11

```
1 copy_from_user(buf, p, n);
2 for (i = 0; i < n; i++) {
3     ► while (*durum_yaz != HAZIR)
4         ; /* meşgul bekle */
5     *veri_yaz = buf[i];
6 }
7 return_to_user();
```

DEĞİŞKENLER

i = 1

n = 2

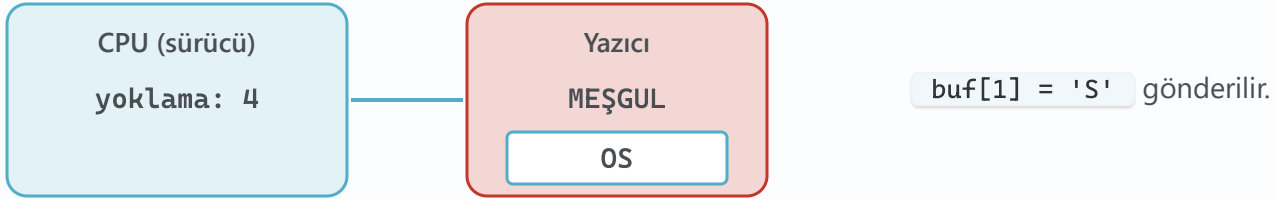
buf = "OS"

durum = HAZIR

ÇIKTI

0

Kod İzleme: Programlı G/Ç (Yoklama)



Adım 9 / 11

```
1 copy_from_user(buf, p, n);
2 for (i = 0; i < n; i++) {
3     while (*durum_yaz != HAZIR)
4         ; /* meşgul bekle */
5     ► *veri_yaz = buf[i];
6 }
7 return_to_user();
```

DEĞİŞKENLER

i = 1

n = 2

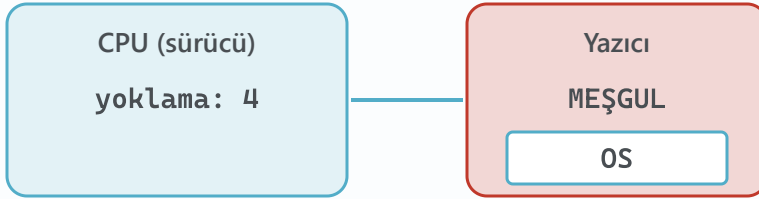
buf = "OS"

durum = MEŞGUL

ÇIKTI

OS

Kod İzleme: Programlı G/Ç (Yoklama)



$i = 2, 2 < 2$ yanlış → döngü biter.

Adım 10 / 11

```
1 copy_from_user(buf, p, n);  
▶ for (i = 0; i < n; i++) {  
3   while (*durum_yaz != HAZIR)  
4     ;           /* meşgul bekle */  
5   *veri_yaz = buf[i];  
6 }  
7 return_to_user();
```

DEĞİŞKENLER

$i = 2$

$n = 2$

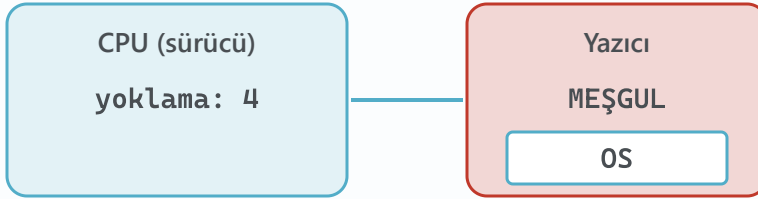
buf = "OS"

durum = MEŞGUL

ÇIKTI

OS

Kod İzleme: Programlı G/Ç (Yoklama)



Kullanıcı moduna dönülür. 2 karakter için 4 yoklama yapıldı; tüm süre boyunca CPU bu işe **kilitlendi**.

Adım 11 / 11

```
1 copy_from_user(buf, p, n);
2 for (i = 0; i < n; i++) {
3     while (*durum_yaz != HAZIR)
4         ; /* meşgul bekle */
5     *veri_yaz = buf[i];
6 }
▶ return_to_user();
```

DEĞİŞKENLER

i = 2

n = 2

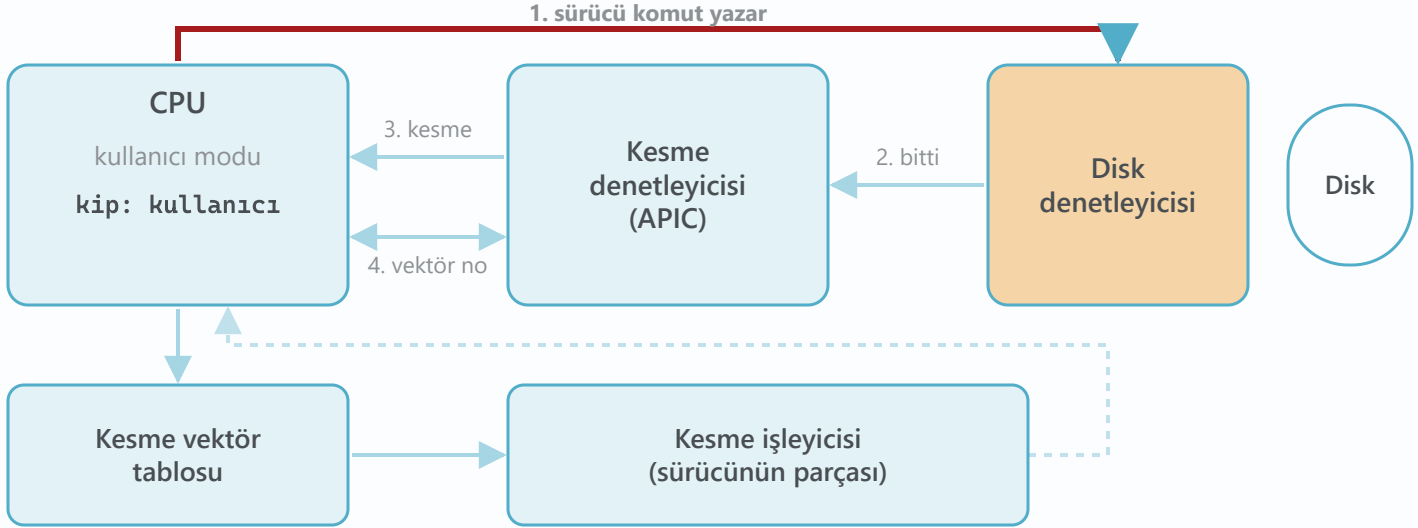
buf = "OS"

durum = MEŞGUL

ÇIKTI

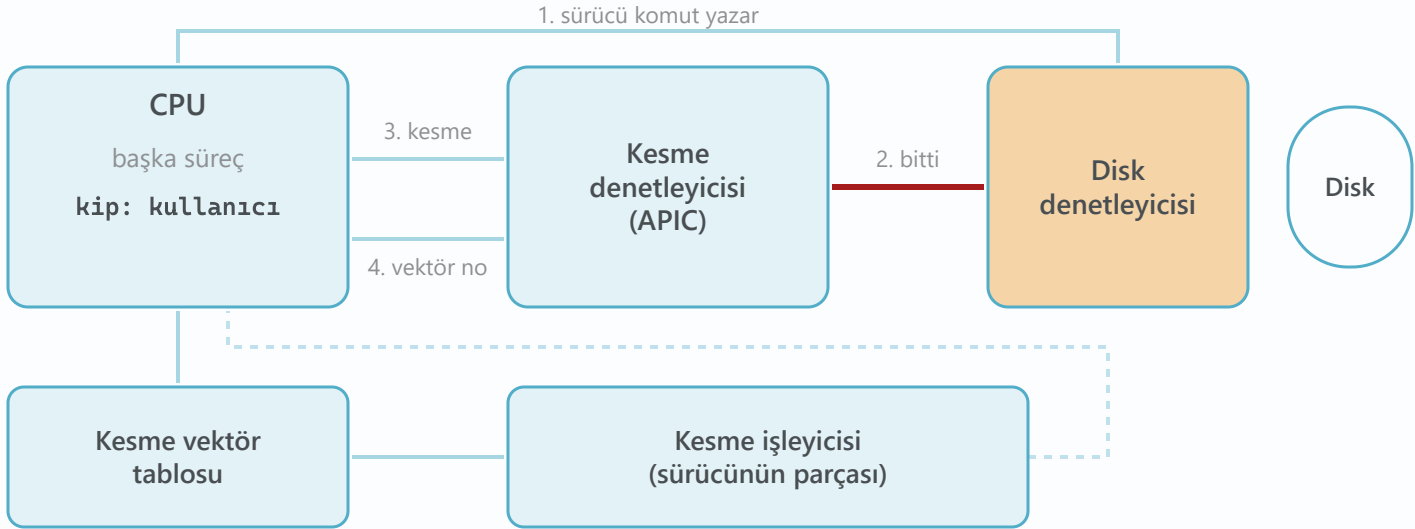
OS

G/Ç Aygıtları: Kesme



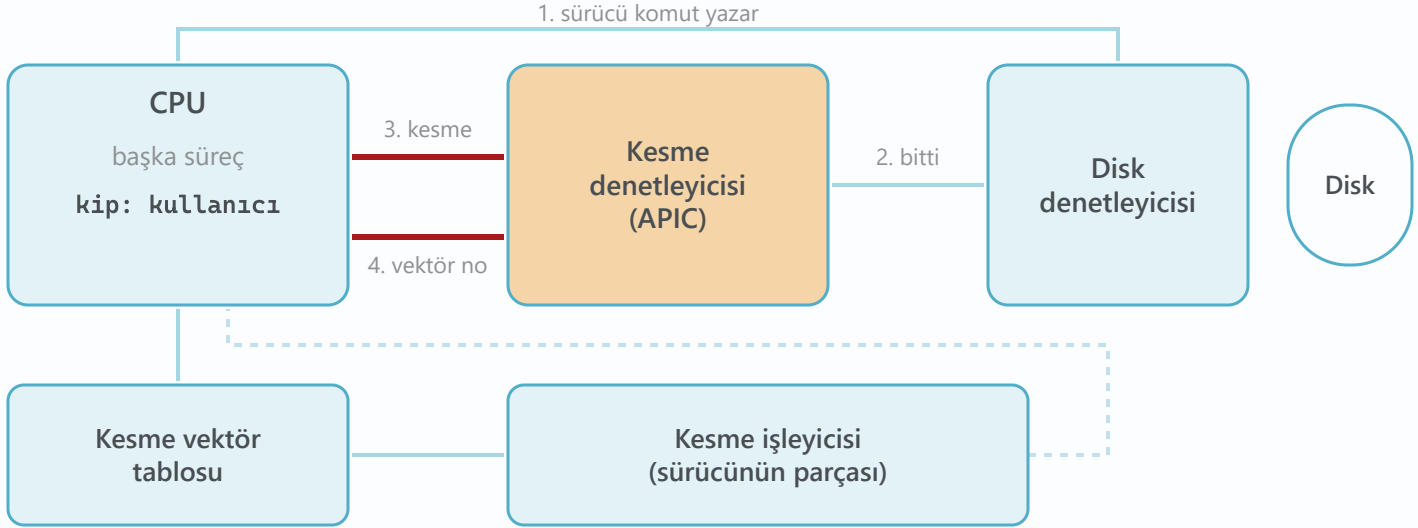
1. Sürücü, disk denetleyicisinin yazmaçlarına "sektör oku" komutunu yazar. Süreç bloke olur; CPU başka bir süreci çalıştırmaya döner.

G/Ç Aygıtları: Kesme



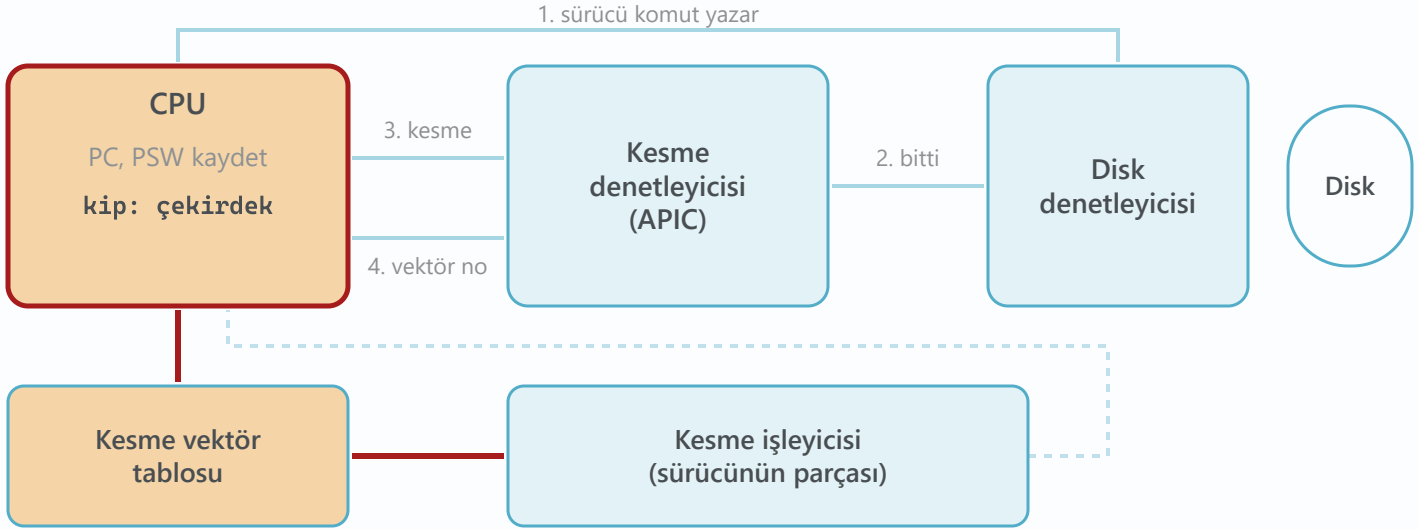
2. Okuma bitince disk denetleyicisi, veri yolu hatları üzerinden **kesme denetleyicisine** sinyal verir.

G/Ç Aygıtları: Kesme



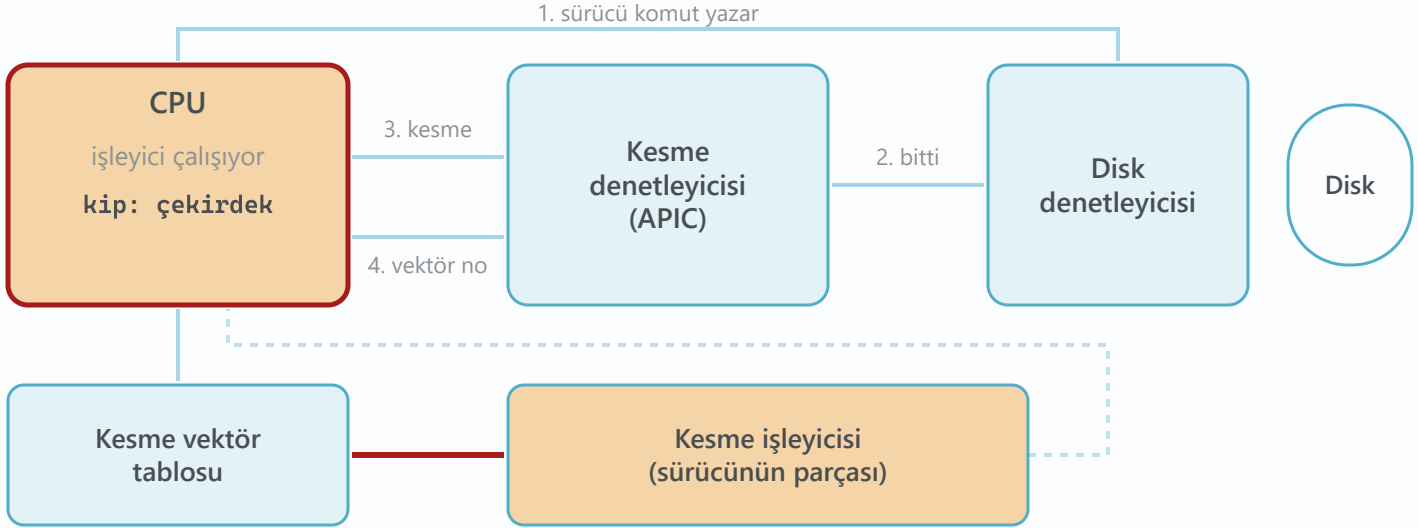
3–4. Kesme denetleyicisi (kesme kabul edilebiliyorsa) CPU’nun kesme hattını etkinleştirir ve hangi aygıt olduğunu belirten **vektör numarasını** verir.

G/Ç Aygıtları: Kesme



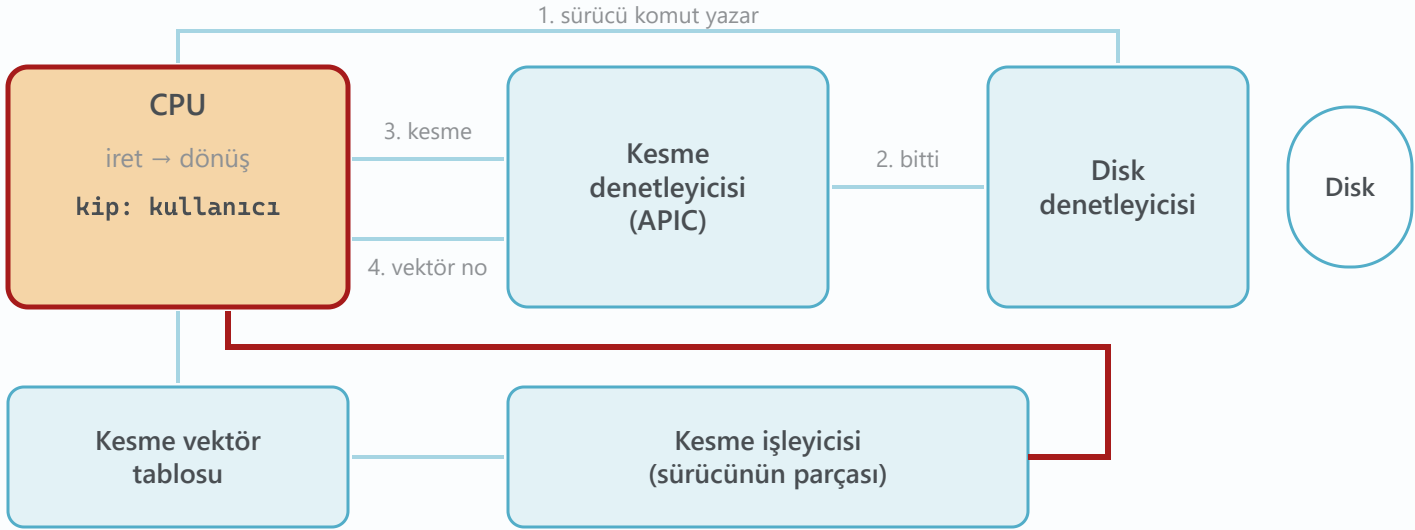
CPU o anki komutu bitirir, **PC ve PSW'yi** (çekirdek) yığına kaydeder, **çekirdek moduna** geçer; vektör tablosundan işleyicinin adresini alır.

G/Ç Aygıtları: Kesme



Kesme işleyicisi denetleyicinin durumunu okur, veriyi alır, bekleyen süreci "hazır" yapar ve kesme denetleyicisine **EOI** (işlem bitti) bildirir.

G/Ç Aygıtları: Kesme



iret : kaydedilen PSW ve PC geri yüklenir, kullanıcı moduna dönülür. Kesilen program (ya da çizelgeleyicinin seçtiği başka bir süreç) kaldığı yerden sürer.

Kod İzleme: Kesmeyle Yazdırma

```
1 /* sistem çağrısı yazdır(p, n) */  
2 ► copy_from_user(buf, p, n);  
3 enable_interrupts();  
4 while (*durum_yaz != HAZIR) ;  
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;  
6 scheduler(); /* A bloke olur */  
7 /* yazıcı kesmesi işleyicisi */  
8 if (kalan == 0) {  
9     unblock_user(); /* A hazır */  
10 } else {  
11     *veri_yaz = buf[i]; i++; kalan--;  
12 }  
13 acknowledge_interrupt();  
14 return_from_interrupt();
```

Süreç A
ÇALIŞIYOR

Süreç B
HAZIR

Yazıcı: HAZIR
(boş)

CPU

A

zaman →

☐ A, çekirdekte

☐ B, kullanıcı modu

☐ kesme işleyicisi

A, "ABC" yazdırmak için sistem çağrısı yaptı; çekirdek dizgeyi kullanıcı belleğinden çekirdek tamponuna kopyalar (**n = 3**).

Adım 1 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = ?

kalan = ?

ÇAĞRI YIĞINI

A: yazdır(p, 3) [çekirdek]

Kod İzleme: Kesmeyle Yazdırma

```
1 /* sistem çağrısı yazdir(p, n) */
2 copy_from_user(buf, p, n);
3 ► enable_interrupts();
4 while (*durum_yaz != HAZIR) ;
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 scheduler();          /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
8 if (kalan == 0) {
9     unblock_user();    /* A hazır */
10 } else {
11     *veri_yaz = buf[i]; i++; kalan--;
12 }
13 acknowledge_interrupt();
14 return_from_interrupt();
```

Süreç A
ÇALIŞIYORSüreç B
HAZIRYazıcı: HAZIR
(boş)

CPU

A

zaman →

☐ A, çekirdekte☐ B, kullanıcı modu☐ kesme işleyicisi

Yazıcı kesmelerine izin verilir: yazıcı her karakteri bitirince CPU'ya **kesme** gönderecek.

Adım 2 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = ?

kalan = ?

ÇAĞRI YIĞINI

A: yazdir(p, 3) [çekirdek]

Kod İzleme: Kesmeyle Yazdırma

```
1 /* sistem çağrısı yazdir(p, n) */
2 copy_from_user(buf, p, n);
3 enable_interrupts();
4 ► while (*durum_yaz != HAZIR) ;
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 scheduler();          /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
8 if (kalan == 0) {
9     unblock_user();    /* A hazır */
10 } else {
11     *veri_yaz = buf[i]; i++; kalan--;
12 }
13 acknowledge_interrupt();
14 return_from_interrupt();
```

Süreç A
ÇALIŞIYOR

Süreç B
HAZIR

Yazıcı: HAZIR
(boş)

CPU

A

zaman →

☐ A, çekirdekte

☐ B, kullanıcı modu

☐ kesme işleyicisi

Yazıcı zaten HAZIR; bu bekleme yalnızca ilk karakter için ve kısa sürer.

Adım 3 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = ?

kalan = ?

ÇAĞRI YIĞINI

A: yazdir(p, 3) [çekirdek]

Kod İzleme: Kesmeyle Yazdırma

```
1 /* sistem çağrısı yazdir(p, n) */
2 copy_from_user(buf, p, n);
3 enable_interrupts();
4 while (*durum_yaz != HAZIR) ;
5 ► *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 scheduler();          /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
8 if (kalan == 0) {
9     unblock_user();    /* A hazır */
10 } else {
11     *veri_yaz = buf[i]; i++; kalan--;
12 }
13 acknowledge_interrupt();
14 return_from_interrupt();
```

Süreç A
ÇALIŞIYOR

Süreç B
HAZIR

Yazıcı: MEŞGUL
"A"

CPU

A

zaman →

☐ A, çekirdekte

☐ B, kullanıcı modu

☐ kesme işleyicisi

İlk karakter **A** veri yazmacına yazılır. Kalan iş kesme işleyicisine bırakılır: **i = 1, kalan = 2**.

Adım 4 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = 1

kalan = 2

ÇAĞRI YIĞINI

A: yazdir(p, 3) [çekirdek]

Kod İzleme: Kesmeyle Yazdırma

```
1 /* sistem çağrısı yazdır(p, n) */
2 copy_from_user(buf, p, n);
3 enable_interrupts();
4 while (*durum_yaz != HAZIR) ;
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 ► scheduler(); /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
8 if (kalan == 0) {
9     unblock_user(); /* A hazır */
10 } else {
11     *veri_yaz = buf[i]; i++; kalan--;
12 }
13 acknowledge_interrupt();
14 return_from_interrupt();
```

Süreç A
BLOKESüreç B
ÇALIŞIYORYazıcı: MEŞGUL
"A"

CPU

A

B

zaman →

☐ A, çekirdekte☐ B, kullanıcı modu☐ kesme işleyicisi

scheduler() : A bloke olur, CPU B'ye verilir. Yazıcı karakteri basarken CPU **boşa dönmez**; yoklamadaki meşgul beklemenin tersi.

Adım 5 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = 1

kalan = 2

ÇAĞRI YIĞINI

B: kullanıcı kodu

Kod İzleme: Kesmeyle Yazdırma

```

1 /* sistem çağrısı yazdır(p, n) */
2 copy_from_user(buf, p, n);
3 enable_interrupts();
4 while (*durum_yaz != HAZIR) ;
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 scheduler(); /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
▶ if (kalan == 0) {
9   unblock_user(); /* A hazır */
10 } else {
11   *veri_yaz = buf[i]; i++; kalan--;
12 }
13 acknowledge_interrupt();
14 return_from_interrupt();

```



1. kesme: yazıcı "A" karakterini bitirdi. B, bulunduğu komutta kesilir; işleyici çekirdek modunda çalışır. kalan = 2 $\neq$ 0.

Adım 6 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = 1

kalan = 2

ÇAĞRI YIĞINI

yazıcı kesmesi [çekirdek]

B: kullanıcı kodu

Kod İzleme: Kesmeyle Yazdırma

```
1 /* sistem çağrısı yazdır(p, n) */
2 copy_from_user(buf, p, n);
3 enable_interrupts();
4 while (*durum_yaz != HAZIR) ;
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 scheduler();          /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
8 if (kalan == 0) {
9     unblock_user();    /* A hazır */
10 } else {
11     ▶ *veri_yaz = buf[i]; i++; kalan--;
12 }
13 acknowledge_interrupt();
14 return_from_interrupt();
```

Süreç A
BLOKE

Süreç B
ÇALIŞIYOR

Yazıcı: MEŞGUL
"AB"

CPU

A

B

kes

zaman →

☐ A, çekirdekte

☐ B, kullanıcı modu

☐ kesme işleyicisi

Sıradaki karakter **B** yazıcıya gönderilir: i = 2, kalan = 1.

Adım 7 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = 2

kalan = 1

ÇAĞRI YIĞINI

yazıcı kesmesi [çekirdek]

B: kullanıcı kodu

Kod İzleme: Kesmeyle Yazdırma

```
1 /* sistem çağrısı yazdır(p, n) */
2 copy_from_user(buf, p, n);
3 enable_interrupts();
4 while (*durum_yaz != HAZIR) ;
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 scheduler();          /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
8 if (kalan == 0) {
9     unblock_user();    /* A hazır */
10 } else {
11     *veri_yaz = buf[i]; i++; kalan--;
12 }
13 ▶ acknowledge_interrupt();
14 ▶ return_from_interrupt();
```

Süreç A
BLOKESüreç B
ÇALIŞIYORYazıcı: MEŞGUL
“AB”

CPU

A

B

kes

B

zaman →

☐ A, çekirdekte☐ B, kullanıcı modu☐ kesme işleyicisi

Kesme denetleyicisine onay verilir, kesmeden dönülür: B kaldığı yerden sürer.

Adım 8 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = 2

kalan = 1

ÇAĞRI YİĞİNİ

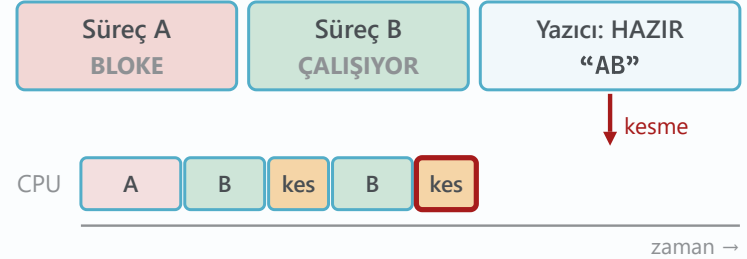
B: kullanıcı kodu

Kod İzleme: Kesmeyle Yazdırma

```

1 /* sistem çağrısı yazdır(p, n) */
2 copy_from_user(buf, p, n);
3 enable_interrupts();
4 while (*durum_yaz != HAZIR) ;
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 scheduler();          /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
▶ if (kalan == 0) {
9   unblock_user();    /* A hazır */
10 } else {
11   *veri_yaz = buf[i]; i++; kalan--;
12 }
13 acknowledge_interrupt();
14 return_from_interrupt();

```



2. kesme: yazıcı "B" karakterini bitirdi. B, bulunduğu komutta kesilir; işleyici çekirdek modunda çalışır. kalan = 1 ≠ 0.

Adım 9 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = 2

kalan = 1

ÇAĞRI YIĞINI

yazıcı kesmesi [çekirdek]

B: kullanıcı kodu

Kod İzleme: Kesmeyle Yazdırma

```
1 /* sistem çağrısı yazdır(p, n) */
2 copy_from_user(buf, p, n);
3 enable_interrupts();
4 while (*durum_yaz != HAZIR) ;
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 scheduler();          /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
8 if (kalan == 0) {
9     unblock_user();    /* A hazır */
10 } else {
11     ▶ *veri_yaz = buf[i]; i++; kalan--;
12 }
13 acknowledge_interrupt();
14 return_from_interrupt();
```

Süreç A
BLOKE

Süreç B
ÇALIŞIYOR

Yazıcı: MEŞGUL
"ABC"

CPU

A

B

kes

B

kes

zaman →

 A, çekirdekte

 B, kullanıcı modu

 kesme işleyicisi

Sıradaki karakter **C** yazıcıya gönderilir: i = 3, kalan = 0.

Adım 10 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = 3

kalan = 0

ÇAĞRI YIĞINI

yazıcı kesmesi [çekirdek]

B: kullanıcı kodu

Kod İzleme: Kesmeyle Yazdırma

```
1 /* sistem çağrısı yazdır(p, n) */
2 copy_from_user(buf, p, n);
3 enable_interrupts();
4 while (*durum_yaz != HAZIR) ;
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 scheduler();          /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
8 if (kalan == 0) {
9     unblock_user();    /* A hazır */
10 } else {
11     *veri_yaz = buf[i]; i++; kalan--;
12 }
13 ▶ acknowledge_interrupt();
14 ▶ return_from_interrupt();
```

Süreç A
BLOKE

Süreç B
ÇALIŞIYOR

Yazıcı: MEŞGUL
"ABC"

CPU

A

B

kes

B

kes

B

zaman →

☐ A, çekirdekte

☐ B, kullanıcı modu

☐ kesme işleyicisi

Kesme denetleyicisine onay verilir, kesmeden dönülür: B kaldığı yerden sürer.

Adım 11 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = 3

kalan = 0

ÇAĞRI YİĞİNİ

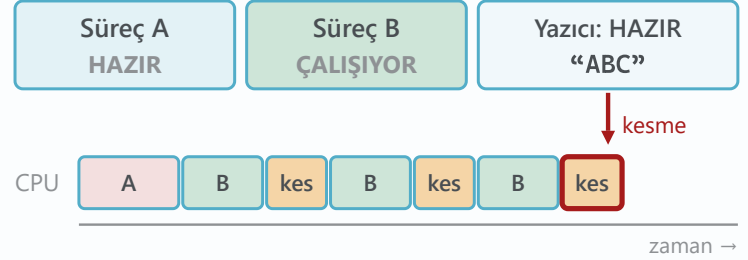
B: kullanıcı kodu

Kod İzleme: Kesmeyle Yazdırma

```

1 /* sistem çağrısı yazdır(p, n) */
2 copy_from_user(buf, p, n);
3 enable_interrupts();
4 while (*durum_yaz != HAZIR) ;
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 scheduler();          /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
8 ▶ if (kalan == 0) {
9   ▶ unblock_user();    /* A hazır */
10 } else {
11   *veri_yaz = buf[i]; i++; kalan--;
12 }
13 acknowledge_interrupt();
14 return_from_interrupt();

```



A, çekirdekte
 B, kullanıcı modu
 kesme işleyicisi

3. kesme: **kalan = 0**, tüm dizge basıldı. `unblock_user()`
 A'yı **hazır** yapar; çizelgeleyici onu daha sonra çalıştıracak.

Adım 12 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

i = 3

kalan = 0

ÇAĞRI YIĞINI

yazıcı kesmesi [çekirdek]

B: kullanıcı kodu

Kod İzleme: Kesmeyle Yazdırma

```

1 /* sistem çağrısı yazdır(p, n) */
2 copy_from_user(buf, p, n);
3 enable_interrupts();
4 while (*durum_yaz != HAZIR) ;
5 *veri_yaz = buf[0]; i = 1; kalan = n - 1;
6 scheduler();          /* A bloke olur */
7 /* yazıcı kesmesi işleyicisi */
8 if (kalan == 0) {
9     unblock_user();    /* A hazır */
10 } else {
11     *veri_yaz = buf[i]; i++; kalan--;
12 }
13 ▶ acknowledge_interrupt();
14 ▶ return_from_interrupt();

```

Süreç A
HAZIRSüreç B
ÇALIŞIYORYazıcı: HAZIR
"ABC"

CPU

A

B

kes

B

kes

B

kes

zaman →

A, çekirdekte

B, kullanıcı modu

kesme işleyicisi

Kesme onaylanır ve dönülür. 3 karakter için CPU yalnızca 3 kısa kesme işleyicisi çalıştırdı; arada hep B'nin işini yaptı. Bedeli: **karakter başına bir kesme**.

Adım 13 / 13

DEĞİŞKENLER

n = 3

buf = "ABC"

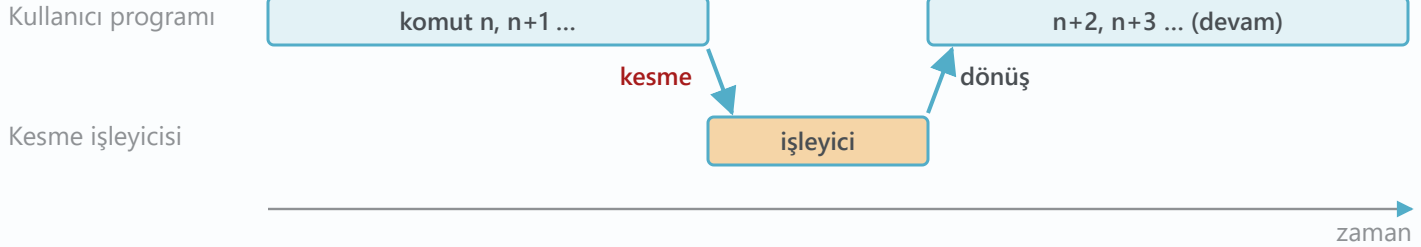
i = 3

kalan = 0

ÇAĞRI YIĞINI

B: kullanıcı kodu

Kesme Türleri



Donanım kesmesi

Dış olay: G/Ç tamamlandı, tuşa basıldı, ağ paketi geldi.

Zamanlayıcı kesmesi İS'ye düzenli olarak denetimi geri verir → kesmeli (preemptive) çoklu görev.

İstisna (trap)

Komutun kendisinden doğar: sıfıra bölme, geçersiz komut, **sayfa hatası**, kullanıcı modunda ayrıcalıklı komut.

Sistem çağırısı

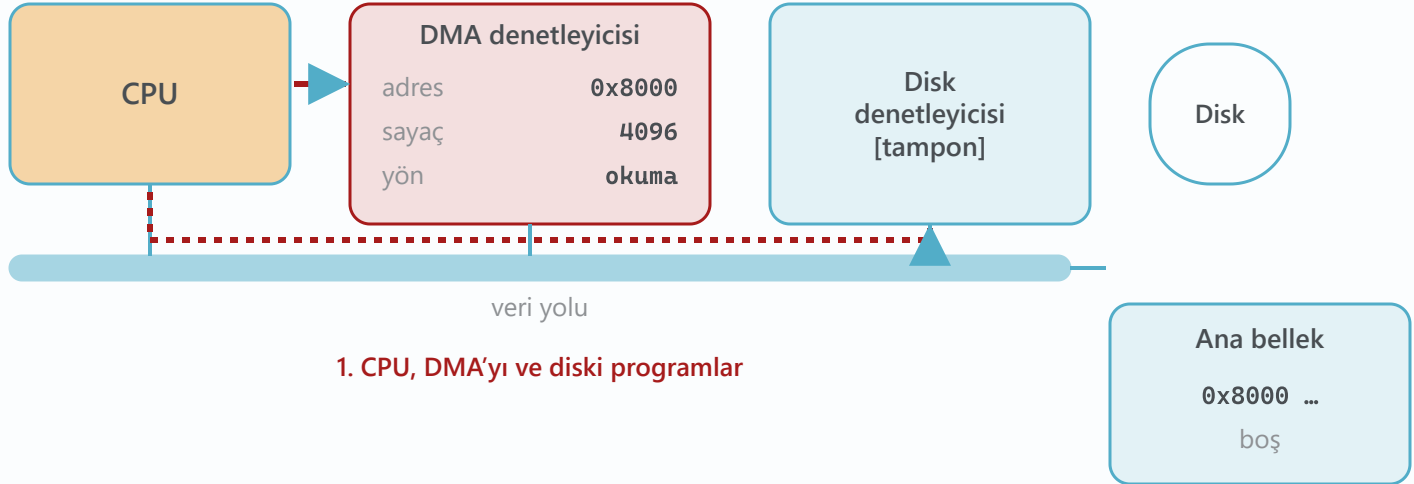
Bilerek üretilen trap:

`syscall`, `svc`, `ecall`.

Kullanıcı programının çekirdekten hizmet isteme yolu.

Kesmeler maskelenebilir ve önceliklendirilebilir; maskelenemeyen kesme (NMI) ciddi donanım hataları içindir.

G/Ç Aygıtları: DMA



1. CPU, DMA denetleyicisine bellek adresini (0x8000), bayt sayısını (4096) ve yönü yazar; diske "oku" komutu verir. Sonra **başka işe** döner.

G/Ç Aygıtları: DMA



2. Disk denetleyicisi bloğu diskten kendi tamponuna okur ve ECC ile doğrular.

G/Ç Aygıtları: DMA



3. DMA denetleyicisi, veri yolu üzerinden disk denetleyicisine aktarım isteği gönderir (hedef: 0x8000).

G/Ç Aygıtları: DMA



4. Disk denetleyicisi veriyi **doğrudan belleğe** yazar ve DMA'ya onay (ACK) gönderir; DMA adresi artırır, sayacı azaltır. Bu, 4096 bayt bitene kadar tekrarlanır (burada 64 baytlık bir parçadan sonra).

G/Ç Aygıtları: DMA



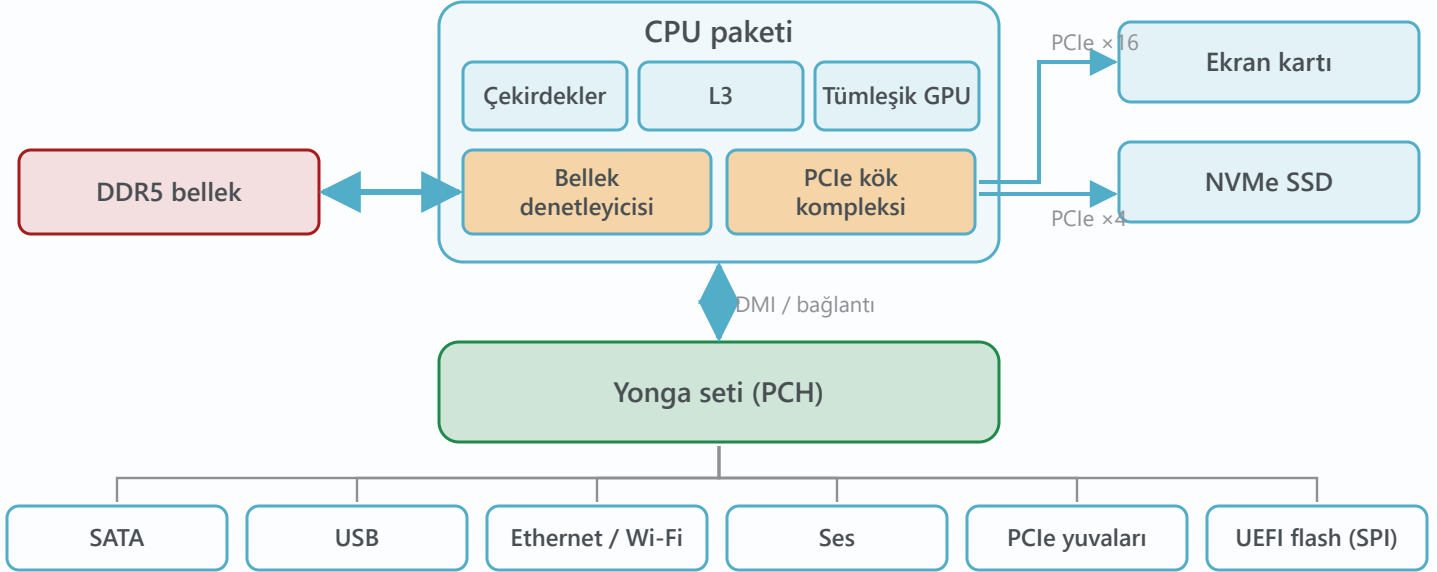
5. Sayaç 0 olunca DMA CPU'ya **tek bir kesme** gönderir. CPU yalnızca başta ve sonda devreye girdi. Bugün çoğu aygıt (NVMe, ağ kartı) kendi DMA motoruna sahiptir ve IOMMU erişimi denetler.

Veri Yolları

Veri yolu	Özellik (2026)	Tipik kullanım
PCIe	Seri, noktadan noktaya "hatlar" ($\times 1$, $\times 4$, $\times 16$). 5.0: 32 GT/s/hat (~ 4 GB/s), 6.0: 64 GT/s.	Ekran kartı ($\times 16$), NVMe SSD ($\times 4$), ağ kartı
USB	2.0: 480 Mb/s · 3.2: 5–20 Gb/s · USB4: 40/80 Gb/s. Tak-çalıştır, güç taşır.	Klavye, fare, bellek, kamera, harici ekran
SATA / SAS	SATA 6 Gb/s; SAS, SCSI'nin seri sürümü (sunucular).	HDD ve eski SSD'ler
DDR5	CPU'daki bellek denetleyicisine doğrudan bağlı bellek kanalları.	Ana bellek
Eski: ISA, PCI, IDE (PATA), paralel SCSI	Paralel, paylaşımlı ve yavaş hatlar; yerlerini PCIe ve SATA aldı.	1980–2000'lerin PC'leri

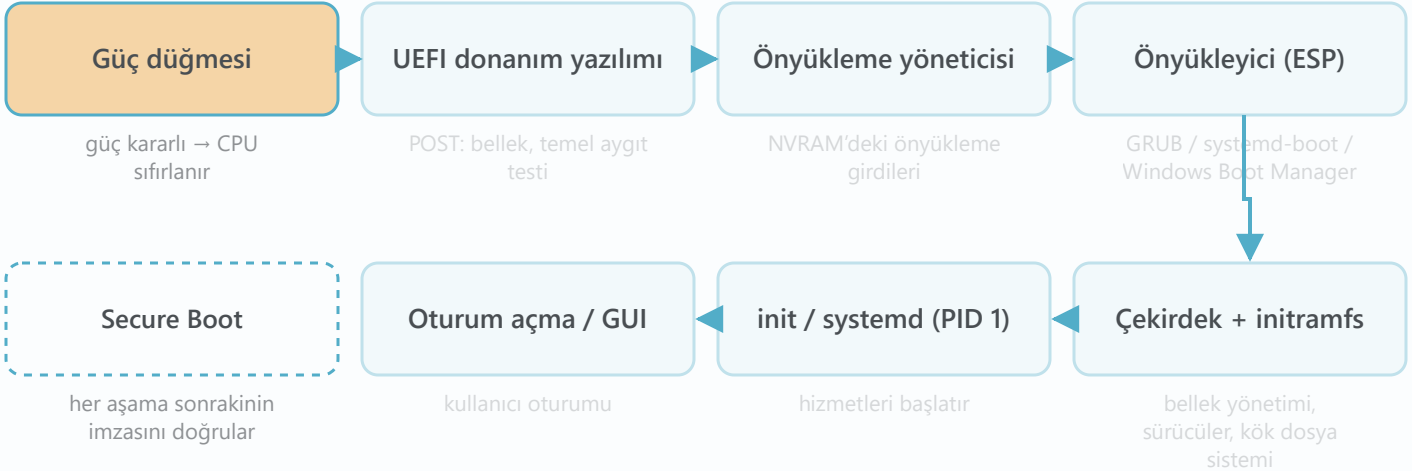
İS açılışta veri yollarını **tarar** (PCIe yapılandırma alanı, USB tanımlayıcıları), her aygıtın kimliğini okur ve uygun sürücüyü yükler: **tak-çalıştır** bu sayede mümkündür.

x86 Sistem Yapısı



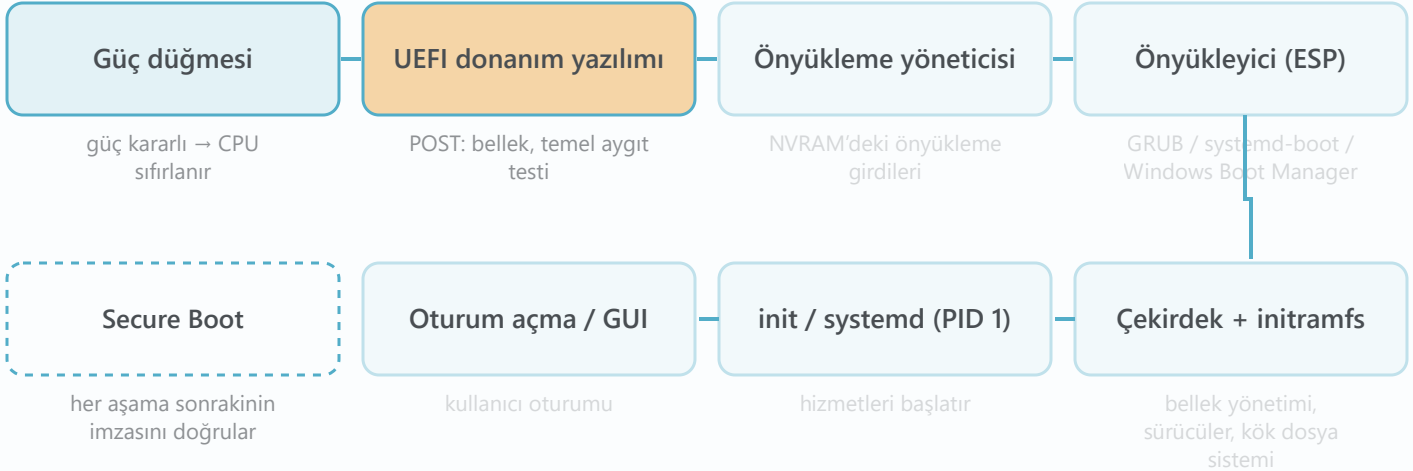
Günümüzde: Pentium dönemindeki kuzey köprüsü (northbridge), PCI, ISA ve IDE yolları artık yok. Bellek denetleyicisi CPU'nun içine taşındı (AMD 2003, Intel 2008); hızlı aygıtlar PCIe ile doğrudan CPU'ya, yavaşlar yonga setine bağlanır.

Bilgisayarın Ayağa Kalkması (UEFI)



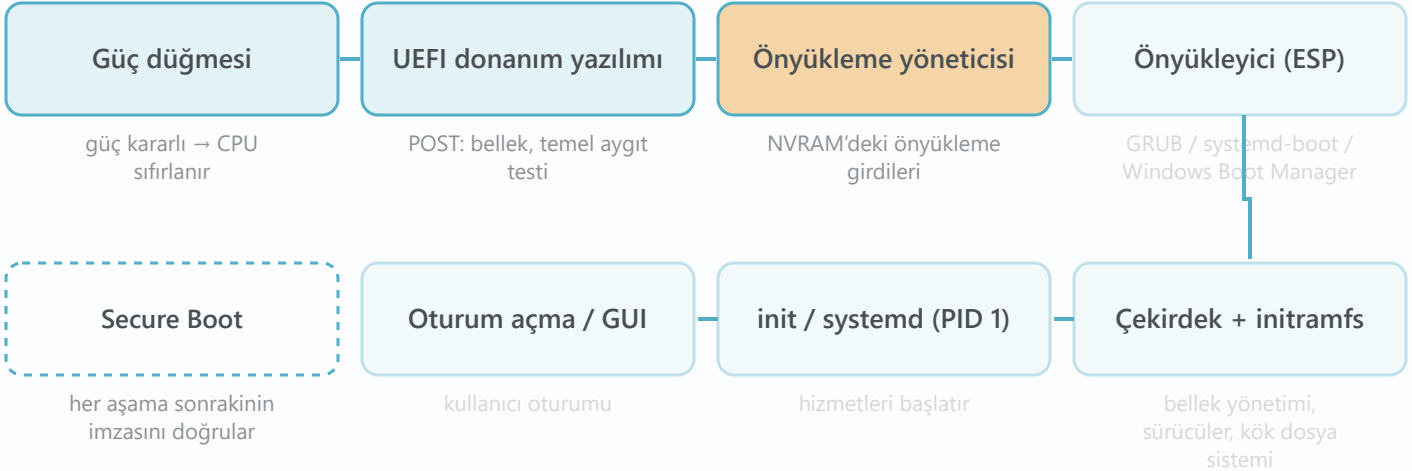
Güç verildiğinde CPU sıfırlanır ve donanımın belirlediği **sabit bir adresten** (flash'taki donanım yazılımı) komut getirmeye başlar.

Bilgisayarın Ayağa Kalkması (UEFI)



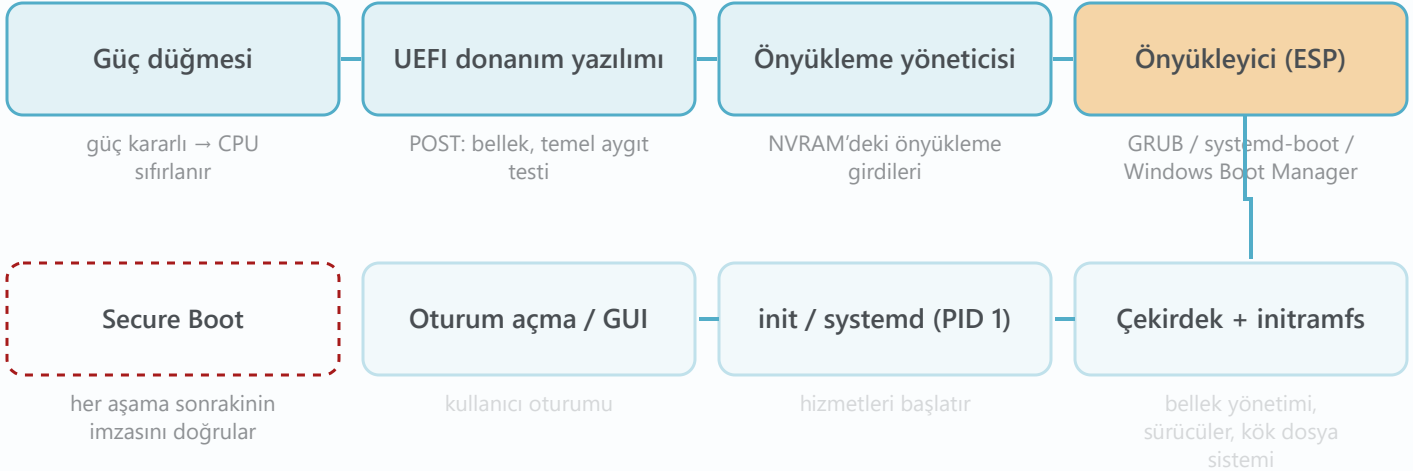
UEFI donanım yazılımı **POST** (açılış öz-sınaması) yapar: belleği ve temel aygıtları dener, PCIe/USB aygıtlarını bulur.

Bilgisayarın Ayağa Kalkması (UEFI)



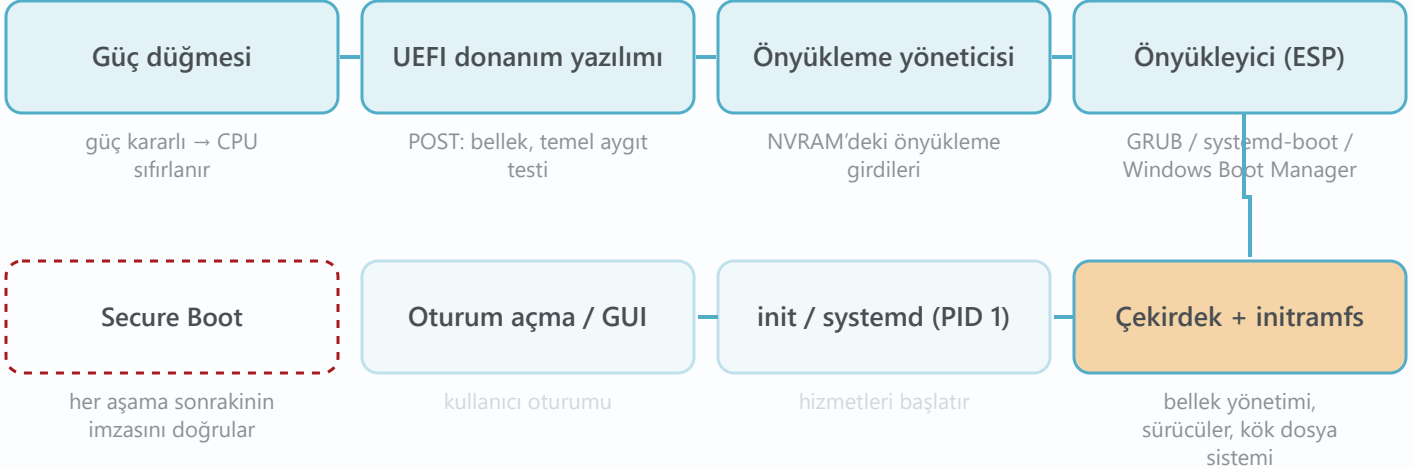
Önyükleme yöneticisi **NVRAM**'deki sırayı okur: hangi diskteki hangi **.efi** dosyası? (NVMe, USB, ağ/PXE).

Bilgisayarın Ayağa Kalkması (UEFI)



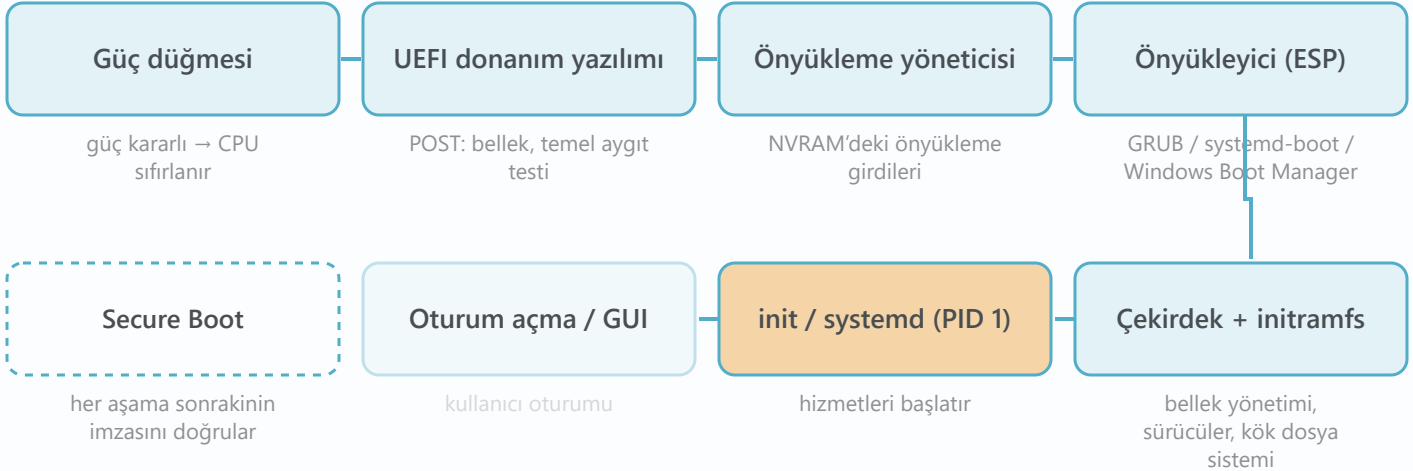
GPT diskteki **EFI Sistem Bölümü**nden (FAT32) önyükleyici yüklenir. Secure Boot açıksa imzası doğrulanır.

Bilgisayarın Ayağa Kalkması (UEFI)



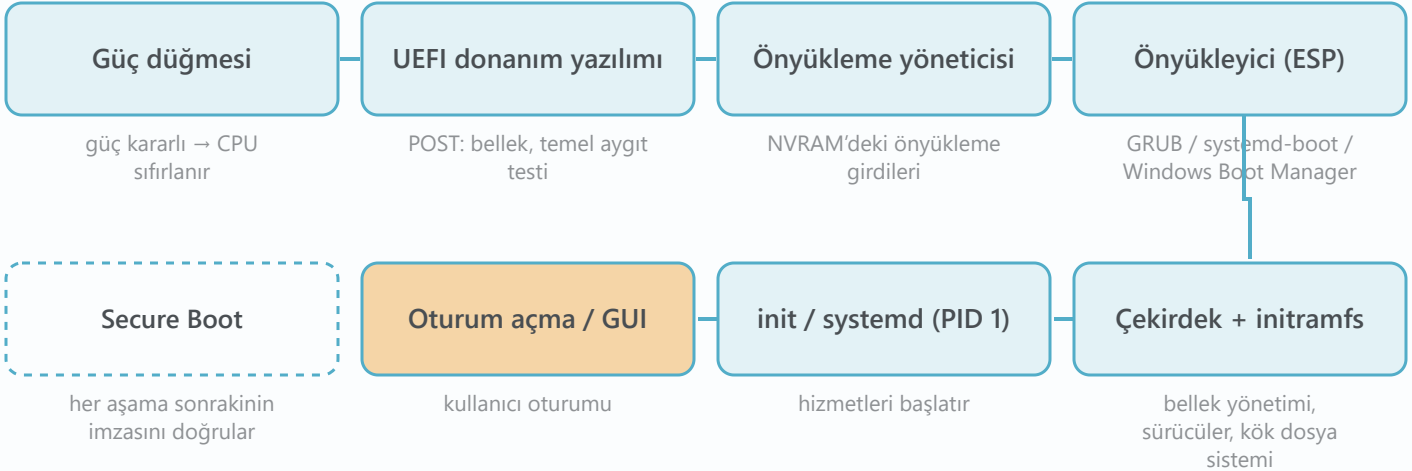
Önyükleyici **çekirdeği** ve **initramfs** 'i belleğe yükleyip denetimi verir. Çekirdek bellek yönetimini kurar, aygıtları tanır, sürücülerini yükler, kök dosya sistemini bağlar.

Bilgisayarın Ayağa Kalkması (UEFI)



Çekirdek ilk kullanıcı sürecini başlatır: **init / systemd** (PID 1). O da ağ, günlük, oturum yöneticisi gibi hizmetleri başlatır.

Bilgisayarın Ayağa Kalkması (UEFI)



Oturum açma ekranı ya da GUI gelir; sistem kullanıma hazır. Windows'ta karşılıkları: winload.efi → ntoskrnl.exe → smss.exe → winlogon.exe.

ÖNYÜKLEME

Eski BIOS ve UEFI

Eski (legacy) BIOS akışı

1. Bellek, klavye ve temel aygıtları dener (POST).
2. Önyükleme aygıtını seçer (eskiden disket, CD-ROM, disk).
3. Aygıtın ilk sektörünü (512 B **MBR**) belleğe okur ve çalıştırır.
4. MBR kodu bölüm tablosunda **etkin bölümü** bulur, onun ilk sektöründeki ikincil önyükleyiciyi okur.
5. Önyükleyici İS'yi o bölümden yükler.

	BIOS	UEFI
Ortaya çıkış	1981 (IBM PC)	2000'ler; bugün standart
Çalışma modu	16 bit gerçek mod	32/64 bit
Bölüm tablosu	MBR: 4 birincil bölüm, 2 TiB sınırı	GPT: 128+ bölüm, devasa diskler
Önyükleyici	Sektörlerdeki ham kod	ESP'de <code>.efi</code> dosyası
Güvenlik	Yok	Secure Boot (imza denetimi)

Birçok UEFI hâlâ eski BIOS'u taklit eden bir uyumluluk modu (CSM) sunar; yeni sistemlerde bu mod kaldırılmaktadır.

İşletim Sistemi Çeşitleri (1)

Anabilgisayar (mainframe)

Çok yüksek G/Ç kapasitesi, güvenilirlik ve erişilebilirlik; toplu iş, işlem (transaction) ve zaman paylaşımı. Bankalar, havayolları. *IBM z/OS, z/VM, Linux on Z, Unisys MCP, Fujitsu BS2000.*

Sunucu (server)

Ağ üzerinden çok kullanıcıya web, veritabanı, dosya, e-posta hizmeti. Veri merkezleri, bulut, sanallaştırma (KVM, Hyper-V), konteynerler. *Linux, Windows Server, FreeBSD.*

Çok işlemcili (multiprocessor)

Birden çok çekirdek/soket; işleri paralel yürütür, senkronizasyon ve NUMA ile uğraşır. Bugün **tüm genel amaçlı İS'ler** bu sınıftadır. *Linux, Windows, macOS.*

Kişisel (personal)

Tek kullanıcı, iyi GUI, çoklu görev; tarayıcı, ofis, oyun, dosya yönetimi. *Windows 11, macOS, Linux dağıtımları, ChromeOS.*

İşletim Sistemi Çeşitleri (2)

Mobil (handheld)

Dokunmatik arayüz, pil ve ısı yönetimi, sensörler, uygulama mağazası ve izin modeli. *Android, iOS/iPadOS, HarmonyOS.* (Windows Phone 2017'de sona erdi.)

Gömülü (embedded)

Araçlar, ev aletleri, ağ aygıtları, uçaklar; genellikle kullanıcı yazılım yüklemez, sistem flash'ta sabittir. *Gömülü Linux (Yocto), Zephyr, FreeRTOS, QNX, VxWorks.*

Algılayıcı düğümü (sensor node)

Birkaç on KB RAM, pil ile yıllarca çalışma, olay güdümlü yapı, düşük güçlü radyo (802.15.4, BLE). IoT ve M2M ağları. *Contiki-NG, RIOT, Zephyr; TinyOS* artık tarihsel.

Akıllı kart (smart card)

Çok kısıtlı CPU ve bellek; kimlik doğrulama, ödeme, kriptografi, güvenli saklama. Banka kartı, SIM/eSIM, kimlik kartı. *Java Card, MULTOS.*



Gerçek Zamanlı İşletim Sistemleri

Katı: fren denetimi

her 5 ms, 2 ms iş



Yumuşak: video karesi

son tarih ~16,7 ms



0 ms

5 ms

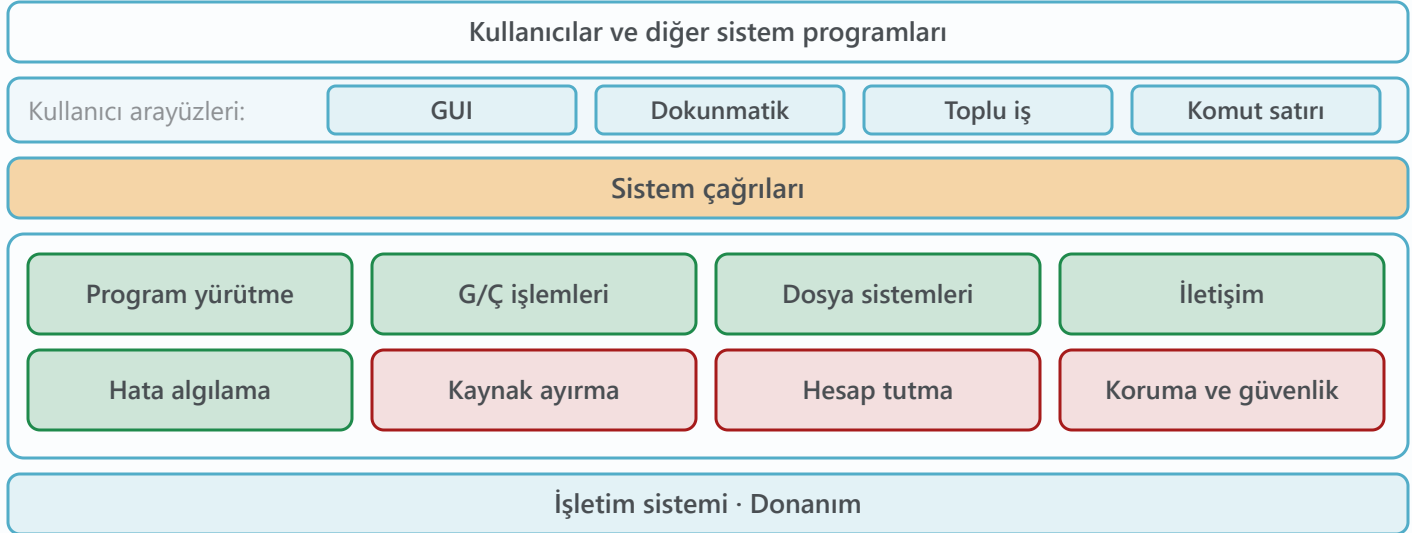
10 ms

15 ms

20 ms

- Önemli olan **hız değil, öngörülebilirlik**: iş belirli bir **son tarihten** (deadline) önce bitmelidir.
- **Katı (hard)**: son tarihi kaçırmak felakettir (fren, hava yastığı, uçuş kontrolü, endüstriyel robot).
- **Yumuşak (soft)**: ara sıra kaçırmak kabul edilir, kalite düşer (ses/video akışı, oyun).
- Örnekler: **VxWorks** (Mars gezginleri), **QNX** (araç sistemleri), FreeRTOS, Zephyr.
- Linux'un **PREEMPT_RT** yamaları 2024'te (6.12) ana çekirdeğe katıldı; eski *RTLinux* projesi sona erdi.
- Ses, video, hareket ve algılayıcı verisi işleyen; tren kontrolü, savunma gibi alanlarda kullanılır.

İşletim Sistemi Hizmetleri



Yeşil hizmetler doğrudan **kullanıcıya** yardım eder; **kırmızı** olanlar sistemin **verimli ve güvenli** çalışmasını sağlar (Silberschatz, Şekil 2.1).

Politika ve Mekanizma

- **Politika** (policy): *ne* yapılacak?
- **Mekanizma** (mechanism): *nasıl* yapılacak?
- İkisini **ayırmak** temel bir tasarım ilkesidir: politika kararları sonradan değiştiğinde mekanizmayı yeniden yazmak gerekmez → en yüksek esneklik.

Örnek: Linux'ta çizelgeleme sınıfı `sched_setscheduler()` ile, G/Ç çizelgeleyicisi `/sys` üzerinden 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 değiştirme altyapısı	Hangi sayfa dışarı atılacak?
cgroups	Hangi konteynere ne kadar CPU/bellek?

dilim = 4 ms



politika takılır

dilim = 10 ms

değiştirilebilir

Mekanizma (sabit çekirdek kodu)

zamanlayıcı kesmesi → bağlam değiştirme

İS tasarımı, bu dengeleri kurmak için mühendislik kadar yaratıcılık da gerektirir.

MS-DOS ve FreeBSD

(a) MS-DOS açılışta



(b) MS-DOS program çalışırken



(c) FreeBSD: çoklu süreç



MS-DOS: tek görevli, tek bellek alanı. Açılışta kabuk (COMMAND.COM) yüklenir; program onun geçici kısmının **üzerine yazılarak** yüklenir; bitince kabuk kendini diskten yeniden yükler. Çoklu görev ve koruma yoktur.

FreeBSD (UNIX türevi): çok görevli. Kullanıcı giriş yapınca kabuk başlar; kabuk `fork()` ile yeni süreç yaratır, `exec()` ile programı o sürece yükler; sonra programı bekler ya da (`&` ile) arka planda çalıştırıp komut almaya devam eder.

Kod İzleme: Kabuk Gibi `fork` + `exec` + `wait`

Ebeveyn · PID 1234

`main()` çalışıyor

Program (PID 1234) `fork()` 'u çağırır.
Çekirdek sürecin bir **kopyasını**
çıkarmak.

Adım 1 / 9

```
1 int main(void) {  
2     pid_t pid = fork();  
3     if (pid == 0) {           /* çocuk */  
4         execlp("ls", "ls", NULL);  
5         perror("exec");       /* yalnız hata olursa  
6     } else {                  /* ebeveyn */  
7         waitpid(pid, NULL, 0);  
8         printf("bitti\n");  
9     }  
10    return 0;  
11 }
```

DEĞİŞKENLER

pid = ?

ÇIKTI

Kod İzleme: Kabuk Gibi `fork` + `exec` + `wait`

Ebeveyn · PID 1234

fork() döndü: pid = 1235

Çocuk · PID 1235

fork() döndü: pid = 0

`fork()` bir kez çağrılır, **iki kez döner**:
ebeveyne çocuğun PID'si (1235), çocuğa
0. Hangisinin önce çalışacağına
çizelgeleyici karar verir.

Adım 2 / 9

```
1 int main(void) {  
2     pid_t pid = fork();  
3     if (pid == 0) {           /* çocuk */  
4         execlp("ls", "ls", NULL);  
5         perror("exec");      /* yalnız hata olursa  
6     } else {                 /* ebeveyn */  
7         waitpid(pid, NULL, 0);  
8         printf("bitti\n");  
9     }  
10    return 0;  
11 }
```

DEĞİŞKENLER

ebeveyn.pid = 1235

cocuk.pid = 0

ÇIKTI

Kod İzleme: Kabuk Gibi `fork` + `exec` + `wait`

Ebeveyn · PID 1234

pid ≠ 0 → else

Çocuk · PID 1235

hazır

Ebeveynde `pid == 0` yanlış → **else** dalı.

Adım 3 / 9

```
1 int main(void) {
2   pid_t pid = fork();
3   ► if (pid == 0) {           /* çocuk */
4       execlp("ls", "ls", NULL);
5       perror("exec");        /* yalnız hata olursa
6       */
7   } else {                   /* ebeveyn */
8       waitpid(pid, NULL, 0);
9       printf("bitti\n");
10  }
11  return 0;
12 }
```

DEĞİŞKENLER

ebeveyn.pid = 1235

cocuk.pid = 0

ÇIKTI

Kod İzleme: Kabuk Gibi `fork` + `exec` + `wait`

Ebeveyn · PID 1234

waitpid: bloke

Çocuk · PID 1235

hazır

Ebeveyn `waitpid()` ile çocuğun bitmesini bekler: **bloke** olur, CPU başkasına geçer.

Adım 4 / 9

```
1 int main(void) {
2     pid_t pid = fork();
3     if (pid == 0) {           /* çocuk */
4         execlp("ls", "ls", NULL);
5         perror("exec");      /* yalnız hata olursa
6     } else {                  /* ebeveyn */
7         waitpid(pid, NULL, 0);
8         printf("bitti\n");
9     }
10    return 0;
11 }
```

DEĞİŞKENLER

ebeveyn.pid = 1235

cocuk.pid = 0

ÇIKTI



Kod İzleme: Kabuk Gibi **fork** + **exec** + **wait**

Ebeveyn · PID 1234

bloke

Çocuk · PID 1235

pid == 0 → doğru

Çocukta `pid == 0` doğru → if dalı.

Adım 5 / 9

```
1 int main(void) {
2   pid_t pid = fork();
3   ► if (pid == 0) {           /* çocuk */
4       execlp("ls", "ls", NULL);
5       perror("exec");        /* yalnız hata olursa
6   } else {                    /* ebeveyn */
7       waitpid(pid, NULL, 0);
8       printf("bitti\n");
9   }
10  return 0;
11 }
```

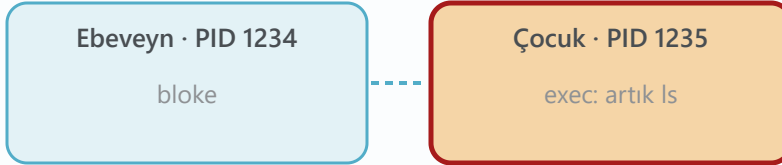
DEĞİŞKENLER

ebeveyn.pid = 1235

cocuk.pid = 0

ÇIKTI

Kod İzleme: Kabuk Gibi `fork` + `exec` + `wait`



`execvp()` çocuğun adres alanını **ls programıyla değiştirir** (PID aynı kalır). Başarılysa satır 5'e hiç dönülmez. `ls` dizini listeler.

Adım 6 / 9

```
1 int main(void) {
2     pid_t pid = fork();
3     if (pid == 0) {                /* çocuk */
4         ▶ execvp("ls", "ls", NULL);
5         perror("exec");            /* yalnız hata olursa
6     } else {                        /* ebeveyn */
7         waitpid(pid, NULL, 0);
8         printf("bitti\n");
9     }
10    return 0;
11 }
```

DEĞİŞKENLER

ebeveyn.pid = 1235

cocuk.pid = 0

ÇIKTI

a.out kod.c

Kod İzleme: Kabuk Gibi `fork` + `exec` + `wait`

Ebeveyn · PID 1234

uyandı

Çocuk · PID 1235

sonlandı

ls `exit(0)` ile biter; çocuk ebeveyn onu toplayana kadar kısa süre **zombi** kalır. `waitpid()` döner, ebeveyn uyanır.

Adım 7 / 9

```
1 int main(void) {
2     pid_t pid = fork();
3     if (pid == 0) {           /* çocuk */
4         execlp("ls", "ls", NULL);
5         perror("exec");      /* yalnız hata olursa
6     } else {                  /* ebeveyn */
7         ▶ waitpid(pid, NULL, 0);
8         printf("bitti\n");
9     }
10    return 0;
11 }
```

DEĞİŞKENLER

ebeveyn.pid = 1235

cocuk.pid = 0

ÇIKTI

a.out kod.c

Kod İzleme: Kabuk Gibi `fork` + `exec` + `wait`

Ebeveyn · PID 1234

`printf`

Çocuk · PID 1235

sonlandı

Ebeveyn "bitti" yazar. `waitpid` sayesinde çıktı sırası **garanti**: önce ls, sonra bitti.

Adım 8 / 9

```
1 int main(void) {
2     pid_t pid = fork();
3     if (pid == 0) {           /* çocuk */
4         execlp("ls", "ls", NULL);
5         perror("exec");      /* yalnız hata olursa
6     } else {                  /* ebeveyn */
7         waitpid(pid, NULL, 0);
8         printf("bitti\n");
9     }
10    return 0;
11 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1235`

ÇIKTI

```
a.out  kod.c
bitti
```

Kod İzleme: Kabuk Gibi **fork** + **exec** + **wait**

Ebeveyn · PID 1234

return 0

Çocuk · PID 1235

sonlandı

Ebeveyn de sonlanır. Kabuk her komutta tam olarak bu **fork** → **exec** → **wait** döngüsünü uygular.

Adım 9 / 9

```
1 int main(void) {
2     pid_t pid = fork();
3     if (pid == 0) {           /* çocuk */
4         execlp("ls", "ls", NULL);
5         perror("exec");      /* yalnız hata olursa
6     } else {                  /* ebeveyn */
7         waitpid(pid, NULL, 0);
8         printf("bitti\n");
9     }
10    return 0;
11 }
```

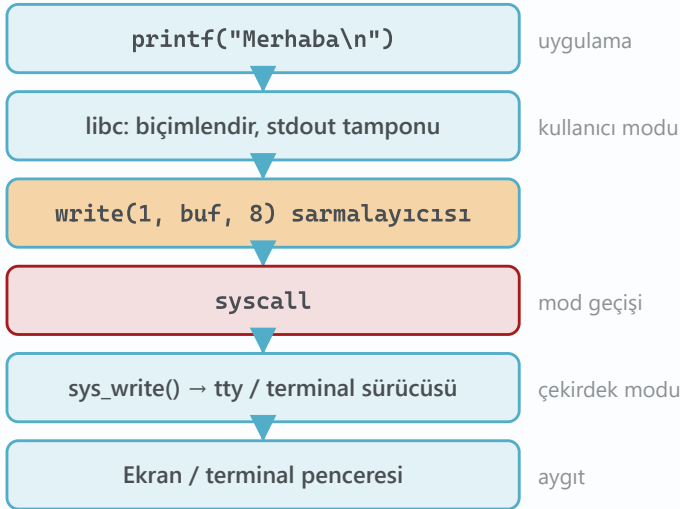
DEĞİŞKENLER

ebeveyn.pid = 1235

ÇIKTI

```
a.out  kod.c
bitti
```

Standart C Kütüphanesi

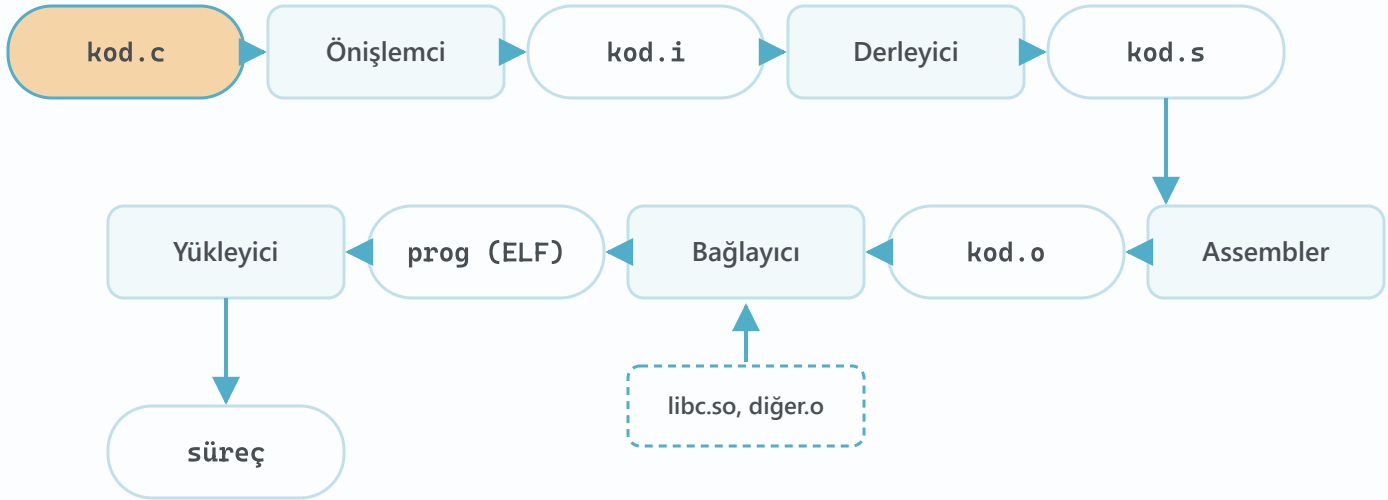


Başlık	Örnek işlevler
<code>stdio.h</code>	<code>printf</code> , <code>fopen</code> , <code>fread</code> , <code>fclose</code>
<code>stdlib.h</code>	<code>malloc</code> , <code>free</code> , <code>exit</code> , <code>qsort</code>
<code>string.h</code>	<code>strlen</code> , <code>strcpy</code> , <code>memcpy</code>
<code>math.h</code> · <code>time.h</code>	<code>sqrt</code> , <code>sin</code> · <code>time</code> , <code>clock</code>
<code>unistd.h</code> (POSIX)	<code>fork</code> , <code>read</code> , <code>write</code> , <code>close</code>

C kütüphanesi (glibc, musl; Windows'ta UCRT) kullanıcı modunda çalışır. Bir kısmı tamamen kütüphane içindedir (`strlen`), bir kısmı sistem çağrılarını sarmalar (`write`).

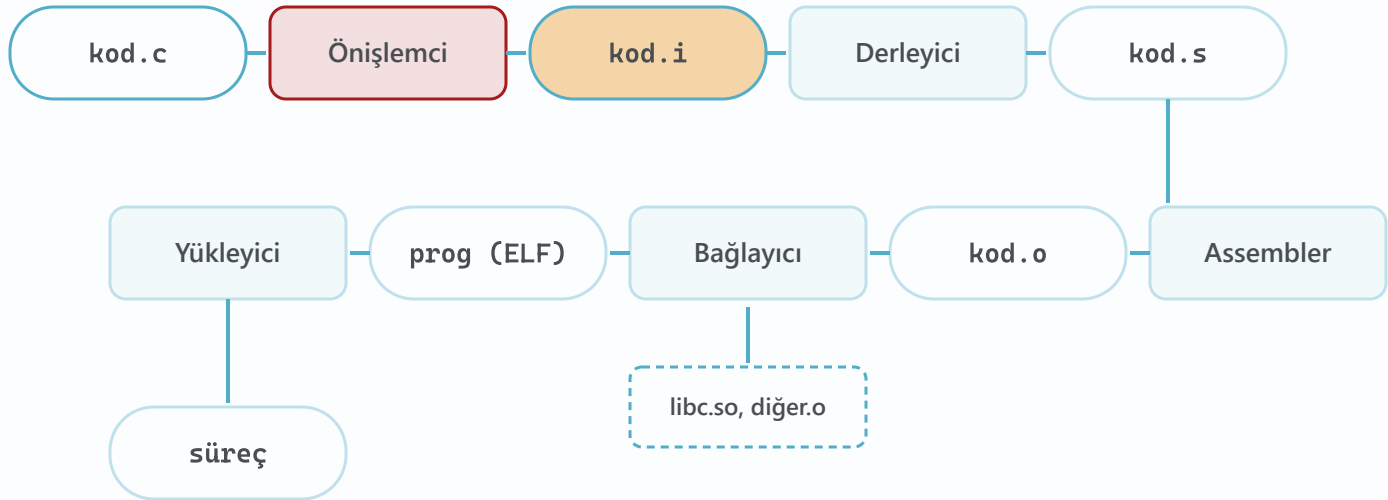
`"Merhaba\\n"` 8 bayttır.

Çalıştırılabilir Dosya Oluşturma



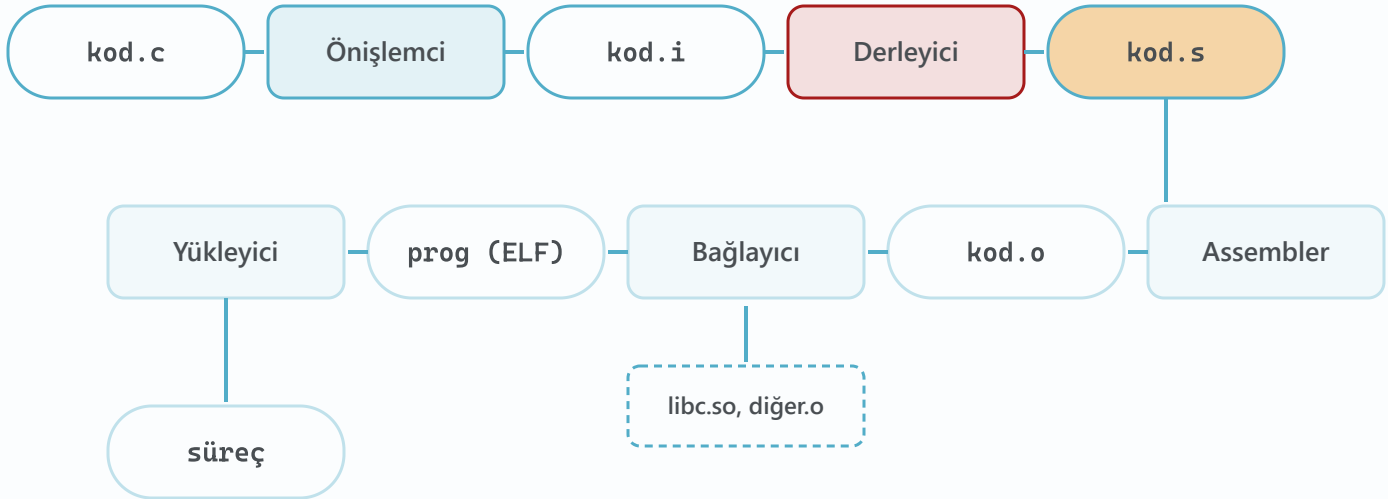
Kaynak dosya `kod.c` : insanın okuduğu C kodu.

Çalıştırılabilir Dosya Oluşturma



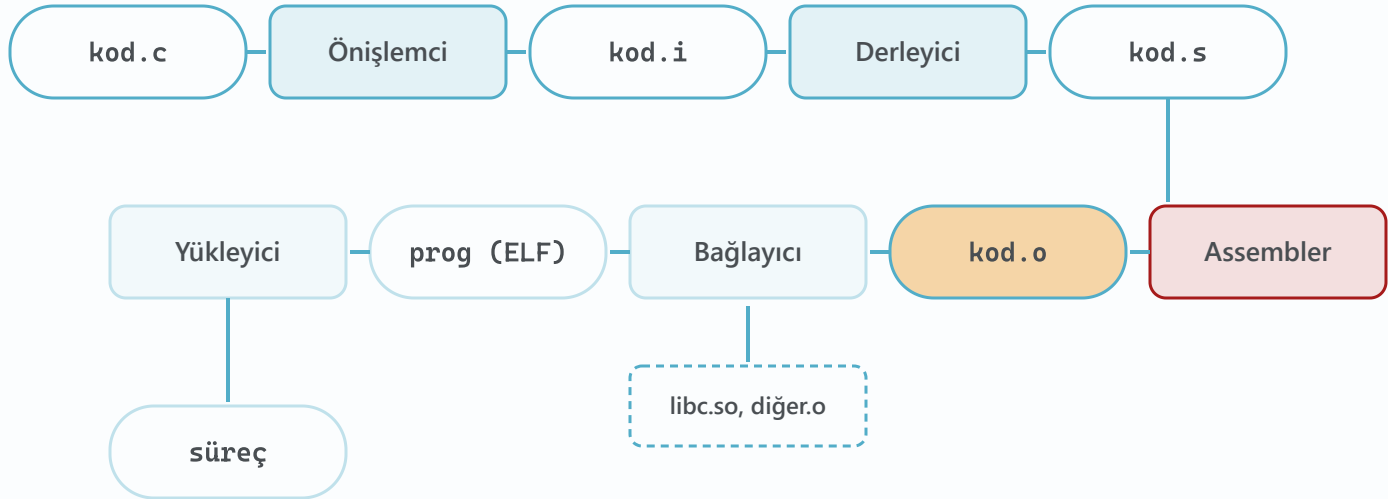
Önişlemci (`gcc -E`): `#include` ile başlık dosyalarını ekler, makroları genişletir, `#ifdef` ile koşullu derlemeyi çözer → `kod.i` .

Çalıştırılabilir Dosya Oluşturma



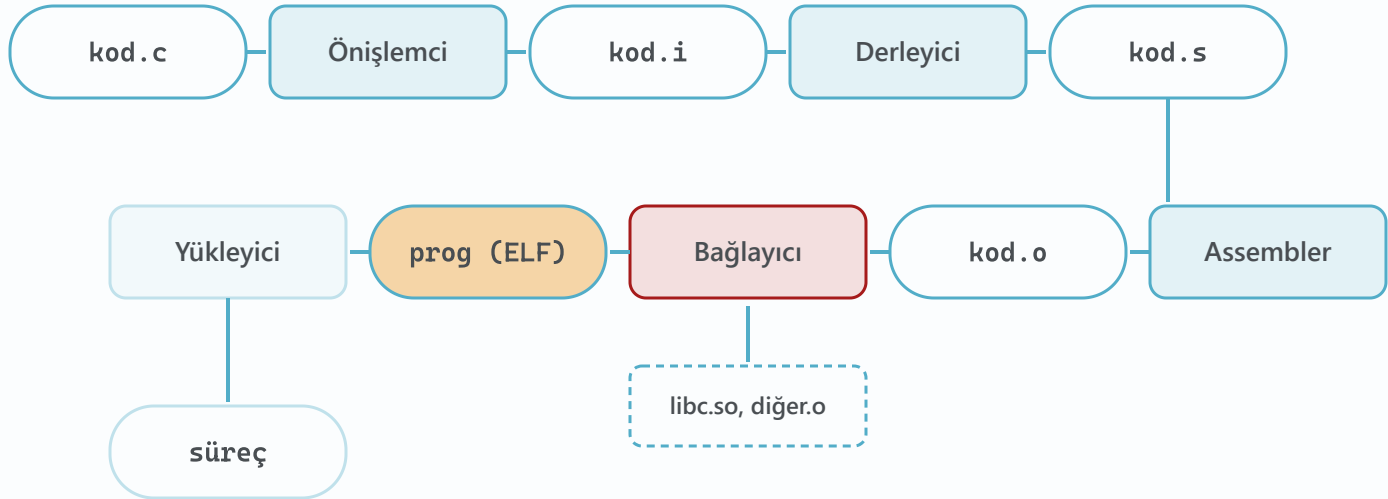
Derleyici (`gcc -S`): C kodunu hedef işlemcinin **assembly** diline çevirir, eniyileme yapar
→ `kod.s` .

Çalıştırılabilir Dosya Oluşturma



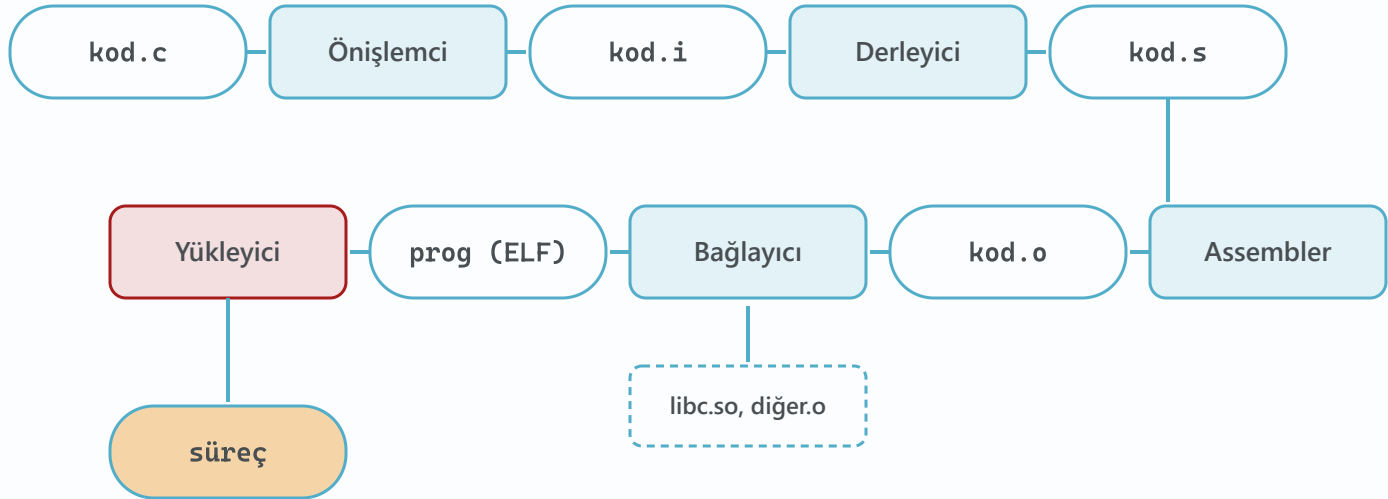
Assembler (`gcc -c`): assembly'yi makine koduna çevirir → **nesne dosyası** `kod.o` . Dış işlevlerin (`printf`) adresleri henüz belli değildir.

Çalıştırılabilir Dosya Oluşturma



Bağlayıcı (`gcc kod.o diğer.o -o prog`): nesne dosyalarını ve kütüphaneleri birleştirir, adresleri çözer → **çalıştırılabilir** dosya (Linux: ELF, Windows: PE/.exe).

Çalıştırılabilir Dosya Oluşturma



`./prog` : kabuk fork + exec yapar; çekirdeğin **yükleyicisi** ELF'i belleğe eşler, dinamik bağlayıcı `libc.so`'yu bağlar ve program bir **süreç** olarak çalışır.

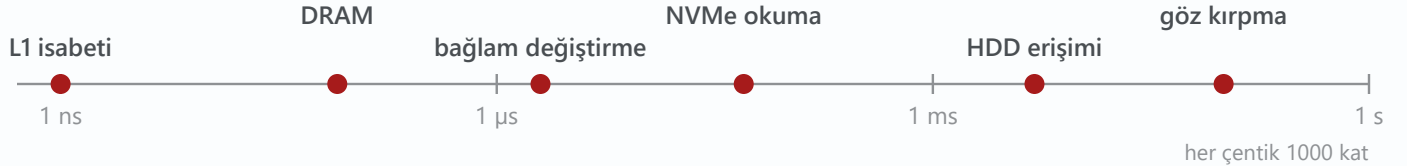
TEMEL KAVRAMLAR

Metrik Birimler

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

İkili (IEC)	Değer
1 KiB (kibibayt)	$2^{10} = 1024$ B
1 MiB	2^{20} B
1 GiB	2^{30} B
1 TiB	2^{40} B

Bellek genelde ikili (16 GiB RAM), disk ve ağ ondalık ölçülür: 1 TB = 10^{12} B $\approx$ 0,91 TiB; 1 Gb/s = 10^9 bit/s. Bayt **B**, bit **b**.



ÖZET

Kontrol Soruları

1. Kullanıcı programı ayrıcalıklı bir komut çalıştırırsa ne olur?
2. Çoklu programlama CPU kullanımını neden artırır? Gerçek paralellikten farkı nedir?
3. 4 KB'lık bir disk bloğunu belleğe almak için yoklama, kesme ve DMA'dan hangisi uygundur? Neden?
4. 7200 dev/dk dönen bir diskte ortalama dönme gecikmesi kaç ms'dir?
5. 4 kümeli, 64 B satırlı doğrudan eşlemeli boş önbellekte 0, 128, 256, 0 adreslerine erişiliyor. Kaç isabet olur?
6. `fork()` kaç kez döner, çocukta ne döndürür? Politika ile mekanizmayı bir örnekle ayırın.

► Cevapları göster

ÖZET

Kaynaklar

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı, Pearson, 2014; 5. baskı, 2023.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), sürüm 1.10, pages.cs.wisc.edu/~remzi/OSTEP.
- Linux kılavuz sayfaları: `syscall(2)` , `fork(2)` , `execve(2)` , `proc(5)` ; UEFI Forum, *UEFI Specification*.

Son

Bölüm 1: Giriş

Sonraki bölüm: İşletim Sistemi Yapıları · [Terim sözlüğü](#)