

Bölüm 15: İşletim Sistemi Tasarımı

İşletim Sistemleri

Tasarım neden zor?

İşletim sisteminin hedefleri

Arayüz ilkeleri ve paradigmlar

Sistem çağırısı arayüzü

Katmanlı, exokernel, mikro çekirdek

Mekanizma, politika, diklik

Adlandırma ve bağlama zamanı

Gerçekleştirme stratejileri

Yararlı teknikler

Performans: takas, önbellek, ipucu

Proje yönetimi ve Brooks yasası

Test ve hata ayıklama

Eğilimler: bulut, Rust, eBPF

Öğrenme Çıktıları

Zorluklar ve hedefler

İşletim sistemi tasarımını zorlaştıran etkenleri ve bir işletim sisteminin dört temel hedefini açıklamak.

Arayüz tasarımı

Basitlik, bütünlük ve verimlilik ilkelerini; yürütme ve veri paradigmalarını sistem çağrısı örnekleriyle tartışmak.

Sistem yapısı

Katmanlı, exokernel, mikro çekirdek tabanlı ve genişletilebilir yapıları ödüneşimleriyle karşılaştırmak.

Tasarım ilkeleri

Mekanizma/politika ayrımını, dikliğı, iki düzeyli adlandırmayı, bağlama zamanını ve statik/dinamik yapıları uygulamak.

Teknikler ve performans

Dolaylılık, yeniden girişlilik, uzay-zaman ödüneşimi, önbellek, ipucu ve yaygın durumu hızlandırma tekniklerini örneklerle kullanmak.

Proje, test ve eğilimler

Brooks yasasını yorumlamak; çekirdek test ve hata ayıklama araçlarını seçmek; sanallaştırma, çok çekirdek, Rust, eBPF ve unikernel eğilimlerini değerlendirmek.

GİRİŞ

İşletim Sistemi Tasarlamak Neden Zor?

1 • Devasa boyut

Linux $\approx$ 40 milyon satır; kimse bütünü kavrayamaz.

2 • Eş zamanlılık

Aygıtlar, kesmeler, iş parçacıkları aynı anda: yarış, kilitlenme.

3 • Düşmanca kullanıcılar

Bazıları sistemi bozmak, veri çalmak ister; her girdi şüpheli.

4 • Yalıtıma rağmen paylaşım

Kullanıcılar ayrılmalı, ama seçtikleriyle bilgi paylaşabilmeli.

5 • Uzun ömür

20-30 yıl yaşar; gelecekteki donanım ve uygulamalar öngörülmesi.

6 • Genellik

Nasıl kullanılacağı bilinmez: telefondan süper bilgisayara.

7 • Taşınabilirlik

Farklı işlemci ve platformlar: x86-64, ARM64, RISC-V...

8 • Geriye dönük uyumluluk

Eski programlar çalışmalı; yeni fikirler eski kısıtlarla sınırlanır.

Tanenbaum'ın sekiz etkeni. Uzun bakım-onarım ihtiyacı 5. ve 8. maddelerin doğal sonucudur.

İşletim Sisteminin Hedefleri

Tarayıcı

Derleyici

Veritabanı

Soyutlamalar

süreç

iş parçacığı

adres alanı

dosya

socket

İlkel işlemler (sistem çağrıları)

fork · execve · mmap · open · read · write · send · recv · futex

Donanım yönetimi

CPU

bellek

SSD/disk

ağ kartı

GPU

- 1. Soyutlamalar tanımlamak** (abstractions): süreç, adres alanı, dosya, socket gibi kavramlar donanımın karmaşıklığını gizler. Tasarımın en zor kısmı doğru soyutlamayı bulmaktır.

İşletim Sisteminin Hedefleri

Tarayıcı

Derleyici

Veritabanı

Soyutlamalar

süreç

iş parçacığı

adres alanı

dosya

soket

İlkel işlemler (sistem çağrıları)

fork · execve · mmap · open · read · write · send · recv · futex

Donanım yönetimi

CPU

bellek

SSD/disk

ağ kartı

GPU

2. İlkel işlemler sağlamak (primitives): bu soyutlamalar üzerinde çalışan işlemler, yani sistem çağrıları. Örn. dosya için `open/read/write/close`.

İşletim Sisteminin Hedefleri

Tarayıcı

Derleyici

Veritabanı

her süreç kendi adres alanında: biri çökerse diğerleri etkilenmez

Soyutlamalar

süreç

iş parçacığı

adres alanı

dosya

soket

İlkel işlemler (sistem çağrıları)

fork · execve · mmap · open · read · write · send · recv · futex

Donanım yönetimi

CPU

bellek

SSD/disk

ağ kartı

GPU

3. Yalıtım sağlamak (isolation): kullanıcılar ve süreçler birbirine karışmamalı; ama denetimli paylaşım (dosya izinleri, paylaşımlı bellek) mümkün olmalı.

İşletim Sisteminin Hedefleri

Tarayıcı

Derleyici

Veritabanı

Soyutlamalar

süreç

iş parçacığı

adres alanı

dosya

soket

İlkel işlemler (sistem çağrıları)

fork · execve · mmap · open · read · write · send · recv · futex

Donanım yönetimi

CPU

bellek

SSD/disk

ağ kartı

GPU

4. Donanımı yönetmek (hardware): kesmeler, zamanlayıcılar, MMU, aygıt denetleyicileri; düşük seviyeli işler çoğunlukla bu katmanda yapılır.

Arayüz Tasarımı İçin Yol Gösterici İlkeler

1 • Basitlik (simplicity)

"Mükemmelliğe, eklenecek bir şey kalmadığında değil, **çıkartılacak bir şey kalmadığında** ulaşılır." (A. de Saint-Exupéry). Az sayıda, iyi düşünülmüş kavram öğrenmesi, gerçekleştirilmesi ve doğrulaması kolaydır.

2 • Bütünlük (completeness)

"Her şey olabildiğince basit olmalı, ama **daha basit değil.**" (Einstein'a atfedilir). Arayüz, kullanıcıların ihtiyaç duyduğu her şeyi yapabilmelidir. Az mekanizma ile çok iş: MINIX 3 çekirdeği iletişim için yalnızca birkaç ilkel (send, receive, sendrec, notify) sunar.

3 • Verimlilik (efficiency)

Bir özellik ya da sistem çağrısı verimli gerçekleştirilemiyorsa muhtemelen **sahip olmaya değmez**. Programcı bir çağrının yaklaşık maliyetini sezebilmelidir: `lseek` ucuz, `read` pahalı olabilir.

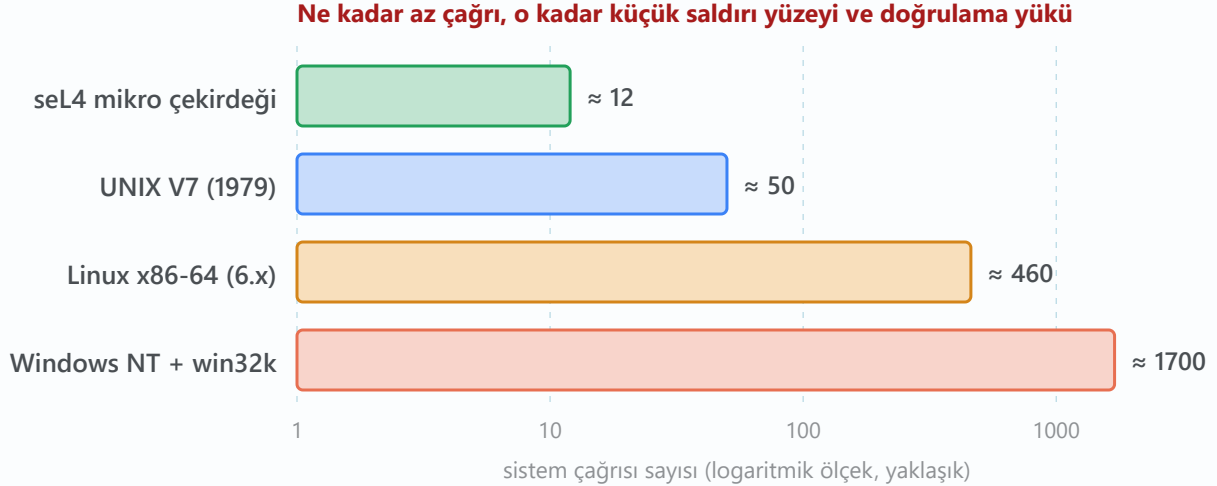
Pratik kural: **her sistem çağrısı tek bir işi iyi yapmalı**. Özellik eklemek kolay, kaldırmak neredeyse imkânsızdır (geriye dönük uyumluluk).

uygulama

kütüphane: printf, fopen, fgets, qsort, getaddrinfo ... (çok, kolay, değişebilir)

çekirdek: open, read, write, close, mmap ... (az, genel; değiştirmek uyumluluğu bozar)

“Küçük Güzeldir”



- **Küçük güzeldir:** az sayıda genel çağrı, çok sayıda özel çağrıdan iyidir.
- seL4'ün çekirdeği yaklaşık 10 bin satırdır ve **biçimsel olarak doğrulanmıştır**; milyonlarca satırlık bir çekirdekte bu mümkün değil.
- Linux'ta çağrı sayısı her yıl birkaç tane artar (ör. `io_uring_*`, `landlock_*`, `mseal`).
- Windows'ta belgelenmiş arayüz Win32 API'dir; çekirdek çağrıları sürümden sürüme değişir.

Kullanıcı Arayüzü Paradigmaları

Komut satırı (CLI)

Kabuk, metin komutları, borular. Otomasyon ve uzaktan yönetim için hâlâ vazgeçilmez.

WIMP / GUI

Pencere, simge, menü, işaretçi (Windows, Icons, Menus, Pointer). Fare tuşlarının anlamı her uygulamada aynı olmalı.

Dokunmatik ve mobil

Hareketler (kaydır, yakınlaştır), tam ekran uygulamalar, bildirim merkezi.

Ses, sohbet ve ajanlar

Sesli asistanlar ve doğal dil arayüzleri; işletim sistemi hizmetlerine yeni bir giriş noktası.

Kullanıcı arayüzü çoğu zaman çekirdeğin parçası değildir, ama **paradigma** tüm sistem tasarımını etkiler: GUI olay güdümlü bir yürütme modeli, hızlı pencere sistemi ve tutarlı bir uygulama çerçevesi gerektirir. Önemli olan **tutarlılıktır**: aynı eylem her yerde aynı sonucu vermelidir.



gecikme bütçesi: dokunmadan ekrana ≈ 16 ms (60 Hz); uzun bir işleyici tüm arayüzü dondurur

Yürütme Paradigmaları

(a) Algoritmik kod

```
int main(void)
{
    int x, y;
    init();
    do_something();
    read(fd, &x, sizeof x);
    do_something_else();
    write(fd, &y, sizeof y);
    keep_going();
    exit(0);
}
```

Program akışı **program** belirler; işletim sistemine hizmet için çağrı yapar. Derleyici, sıralayıcı, toplu işler.

Yürütme paradigmasının seçimi, sistem çağrılarını da biçimlendirir: algoritmik model bloke eden (blocking) `read` ister; olay güdümlü model ise "sıradaki olayı ver" türünden bir çağrı (`get_message`, `epoll_wait`, `io_uring_enter`) ister.

(b) Olay güdümlü kod (event-driven)

```
int main(void)
{
    msg_t msg;
    init();
    while (get_message(&msg)) {
        switch (msg.type) {
            case 1: ... ; break;
            case 2: ... ; break;
            case 3: ... ; break;
        }
    }
}
```

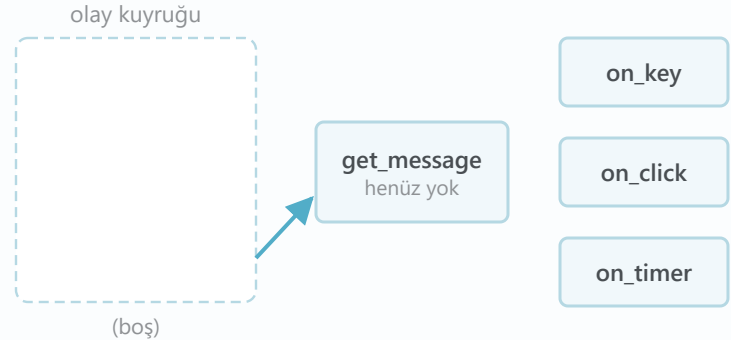
Akışı **dış olaylar** (tuş, tıklama, paket, zamanlayıcı) belirler. GUI'ler, sunucular, mikro çekirdek sunucuları.

Olay Döngüsü Nasıl Çalışır?

```

▶ int main(void) {
▶     msg_t m;
▶     init();

4     while (get_message(&m)) {
5         switch (m.type) {
6             case KEY:    on_key(&m);    break;
7             case CLICK: on_click(&m); break;
8             case TIMER: on_timer(&m); break;
9         }
10    }
11    return 0;
12 }
    
```



Program başlar, `init()` pencereyi ve kaynakları hazırlar. Kuyruk boş.

Adım 1 / 8

DEĞİŞKENLER

`m.type = -`

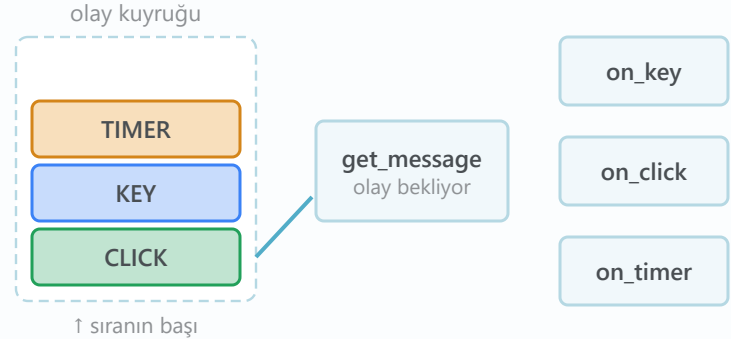
`kuyruk = []`

Olay Döngüsü Nasıl Çalışır?

```

1 int main(void) {
2     msg_t m;
3     init();
4     while (get_message(&m)) {
5         switch (m.type) {
6             case KEY: on_key(&m); break;
7             case CLICK: on_click(&m); break;
8             case TIMER: on_timer(&m); break;
9         }
10    }
11    return 0;
12 }

```



Kullanıcı tıklar, bir tuşa basar, zamanlayıcı dolar. Olayları işletim sistemi/pencere sistemi toplar ve uygulamanın kuyruğına ekler.

Adım 2 / 8

DEĞİŞKENLER

m.type = -

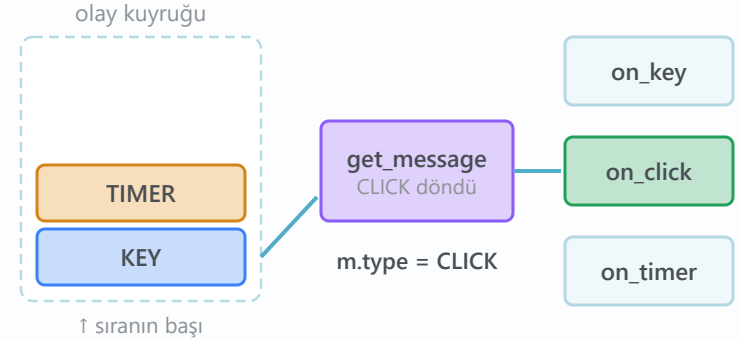
kuyruk = [CLICK, KEY, TIMER]

Olay Döngüsü Nasıl Çalışır?

```

1 int main(void) {
2     msg_t m;
3     init();
4     while (get_message(&m)) {
5         switch (m.type) {
6             case KEY: on_key(&m); break;
7             case CLICK: on_click(&m); break;
8             case TIMER: on_timer(&m); break;
9         }
10    }
11    return 0;
12 }

```



get_message sıranın başındaki CLICK'i döndürür;
switch onu on_click'e yönlendirir.

Adım 3 / 8

DEĞİŞKENLER

m.type = **CLICK**

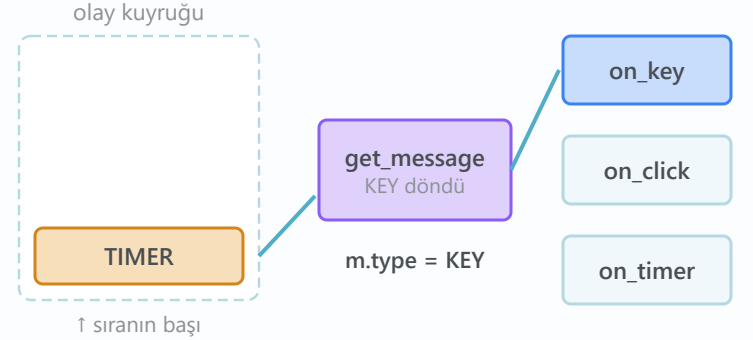
kuyruk = [KEY, TIMER]

Olay Döngüsü Nasıl Çalışır?

```

1 int main(void) {
2     msg_t m;
3     init();
4     while (get_message(&m)) {
5         switch (m.type) {
6             case KEY: on_key(&m); break;
7             case CLICK: on_click(&m); break;
8             case TIMER: on_timer(&m); break;
9         }
10    }
11    return 0;
12 }

```



İşleyici kısa sürede biter ve döngü başa döner. Sıradaki olay KEY: **on_key** çalışır.

Adım 4 / 8

DEĞİŞKENLER

m.type = KEY

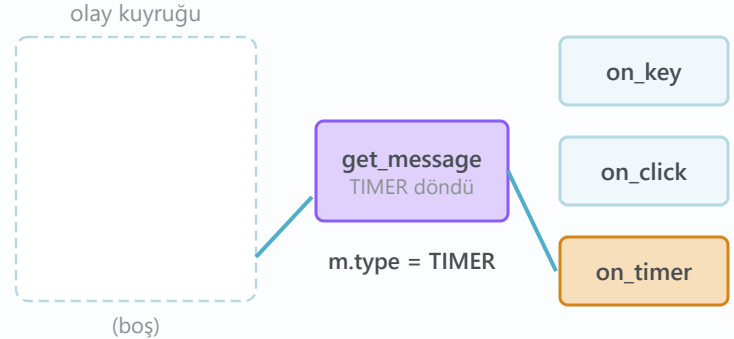
kuyruk = [TIMER]

Olay Döngüsü Nasıl Çalışır?

```

1 int main(void) {
2     msg_t m;
3     init();
4     while (get_message(&m)) {
5         switch (m.type) {
6             case KEY: on_key(&m); break;
7             case CLICK: on_click(&m); break;
8             case TIMER: on_timer(&m); break;
9         }
10    }
11    return 0;
12 }

```



TIMER olayı `on_timer` 'a gider (ör. imleci yakıp söndürmek). Uzun süren bir işleyici tüm arayüzü dondurur; uzun işler ayrı iş parçacığına verilir.

Adım 5 / 8

DEĞİŞKENLER

`m.type = TIMER`

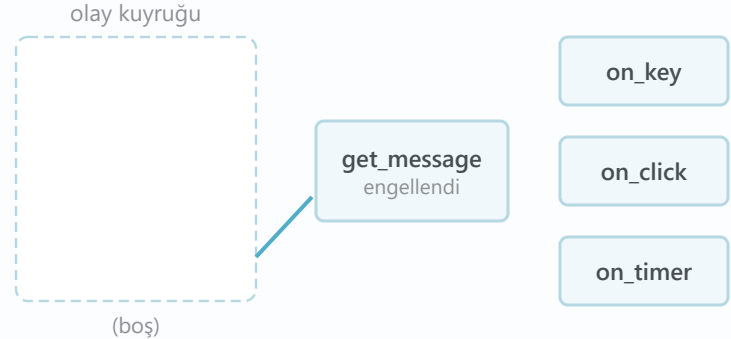
`kuyruk = []`

Olay Döngüsü Nasıl Çalışır?

```

1 int main(void) {
2     msg_t m;
3     init();
4     while (get_message(&m)) {
5         switch (m.type) {
6             case KEY:    on_key(&m);    break;
7             case CLICK: on_click(&m); break;
8             case TIMER: on_timer(&m); break;
9         }
10    }
11    return 0;
12 }

```



Kuyruk boş: **get_message engellenir**, süreç uyur ve CPU harcamaz. Yeni bir olay onu uyandırır.

Adım 6 / 8

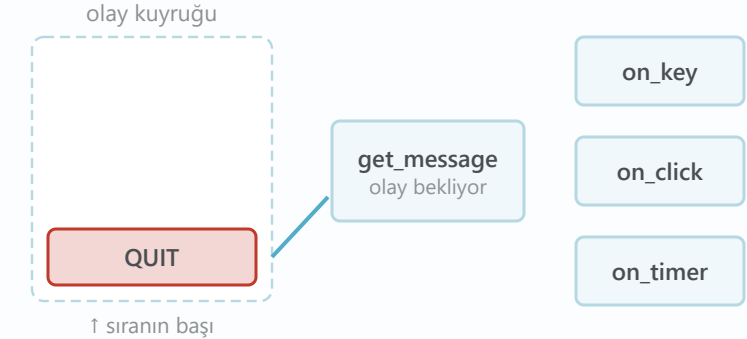
DEĞİŞKENLER

m.type = **TIMER**

kuyruk = []

Olay Döngüsü Nasıl Çalışır?

```
1 int main(void) {  
2     msg_t m;  
3     init();  
4     while (get_message(&m)) {  
5         switch (m.type) {  
6             case KEY:  on_key(&m);  break;  
7             case CLICK: on_click(&m); break;  
8             case TIMER: on_timer(&m); break;  
9         }  
10    }  
11    return 0;  
12 }
```



Kullanıcı pencereyi kapatır: kuyruğa QUIT olayı gelir.

Adım 7 / 8

DEĞİŞKENLER

m.type = -

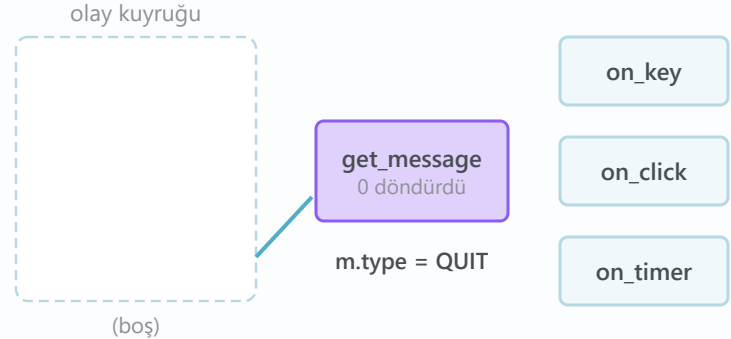
kuyruk = [QUIT]

Olay Döngüsü Nasıl Çalışır?

```

1 int main(void) {
2     msg_t m;
3     init();
4     while (get_message(&m)) {
5         switch (m.type) {
6             case KEY:    on_key(&m);    break;
7             case CLICK: on_click(&m);  break;
8             case TIMER: on_timer(&m);  break;
9         }
10    }
11    return 0;
12 }

```



QUIT için `get_message` 0 döndürür, döngü biter, program sonlanır. Aynı kalıp nginx/Node.js (epoll) ve mikro çekirdek sunucularında da kullanılır.

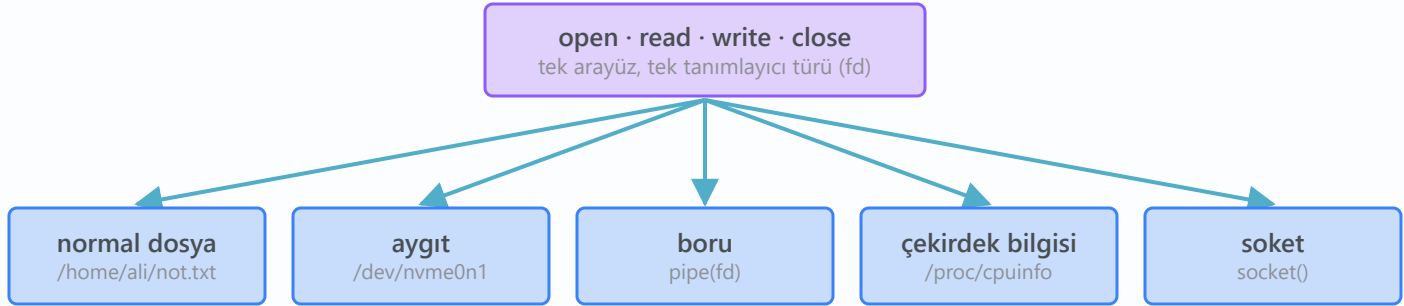
Adım 8 / 8

DEĞİŞKENLER

m.type = QUIT

kuyruk = []

Veri Paradigmaları

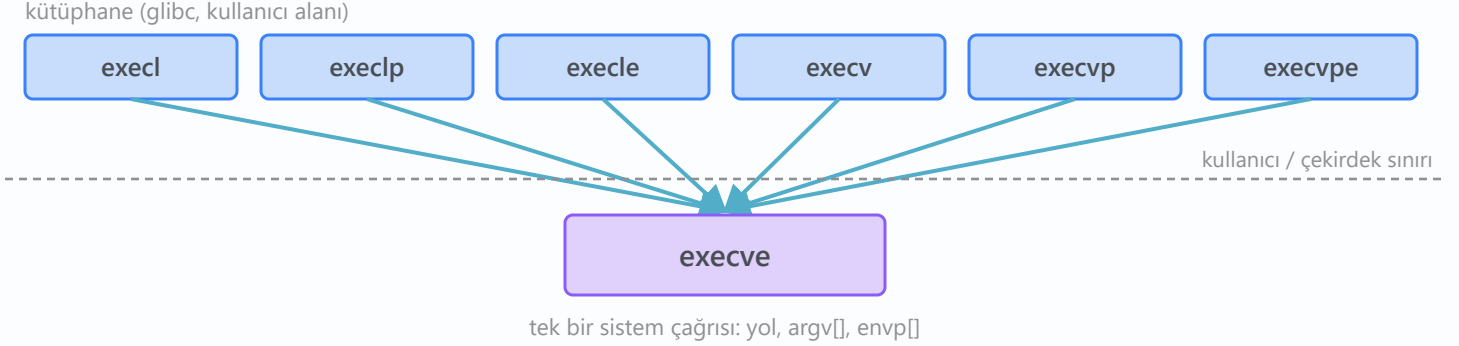


UNIX: "her şey bir dosyadır" · Plan 9: ağ ve pencereler de dosya sistemi

Windows: "her şey bir nesnedir" → HANDLE
nesne yöneticisi: \Device\HarddiskVolume1, olay, mutex, süreç, iş parçacığı

- Ortak bir veri paradigması, aynı aracın (`cat`, `dd`, yönlendirme) her kaynakta çalışmasını sağlar: `cat /proc/loadavg`.
- Bedeli: bazı aygıtlar dosya modeline sığmaz; o zaman her şey `ioctl` gibi "kaçış kapılarına" yığılır.

Sistem Çağrısı Arayüzü



- **Ayrımı koru:** çağrı az ve genel, kolaylık fonksiyonları kütüphanede. Yeni bir kolaylık eklemek çekirdeği değiştirmez.
- **Bağımsız bileşenler:** UNIX süreç oluşturmayı ikiye böler: `fork` + `execve`. Arada yönlendirme, `chdir` vb. yapılabilir.
- Windows'ta `CreateProcess` tek çağrıda 10 parametre alır: güçlü ama karmaşık.
- Modern eğilim: `posix_spawn` (fork maliyetini önler) ve **toplu çağrı** arayüzleri (`io_uring` : tek geçişte çok sayıda istek).

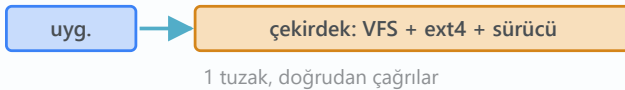
SİSTEM YAPISI

Sistem Yapısı Seçenekleri

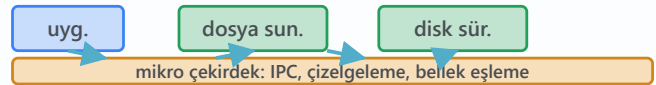
Yapı	Çekirdekte ne var?	Artısı	Eksisi	Örnek
Monolitik	Her şey tek adres alanında	Hızlı, doğrudan çağrı	Bir hata tüm sistemi düşürür	Linux, FreeBSD
Katmanlı	Hiyerarşik katmanlar	Modüler, katman katman test	Katman sırası ve geçiş maliyeti	THE (1968), MULTICS
Exokernel	Yalnızca güvenli kaynak paylaşdırma	Uygulamaya özel soyutlama	Kütüphane işletim sistemi (libOS) karmaşıklığı	Aegis/ExOS (MIT)
Mikro çekirdek	IPC, çizelgeleme, bellek eşleme	Güvenilirlik, yalıtım	İleti (IPC) maliyeti	MINIX 3, QNX, seL4, Zircon
Genişletilebilir	Çekirdek + güvenli eklentiler	Çalışırken uyarlanır	Eklentinin güvenliği	SPIN, Linux eBPF
Hibrit	Mikro çekirdek fikri, monolitik uygulama	Pragmatik denge	Saf modelin yararları kısmen kaybolur	Windows NT, macOS XNU

Bu bölümdeki beş yapı (katmanlı, exokernel, mikro çekirdek tabanlı istemci-sunucu, genişletilebilir sistemler ve çekirdek iş parçacıkları) birbirini dışlamaz: gerçek sistemler birkaçını birlikte kullanır. Örneğin Linux monoliktir ama modüller ve eBPF ile genişletilebilir, çekirdek iş parçacıkları kullanır.

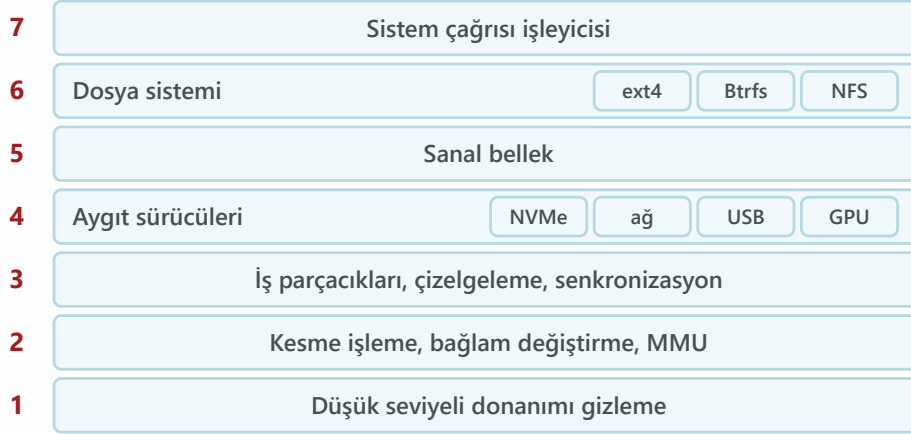
monolitik: tek adres alanı



mikro çekirdek: iletiler



Katmanlı Sistemler: Yedi Katmanlı Bir Tasarım

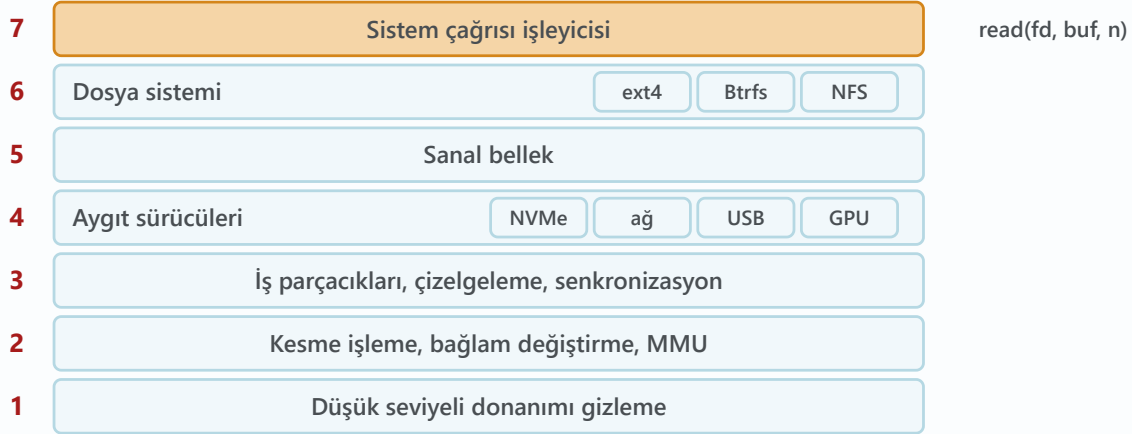


Tanenbaum'ın önerdiği **yedi katmanlı** tasarım. 1. katman donanım farklılıklarını gizler; üst katmanlar hep alttakinin sunduğu soyutlamayla çalışır.

Adım 1 / 7

Her katman yalnızca altındakini kullanır; arayüzler iyi tanımlıysa bir katmandaki değişiklik diğerlerini etkilemez ve katmanlar alttan üste tek tek test edilir. Bedeli: katman sırasını seçmek ve geçiş maliyeti. Kaba görünüm: uygulama → kütüphane → sistem çağrısı → çekirdek → donanım.

Katmanlı Sistemler: Yedi Katmanlı Bir Tasarım

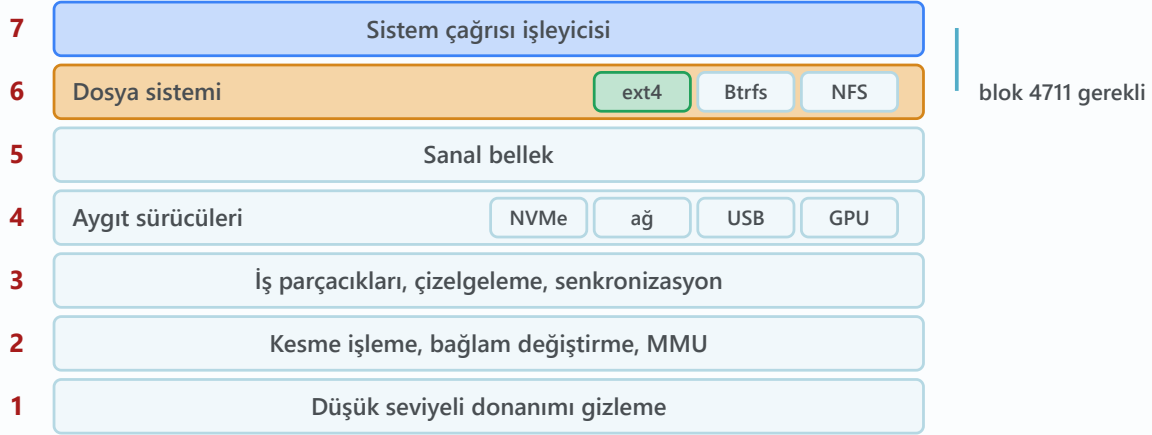


Uygulama `read` çağırır; tuzak 7. katmana gelir. Parametreler (fd geçerli mi, tampon kullanıcı alanında mı?) burada denetlenir.

Adım 2 / 7

Her katman yalnızca altındakini kullanır; arayüzler iyi tanımlıysa bir katmandaki değişiklik diğerlerini etkilemez ve katmanlar alttan üste tek tek test edilir. Bedeli: katman sırasını seçmek ve geçiş maliyeti. Kaba görünüm: uygulama → kütüphane → sistem çağrısı → çekirdek → donanım.

Katmanlı Sistemler: Yedi Katmanlı Bir Tasarım

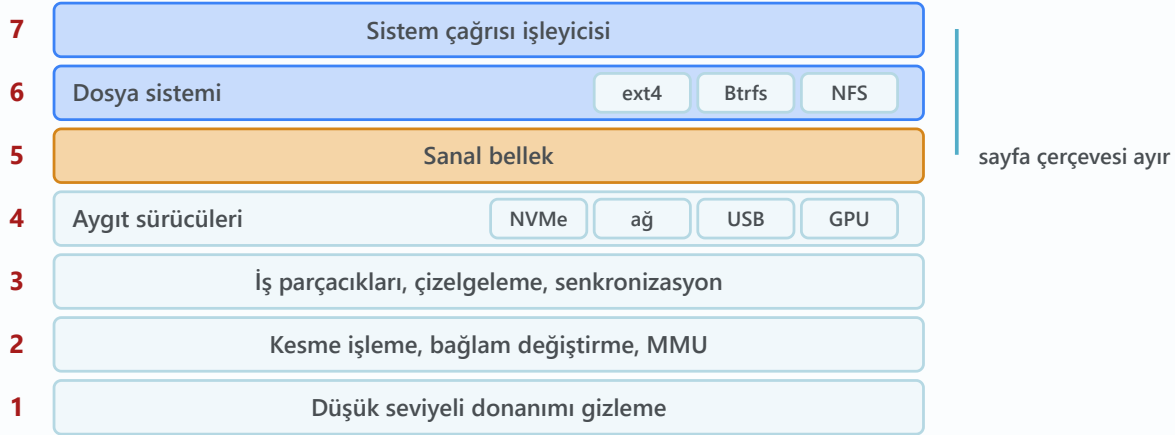


Dosya sistemi (ext4) dosya ofsetini disk bloğuna çevirir. Blok önbellekte yoksa alt katmanlardan istenir.

Adım 3 / 7

Her katman yalnızca altındakini kullanır; arayüzler iyi tanımlıysa bir katmandaki değişiklik diğerlerini etkilemez ve katmanlar alttan üste tek tek test edilir. Bedeli: katman sırasını seçmek ve geçiş maliyeti. Kaba görünüm: uygulama → kütüphane → sistem çağrısı → çekirdek → donanım.

Katmanlı Sistemler: Yedi Katmanlı Bir Tasarım

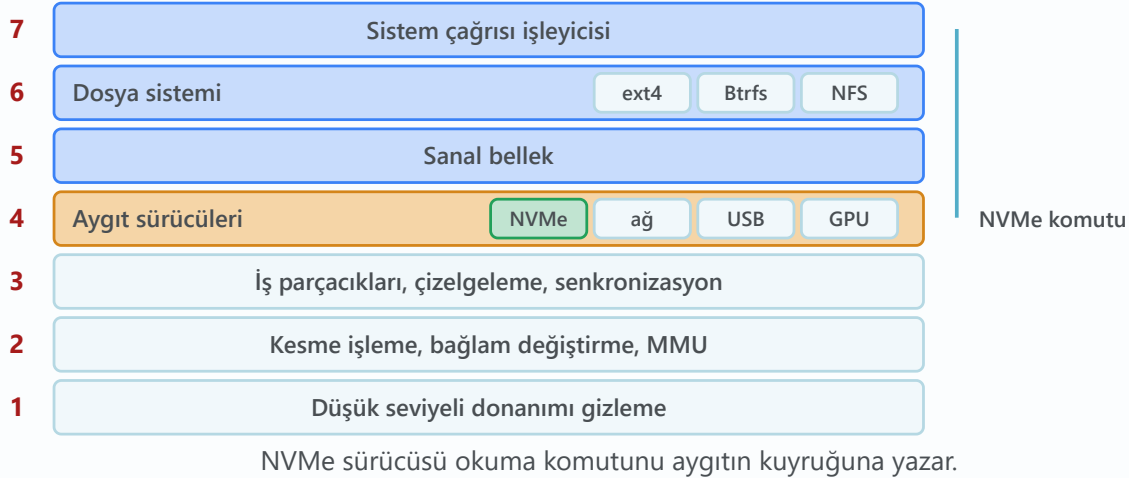


Sanal bellek, verinin konacağı sayfa ön belleği çerçevesini ayırır ve kullanıcı tamponunun eşleşmesini doğrular.

Adım 4 / 7

Her katman yalnızca altındakini kullanır; arayüzler iyi tanımlıysa bir katmandaki değişiklik diğerlerini etkilemez ve katmanlar alttan üste tek tek test edilir. Bedeli: katman sırasını seçmek ve geçiş maliyeti. Kaba görünüm: uygulama → kütüphane → sistem çağrısı → çekirdek → donanım.

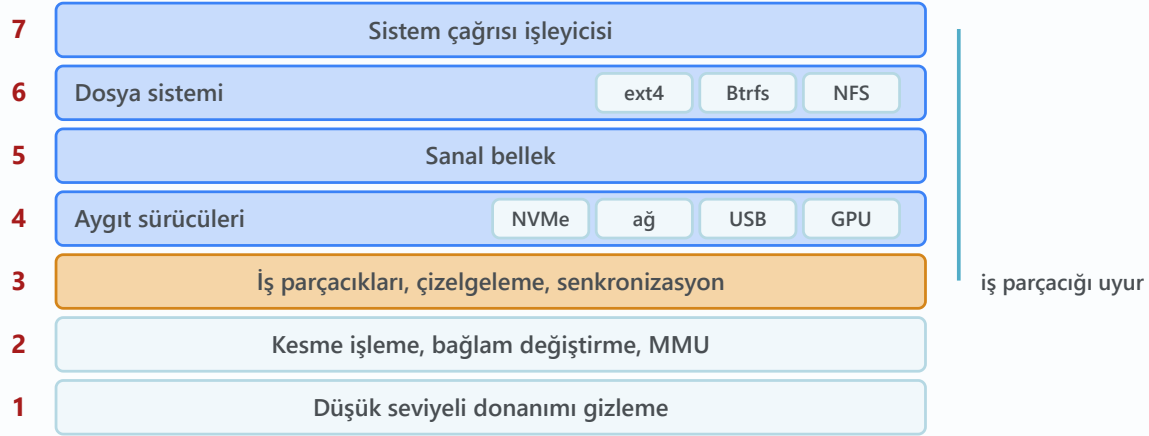
Katmanlı Sistemler: Yedi Katmanlı Bir Tasarım



Adım 5 / 7

Her katman yalnızca altındakini kullanır; arayüzler iyi tanımlıysa bir katmandaki değişiklik diğerlerini etkilemez ve katmanlar alttan üste tek tek test edilir. Bedeli: katman sırasını seçmek ve geçiş maliyeti. Kaba görünüm: uygulama → kütüphane → sistem çağrısı → çekirdek → donanım.

Katmanlı Sistemler: Yedi Katmanlı Bir Tasarım

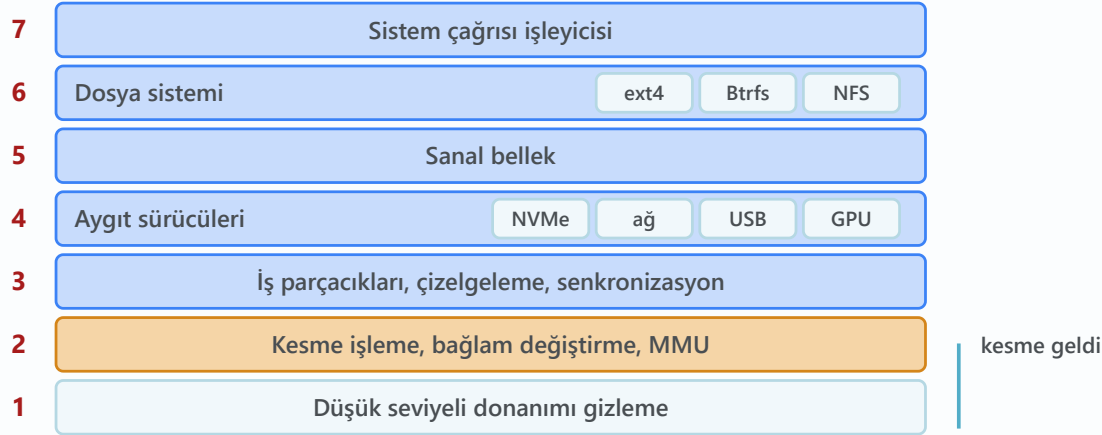


Çağırılan iş parçacığı G/Ç bitene kadar **engellenir**; çizelgeleyici başka bir iş parçacığını çalıştırır.

Adım 6 / 7

Her katman yalnızca altındakini kullanır; arayüzler iyi tanımlıysa bir katmandaki değişiklik diğerlerini etkilemez ve katmanlar alttan üste tek tek test edilir. Bedeli: katman sırasını seçmek ve geçiş maliyeti. Kaba görünüm: uygulama → kütüphane → sistem çağrısı → çekirdek → donanım.

Katmanlı Sistemler: Yedi Katmanlı Bir Tasarım



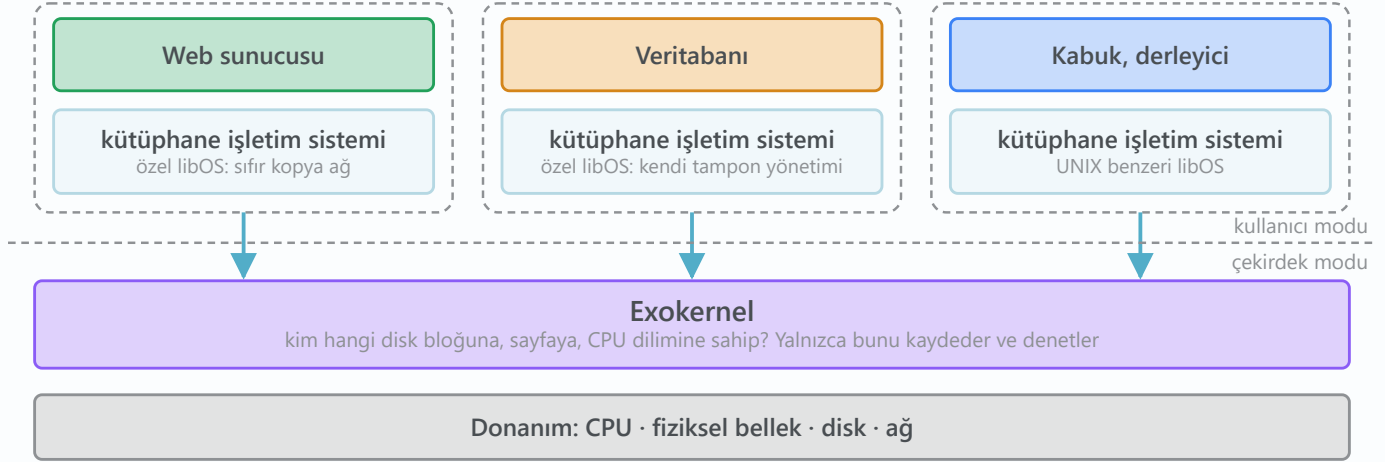
Aygıt bitince **kesme** üretir; 2. katman kesmeyi işler, uyuyan iş parçacığını hazır yapar. Sonuç katmanlar boyunca yukarı döner.

Adım 7 / 7

Her katman yalnızca altındakini kullanır; arayüzler iyi tanımlıysa bir katmandaki değişiklik diğerlerini etkilemez ve katmanlar alttan üste tek tek test edilir. Bedeli: katman sırasını seçmek ve geçiş maliyeti. Kaba görünüm: uygulama → kütüphane → sistem çağrısı → çekirdek → donanım.

SİSTEM YAPISI

Exokernel



- Donanım ile yazılım arasında **en ince** soyutlama katmanı.
- Kaynaklar **ayrıntılı** (blok, sayfa, zaman dilimi) düzeyinde ayrılır; soyutlamalar (dosya, süreç) kullanıcı alanındaki **kütüphane işletim sisteminde** (libOS).
- Sanal makineden farkı: her VM'e makinenin kopyası değil, kaynağın bir **bölümü** verilir; yeniden eşleme katmanı gerekmez.
- Bedeli: tasarım ve uygulama karmaşıklığı, uygulamalar arası paylaşımın zorluğu. Fikri bugün **unikernel**'lerde yaşıyor.

Çekirdek bileşenlerinin dinamik yüklenmesi exokernel'in değil, genişletilebilir sistemlerin özelliğidir.

Mikro Çekirdek Tabanlı İstemci-Sunucu Sistemleri



Çekirdek **olabildiğince küçük**: yalnızca süreç yönetimi, süreçler arası iletişim (IPC) ve temel bellek eşleme. Dosya sistemi, sürücüler, hatta bellek politikası kullanıcı süreçleridir.

Adım 1 / 7

Örnekler: Mach (macOS'taki XNU melez bir türevidir), QNX (otomotiv), L4 ailesi ve biçimsel doğrulanmış **seL4**, MINIX 3, Google Fuchsia'nın Zircon çekirdeği. Sunucular ağdaki farklı düğümlerde de çalışabilir; ancak "ileti alışverişi kendiliğinden güvenlidir" demek abartıdır: güvenlik, çekirdeğin yetki denetimine bağlıdır.

Mikro Çekirdek Tabanlı İstemci-Sunucu Sistemleri



İstemci bir dosya okumak ister: dosya sunucusuna **ileti** gönderir. Bu, çekirdeğe yapılan tek tür çağrıdır.

Adım 2 / 7

Örnekler: Mach (macOS'taki XNU melez bir türevidir), QNX (otomotiv), L4 ailesi ve biçimsel doğrulanmış **seL4**, MINIX 3, Google Fuchsia'nın Zircon çekirdeği. Sunucular ağdaki farklı düğümlerde de çalışabilir; ancak "ileti alışverişi kendiliğinden güvenlidir" demek abartıdır: güvenlik, çekirdeğin yetki denetimine bağlıdır.

Mikro Çekirdek Tabanlı İstemci-Sunucu Sistemleri

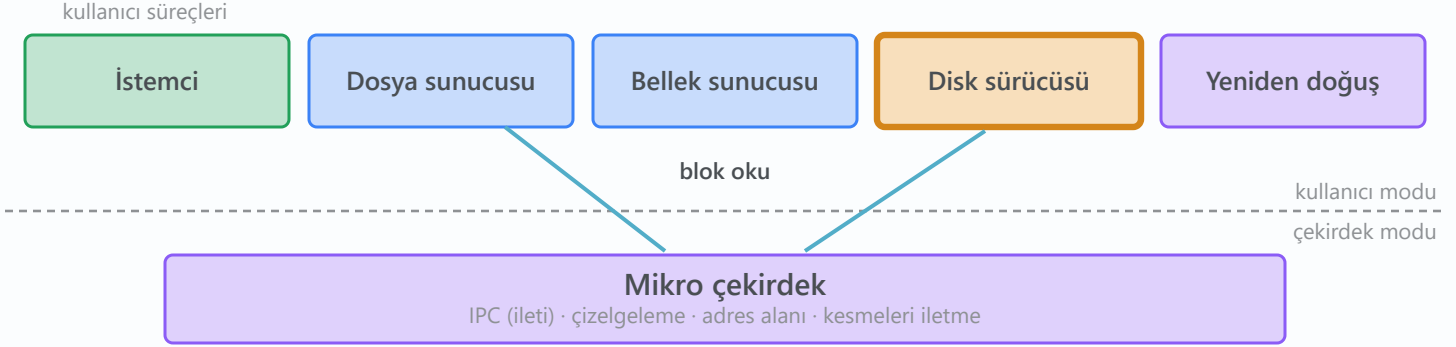


Mikro çekirdek iletiyi dosya sunucusunun adres alanına kopyalar (veya eşler) ve onu çalıştırır.

Adım 3 / 7

Örnekler: Mach (macOS'taki XNU melez bir türevidir), QNX (otomotiv), L4 ailesi ve biçimsel doğrulanmış **seL4**, MINIX 3, Google Fuchsia'nın Zircon çekirdeği. Sunucular ağdaki farklı düğümlerde de çalışabilir; ancak "ileti alışverişi kendiliğinden güvenlidir" demek abartıdır: güvenlik, çekirdeğin yetki denetimine bağlıdır.

Mikro Çekirdek Tabanlı İstemci-Sunucu Sistemleri

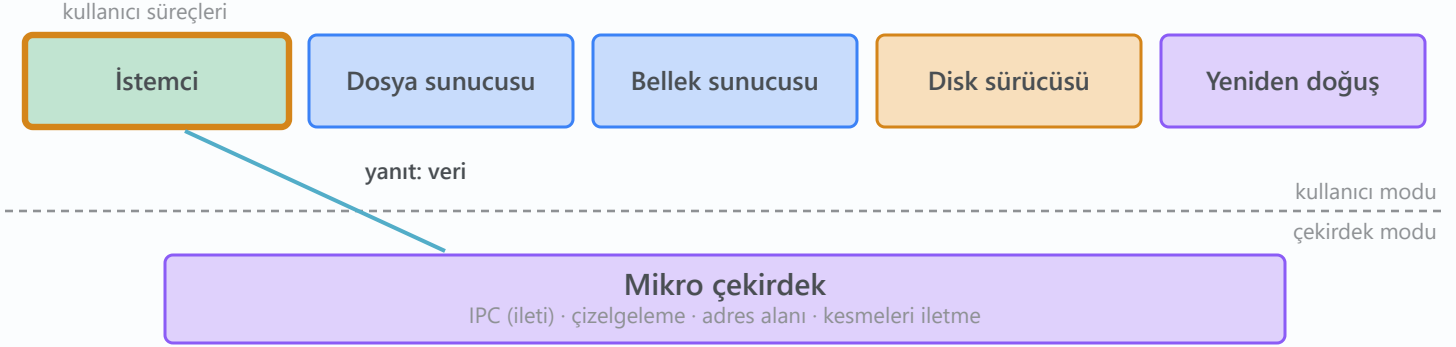


Dosya sunucusu bloğu disk sürücüsünden ister; o da bir kullanıcı sürecidir. Aygıt yazmaçlarına erişim çekirdeğin verdiği izinlerle olur.

Adım 4 / 7

Örnekler: Mach (macOS'taki XNU melez bir türevidir), QNX (otomotiv), L4 ailesi ve biçimsel doğrulanmış **seL4**, MINIX 3, Google Fuchsia'nın Zircon çekirdeği. Sunucular ağdaki farklı düğümlerde de çalışabilir; ancak "ileti alışverişi kendiliğinden güvenlidir" demek abartıdır: güvenlik, çekirdeğin yetki denetimine bağlıdır.

Mikro Çekirdek Tabanlı İstemci-Sunucu Sistemleri

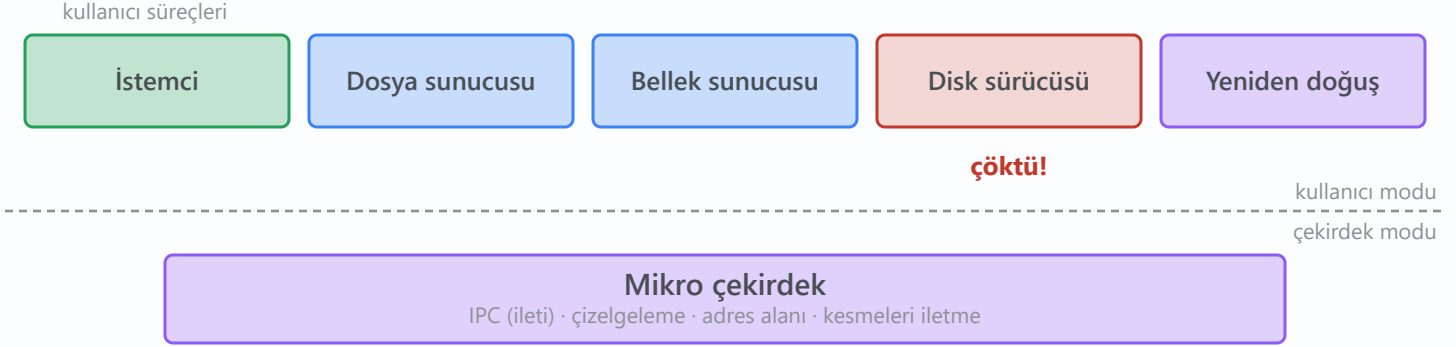


Yanıt aynı yoldan istemciye döner. Her istek birden çok bağlam değiştirme ve kopyalama demektir: **performans bedeli**.

Adım 5 / 7

Örnekler: Mach (macOS'taki XNU melez bir türevidir), QNX (otomotiv), L4 ailesi ve biçimsel doğrulanmış **seL4**, MINIX 3, Google Fuchsia'nın Zircon çekirdeği. Sunucular ağdaki farklı düğümlerde de çalışabilir; ancak "ileti alışverişi kendiliğinden güvenlidir" demek abartıdır: güvenlik, çekirdeğin yetki denetimine bağlıdır.

Mikro Çekirdek Tabanlı İstemci-Sunucu Sistemleri



Kazanç: sürücü çökerse yalnız o süreç ölür, çekirdek ve diğer sunucular çalışmaya devam eder.

Adım 6 / 7

Örnekler: Mach (macOS'taki XNU melez bir türevidir), QNX (otomotiv), L4 ailesi ve biçimsel doğrulanmış **seL4**, MINIX 3, Google Fuchsia'nın Zircon çekirdeği. Sunucular ağdaki farklı düğümlerde de çalışabilir; ancak "ileti alışverişi kendiliğinden güvenlidir" demek abartıdır: güvenlik, çekirdeğin yetki denetimine bağlıdır.

Mikro Çekirdek Tabanlı İstemci-Sunucu Sistemleri

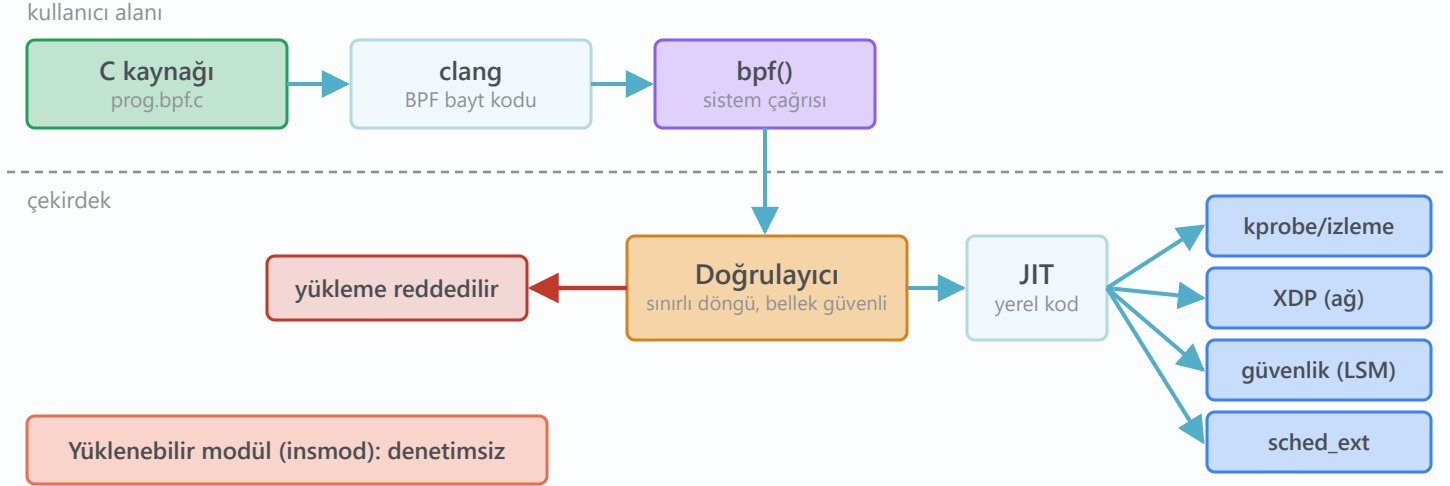


MINIX 3'te **yeniden doğuş sunucusu** (reincarnation server) çöken sürücüyü fark edip yeniden başlatır: kendini onaran sistem.

Adım 7 / 7

Örnekler: Mach (macOS'taki XNU melez bir türevidir), QNX (otomotiv), L4 ailesi ve biçimsel doğrulanmış **seL4**, MINIX 3, Google Fuchsia'nın Zircon çekirdeği. Sunucular ağdaki farklı düğümlerde de çalışabilir; ancak "ileti alışverişi kendiliğinden güvenlidir" demek abartıdır: güvenlik, çekirdeğin yetki denetimine bağlıdır.

Geniřletilebilir Sistemler



- Çekirdek temel işlevleri sağlar; değişen ihtiyaçlar için **çalışırken** yeni bileşen eklenir.
- Üçüncü taraflar sisteme özellik ekleyebilir; uyumluluk ve **güvenlik** en büyük sorundur.

- Klasik modül tam yetkilidir: bir hata çekirdeği çökertir. SPIN (1995) eklentileri güvenli dil (Modula-3) ile sınırladı.
- Bugün Linux'ta **eBPF**: doğrulanmış küçük programlar; izleme, ağ, güvenlik ve 6.12'den beri çizelgeleyici (`sched_ext`).

Çekirdek İş Parçacıkları

kullanıcı süreçleri (kendi adres alanları)

bash

firefox

postgres

çekirdek adres alanı (tüm süreçlerde ortak)

kthreadd

kworker/0:1

ksoftirqd/0

kswapd0

jbd2/nvme0n1

- Çekirdek tarafından oluşturulur ve **yönetilir** (`kthread_create`); kullanıcı adres alanları yoktur, yalnızca çekirdek belleğinde çalışırlar.
- Normal iş parçacığı gibi **zamanlanır**, engellenebilir, uyuyabilir: kesme işleyicisinde yapılamayan uzun işler buraya ertelenir.
- Bağlam değiştirme ucuz: adres alanı (sayfa tablosu) değişmez.
- Örnekler: ertelenmiş iş (kworker), yumuşak kesmeler (ksoftirqd), sayfa geri kazanma (kswapd), günlük yazma (jbd2).
- `ps -ef` çıktısında köşeli parantez içinde görünürler: `[kswapd0]`.

Kullanıcı süreçleri çekirdek iş parçacığı **oluşturamaz**; sistem çağrıları yalnızca çekirdeğin zaten çalıştırdığı iş parçacıklarına iş verebilir (ör. `io_uring` 'in iş havuzu).

Mekanizma ve Politikanın Ayrılması

Politika (kullanıcı alanı)

henüz müdahale yok

çekirdek

Mekanizma: öncelik tabanlı çizelgeleyici (en yüksek öncelikli hazır iş parçacığını seç)

T1 arayüz
öncelik 5

T2 sorgu
öncelik 3

T3 yedekleme
öncelik 8

T4 günlük
öncelik 1

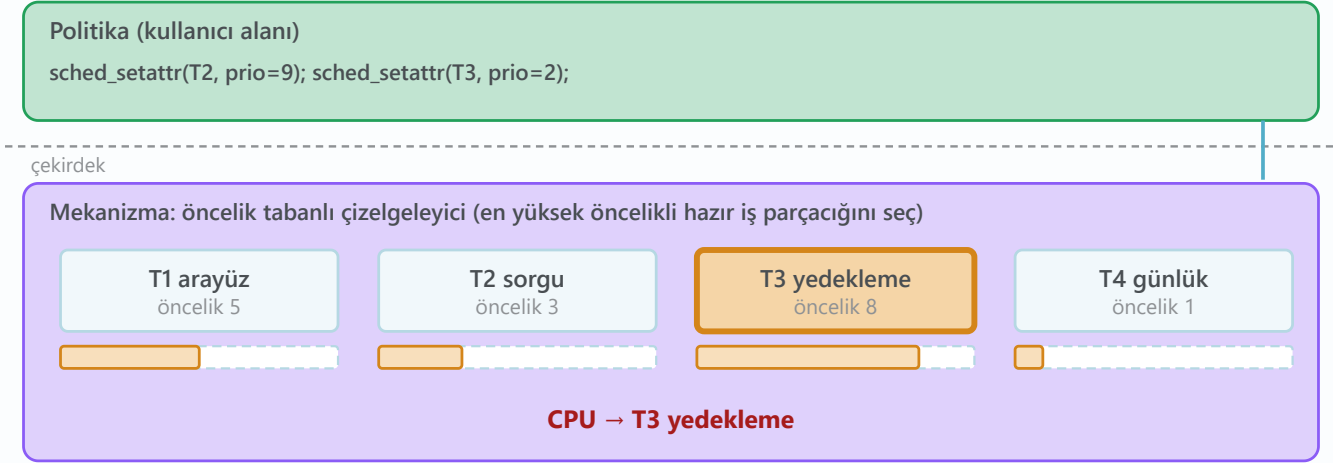
CPU → T3 yedekleme

Çekirdek yalnızca bir **mekanizma** sunar: “en yüksek öncelikli hazır iş parçacığını çalıştır”. Şu an T3 (yedekleme, öncelik 8) seçiliyor.

Adım 1 / 4

Diğer örnekler: sayfalama (mekanizma: MMU, çerçeve tahsisi; politika: hangi sayfa çıkarılsın), CPU frekansı (mekanizma: cpufreq sürücüsü; politika: `schedutil` / `powersave` yönetici), güvenlik (mekanizma: LSM kancaları; politika: SELinux kuralları).

Mekanizma ve Politikanın Ayrılması



Veritabanı süreci kendi iş parçacıklarını en iyi bilen taraftır: sorguların önemli, yedeklemenin bekleyebileceğine karar verir. Bu bir **politika**dır ve bir sistem çağrısıyla aktarılır.

Adım 2 / 4

Diğer örnekler: sayfalama (mekanizma: MMU, çerçeve tahsisi; politika: hangi sayfa çıkarılsın), CPU frekansı (mekanizma: cpufreq sürücüsü; politika: `schedutil` / `powersave` yönetici), güvenlik (mekanizma: LSM kancaları; politika: SELinux kuralları).

Mekanizma ve Politikanın Ayrılması

Politika (kullanıcı alanı)

```
sched_setattr(T2, prio=9); sched_setattr(T3, prio=2);
```

çekirdek

Mekanizma: öncelik tabanlı çizelgeleyici (en yüksek öncelikli hazır iş parçacığını seç)



Çekirdek kodu **hiç değişmeden** artık T2'yi seçer. Aynı mekanizma, farklı politikayla farklı davranış üretir.

Adım 3 / 4

Diğer örnekler: sayfalama (mekanizma: MMU, çerçeve tahsisi; politika: hangi sayfa çıkarılsın), CPU frekansı (mekanizma: cpufreq sürücüsü; politika: `schedutil` / `powersave` yönetici), güvenlik (mekanizma: LSM kancaları; politika: SELinux kuralları).

Mekanizma ve Politikanın Ayrılması

Politika (kullanıcı alanı)

etkileşimli politika: arayüz iş parçacığı önce

çekirdek

Mekanizma: öncelik tabanlı çizelgeleyici (en yüksek öncelikli hazır iş parçacığını seç)

T1 arayüz

öncelik 9

T2 sorgu

öncelik 6

T3 yedekleme

öncelik 2

T4 günlük

öncelik 1

CPU → T1 arayüz

Başka bir uygulama başka bir politika uygular. Linux 6.12'deki **sched_ext** bunu bir adım öteye taşır: çizelgeleme politikasının tamamı bir eBPF programı olarak yüklenebilir.

Adım 4 / 4

Diğer örnekler: sayfalama (mekanizma: MMU, çerçeve tahsisi; politika: hangi sayfa çıkarılsın), CPU frekansı (mekanizma: cpufreq sürücüsü; politika: **schedutil** / **powersave** yönetici), güvenlik (mekanizma: LSM kancaları; politika: SELinux kuralları).

Diklik (Orthogonality)

CLONE_VM

adres alanı

CLONE_FS

kök/çalışma dizini

CLONE_FILES

açık dosyalar

CLONE_SIGHAND

sinyal işleyiciler

CLONE_THREAD

aynı iş parçacığı grubu

fork()

ara biçim pthread_create()

kopyala

kopyala

paylaş

kopyala

paylaş

paylaş

kopyala

paylaş

paylaş

kopyala

kopyala

paylaş

kopyala

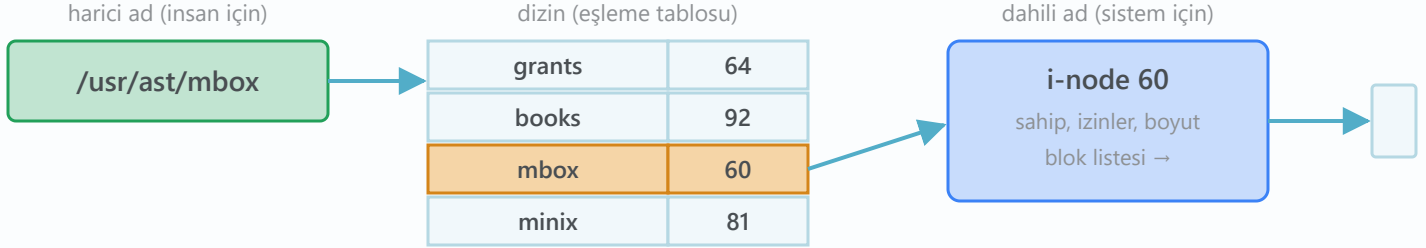
kopyala

paylaş

süreç ← tek çağrı: clone(flags) → iş parçacığı

- **Diklik** (orthogonality): ayrı kavramları **bağımsız** olarak birleştirebilme. Birini değiştirmek diğerini etkilemez.
- Linux **clone** : neyin paylaşılacağı ayrı ayrı bayraklarla seçilir. Süreç ve iş parçacığı aynı mekanizmanın iki ucudur; konteynerler de **CLONE_NEW*** bayraklarıyla kurulur.
- Windows: süreç bir **kaynak kabıdır**, iş parçacığı ise **çizelgeleme birimi**; iki kavram ayrıdır.
- C dilinde de örnek: **static** (ömür) ile tür bağımsız seçilir.

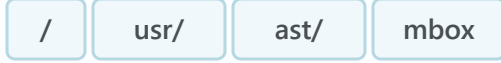
Adlandırma: İki Düzeyli Adlar



Harici ad	Eşleyen	Dahili ad
/usr/ast/mbox	dizin	i-node numarası
www.giresun.edu.tr	DNS	IP adresi → (ARP) MAC adresi
dosya tanımlayıcısı 3	süreç dosya tablosu	çekirdekteki struct file
sanal adres	sayfa tablosu	fiziksel çerçeve
\Device\Harddisk0	nesne yöneticisi	çekirdek nesnesi (HANDLE)

Ad, **insanların** hatırlayabileceği biçimde (ASCII/Unicode dizgisi) seçilir; sistem içinde ise hızlı ve benzersiz bir sayı kullanılır. Aradaki eşleme, adı değiştirmeden nesneyi taşımaya olanak verir.

Ad Çözümleme Adım Adım

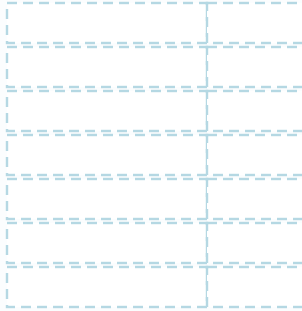


disk okuması: 0

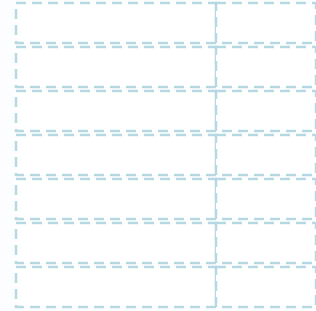
i-node 1 → kök dizin · blok 1



i-node 6 → /usr · blok 132



i-node 26 → /usr/ast · blok 406



Çekirdek `open("/usr/ast/mbox")` çağrısını alır. Yol bileşenlere ayrılır; hiçbir şey önbellekte yoksa her bileşen için diskten okunmak gerekir.

Ad Çözümleme Adım Adım



disk okuması: 2

i-node 1 → kök dizin · blok 1

.	1
..	1
bin	4
dev	7
lib	14
etc	9
usr	6
tmp	8

i-node 6 → /usr · blok 132

A 6x8 grid of dashed boxes representing the structure of i-node 6.

i-node 26 → /usr/ast · blok 406

A 6x8 grid of dashed boxes representing the structure of i-node 26.

Kök dizinin i-node'u (i-node 1) okunur, ardından kök dizinin verisi (blok 1). Doğrusal arama: **usr** → i-node 6. **2 okuma.**

Ad Çözümleme Adım Adım

/ usr/ ast/ mbox

disk okuması: 4

i-node 1 → kök dizin · blok 1

.	1
..	1
bin	4
dev	7
lib	14
etc	9
usr	6
tmp	8

i-node 6 → /usr · blok 132

.	6
..	1
dick	19
erik	30
jim	51
ast	26
bal	45

i-node 26 → /usr/ast · blok 406

/usr için i-node 6 okunur, sonra /usr dizini (blok 132). **ast** → i-node 26. **4 okuma.**

Ad Çözümleme Adım Adım

/ usr/ ast/ mbox

disk okuması: 6

i-node 1 → kök dizin · blok 1

.	1
..	1
bin	4
dev	7
lib	14
etc	9
usr	6
tmp	8

i-node 6 → /usr · blok 132

.	6
..	1
dick	19
erik	30
jim	51
ast	26
bal	45

i-node 26 → /usr/ast · blok 406

.	26
..	6
grants	64
books	92
mbox	60
minix	81
src	17

/usr/ast için i-node 26 okunur, sonra dizin bloğu 406. **mbox** → i-node 60. **6 okuma.**

Ad Çözümleme Adım Adım

/ usr/ ast/ mbox

disk okuması: 7

i-node 1 → kök dizin · blok 1

.	1
..	1
bin	4
dev	7
lib	14
etc	9
usr	6
tmp	8

i-node 6 → /usr · blok 132

.	6
..	1
dick	19
erik	30
jim	51
ast	26
bal	45

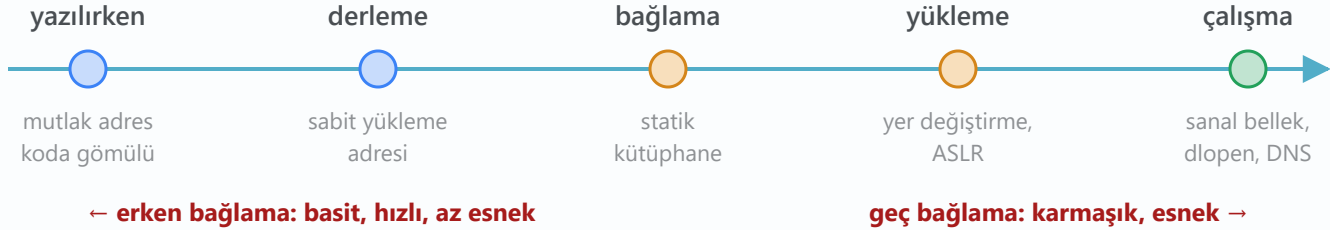
i-node 26 → /usr/ast · blok 406

.	26
..	6
grants	64
books	92
mbox	60
minix	81
src	17

mbox → i-node 60 (7. okuma)

Son olarak dosyanın kendi i-node'u (60) okunur. Sadece dosyayı **bulmak** için 7 disk erişimi gerekti: bu yüzden önbellek şart.

Bağlama Zamanı



Erken bağlama (early binding)

Ad ile nesne arasındaki ilişki önceden sabitlenir.

Basit ve hızlı, ama **az esnek**: program yalnızca belirli adreste çalışır, kütüphane güncellenince yeniden derlenir.

Geç bağlama (late binding)

İlişki mümkün olan en son anda kurulur. **Daha karmaşık** (ek dolaylılık, çalışma zamanı maliyeti) ama **daha esnek**: sayfa nereye yüklenirse yüklensin, kütüphane güncellenince program değişmez.

Bağlama zamanı, “bir ad hangi anda gerçek nesneye bağlanır?” sorusudur. İşletim sistemleri genel olarak geç bağlamayı seçer; örneğin sanal bellek, bir sayfanın fiziksel yerini her erişimde (TLB yardımıyla) yeniden çözer.

Statik ve Dinamik Yapılar

```
► found = 0;
2 for (p = &proc_table[0];
3     p < &proc_table[PROC_TABLE_SIZE];
4     p++) {
5     if (p->proc_pid == pid) {
6         found = 1;
7         break;
8     }
9 }
```

proc_table[PROC_TABLE_SIZE = 6]

1	314	boş	42	977	boş
[0]	[1]	[2]	[3]	[4]	[5]

aranan pid = 42

Statik süreç tablosu: altı giriş, bazıları boş (pid 0). Aranan pid 42.

found sıfırlanır.

Adım 1 / 5

DEĞİŞKENLER

pid = 42

found = 0

p = -

Statik tablo: basit, ayırıcı gerektirmez, ama dolunca yeni süreç oluşturulamaz (eski UNIX'te `NPROC`). **Dinamik** yapı (bağlı liste, hash tablosu, Linux'ta xarray): esnek ve aramada $O(1)$ 'e yakın, ama bellek ayırıcı, parçalanma ve karmaşık kilitleme getirir. Ara yol: boyutu ayarlanabilir tablolar (`/proc/sys/kernel/pid_max`).

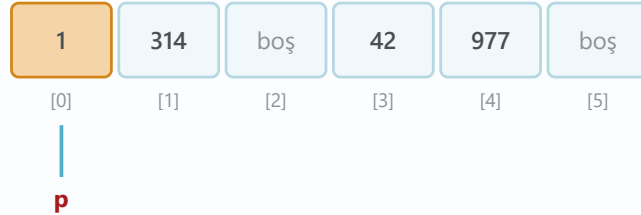
Statik ve Dinamik Yapılar

```

1 found = 0;
▶ for (p = &proc_table[0];
3   p < &proc_table[PROC_TABLE_SIZE];
4   p++) {
▶   if (p->proc_pid == pid) {
6       found = 1;
7       break;
8   }
9 }

```

proc_table[PROC_TABLE_SIZE = 6]



aranan pid = 42

İşaretçi tablonun başına konur. `proc_table[0]` 'ın pid'i 1: eşleşmez.

Adım 2 / 5

DEĞİŞKENLER

pid = 42

found = 0

p = &proc_table[0]

Statik tablo: basit, ayırıcı gerektirmez, ama dolunca yeni süreç oluşturulamaz (eski UNIX'te `NPROC`). **Dinamik** yapı (bağlı liste, hash tablosu, Linux'ta xarray): esnek ve aramada $O(1)$ 'e yakın, ama bellek ayırıcı, parçalanma ve karmaşık kilitleme getirir. Ara yol: boyutu ayarlanabilir tablolar (`/proc/sys/kernel/pid_max`).

Statik ve Dinamik Yapılar

```

1 found = 0;
2 for (p = &proc_table[0];
3     p < &proc_table[PROC_TABLE_SIZE];
4     p++) {
5     if (p->proc_pid == pid) {
6         found = 1;
7         break;
8     }
9 }

```

proc_table[PROC_TABLE_SIZE = 6]



aranan pid = 42

p++ bir sonraki **yapıya** ilerler (bayta değil). 314 ≠ 42.

Adım 3 / 5

DEĞİŞKENLER

pid = 42

found = 0

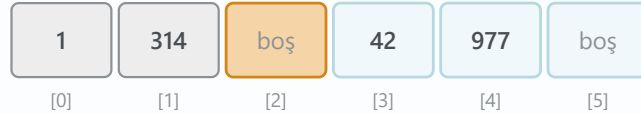
p = &proc_table[1]

Statik tablo: basit, ayırıcı gerektirmez, ama dolunca yeni süreç oluşturulamaz (eski UNIX'te `NPROC`). **Dinamik** yapı (bağlı liste, hash tablosu, Linux'ta xarray): esnek ve aramada $O(1)$ 'e yakın, ama bellek ayırıcı, parçalanma ve karmaşık kitleme getirir. Ara yol: boyutu ayarlanabilir tablolar (`/proc/sys/kernel/pid_max`).

Statik ve Dinamik Yapılar

```
1 found = 0;
2 for (p = &proc_table[0];
3     p < &proc_table[PROC_TABLE_SIZE];
4     p++) {
5     if (p->proc_pid == pid) {
6         found = 1;
7         break;
8     }
9 }
```

proc_table[PROC_TABLE_SIZE = 6]



p

aranan pid = 42

Boş giriş (pid 0) da tek tek taranır: doğrusal aramanın bedeli.

Adım 4 / 5

DEĞİŞKENLER

pid = 42

found = 0

p = &proc_table[2]

Statik tablo: basit, ayırıcı gerektirmez, ama dolunca yeni süreç oluşturulamaz (eski UNIX'te `NPROC`). **Dinamik** yapı (bağlı liste, hash tablosu, Linux'ta xarray): esnek ve aramada $O(1)$ 'e yakın, ama bellek ayırıcı, parçalanma ve karmaşık kilitleme getirir. Ara yol: boyutu ayarlanabilir tablolar (`/proc/sys/kernel/pid_max`).

Statik ve Dinamik Yapılar

```
1 found = 0;
2 for (p = &proc_table[0];
3     p < &proc_table[PROC_TABLE_SIZE];
4     p++) {
5     if (p->proc_pid == pid) {
6         found = 1;
7         break;
8     }
9 }
```

proc_table[PROC_TABLE_SIZE = 6]

1	314	boş	42	977	boş
[0]	[1]	[2]	[3]	[4]	[5]

p

aranan pid = 42

Eşleşme: `found = 1`, `break` ile döngüden çıkılır. En kötü durumda tüm tablo taranır: $O(n)$. Küçük ve sabit bir tablo için bu basit kod yeterlidir.

Adım 5 / 5

DEĞİŞKENLER

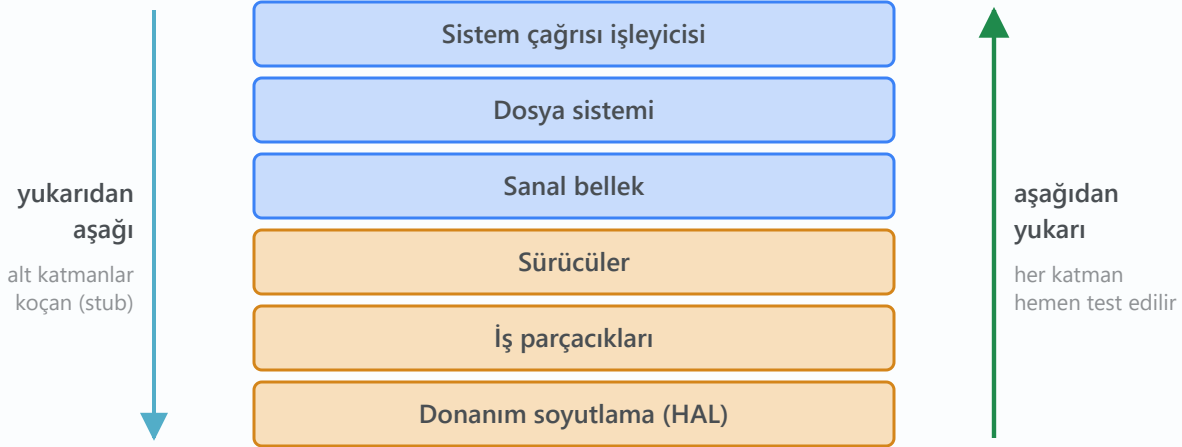
pid = 42

found = 1

p = &proc_table[3]

Statik tablo: basit, ayırıcı gerektirmez, ama dolunca yeni süreç oluşturulamaz (eski UNIX'te `NPROC`). **Dinamik** yapı (bağlı liste, hash tablosu, Linux'ta xarray): esnek ve aramada $O(1)$ 'e yakın, ama bellek ayırıcı, parçalanma ve karmaşık kilitleme getirir. Ara yol: boyutu ayarlanabilir tablolar (`/proc/sys/kernel/pid_max`).

Yukarıdan Aşağı ve Aşağıdan Yukarı Gerçekleştirme



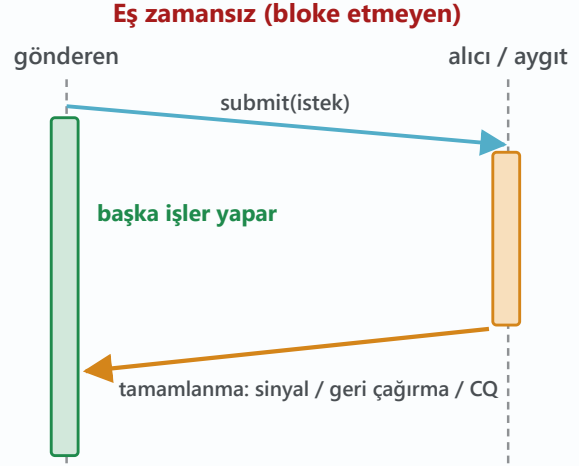
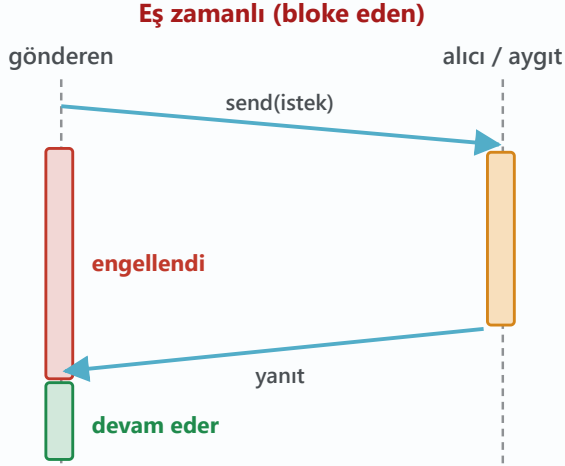
Yukarıdan aşağı

Önce sistem çağrısı işleyicileri yazılır, alt katmanlar **koçan** (stub) olarak bırakılır. Arayüz erken netleşir; ama tüm sistem bitmeden gerçekçi test zordur.

Aşağıdan yukarı

Önce donanımı gizleyen katman, kesmeler, zamanlayıcı, sonra iş parçacıkları... Her katman bittiğinde **test** edilir. Tanenbaum gerçekleştirmede bu yolu önerir; tasarım ise yukarıdan aşağı düşünülmelidir.

Eş Zamanlı ve Eş Zamansız İletişim



- **Eş zamanlı:** programlaması kolay, akış sıralı okunur; ama her istek için bir iş parçacığı bekler.
- Mikro çekirdeklerde IPC genelde eş zamanlıdır (L4, sel4): tampon gerekmez, çok hızlıdır.
- **Eş zamansız:** tek iş parçacığı binlerce isteği yürütebilir; ama yarış, hata yönetimi ve tamponlama zorlaşır.
- Linux `io_uring` : gönderim (SQ) ve tamamlanma (CQ) halkaları, sistem çağırısı olmadan bile istek gönderilebilir.

Donanımı Gizleme

CPU türüne bağlı koşullu derleme

```
#include "config.h"
void init(void)
{
    #if (CPU == X86_64)
        /* x86-64 başlatma kodu */
    #endif
    #if (CPU == ARM64)
        /* ARM64 başlatma kodu */
    #endif
    #if (CPU == RISCV64)
        /* RISC-V başlatma kodu */
    #endif
}
```

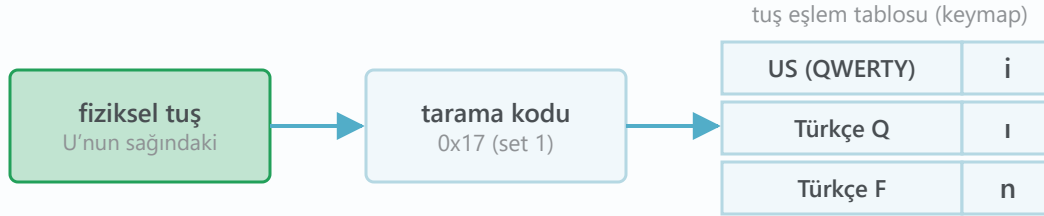
Sözcük uzunluğuna bağlı derleme

```
#include "config.h"
#include <stdint.h>
#if (WORD_LENGTH == 32)
    typedef int Register;
#elif (WORD_LENGTH == 64)
    typedef long Register;
#else /* bugün: taşınabilir */
    typedef uintptr_t Register;
#endif
Register R0, R1, R2, R3;
```

Donanıma bağımlı kodu **az sayıda dosyada toplamak** taşınabilirliğin anahtarıdır: Windows'ta **HAL** (Hardware Abstraction Layer), Linux'ta `arch/x86`, `arch/arm64`, `arch/riscv` dizinleri. Yeni bir işlemciye taşırken yalnız bu kısım yeniden yazılır. Koşullu derleme ölçülü kullanılmalı: her yere dağılan `#ifdef` kodu okunmaz hâle getirir.

Kodlar Tanenbaum'daki Pentium/UltraSPARC örneğinin güncellenmiş hâlidir.

Dolaylılık (Indirection)

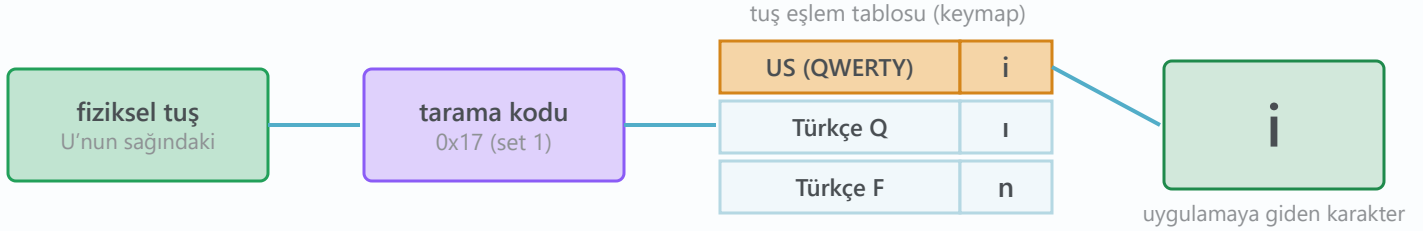


Kullanıcı U'nun sağındaki tuşa basar. Klavye karakter değil, tuşun **konumunu** bildirir.

Adım 1 / 4

“Bilgisayar bilimindeki her sorun, bir dolaylılık düzeyi daha ile çözülebilir.” (David Wheeler).
İşletim sistemlerinde: sanal bellek (sayfa tablosu), dosya tanımlayıcıları, sembolik bağlar, i-node, aygıt sınıfları.

Dolaylılık (Indirection)

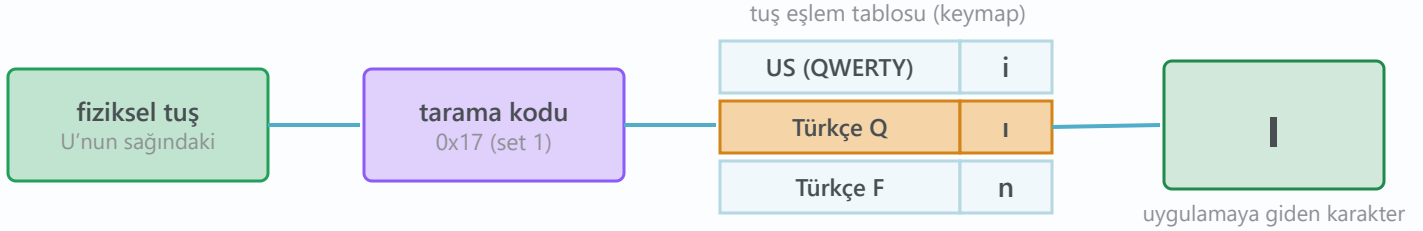


Klavye 0x17 tarama kodunu gönderir (USB klavyelerde HID kullanım kodu; sürücü ortak bir koda çevirir). ABD eşlemesiyle bu **i** olur.

Adım 2 / 4

“Bilgisayar bilimindeki her sorun, bir dolaylılık düzeyi daha ile çözülebilir.” (David Wheeler). İşletim sistemlerinde: sanal bellek (sayfa tablosu), dosya tanımlayıcıları, sembolik bağlar, i-node, aygıt sınıfları.

Dolaylılık (Indirection)

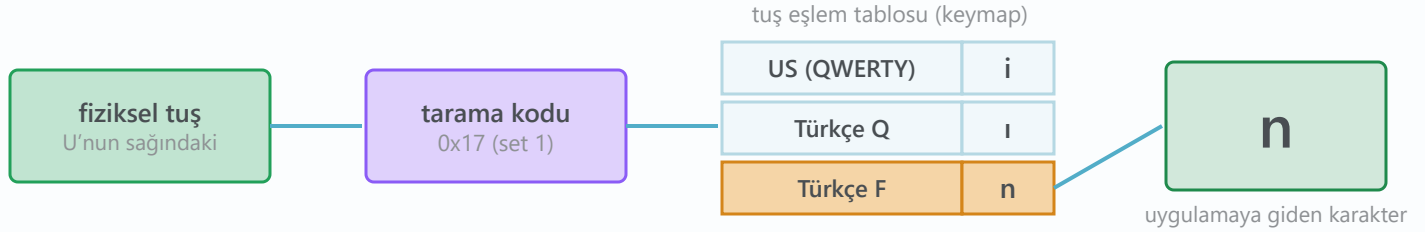


Yalnızca **tablo** Türkçe Q ile değiştirilir: aynı tuş artık **ı** üretir. Donanım ve uygulama değişmedi.

Adım 3 / 4

“Bilgisayar bilimindeki her sorun, bir dolaylılık düzeyi daha ile çözülebilir.” (David Wheeler).
İşletim sistemlerinde: sanal bellek (sayfa tablosu), dosya tanımlayıcıları, sembolik bağlar, i-node, aygıt sınıfları.

Dolaylılık (Indirection)



Türkçe F eşlemesinde aynı tuş **n** olur. Ek bir adım (dolaylılık) esneklik sağlar; bedeli bir tablo okuması.

Adım 4 / 4

“Bilgisayar bilimindeki her sorun, bir dolaylılık düzeyi daha ile çözülebilir.” (David Wheeler).
İşletim sistemlerinde: sanal bellek (sayfa tablosu), dosya tanımlayıcıları, sembolik bağlar, i-node, aygıt sınıfları.

Yeniden Kullanım ve Yeniden Girişlilik

Yeniden kullanılabilirlik

Aynı kodu farklı yerlerde kullanmak: tek bir **biteşlem** (bitmap) modülü boş disk bloklarını, bellek çerçevelerini ve PID'leri izleyebilir. Linux'ta ortak `list.h`, kırmızı-siyah ağaç, xarray.

Yeniden girişlilik (reentrancy)

Bir kod parçası aynı anda **iki kez** çalışabilmelidir: biri yürütülürken kesme gelip aynı fonksiyonu çağırabilir ya da başka bir çekirdek aynı kodu çalıştırır. Kural: paylaşılan statik değişken yok, gerekiyorsa kilit.

```
char *strtok(char *s, const char *d)
{
    static char *last; /* paylaşılan durum! */
    ...
}

/* yeniden girişli sürüm: durum çağıranda */
char *strtok_r(char *s, const char *d,
               char **saveptr);
```

`strtok` içindeki `static` değişken, iki iş parçası ya da bir kesme işleyicisi aynı anda kullanınca bozulur.

strtok ve Yeniden Girişlilik: Kod İzleme

```
▶ char *my_strtok(char *s, char ayr) {  
▶     static char *kalan; /* herkes paylaşır! */  
3     if (s) kalan = s;  
4     if (!kalan || !*kalan) return NULL;  
5     char *bas = kalan;  
6     while (*kalan && *kalan != ayr) kalan++;  
7     if (*kalan) *kalan++ = '\\0';  
8     return bas;  
9 }  
10 /* main:   my_strtok(a, ',') a = "ali,veli" */  
11 /* kesme:  my_strtok(b, ';') b = "x;y"      */  
12 /* main:   my_strtok(NULL, ',')             */
```

a

a	l	i	,	v	e	l	i	\\0
---	---	---	---	---	---	---	---	-----

static kalan: **NULL**

main

a = "ali,veli"

Ana program virgülle ayrılmış adları ayıklıyor. Fonksiyonun durumu tek bir static işaretçide: kalan, bir sonraki çağrının nereden süreceğini tutar.

Adım 1 / 6

DEĞİŞKENLER

kalan = NULL

ÇIKTI



strtok ve Yeniden Girişlilik: Kod İzleme

```
1 char *my_strtok(char *s, char ayr) {  
2     static char *kalan; /* herkes paylaşır! */  
▶   if (s) kalan = s;  
4   if (!kalan || !*kalan) return NULL;  
▶   char *bas = kalan;  
▶   while (*kalan && *kalan != ayr) kalan++;  
▶   if (*kalan) *kalan++ = '\\0';  
▶   return bas;  
9 }  
10 /* main:   my_strtok(a, ',') a = "ali,veli" */  
11 /* kesme:  my_strtok(b, ';') b = "x;y"      */  
12 /* main:   my_strtok(NULL, ',')             */
```

a

static kalan: **a dizisinin 4. hücresi**

main

1. çağrı → "ali"

İlk çağrı: kalan = a. Virgüle kadar ilerler, virgülün yerine \\0 yazar, kalan "veli"nin başını gösterir.

Dönüş: **ali**.

Adım 2 / 6

DEĞİŞKENLER

s = a

kalan = a+4

dönüş = ali

ÇIKTI

main: ali

strtok ve Yeniden Girişlilik: Kod İzleme

```
1 char *my_strtok(char *s, char ayr) {  
2     static char *kalan; /* herkes paylaşır! */  
3     if (s) kalan = s;  
4     if (!kalan || !*kalan) return NULL;  
5     char *bas = kalan;  
6     while (*kalan && *kalan != ayr) kalan++;  
7     if (*kalan) *kalan++ = '\\0';  
8     return bas;  
9 }  
  
▶ /* main: my_strtok(a, ',') a = "ali,veli" */  
11 /* kesme: my_strtok(b, ';') b = "x;y" */  
12 /* main: my_strtok(NULL, ',') */
```



static kalan: **a dizisinin 4. hücresi**

kesme geldi

b = "x;y"

Tam bu anda bir kesme gelir; işleyici de aynı fonksiyonla kendi dizgisini (b = "x;y") ayıklamaya başlar.

Adım 3 / 6

DEĞİŞKENLER

kalan = a+4

ÇIKTI

main: ali

strtok ve Yeniden Girişlilik: Kod İzleme

```

1 char *my_strtok(char *s, char ayr) {
2     static char *kalan; /* herkes paylaşır! */
3     ▶ if (s) kalan = s;
4     if (!kalan || !*kalan) return NULL;
5     ▶ char *bas = kalan;
6     ▶ while (*kalan && *kalan != ayr) kalan++;
7     ▶ if (*kalan) *kalan++ = '\0';
8     ▶ return bas;
9 }
10 /* main: my_strtok(a, ',') a = "ali,veli" */
11 /* kesme: my_strtok(b, ',') b = "x;y" */
12 /* main: my_strtok(NULL, ',') */

```

a

a	l	i	\0	v	e	l	i	\0
---	---	---	----	---	---	---	---	----

b

x	\0	y	\0
---	----	---	----

static kalan:

b dizisinin 2. hücresi

kesme işleyicisi

işleyici → "x"

İşleyicinin çağırısı `s = b` ile static **kalan**'ın üzerine yazar: artık b'nin içini ("y") gösteriyor. main'in kaldığı yer kayboldu.

Adım 4 / 6

DEĞİŞKENLER

s = b

kalan = b+2

dönüş = x

ÇIKTI

```

main: ali
kesme: x

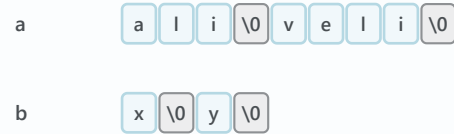
```

strtok ve Yeniden Girişlilik: Kod İzleme

```

1 char *my_strtok(char *s, char ayr) {
2     static char *kalan; /* herkes paylaşır! */
3     if (s) kalan = s;
4     if (!kalan || !*kalan) return NULL;
5     char *bas = kalan;
6     while (*kalan && *kalan != ayr) kalan++;
7     if (*kalan) *kalan++ = '\0';
8     return bas;
9 }
10 /* main: my_strtok(a, ',') a = "ali,veli" */
11 /* kesme: my_strtok(b, ';') b = "x;y" */
12 /* main: my_strtok(NULL, ',') */

```



static kalan: **b dizisinin 3. hücresi** main sürüyor
 main bekliyordu: "veli" — aldı: "y"
 main kaldığı yerden sürdürmek için NULL geçer
 ama kalan b'yi gösterir: dönüş **y**, beklenen "veli"
 değil. Hata nadiren ve zamanlamaya bağlı
 görünür.

Adım 5 / 6

DEĞİŞKENLER

s = NULL

kalan = b+3

dönüş = y

ÇIKTI

```

main: ali
kesme: x
main: y (hatalı)

```

strtok ve Yeniden Girişlilik: Kod İzleme

```

1 char *my_strtok(char *s, char ayr) {
2     static char *kalan; /* herkes paylaşır! */
3     if (!s) kalan = s;
4     if (!kalan || !*kalan) return NULL;
5     char *bas = kalan;
6     while (*kalan && *kalan != ayr) kalan++;
7     if (*kalan) *kalan++ = '\0';
8     return bas;
9 }
10 /* main: my_strtok(a, ',') a = "ali,veli" */
11 /* kesme: my_strtok(b, ';') b = "x;y" */
12 /* main: my_strtok(NULL, ',') */

```

a

a	l	i	\0	v	e	l	i	\0
---	---	---	----	---	---	---	---	----

b

x	\0	y	\0
---	----	---	----

static kalan: **b dizisinin 3. hücresi**

çözüm: strtok_r(s, d, &kalan)

Kök neden: çağrılar arası durum static (paylaşılan) değişkende. Yeniden girişli strtok_r durumu çağırının verdiği işaretçide tutar; her çağırın kendi kalan'ına sahip olur.

Adım 6 / 6

DEĞİŞKENLER

kalan = **b+3**

ÇIKTI

```

main: ali
kesme: x
main: y (hatalı)

```

Kaba Kuvvet ve Önce Hata Denetimi

Kaba kuvvet (brute force)

- Basit ama “yavaş” görünen çözüm çoğu zaman yeterlidir: 50 girişlik bir tabloda doğrusal arama, hash tablosundan **hızlı ve hatasız** olabilir.
- Karmaşık algoritma ancak ölçüm gerekli gösterirse: n büyükse veya kod sıcak yoldaysa.
- Sıralama: küçük dizilerde ekleme sıralaması, büyüklerde birleştirme/öbek sıralaması.

Önce hataları denetle

Kötü: kaynak alındıktan sonra denetim

```
lock(&tbl_lock);
buf = kmalloc(n, GFP_KERNEL);
if (n > MAX || !access_ok(u, n)) {
    kfree(buf);
    unlock(&tbl_lock);
    return -EINVAL;
}
```

İyi: önce denetim, sonra kaynak

```
if (n > MAX || !access_ok(u, n))
    return -EINVAL; /* önce denetle */
buf = kmalloc(n, GFP_KERNEL);
if (!buf)
    return -ENOMEM;
lock(&tbl_lock);
...
```

Parametreleri kilit ve bellek almadan önce doğrulamak, hata yolundaki geri alma kodunu (ve unutilan `unlock` hatalarını) ortadan kaldırır.

Performans: Neden Yavaş, Neyi Hızlandırmalı?

Neden yavaş?

Özellik şişkinliği: her sürüm yeni özellik ekler, eskisini çıkaramaz. Geriye uyumluluk katmanları, güvenlik önlemleri (ör. Spectre/Meltdown yamaları sistem çağrısı maliyetini artırdı), genellik uğruna eklenen dolaylılık.

Neler optimize edilmeli?

Yalnızca **ölçümle** darboğaz olduğu gösterilen kısımlar. "İyi yeterlidir": %5'lik kazanç için kodu iki kat karmaşıktırmak çoğu zaman değmez.

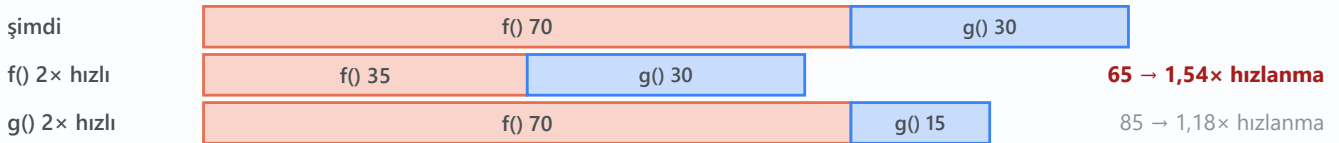
Önce doğru, sonra hızlı

"Erken optimizasyon tüm kötülüklerin anasıdır." (D. Knuth). Ölçme araçları: **perf**, eBPF tabanlı **bpftrace**, alev grafikleri.

Klasik teknikler

Uzay-zaman ödünleşimi · önbellekleme · ipuçları · yerelliği kullanmak · yaygın durumu hızlandırmak. Sonraki slaytlarda her biri örnekleniyor.

Donanım on yıllarda binlerce kat hızlandı, ama hissedilen hız aynı oranda artmadı: artışın önemli kısmını yazılımın büyümesi tüketti.



Uzay-Zaman Ödünleşimi: (a) Döngüyle Bit Sayma

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
5     for (i = 0; i < BYTE_SIZE; i++)
6         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



count = 0

Örnek bayt 0xB7 (183). Döngüyle her bit tek tek sınanacak; sayaç sıfırdan başlar.

Adım 1 / 9

DEĞİŞKENLER

byte = 0xB7

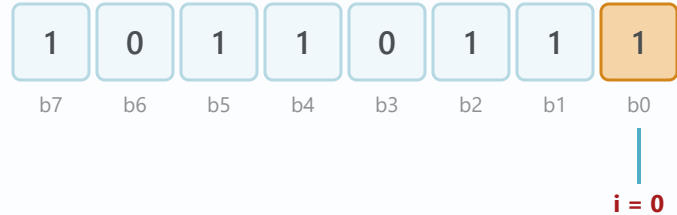
i = -

count = 0

Uzay-Zaman Ödünleşimi: (a) Döngüyle Bit Sayma

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
5     for (i = 0; i < BYTE_SIZE; i++)
6         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



count = 1

(byte >> 0) & 1 = 1: sayaç artar.

Adım 2 / 9

DEĞİŞKENLER

byte = 0xB7

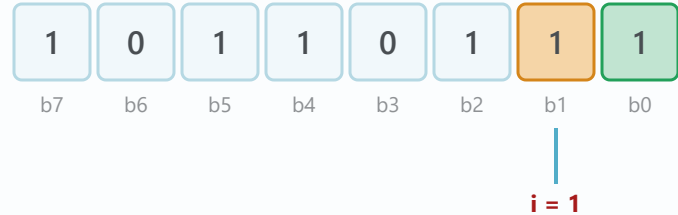
i = 0

count = 1

Uzay-Zaman Ödünleşimi: (a) Döngüyle Bit Sayma

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
5     for (i = 0; i < BYTE_SIZE; i++)
6         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



count = 2

Kaydır, maskele, karşılaştır: her bit için birkaç komut ve bir dal.

Adım 3 / 9

DEĞİŞKENLER

byte = 0xB7

i = 1

count = 2

Uzay-Zaman Ödünleşimi: (a) Döngüyle Bit Sayma

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
5     for (i = 0; i < BYTE_SIZE; i++)
6         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



i = 2

count = 3

Bit 2 de 1.

Adım 4 / 9

DEĞİŞKENLER

byte = 0xB7

i = 2

count = 3

Uzay-Zaman Ödünleşimi: (a) Döngüyle Bit Sayma

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
5     for (i = 0; i < BYTE_SIZE; i++)
6         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



i = 3

count = 3

Bit 3 sıfır: dal alınmaz, sayaç değişmez.

Adım 5 / 9

DEĞİŞKENLER

byte = 0xB7

i = 3

count = 3

Uzay-Zaman Ödünleşimi: (a) Döngüyle Bit Sayma

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
5     for (i = 0; i < BYTE_SIZE; i++)
6         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



i = 5

count = 5

Bit 4 ve 5 de 1.

Adım 6 / 9

DEĞİŞKENLER

byte = 0xB7

i = 5

count = 5

Uzay-Zaman Ödünleşimi: (a) Döngüyle Bit Sayma

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
5     for (i = 0; i < BYTE_SIZE; i++)
6         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



i = 6

count = 5

Bit 6 sıfır.

Adım 7 / 9

DEĞİŞKENLER

byte = 0xB7

i = 6

count = 5

Uzay-Zaman Ödünleşimi: (a) Döngüyle Bit Sayma

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
5     for (i = 0; i < BYTE_SIZE; i++)
6         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



count = 6

Son bit 1. Döngü 8 kez döndü.

Adım 8 / 9

DEĞİŞKENLER

byte = 0xB7

i = 7

count = 6

Uzay-Zaman Ödünleşimi: (a) Döngüyle Bit Sayma

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
5     for (i = 0; i < BYTE_SIZE; i++)
6         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



count = 6

Sonuç 6. Bellek yok denecek kadar az, ama bayt başına yaklaşık 8 tur × birkaç komut. Daha hızlısı için **yer harcıyıp zaman kazanmak** gerekir.

Adım 9 / 9

DEĞİŞKENLER

byte = 0xB7

i = 8

count = 6

PERFORMANS

Uzay-Zaman Ödünleşimi: (b) Makro ve (c) Tablo

```
/* (b) makro: döngü yok */
#define bit_count(b) ((b&1) + ((b>>1)&1) + \
    ((b>>2)&1) + ... + ((b>>7)&1))

/* (c) tablo: 256 bayt */
char bits[256] =
    {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...};
#define bit_count(b) (int) bits[b]
```

Yöntem	Bellek	Bayt başına iş
(a) döngü	~0	8 tur: kaydır, maskele, dallan
(b) makro	~0	~24 komut, dal yok
(c) 256'lık tablo	256 B	1 bellek okuması
16'lık tablo (yarım bayt)	16 B	2 okuma + toplama
POPCNT komutu	0	1 komut (<code>__builtin_popcount</code>)

bits[0..31] (Python ile hesaplandı: `bin(i).count("1")`)

0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	2	3	2	3	3	4	2	3	3	4	3	4	4	5
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

bits[0xB7] = bits[183] = 6

Yarım Bayt Tablosuyla Bit Sayma: Kod İzleme

```
1 char yarim[16];          /* 16 bayt */
2 void tablo_kur(void) {
3     ▶ yarim[0] = 0;
4     for (int i = 1; i < 16; i++)
5         yarim[i] = (i & 1) + yarim[i >> 1];
6 }
7 int bit_say(unsigned char b) {
8     return yarim[b & 0x0F] + yarim[b >> 4];
9 }
```



256 baytlık tablo yerine 16 baytlık tablo: her 4 bitlik değerin bir sayısı. 0'da hiç bit yok.

Adım 1 / 7

DEĞİŞKENLER

yarim[0] = 0

ÇIKTI



Yarım Bayt Tablosuyla Bit Sayma: Kod İzleme

```
1 char yarim[16];          /* 16 bayt */
2 void tablo_kur(void) {
3     yarim[0] = 0;
4     for (int i = 1; i < 16; i++)
5         yarim[i] = (i & 1) + yarim[i >> 1];
6 }
7 int bit_say(unsigned char b) {
8     return yarim[b & 0x0F] + yarim[b >> 4];
9 }
```



$i = 1 \dots 3$: i 'nin bit sayısı = son biti + $(i \gg 1)$ 'in bit sayısı; $i \gg 1$ her zaman daha önce doldurulmuştur.
Örn. $\text{yarim}[3] = 1 + \text{yarim}[1] = 2$.

Adım 2 / 7

DEĞİŞKENLER

$i = 3$

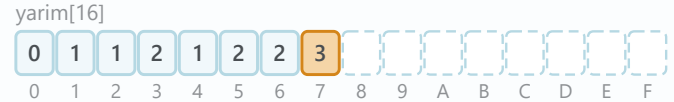
$\text{yarim}[3] = 2$

ÇIKTI



Yarım Bayt Tablosuyla Bit Sayma: Kod İzleme

```
1 char yarim[16];           /* 16 bayt */
2 void tablo_kur(void) {
3     yarim[0] = 0;
4     for (int i = 1; i < 16; i++)
5         yarim[i] = (i & 1) + yarim[i >> 1];
6 }
7 int bit_say(unsigned char b) {
8     return yarim[b & 0x0F] + yarim[b >> 4];
9 }
```



i = 4 ... 7: i'nin bit sayısı = son biti + (i >> 1)'in bit sayısı; i >> 1 her zaman daha önce doldurulmuştur.
Örn. yarim[7] = 1 + yarim[3] = 3.

Adım 3 / 7

DEĞİŞKENLER

i = 7

yarim[7] = 3

ÇIKTI



Yarım Bayt Tablosuyla Bit Sayma: Kod İzleme

```
1 char yarim[16];          /* 16 bayt */
2 void tablo_kur(void) {
3     yarim[0] = 0;
4     for (int i = 1; i < 16; i++)
5         yarim[i] = (i & 1) + yarim[i >> 1];
6 }
7 int bit_say(unsigned char b) {
8     return yarim[b & 0x0F] + yarim[b >> 4];
9 }
```

yarim[16]

0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

i = 8 ... 15: i'nin bit sayısı = son biti + (i >> 1)'in bit sayısı; i >> 1 her zaman daha önce doldurulmuştur.
Örn. yarim[15] = 1 + yarim[7] = 4.

Adım 4 / 7

DEĞİŞKENLER

i = 15

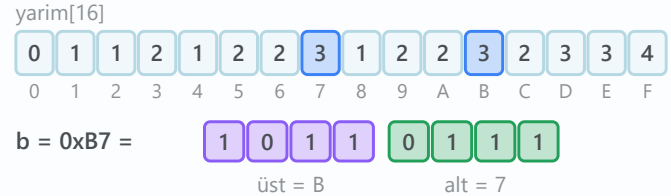
yarim[15] = 4

ÇIKTI



Yarım Bayt Tablosuyla Bit Sayma: Kod İzleme

```
1 char yarim[16];          /* 16 bayt */
2 void tablo_kur(void) {
3     yarim[0] = 0;
4     for (int i = 1; i < 16; i++)
5         yarim[i] = (i & 1) + yarim[i >> 1];
6 }
7 ▶ int bit_say(unsigned char b) {
8     return yarim[b & 0x0F] + yarim[b >> 4];
9 }
```



bit_say(0xB7): bayt iki yarıya ayrılır: alt 7 = 0111, üst B = 1011. Döngü yok, dal yok.

Adım 5 / 7

DEĞİŞKENLER

b = 0xB7

alt = 7

üst = B

ÇIKTI



Yarım Bayt Tablosuyla Bit Sayma: Kod İzleme

```

1 char yarim[16];          /* 16 bayt */
2 void tablo_kur(void) {
3     yarim[0] = 0;
4     for (int i = 1; i < 16; i++)
5         yarim[i] = (i & 1) + yarim[i >> 1];
6 }
7 int bit_say(unsigned char b) {
8     return yarim[b & 0x0F] + yarim[b >> 4];
9 }

```



b = 0xB7 =

1	0	1	1	0	1	1	1
üst = B				alt = 7			

yarim[7] + yarim[B] = 3 + 3 = 6

İki bellek okuması ve bir toplama: 3 + 3 = 6.

Döngülü sürüm aynı bayt için 8 tur dönmüştü.

Adım 6 / 7

DEĞİŞKENLER

b = 0xB7

yarim[7] = 3

yarim[B] = 3

ÇIKTI

bit_say(0xB7) = 6

Yarım Bayt Tablosuyla Bit Sayma: Kod İzleme

```
1 char yarim[16];          /* 16 bayt */
2 void tablo_kur(void) {
3     yarim[0] = 0;
4     for (int i = 1; i < 16; i++)
5         yarim[i] = (i & 1) + yarim[i >> 1];
6 }
7
8 ▶ int bit_say(unsigned char b) {
9     return yarim[b & 0x0F] + yarim[b >> 4];
10 }
```



b = 0x5C =

0	1	0	1	1	1	0	0
üst = 5				alt = C			

yarim[C] + yarim[5] = 2 + 2 = 4

bit_say(0x5C): yarim[C] + yarim[5] = 2 + 2 = **4**. 16 bayt bellek, 2 okuma: 256'lık tablo ile döngü arasında bir orta nokta.

Adım 7 / 7

DEĞİŞKENLER

b = 0x5C

yarim[C] = 2

yarim[5] = 2

ÇIKTI

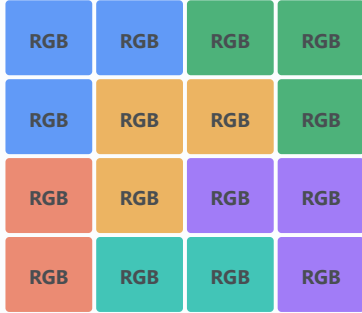
bit_say(0xB7) = 6

bit_say(0x5C) = 4

PERFORMANS

Uzay-Zaman Ödünleşimi: Renk Paleti

(a) 24 bit/piksel



3 bayt/piksel

(b) 8 bit indis/piksel

0	0	1	1
0	2	2	1
3	2	4	4
3	5	5	4

1 bayt/piksel

(c) renk paleti

0	#3b82f6
1	#22a05a
2	#e9a23b
3	#e76f51
4	#8b5cf6
5	#14b8a6

256 giriş × 3 bayt

1024 × 768 görüntü	Boyut (bayt)	Oran
(a) 24 bit/piksel	2.359.296 (2,25 MiB)	1
(b)+(c) 8 bit indis + palet	786.432 + 768 = 787.200	≈ 1/3

Yer yaklaşık 3 kat azalır; bedeli her piksel gösterilirken bir **tablo bakışı** (zaman) ve en fazla 256 renk sınırıdır. GIF ve PNG-8 bu yöntemi kullanır (GIF ayrıca LZW ile sıkıştırır). Boyutlar Python ile hesaplanmıştır.

Önbelleğe Almak (Caching)

Yol → i-node önbelleği

Yol	i-node
/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/ast/books	92
/usr/bal	45
/usr/bal/paper.ps	85

- Önbellek yokken `/usr/ast/mbox` 'ı bulmak: kök i-node'u, kök dizini, /usr i-node'u, /usr dizini, /usr/ast i-node'u, /usr/ast dizini → **6 disk okuması** (+ dosyanın i-node'u).
- Önbellekte tam yol varsa: **bir bellek araması**, sıfır disk erişimi.
- Önbellekte yalnızca `/usr/ast` varsa, arama oradan devam eder: kısmi isabet de kazançtır.
- Linux: dizin girdisi önbelleği (**dcache**), i-node önbelleği ve sayfa önbelleği. Boş RAM'in "dolu" görünmesinin nedeni budur.
- Önbelleğin bedeli: tutarlılık. Dosya silinir veya yeniden adlandırılırsa kayıt geçersiz kılınmalıdır.

Genel kural: aynı sonucu tekrar tekrar hesaplıyorsanız, sonucu **saklayın**. İşletim sistemindeki önbellekler: TLB, CPU önbellekleri, sayfa önbelleği, dcache, ARP ve DNS önbellekleri.

Yol Önbellesi ile Ad Çözümleme: Kod İzleme

```
1 /* bil[]: "usr","bal","tez.tex"; n = 3 */
2 int cozumle(char *bil[], int n) {
3     int k = n, ino = KOK, okuma = 0;
4     while (k > 0 && !onbellekte(bil, k, &ino))
5         k--;
6     for (int j = k; j < n; j++) {
7         ino = dizinde_ara(ino, bil[j]);
8         okuma += 2;
9         onbellege_ekle(bil, j + 1, ino);
10    }
11    return ino;
12 }
```

yol: **usr** **bal** **tez.tex**

disk okuma: 0

önbellek (yol → i-node)

/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/bal	45
/usr/bal/paper.ps	85

/usr/bal/tez.tex önbellette yok

Önbellette beş yol var. /usr/bal/tez.tex aranıyor:
tam yol önbellette yok.

Adım 1 / 7

DEĞİŞKENLER

k = 3

ino = KOK

okuma = 0

ÇIKTI



Yol Önbelleği ile Ad Çözümleme: Kod İzleme

```
1 /* bil[]: "usr","bal","tez.tex"; n = 3 */
2 int cozumle(char *bil[], int n) {
3     int k = n, ino = KOK, okuma = 0;
4     while (k > 0 && !onbellekte(bil, k, &ino))
5         k--;
6     for (int j = k; j < n; j++) {
7         ino = dizinde_ara(ino, bil[j]);
8         okuma += 2;
9         onbellege_ekle(bil, j + 1, ino);
10    }
11    return ino;
12 }
```

yol: **usr** **bal** tez.tex

disk okuma: 0

önbellek (yol → i-node)

/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/bal	45
/usr/bal/paper.ps	85

kısmi isabet: /usr/bal → 45

Son bileşen atılır: /usr/bal önbellekte, ino = **45**.
Kökten başlamaya gerek kalmadı: iki bileşen bedava.

Adım 2 / 7

DEĞİŞKENLER

k = 2**ino = 45**

okuma = 0

ÇIKTI



Yol Önbellesi ile Ad Çözümleme: Kod İzleme

```

1 /* bil[]: "usr","bal","tez.tex"; n = 3 */
2 int cozumle(char *bil[], int n) {
3     int k = n, ino = KOK, okuma = 0;
4     while (k > 0 && !onbellette(bil, k, &ino))
5         k--; /* en uzun önek */
6     for (int j = k; j < n; j++) {
7         ino = dizinde_ara(ino, bil[j]);
8         okuma += 2; /* i-node + blok */
9         onbellege_ekle(bil, j + 1, ino);
10    }
11    return ino;
12 }

```

yol: **usr** **bal** **tez.tex**

disk okuma: 2

önbellet (yol → i-node)

/usr	6	/usr/bal/tez.tex	131
/usr/ast	26		
/usr/ast/mbox	60		
/usr/bal	45		
/usr/bal/paper.ps	85		

diskten: /usr/bal dizini → tez.tex

Kalan tek bileşen diskten: 45'in i-node'u ve dizin bloğu okunur, tez.tex → **131**. Sonuç önbellete eklenir.

Adım 3 / 7

DEĞİŞKENLER

j = 2

ino = 131

okuma = 2

ÇIKTI

/usr/bal/tez.tex → 131 (2 okuma)

Yol Önbellesi ile Ad Çözümleme: Kod İzleme

```
1 /* bil[]: "usr","bal","tez.tex"; n = 3 */
2 int cozumle(char *bil[], int n) {
3     int k = n, ino = KOK, okuma = 0;
4     while (k > 0 && !onbellekte(bil, k, &ino))
5         k--; /* en uzun önek */
6     for (int j = k; j < n; j++) {
7         ino = dizinde_ara(ino, bil[j]);
8         okuma += 2; /* i-node + blok */
9         onbellege_ekle(bil, j + 1, ino);
10    }
11    return ino;
12 }
```

yol: **usr** **bal** **tez.tex**

disk okuma: 0

önbellek (yol → i-node)

/usr	6	/usr/bal/tez.tex	131
/usr/ast	26		
/usr/ast/mbox	60		
/usr/bal	45		
/usr/bal/paper.ps	85		

tam isabet: hiç disk erişimi yok

Aynı yol yeniden açılınca tam yol önbellette:
döngüler hiç dönmez, **0** disk okuması.

Adım 4 / 7

DEĞİŞKENLER

k = 3**ino = 131****okuma = 0**

ÇIKTI

/usr/bal/tez.tex → 131 (2 okuma)**/usr/bal/tez.tex → 131 (0 okuma)**

Yol Önbelleği ile Ad Çözümleme: Kod İzleme

```

1 /* bil[]: "usr","bal","tez.tex"; n = 3 */
2 int cozumle(char *bil[], int n) {
3     int k = n, ino = KOK, okuma = 0;
4     while (k > 0 && !onbellekte(bil, k, &ino))
5         k--;
6     for (int j = k; j < n; j++) {
7         ino = dizinde_ara(ino, bil[j]);
8         okuma += 2;
9         onbellege_ekle(bil, j + 1, ino);
10    }
11    return ino;
12 }

```

yol: **etc** **passwd**

disk okuma: 0

önbellek (yol → i-node)

/usr	6	/usr/bal/tez.tex	131
/usr/ast	26		
/usr/ast/mbox	60		
/usr/bal	45		
/usr/bal/paper.ps	85		

/etc/passwd ve /etc önbellekte yok

/etc/passwd: ne tam yol ne /etc önbellekte; k 0'a iner, çözüm kök i-node'undan (KOK = 1) başlar.

Adım 5 / 7

DEĞİŞKENLER

k = 0

ino = KOK

okuma = 0

ÇIKTI

/usr/bal/tez.tex → 131 (2 okuma)

/usr/bal/tez.tex → 131 (0 okuma)

Yol Önbelleği ile Ad Çözümleme: Kod İzleme

```

1 /* bil[]: "usr","bal","tez.tex"; n = 3 */
2 int cozumle(char *bil[], int n) {
3     int k = n, ino = KOK, okuma = 0;
4     while (k > 0 && !onbellekte(bil, k, &ino))
5         k--; /* en uzun önek */
6     for (int j = k; j < n; j++) {
7         ino = dizinde_ara(ino, bil[j]);
8         okuma += 2; /* i-node + blok */
9         onbellege_ekle(bil, j + 1, ino);
10    }
11    return ino;
12 }

```

yol: **etc** passwd

disk okuma: 2

önbellek (yol → i-node)

/usr	6	/usr/bal/tez.tex	131
/usr/ast	26	/etc	14
/usr/ast/mbox	60		
/usr/bal	45		
/usr/bal/paper.ps	85		

kök dizinde etc → 14

Kök i-node'u ve kök dizin bloğu okunur: etc → **14**; /etc önbelleğe girer.

Adım 6 / 7

DEĞİŞKENLER

j = 0

ino = 14

okuma = 2

ÇIKTI

```

/usr/bal/tez.tex → 131 (2 okuma)
/usr/bal/tez.tex → 131 (0 okuma)

```

Yol Önbellesi ile Ad Çözümleme: Kod İzleme

```

1 /* bil[]: "usr","bal","tez.tex"; n = 3 */
2 int cozumle(char *bil[], int n) {
3     int k = n, ino = KOK, okuma = 0;
4     while (k > 0 && !onbellette(bil, k, &ino))
5         k--; /* en uzun önek */
6     for (int j = k; j < n; j++) {
7         ino = dizinde_ara(ino, bil[j]);
8         okuma += 2; /* i-node + blok */
9         onbellege_ekle(bil, j + 1, ino);
10    }
11    return ino;
12 }

```

yol: **etc** **passwd**

disk okuma: 4

önbellek (yol → i-node)

/usr	6	/usr/bal/tez.tex	131
/usr/ast	26	/etc	14
/usr/ast/mbox	60	/etc/passwd	212
/usr/bal	45		
/usr/bal/paper.ps	85		

etc dizininde passwd → 212

/etc i-node'u ve dizin bloğu: passwd → **212**.

Önbelleksiz yolda bileşen başına 2 okuma:
toplam 4. Linux'ta bu önbellek dcache'tir.

Adım 7 / 7

DEĞİŞKENLER

j = 1

ino = 212

okuma = 4

ÇIKTI

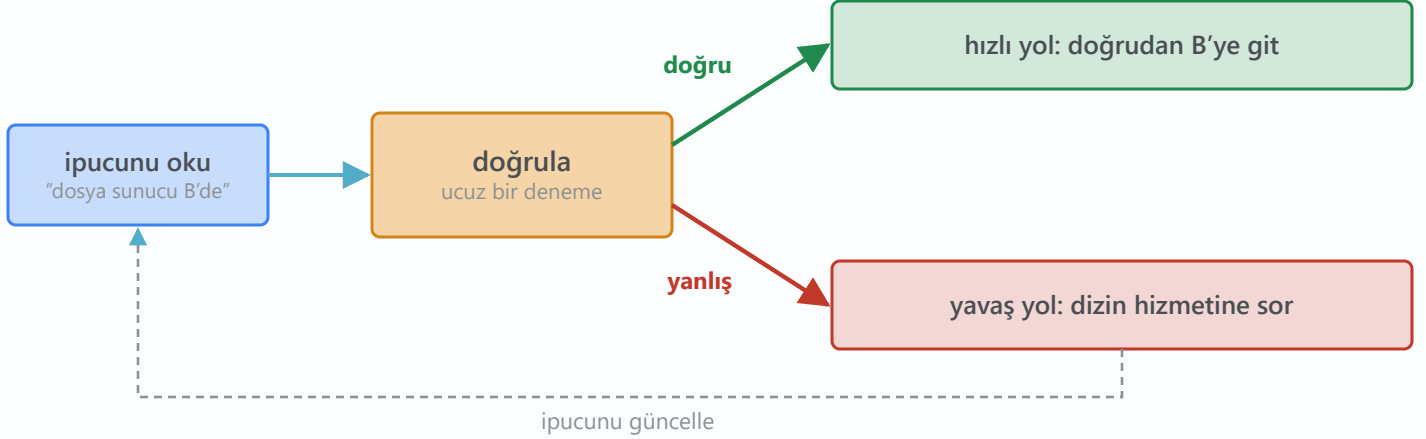
```

/usr/bal/tez.tex → 131 (2 okuma)
/usr/bal/tez.tex → 131 (0 okuma)
/etc/passwd → 212 (4 okuma)

```

PERFORMANS

İpuçları (Hints)



Önbellekten farkı

Önbellekteki değer her zaman doğru olmalıdır. **İpucu yanlış olabilir**; bu yüzden geçersiz kılma gerekmez, ama kullanmadan önce ucuz bir doğrulama şarttır.

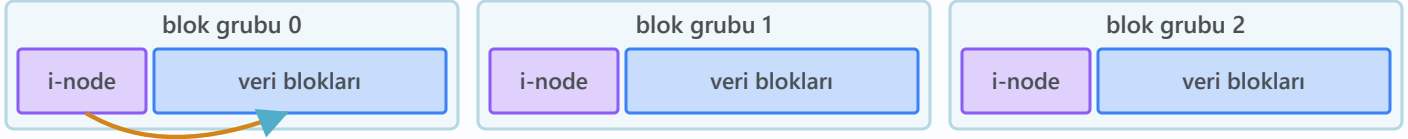
Örnekler

Dağıtık dosya sisteminde "dosya muhtemelen sunucu B'de"; web'de yönlendirme önbelleği; dal tahmini; uygulamadan çekirdeğe `advise(MADV_SEQUENTIAL)` ile verilen erişim ipucu.

PERFORMANS

Yerelliği Kullanmak (Exploiting Locality)

ext4 disk düzeni: blok grupları



i-node ile verisi aynı grupta: kısa arama

CPU yakınlığı (affinity)



iş parçacığı aynı çekirdekte kalırsa önbelleği “sıcak” kalır

- **Zamansal yerellik:** yakın zamanda kullanılan tekrar kullanılır (önbelleğin temeli). **Uzamsal yerellik:** yakınındaki de kullanılır (önden okuma, büyük sayfalar).
- Birlikte kullanılan verileri yan yana koymak: FFS silindir grupları, ext4 blok grupları; NUMA'da belleği iş parçacığının düğümünden ayırmak.

Yaygın Durumu Hızlandırmak

kullanıcı alanı

İş parçacığı A
kilidi istiyor

kilit sözcüğü
değer = 0

İş parçacığı B
çalışıyor

çekirdek

futex bekleme kuyruğu
boş

kilit boş (0)

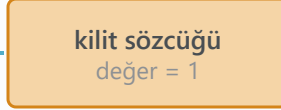
Kilitlerin büyük çoğunluğu **çekişmesiz** alınır. Yaygın durum bu ise, hızlandırılması gereken de budur.

Adım 1 / 4

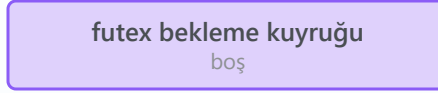
Genel ilke (Lampson): **yaygın durumu hızlı, nadir durumu doğru yap**. Diğer örnekler: sistem çağrılarının hızlı yolu, vDSO ile çekirdeğe girmeden `clock_gettime`, TLB isabeti.

Yaygın Durumu Hızlandırmak

kullanıcı alanı



çekirdek



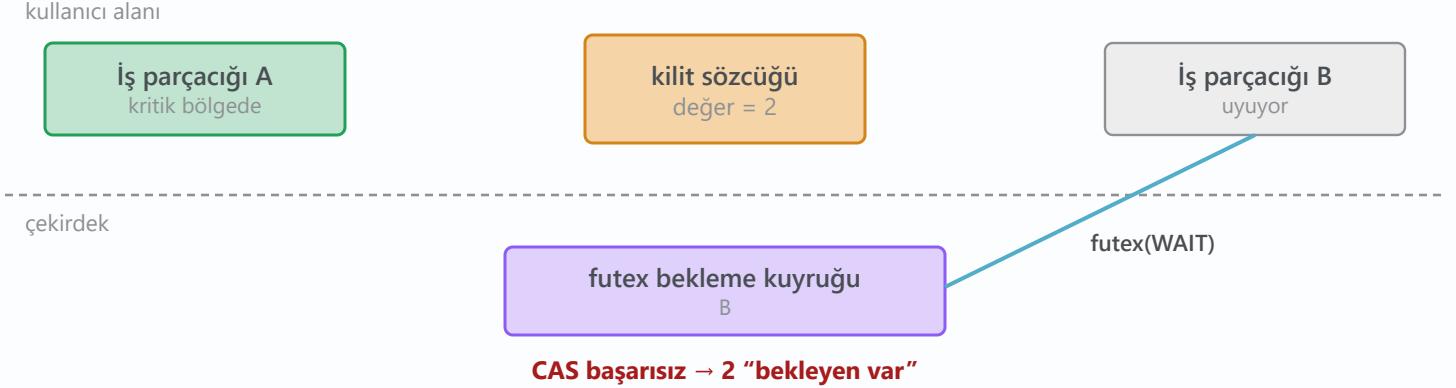
CAS 0→1 başarılı: sistem çağrısı yok

A, atomik karşılaştır-değiştir (CAS) ile 0'ı 1 yapar. **Çekirdeğe hiç girilmez:** birkaç nanosaniye.

Adım 2 / 4

Genel ilke (Lampson): **yaygın durumu hızlı, nadir durumu doğru yap.** Diğer örnekler: sistem çağrılarının hızlı yolu, vDSO ile çekirdeğe girmeden `clock_gettime`, TLB isabeti.

Yaygın Durumu Hızlandırmak

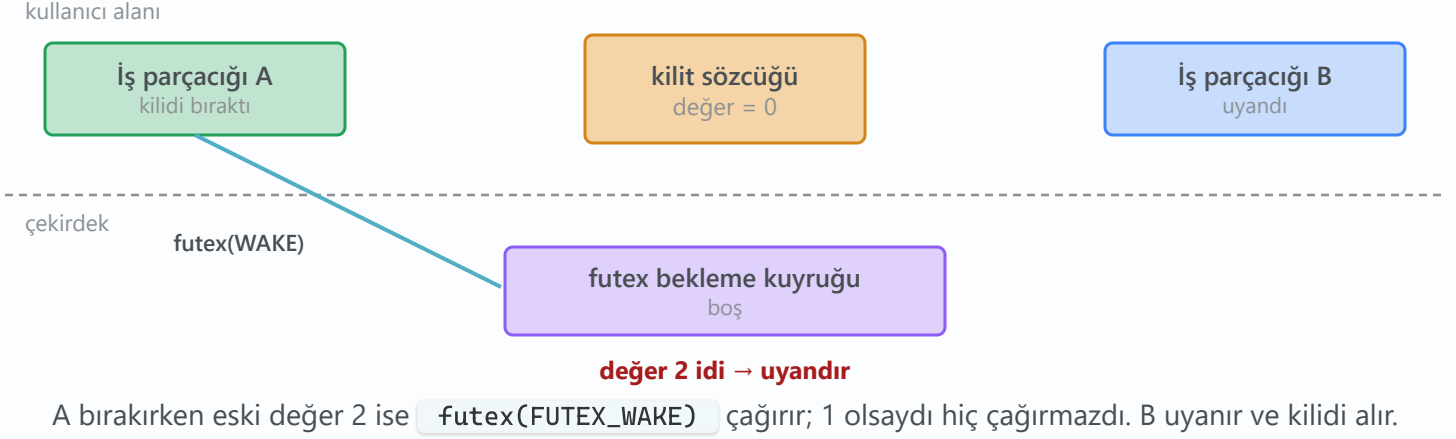


Nadir durum: B de kilidi ister, CAS başarısız olur. Değeri 2 yapar ve `futex(FUTEX_WAIT)` ile çekirdekte uyur. Yavaş yol yalnızca burada.

Adım 3 / 4

Genel ilke (Lampson): **yaygın durumu hızlı, nadir durumu doğru yap**. Diğer örnekler: sistem çağrılarının hızlı yolu, vDSO ile çekirdeğe girmeden `clock_gettime`, TLB isabeti.

Yaygın Durumu Hızlandırmak



Adım 4 / 4

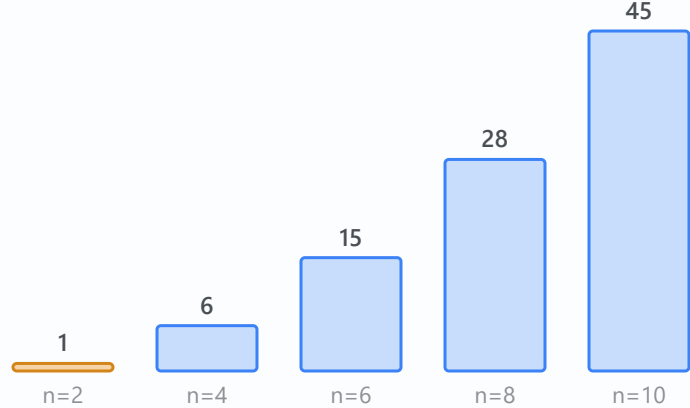
Genel ilke (Lampson): **yaygın durumu hızlı, nadir durumu doğru yap**. Diğer örnekler: sistem çağrılarının hızlı yolu, vDSO ile çekirdeğe girmeden `clock_gettime`, TLB isabeti.

Efsanevi Adam-Ay (The Mythical Man-Month)



n = 2 kişi → 1 iletişim yolu

"Adam-ay" (man-month) yanıltıcı bir birimdir: iş, kişiler arasında bölünebilir varsayar. 2 kişi arasında yalnızca **1** iletişim yolu var.

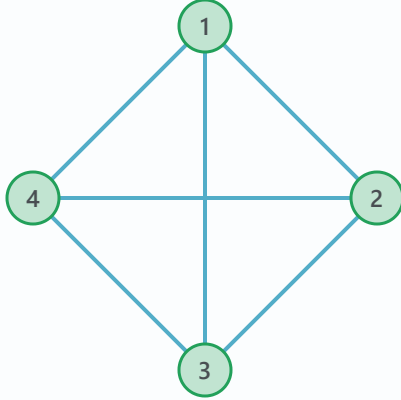


$yol\ sayısı = n(n-1)/2$

Adım 1 / 4

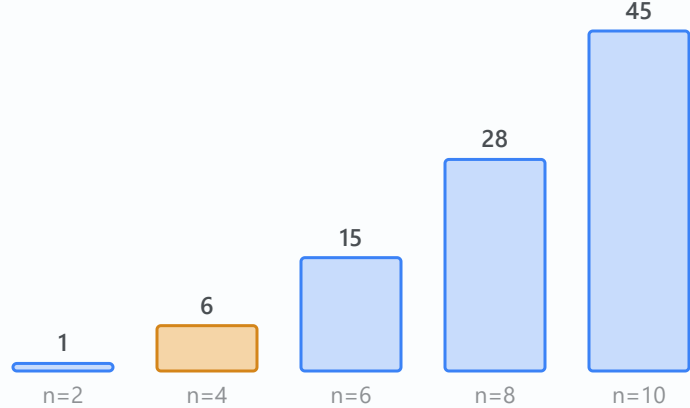
Brooks'un OS/360 deneyiminden önerdiği zaman dağılımı: 1/3 planlama, 1/6 kodlama, 1/4 modül testi, 1/4 sistem testi. Yani kod yazmak işin yalnızca altıda biridir.

Efsanevi Adam-Ay (The Mythical Man-Month)



n = 4 kişi → 6 iletişim yolu

4 kişide **6** yol. Her yeni kişi herkesle eşgüdüm kurmak zorunda; toplantı, belge ve arayüz anlaşmaları artar.

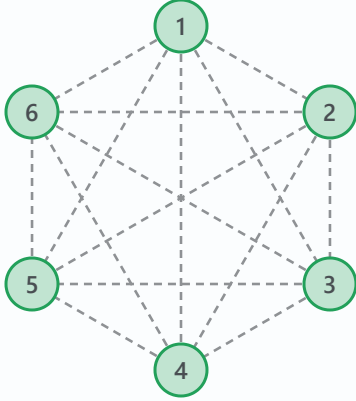


$$\text{yol sayısı} = n(n-1)/2$$

Adım 2 / 4

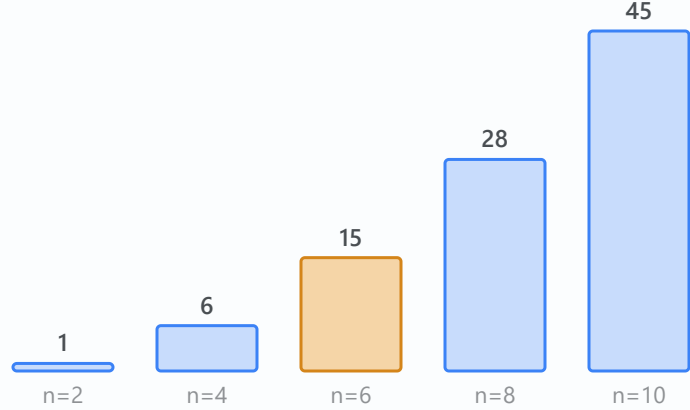
Brooks'un OS/360 deneyiminden önerdiği zaman dağılımı: 1/3 planlama, 1/6 kodlama, 1/4 modül testi, 1/4 sistem testi. Yani kod yazmak işin yalnızca altıda biridir.

Efsanevi Adam-Ay (The Mythical Man-Month)



n = 6 kişi → 15 iletişim yolu

6 kişide **15** yol. Gecikmiş projeye eklenen kişiyi eğitmek, mevcut ekibin zamanını da tüketir.

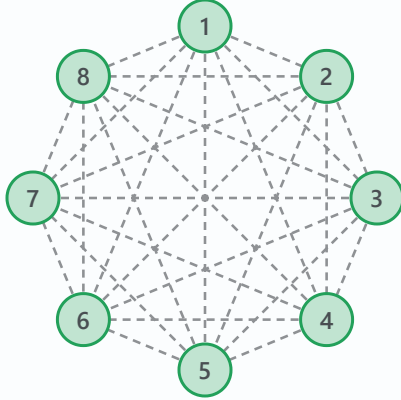


$$\text{yol sayısı} = n(n-1)/2$$

Adım 3 / 4

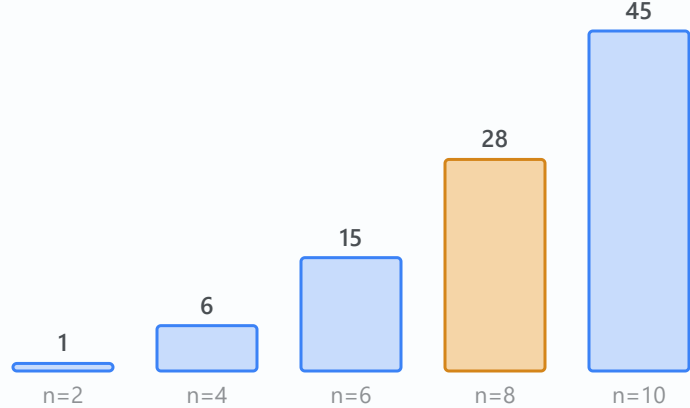
Brooks'un OS/360 deneyiminden önerdiği zaman dağılımı: 1/3 planlama, 1/6 kodlama, 1/4 modül testi, 1/4 sistem testi. Yani kod yazmak işin yalnızca altıda biridir.

Efsanevi Adam-Ay (The Mythical Man-Month)



n = 8 kişi → 28 iletişim yolu

8 kişide **28** yol. **Brooks yasası**: "Gecikmiş bir yazılım projesine eleman eklemek, onu daha da geciktirir."

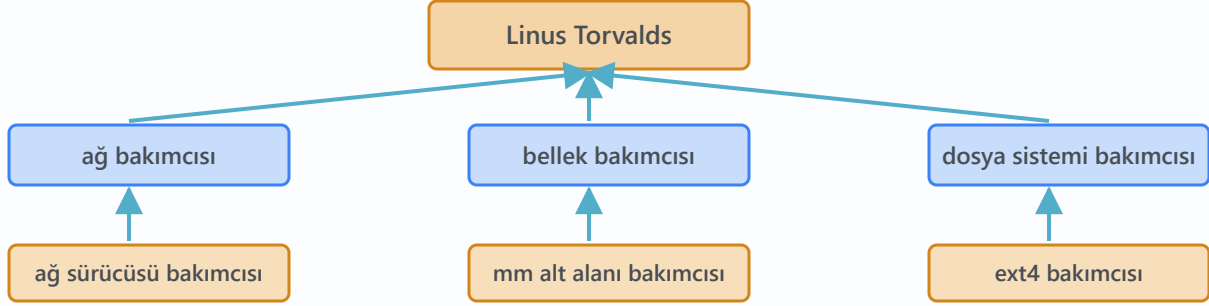


yol sayısı = $n(n-1)/2$

Adım 4 / 4

Brooks'un OS/360 deneyiminden önerdiği zaman dağılımı: 1/3 planlama, 1/6 kodlama, 1/4 modül testi, 1/4 sistem testi. Yani kod yazmak işin yalnızca altıda biridir.

Takım Yapısı, Deneyim ve “Gümüş Kurşun Yok”



binlerce katkıcının yamaları yukarı akar (yılıda ~4-5 bin geliştirici)

Takım yapısı

Mills/Brooks'un **cerrah ekibi**: baş programcı tasarlar ve kodlar, diğerleri destekler. Kavramsal bütünlük birkaç kişide kalır.

Deneyimin rolü

İyi programcı ortalamadan kat kat verimlidir. **“Birini atmayı planlayın”**; ikinci sistem etkisine dikkat.

Gümüş kurşun yok

Brooks (1986): araçlar yalnız **arızı** karmaşıklığı azaltır, **özel** karmaşıklık kalır. Yapay zekâ destekli kodlama da tasarım sorumluluğunu kaldırmaz.

İS Kodunu Test Etmek Neden Zor?

Neden zor?

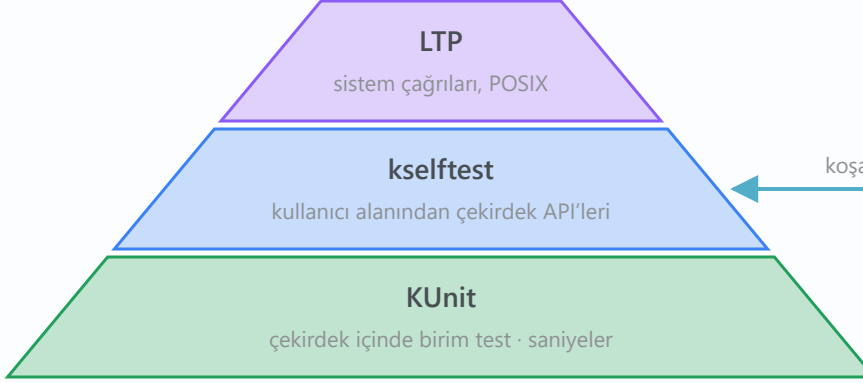
- **Ayrıcalıklı mod:** hatayı yakalayacak bir üst katman yoktur; çekirdekte tek bir hatalı işaretçi oops/panic ile **tüm sistemi**, test düzeneğiyle birlikte düşürür.
- **Eş zamanlılık:** kesmeler, ön alım ve çok çekirdek sayısız iç içe geçme üretir; hata yalnız nadir bir zamanlamada ortaya çıkar.
- **Belirlenimsizlik:** "heisenbug"lar her çalıştırmada tekrarlanmaz; `printk` eklemek bile zamanlamayı değiştirip hatayı saklayabilir.
- **Çeşitlilik:** binlerce sürücü, birçok mimari (x86-64, arm64, RISC-V) ve on binlerce Kconfig seçeneği; tüm bileşimler denenemez.
- **Uzun ömürlü durum:** sızıntılar ve sayaç taşmaları ancak günlerce çalıştıktan sonra görünür.

Test türleri

Tür	Ne sınar? (çekirdek örneği)
Birim	Tek işlev ya da modül, yalıtılmış hâlde (KUnit)
Entegrasyon	Alt sistemlerin birlikte çalışması, kullanıcı alanı arayüzü (kseltest)
Sistem	Tüm İS, gerçek ya da sanal makinede (LTP, açılış testleri)
Regresyon	Düzeltilen hata geri geldi mi? Her değişiklikte yeniden koşulur (KernelCI)
Stres / yük	Aşırı yük, uzun süre, kaynak kıtlığı (stress-ng, rcutorture)

İS'de başarısız bir test çoğu zaman makinenin kendisini kaybetmek demektir; bu yüzden testler sanal makinelerde, zaman aşımı ve otomatik yeniden başlatmayla koşulur.

Çekirdek Test Piramidi



↑ gerçekçilik ve maliyet artar · ↓ hız ve test sayısı artar

KernelCI
sürekli entegrasyon

- birçok ağaç ve dal
- çok sayıda mimari, Kconfig
- QEMU + gerçek kart lab'ları
- KUnit, kseltest, LTP koşar
- gerilemeyi otomatik bisect

benzerleri: 0-day bot, syzbot

KUnit

Testler çekirdekle birlikte derlenir, iç işlevleri doğrudan çağırır; UML ya da QEMU'da saniyeler içinde koşar:

`kunit.py run` . Çıktı KTAP biçimindedir.

kseltest

`tools/testing/selftests` altında; çalışan çekirdeğin sistem çağrılarını, cgroup, bpf, ağ gibi arayüzlerini dışarıdan sınar: `make kseltest` .

LTP

Linux Test Project: binlerce test; sistem çağrısı anlamı, POSIX uyumu, dosya sistemi ve stres senaryoları. Dağıtımlar sürüm öncesi kullanır.

Dinamik Denetleyiciler: KASAN, KCSAN, lockdep

Araç	Yakaladığı hata	Nasıl çalışır?
KASAN CONFIG_KASAN	use-after-free; öbek, yığın ve global dizi taşması; çift serbest bırakma	Derleyici her bellek erişiminin önüne denetim ekler; gölge bellek (her 8 bayta 1 bayt) hangi baytların geçerli olduğunu tutar. Rapor, ayırma ve serbest bırakma yığın izlerini de verir.
KCSAN CONFIG_KCSAN	Veri yarışı: en az biri yazma olan, eş zamanlı ve kilsiz erişim	Rastgele erişimlerde gözlem noktası kurup kısa süre bekler; bu arada başka bir çekirdek aynı adrese dokunursa raporlar.
UBSAN CONFIG_UBSAN	Tanımsız davranış: işaretli taşma, sıfıra bölme, geçersiz kaydırma, dizi sınırı	Derleyicinin eklediği çalışma zamanı denetimleri.
lockdep CONFIG_PROVE_LOCKING	Kilit sırası ihlali (AB-BA), kesme bağlamında hatalı kilit kullanımı	Kilit sınıfları arasındaki “önce-sonra” çizgesini tutar; çevrim görünce kilitlenme gerçekleşmeden uyarır.
kmemleak CONFIG_DEBUG_KMEMLEAK	Bellek sızıntısı	Çöp toplayıcı gibi belleği tarar; hiçbir işaretçinin göstermediği ayrılmış blokları raporlar.

BUG: KASAN: slab-use-after-free in mydrv_release+0x54/0x90
Read of size 8 at addr ffff888012a4c010 by task cat/1432

Maliyet: KASAN çekirdeği birkaç kat yavaşlatır ve belleğin yaklaşık 1/8’ini gölgeye ayırır; KCSAN ve lockdep de belirgin yük getirir. Bu yüzden **geliştirme, test ve CI çekirdeklerinde** açılır. Üretimde örneklemeli ve düşük maliyetli **KFENCE** kullanılabilir.

KASAN Gölge Bellek Denetimi: Kod İzleme

► /* 8 baytlık bölge başına 1 gölge bayt:

► 0: hepsi geçerli, 1..7: ilk k bayt,

► < 0: kırmızı bölge ya da serbest */

```
4 void kasan_denetle(void *a, int boy) {
5     int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
6                     + GOLGE_OFS);
7     if (s == 0) return; /* hızlı yol */
8     if (((uintptr_t)a & 7) + boy - 1 >= s)
9         rapor(a, s);
10 }
11 /* p = kmalloc(13); p[3]; p[12]; p[13]; */
12 /* kfree(p); p[0]; */
```

bellek p[0..23]



gölge

0x00

0x05

0xfc (-4)

p = kmalloc(13): 13 bayt geçerli

kmalloc(13) 16 baytlık bir yuva verir. Gölge: ilk 8 bayt tam geçerli (0), sonraki bölgenin ilk 5 baytı geçerli (13 - 8), arkası kırmızı bölge (0xfc).

Adım 1 / 6

DEĞİŞKENLER

gölge = 00 05 fc

ÇIKTI



KASAN Gölge Bellek Denetimi: Kod İzleme

```
1 /* 8 baytlık bölge başına 1 gölge bayt:
2    0: hepsi geçerli, 1..7: ilk k bayt,
3    < 0: kırmızı bölge ya da serbest */
4 void kasan_denetle(void *a, int boy) {
5     ▶ int8_t s = *((int8_t *)(((uintptr_t)a >> 3)
6     ▶                               + GOLGE_OFS);
7     ▶ if (s == 0) return;           /* hızlı yol */
8     if (((uintptr_t)a & 7) + boy - 1 >= s)
9         rapor(a, s);
10 }
11 /* p = kmalloc(13); p[3]; p[12]; p[13]; */
12 /* kfree(p); p[0]; */
```

bellek p[0..23]



gölge

0x00

0x05

0xfc (-4)

p[3]: gölge 0 → hızlı yol

p[3]: gölge baytı $p+3 \gg 3 = 0$. bölgeden okunur: 0, denetim tek karşılaştırmayla biter.

Adım 2 / 6

DEĞİŞKENLER

a = p+3

s = 0

ÇIKTI



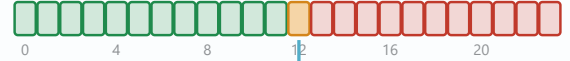
KASAN Gölge Bellek Denetimi: Kod İzleme

```

1 /* 8 baytlık bölge başına 1 gölge bayt:
2    0: hepsi geçerli, 1..7: ilk k bayt,
3    < 0: kırmızı bölge ya da serbest */
4 void kasan_denetle(void *a, int boy) {
5     ▶ int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
6     ▶                               + GOLGE_OFS);
7     ▶ if (s == 0) return;           /* hızlı yol */
8     ▶ if (((uintptr_t)a & 7) + boy - 1 >= s)
9         rapor(a, s);
10 }
11 /* p = kmalloc(13); p[3]; p[12]; p[13]; */
12 /* kfree(p); p[0]; */

```

bellek p[0..23]



gölge

0x00

0x05

0xfc (-4)

p[12]: 4 ≥ 5? hayır → geçerli

p[12]: gölge 5; bölge içi konum 12 & 7 = 4.
4 ≥ 5 değil: son geçerli bayt, erişim serbest.

Adım 3 / 6

DEĞİŞKENLER

a = p+12

s = 5

a&7 = 4

ÇIKTI

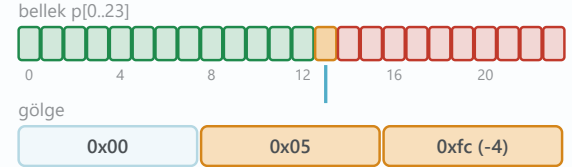


KASAN Gölge Bellek Denetimi: Kod İzleme

```

1 /* 8 baytlık bölge başına 1 gölge bayt:
2    0: hepsi geçerli, 1..7: ilk k bayt,
3    < 0: kırmızı bölge ya da serbest */
4 void kasan_denetle(void *a, int boy) {
5     int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
6                     + GOLGE_OFS);
7     if (s == 0) return; /* hızlı yol */
8     if (((uintptr_t)a & 7) + boy - 1 >= s)
9         rapor(a, s);
10 }
11 /* p = kmalloc(13); p[3]; p[12]; p[13]; */
12 /* kfree(p); p[0]; */

```



p[13]: $5 \geq 5 \rightarrow$ rapor

p[13]: $5 \geq 5$: dizinin bir bayt ötesi. Gerçek çekirdekte bu bayt başka bir nesneye ait olabilirdi ve hata sessiz kalırdı; KASAN ilk geçersiz erişimde raporlar.

Adım 4 / 6

DEĞİŞKENLER

a = p+13

s = 5

a&7 = 5

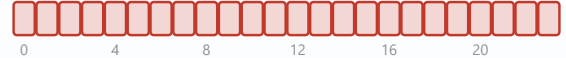
ÇIKTI

BUG: KASAN: slab-out-of-bounds p+13

KASAN Gölge Bellek Denetimi: Kod İzleme

```
1 /* 8 baytlık bölge başına 1 gölge bayt:
2    0: hepsi geçerli, 1..7: ilk k bayt,
3    < 0: kırmızı bölge ya da serbest */
4 void kasan_denetle(void *a, int boy) {
5     int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
6                       + GOLGE_OFS);
7     if (s == 0) return; /* hızlı yol */
8     if (((uintptr_t)a & 7) + boy - 1 >= s)
9         rapor(a, s);
10 }
11 /* p = kmalloc(13); p[3]; p[12]; p[13]; */
12 ► /* kfree(p); p[0]; */
```

bellek p[0..23]



gölge

0xfb (-5)

0xfb (-5)

0xfc (-4)

kfree(p): gölge 0xfb (serbest)

kfree(p): nesnenin gölgesi 0xfb (serbest) olarak işaretlenir; yuva bir süre karantineda bekletilir ki hemen yeniden kullanılıp hata gizlenmesin.

Adım 5 / 6

DEĞİŞKENLER

gölge = fb fb fc

ÇIKTI

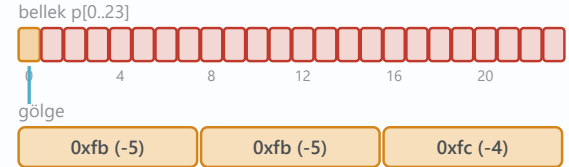
BUG: KASAN: slab-out-of-bounds p+13

KASAN Gölge Bellek Denetimi: Kod İzleme

```

1 /* 8 baytlık bölge başına 1 gölge bayt:
2    0: hepsi geçerli, 1..7: ilk k bayt,
3    < 0: kırmızı bölge ya da serbest */
4 void kasan_denetle(void *a, int boy) {
5     ▶ int8_t s = *((int8_t *)(((uintptr_t)a >> 3)
6                 + GOLGE_OFS);
7     ▶ if (s == 0) return; /* hızlı yol */
8     ▶ if (((uintptr_t)a & 7) + boy - 1 >= s)
9         ▶ rapor(a, s);
10 }
11 /* p = kmalloc(13); p[3]; p[12]; p[13]; */
12 /* kfree(p); p[0]; */

```



p[0]: s = -5 < 0 → rapor

p[0]: s = -5. Negatif gölge her konumda “≥ s” koşulunu sağlar: **serbest bırakma sonrası kullanım** raporu, ayırma ve serbest bırakma yığın izleriyle birlikte.

Adım 6 / 6

DEĞİŞKENLER

a = p+0

s = -5

ÇIKTI

BUG: KASAN: slab-out-of-bounds p+13

BUG: KASAN: use-after-free p+0

lockdep Kilit Sırası Çizgesi: Kod İzleme

```
1 void lock_acquire(task *t, int L) {  
2     for (int i = 0; i < t->n; i++) {  
3         int H = t->tutulan[i];  
4         if (yol_var(L, H)) /* L → ... → H */  
5             uyar("olası kilitletme", H, L);  
6         kenar_ekle(H, L); /* H, L'den önce */  
7     }  
8     t->tutulan[t->n++] = L;  
9 }  
10 void lock_release(task *t) { t->n--; }  
▶ /* T1: A,B   T2: B,C   T3: C,A */
```

tutulan:
—

lockdep kilit örneklerini değil sınıflarını izler ve “şu kilit tutulurken bu kilit alındı” ilişkisini bir çizgede biriktirir. Başta çizge boş.

Adım 1 / 7

DEĞİŞKENLER

kenar = 0

ÇIKTI



lockdep Kilit Sırası Çizgesi: Kod İzleme

```
▶ void lock_acquire(task *t, int L) {  
2   for (int i = 0; i < t->n; i++) {  
3       int H = t->tutulan[i];  
4       if (yol_var(L, H)) /* L → ... → H */  
5           uyar("olası kilitlenme", H, L);  
6       kenar_ekle(H, L); /* H, L'den önce */  
7   }  
▶   t->tutulan[t->n++] = L;  
9 }  
10 void lock_release(task *t) { t->n--; }  
11 /* T1: A,B   T2: B,C   T3: C,A */
```



tutulan:
A



T1 A'yı alır: tutulan başka kilit yok, döngü dönmez; A tutulanlar listesine eklenir.

Adım 2 / 7

DEĞİŞKENLER

görev = T1

L = A

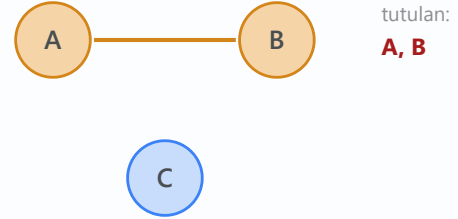
n = 1

ÇIKTI



lockdep Kilit Sırası Çizgesi: Kod İzleme

```
1 void lock_acquire(task *t, int L) {  
    ▶ for (int i = 0; i < t->n; i++) {  
        ▶ int H = t->tutulan[i];  
        ▶ if (yol_var(L, H)) /* L → ... → H */  
            5 uyar("olası kilitlenme", H, L);  
        ▶ kenar_ekle(H, L); /* H, L'den önce */  
    7 }  
    ▶ t->tutulan[t->n++] = L;  
    9 }  
10 void lock_release(task *t) { t->n--; }  
11 /* T1: A,B   T2: B,C   T3: C,A */
```



T1 A'yı tutarken B'yi alır: B'den A'ya yol yok, uyarı yok. Kenar **A → B** eklenir. Sonra iki kilit bırakılır.

Adım 3 / 7

DEĞİŞKENLER

görev = T1

L = B

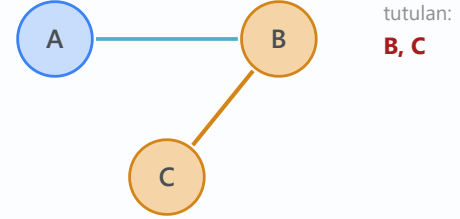
H = A

ÇIKTI



lockdep Kilit Sırası Çizgesi: Kod İzleme

```
1 void lock_acquire(task *t, int L) {  
    ▶ for (int i = 0; i < t->n; i++) {  
    ▶     int H = t->tutulan[i];  
    ▶     if (yol_var(L, H)) /* L → ... → H */  
5         uyar("olası kilitlenme", H, L);  
    ▶     kenar_ekle(H, L); /* H, L'den önce */  
7     }  
    ▶     t->tutulan[t->n++] = L;  
9 }  
10 void lock_release(task *t) { t->n--; }  
11 /* T1: A,B   T2: B,C   T3: C,A */
```



Daha sonra T2 B'yi tutarken C'yi alır: kenar **B → C**.
Çizgede artık A → B → C sırası var.

Adım 4 / 7

DEĞİŞKENLER

görev = T2

L = C

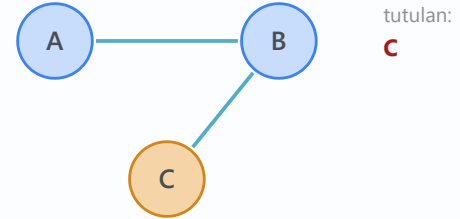
H = B

ÇIKTI



lockdep Kilit Sırası Çizgesi: Kod İzleme

```
► void lock_acquire(task *t, int L) {  
2   for (int i = 0; i < t->n; i++) {  
3       int H = t->tutulan[i];  
4       if (yol_var(L, H)) /* L → ... → H */  
5           uyar("olası kilitlenme", H, L);  
6       kenar_ekle(H, L); /* H, L'den önce */  
7   }  
►   t->tutulan[t->n++] = L;  
9 }  
10 void lock_release(task *t) { t->n--; }  
11 /* T1: A,B   T2: B,C   T3: C,A */
```



Başka bir anda T3 önce C'yi alır; tek kilit, sorun yok.

Adım 5 / 7

DEĞİŞKENLER

görev = T3

L = C

n = 1

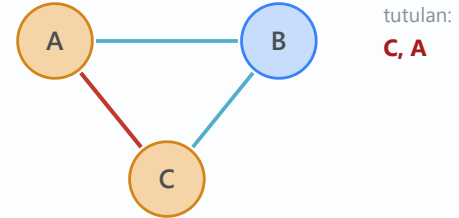
ÇIKTI

lockdep Kilit Sırası Çizgesi: Kod İzleme

```

1 void lock_acquire(task *t, int L) {
    for (int i = 0; i < t->n; i++) {
        int H = t->tutulan[i];
        if (yol_var(L, H)) /* L → ... → H */
            uyar("olası kilitletlenme", H, L);
        kenar_ekle(H, L); /* H, L'den önce */
    }
    t->tutulan[t->n++] = L;
}
10 void lock_release(task *t) { t->n--; }
11 /* T1: A,B   T2: B,C   T3: C,A */

```



A → B → C → A: çevrim!

T3 C'yi tutarken A'yı ister: A'dan C'ye yol var (A → B → C). Yeni kenar C → A bir **çevrim** kapatır: lockdep uyarır.

Adım 6 / 7

DEĞİŞKENLER

görev = T3

L = A

H = C

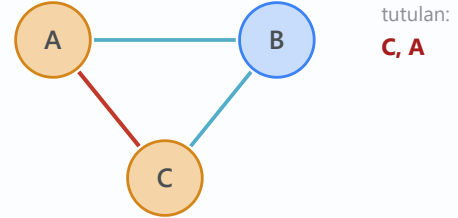
ÇIKTI

WARNING: possible circular locking dependency

C → A, ama A → B → C

lockdep Kilit Sırası Çizgesi: Kod İzleme

```
1 void lock_acquire(task *t, int L) {  
2     for (int i = 0; i < t->n; i++) {  
3         int H = t->tutulan[i];  
4         if (yol_var(L, H)) /* L → ... → H */  
5             uyar("olası kilitlenme", H, L);  
6         kenar_ekle(H, L); /* H, L'den önce */  
7     }  
8     t->tutulan[t->n++] = L;  
9 }  
10 void lock_release(task *t) { t->n--; }  
11 /* T1: A,B   T2: B,C   T3: C,A */
```



uyarı: kilitlenme olmadan önce

Üç görev hiç aynı anda çalışmadı, gerçek kilitlenme olmadı. Ama T1, T2 ve T3 bir gün aynı anda koşarsa her biri bir sonrakinin kilidini bekler. lockdep bunu tek bir çalıştırmada, olmadan önce gösterir.

Adım 7 / 7

DEĞİŞKENLER

görev = T3

L = A

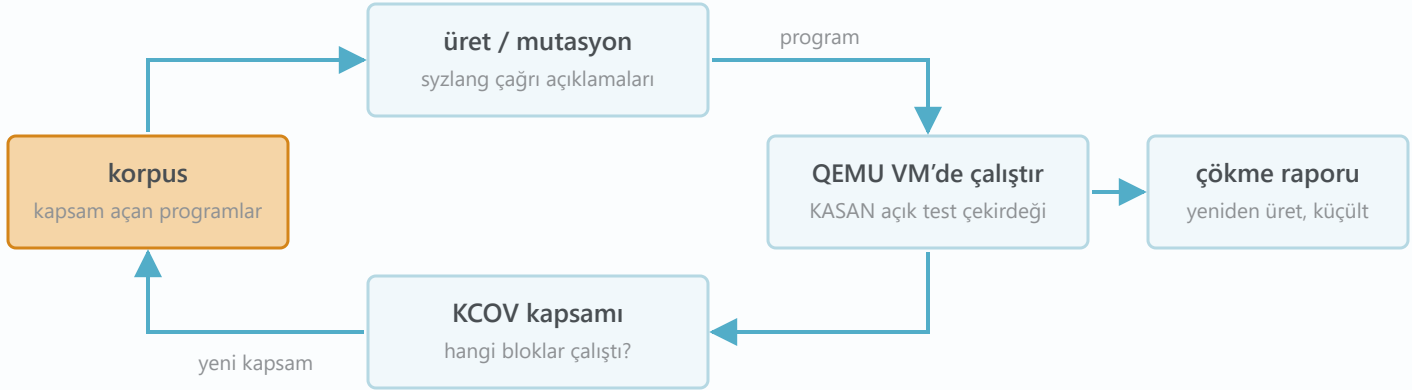
H = C

ÇIKTI

WARNING: possible circular locking dependency

C → A, ama A → B → C

Fuzzing: syzkaller Döngüsü

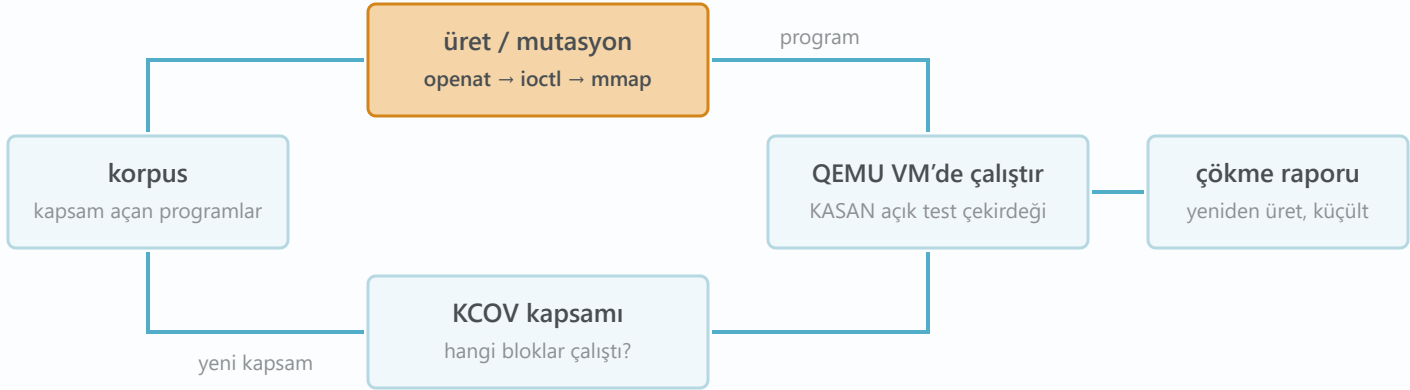


syzkaller sistem çağrılarının imzalarını **syzlang** açıklamalarından bilir: hangi argüman dosya tanıtıcısı, hangisi bayrak, hangisi işaretçi. Korpus başta boştur ya da tohum programlardan oluşur.

Adım 1 / 6

Fuzzer tek başına yalnız çökmeleri görür; KASAN gibi denetleyiciler sessiz bellek bozulmalarını ilk geçersiz erişimde görünür kılar. Bu yüzden ikisi birlikte kullanılır.

Fuzzing: syzkaller Döngüsü

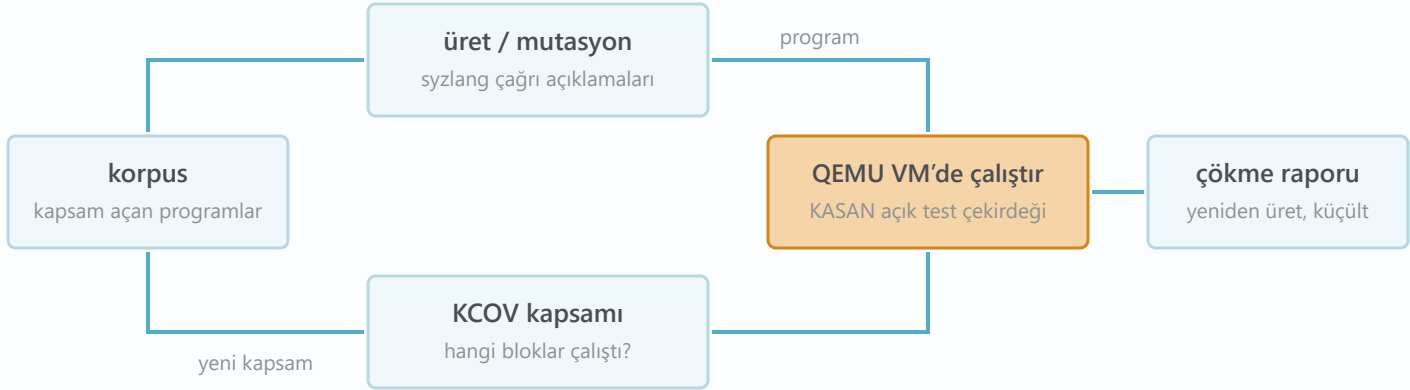


Üretici korpustan bir program seçip mutasyona uğratır ya da yenisini üretir: ör. `openat → ioctl → mmap → close`. Argümanlar tipe uygun seçildiği için çağrılar hemen `EINVAL` ile reddedilmez.

Adım 2 / 6

Fuzzer tek başına yalnız çökmeleri görür; KASAN gibi denetleyiciler sessiz bellek bozulmalarını ilk geçersiz erişimde görünür kılar. Bu yüzden ikisi birlikte kullanılır.

Fuzzing: syzkaller Döngüsü

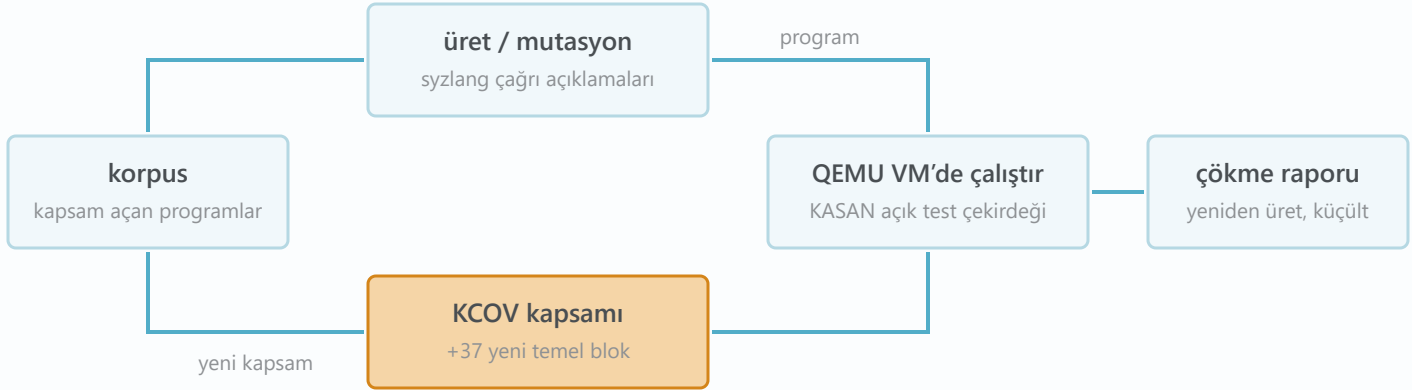


Program, KASAN/KCSAN/lockdep açık bir test çekirdeğiyle çalışan QEMU sanal makinelerinde koşar. VM çökse bile ana makine etkilenmez; `syz-manager` VM'i yeniden başlatır.

Adım 3 / 6

Fuzzer tek başına yalnız çökmeleri görür; KASAN gibi denetleyiciler sessiz bellek bozulmalarını ilk geçersiz erişimde görünür kılar. Bu yüzden ikisi birlikte kullanılır.

Fuzzing: syzkaller Döngüsü

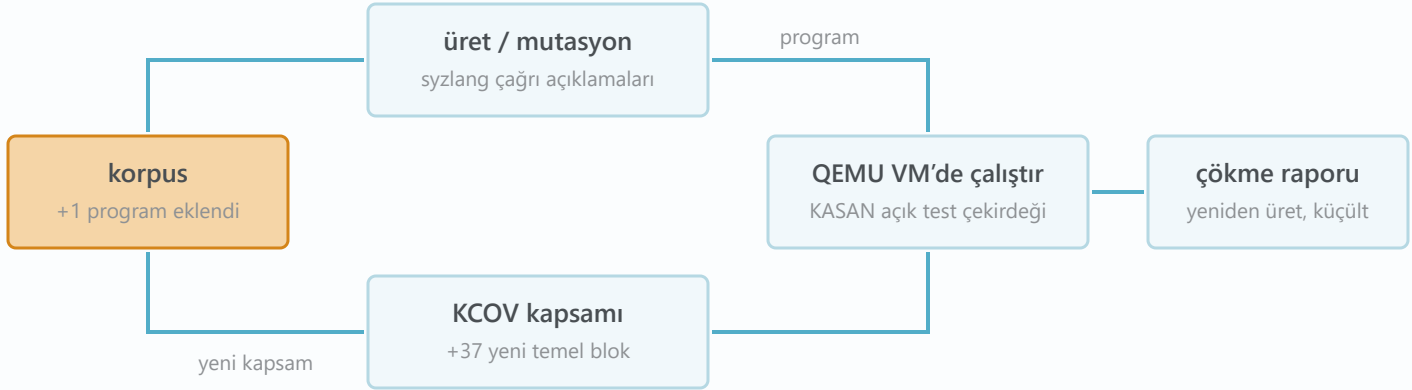


KCOV, çekirdekte geçilen temel blokları (PC değerlerini) kaydeder. Bu **kapsam geri bildirimi**, rastgele denemeyi yönlendirilmiş bir aramaya çevirir.

Adım 4 / 6

Fuzzer tek başına yalnız çökmeleri görür; KASAN gibi denetleyiciler sessiz bellek bozulmalarını ilk geçersiz erişimde görünür kılar. Bu yüzden ikisi birlikte kullanılır.

Fuzzing: syzkaller Döngüsü

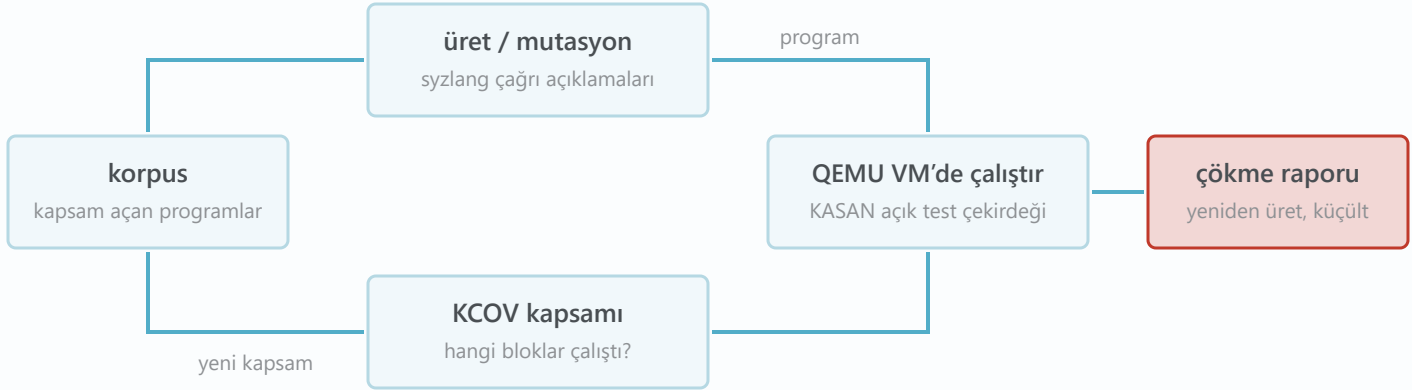


Yeni kod yolu açan program korpusa eklenir ve sonraki mutasyonların tohumu olur. Döngü, her VM'de saniyede yüzlerce programla durmadan döner.

Adım 5 / 6

Fuzzer tek başına yalnız çökmeleri görür; KASAN gibi denetleyiciler sessiz bellek bozulmalarını ilk geçersiz erişimde görünür kılar. Bu yüzden ikisi birlikte kullanılır.

Fuzzing: syzkaller Döngüsü



Çökme ya da KASAN raporu gelirse syzkaller programı yeniden üretir, en küçük hâline indirir ve çoğu zaman bir C yeniden üreticisi çıkarır. **syzbot** bunu otomatik olarak geliştiricilere bildirir.

Adım 6 / 6

Fuzzer tek başına yalnız çökmeleri görür; KASAN gibi denetleyiciler sessiz bellek bozulmalarını ilk geçersiz erişimde görünür kılar. Bu yüzden ikisi birlikte kullanılır.

Gözlem ve İz Sürme Araçları

Araç	Ne sağlar?	Örnek
printk	Log düzeyleri 0 (EMERG) ... 7 (DEBUG); halka tampona yazılır, eşğin üstündekiler konsola basılır	<code>pr_warn("kuyruk dolu: %d\n", n);</code>
dmesg	Halka tamponu okur; <code>kernel.dmesg_restrict=1</code> ise yalnız CAP_SYSLOG sahibi okur (adres sızıntısını önler)	<code>dmesg -w --level=err,warn</code>
dynamic debug	<code>pr_debug</code> satırlarını yeniden derlemeden, çalışırken açar	<code>echo 'module e1000e +p' > .../dynamic_debug/control</code>
ftrace	Çekirdek içi izleyici: işlev ve işlev çizgesi izleme, tracepoint olayları (tracefs)	<code>trace-cmd record -p function_graph -g vfs_read</code>
perf	Donanım sayaçları ve örneklemeli profil: sıcak noktalar, önbellek iskalari, çizelgeleyici gecikmesi	<code>perf record -g -- ./uyg , perf report</code>
eBPF / bpftrace	Doğrulayıcıdan geçen küçük programlar kprobe ve tracepoint'lere bağlanır; üretimde güvenli gözlem	aşağıdaki tek satırlık örnek
kdump / crash	Panikte kexec ile yedek çekirdeğe geçip bellek dökümünü (<code>/proc/vmcore</code>) kaydeder; sonra çevrim dışı incelenir	<code>crash vmlinux vmcore</code>

```
# hangi süreç hangi dosyayı açıyor?
```

```
bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm, str(args.filename)); }'
```

Önce gözle, sonra tahmin et: log ve izler hipotez kurmayı sağlar. Ama ağır `printk` zamanlamayı değiştirip yarış hatasını gizleyebilir; düşük yüklü izleyiciler (ftrace, eBPF) bu yüzden tercih edilir.

QEMU ve gdb ile Çekirdek Hata Ayıklama

```
# 1. terminal: -s → gdb sunucusu :1234
#           -S → CPU başta durur
qemu-system-x86_64 -kernel bzImage \
  -append "console=ttyS0 nokaslr" \
  -nographic -s -S
# 2. terminal
gdb vmlinux
(gdb) target remote :1234
(gdb) hbreak start_kernel
(gdb) continue
(gdb) bt
(gdb) lx-dmesg # çekirdek gdb betiği
```

- **vmlinux** : sembollü (**CONFIG_DEBUG_INFO**), sıkıştırılmamış çekirdek; **nokaslr** adresleri sembollerle eşler.
- **kgdb**: aynı gdb oturumu gerçek donanımda seri hat üzerinden (**kgdboc=ttyS0,115200**).

```
BUG: kernel NULL pointer dereference,
      address: 0000000000000010
Oops: 0000 [#1] PREEMPT SMP NOPTI
CPU: 2 PID: 1432 Comm: cat
Tainted: G          0 6.12.0
RIP: 0010:mydrv_read+0x2a/0x80 [mydrv]
Call Trace:
  vfs_read+0xa4/0x2f0
  ksys_read+0x63/0xe0
  do_syscall_64+0x82/0x170
```

- **RIP**: işlem + ofset / işlem boyu; **scripts/faddr2line** kaynak satırını verir.
- **Tainted O**: ağaç dışı modül yüklü. Çağrı izi aşağıdan yukarı okunur: **read** → VFS → sürücü.
- **Oops** o süreci öldürür, sistem sürebilir; **panic_on_oops=1** ise doğrudan **panic**.

Windows: WinDbg ile çekirdek modunda hata ayıklama (ağ, USB ya da sanal makine üzerinden) ve mavi ekran dökümlerinde **!analyze -v**; **Driver Verifier** (**verifier.exe**) sürücülerini özel havuz, IRQL ve kaynak sızıntısı denetimleriyle zorlayarak hataları erken yakalar.

Regresyon Avı: git bisect

- ▶ `git bisect start`
- ▶ `git bisect bad v6.12`
- ▶ `git bisect good v6.11`

```
4 # #500: derle, başlat, sina
5 git bisect bad
6 git bisect good # #250
7 git bisect good # #375
8 git bisect bad # #437
9 git bisect run ./sina.sh
10 git bisect reset
```

aralık: 1000 commit



v6.11 · iyi

v6.12 · kötü

kalan aday: 1000 commit

Bilinen iyi (v6.11) ve kötü (v6.12) uçlar verilir. İlk kötü commit bu 1000 commit'in içindedir; git ortadakini, **#500**'ü çıkarır.

Adım 1 / 7

DEĞİŞKENLER

aralık = **[0, 1000]**

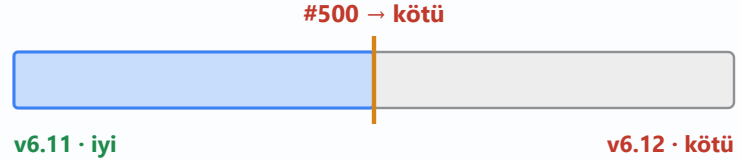
kalan = **1000**

test = **0**

`./sina.sh` 0 ile biterse commit iyi, 1–127 (125 hariç) ile biterse kötü sayılır; 125 "bu commit sınanamaz, atla" demektir.

Regresyon Avı: git bisect

```
1 git bisect start
2 git bisect bad v6.12
3 git bisect good v6.11
▶ # #500: derle, başlat, sina
▶ git bisect bad
6 git bisect good # #250
7 git bisect good # #375
8 git bisect bad # #437
9 git bisect run ./sina.sh
10 git bisect reset
```



kalan aday: 500 commit · test: 1

#500 derlenip sınanır: hata var, `git bisect bad`. Hata #500'de ya da öncesinde başlamıştır; sağ yarı elenir.

Adım 2 / 7

DEĞİŞKENLER

aralık = [0, 500]

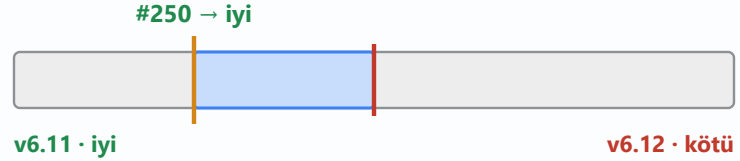
kalan = 500

test = 1

`./sina.sh` 0 ile biterse commit iyi, 1–127 (125 hariç) ile biterse kötü sayılır; 125 "bu commit sınanamaz, atla" demektir.

Regresyon Avı: git bisect

```
1 git bisect start
2 git bisect bad v6.12
3 git bisect good v6.11
4 # #500: derle, başlat, sina
5 git bisect bad
6 ► git bisect good # #250
7 git bisect good # #375
8 git bisect bad # #437
9 git bisect run ./sina.sh
10 git bisect reset
```



kalan aday: 250 commit · test: 2

#250'de hata yok: **good**. İlk kötü commit 251–500 arasındadır.

Adım 3 / 7

DEĞİŞKENLER

aralık = (250, 500]

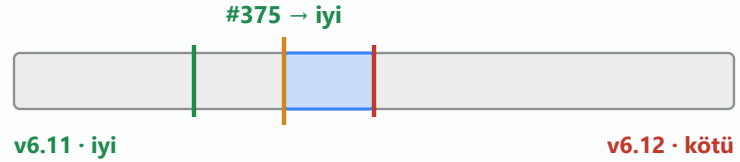
kalan = 250

test = 2

./sina.sh 0 ile biterse commit iyi, 1–127 (125 hariç) ile biterse kötü sayılır; 125 "bu commit sınanamaz, atla" demektir.

Regresyon Avı: git bisect

```
1 git bisect start
2 git bisect bad v6.12
3 git bisect good v6.11
4 # #500: derle, başlat, sina
5 git bisect bad
6 git bisect good # #250
7 ► git bisect good # #375
8 git bisect bad # #437
9 git bisect run ./sina.sh
10 git bisect reset
```



kalan aday: 125 commit · test: 3

#375 de iyi. Her yanıt aday sayısını yarıya indirir: 1000 → 500 → 250 → 125.

Adım 4 / 7

DEĞİŞKENLER

aralık = [375, 500]

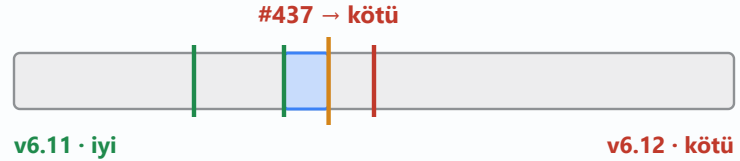
kalan = 125

test = 3

`./sina.sh` 0 ile biterse commit iyi, 1–127 (125 hariç) ile biterse kötü sayılır; 125 "bu commit sınanamaz, atla" demektir.

Regresyon Avı: git bisect

```
1 git bisect start
2 git bisect bad v6.12
3 git bisect good v6.11
4 # #500: derle, başlat, sıra
5 git bisect bad
6 git bisect good # #250
7 git bisect good # #375
8 ► git bisect bad # #437
9 git bisect run ./sina.sh
10 git bisect reset
```



kalan aday: 62 commit · test: 4

#437 kötü: aralık 376–437. Elle sürdürmek yerine sınamayı bir betiğe bırakabiliriz.

Adım 5 / 7

DEĞİŞKENLER

aralık = (375, 437]

kalan = 62

test = 4

`./sina.sh` 0 ile biterse commit iyi, 1–127 (125 hariç) ile biterse kötü sayılır; 125 “bu commit sınanamaz, atla” demektir.

Regresyon Avı: git bisect

```
1 git bisect start
2 git bisect bad v6.12
3 git bisect good v6.11
4 # #500: derle, başlat, sina
5 git bisect bad
6 git bisect good # #250
7 git bisect good # #375
8 git bisect bad # #437
9 ► git bisect run ./sina.sh
10 git bisect reset
```



10 testte bulundu = $\lceil \log_2 1000 \rceil$

`git bisect run` kalan adımları kendisi yürütür: 406 iyi, 421 kötü, 413 kötü, 409 iyi, 411 iyi, 412 kötü. Toplam **10 test**: $2^{10} = 1024 \geq 1000$.

Adım 6 / 7

DEĞİŞKENLER

aralık = [411, 412]

kalan = 1

test = 10

`./sina.sh` 0 ile biterse commit iyi, 1–127 (125 hariç) ile biterse kötü sayılır; 125 “bu commit sınanamaz, atla” demektir.

Regresyon Avı: git bisect

```
1 git bisect start
2 git bisect bad v6.12
3 git bisect good v6.11
4 # #500: derle, başlat, sıra
5 git bisect bad
6 git bisect good # #250
7 git bisect good # #375
8 git bisect bad # #437
9 git bisect run ./sina.sh
▶ git bisect reset
```



git bisect reset → çalışma dalına dön

Sonuç: "#412 is the first bad commit". `reset` ile başlangıç dalına dönülür. 1 milyon commit olsaydı bile yalnızca $\lceil \log_2 10^6 \rceil = 20$ test gerekirdi.

Adım 7 / 7

DEĞİŞKENLER

aralık = **[411, 412]**

kalan = **1**

test = **10**

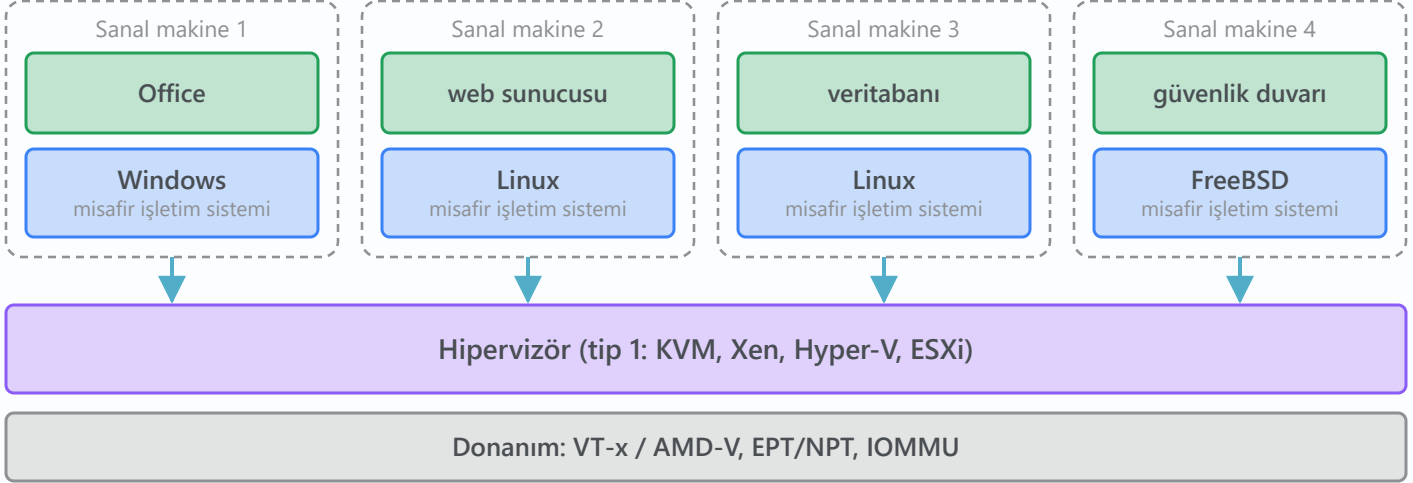
`./sina.sh` 0 ile biterse commit iyi, 1–127 (125 hariç) ile biterse kötü sayılır; 125 "bu commit sınanamaz, atla" demektir.

İşletim Sistemlerinde Eğilimler

Eğilim	İşletim sistemine etkisi
Sanallaştırma ve bulut	Hipervizör, konteyner, mikro-VM (Firecracker); çok kiracılı yalıtım
Çok çekirdekli yongalar	Ölçeklenebilir kilitler, RCU, NUMA farkındalığı, heterojen çekirdekler
Geniş adres alanı	64 bit, 5 düzeyli sayfa tabloları; tek adres alanlı sistem fikirleri
Kesintisiz veri erişimi	Ağ, bulut depolama ve yerel disk arasında şeffaf erişim; NVMe, CXL bellek
Pille çalışan sistemler	DVFS, “hızla bitir ve uyu” (race-to-idle), Android Doze, enerji farkında çizelgeleme (EAS)
Gömülü ve sensör düğümleri	Gerçek zamanlı küçük çekirdekler: Zephyr, FreeRTOS, Contiki-NG; kilobaytlarca bellek
Güvenli diller ve programlanabilir çekirdek	Linux’ta Rust (6.1’den beri), eBPF ve sched_ext, unikernel’ler

Ağ işlemleri, paralel ve dağıtık sistemler ve çoklu ortam artık her sistemin olağan parçası olduğundan ayrı eğilim olarak değil, ilgili satırların içinde ele alınır.

Sanallaştırma ve Bulut

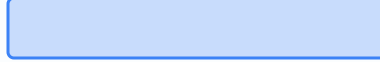


- Sunucu birleştirme, hızlı taşıma, anlık görüntü; bulutun temeli.
- İşletim sistemi tasarımına etkisi: misafir sistem sanal ortamda olduğunu bilir (paravirtio sürücüləri).
- **Konteynerler** çekirdeği paylaşır: daha hafif, daha zayıf yalıtım.
- **Mikro-VM**'ler (Firecracker, AWS Lambda): VM yalıtımı, milisaniyede açılış.

Çok Çekirdek ve Geniş Adres Alanları

sanal adres alanı (çubuk uzunluğu = adres bit sayısı; her bit boyutu ikiye katlar)

32 bit



2^{32} B = 4 GiB

48 bit (4 düzeyli sayfa tablosu)



2^{48} B = 256 TiB

57 bit (5 düzeyli, LA57)



2^{57} B = 128 PiB

Çok çekirdekli yongalar

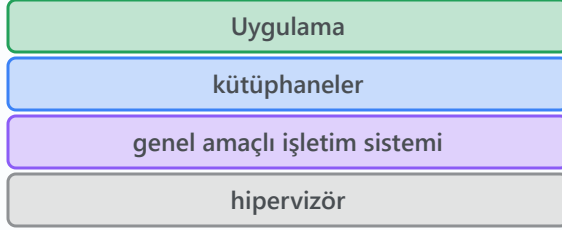
Sunucularda yuva başına 128-192 çekirdek, telefonlarda büyük/küçük (big.LITTLE) karışık çekirdekler. Tek bir büyük çekirdek kilidi ölçeklenmez: ince taneli kilitler, RCU, çekirdek başına veri yapıları gerekir.

Geniş adres alanları

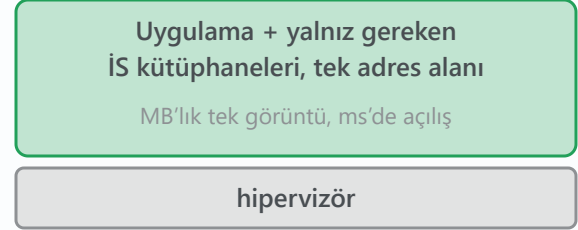
Tüm dosyaları belleğe eşlemek (mmap) ya da tek adres alanlı sistemler mümkün olur. Bedeli: daha derin sayfa tablosu yürüyüşü, daha fazla TLB baskısı.

Rust, eBPF ve Unikernel'ler

Geleneksel yığın



Unikernel



Rust ile çekirdek

Linux **6.1** (Aralık 2022) ile Rust desteği ana dala girdi, ardından ilk Rust sürücüleri geldi. Amaç: kullanım-sonrası-serbest bırakma ve taşma gibi bellek hatalarını derlemede önlemek. Windows çekirdeğinde de Rust bileşenleri var.

eBPF

Doğrulanmış küçük programlarla çekirdeği yeniden derlemeden genişletmek: izleme, ağ (XDP), güvenlik, çizelgeleme (sched_ext).

Unikernel

Exokernel'in libOS fikrinin bulut sürümü: MirageOS, Unikraft. Tek uygulamalı mikro hizmetler için.

ÖZET

Özet

Konu	Ana fikir
Zorluklar	Boyut, eş zamanlılık, düşman kullanıcılar, uzun ömür, uyumluluk
Hedefler	Soyutlama, ilkel işlemler, yalıtım, donanım yönetimi
Arayüz	Basit, bütün, verimli; az ve genel sistem çağırısı
Yapı	Katmanlı, exokernel, mikro çekirdek, genişletilebilir, çekirdek iş parçacıkları
İlkeler	Mekanizma/politika, diklik, iki düzeyli ad, geç bağlama
Teknikler	Donanımı gizle, dolaylılık, yeniden girişlilik, önce hata denetimi
Performans	Ölç; uzay-zaman ödünleşimi, önbellek, ipucu, yerellik, yaygın durum
Proje	Brooks yasası, küçük güçlü takımlar, gümüş kurşun yok
Test	KUnit, kselftest, LTP; KASAN, KCSAN, lockdep; syzkaller; ftrace, eBPF; QEMU+gdb; git bisect
Eğilimler	Bulut, çok çekirdek, enerji, Rust, eBPF, unikernel

ÖZET

Alıştırma: Test ve Hata Ayıklama

1. Bir sürücüde ara sıra görülen use-after-free hatasını hangi araçlarla, hangi sırayla yakalarsınız? syzkaller neden KASAN açık bir çekirdekle çalıştırılır?
2. v6.11 ile v6.12 arasında 12 000 commit var ve bir performans gerilemesi v6.12'de ortaya çıkmış. `git bisect` en çok kaç test gerektirir? Sınama betiği hangi çıkış kodlarını kullanmalıdır?
3. Bir yarış hatası `printk` eklenince kayboluyor. Bunun nedeni nedir; hangi araçlarla incelemeye devam edersiniz?

► Cevaplar

Kontrol Soruları

1. Tanenbaum'a göre işletim sistemi tasarımını zorlaştıran sekiz etkenden dördünü açıklayın.
2. Basitlik, bütünlük ve verimlilik ilkelerini birer sistem çağrısı örneğiyle tartışın.
3. Algoritmik ve olay güdümlü yürütme paradigmalarını karşılaştırın. Olay döngüsünde uzun süren bir işleyici neden sorun olur?
4. Exokernel ile sanal makine monitörü arasındaki temel fark nedir?
5. Mikro çekirdekte bir sürücü çöktüğünde ne olur? Monolitik çekirdekte aynı durum neden daha tehlikelidir?
6. Mekanizma ile politikayı ayırmayı çizelgeleyici örneğiyle açıklayın. sched_ext bu ilkeyi nasıl uygular?
7. /usr/ast/mbox yolunun çözülmesi önbelleksiz kaç disk erişimi gerektirir? Önbellek ile ipucu arasındaki fark nedir?
8. Bit sayma örneğinde döngü, makro ve tablo yöntemlerini yer ve zaman açısından karşılaştırın.
9. futex neden yaygın durumu hızlandırmanın iyi bir örneğidir?
10. Brooks yasasını iletişim yolu sayısı ile gerekçelendirin: 10 kişilik ekipte kaç yol vardır?

► Cevaplar

Kaynaklar

- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 12: Operating System Design.
- B. W. Lampson, "Hints for Computer System Design", *ACM SOSP*, 1983 (ve 2020 güncellemesi "Hints and Principles for Computer System Design").
- F. P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering*, 20. yıl baskısı, Addison-Wesley, 1995 — "No Silver Bullet" (1986) dahil.
- J. H. Saltzer, M. F. Kaashoek, *Principles of Computer System Design: An Introduction*, Morgan Kaufmann, 2009.
- D. R. Engler, M. F. Kaashoek, J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management", *SOSP* 1995.
- G. Klein vd., "seL4: Formal Verification of an OS Kernel", *SOSP* 2009; J. N. Herder vd., MINIX 3 güvenilirlik çalışmaları.
- Linux çekirdeği belgeleri: Rust (docs.kernel.org/rust), BPF ve sched_ext; H. Franke vd., "Fuss, Futexes and Furwocks", 2002.

Son

Bölüm 15: İşletim Sistemi Tasarımı

Sonraki bölüm: Linux · [Terim sözlüğü](#)