Bölüm 15: Linux

İşletim Sistemleri

UNIX ve Linux tarihçesi · Tasarım hedefleri ve arayüzler · Kabuk ve boru hatları · Çekirdek yapısı ve modüller · Süreçler, sinyaller, iş parçacıkları · Çizelgeleme: CFS ve EEVDF · Önyükleme ve systemd · Bellek: sayfalama, bölgeler, SLUB · G/Ç: aygıtlar, blk-mq, ağ · Dosya sistemleri: VFS, ext4 · Güvenlik: izinler, konteynerler · Android

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_15_Linux.html

GİRİŞ

Öğrenme Çıktıları

Tarih ve aile
UNIX'ten BSD, MINIX, GNU ve Linux'a uzanan çizgiyi; dağıtımları ve Android'i konumlandırmak.

Arayüzler ve kabuk
Kütüphane / sistem çağrısı / çekirdek katmanlarını ayırmak; boru hatlarının fork, pipe, dup2 ile kurulduğunu göstermek.

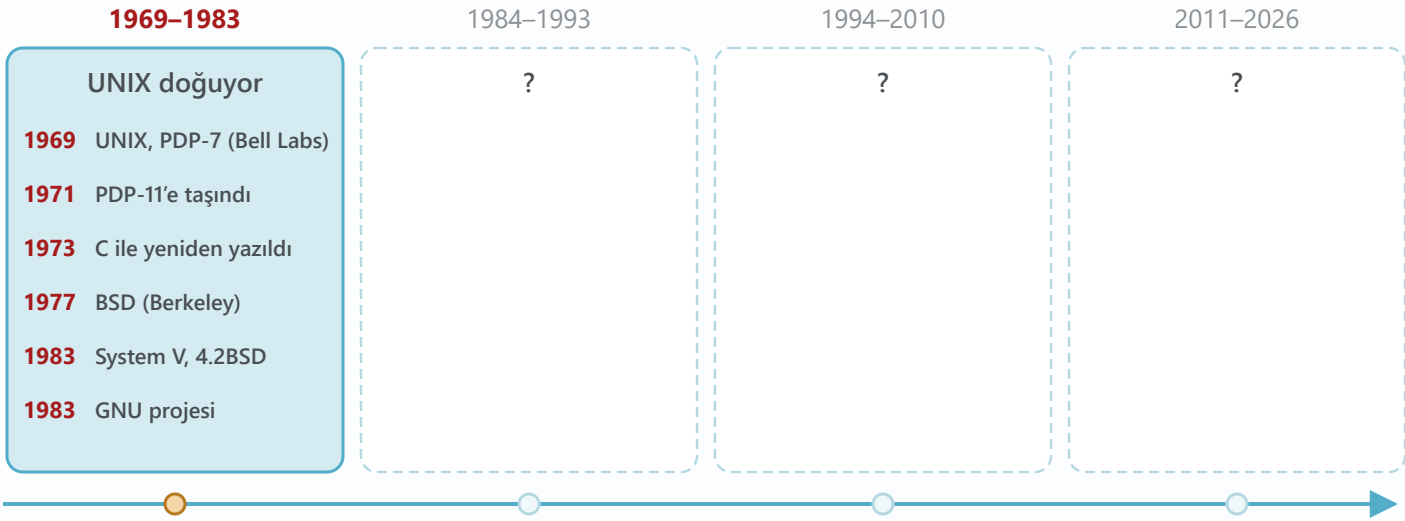
Süreçler ve çizelgeleme
fork ve execve'nin çekirdekteki adımlarını, sinyal teslimini, task_struct, clone() bayraklarını ve NPTL'i; CFS'ten EEVDF'e geçişte nelerin değiştiğini açıklamak.

Bellek
Adres alanı düzenini, 4/5 düzeyli sayfa tablolarını, bellek bölgelerini, SLUB ayırıcısını, sayfa önbelleğini ve OOM'u yorumlamak.

G/Ç ve dosyalar
Aygıt dosyaları, blk-mq, ağ yığını; VFS, i-node, ext4 extent ve günlükleme, bağlar ve /proc.

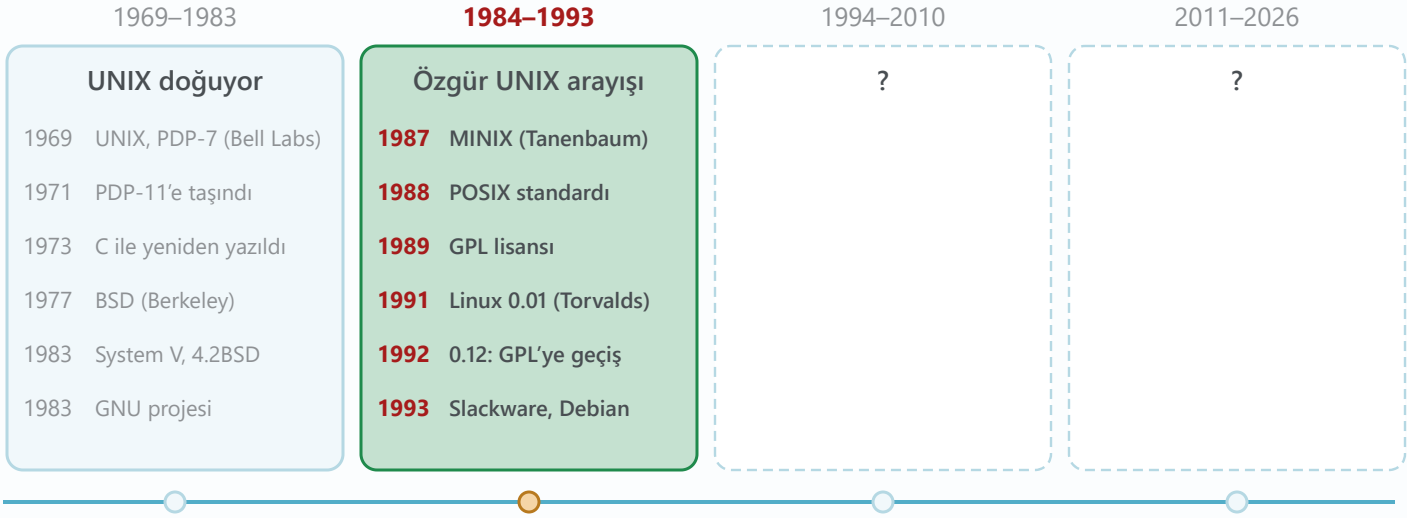
Güvenlik
rwx/setuid/umask hesaplamak; yetenekler, namespaces + cgroups, SELinux/AppArmor, seccomp ve Android korumalı alanını tanımak.

UNIX’ten Linux’a Zaman Çizelgesi



Ken Thompson ve Dennis Ritchie UNIX’i 1969’da PDP-7 üzerinde assembly ile yazdı; 1973’te **C ile yeniden yazılması** taşınabilirliği getirdi. AT&T kaynak kodunu üniversitelere verdi: Berkeley’den **BSD** doğdu.

UNIX’ten Linux’a Zaman Çizelgesi



Ticari UNIX’ler (System V, SunOS, AIX…) birbirinden ayrışınca **POSIX** ortak arayüzü tanımladı. Tanenbaum öğretim için **MINIX**’i yazdı; Linus Torvalds 1991’de MINIX üzerinde kendi çekirdeğini geliştirdi ve GNU araçlarıyla birleştirdi.

UNIX’ten Linux’a Zaman Çizelgesi

1969–1983

UNIX doğuyor

- 1969 UNIX, PDP-7 (Bell Labs)
- 1971 PDP-11’e taşındı
- 1973 C ile yeniden yazıldı
- 1977 BSD (Berkeley)
- 1983 System V, 4.2BSD
- 1983 GNU projesi

1984–1993

Özgür UNIX arayışı

- 1987 MINIX (Tanenbaum)
- 1988 POSIX standardı
- 1989 GPL lisansı
- 1991 Linux 0.01 (Torvalds)
- 1992 0.12: GPL’ye geçiş
- 1993 Slackware, Debian

1994–2010

Linux olgunlaşıyor

- **1994** Linux 1.0
- **1996** 2.0: SMP desteği
- **1999** Red Hat halka arz
- **2003** 2.6 serisi
- **2005** git
- **2008** Android: HTC Dream

2011–2026

?

Linux 1.0 (1994), 2.0 ile çok işlemci, 2.6 ile kesilebilir çekirdek ve O(1)/CFS çizelgeleyicileri. 2005’te BitKeeper sorunu üzerine Torvalds **git**’i yazdı. İlk Android telefonu 2008’de çıktı.

UNIX’ten Linux’a Zaman Çizelgesi

1969–1983 — UNIX doğuyor

1969	UNIX, PDP-7 (Bell Labs)
1971	PDP-11’e taşındı
1973	C ile yeniden yazıldı
1977	BSD (Berkeley)
1983	System V, 4.2BSD
1983	GNU projesi

1984–1993 — Özgür UNIX arayışı

1987	MINIX (Tanenbaum)
1988	POSIX standardı
1989	GPL lisansı
1991	Linux 0.01 (Torvalds)
1992	0.12: GPL’ye geçiş
1993	Slackware, Debian

1994–2010 — Linux olgunlaşıyor

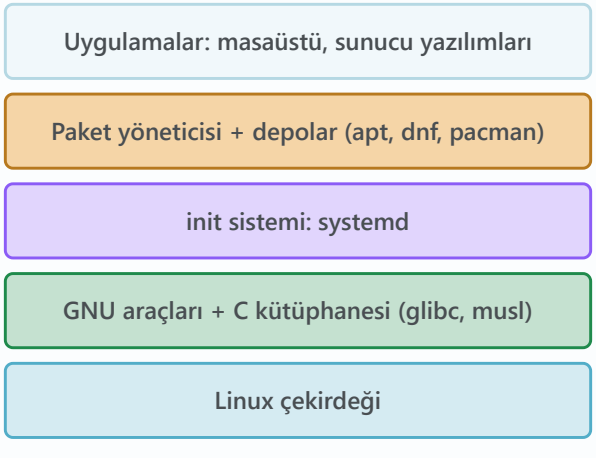
1994	Linux 1.0
1996	2.0: SMP desteği
1999	Red Hat halka arz
2003	2.6 serisi
2005	git
2008	Android: HTC Dream

2011–2026 — Her yerde Linux

2011	**3.0 numarası**
2017	**Top500: %100 Linux**
2022	**6.0; 6.1’de Rust**
2023	**6.6: EEVDF**
2024	**6.12: sched_ext, RT**
2026	**6.x → 7.0 serisi**

2017’den beri dünyanın en hızlı 500 süper bilgisayarının tamamı Linux çalıştırıyor. 6.x serisinde Rust desteği, EEVDF ve sched_ext geldi; numaralar yalnızca büyüdüğü için artar (6.19’un ardından 7.0).

GNU, Linux ve Dağıtımlar



Android: Linux çekirdeği + Bionic libc, GNU araçları yok

Aile	Örnekler	Paket
Debian	Debian, Ubuntu, Mint, Pardus	.deb / apt
Red Hat	Fedora, RHEL, Rocky, Alma	.rpm / dnf
Arch	Arch, Manjaro	pacman
SUSE	openSUSE, SLES	zypper
Hafif	Alpine (musl + BusyBox)	apk

Sürüm modeli: ~9–10 haftada bir ana sürüm (2 haftalık birleştirme penceresi + rc1…rc7). Her yılın son sürümü **LTS** olur: 6.6, 6.12, 6.18…

"Linux" tam olarak yalnızca çekirdeğin adıdır; kullanıcıların kurduğu sistem bir **dağıtımdır** (GNU/Linux).

Tasarım Hedefleri ve UNIX Felsefesi

Basit (simple)
Az sayıda, iyi tanımlı kavram: süreç, dosya, boru.

Şık (elegant)
Küçük, tek iş yapan araçlar birleştirilerek büyük iş yapılır.

Tutarlı (consistent)
“Her şey dosyadır”: disk, terminal, boru aynı read/write ile.

Güçlü (powerful)
Kabuk betikleri, boru hatları, düzenli ifadeler.

Esnek (flexible)
Metin akışları evrensel arayüz; araçlar birbirini bilmeden çalışır.

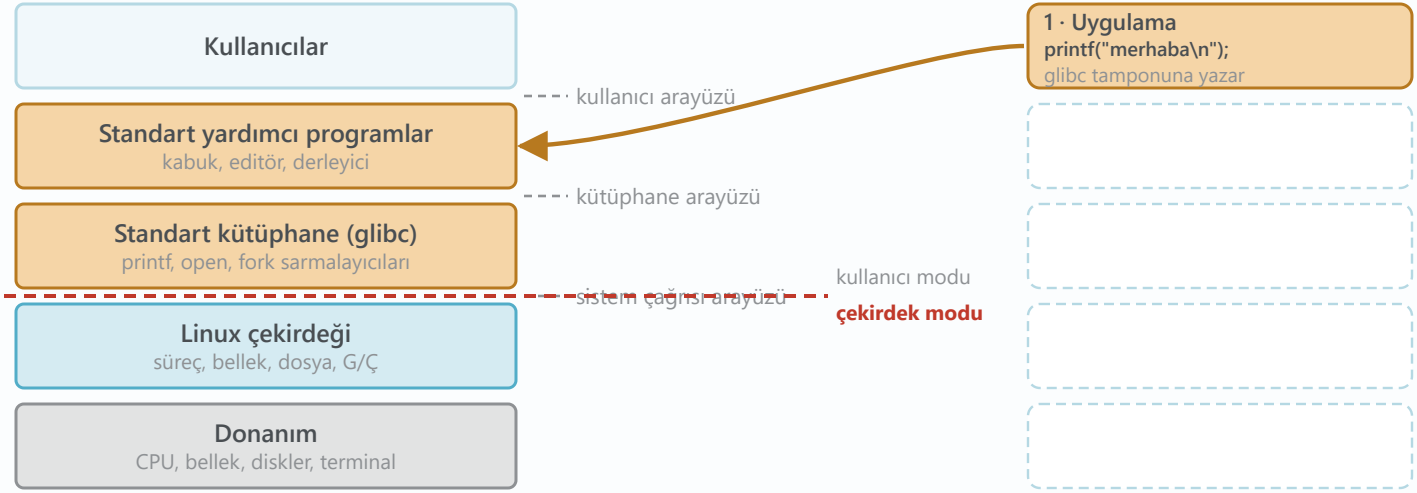
Programcı için
Sistemi programcılar programcılar için tasarladı: az “sihir”, sessiz başarı.

UNIX felsefesi (McIlroy): Her program tek bir işi iyi yapmalı ve başka programlarla birlikte çalışabilmelidir. Programlar metin akışlarını işlemelidir, çünkü metin evrensel arayüzdür.



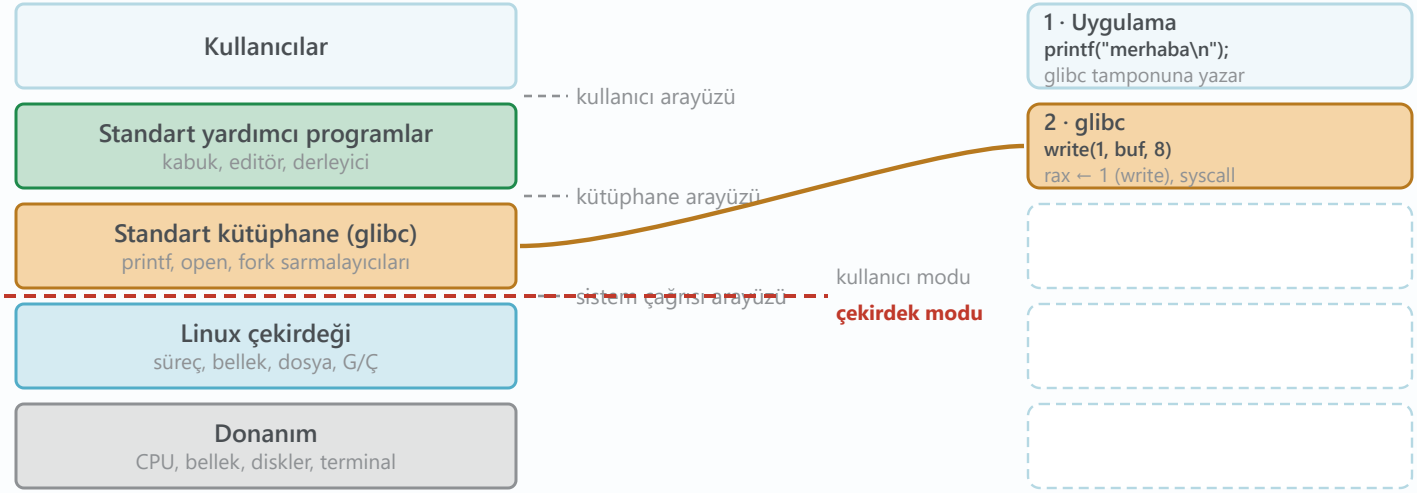
POSIX bu arayüzü standartlaştırır: aynı kaynak kod Linux, BSD ve macOS’ta derlenebilir.

Katmanlar ve Arayüzler: Bir printf'in Yolculuğu



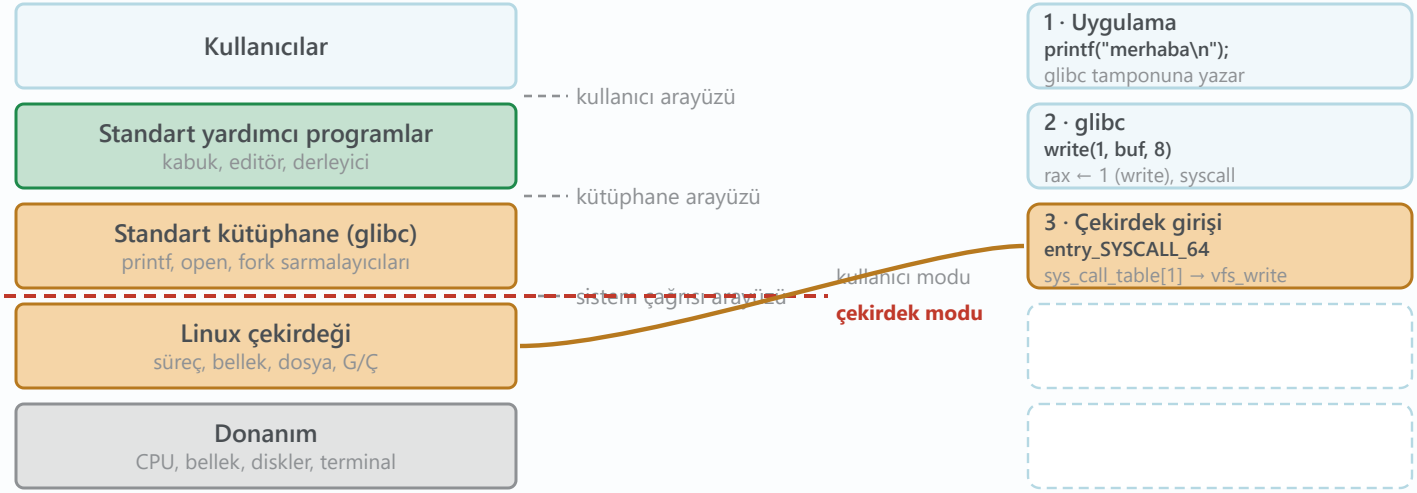
Program `printf` çağırır. Bu henüz bir sistem çağrısı değildir: glibc metni kullanıcı alanındaki bir tampona yazar.

Katmanlar ve Arayüzler: Bir printf'in Yolculuğu



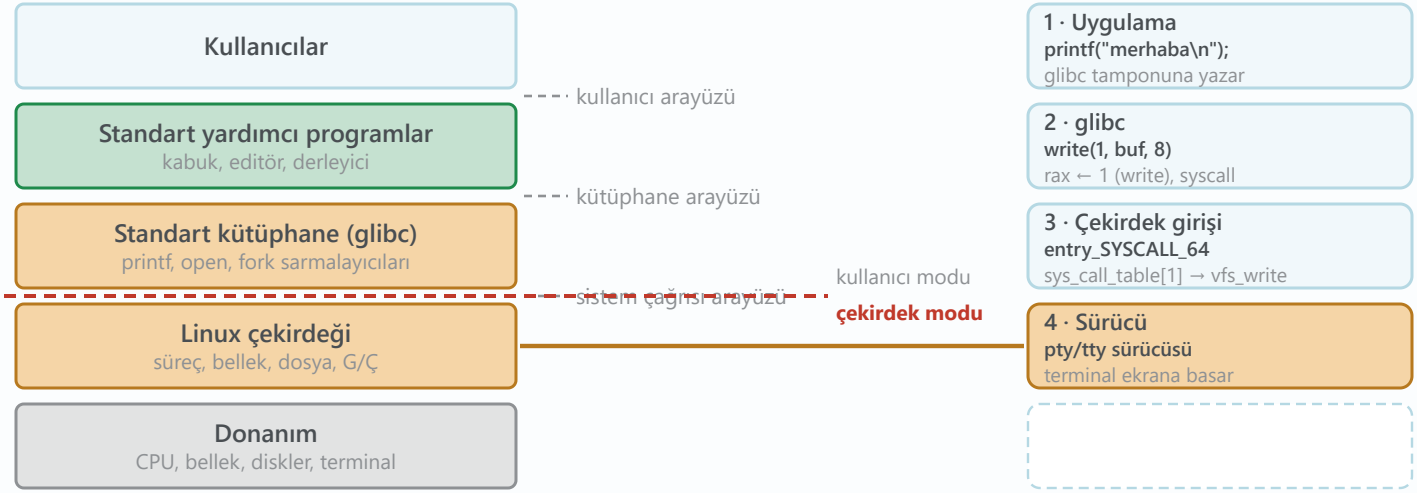
Tampon boşaltılırken glibc'nin `write()` sarmalayıcısı çalışır: sistem çağrısı numarasını (x86-64'te write = 1) `rax`'a, argümanları yazmaçlara koyar ve `syscall` komutunu yürütür.

Katmanlar ve Arayüzler: Bir printf'in Yolculuğu



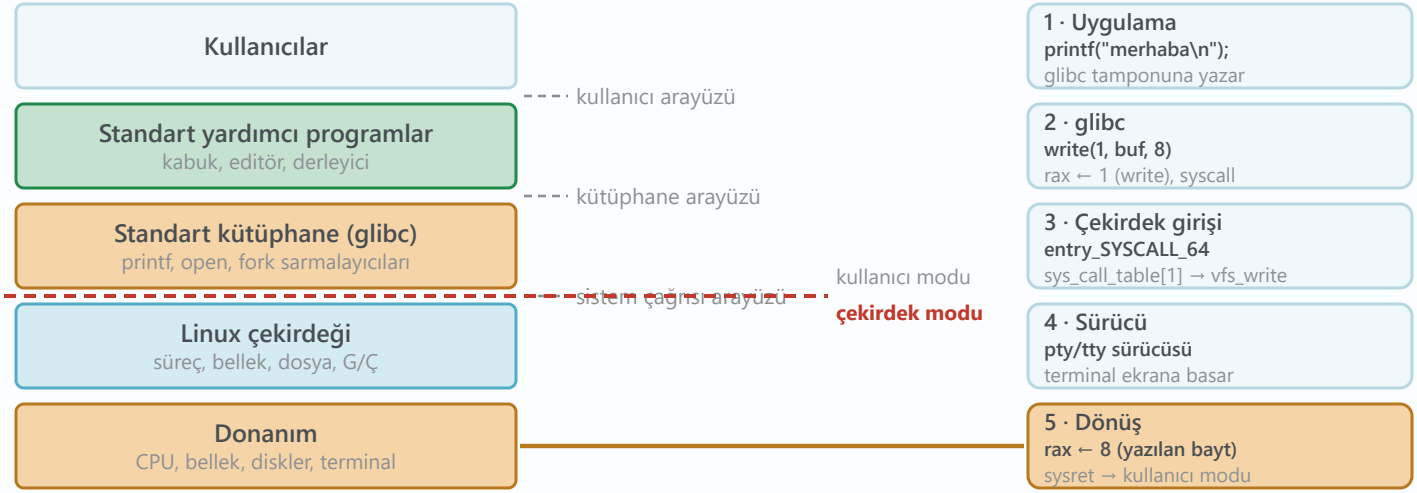
CPU çekirdek moduna geçer. Giriş kodu numarayla `sys_call_table`'dan işleyiciyi seçer; VFS dosya tanımlayıcısı 1'in arkasındaki nesneyi bulur.

Katmanlar ve Arayüzler: Bir printf'in Yolculuğu



fd 1 bir sözde terminale (pty) bağlıdır; tty katmanı veriyi terminal öykünücüsüne iletir. Disk olsaydı yol sayfa önbelleğine ve blok katmanına giderdi.

Katmanlar ve Arayüzler: Bir printf'in Yolculuğu



Dönüş değeri `rax` 'ta (8 bayt) döner, `sysret` ile kullanıcı moduna dönülür. Hata olursa negatif değer glibc'de `errno` 'ya çevrilir ve −1 döner.

Temel Komutlar ve Yönlendirme

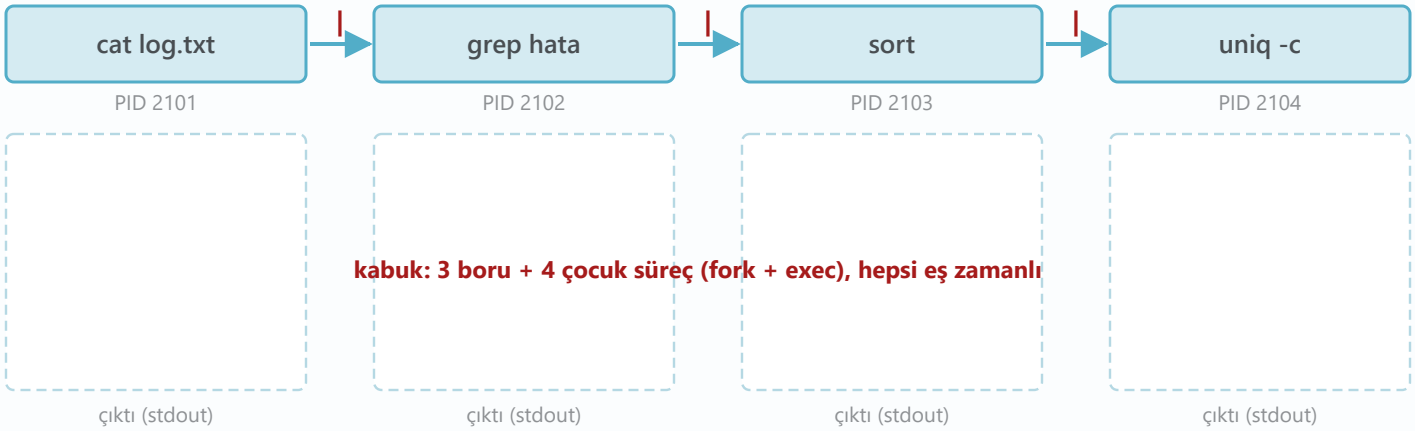
Kategori	Komutlar	Not
Dosya / dizin	`ls cd pwd mkdir rm -r cp mv touch find tar`	`touch` zaman damgasını günceller; dosya yoksa boş dosya oluşturur
Metin (filtre)	`cat less head tail grep sort uniq wc echo`	stdin'den okur, stdout'a yazar: boru hattında birleşir
Süreç / sistem	`ps top kill chmod chown df du sync`	`sync` kirli tamponları **diske** yazar (makineler arası eşitleme: `rsync`)
Ağ	`ssh scp ping curl wget traceroute`	`ip addr` , `ip route` , `ss` , `nft` ; eski `ifconfig` , `route` , `netstat` , `iptables` 'ın yerini aldı
Bilgi	`man uname free lsblk`	`/proc` komut değil, sözde dosya sistemidir: `cat /proc/meminfo`

```
# yönlendirme: 0 = stdin, 1 = stdout, 2 = stderr
sort < girdi.txt > cikti.txt 2> hatalar.log   # < okur, > yazar (üzerine)
echo "satır" >> gunluk.txt                    # >> sona ekler
make 2>&1 | tee derleme.log &                 # stderr→stdout, boru, arka plan
```

sort'un tanımlayıcı tablosu (kabuk exec'ten önce kurar):

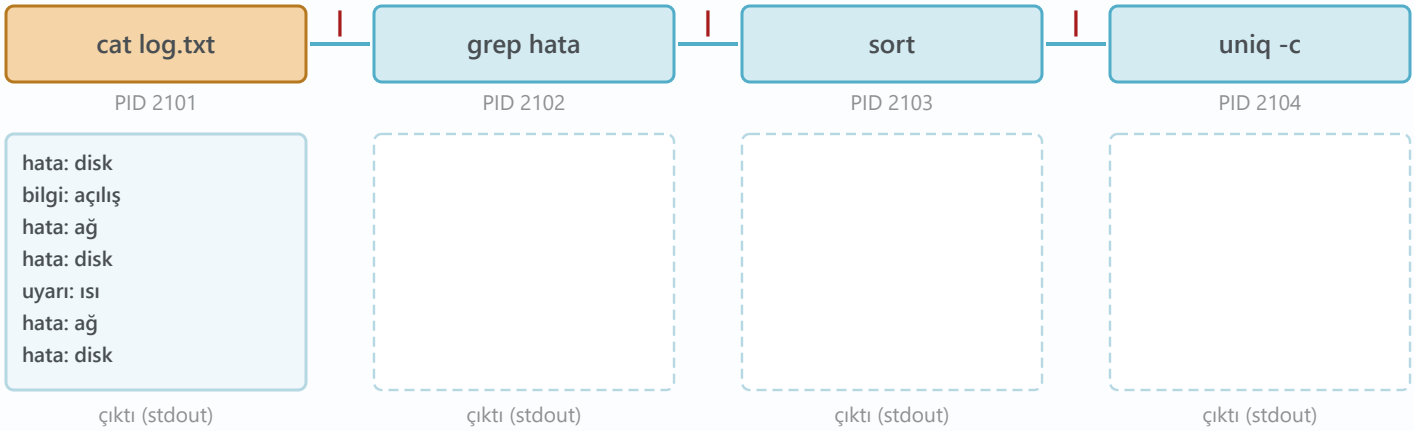


Boru Hattı: Veri Aşamalardan Akar



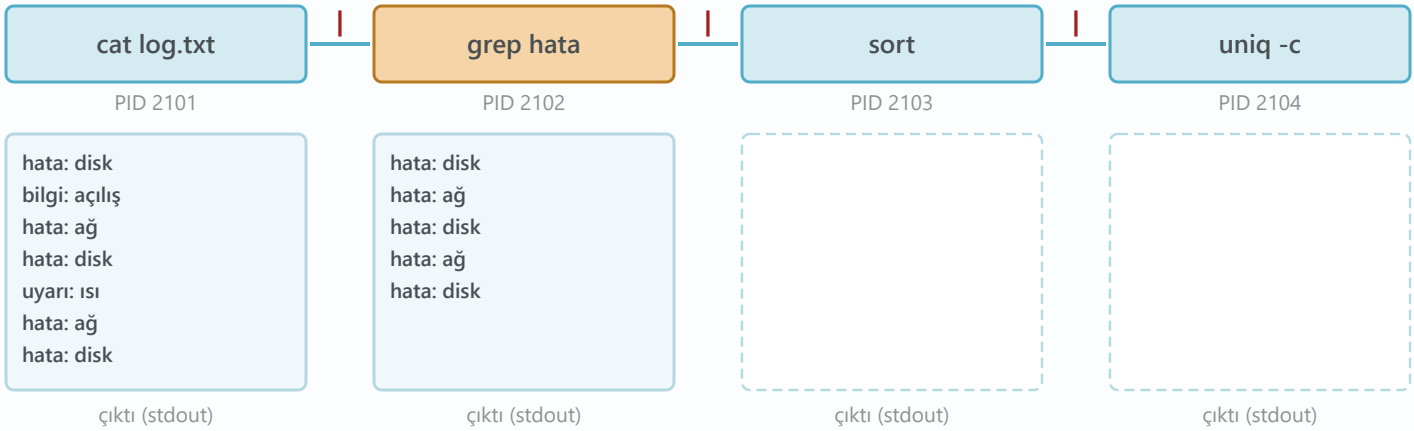
Kabuk satırı ayrıştırır, 3 boru ve 4 süreç oluşturur. Süreçler **aynı anda** çalışır; her biri yalnızca kendi stdin/stdout'unu bilir.

Boru Hattı: Veri Aşamalardan Akar



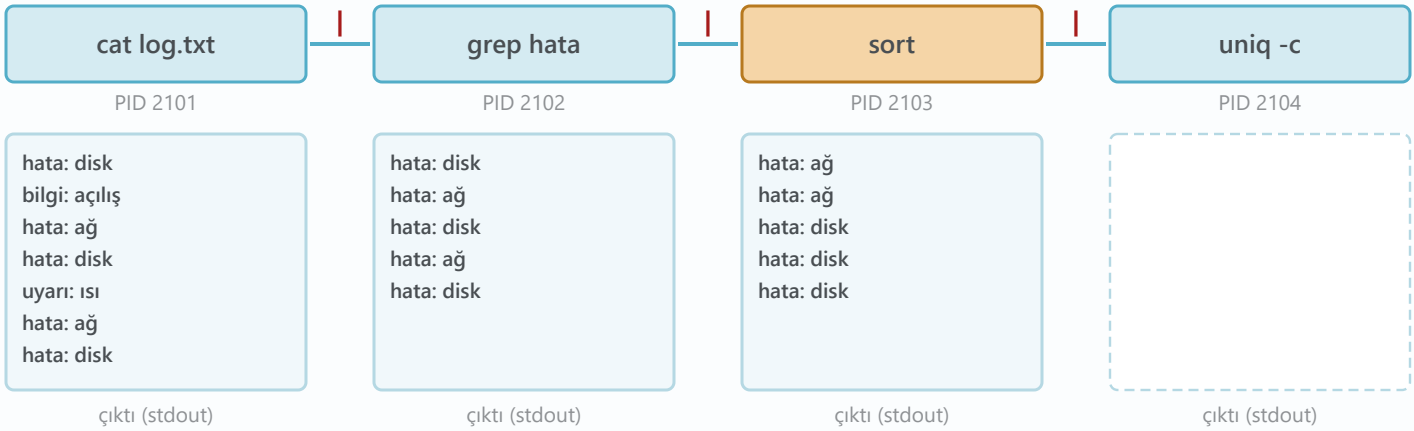
`cat` dosyanın 7 satırını boruya yazar. Boru tamponu (varsayılan 64 KiB) dolarsa yazan süreç **bloke olur**: üretici–tüketici.

Boru Hattı: Veri Aşamalardan Akar



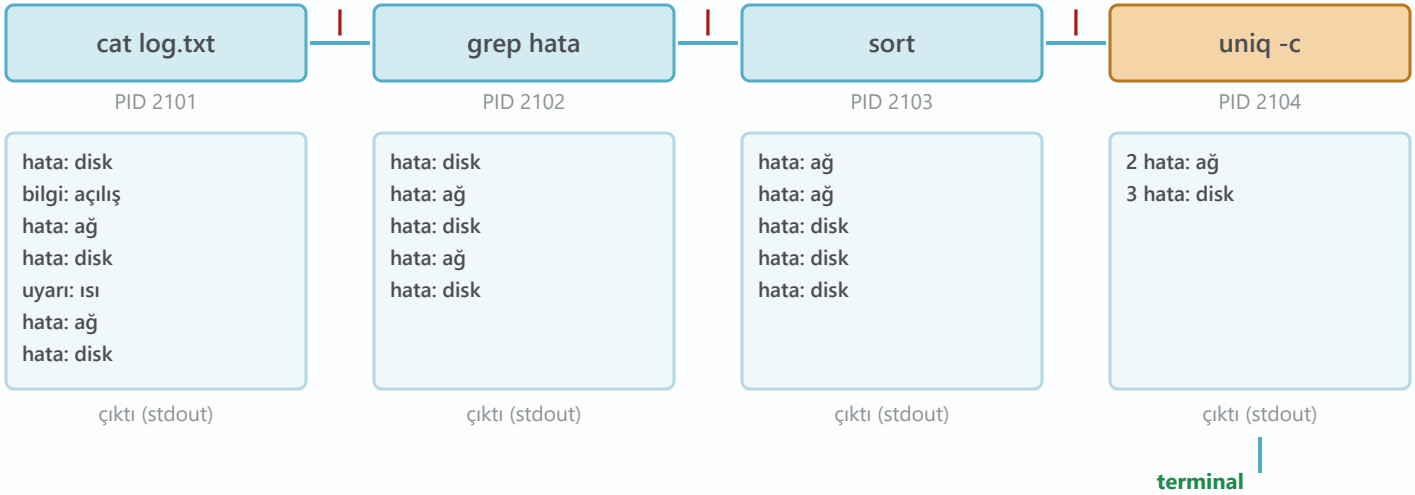
`grep` satır satır okur, eşleşen 5 satırı hemen iletir. Filtre durumsuzdur, akışı bekletmez.

Boru Hattı: Veri Aşamalardan Akar



`sort` ise **tüm girdiyi** (EOF'a kadar) okumadan çıktı veremez: son satır ilk sırayı değiştirebilir.

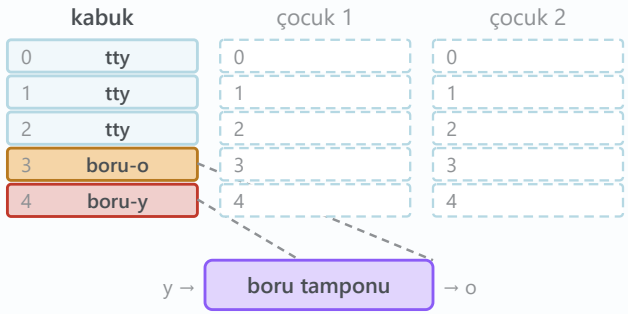
Boru Hattı: Veri Aşamalardan Akar



`uniq -c` yalnızca **ardışık** tekrarları sayar; bu yüzden önce sort gerekir. Sonuç: 2 × “hata: ağ”, 3 × “hata: disk”.

Kabuk Bir Boru Hattını Nasıl Kurar?

```
▶ int fd[2];
▶ pipe(fd);                 /* 3: okuma, 4: yazma */
3 if (fork() == 0) {     /* çocuk 1 */
4     dup2(fd[1], 1);
5     close(fd[0]); close(fd[1]);
6     execlp("ls", "ls", NULL);
7 }
8 if (fork() == 0) {     /* çocuk 2 */
9     dup2(fd[0], 0);
10    close(fd[0]); close(fd[1]);
11    execlp("wc", "wc", "-l", NULL);
12 }
13 /* kabuk: iki ucu da kapat */
14 close(fd[0]); close(fd[1]);
15 wait(NULL); wait(NULL);
```



`pipe()` çekirdekte bir tampon ve iki uç oluşturur: en küçük boş numaralar **3 (okuma)** ve **4 (yazma)**.

Adım 1 / 6

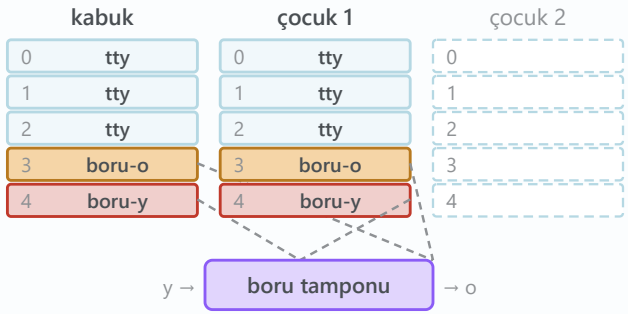
DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4`

ÇIKTI

Kabuk Bir Boru Hattını Nasıl Kurar?

```
1  int fd[2];
2  pipe(fd);                /* 3: okuma, 4: yazma */
▶  if (fork() == 0) {       /* çocuk 1 */
4      dup2(fd[1], 1);
5      close(fd[0]); close(fd[1]);
6      execlp("ls", "ls", NULL);
7  }
8  if (fork() == 0) {       /* çocuk 2 */
9      dup2(fd[0], 0);
10     close(fd[0]); close(fd[1]);
11     execlp("wc", "wc", "-l", NULL);
12 }
13 /* kabuk: iki ucu da kapat */
14 close(fd[0]); close(fd[1]);
15 wait(NULL); wait(NULL);
```



`fork()` : çocuk, tanımlayıcı tablosunun **kopyasını** alır; iki süreç aynı boruyu işaret eder.

Adım 2 / 6

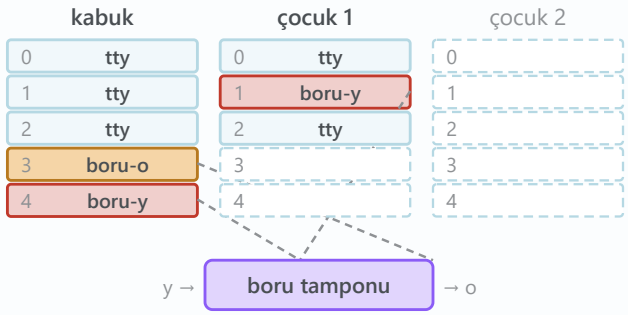
DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4` `çocuk1 = PID 2201`

ÇIKTI

Kabuk Bir Boru Hattını Nasıl Kurar?

```
1  int fd[2];
2  pipe(fd);                /* 3: okuma, 4: yazma */
3  if (fork() == 0) {     /* çocuk 1 */
▶      dup2(fd[1], 1);
▶      close(fd[0]); close(fd[1]);
6      execlp("ls", "ls", NULL);
7  }
8  if (fork() == 0) {     /* çocuk 2 */
9      dup2(fd[0], 0);
10     close(fd[0]); close(fd[1]);
11     execlp("wc", "wc", "-l", NULL);
12 }
13 /* kabuk: iki ucu da kapat */
14 close(fd[0]); close(fd[1]);
15 wait(NULL); wait(NULL);
```



`dup2(4, 1)` : stdout artık borunun yazma ucu. Fazla uçlar kapatılır.

Adım 3 / 6

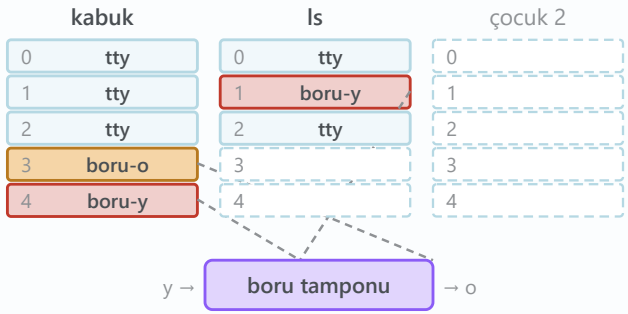
DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4` `çocuk1 = PID 2201`

ÇIKTI

Kabuk Bir Boru Hattını Nasıl Kurar?

```
1  int fd[2];
2  pipe(fd);                /* 3: okuma, 4: yazma */
3  if (fork() == 0) {     /* çocuk 1 */
4      dup2(fd[1], 1);
5      close(fd[0]); close(fd[1]);
▶      execlp("ls", "ls", NULL);
7  }
8  if (fork() == 0) {     /* çocuk 2 */
9      dup2(fd[0], 0);
10     close(fd[0]); close(fd[1]);
11     execlp("wc", "wc", "-l", NULL);
12 }
13 /* kabuk: iki ucu da kapat */
14 close(fd[0]); close(fd[1]);
15 wait(NULL); wait(NULL);
```



`execlp` görüntüyü `ls` ile değiştirir; tanımlayıcı tablosu **korunur**. ls hiçbir şey bilmeden boruya yazar.

Adım 4 / 6

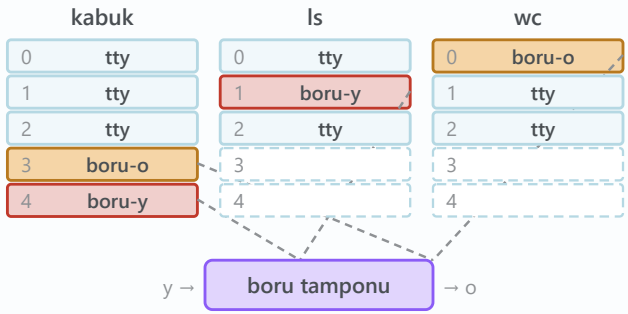
DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4` `çocuk1 = ls`

ÇIKTI

Kabuk Bir Boru Hattını Nasıl Kurar?

```
1  int fd[2];
2  pipe(fd);                /* 3: okuma, 4: yazma */
3  if (fork() == 0) {     /* çocuk 1 */
4      dup2(fd[1], 1);
5      close(fd[0]); close(fd[1]);
6      execlp("ls", "ls", NULL);
7  }
▶  if (fork() == 0) {     /* çocuk 2 */
▶      dup2(fd[0], 0);
▶      close(fd[0]); close(fd[1]);
▶      execlp("wc", "wc", "-l", NULL);
12 }
13 /* kabuk: iki ucu da kapat */
14 close(fd[0]); close(fd[1]);
15 wait(NULL); wait(NULL);
```



İkinci çocuk stdin'ini borunun okuma ucuna bağlar ve `wc -l` olur.

Adım 5 / 6

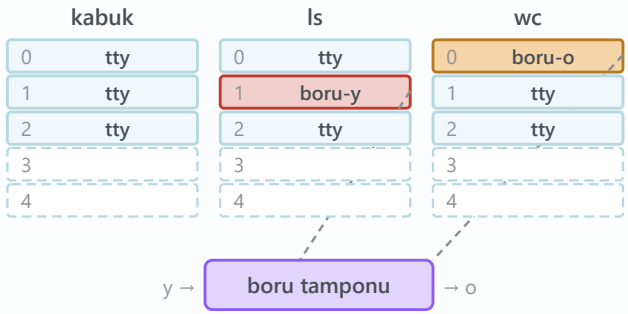
DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4` `çocuk1 = ls` `çocuk2 = wc`

ÇIKTI

Kabuk Bir Boru Hattını Nasıl Kurar?

```
1  int fd[2];
2  pipe(fd);                /* 3: okuma, 4: yazma */
3  if (fork() == 0) {     /* çocuk 1 */
4      dup2(fd[1], 1);
5      close(fd[0]); close(fd[1]);
6      execlp("ls", "ls", NULL);
7  }
8  if (fork() == 0) {     /* çocuk 2 */
9      dup2(fd[0], 0);
10     close(fd[0]); close(fd[1]);
11     execlp("wc", "wc", "-l", NULL);
12 }
▶ /* kabuk: iki ucu da kapat */
▶ close(fd[0]); close(fd[1]);
▶ wait(NULL); wait(NULL);
```



Kabuk iki ucu da kapatır. Kapatmasaydı yazma ucu açık kalır, wc **hiç EOF görmez** ve sonsuza dek beklerdi. Sonra iki çocuğu bekler.

Adım 6 / 6

DEĞİŞKENLER

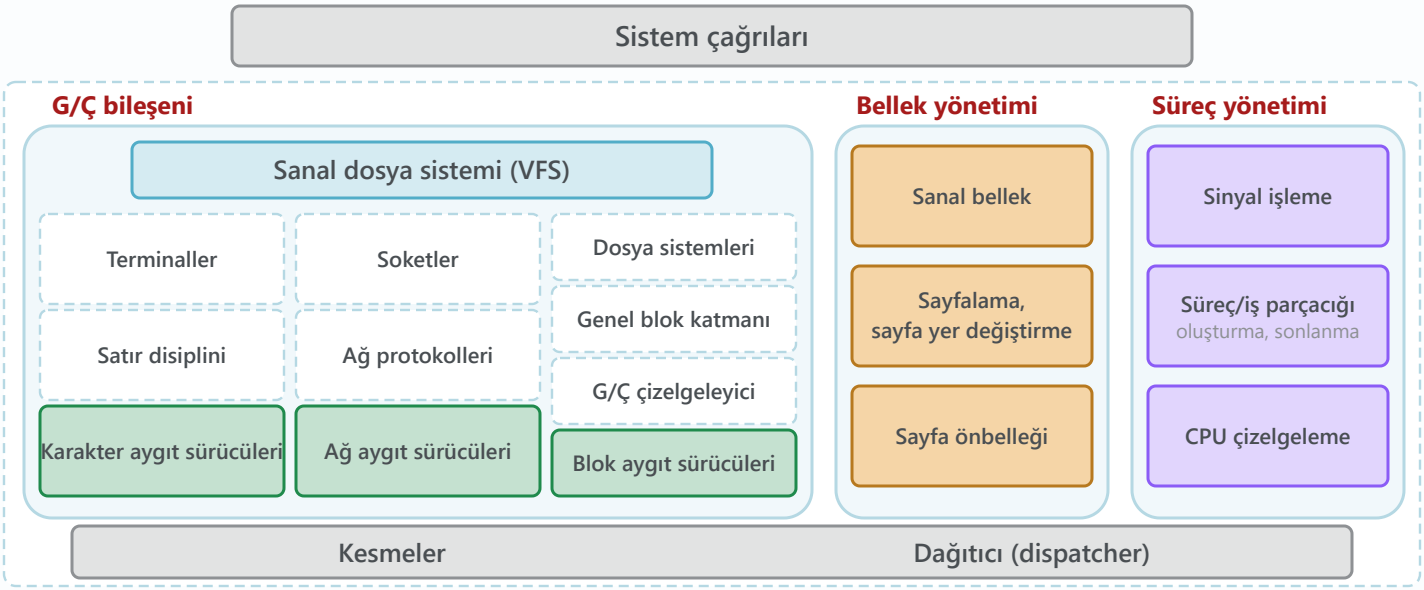
`fd[0] = –` `fd[1] = –` `çocuk1 = bitti` `çocuk2 = bitti`

ÇIKTI

```
7
```

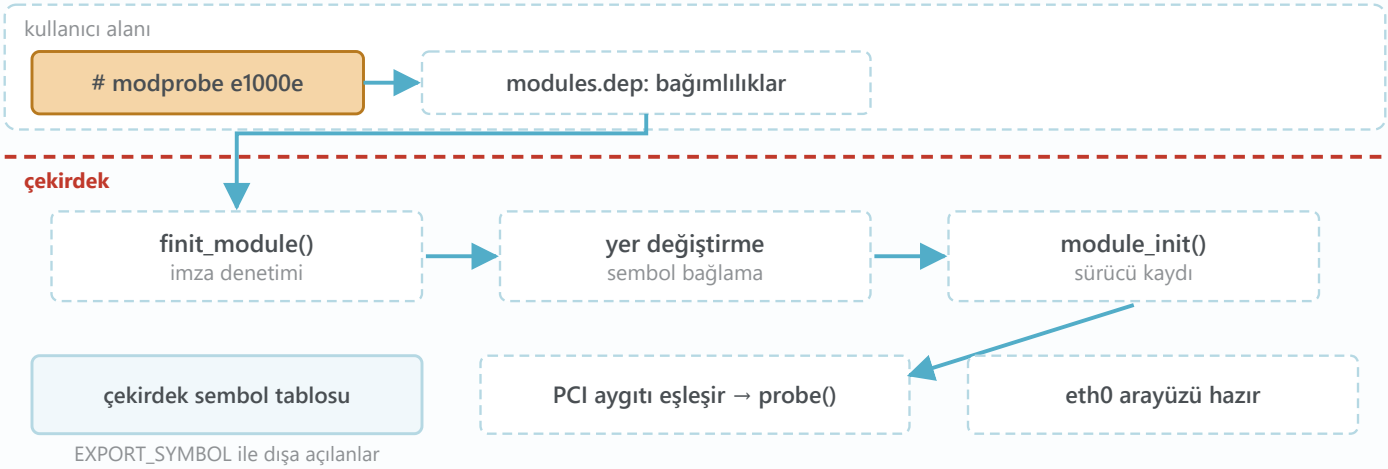
ÇEKİRDEK YAPISI

Linux Çekirdeğinin Yapısı



Üstte sistem çağrısı arayüzü, altta kesmeler ve dağıtıcı (bağlam değiştirme). Tüm bileşenler **tek adres alanında** çalışır: Linux monolitik bir çekirdektir. Klasik 4.4BSD çekirdeği de aynı üç bileşene ayrılıyordu.

Yüklenebilir Çekirdek Modülleri (LKM)

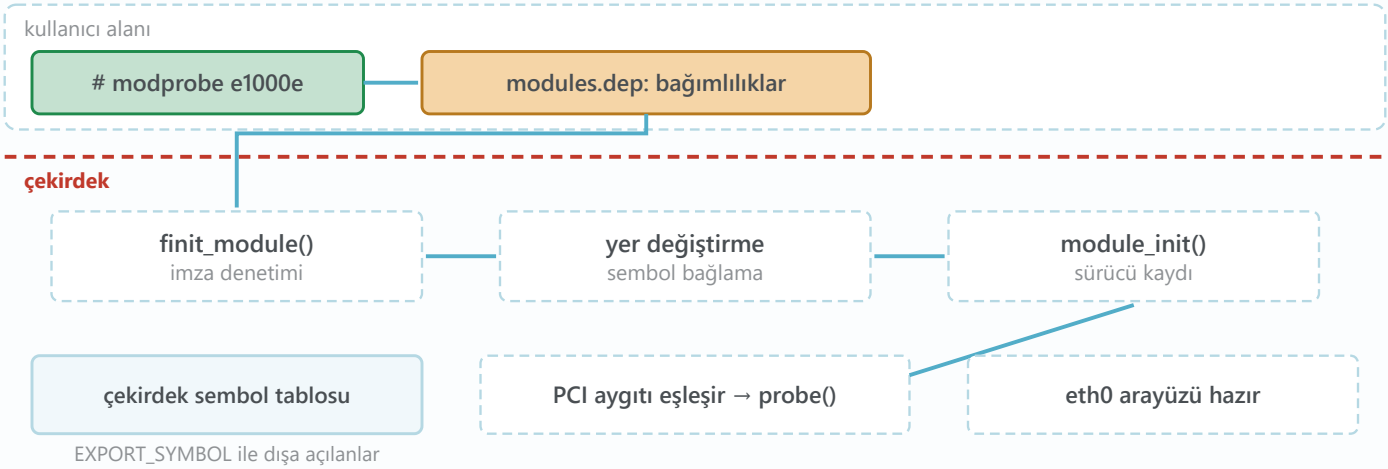


Monolitik ama **modüler**: sürücülerin çoğu yüklenebilir çekirdek modülüdür (`.ko`). Yönetici `modprobe` ile ister ya da udev aygıt görününce otomatik yükler.

Adım 1 / 6

Kaynak ağacı ~40 milyon satır; büyük kısmı sürücüler. 6.1'den beri çekirdekte Rust desteği var, ilk Rust sürücüsü 6.8'de girdi; eBPF güvenli, doğrulanmış küçük programları çekirdekte çalıştırır.

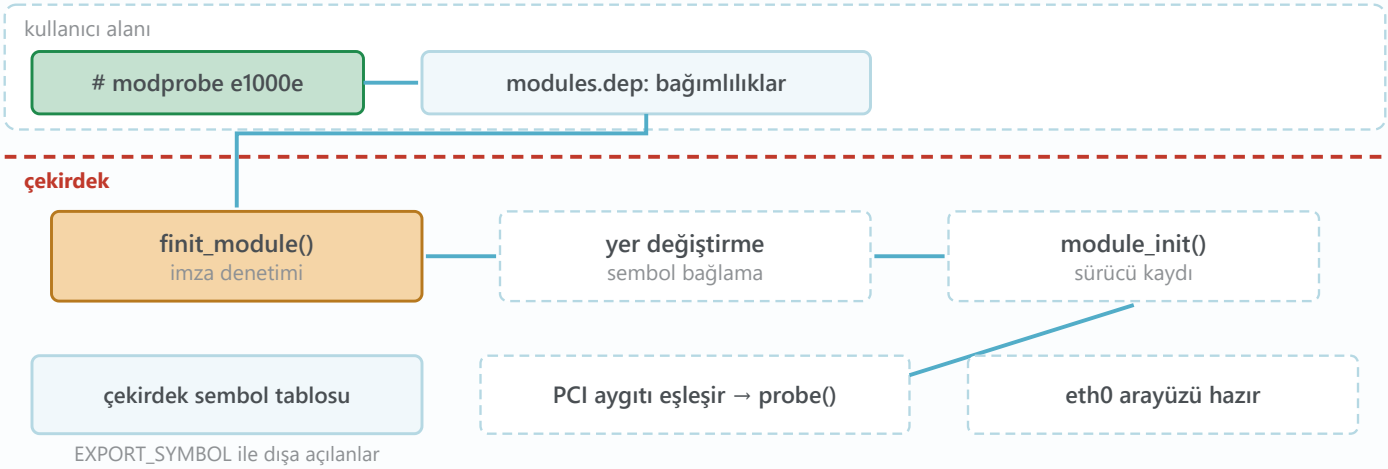
Yüklenebilir Çekirdek Modülleri (LKM)



`modprobe`, `/lib/modules/$(uname -r)/modules.dep` dosyasından bağımlılıkları bulur ve önce onları yükler (`insmod` bunu yapmaz).

Kaynak ağacı ~40 milyon satır; büyük kısmı sürücüler. 6.1'den beri çekirdekte Rust desteği var, ilk Rust sürücüsü 6.8'de girdi; eBPF güvenli, doğrulanmış küçük programları çekirdekte çalıştırır.

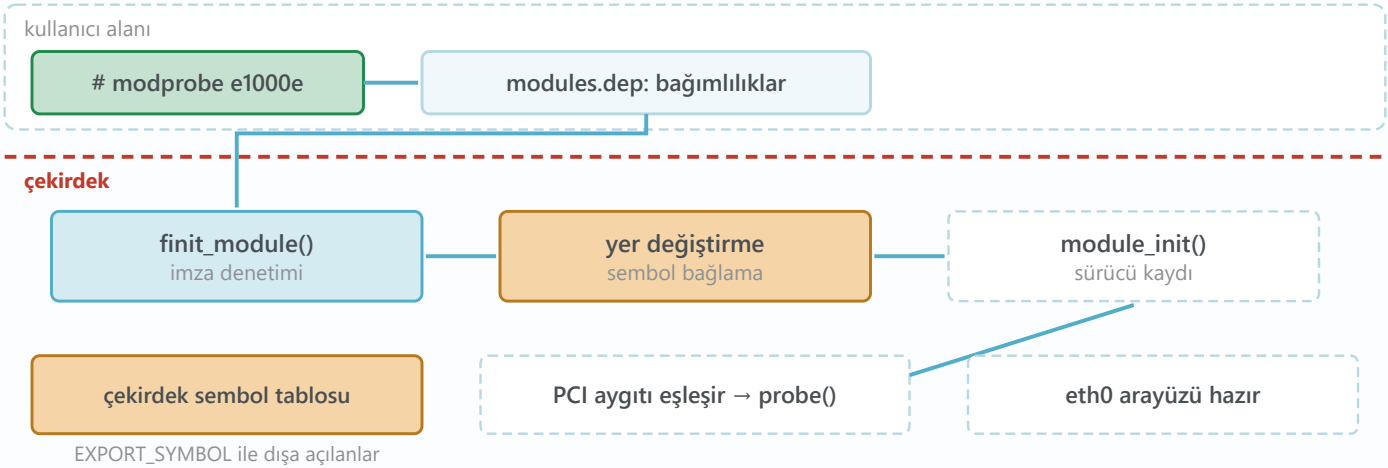
Yüklenebilir Çekirdek Modülleri (LKM)



`finit_module()` sistem çağrısı modülü çekirdeğe kopyalar; Secure Boot açıkken **imzası** doğrulanır. CAP_SYS_MODULE yeteneği gerekir.

Kaynak ağacı ~40 milyon satır; büyük kısmı sürücüler. 6.1'den beri çekirdekte Rust desteği var, ilk Rust sürücüsü 6.8'de girdi; eBPF güvenli, doğrulanmış küçük programları çekirdekte çalıştırır.

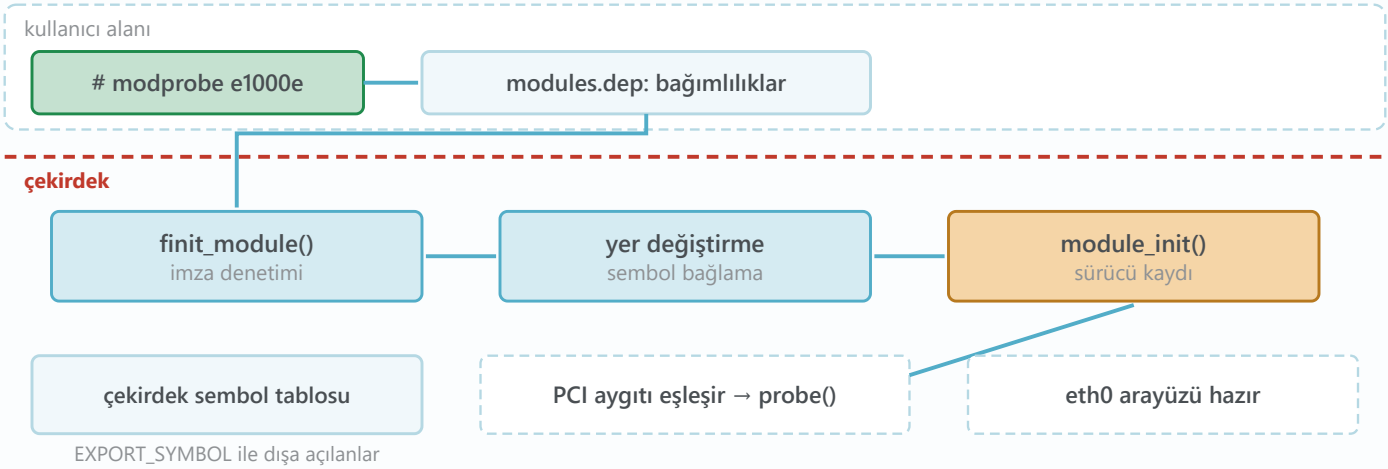
Yüklenebilir Çekirdek Modülleri (LKM)



Çekirdek modülün yer değiştirmelerini yapar ve kullandığı sembolleri `EXPORT_SYMBOL` ile dışa açılmış çekirdek fonksiyonlarına bağlar.

Kaynak ağacı ~40 milyon satır; büyük kısmı sürücüler. 6.1'den beri çekirdekte Rust desteği var, ilk Rust sürücüsü 6.8'de girdi; eBPF güvenli, doğrulanmış küçük programları çekirdekte çalıştırır.

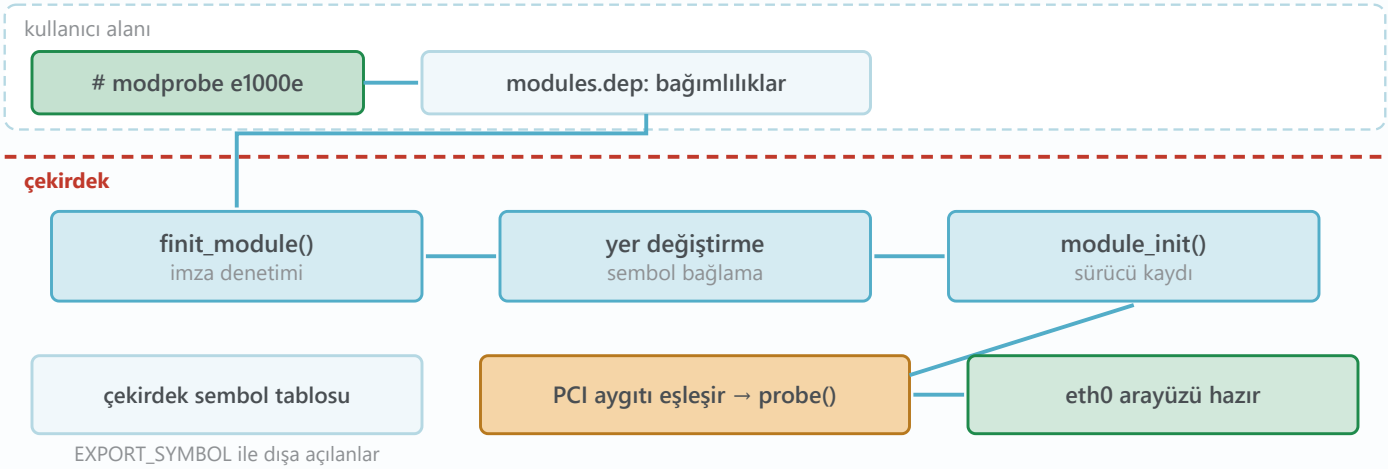
Yüklenebilir Çekirdek Modülleri (LKM)



`module_init` fonksiyonu çalışır: sürücü kendini PCI alt sistemine kaydeder (`pci_register_driver`).

Kaynak ağacı ~40 milyon satır; büyük kısmı sürücüler. 6.1'den beri çekirdekte Rust desteği var, ilk Rust sürücüsü 6.8'de girdi; eBPF güvenli, doğrulanmış küçük programları çekirdekte çalıştırır.

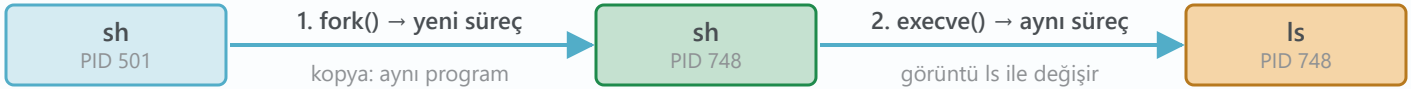
Yüklenebilir Çekirdek Modülleri (LKM)



Eşleşen aygıt için `probe()` çağrılır, ağ arayüzü oluşur. `lsmod` listeler, `rmmod` kaldırır. Modül çekirdek ayrıcalığıyla çalışır: hatası tüm sistemi çökertebilir.

Kaynak ağacı ~40 milyon satır; büyük kısmı sürücüler. 6.1'den beri çekirdekte Rust desteği var, ilk Rust sürücüsü 6.8'de girdi; eBPF güvenli, doğrulanmış küçük programları çekirdekte çalıştırır.

fork ve exec Çekirdekte Ne Yapar?



fork() çekirdekte

- Yeni `task_struct` ayır, ebeveynden doldur, yeni **PID** ata
- Çekirdek yığını ayır; yazmaçları kopyala (çocukta dönüş değeri 0)
- Sayfa tablolarını kopyala, yazılabilir sayfaları **salt okunur** işaretle (COW)
- Açık dosyaları paylaş: `struct file` referans sayısı artar
- Sinyal işleyicileri, cwd, kimlik bilgileri kopyalanır

execve() çekirdekte

- ELF başlığı okunur; eski `mm_struct` bırakılır, yenisi kurulur ve segmentler `mmap` edilir
- setuid/setgid bitine göre yeni `cred` hazırlanır
- PC ← `ld.so` (dinamik bağlayıcı) ya da programın giriş noktası
- Başarısız `execve` −1 döner; çocuk geleneksel olarak `_exit(127)` ile çıkar

Eski UNIX fork'ta veri ve yığını gerçekten kopyalardı; Linux yalnızca sayfa tablolarını kopyalar (COW). Çoğu fork'u hemen exec izlediği için kopyalanan sayfa sayısı çok azdır.

SÜREÇLER

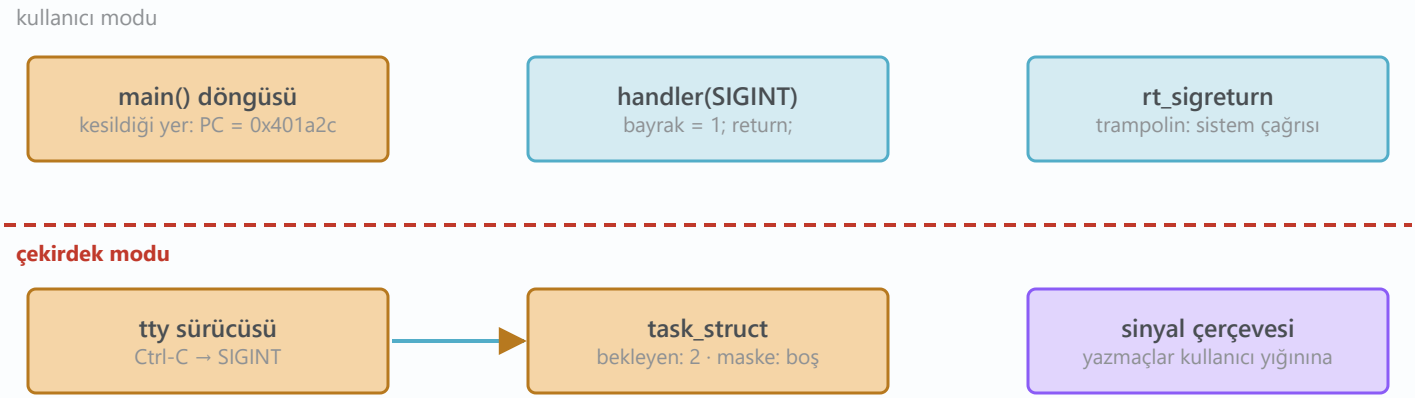
POSIX Sinyalleri

Sinyal, bir sürece **olay bildiren** yazılım kesmesidir. **Senkron** (SIGSEGV, SIGFPE: sürecin kendi komutu) ya da **eş zamansız** (SIGINT, SIGTERM: dışarıdan) olabilir. Süreç her sinyal için varsayılan eylemi, yok saymayı ya da bir **işleyici** (kaynak temizleme, günlük yazma) seçebilir.

Sinyal	No	Ne zaman üretilir?	Varsayılan
SIGHUP	1	Terminal bağlantısı koptu; cinlerde "yapılandırmayı yeniden oku"	sonlandır
SIGINT	2	Ctrl-C (ön plan süreç grubuna)	sonlandır
SIGQUIT	3	Ctrl-\	çekirdek dökümü
SIGILL	4	Geçersiz makine komutu	çekirdek dökümü
SIGABRT	6	`abort()` : süreç kendini anormal sonlandırır	çekirdek dökümü
SIGFPE	8	Tam sayı sıfıra bölme gibi aritmetik hata	çekirdek dökümü
SIGKILL	9	`kill -9` ; **yakalanamaz, yok sayılamaz**	sonlandır
SIGSEGV	11	Geçersiz bellek erişimi	çekirdek dökümü
SIGPIPE	13	Okuyanı kalmamış boruya yazma	sonlandır
SIGALRM	14	`alarm()` süresi doldu	sonlandır
SIGTERM	15	`kill` komutunun **varsayılanı**; nazik sonlandırma isteği	sonlandır
SIGUSR1/2	10/12	Kullanıcı tanımlı	sonlandır
SIGCHLD	17	Çocuk süreç bitti/durdu	yok say
SIGSTOP / SIGCONT	19/18	Durdur (yakalanamaz) / devam ettir; Ctrl-Z → SIGTSTP	durdur / sürdür

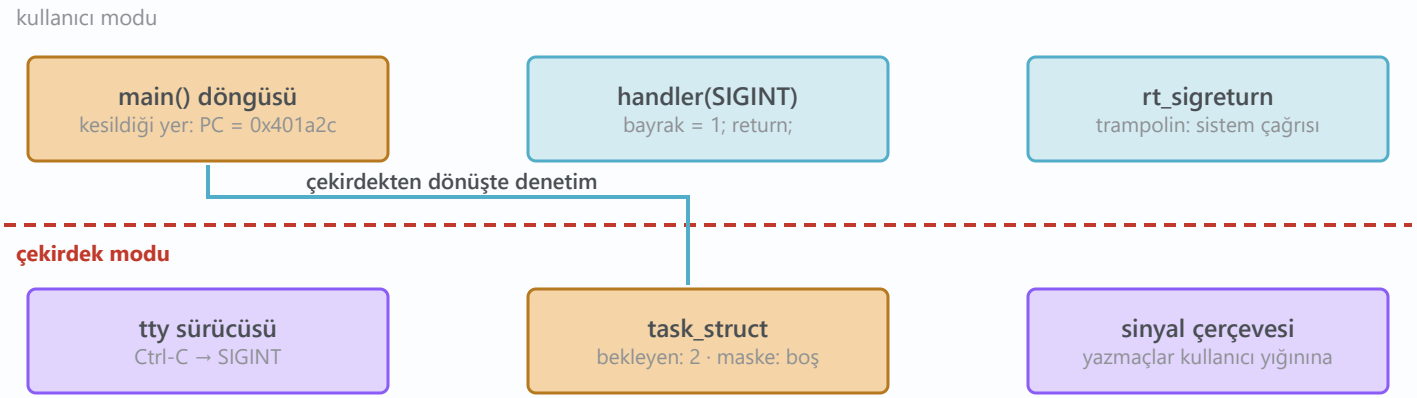
Numaralar x86/ARM Linux içindir. SIGKILL ve SIGTERM'in ikisi de `kill` komutuyla gönderilebilir; fark yakalanabilirlikte.

Sinyal Teslimi



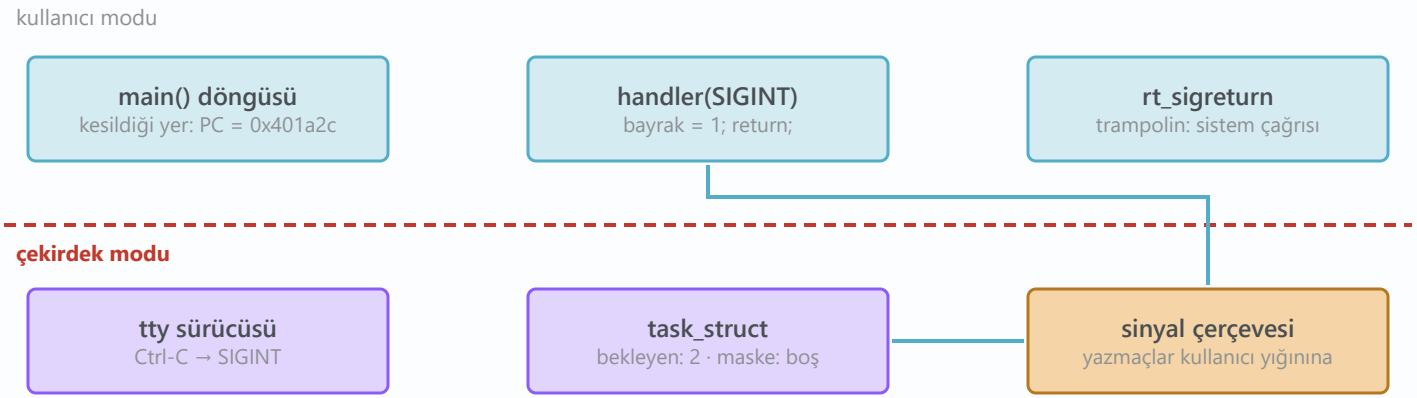
Kullanıcı Ctrl-C'ye basar: tty satır disiplini ön plan süreç grubuna **SIGINT** üretir. Çekirdek yalnızca hedef görevin **bekleyen** sinyal kümesinde 2. biti açar.

Sinyal Teslimi



Sinyal hemen işlenmez. Süreç bir sistem çağrısından ya da kesmeden **kullanıcı moduna dönerken** çekirdek "bekleyen ve maskelenmemiş" sinyal var mı diye bakar.

Sinyal Teslimi



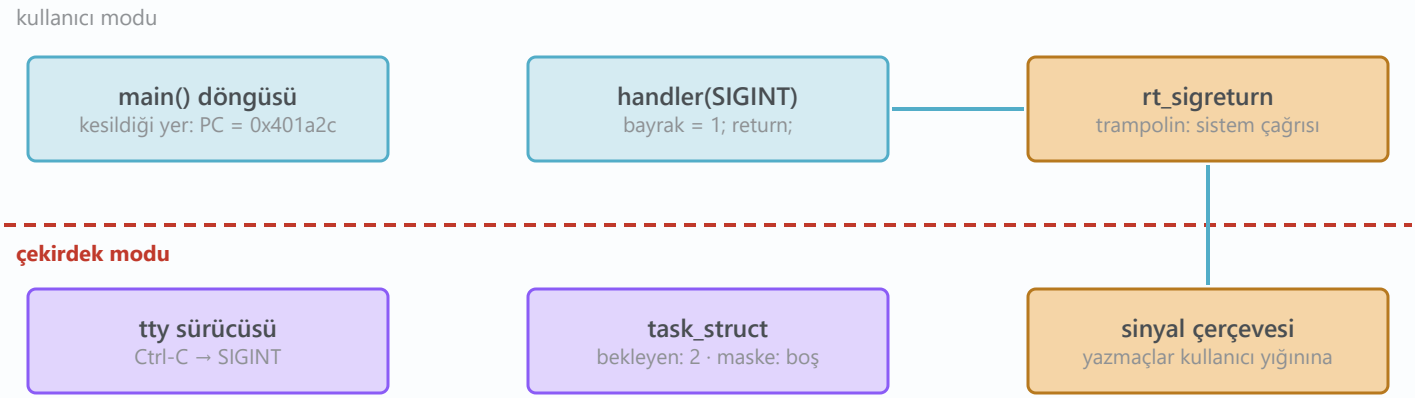
`sigaction` ile işleyici kurulmuşsa çekirdek kullanıcı yığınına bir **sinyal çerçevesi** (kaydedilmiş yazmaçlar + dönüş trampolini) yazar ve PC'yi işleyiciye ayarlar.

Sinyal Teslimi



İşleyici kullanıcı modunda çalışır. İçinde yalnızca async-signal-safe fonksiyonlar (ör. `write`, bir `volatile sig_atomic_t` bayrağı) güvenlidir; `printf`, `malloc` değil.

Sinyal Teslimi



İşleyici dönünce trampolin `rt_sigreturn` sistem çağrısını yapar: çekirdek çerçevedeki yazmaçları geri yükler. (sigreturn işleyiciye **girmez**, işleyiciden **döner**.)

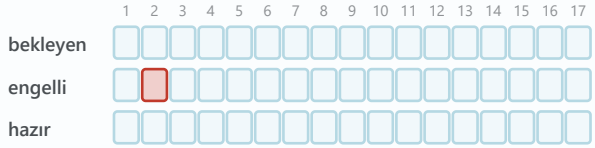
Sinyal Teslimi



Süreç kesildiği komuttan devam eder. İşleyici yoksa varsayılan eylem uygulanırdı: SIGINT için sonlandırma.

Bekleyen Sinyalleri Seçme: Kod İzleme

```
1  /* kullanıcı moduna dönerken (çekirdek) */
2  void sinyal_bak(task *t) {
3      unsigned long h = t->bekl & ~t->engl;
4      while (h) {
5          int sig = __builtin_ctzl(h) + 1;
6          t->bekl &= ~(1UL << (sig - 1));
7          teslim_et(t, sig); /* işleyici/varsayılan */
8          h = t->bekl & ~t->engl;
9      }
10 }
▶ /* önce:  sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* sonra: sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

SIGINT engellendi

Süreç kritik bir bölüme girmeden SIGINT'i sigprocmask ile engelledi (engelli kümesinde 2. bit). Sinyaller her görevde bir bit kümesidir.

Adım 1 / 8

DEĞİŞKENLER

`bekl = {}` `engl = {2}`

ÇIKTI

Bekleyen Sinyalleri Seçme: Kod İzleme

```
1  /* kullanıcı moduna dönerken (çekirdek) */
2  void sinyal_bak(task *t) {
3      unsigned long h = t->bekl & ~t->engl;
4      while (h) {
5          int sig = __builtin_ctzl(h) + 1;
6          t->bekl &= ~(1UL << (sig - 1));
7          teslim_et(t, sig); /* işleyici/varsayılan */
8          h = t->bekl & ~t->engl;
9      }
10 }
▶ /* önce:  sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* sonra: sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

Ctrl-C, kill -USR1, çocuk bitti

Süreç uyurken üç sinyal gelir: Ctrl-C (2), kill -USR1 (10) ve bir çocuğun bitişi (17). Çekirdek yalnızca bekleyen kümesinde bitleri açar; hiçbir şey hemen çalışmaz.

Adım 2 / 8

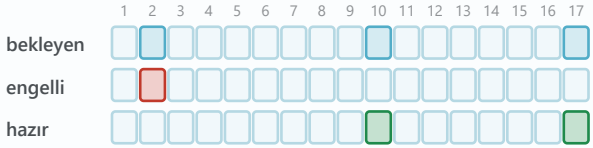
DEĞİŞKENLER

`bekl = {2,10,17}` `engl = {2}`

ÇIKTI

Bekleyen Sinyalleri Seçme: Kod İzleme

```
1  /* kullanıcı moduna dönerken (çekirdek) */
▶  void sinyal_bak(task *t) {
▶      unsigned long h = t->bekl & ~t->engl;
4      while (h) {
5          int sig = __builtin_ctzl(h) + 1;
6          t->bekl &= ~(1UL << (sig - 1));
7          teslim_et(t, sig); /* işleyici/varsayılan */
8          h = t->bekl & ~t->engl;
9      }
10 }
11 /* önce:  sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* sonra: sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

hazır = bekleyen & ~engelli = {10, 17}

Süreç kullanıcı moduna dönerken çekirdek bakar: hazır = bekleyen & ~engelli. SIGINT bekliyor ama engelli; teslime hazır olanlar **10** ve **17**.

Adım 3 / 8

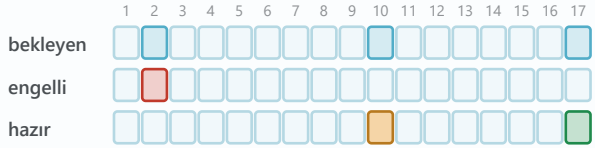
DEĞİŞKENLER

h = {10,17}

ÇIKTI

Bekleyen Sinyalleri Seçme: Kod İzleme

```
1  /* kullanıcı moduna dönerken (çekirdek) */
2  void sinyal_bak(task *t) {
3      unsigned long h = t->bekl & ~t->engl;
▶      while (h) {
▶          int sig = __builtin_ctzl(h) + 1;
▶          t->bekl &= ~(1UL << (sig - 1));
▶          teslim_et(t, sig); /* işleyici/varsayılan */
▶          h = t->bekl & ~t->engl;
9      }
10 }
11 /* önce:  sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* sonra: sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

SIGUSR1 → işleyici çalışır

En küçük numaralı hazır bit (ctz): **10**. Bekleyenden silinir ve teslim edilir: SIGUSR1 → işleyici çalışır. Hazır küme yeniden hesaplanır.

Adım 4 / 8

DEĞİŞKENLER

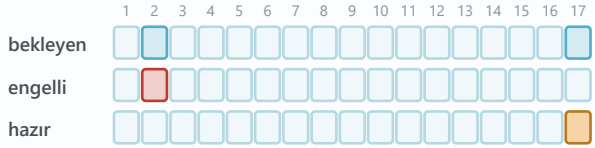
`sig = 10` `bekl = {2,17}`

ÇIKTI

```
teslim: 10 (SIGUSR1)
```

Bekleyen Sinyalleri Seçme: Kod İzleme

```
1  /* kullanıcı moduna dönerken (çekirdek) */
2  void sinyal_bak(task *t) {
3      unsigned long h = t->bekl & ~t->engl;
▶      while (h) {
▶          int sig = __builtin_ctzl(h) + 1;
▶          t->bekl &= ~(1UL << (sig - 1));
▶          teslim_et(t, sig); /* işleyici/varsayılan */
▶          h = t->bekl & ~t->engl;
9      }
10 }
11 /* önce:  sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* sonra: sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

SIGCHLD → varsayılan: yok say

En küçük numaralı hazır bit (ctz): **17**. Bekleyenden silinir ve teslim edilir: SIGCHLD → varsayılan: yok say. Hazır küme yeniden hesaplanır.

Adım 5 / 8

DEĞİŞKENLER

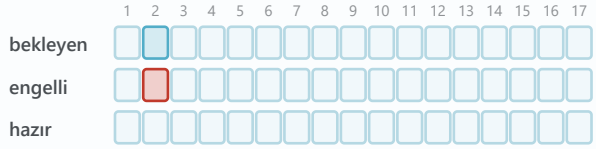
`sig = 17` `bekl = {2}`

ÇIKTI

```
teslim: 10 (SIGUSR1)
teslim: 17 (SIGCHLD)
```

Bekleyen Sinyalleri Seçme: Kod İzleme

```
1  /* kullanıcı moduna dönerken (çekirdek) */
2  void sinyal_bak(task *t) {
3      unsigned long h = t->bekl & ~t->engl;
▶      while (h) {
5          int sig = __builtin_ctzl(h) + 1;
6          t->bekl &= ~(1UL << (sig - 1));
7          teslim_et(t, sig); /* işleyici/varsayılan */
8          h = t->bekl & ~t->engl;
9      }
▶  }
11 /* önce:  sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* sonra: sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

hazır boş: döngü biter, SIGINT bekliyor

Hazır küme boş, döngü biter ve süreç kullanıcı modunda sürer. SIGINT kaybolmadı: bekleyen kümesinde duruyor.

Adım 6 / 8

DEĞİŞKENLER

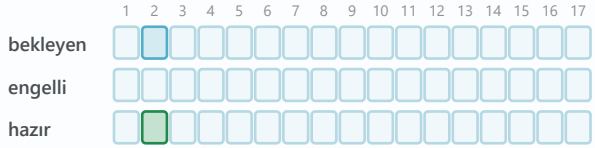
`h = {}` `bekl = {2}`

ÇIKTI

```
teslim: 10 (SIGUSR1)
teslim: 17 (SIGCHLD)
```

Bekleyen Sinyalleri Seçme: Kod İzleme

```
1  /* kullanıcı moduna dönerken (çekirdek) */
2  void sinyal_bak(task *t) {
▶      unsigned long h = t->bekl & ~t->engl;
4      while (h) {
5          int sig = __builtin_ctzl(h) + 1;
6          t->bekl &= ~(1UL << (sig - 1));
7          teslim_et(t, sig); /* işleyici/varsayılan */
8          h = t->bekl & ~t->engl;
9      }
10 }
11 /* önce:  sigprocmask(SIG_BLOCK, {SIGINT}) */
▶  /* sonra: sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

engel kalktı: hazır = {2}

Kritik bölüm bitti, SIGINT'in engeli kaldırıldı. sigprocmask'tan dönüşte aynı denetim yapılır: hazır = {2}.

Adım 7 / 8

DEĞİŞKENLER

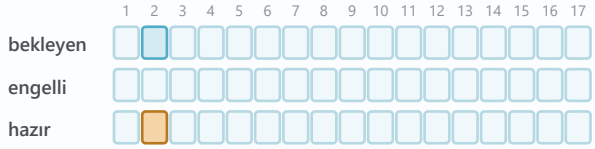
`engl = {}` `h = {2}`

ÇIKTI

```
teslim: 10 (SIGUSR1)
teslim: 17 (SIGCHLD)
```

Bekleyen Sinyalleri Seçme: Kod İzleme

```
1  /* kullanıcı moduna dönerken (çekirdek) */
2  void sinyal_bak(task *t) {
3      unsigned long h = t->bekl & ~t->engl;
4      while (h) {
▶          int sig = __builtin_ctzl(h) + 1;
▶          t->bekl &= ~(1UL << (sig - 1));
▶          teslim_et(t, sig); /* işleyici/varsayılan */
8          h = t->bekl & ~t->engl;
9      }
10 }
11 /* önce:  sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* sonra: sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

SIGINT → varsayılan: sonlandır

SIGINT teslim edilir; işleyici kurulmadığı için varsayılan eylem: süreç sonlanır. Engellemek sinyali ertelemek demektir, yok saymak değil. Aynı sinyal birden çok gelseydi tek bit olarak birleşirdi.

Adım 8 / 8

DEĞİŞKENLER

`sig = 2` `bekl = {}`

ÇIKTI

```
teslim: 10 (SIGUSR1)
teslim: 17 (SIGCHLD)
teslim: 2 (SIGINT → varsayılan:
sonlandır)
```

Süreç Yönetimi Sistem Çağrıları

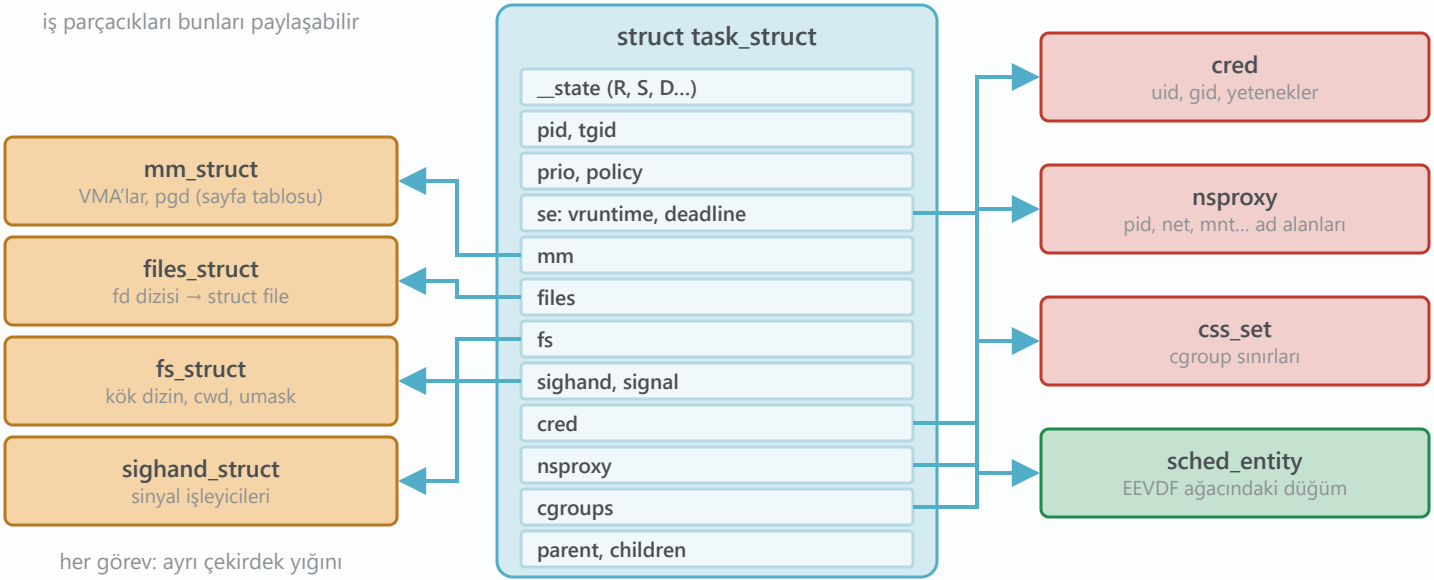
Sistem çağrısı	Açıklama
`fork()`	Süreci kopyalayarak yeni süreç oluşturur (COW)
`clone()`, `clone3()`	Neyin paylaşılacağı bayraklarla seçilen yeni görev (iş parçacığı, konteyner)
`execve()`	Mevcut süreç görüntüsünü yeni programla değiştirir
`waitpid()`	Çocuğun bitmesini bekler, çıkış durumunu alır (zombiyi toplar)
`exit()` / `_exit()`	Çağıran süreci sonlandırır (exit stdio tamponlarını boşaltır)
`kill()`	Sürece **sinyal gönderir**; sinyal sonlandırma olmak zorunda değil
`sigaction()`	Bir sinyal için işleyici ya da eylem atar
`sigprocmask()`	Sinyal maskesini (bloke edilen sinyaller) okur/değiştirir
`sigpending()`	Bloke olduğu için bekleyen sinyalleri döndürür
`sigsuspend()`	Maskeyi geçici değiştirir ve bir sinyal gelene kadar uyur
`rt_sigreturn()`	İşleyiciden dönüşte kaydedilmiş bağlamı geri yükler
`alarm()` / `pause()`	n saniye sonra SIGALRM / bir sinyal gelene kadar uyu



kill()/sigaction() ile sinyal, alarm()/pause() ile zamanlama her aşamada araya girebilir

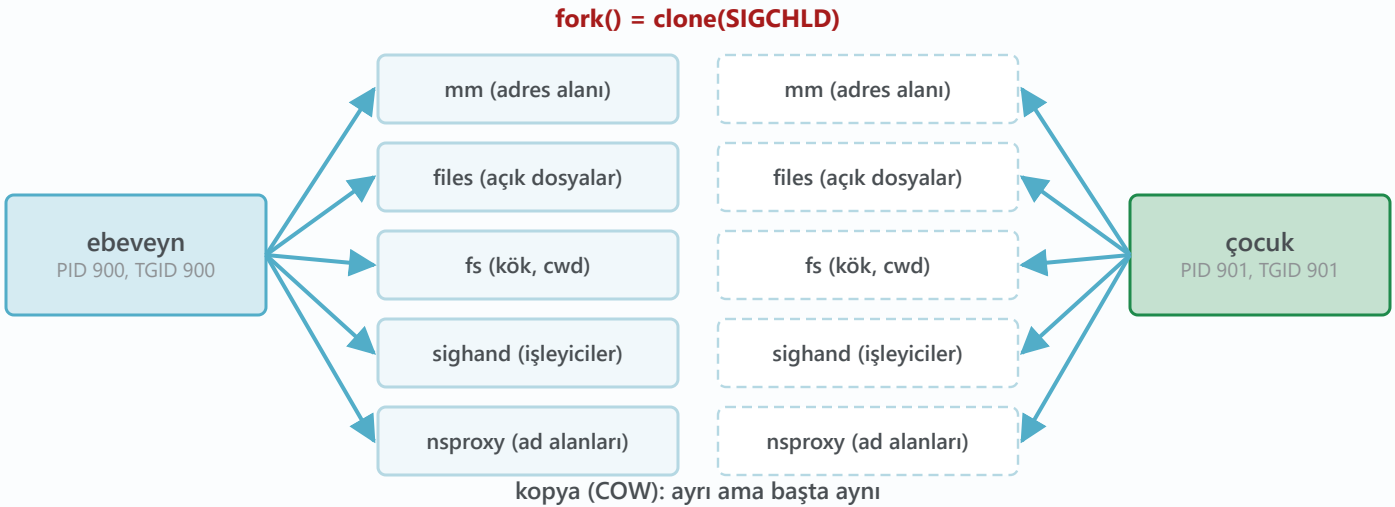
Sistem çağrısı, kullanıcı programının çekirdekten hizmet istemesinin tek yoludur; Linux x86-64'te ~460 tane vardır. Süreç ağacı `pstree -p` ya da `ps -ef --forest` ile görülür.

Süreç Tanımlayıcısı: task_struct



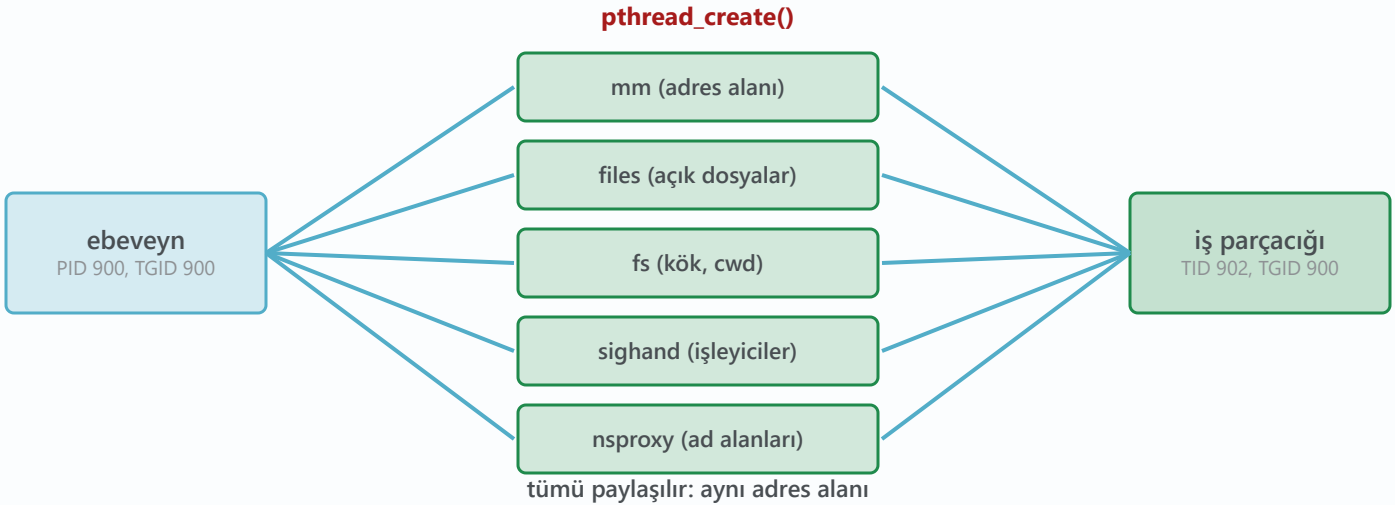
Linux'ta süreç ile iş parçacığı arasında yapısal fark yoktur: ikisi de bir **görev** (task_struct). Fark, hangi alt yapıları **paylaştıklarıdır**. Kullanıcının gördüğü PID aslında `tgid` 'dir (iş parçacığı grubu kimliği). `__state` D (kesilemez uyku) iken görev SIGKILL ile bile hemen ölmez; bu yüzden çoğu bekleme artık TASK_KILLABLE kullanır.

clone() ve İş Parçacıkları



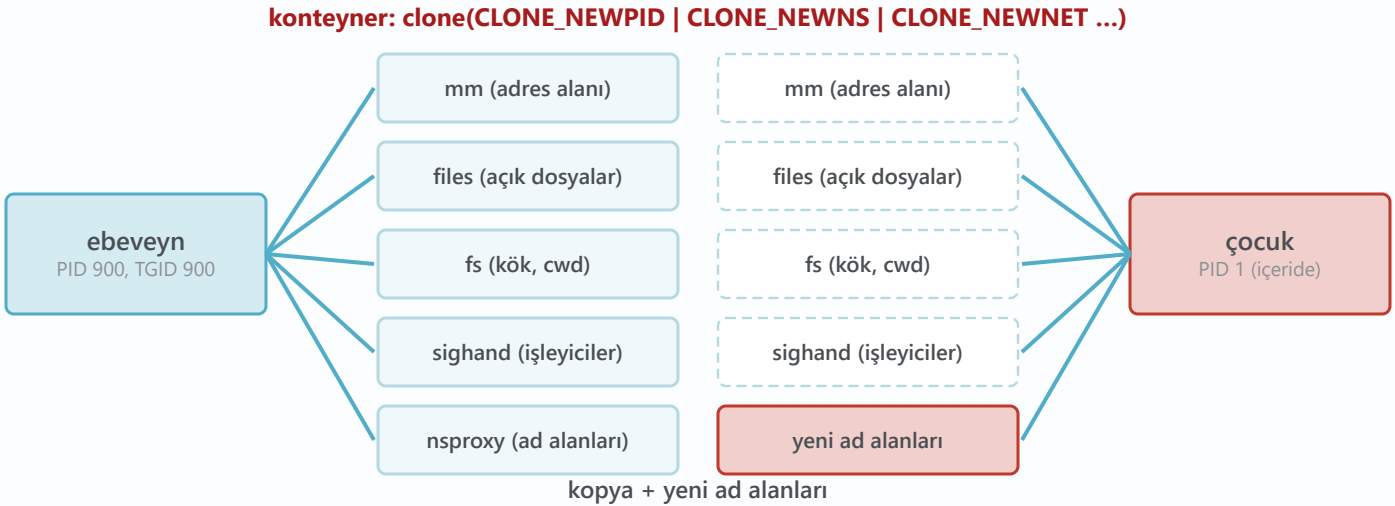
`fork()` glibc'de yalnız `SIGCHLD` bayrağıyla çağrılan bir `clone()`'dur: çocuk her yapının **kopyasını** alır (bellek COW ile). Ayrı PID, ayrı TGID: yeni bir süreç.

clone() ve İş Parçacıkları



`pthread_create()` → `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD | …)`. Hepsi paylaşılır; yeni görev aynı TGID'e girer. **NPTL**: her iş parçacığı bir çekirdek görevi (1:1 model).

clone() ve İş Parçacıkları



Ad alanı bayrakları (`CLONE_NEWPID` , `CLONE_NEWNS` , `CLONE_NEWNET` ...) çocuğa yeni bir görünüm verir: içeride kendini PID 1 sanır. Konteynerlerin temeli budur.

CFS'ten EEVDF'e: Adil Paylaşım

Sürüm	Adil sınıfın seçim kuralı	Veri yapısı
2.4 (2001)	O(n): her seçimde tüm hazır görevler taranır	tek liste
2.6.0–2.6.22 (2003)	O(1): öncelik başına liste, etkin ve süresi dolmuş iki dizi	140 listelik dizi + biteşlem
2.6.23–6.5 (2007)	CFS: en küçük vruntime	kırmızı-siyah ağaç
6.6+ (2023)	EEVDF: uygun görevler (v ≤ V) içinde en erken sanal son tarih	aynı ağaç, alt ağaç bilgisiyle zenginleştirilmiş

- Değişmeyenler: `sched_entity` içindeki vruntime, nice → ağırlık tablosu (`sched_prio_to_weight`, nice 0 → 1024) ve çalışma kuyruğundaki ağaç.
- CFS'in gecikme ayarları (`sched_latency`, uyanma ayrıntı düzeyi) yerine tek bir temel dilim (`base_slice_ns`) kaldı; sanal son tarih d = v + dilim × 1024 / ağırlık.
- Gecikmeye duyarlı görev `sched_setattr()` ile daha kısa dilim isteyebilir (6.12+): kısa dilim, erken son tarih ve daha sık seçilme demektir.

Önyükleme: UEFI'den systemd'ye



Güç verilince **UEFI** ürün yazılımı donanımı sınar ve ESP bölümündeki EFI uygulamasını yükler. Secure Boot açıksa her aşama bir sonrakinin imzasını doğrular.

Önyükleme: UEFI'den systemd'ye



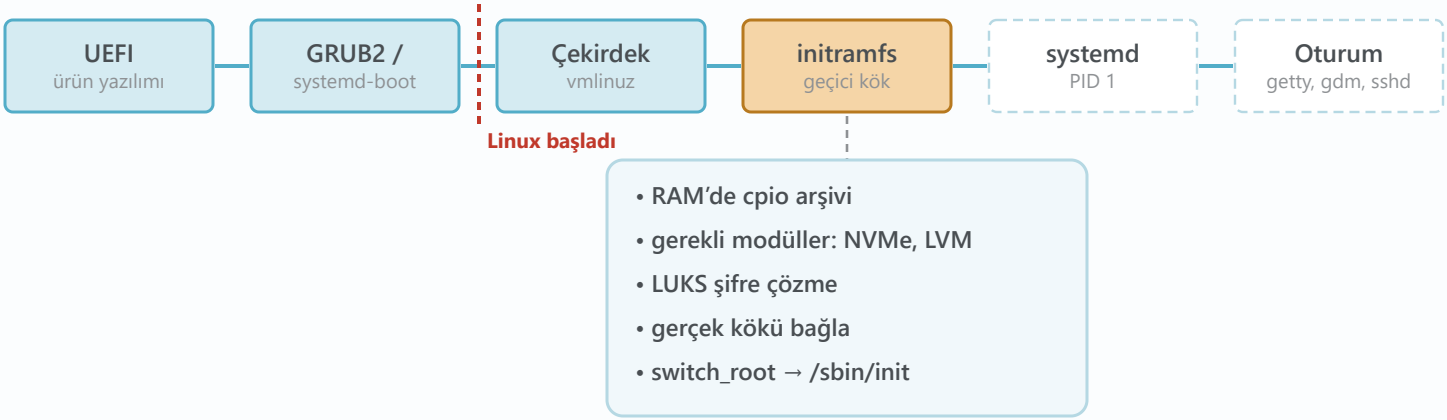
Önyükleyici çekirdek görüntüsünü (`vmlinuz`) ve `initramfs` 'i belleğe yükler, komut satırı parametrelerini verir. Çekirdek EFI uygulaması olarak doğrudan da başlatılabilir (UKI).

Önyükleme: UEFI'den systemd'ye



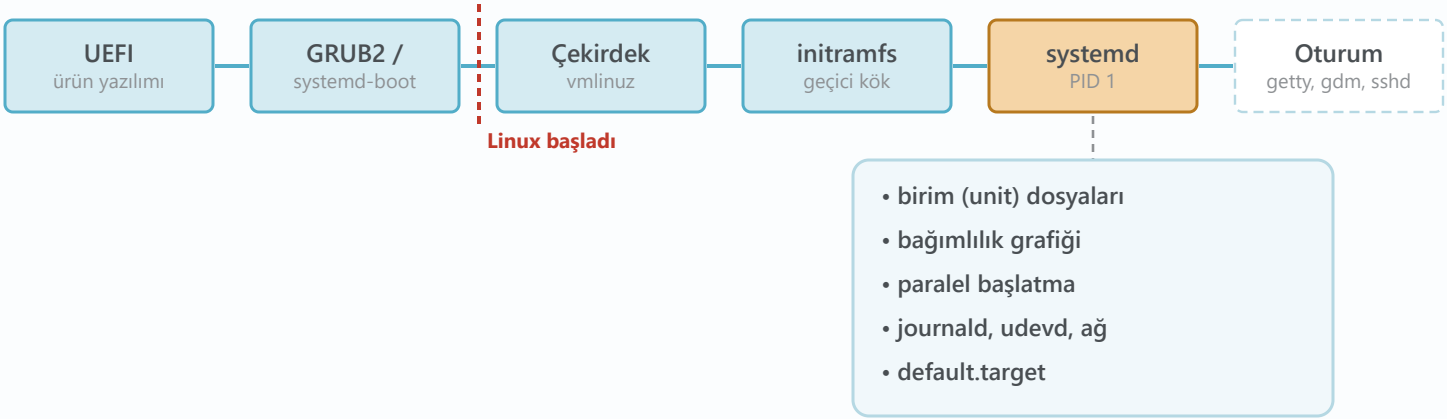
Çekirdek kendini açar, `start_kernel()` bellek yönetimini, kesmeleri, çizelgeleyiciyi kurar. İlk görev PID 0 boşta görevi olur; `kernel_init` (PID 1) ve `kthreadd` (PID 2) oluşturulur.

Önyükleme: UEFI'den systemd'ye



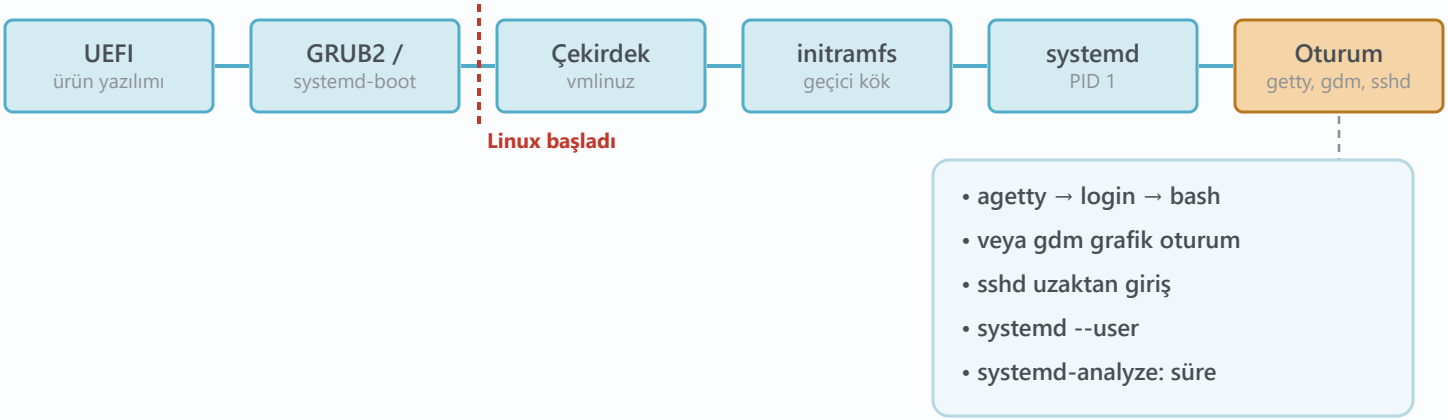
Kök dosya sistemini bağlamak için gereken sürücüler çekirdekte olmayabilir (NVMe, RAID, şifreli disk). **initramfs** RAM'de geçici bir kök sağlar, gerçek kökü bağlar ve `switch_root` yapar.

Önyükleme: UEFI'den systemd'ye



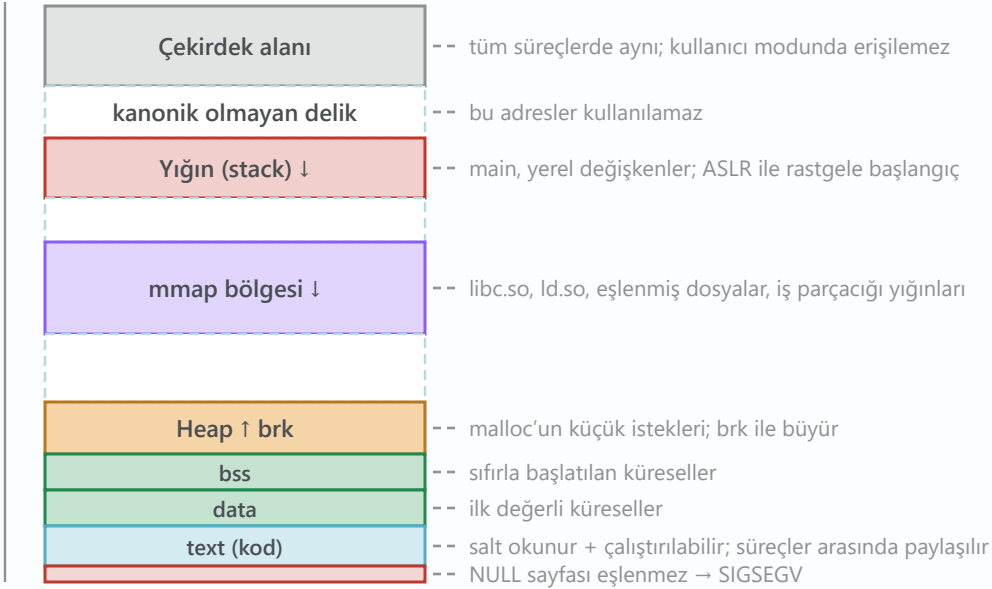
PID 1 artık **systemd**: birim dosyalarındaki bağımlılıklara göre hizmetleri **paralel** başlatır, hedefe (`multi-user.target` veya `graphical.target`) ulaşır.

Önyükleme: UEFI'den systemd'ye



Konsolda `agetty` → `login` → kabuk; masaüstünde gdm; uzaktan sshd. Klasik UNIX'teki init → getty → login → sh zinciri hâlâ tanınabilir.

Sürecin Sanal Adres Alanı



x86-64, 4 düzey (48 bit):

kullanıcı 2^{47} B = 128 TiB

çekirdek 128 TiB

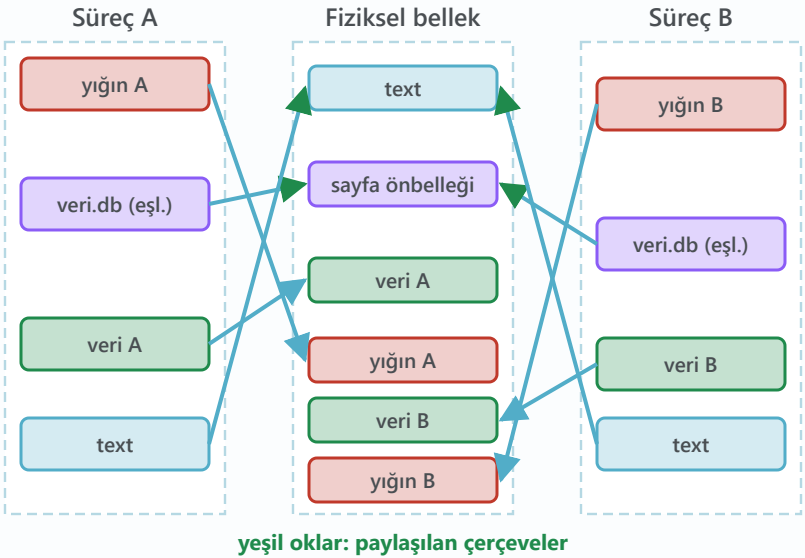
5 düzey, LA57 (57 bit):

kullanıcı 2^{56} B = 64 PiB

cat /proc/PID/maps

Her bölge bir **VMA** (vm_area_struct) ile tanımlanır; VMA'lar `mm_struct` içinde bir "maple tree"de tutulur. Yığın ve mmap aşağı, heap yukarı büyür; aradaki boşluk yalnızca sanaldır, fiziksel bellek harcamaz.

Paylaşım, Eşlenmiş Dosyalar ve Bellek Çağrıları

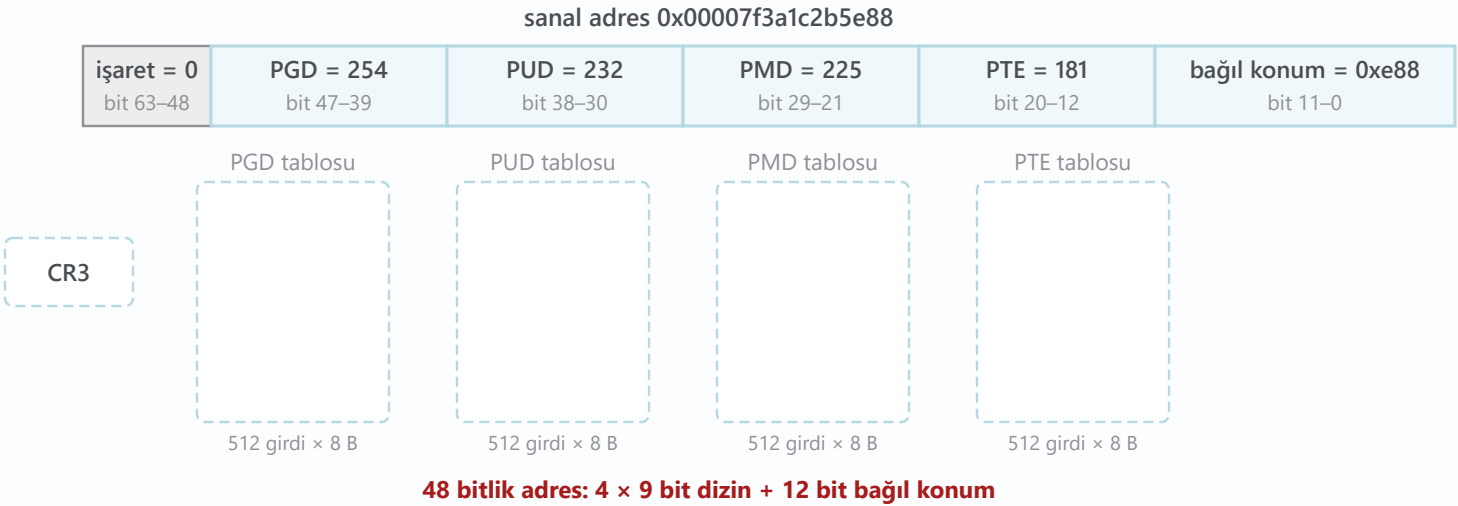


yeşil oklar: paylaşılan çerçeveler

Çağrı	İşlev
`brk` / `sbrk`	Heap'in (veri kesiminin) sonunu taşıyarak büyütür/küçültür
`mmap`	Dosyayı, aygıtı ya da anonim belleği adres alanına eşler
`munmap`	Eşlemeyi kaldırır, belleği sisteme geri verir
`mprotect`	Bölgenin izinlerini (r/w/x) değiştirir
`madvise`	Çekirdeğe kullanım ipucu verir (ör. MADV_DONTNEED)
`msync`	Paylaşımlı eşlemenin kirli sayfalarını dosyaya yazar

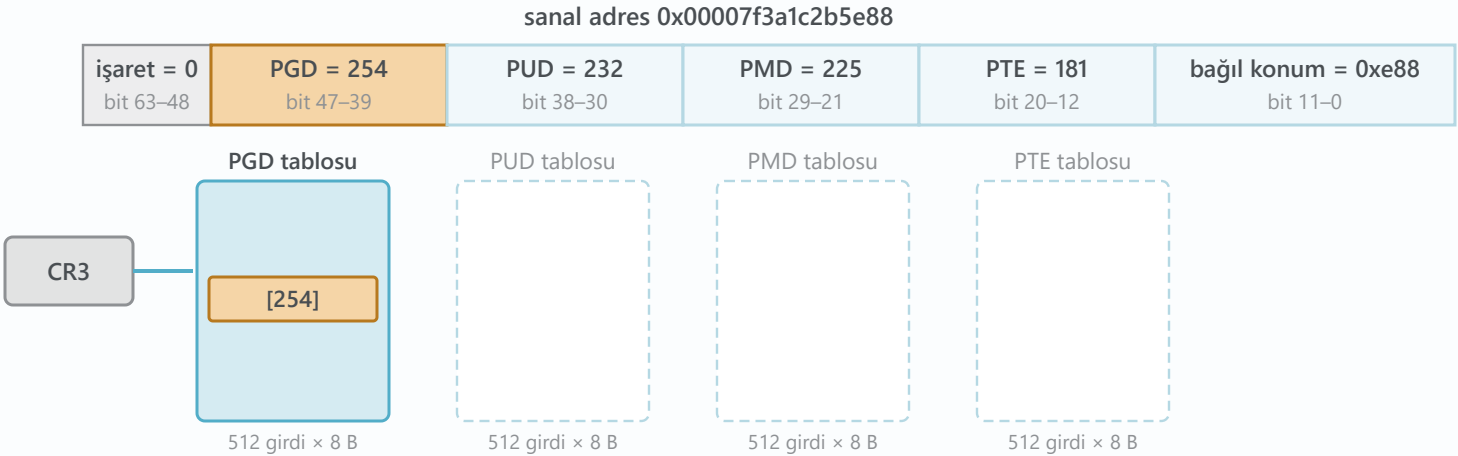
MAP_SHARED: yazılanlar dosyaya ve diğer süreçlere görünür.
MAP_PRIVATE: yazınca kopyala, dosya değişmez. glibc `malloc` küçük istekleri heap'ten, ≥ 128 KiB istekleri doğrudan `mmap` ile karşılar.

Sayfa Tablosu Yürüyüşü: 4 ve 5 Düzey



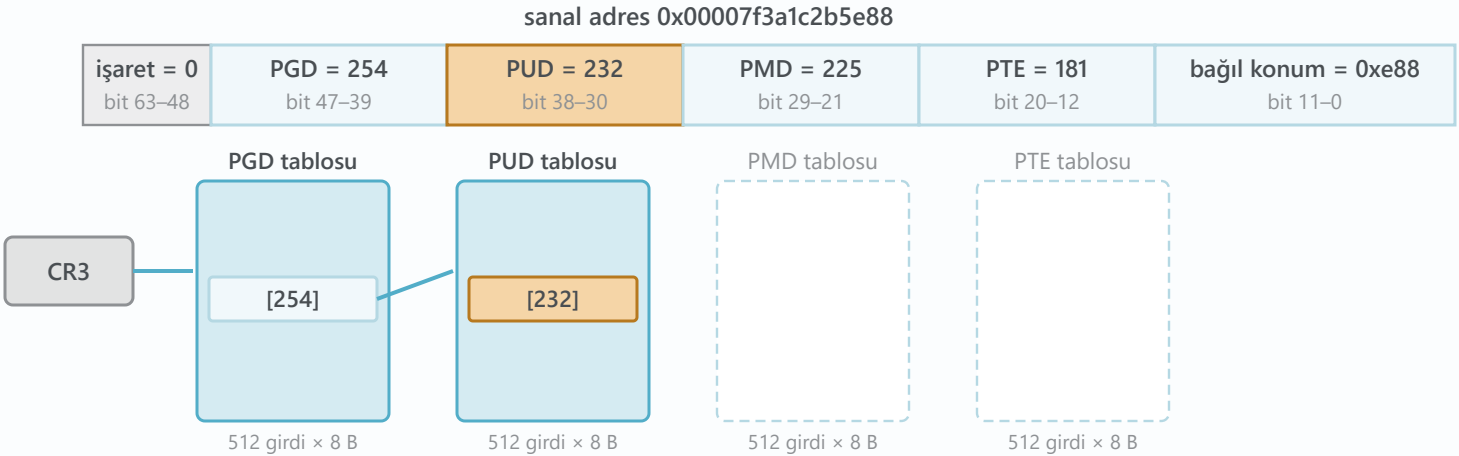
x86-64'te 4 KiB sayfa → 12 bit bağıl konum. Kalan 36 bit, her biri 512 girdilik (9 bit) dört tabloya bölünür. 63–48. bitler 47. bitin kopyasıdır (kanonik adres).

Sayfa Tablosu Yürüyüşü: 4 ve 5 Düzey



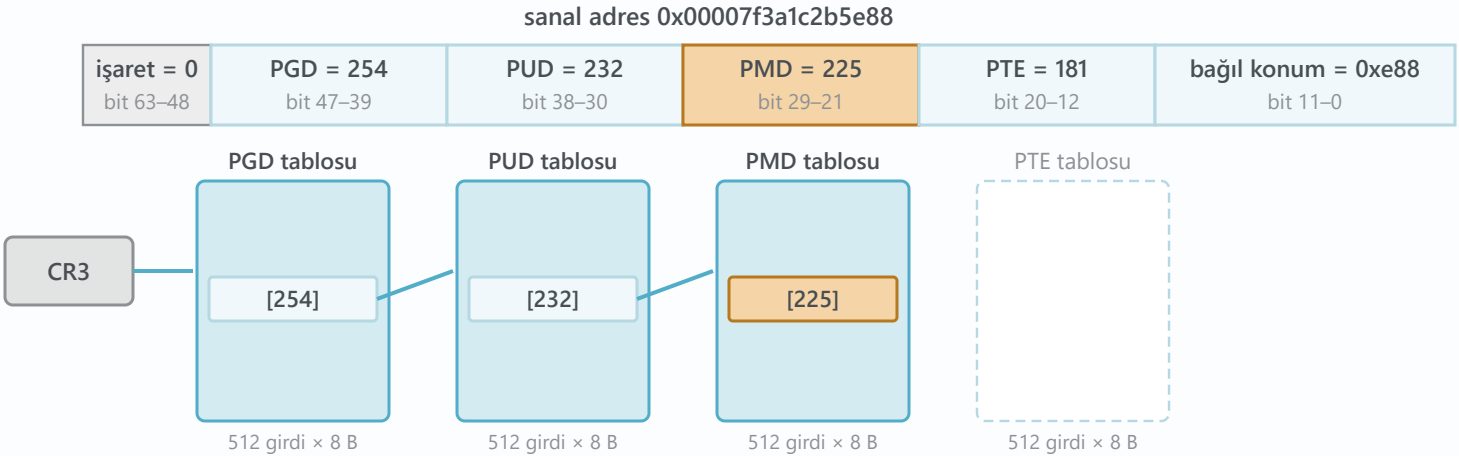
CR3 yazmacı sürecin en üst tablosunu (PGD) gösterir; bağlam değiştirmede CR3 değişir. Dizin 254 → PUD tablosunun adresi.

Sayfa Tablosu Yürüyüşü: 4 ve 5 Düzey



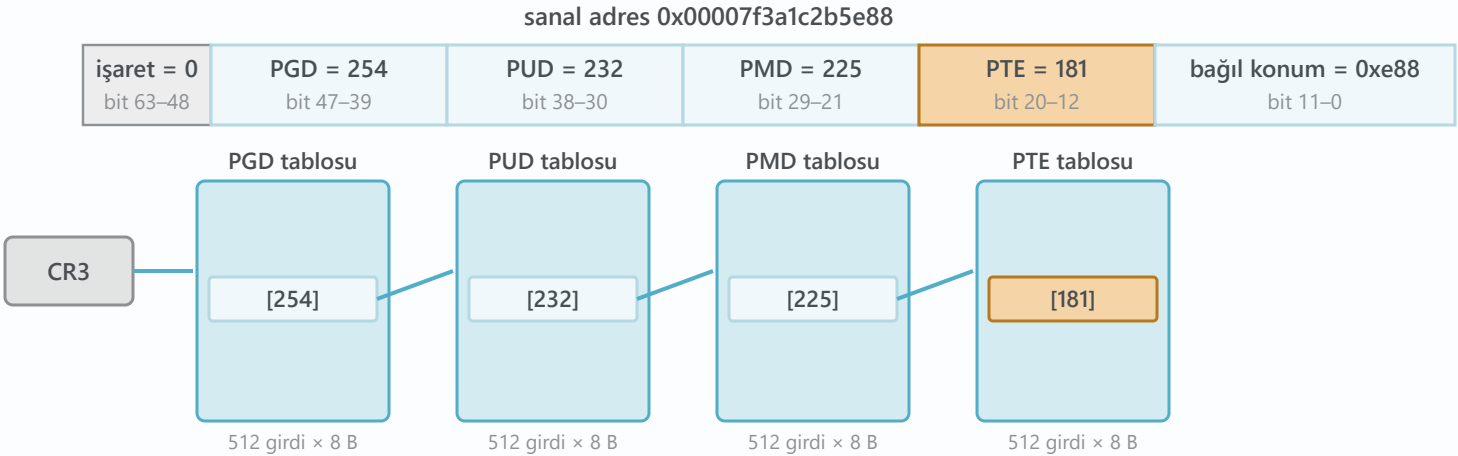
PUD[232] bir PMD tablosunu gösterir. (Burada 1 GiB'lık dev sayfa da bitebilir.)

Sayfa Tablosu Yürüyüşü: 4 ve 5 Düzey



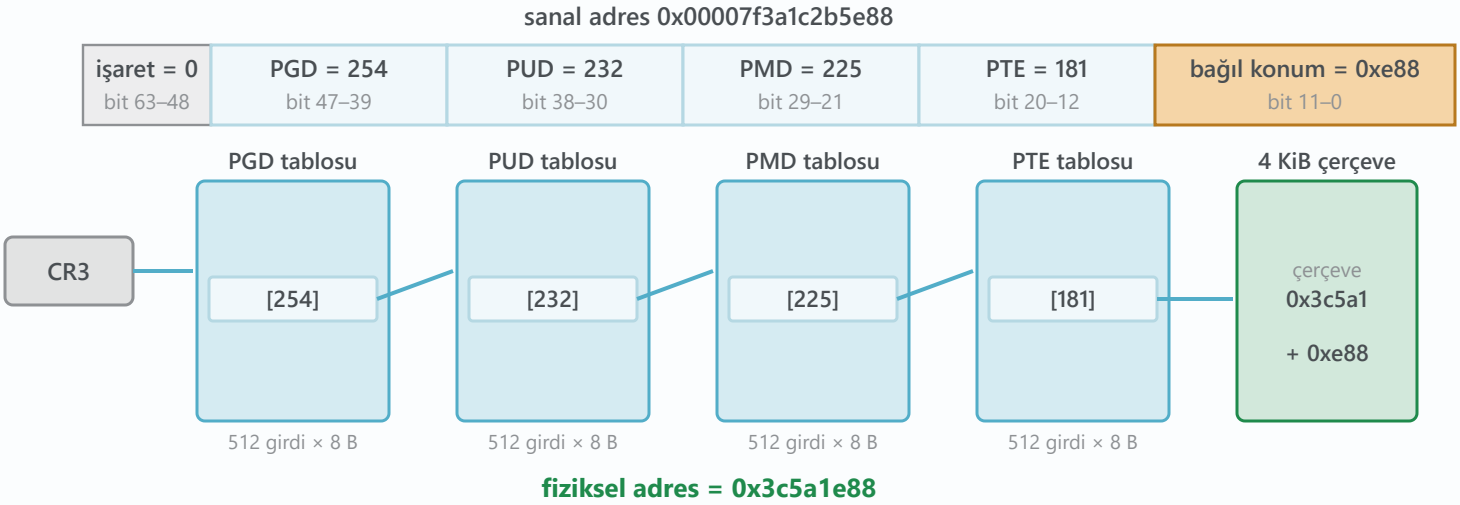
PMD[225] bir PTE tablosunu gösterir. (2 MiB'lık dev sayfa / THP burada biter.)

Sayfa Tablosu Yürüyüşü: 4 ve 5 Düzey



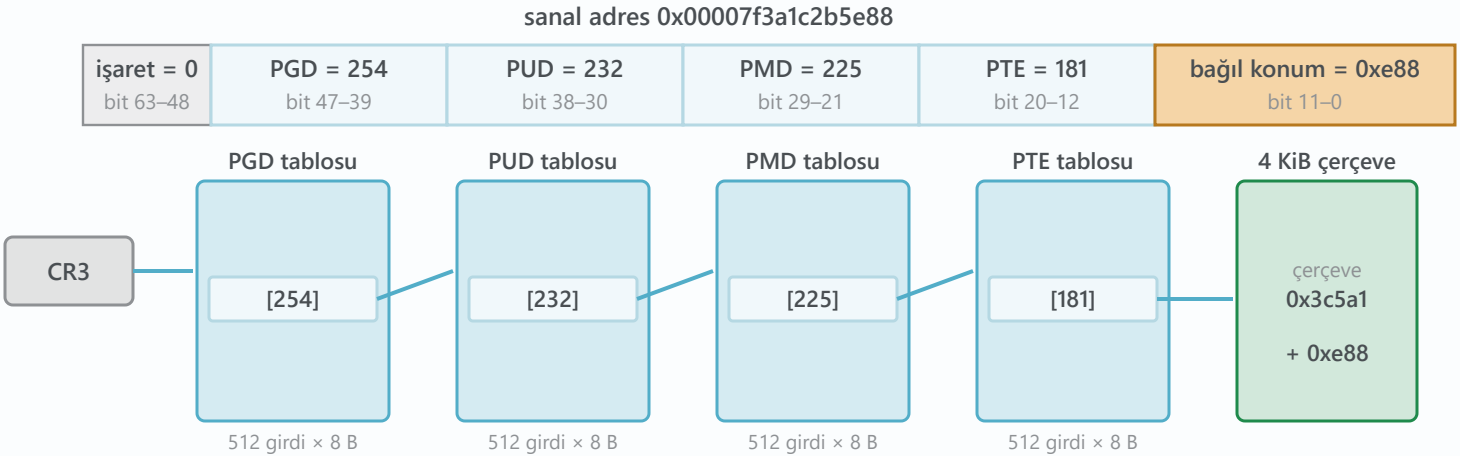
PTE[181]: geçerli (present) biti, çerçeve numarası, R/W, U/S, NX, kirli (D) ve erişildi (A) bitleri. Present = 0 ise **sayfa hatası**.

Sayfa Tablosu Yürüyüşü: 4 ve 5 Düzey



Çerçeve numarası bağıl konumla birleşir: fiziksel adres = çerçeve × 4096 + bağıl konum.

Sayfa Tablosu Yürüyüşü: 4 ve 5 Düzey

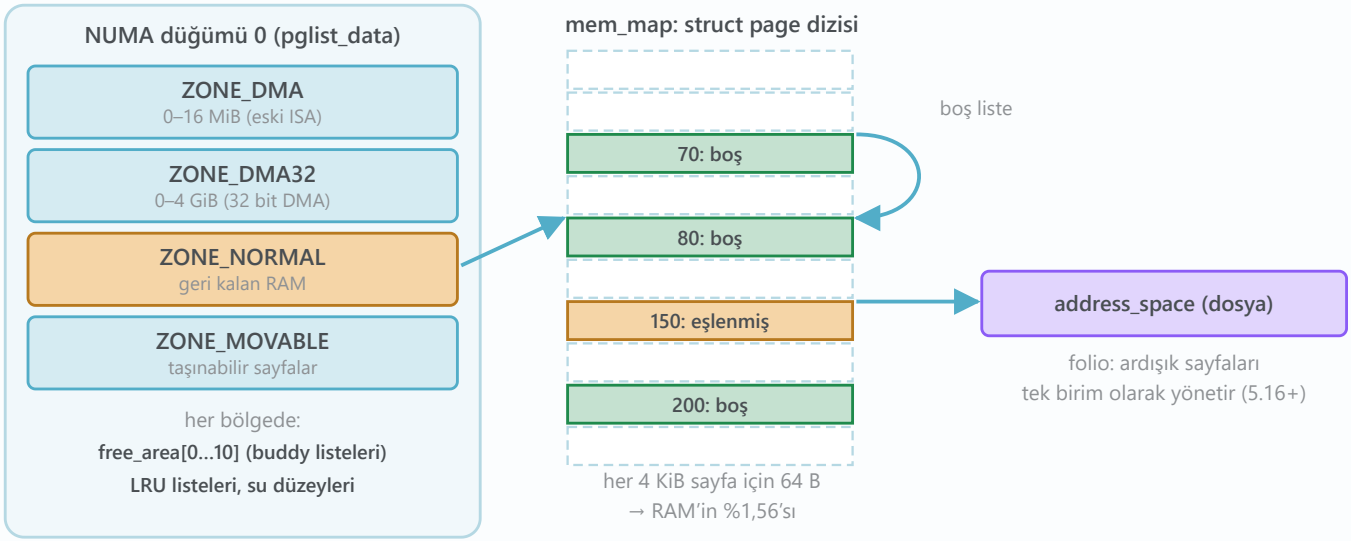


fiziksel adres = 0x3c5a1e88

TLB: sonraki erişimde 4 bellek okuması atlanır

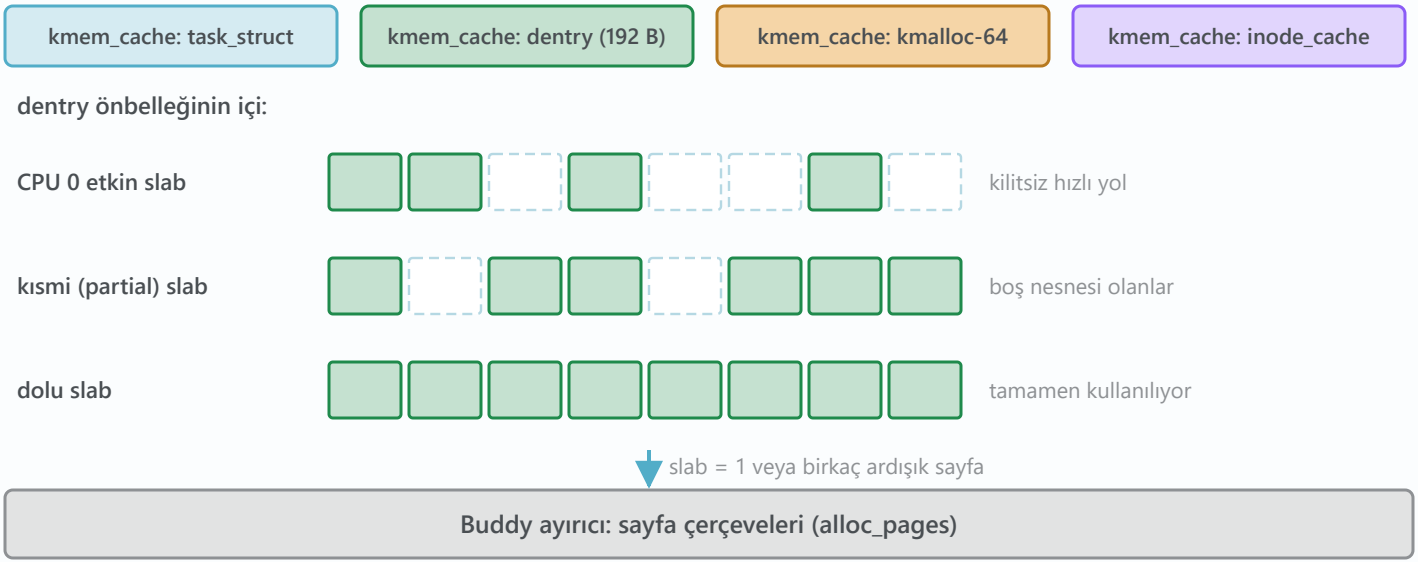
Her erişimde 4 ek okuma pahalıdır: **TLB** çeviriyi önbelleğe alır. 5 düzeyli sayfalama (LA57, Linux 4.14+) araya **P4D** ekleyerek 57 bit adres sağlar. Linux'un genel modeli 5 düzeydir; donanım daha azını kullanıyorsa düzeyler katlanır.

Fiziksel Belleğin Gösterimi



Linux fiziksel belleği **düğüm** → **bölge** → **sayfa** hiyerarşisiyle tanımlar. Çok soketli makinede her CPU kendi düğümünden ayırmayı tercih eder (NUMA). 64 bit sistemlerde ZONE_HIGHMEM yoktur; yalnızca 32 bit çekirdeklerde kalmıştır. Her bölgenin `free_area` dizisi buddy ayırıcının order 0–10 (4 KiB–4 MiB) boş listelerini tutar; durum `/proc/buddyinfo`'da.

SLUB Nesne Ayırıcısı



Çekirdek çok sayıda küçük, aynı boyda nesne (task_struct, dentry, inode) ayırır; her biri için sayfa ayırmak israftır. **SLUB** nesne önbellekleri tutar; `kmalloc` 8 B–8 KiB boy sınıflarını kullanır. SLOB (6.4) ve SLAB (6.8) kaldırıldı: tek ayırıcı SLUB. Büyük, fiziksel bitişik olması gerekmeyen alan için `vmalloc`.

SLUB Serbest Listesi: Kod İzleme

```
1  void *slab_ayir(kmem_cache *c) {
2      void *nesne = c->freelist;    /* listenin başı */
3      if (!nesne)
4          nesne = yeni_slab(c);     /* yavaş yol */
5      c->freelist = *(void **)nesne;  /* sonraki */
6      return nesne;
7  }
8  void slab_birak(kmem_cache *c, void *nesne) {
9      *(void **)nesne = c->freelist;  /* nesnenin içine */
10     c->freelist = nesne;
11 }
▶  /* a,b = ayir; birak(a); c,d,e,f = ayir */
```



freelist → +0 → +1024 → +2048 → +3072 → NULL

Yeni bir slab: 4 KiB'lık sayfa 1 KiB'lık dört nesneye bölünmüş. Boş nesneler bir listede; "sonraki" işaretçisi ayrı bir yapıda değil, boş nesnenin kendi içinde saklanır.

Adım 1 / 7

DEĞİŞKENLER

`freelist = +0`

ÇIKTI

SLUB Serbest Listesi: Kod İzleme

```
1  void *slab_ayir(kmem_cache *c) {
▶      void *nesne = c->freelist;    /* listenin başı */
3      if (!nesne)
4          nesne = yeni_slab(c);     /* yavaş yol */
▶      c->freelist = *(void **)nesne;  /* sonraki */
▶      return nesne;
7  }
8  void slab_birak(kmem_cache *c, void *nesne) {
9      *(void **)nesne = c->freelist;  /* nesnenin içine */
10     c->freelist = nesne;
11 }
12 /* a,b = ayir; birak(a); c,d,e,f = ayir */
```



freelist → +1024 → +2048 → +3072 → NULL

slab_ayir: listenin başı (+0) alınır, freelist onun içindeki sonraki işaretçiye (+1024) ilerler. Kilit yok, sayfa ayırma yok: birkaç komut.

Adım 2 / 7

DEĞİŞKENLER

`nesne = +0` `freelist = +1024`

ÇIKTI

```
a = +0
```

SLUB Serbest Listesi: Kod İzleme

```
1  void *slab_ayir(kmem_cache *c) {
▶      void *nesne = c->freelist;    /* listenin başı */
3      if (!nesne)
4          nesne = yeni_slab(c);     /* yavaş yol */
▶      c->freelist = *(void **)nesne;  /* sonraki */
▶      return nesne;
7  }
8  void slab_birak(kmem_cache *c, void *nesne) {
9      *(void **)nesne = c->freelist;  /* nesnenin içine */
10     c->freelist = nesne;
11 }
12 /* a,b = ayir; birak(a); c,d,e,f = ayir */
```



freelist → +2048 → +3072 → NULL

b için +1024 verilir; freelist +2048'i gösterir.

Adım 3 / 7

DEĞİŞKENLER

`nesne = +1024` `freelist = +2048`

ÇIKTI

```
a = +0    b = +1024
```

SLUB Serbest Listesi: Kod İzleme

```
1  void *slab_ayir(kmem_cache *c) {
2      void *nesne = c->freelist;    /* listenin başı */
3      if (!nesne)
4          nesne = yeni_slab(c);     /* yavaş yol */
5      c->freelist = *(void **)nesne;  /* sonraki */
6      return nesne;
7  }
▶ void slab_birak(kmem_cache *c, void *nesne) {
▶      *(void **)nesne = c->freelist;  /* nesnenin içine */
▶      c->freelist = nesne;
11 }
12 /* a,b = ayir; birak(a); c,d,e,f = ayir */
```



freelist → +0 → +2048 → +3072 → NULL

a serbest bırakılır: nesnenin ilk 8 baytına eski baş (+2048) yazılır ve freelist +0 olur. Liste bir yığın gibi çalışır (son giren ilk çıkar).

Adım 4 / 7

DEĞİŞKENLER

`nesne = +0` `freelist = +0`

ÇIKTI

```
a = +0    b = +1024
```

SLUB Serbest Listesi: Kod İzleme

```
1  void *slab_ayir(kmem_cache *c) {
▶      void *nesne = c->freelist;    /* listenin başı */
3      if (!nesne)
4          nesne = yeni_slab(c);     /* yavaş yol */
▶      c->freelist = *(void **)nesne;  /* sonraki */
▶      return nesne;
7  }
8  void slab_birak(kmem_cache *c, void *nesne) {
9      *(void **)nesne = c->freelist;  /* nesnenin içine */
10     c->freelist = nesne;
11 }
12 /* a,b = ayir; birak(a); c,d,e,f = ayir */
```



freelist → +2048 → +3072 → NULL

c, a'nın az önce bıraktığı +0'ı alır: büyük olasılıkla hâlâ CPU önbelleğinde olduğu için "sıcak" bir nesne.

Adım 5 / 7

DEĞİŞKENLER

`nesne = +0` `freelist = +2048`

ÇIKTI

```
a = +0    b = +1024    c = +0
```

SLUB Serbest Listesi: Kod İzleme

```
1  void *slab_ayir(kmem_cache *c) {
▶      void *nesne = c->freelist;    /* listenin başı */
3      if (!nesne)
4          nesne = yeni_slab(c);     /* yavaş yol */
▶      c->freelist = *(void **)nesne;  /* sonraki */
▶      return nesne;
7  }
8  void slab_birak(kmem_cache *c, void *nesne) {
9      *(void **)nesne = c->freelist;  /* nesnenin içine */
10     c->freelist = nesne;
11 }
12 /* a,b = ayir; birak(a); c,d,e,f = ayir */
```



freelist → NULL

d +2048'i, e +3072'yi alır. +3072'nin içindeki sonraki işaretçi NULL'dı: slab doldu, freelist boş.

Adım 6 / 7

DEĞİŞKENLER

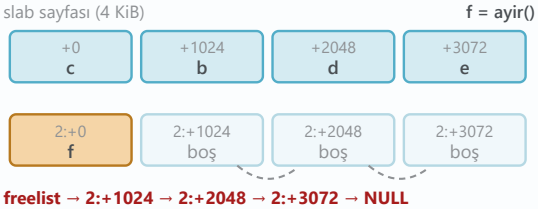
`nesne = +3072` `freelist = NULL`

ÇIKTI

```
a = +0    b = +1024    c = +0
d = +2048    e = +3072
```

SLUB Serbest Listesi: Kod İzleme

```
1  void *slab_ayir(kmem_cache *c) {
▶      void *nesne = c->freelist;    /* listenin başı */
▶      if (!nesne)
▶          nesne = yeni_slab(c);     /* yavaş yol */
▶      c->freelist = *(void **)nesne;  /* sonraki */
▶      return nesne;
7  }
8  void slab_birak(kmem_cache *c, void *nesne) {
9      *(void **)nesne = c->freelist;  /* nesnenin içine */
10     c->freelist = nesne;
11 }
12 /* a,b = ayir; birak(a); c,d,e,f = ayir */
```



f için liste boş: **yavaş yol**. Sayfa ayırıcıdan (buddy) yeni bir sayfa alınır, dört nesneye bölünür, ilki f'ye verilir. Yaygın durum hızlı, nadir durum doğru.

Adım 7 / 7

DEĞİŞKENLER

`nesne = 2:+0` `freelist = 2:+1024`

ÇIKTI

```
a = +0    b = +1024    c = +0
d = +2048    e = +3072    f = 2:+0
(yeni slab)
```

Sayfa Önbelleği, Geri Kazanım ve OOM



Boş bellek israftır: Linux kullanılmayan RAM'i **sayfa önbelleği** olarak dosya verisi için kullanır. `free -h`'deki "available" bu önbelleğin geri alınabilir kısmını da sayar.

Sayfa Önbelleği, Geri Kazanım ve OOM



Uygulama büyüdü, boş bellek **düşük** su düzeyinin altına indi: `kswapd` uyanır ve arka planda temiz önbellek sayfalarını bırakır (kirli olanlar önce diske yazılır).

Sayfa Önbelleği, Geri Kazanım ve OOM



MGLRU (6.1) sayfaları erişim nesillerine ayırır; en soğuk nesil önce gider. Soğuk **anonim** sayfaların dosyası yoktur: **zswap** ile RAM'de sıkıştırılır, gerekirse takas alanına yazılır.

Sayfa Önbelleği, Geri Kazanım ve OOM



Ayırma hızı geri kazanımı geçti, boş bellek **min** düzeyinin altında: ayıran süreç kendisi **doğrudan geri kazanım** yapar ve takılır (gecikme artar).

Sayfa Önbelleği, Geri Kazanım ve OOM

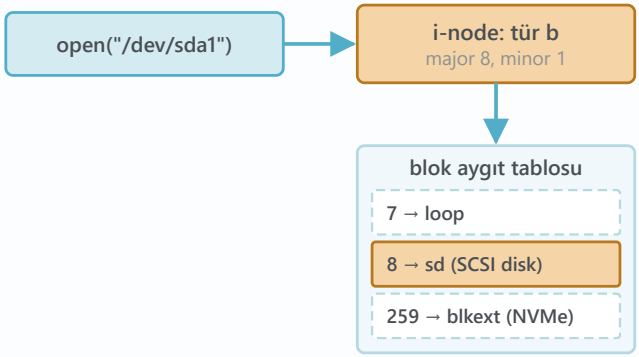


Hiçbir şey yetmezse **OOM killer** en yüksek `oom_score`'lu (çoğunlukla en büyük) süreci SIGKILL ile öldürür; `oom_score_adj` ile etkilenebilir. cgroup sınırı aşılırsa yalnız o gruptan seçilir.

Aygıtlar Dosyadır: Major ve Minor Numaralar

```
$ ls -l /dev     # seçilmiş satırlar
brw-rw---- 1 root disk   8, 0 sda
brw-rw---- 1 root disk   8, 1 sda1
brw-rw---- 1 root disk 259, 0 nvme0n1
crw-rw-rw- 1 root root   1, 3 null
crw-rw-rw- 1 root tty    5, 0 tty
```

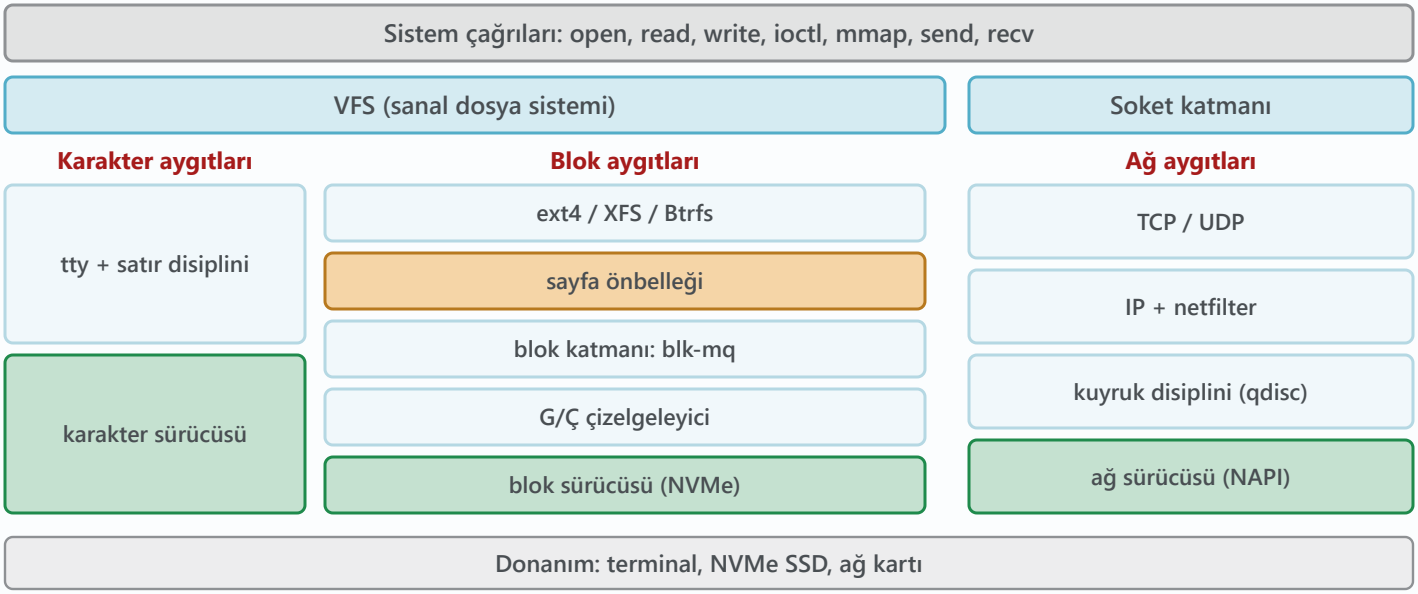
- **c** karakter aygıtı: bayt akışı, genelde önbelleksiz (terminal, `/dev/null`, fare)
- **b** blok aygıtı: sabit boy bloklar, rastgele erişim; sayfa önbelleği ve G/Ç çizelgeleyici kullanır
- Ağ arayüzleri (`eth0`) `/dev` 'de yoktur; soketlerle kullanılır
- Düğümleri devtmpfs + **udev** aygıt görününce dinamik oluşturur



minor: 0 = tüm disk, 1 = bölüm 1
disk başına 16 minor: sdb = 16

Boyut sütunu yerine "major, minor" görünür. **Major** sürücüyü, **minor** o sürücünün yönettiği birimi seçer. Uygulama diske de terminale de aynı `open/read/write` ile erişir.

Linux G/Ç Sisteminin Yapısı



Üç aygıt türü üç ayrı yoldan iner. Klasik BSD'de de benzer ayrım vardı; System V'in **STREAMS** modül yığını ise Linux ana çekirdeğine hiç girmedi. Ağ yolunda veri `sk_buff` yapılarında taşınır; XDP/eBPF paketleri sürücü düzeyinde işleyebilir.

Sürücü Arayüzü: file_operations ve Terminal

```
static const struct file_operations my_fops = {
    .owner          = THIS_MODULE,
    .open           = my_open,
    .release        = my_release,     /* close */
    .read           = my_read,
    .write          = my_write,
    .unlocked_ioctl = my_ioctl,
    .poll           = my_poll,
    .mmap           = my_mmap,
};
cdev_init(&cdev, &my_fops);  /* major:minor → fops */
```

Eski UNIX girişleri	Linux karşılığı
`k_open, tty_open, pr_open, ip_open`	`.open`
`k_close, tty_close, pr_close…`	`.release`
`mem_read, k_read, tty_read`	`.read`
`mem_write, tty_write, pr_write…`	`.write`
`k_ioctl, tty_ioctl, pr_ioctl…`	`.unlocked_ioctl`

Terminal G/Ç (termios): `tcgetattr/tcsetattr` terminal parametrelerini okur/yazar: kanonik mod (satır düzenleme, ICANON) ya da ham mod, ECHO. `cfset/cfget[io]speed` giriş/çıkış baud hızı; bugün çoğunlukla seri portlar için, terminaller ise sözde terminaldir (pty).

Klasik ders kitabı tablosu her sürücü için ayrı giriş noktaları (k_*, tty_*, pr_*, ip_*) listeler; Linux bunları tek bir fonksiyon işaretçisi yapısında toplar. VFS her `read()`'i ilgili sürücünün `.read`'ine yönlendirir.

GİRİŞ/ÇIKIŞ

Blok Katmanı: blk-mq ve G/Ç Çizelgeleyiciler

Linux blok katmanı **blk-mq** çok kuyrukludur: her CPU kendi yazılım kuyruğuna yazar, istekler isteğe bağlı bir G/Ç çizelgeleyiciden (`none`, `mq-deadline`, `bfq`, `kyber`) geçip aygıtın donanım kuyruklarına dağıtılır. Etkin çizelgeleyici aygıt başına `/sys/block/<aygıt>/queue/scheduler` dosyasından okunur ve değiştirilir.

Ağ: Soketler ve Bağlantı Kurulumu



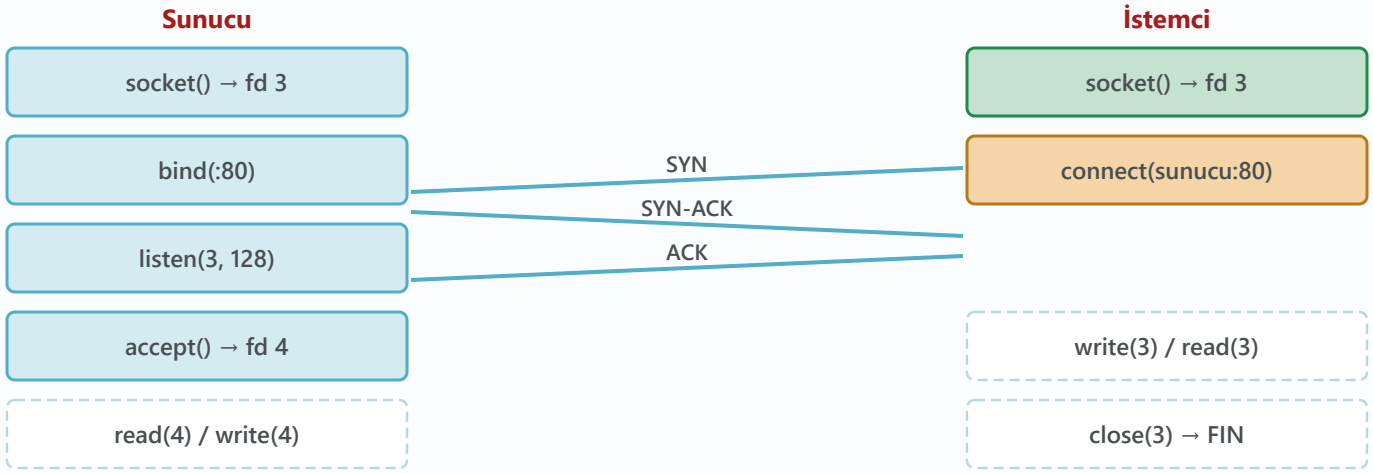
Sunucu bir soket oluşturur (fd 3), 80. porta bağlar (`bind`) ve `listen` ile dinleme kuyruğu açar. Soket de bir dosya tanımlayıcısıdır.

Ağ: Soketler ve Bağlantı Kurulumu



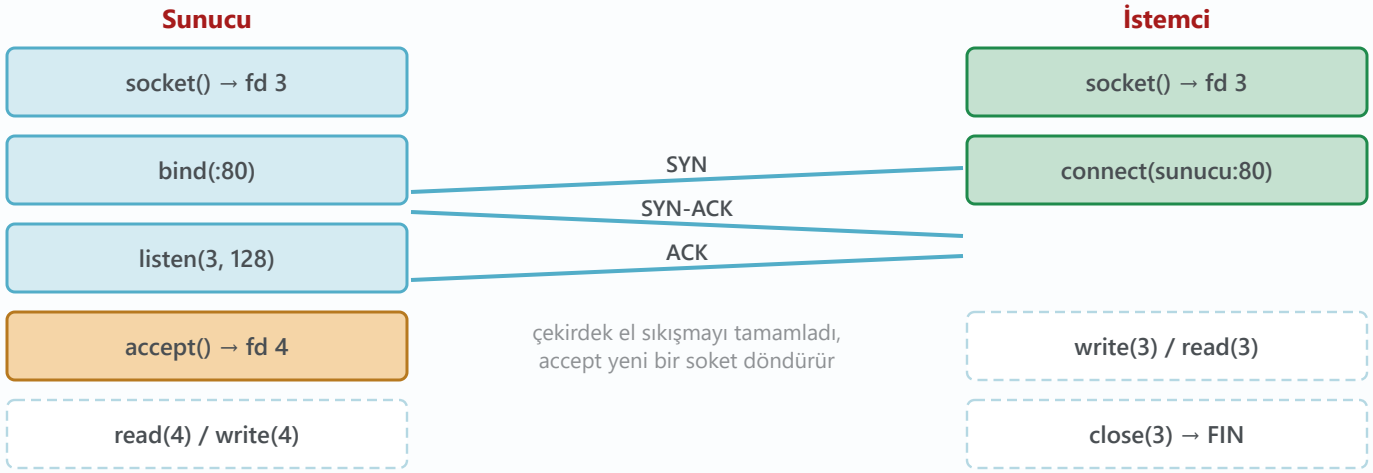
Sunucu `accept`'te bloke olur. İstemci kendi soketini oluşturur.

Ağ: Soketler ve Bağlantı Kurulumu



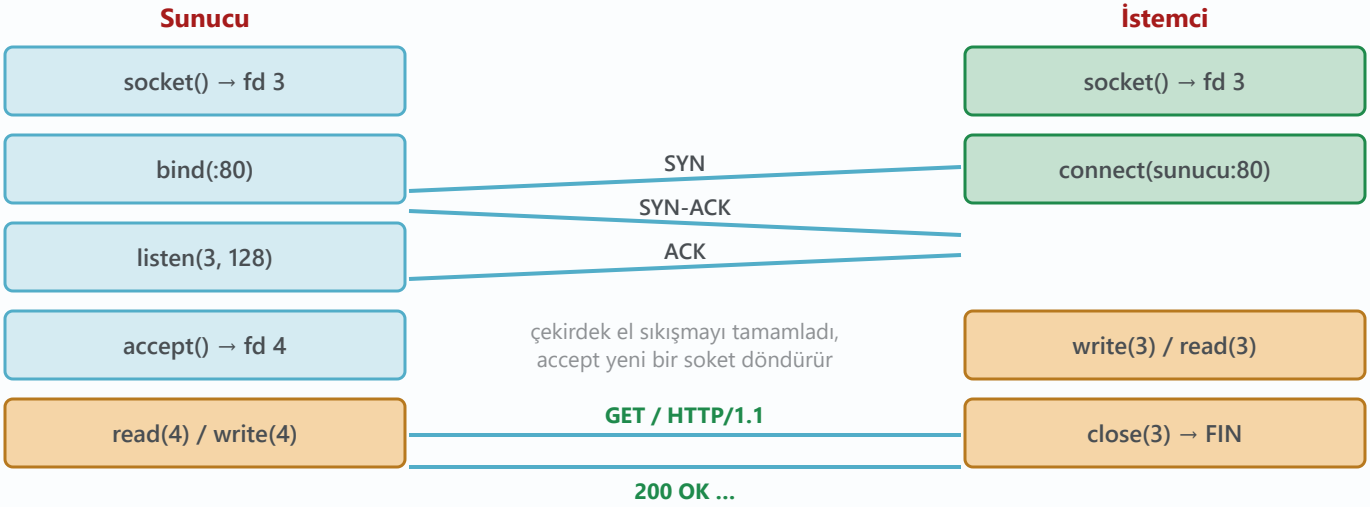
`connect` çağrısıyla istemci çekirdeği **üçlü el sıkışmayı** (SYN, SYN-ACK, ACK) yapar; uygulama bu paketleri hiç görmez.

Ağ: Soketler ve Bağlantı Kurulumu



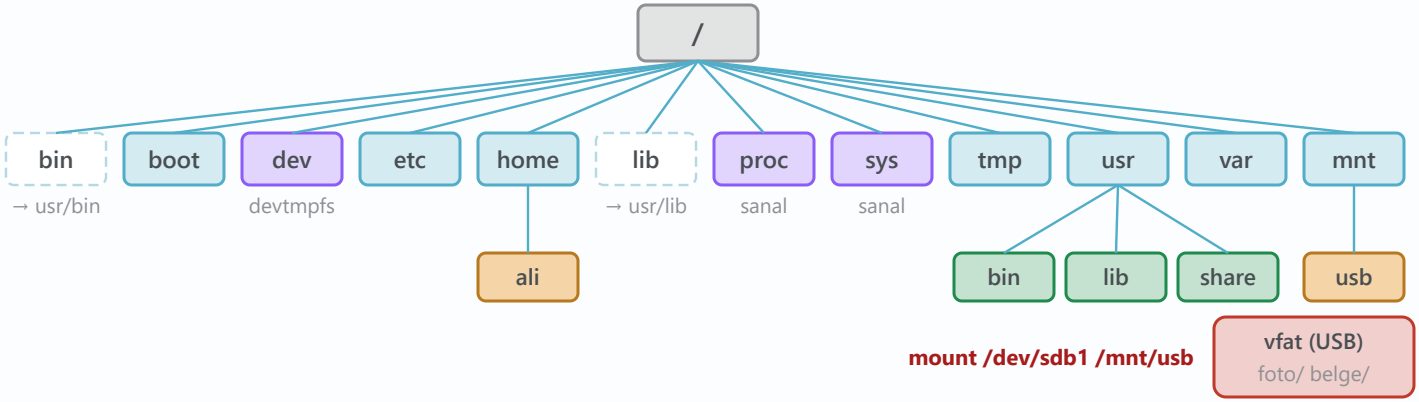
Bağlantı kurulunca `accept` **yeni** bir soket (fd 4) döndürür; fd 3 dinlemeye devam eder. Çok istemcili sunucular `epoll` veya iş parçacıklarıyla binlerce soketi yönetir.

Ağ: Soketler ve Bağlantı Kurulumu



Veri `write` / `read` ile akar: çekirdek `sk_buff` 'lara koyar, TCP/IP başlıklarını ekler, sürücü kartın halkasına yazar. `close` FIN gönderir.

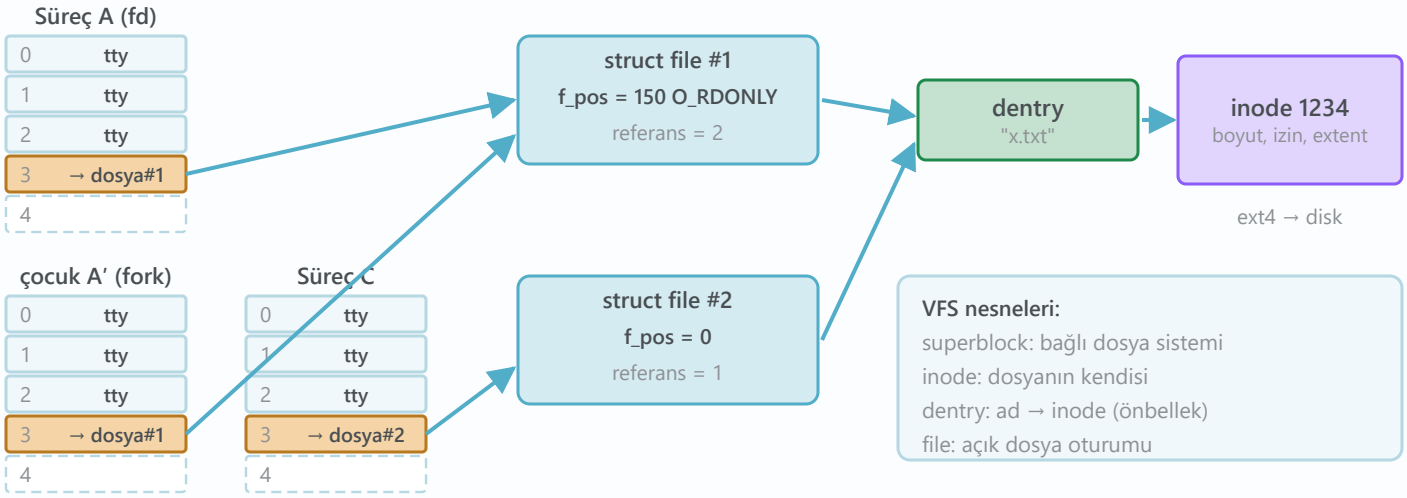
Dizin Ağacı ve Bağlama



- `/usr/bin` : komutlar (`/bin` artık ona bağlantı)
- `/etc` : metin yapılandırma dosyaları
- `/dev` : aygıt dosyaları
- `/usr/lib` : paylaşımlı kütüphaneler, `/usr/lib/modules` : çekirdek modülleri
- `/usr` : salt okunur, paylaşılabilir yazılım (bin, lib, share)
- `/home` : kullanıcı dizinleri; `/root` : yöneticinin
- `/var` : değişken veri (günlükler, kuyruklar, önbellek)
- `/proc` , `/sys` : çekirdeğin sözde dosya sistemleri
- `/boot` : çekirdek ve initramfs
- `/tmp` : geçici (çoğu dağıtımda tmpfs)

Windows'taki sürücü harfleri yerine tek bir ağaç vardır: başka bir dosya sistemi herhangi bir dizine **bağlanır** (mount). Düzen FHS standardına uyar; çoğu dağıtım "usrmerge" ile /bin, /lib'i /usr altına taşıdı.

VFS: Tanımlayıcılar, Açık Dosyalar ve i-node'lar



Linux'ta açık dosya `struct file`'dır (`f_pos`, bayraklar, referans sayısı); süreç tablosu `task_struct → files_struct` içindedir ve `fork()` sonrası iki süreç aynı `struct file`'ı gösterir. Nesnelerin işlem tabloları (`super_operations`, `inode_operations`, `file_operations`) çağrıyı ext4, XFS, Btrfs, NFS ya da procfs'in kendi işlevine yönlendirir; dentry'ler dcache'te tutulur.

Dosya ve Dizin Sistem Çağrıları, stat

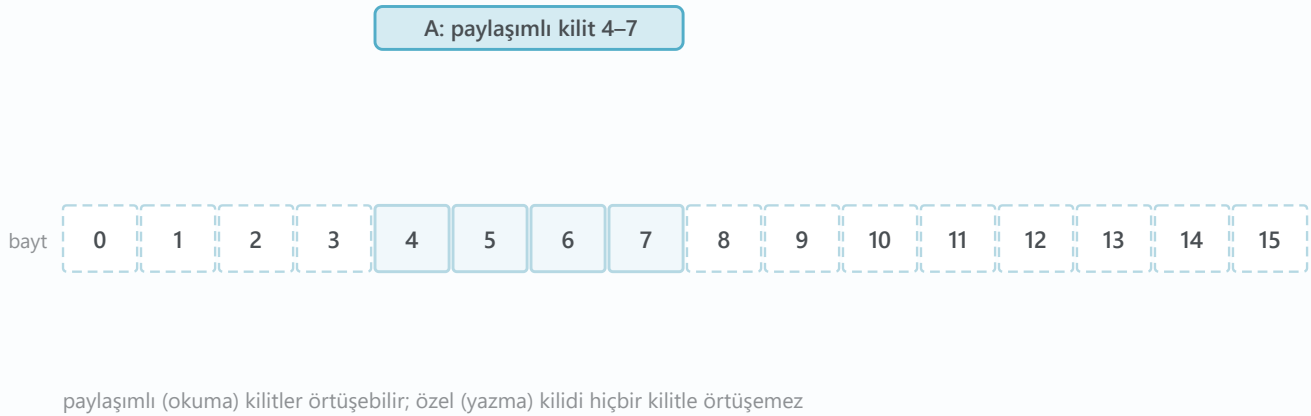
Dosya çağrısı	İşlev
`open` , `creat`	Aç / oluştur, fd döndür
`close`	Tanımlayıcıyı kapat
`read` , `write`	Konumdan oku / yaz
`lseek`	Konumu taşı
`stat` , `fstat` , `lstat`	Bilgi al (lstat: bağın kendisi)
`pipe` , `dup2`	Boru oluştur / fd kopyala
`fcntl`	Kilit, bayraklar vb.
`mkdir` , `rmdir`	Dizin oluştur / sil (boşsa)
`link` , `unlink`	Ad ekle / sil
`rename` , `chdir`	Taşı (atomik) / cwd değiştir
`getdents64`	Dizin girdilerini oku

stat alanı	Anlamı
`st_dev`	Dosyanın bulunduğu aygıt
`st_ino`	i-node numarası
`st_mode`	Tür + izin bitleri
`st_nlink`	Katı bağ sayısı
`st_uid` , `st_gid`	Sahip ve grup
`st_size`	Bayt cinsinden boyut
`st_atime`	Son erişim zamanı
`st_mtime`	Son içerik değişikliği
`st_ctime`	Son i-node değişikliği (oluşturma değil!)
`st_blocks`	Ayrılan 512 B blok sayısı



`opendir/readdir/closedir` sistem çağrısı değil, `getdents64` üzerine kurulu kütüphane fonksiyonlarıdır. Oluşturma zamanı yalnız yeni `statx()` ile (stx_btime) alınabilir.

Dosya Kilitleme

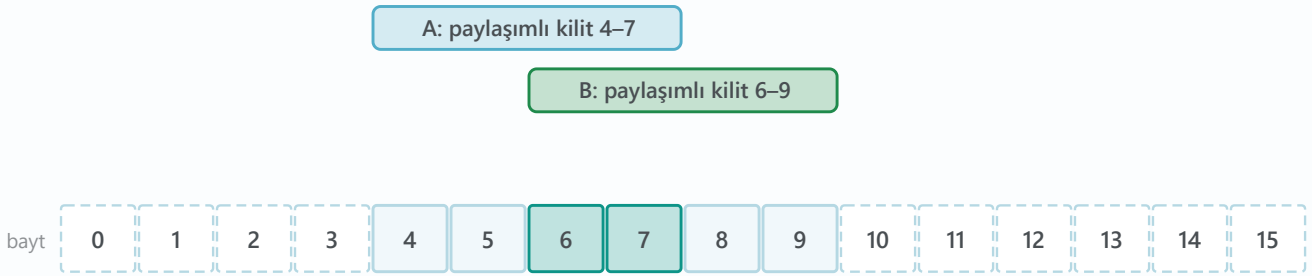


A süreci 4–7. baytlara **paylaşımlı** (okuma) kilit koyar: `fcntl(fd, F_SETLK, &fl)` ile `l_type = F_RDLCK`, `l_start = 4`, `l_len = 4`.

Adım 1 / 4

Bu kilitler **tavsiye niteliğindedir** (advisory): kilidi sormayan bir süreç yine yazabilir. Bütün dosya için `flock()`; iş parçacıklarıyla güvenli kullanım için açık dosyaya bağlı OFD kilitleri vardır.

Dosya Kilitleme

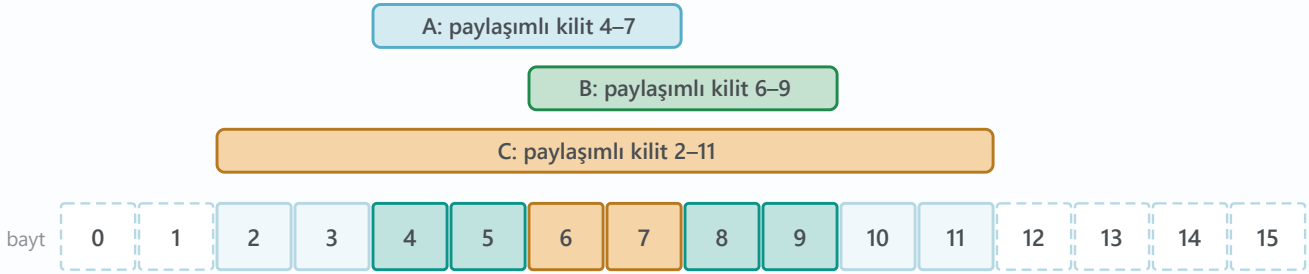


paylaşımlı (okuma) kilitler örtüşebilir; özel (yazma) kilidi hiçbir kilitle örtüşemez

B 6–9'a paylaşımlı kilit ister: örtüşme paylaşımlı kilitler arasında serbesttir, istek kabul edilir.

Bu kilitler **tavsiye niteliğindedir** (advisory): kilidi sormayan bir süreç yine yazabilir. Bütün dosya için `flock()`; iş parçacıklarıyla güvenli kullanım için açık dosyaya bağlı OFD kilitleri vardır.

Dosya Kilitleme

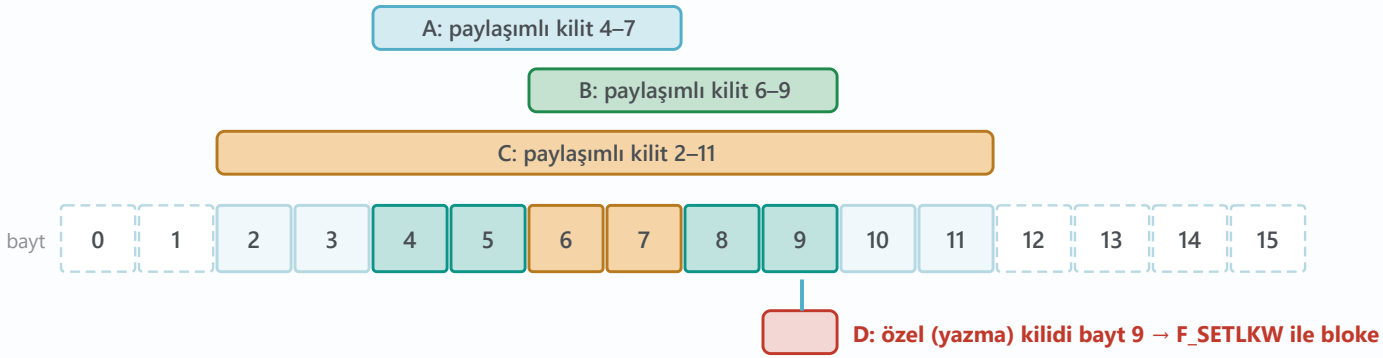


paylaşımlı (okuma) kilitler örtüşebilir; özel (yazma) kilidi hiçbir kilitle örtüşemez

C 2–11'e paylaşımlı kilit koyar; 6–7 baytları artık üç kilitle kaplı.

Bu kilitler **tavsiye niteliğindedir** (advisory): kilidi sormayan bir süreç yine yazabilir. Bütün dosya için `flock()`; iş parçacıklarıyla güvenli kullanım için açık dosyaya bağlı OFD kilitleri vardır.

Dosya Kilitleme

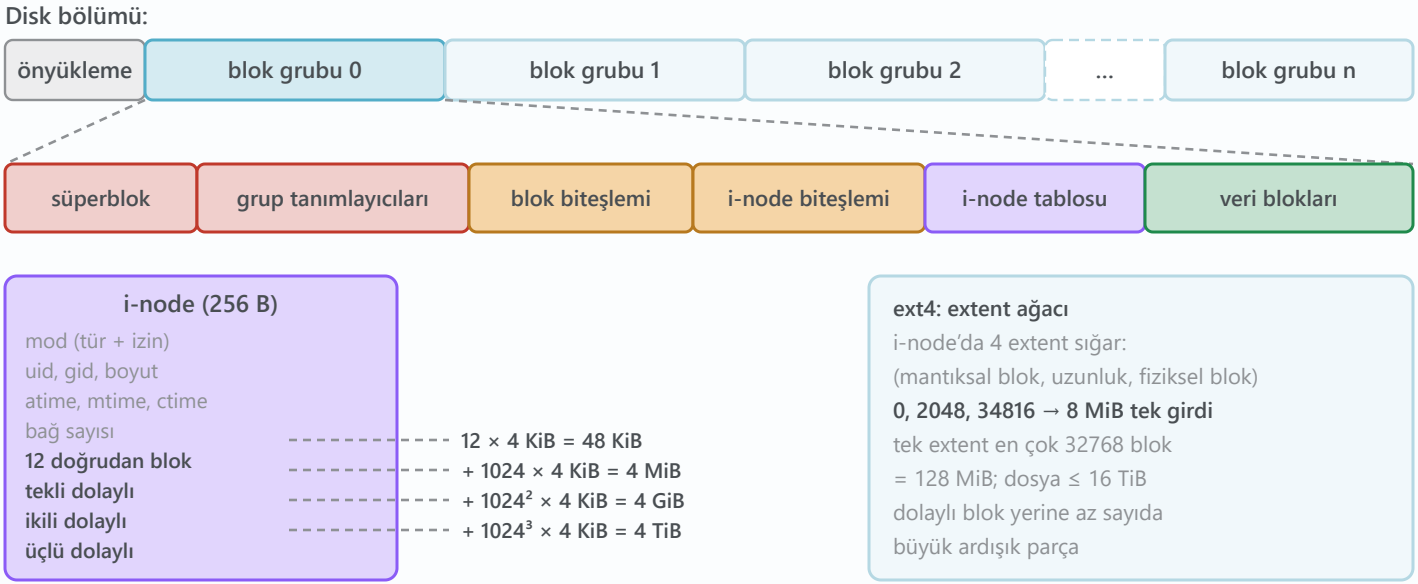


paylaşımlı (okuma) kilitler örtüşebilir; özel (yazma) kilidi hiçbir kilitle örtüşemez

D bayt 9'a **özel** (F_WRLCK) kilit ister: B ve C'nin kilitleriyle çakışır. `F_SETLK` hemen hata döndürür, `F_SETLKW` ise kilitler kalkana dek bekler. Çekirdek bekleme döngüsünü fark ederse `EDEADLK` döndürür.

Bu kilitler **tavsiye niteliğindedir** (advisory): kilidi sormayan bir süreç yine yazabilir. Bütün dosya için `flock()`; iş parçacıklarıyla güvenli kullanım için açık dosyaya bağlı OFD kilitleri vardır.

ext2/ext4 Disk Düzeni ve i-node



Klasik UNIX düzeni (önyükleme bloğu, süperblok, i-node'lar, veri) ext2'de **blok gruplarına** bölündü: i-node ile verisi aynı gruba yakın konur, süperblok yedeklenir. Dolaylı bloklarla teorik sınır ~4 TiB'tır (ext2'de i_blocks alanı nedeniyle 2 TiB). Kök dizinin i-node numarası her zaman 2'dir.

ext4 Günlükleme (Journaling)



Bir `write()` + `fsync()` birkaç bloğu değiştirir: veri (D) ile birlikte i-node, blok biteşlemi ve extent gibi **üst veriler**. Bunlardan biri yazılıp diğeri yazılmadan çökme olursa dosya sistemi tutarsız kalır.

Adım 1 / 5

Diğer modlar: **data=journal** (veri de günlüğe; en güvenli, en yavaş) ve **data=writeback** (sıra garantisi yok). XFS de üst veri günlüğü kullanır; Btrfs ise günlük yerine yazınca kopyala ile tutarlıdır.

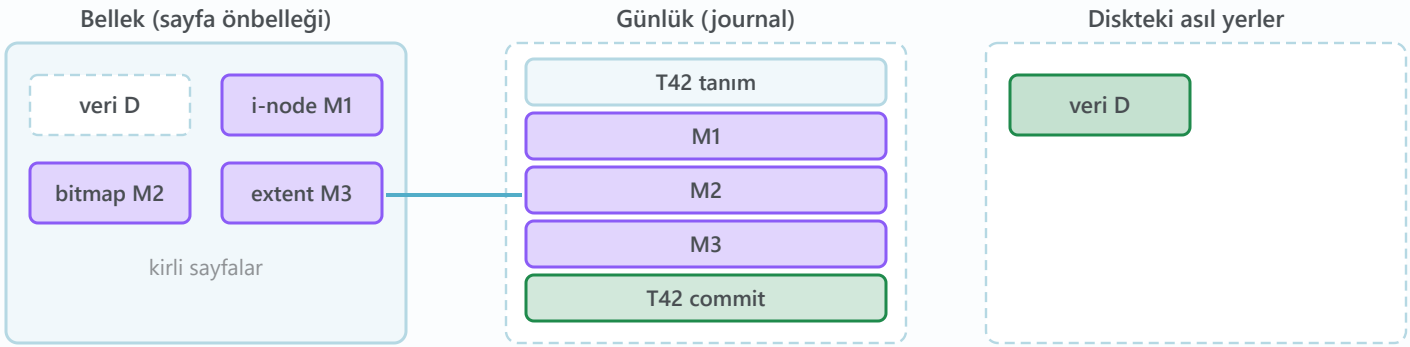
ext4 Günlükleme (Journaling)



Varsayılan **data=ordered** modunda önce veri blokları asıl yerine yazılır; böylece üst veri hiçbir zaman çöp içeren bloğu göstermez.

Diğer modlar: **data=journal** (veri de günlüğe; en güvenli, en yavaş) ve **data=writeback** (sıra garantisi yok). XFS de üst veri günlüğü kullanır; Btrfs ise günlük yerine yazınca kopyala ile tutarlıdır.

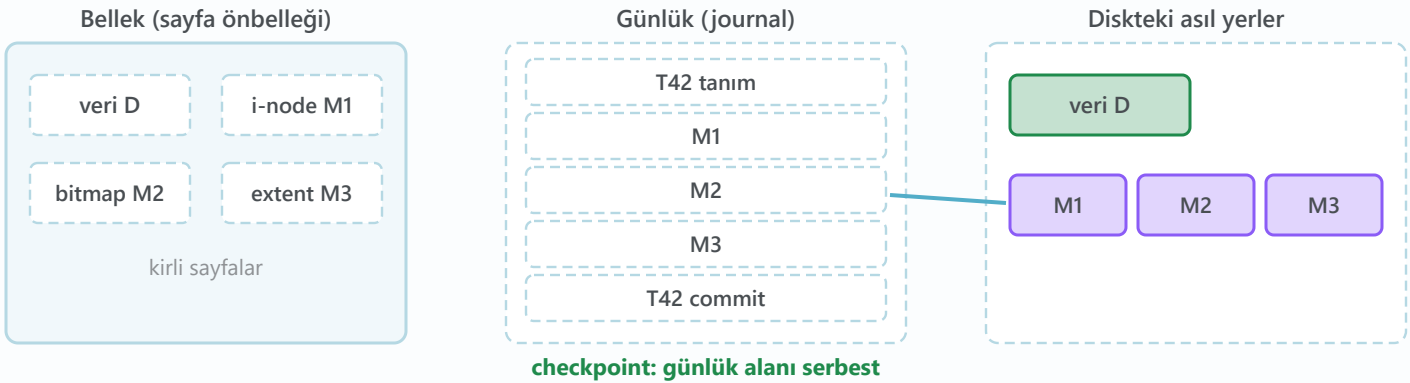
ext4 Günlükleme (Journaling)



Üst veri blokları **günlüğe** bir işlem (T42) olarak yazılır; en sonda **commit** kaydı. Commit diske ulaştığı an işlem kalıcıdır.

Diğer modlar: **data=journal** (veri de günlüğe; en güvenli, en yavaş) ve **data=writeback** (sıra garantisi yok). XFS de üst veri günlüğü kullanır; Btrfs ise günlük yerine yazınca kopyala ile tutarlıdır.

ext4 Günlükleme (Journaling)



Checkpoint: üst veri asıl yerlerine yazılır, günlükteki alan yeniden kullanılır (dairesel tampon).

Diğer modlar: **data=journal** (veri de günlüğe; en güvenli, en yavaş) ve **data=writeback** (sıra garantisi yok). XFS de üst veri günlüğü kullanır; Btrfs ise günlük yerine yazınca kopyala ile tutarlıdır.

ext4 Günlükleme (Journaling)

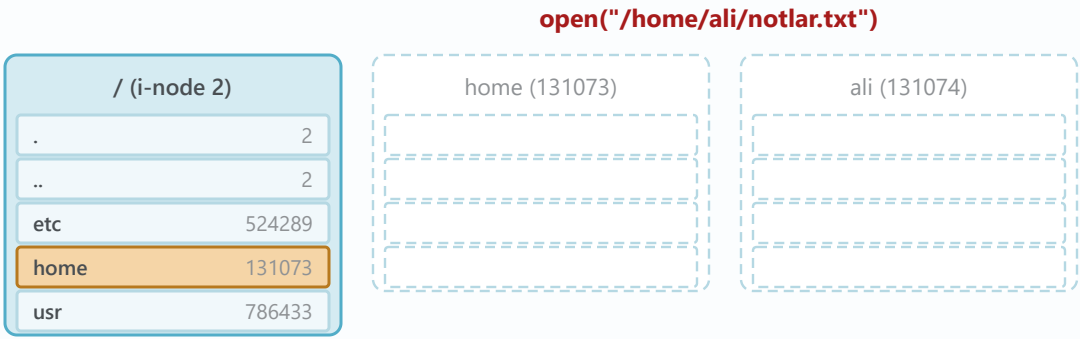


commit'ten sonra çökme → bağlarken T42 yeniden oynatılır
commit'ten önce çökme → T42 yok sayılır; eski tutarlı hâl

Çökmeden sonra tam `fsck` taraması yerine günlük okunur: commit'li işlemler yeniden oynatılır, yarım kalanlar atılır. Saniyeler içinde tutarlılık.

Diğer modlar: **data=journal** (veri de günlüğe; en güvenli, en yavaş) ve **data=writeback** (sıra garantisi yok). XFS de üst veri günlüğü kullanır; Btrfs ise günlük yerine yazınca kopyala ile tutarlıdır.

Dizinler ve Yol Çözümleme



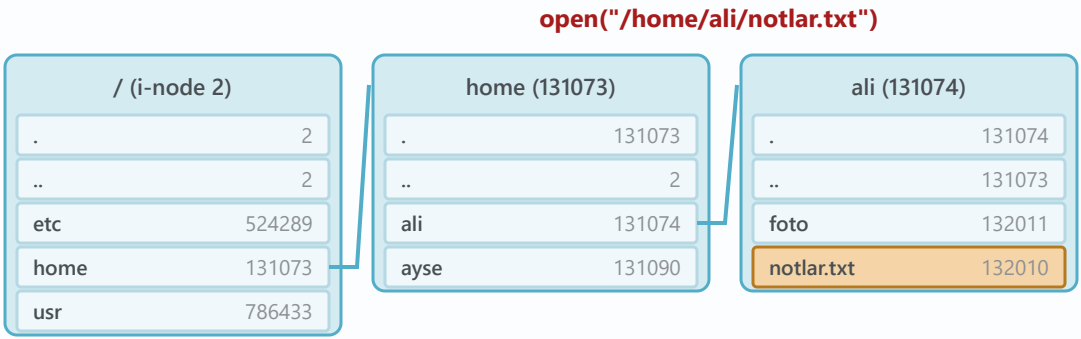
Mutlak yol kökten başlar: kök dizinin i-node'u (2) okunur, veri bloklarındaki girdilerde "home" aranır → 131073. Her dizin için **x** (arama) izni denetlenir.

Dizinler ve Yol Çözümleme



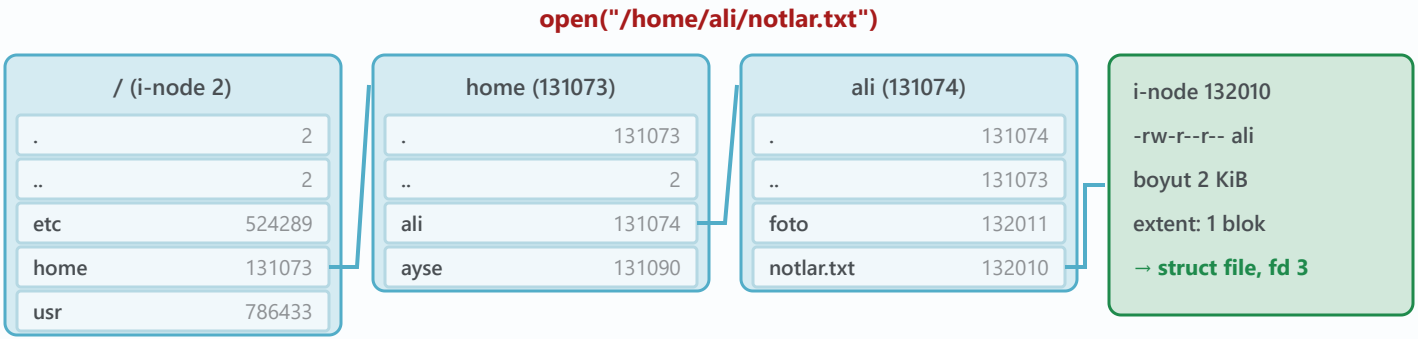
“home” dizininin verisinde “ali” aranır → 131074.

Dizinler ve Yol Çözümleme



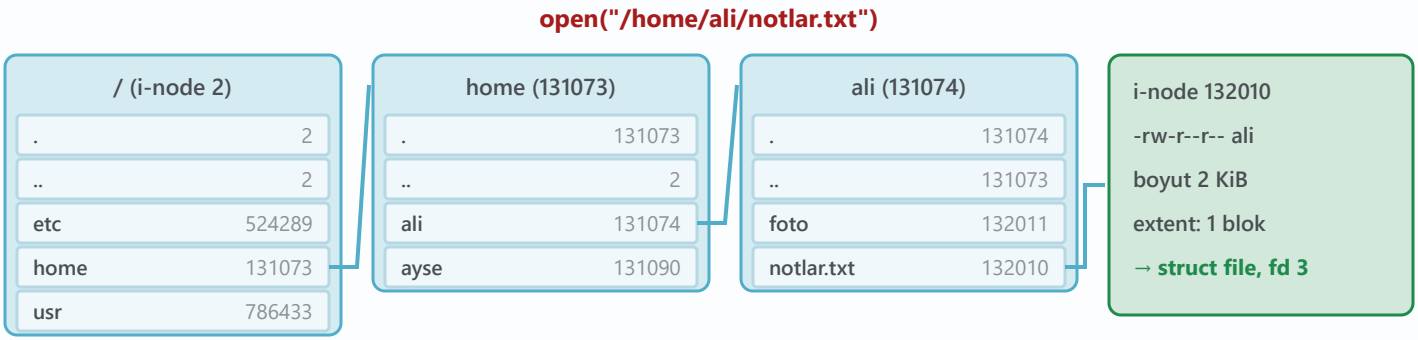
"ali" dizininde "notlar.txt" → 132010. (Göreli yol olsaydı süreç cwd'sinden başlanırdı.)

Dizinler ve Yol Çözümleme



Son bileşenin i-node'u okunur, izinler denetlenir, struct file oluşturulup fd döndürülür.

Dizinler ve Yol Çözümleme

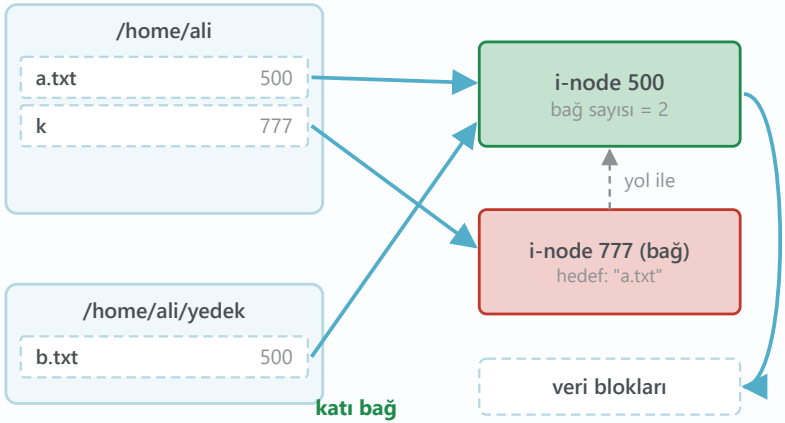


dizin girdisi (ext2/4): i-node no · kayıt uzunluğu · ad uzunluğu · tür · ad

silinen girdi bir öncekinin kayıt uzunluğuna eklenir; büyük dizinlerde htree (hash ağaç) kullanılır

Her adım disk okuması gerektirseydi yavaş olurdu: **dentry önbelleği** (dcache) ad → i-node eşlemelerini bellekte tutar; sıcak yollar diske hiç gitmeden çözülür.

Katı ve Sembolik Bağlar

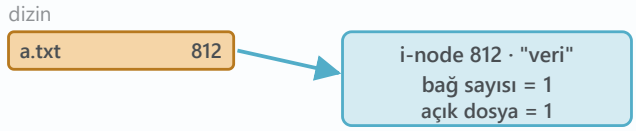



```
$ ln a.txt yedek/b.txt     # katı
$ ln -s a.txt k            # sembolik
$ ls -li
500 -rw-r--r-- 2 ali ali 9 a.txt
777 lrwxrwxrwx 1 ali ali 5 k -> a.txt
```

- Katı bağ: **aynı i-node**, bağ sayısı artar. `rm a.txt` yalnız adı siler; veri sayaç 0 olunca (ve açık fd kalmayınca) serbest kalır.
- Katı bağ dosya sistemi sınırını aşamaz, dizinlere konamaz.
- Sembolik bağ: hedefin **yolunu** tutar; başka dosya sistemine, dizine işaret edebilir. Hedef silinirse bağ **sarkar**.

Bağ Sayısı ve unlink: Kod İzleme

```
▶ int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
▶ write(fd, "veri", 4);
3  link("a.txt", "b.txt");      /* katı bağ */
4  symlink("a.txt", "s.txt");   /* sembolik */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



a.txt oluşturuldu, 4 bayt yazıldı

open yeni bir i-node (812) ayırır, dizine “a.txt → 812” girdisini ekler: bağ sayısı 1. fd 3 açık: açık dosya sayısı 1.

Adım 1 / 9

DEĞİŞKENLER

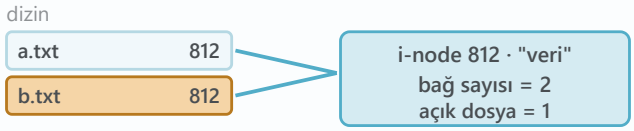
`fd = 3` `bağ = 1` `açık = 1`

ÇIKTI

```
fd = 3
```

Bağ Sayısı ve unlink: Kod İzleme

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "veri", 4);
▶  link("a.txt", "b.txt");      /* katı bağ */
4  symlink("a.txt", "s.txt");   /* sembolik */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



b.txt aynı i-node'a ikinci ad

link yalnızca yeni bir dizin girdisi yazar: b.txt → **812**. Veri kopyalanmaz; bağ sayısı 2. a.txt ile b.txt eşittir, hangisinin "asıl" olduğu yoktur.

Adım 2 / 9

DEĞİŞKENLER

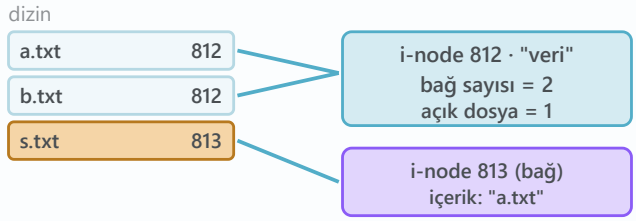
`fd = 3` `bağ = 2` `açık = 1`

ÇIKTI

```
fd = 3
```

Bağ Sayısı ve unlink: Kod İzleme

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "veri", 4);
3  link("a.txt", "b.txt");      /* katı bağ */
▶  symlink("a.txt", "s.txt");   /* sembolik */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



s.txt: yolu tutan ayrı dosya

symlink ayrı bir i-node (813) oluşturur; içeriği yalnızca “a.txt” yoludur. 812’nin bağ sayısı değişmez.

Adım 3 / 9

DEĞİŞKENLER

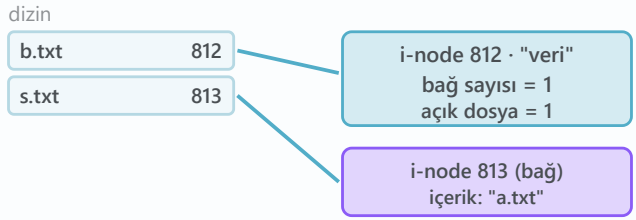
`fd = 3` `bağ = 2` `açık = 1`

ÇIKTI

```
fd = 3
```

Bağ Sayısı ve unlink: Kod İzleme

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "veri", 4);
3  link("a.txt", "b.txt");      /* katı bağ */
4  symlink("a.txt", "s.txt");   /* sembolik */
▶  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



a.txt adı silindi, veri duruyor

unlink("a.txt") adı siler, bağ sayısını 1'e indirir. Veri b.txt üzerinden hâlâ erişilebilir.

Adım 4 / 9

DEĞİŞKENLER

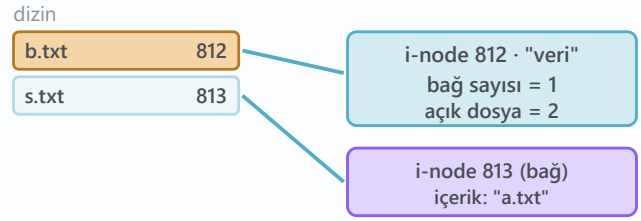
`fd = 3` `bağ = 1` `açık = 1`

ÇIKTI

```
fd = 3
```

Bağ Sayısı ve unlink: Kod İzleme

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "veri", 4);
3  link("a.txt", "b.txt");      /* katı bağ */
4  symlink("a.txt", "s.txt");   /* sembolik */
5  unlink("a.txt");
▶  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



b.txt açıldı

b.txt sorunsuz açılır: r1 = **4**, açık dosya sayısı 2.

Adım 5 / 9

DEĞİŞKENLER

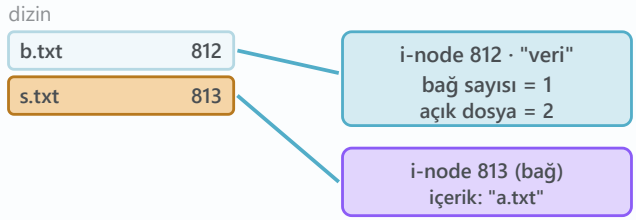
`r1 = 4` `bağ = 1` `açık = 2`

ÇIKTI

```
fd = 3
r1 = 4
```

Bağ Sayısı ve unlink: Kod İzleme

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "veri", 4);
3  link("a.txt", "b.txt");      /* katı bağ */
4  symlink("a.txt", "s.txt");   /* sembolik */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
▶  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



s.txt → "a.txt" → yok

s.txt açılınca çekirdek içindeki yolu ("a.txt") izler; o ad artık yok. Sembolik bağ **sarkar**: r2 = −1, ENOENT.

Adım 6 / 9

DEĞİŞKENLER

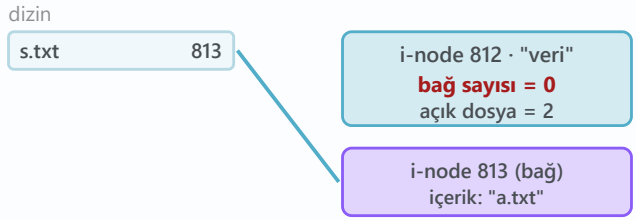
`r2 = -1` `errno = ENOENT`

ÇIKTI

```
fd = 3
r1 = 4
r2 = -1 (ENOENT)
```

Bağ Sayısı ve unlink: Kod İzleme

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "veri", 4);
3  link("a.txt", "b.txt");      /* katı bağ */
4  symlink("a.txt", "s.txt");   /* sembolik */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
▶  unlink("b.txt");
9  close(r1);
10 close(fd);
```



hiç adı kalmadı, ama iki fd açık

Son ad da silindi: bağ sayısı **0**. Ama iki tanımlayıcı açık olduğu için i-node ve veri henüz serbest bırakılmaz; ls onu artık göstermez.

DEĞİŞKENLER

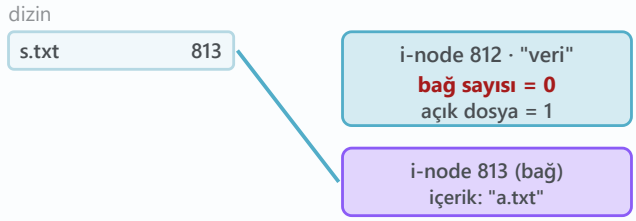
`bağ = 0` `açık = 2`

ÇIKTI

```
fd = 3
r1 = 4
r2 = -1 (ENOENT)
```

Bağ Sayısı ve unlink: Kod İzleme

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "veri", 4);
3  link("a.txt", "b.txt");      /* katı bağ */
4  symlink("a.txt", "s.txt");   /* sembolik */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
▶  close(r1);
10 close(fd);
```



r1 kapandı

close(r1): açık dosya sayısı 1. fd 3 üzerinden okuma/yazma hâlâ çalışır.

Adım 8 / 9

DEĞİŞKENLER

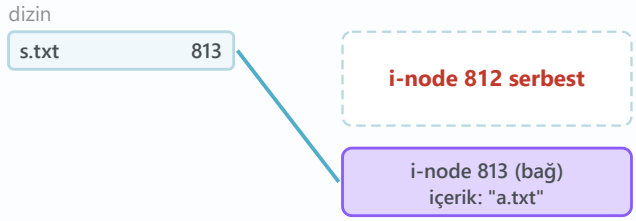
`bağ = 0` `açık = 1`

ÇIKTI

```
fd = 3
r1 = 4
r2 = -1 (ENOENT)
```

Bağ Sayısı ve unlink: Kod İzleme

```
1 int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2 write(fd, "veri", 4);
3 link("a.txt", "b.txt");      /* katı bağ */
4 symlink("a.txt", "s.txt");   /* sembolik */
5 unlink("a.txt");
6 int r1 = open("b.txt", O_RDONLY);
7 int r2 = open("s.txt", O_RDONLY);
8 unlink("b.txt");
9 close(r1);
▶ close(fd);
```



bağ 0 ve açık 0: bloklar serbest

Son tanımlayıcı da kapandı: iki sayaç da 0, i-node ve veri blokları serbest kalır. Geçici dosyaların “aç, sonra hemen unlink et” kalıbı buna dayanır.

Adım 9 / 9

DEĞİŞKENLER

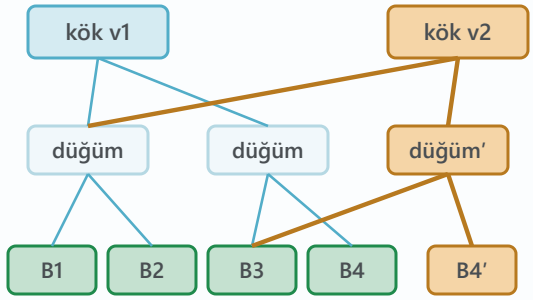
bağ = 0 açık = 0

ÇIKTI

```
fd = 3
r1 = 4
r2 = -1 (ENOENT)
```

ext4'ün Ötesi: XFS, Btrfs ve Diğerleri

Dosya sistemi	Öne çıkan özellik	Tipik kullanım
ext4	Günlükleme, extent, olgun ve hızlı fsck	Çoğu dağıtımda varsayılan
XFS	Paralel G/Ç, büyük dosyalar, gecikmeli ayırma	RHEL varsayılanı, büyük sunucular
Btrfs	Yazınca kopyala, anlık görüntü, sağlama toplamı, sıkıştırma, alt birimler	Fedora, openSUSE varsayılanı
F2FS	Flash'a uygun günlük yapılı (log-structured)	Android telefonlar
tmpfs	Tamamen RAM'de (sayfa önbelleği + takas)	/tmp, /run, /dev/shm
NFS, SMB	Ağ dosya sistemleri	Paylaşılan ev dizinleri



B4 değişince yalnız yol kopyalanır;
kök v1 = anlık görüntü (snapshot)

VFS sayesinde hepsi aynı sistem çağrılarıyla kullanılır; `findmnt` veya `df -T` hangi dizinde hangi dosya sisteminin bağlı olduğunu gösterir.

/proc ve /sys: Çekirdeğe Açılan Pencereler

```
$ cat /proc/loadavg
0.42 0.37 0.30 2/1234 98765
$ grep -E "MemTotal|MemAvailable" /proc/meminfo
MemTotal:       16303452 kB
MemAvailable:    9876544 kB
$ ls /proc/self
cmdline  cwd  environ  exe  fd  maps  status  …
$ cat /sys/block/nvme0n1/queue/scheduler
[none] mq-deadline kyber bfq
$ sysctl vm.swappiness        # = /proc/sys/vm/swappiness
vm.swappiness = 60
```

/proc

Süreç başına dizinler (`/proc/PID/status` , `maps` , `fd/`) ve çekirdek istatistikleri. `ps` , `top` , `free` verisini buradan okur. `/proc/sys` : ayarlanabilir çekirdek parametreleri.

/sys (sysfs)

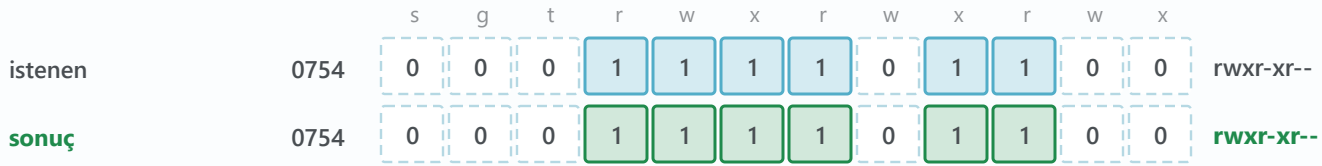
Aygıt modeli: her aygıt, sürücü ve veri yolu bir dizin; **dosya başına tek değer**. `/sys/fs/cgroup` cgroup v2 arayüzüdür.

Etkileşimli simülatör: Linux Süreç Gezgini: `ps` , `top` ve `/proc` ile PID/PPID ilişkileri, süreç ağacı, dosya tanımlayıcıları ve bellek haritaları.

GÜVENLİK · ADIM ADIM

chmod ve umask Hesapları

$ chmod 754 betik.sh



Her sekizli rakam üç bittir: 7 = 111 = rwx, 5 = 101 = r-x, 4 = 100 = r--.

Adım 1 / 5

Kontrol: `stat -c "%a %A" dosya` sekizli ve sembolik izni birlikte gösterir; `umask -S` maskeyi sembolik yazar. Etkileşimli simülatör: İşletim Sistemi Koruma ve İzinler Laboratuvarı.

chmod ve umask Hesapları

$ chmod g+w,o-r betik.sh

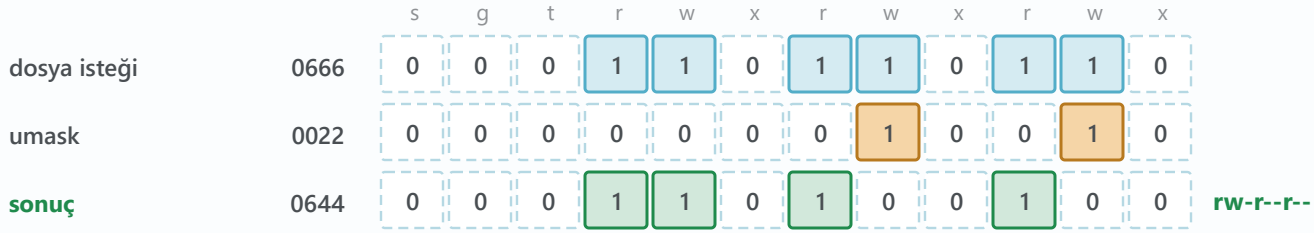
		s	g	t	r	w	x	r	w	x	r	w	x	
önceki	0754	0	0	0	1	1	1	1	0	1	1	0	0	rwxr-xr--
g+w: OR 020	0020	0	0	0	0	0	0	0	1	0	0	0	0	
o-r: AND ~004	0004	0	0	0	0	0	0	0	0	0	1	0	0	
sonuç	0770	0	0	0	1	1	1	1	1	1	0	0	0	**rwxrwx---**

Sembolik mod mevcut bitleri değiştirir: grup yazma biti eklenir, diğerlerinin okuma biti silinir → 0770.

Kontrol: `stat -c "%a %A" dosya` sekizli ve sembolik izni birlikte gösterir; `umask -S` maskeyi sembolik yazar. Etkileşimli simülatör: İşletim Sistemi Koruma ve İzinler Laboratuvarı.

chmod ve umask Hesapları

$ umask 022 ; touch a ; mkdir d



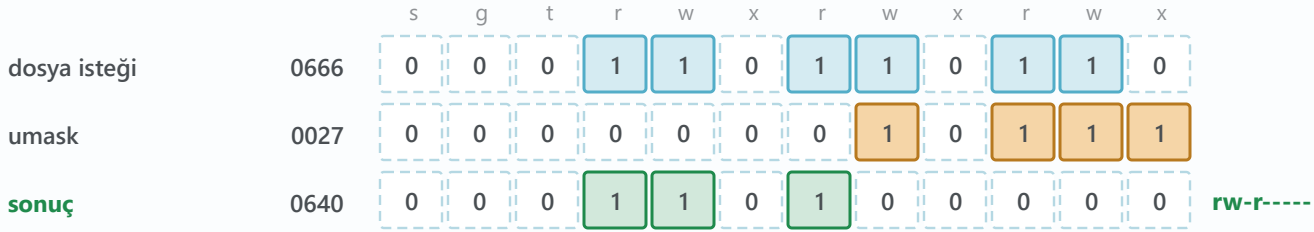
Yeni dosya 666, dizin 777 ister; umask'taki bitler **silinir**: 666 & ~022 = 0644, dizin 777 & ~022 = 0755. (Çıkarma değil, bit temizleme!)

Kontrol: `stat -c "%a %A" dosya` sekizli ve sembolik izni birlikte gösterir; `umask -S` maskeyi sembolik yazar. Etkileşimli simülatör: İşletim Sistemi Koruma ve İzinler Laboratuvarı.

GÜVENLİK · ADIM ADIM

chmod ve umask Hesapları

$ umask 027 ; touch a ; mkdir d

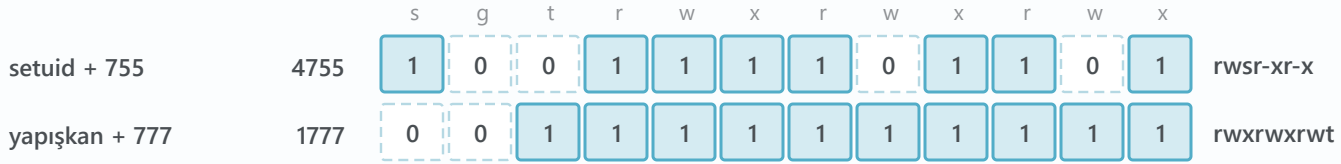


Daha sıkı umask: dosya 666 & ~027 = 0640 (rw-r-----), dizin 777 & ~027 = 0750 (rwxr-x---). Diğerleri hiçbir şey göremez.

Kontrol: `stat -c "%a %A" dosya` sekizli ve sembolik izni birlikte gösterir; `umask -S` maskeyi sembolik yazar. Etkileşimli simülatör: İşletim Sistemi Koruma ve İzinler Laboratuvarı.

chmod ve umask Hesapları

$ chmod 4755 passwd ; chmod 1777 /tmp



4xxx setuid: çalışan süreç dosya sahibinin kimliğini alır → rwsr-xr-x. **2**xxx setgid (dizinde: yeni dosyalar grubu miras alır). **1**xxx yapışkan bit: /tmp'de herkes yazar ama yalnız kendi dosyasını silebilir → rwxrwxrwt.

Kontrol: `stat -c "%a %A" dosya` sekizli ve sembolik izni birlikte gösterir; `umask -S` maskeyi sembolik yazar. Etkileşimli simülatör: İşletim Sistemi Koruma ve İzinler Laboratuvarı.

GÜVENLİK

Erişim Denetimi, root ve Yetenekler

1. euid = 0 (root) → izin ver (çalıştırma için en az bir **x** biti gerekir)
2. euid = dosya sahibi → **yalnız** sahip bitlerine bak
3. egid ya da ek gruplardan biri = dosya grubu → grup bitleri
4. aksi halde diğer bitleri

İlk eşleşen sınıf kazanır: `----rw-rw-` dosyasını sahibi ali **okuyamaz**, başkaları okur.

Yetenekler (capabilities): `setcap cap_net_raw+ep /usr/bin/ping` ile `ping` setuid root olmadan ham soket açar.

Çağrı	İşlev
`chmod`	Dosyanın izin modunu değiştirir
`chown`	Sahip ve grubu değiştirir
`access`	Erişimi **gerçek** uid/gid ile sınar
`getuid`, `geteuid`	Gerçek / etkin uid
`getgid`, `getegid`	Gerçek / etkin gid
`setuid`, `setresuid`	uid değiştirir (ayrıcalık bırakma)
`setgid`	gid değiştirir
`umask`	Yeni dosyalar için izin maskesi

Kullanıcı ve grup kimlikleri `/etc/passwd` ve `/etc/group`'ta; parola özetleri yalnız root'un okuyabildiği `/etc/shadow`'da tutulur. `sudo` yetkilendirilmiş kullanıcıya komut başına root yetkisi verir.

Dosya İzin Denetimi: Kod İzleme

```
▶ int izin_var(cred *c, inode *f, int istek) {
2     int bit, m = f->mode;        /* r4 w2 x1 */
3     if (c->euid == 0)            /* root */
4         return !(istek & 1) || (m & 0111);
5     if (c->euid == f->uid)
6         bit = (m >> 6) & 7;      /* sahip */
7     else if (grupta(c, f->gid))
8         bit = (m >> 3) & 7;      /* grup */
9     else
10        bit = m & 7;             /* diğer */
11    return (bit & istek) == istek;
12 }
```

rapor.txt: sahip ali, grup ekip, mod 0640



istek: — · —

rapor.txt: sahibi ali, grubu ekip, mod 0640 = rw-r-----. veli ekip grubunda, can değil.

Adım 1 / 7

DEĞİŞKENLER

`mode = 0640`

ÇIKTI

Dosya İzin Denetimi: Kod İzleme

```
1  int izin_var(cred *c, inode *f, int istek) {
2      int bit, m = f->mode;        /* r4 w2 x1 */
3      if (c->euid == 0)            /* root */
4          return !(istek & 1) || (m & 0111);
▶      if (c->euid == f->uid)
▶          bit = (m >> 6) & 7;      /* sahip */
7      else if (grupta(c, f->gid))
8          bit = (m >> 3) & 7;      /* grup */
9      else
10         bit = m & 7;             /* diğer */
▶      return (bit & istek) == istek;
12 }
```

rapor.txt: sahip ali, grup ekip, mod 0640



istek: ali (uid 1000) · w

izin

ali dosyanın sahibi: **sahip** bitleri seçilir, bit = 6 (rw-). 6 & 2 == 2: **izin**.

Adım 2 / 7

DEĞİŞKENLER

`euid = 1000` `istek = w` `sınıf = sahip` `sonuç = izin`

ÇIKTI

```
ali w: izin
```

Dosya İzin Denetimi: Kod İzleme

```
1  int izin_var(cred *c, inode *f, int istek) {
2      int bit, m = f->mode;        /* r4 w2 x1 */
3      if (c->euid == 0)            /* root */
4          return !(istek & 1) || (m & 0111);
▶      if (c->euid == f->uid)
6          bit = (m >> 6) & 7;      /* sahip */
▶      else if (grupta(c, f->gid))
▶          bit = (m >> 3) & 7;      /* grup */
9      else
10         bit = m & 7;             /* diğer */
▶      return (bit & istek) == istek;
12 }
```

rapor.txt: sahip ali, grup ekip, mod 0640



istek: veli (uid 1001) · w

ret (EACCES)

veli ekip grubunda: **grup** bitleri seçilir, bit = 4 (r--). 4 & 2 ≠ 2: **ret**.

Adım 3 / 7

DEĞİŞKENLER

`euid = 1001` `istek = w` `sınıf = grup` `sonuç = ret`

ÇIKTI

```
ali w: izin   veli w: ret
```

Dosya İzin Denetimi: Kod İzleme

```
1  int izin_var(cred *c, inode *f, int istek) {
2      int bit, m = f->mode;        /* r4 w2 x1 */
3      if (c->euid == 0)            /* root */
4          return !(istek & 1) || (m & 0111);
▶      if (c->euid == f->uid)
6          bit = (m >> 6) & 7;      /* sahip */
▶      else if (grupta(c, f->gid))
▶          bit = (m >> 3) & 7;      /* grup */
9      else
10         bit = m & 7;             /* diğer */
▶      return (bit & istek) == istek;
12 }
```

rapor.txt: sahip ali, grup ekip, mod 0640



istek: veli (uid 1001) · r

izin

veli ekip grubunda: **grup** bitleri seçilir, bit = 4 (r--). 4 & 4 == 4: **izin**.

Adım 4 / 7

DEĞİŞKENLER

`euid = 1001` `istek = r` `sınıf = grup` `sonuç = izin`

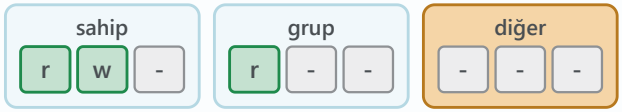
ÇIKTI

```
ali w: izin   veli w: ret
veli r: izin
```

Dosya İzin Denetimi: Kod İzleme

```
 1 int izin_var(cred *c, inode *f, int istek) {
 2     int bit, m = f->mode;        /* r4 w2 x1 */
 3     if (c->euid == 0)            /* root */
 4         return !(istek & 1) || (m & 0111);
 ▶     if (c->euid == f->uid)
 6         bit = (m >> 6) & 7;      /* sahip */
 ▶     else if (grupta(c, f->gid))
 8         bit = (m >> 3) & 7;      /* grup */
 9     else
 ▶         bit = m & 7;             /* diğer */
 ▶     return (bit & istek) == istek;
12 }
```

rapor.txt: sahip ali, grup ekip, mod 0640



istek: can (uid 1002) · r

ret (EACCES)

can ne sahip ne grupta: **diğer** bitleri seçilir, bit = 0 (---). 0 & 4 ≠ 4: **ret**.

Adım 5 / 7

DEĞİŞKENLER

`euid = 1002` `istek = r` `sınıf = diğer` `sonuç = ret`

ÇIKTI

```
ali w: izin   veli w: ret
veli r: izin   can r: ret
```

Dosya İzin Denetimi: Kod İzleme

```
1  int izin_var(cred *c, inode *f, int istek) {
2      int bit, m = f->mode;        /* r4 w2 x1 */
▶      if (c->euid == 0)            /* root */
▶          return !(istek & 1) || (m & 0111);
5      if (c->euid == f->uid)
6          bit = (m >> 6) & 7;      /* sahip */
7      else if (grupta(c, f->gid))
8          bit = (m >> 3) & 7;      /* grup */
9      else
10         bit = m & 7;             /* diğer */
11     return (bit & istek) == istek;
12 }
```

rapor.txt: sahip ali, grup ekip, mod 0640



istek: root (uid 0) · r

euid 0: izin bitlerine bakılmaz

izin

root (euid 0): okuma ve yazma için izin bitlerine hiç bakılmaz; **izin**. Yalnız çalıştırmada en az bir x biti aranır.

Adım 6 / 7

DEĞİŞKENLER

`euid = 0` `istek = r` `sınıf = root` `sonuç = izin`

ÇIKTI

```
ali w: izin   veli w: ret
veli r: izin   can r: ret
root r: izin
```

Dosya İzin Denetimi: Kod İzleme

```
1  int izin_var(cred *c, inode *f, int istek) {
2      int bit, m = f->mode;        /* r4 w2 x1 */
3      if (c->euid == 0)            /* root */
4          return !(istek & 1) || (m & 0111);
▶      if (c->euid == f->uid)
▶          bit = (m >> 6) & 7;      /* sahip */
7      else if (grupta(c, f->gid))
8          bit = (m >> 3) & 7;      /* grup */
9      else
10         bit = m & 7;             /* diğer */
▶      return (bit & istek) == istek;
12 }
```

rapor.txt: sahip ali, grup ekip, mod 0066



istek: ali (uid 1000) · r

ret (EACCES)

Mod 0066 (----rw-rw-) olsun: can ve veli okuyabilir, ama ali sahip sınıfına düşer ve orada bit yok: **ret**. İlk eşleşen sınıf kazanır; diğer bitlere hiç bakılmaz.

Adım 7 / 7

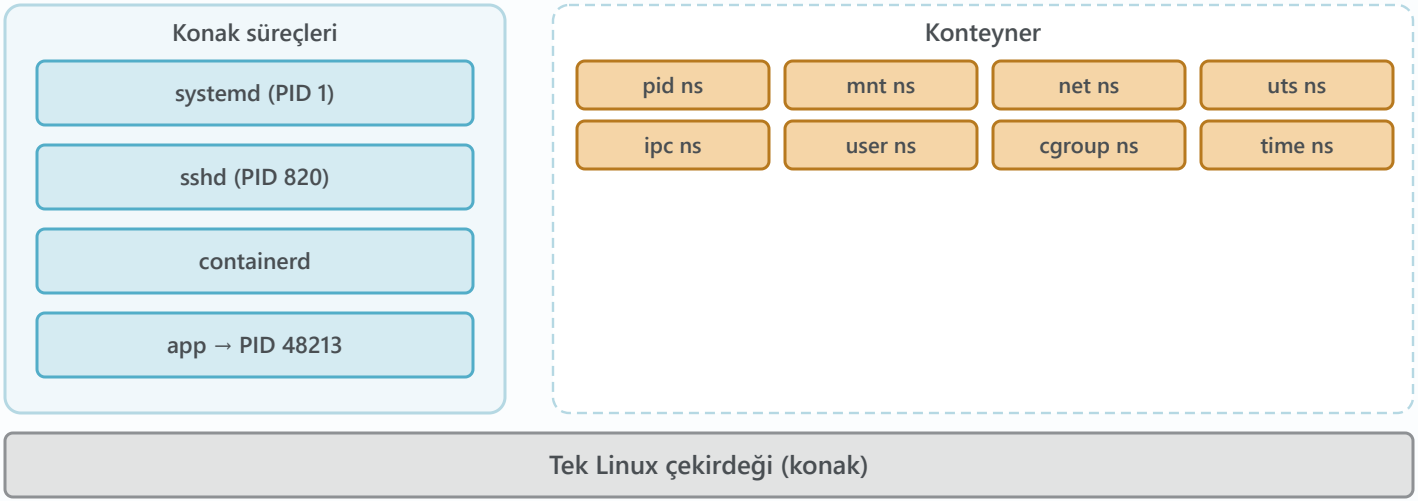
DEĞİŞKENLER

`euid = 1000` `istek = r` `sınıf = sahip` `sonuç = ret`

ÇIKTI

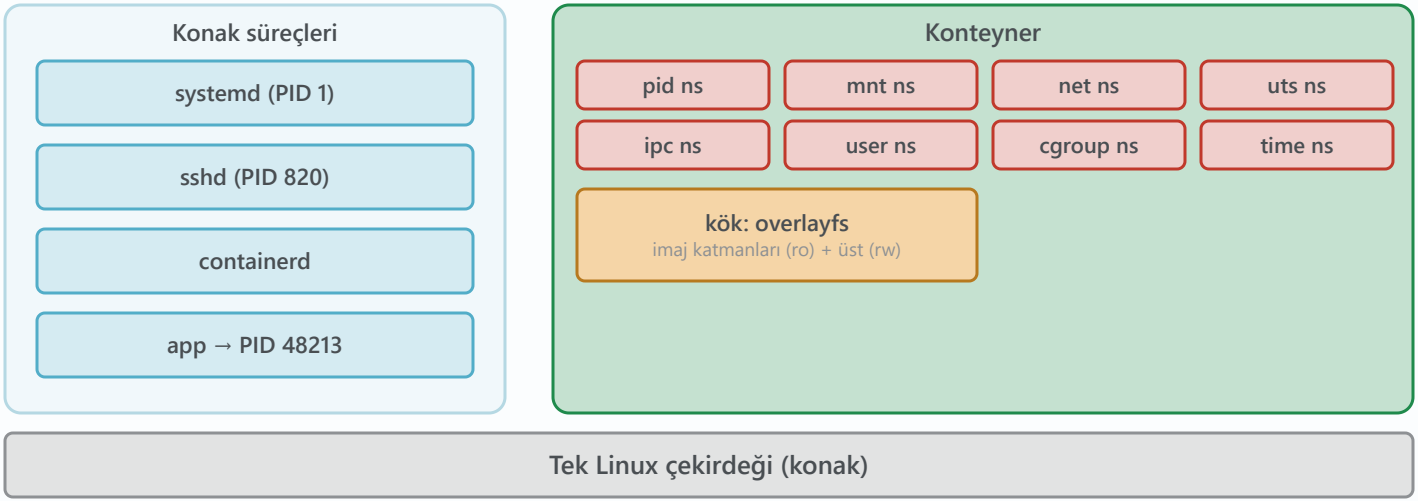
```
ali w: izin   veli w: ret
veli r: izin   can r: ret
root r: izin   0066 ali r: ret
```

Konteynerler: Ad Alanları + cgroups



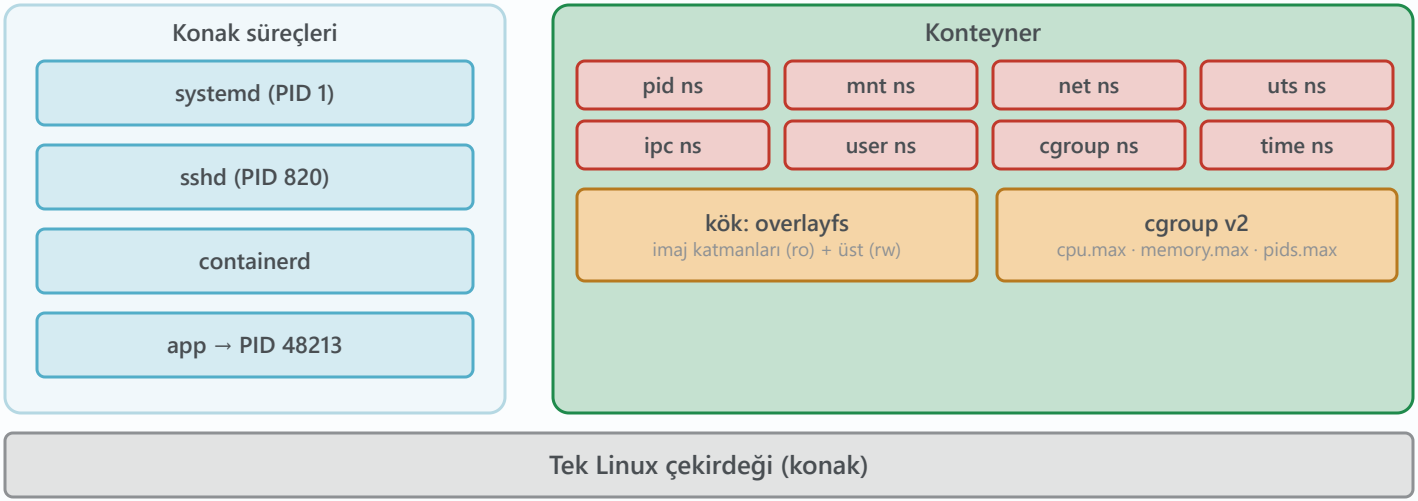
Çalışma zamanı (runc, crun) `clone()` / `unshare()` ile yeni **ad alanları** oluşturur: kendi PID numaraları, bağlama ağacı, ağ yığını, ana makine adı, IPC, kullanıcı eşlemesi (içerideki root dışarıda sıradan uid), cgroup ve saat görünümü.

Konteynerler: Ad Alanları + cgroups



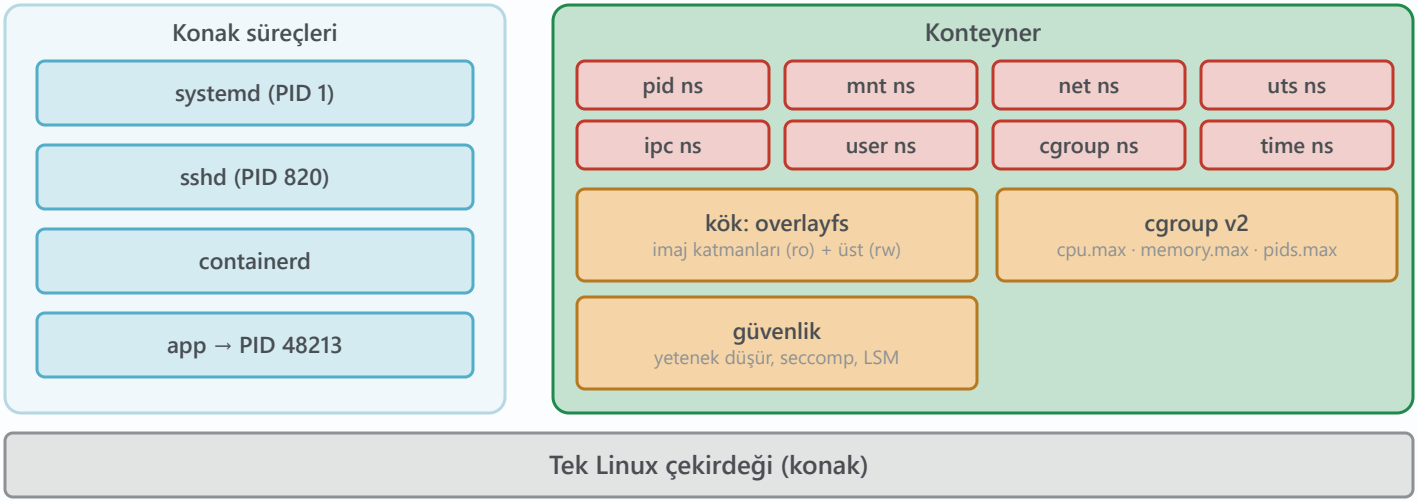
Bağlama ad alanında `pivot_root` ile imajın kök dosya sistemine geçilir. Kök genelde **overlayfs**'tir: salt okunur imaj katmanları + konteynere özel yazılabilir üst katman.

Konteynerler: Ad Alanları + cgroups



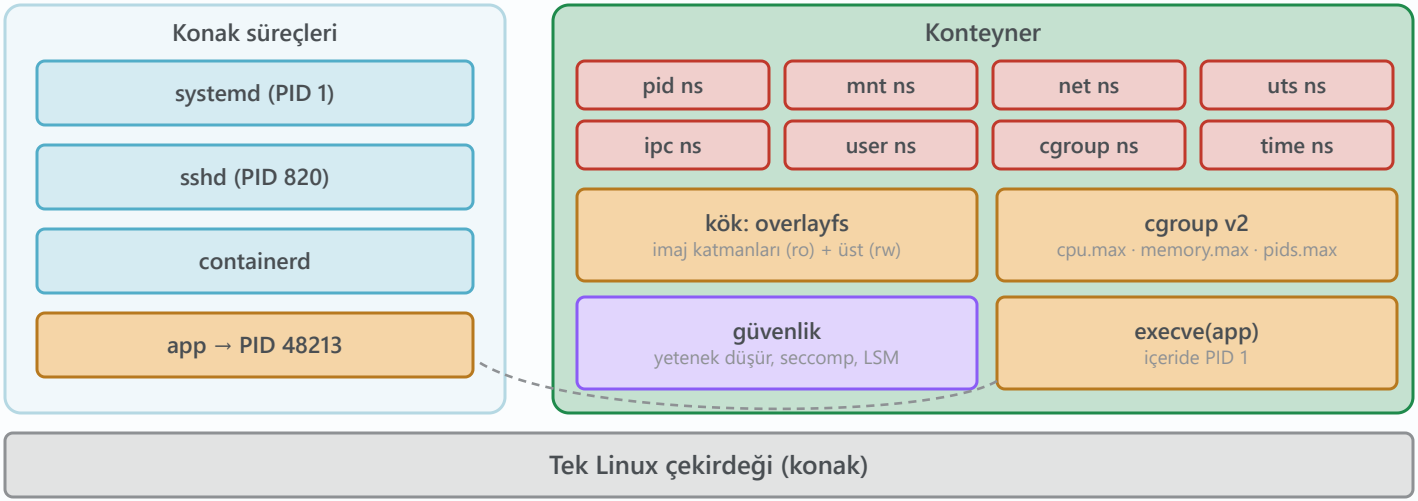
cgroup v2 kaynakları sınırlar: `cpu.max = "50000 100000"` (yarım CPU), `memory.max = 512M`, `pids.max = 100`. Ad alanı ne **görüleceğini**, cgroup ne kadar **kullanılacağını** belirler.

Konteynerler: Ad Alanları + cgroups



Saldırı yüzeyi daraltılır: gereksiz yetenekler düşürülür, `no_new_privs`, seccomp filtresi (onlarca riskli sistem çağrısı engellenir) ve AppArmor/SELinux profili uygulanır.

Konteynerler: Ad Alanları + cgroups



Son olarak `execve`: uygulama içeride PID 1, konakta PID 48213. Hepsi **aynı çekirdeği** paylaşır: VM'den çok hafiftir ama çekirdek açığı tüm konteynerleri etkiler.

GÜVENLİK

seccomp ve Linux Güvenlik Modülleri (LSM)



herhangi bir aşama "hayır" derse: EPERM / EACCES (seccomp: süreç öldürülebilir)

SELinux

Etiket tabanlı zorunlu erişim denetimi (MAC): her süreç ve dosyanın bir tipi var; politika hangi tipin hangisine ne yapabileceğini söyler. RHEL, Fedora, Android.

AppArmor

Yol tabanlı profiller: "bu program yalnız şu dosyalara erişebilir". Yazması kolay. Ubuntu, SUSE, Debian.

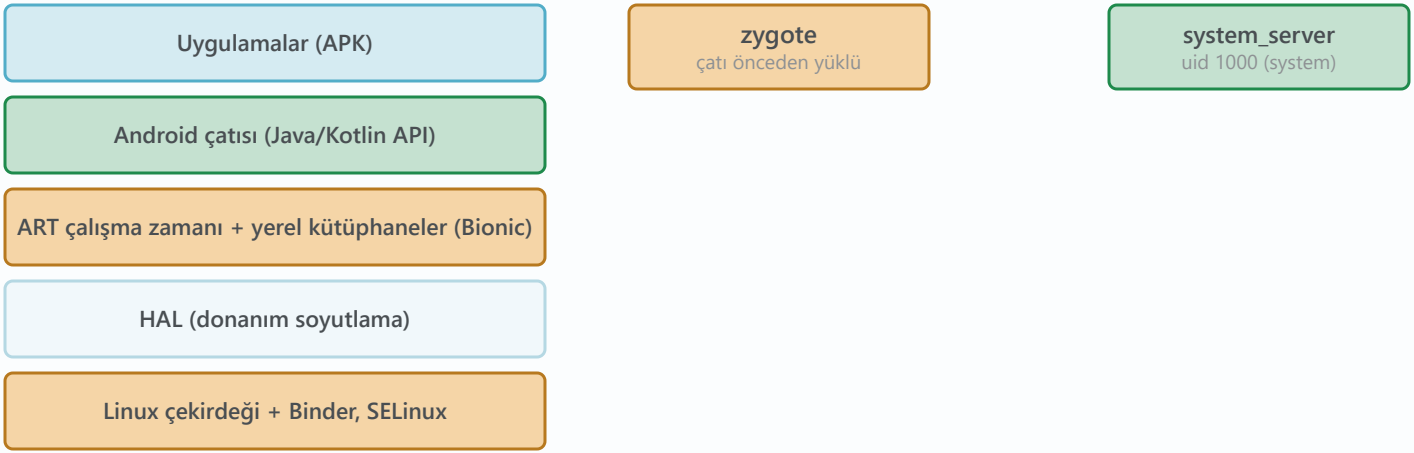
Landlock (5.13+)

Ayrıcalıksız süreç **kendini** korumalı alana alır: "bundan sonra yalnız ./veri altına yazabilirim".

seccomp-BPF

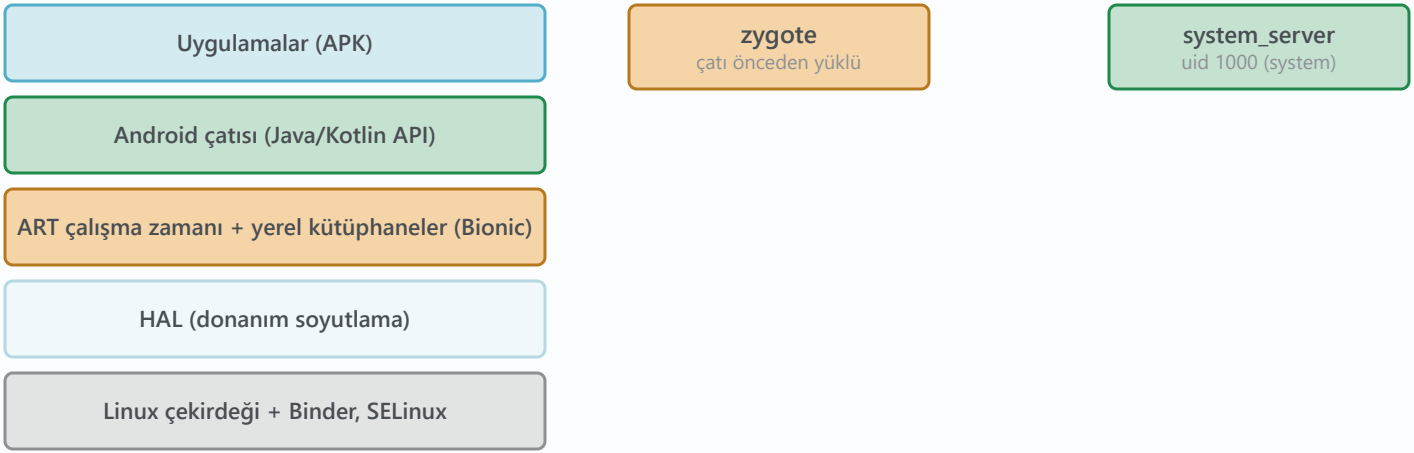
Sürecin yapabileceği sistem çağrılarını filtreler; Chrome, systemd hizmetleri, Docker, Android uygulamaları kullanır.

Android: Binder, Zygote ve Korumalı Alan



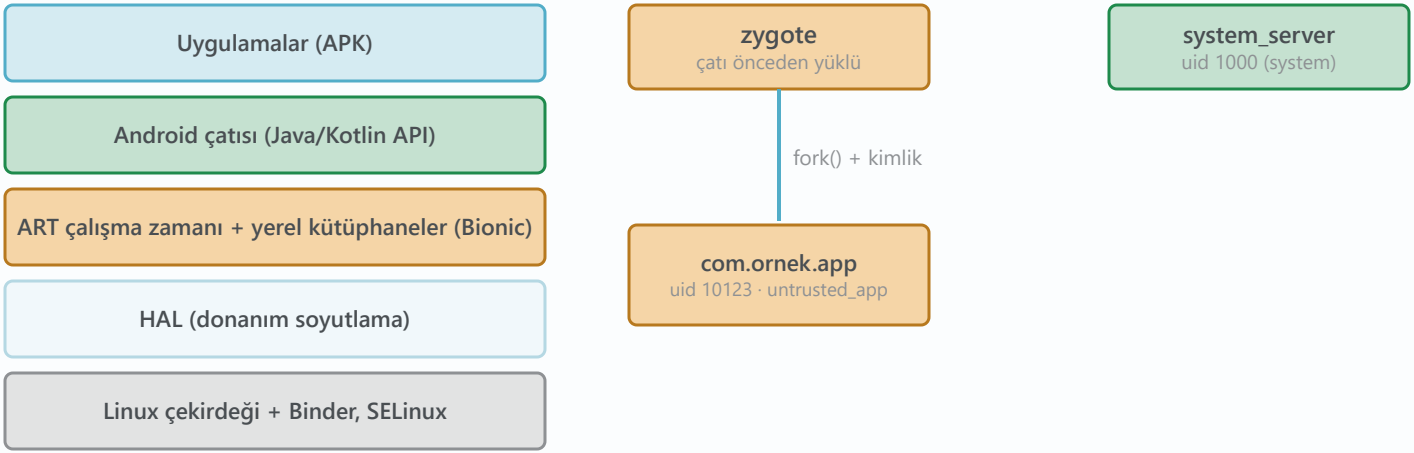
Android Linux çekirdeği üzerinde çalışır ama GNU kullanıcı alanı yoktur: C kütüphanesi **Bionic**, uygulamalar ART'ta çalışır. Çekirdeğe eklenenler: **Binder** IPC, enerji yönetimi (wakelock), zorunlu SELinux.

Android: Binder, Zygote ve Korumalı Alan



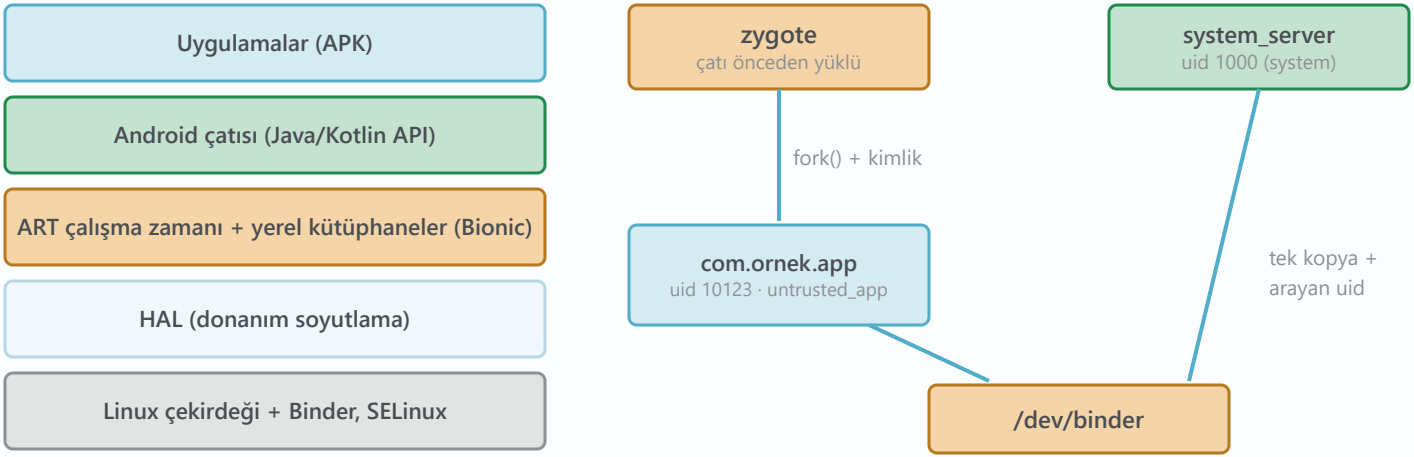
Önyüklemede init **zygote** sürecini başlatır; zygote çatı sınıflarını ve kaynakları bir kez yükleyip bekler.

Android: Binder, Zygote ve Korumalı Alan



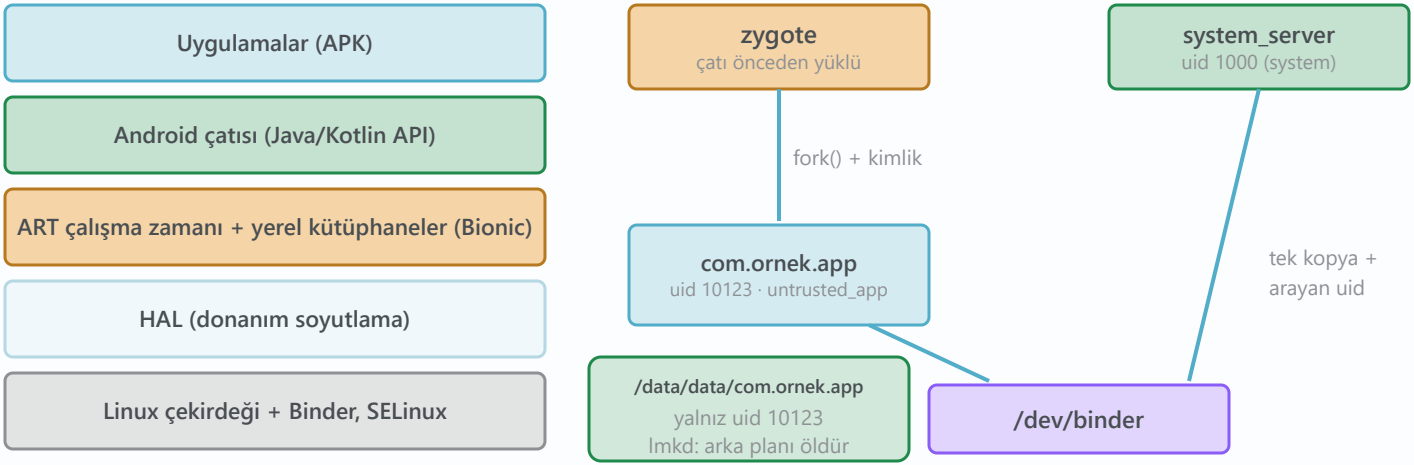
Uygulama açılırken zygote `fork()` eder: yüklü sayfalar **COW** ile paylaşılır, başlatma hızlanır. Çocuk uygulamanın **benzersiz uid**'ine, SELinux bağlamına ve seccomp filtresine geçer.

Android: Binder, Zygote ve Korumalı Alan



Uygulama bir hizmeti (konum, kamera) **Binder** ile çağırır: çekirdek veriyi alıcının eşlenmiş tamponuna **tek kopyayla** taşır ve arayanın uid'ini iletir; hizmet izni buna göre denetler.

Android: Binder, Zygote ve Korumalı Alan



Korumalı alan: her uygulama ayrı bir Linux kullanıcısıdır, verisi yalnız kendisine açıktır. Bellek azalınca kullanıcı alanındaki `lmkd`, PSI sinyalleriyle önemsiz arka plan uygulamalarını öldürür.

Özet: Dersin Kavramları Linux'ta

Konu	Kavram	Linux'taki karşılığı
Yapılar	Sistem çağrısı, monolitik çekirdek	syscall, LKM, eBPF
Süreçler	PCB, oluşturma, IPC	task_struct, fork/exec/wait, sinyal, boru
İş parçacıkları	Çekirdek iş parçacığı modeli	clone(), NPTL 1:1
Çizelgeleme	Adil paylaşım, gerçek zaman	EEVDF, FIFO/RR/DEADLINE, sched_ext
Senkronizasyon	Kilit, koşul değişkeni	futex, spinlock, RCU
Kilitlenme	Tespit ve önleme	lockdep, fcntl EDEADLK
Bellek	Ayırma, parçalanma	bölgeler, buddy listeleri, SLUB
Sanal bellek	Sayfalama, değiştirme	4/5 düzey tablo, COW, MGLRU, zswap, OOM
Dosya sistemleri	i-node, dizin, günlük	VFS, ext4, XFS, Btrfs
G/Ç	Sürücü, çizelgeleme	aygıt dosyaları, blk-mq, io_uring
Çok işlemcili	Yük dengeleme, NUMA	CPU başına kuyruklar, çizelgeleme alanları
Güvenlik	Erişim denetimi, yalıtım	rwx, yetenekler, namespaces, SELinux, seccomp
Tasarım	Mekanizma ve politika ayrımı	cgroup, sched_ext: politika kullanıcıda

Kontrol Soruları

1. `fork()` çekirdekte hangi yapıları kopyalar, hangilerini paylaşır? COW neyi kazandırır?
2. Kabuk `ls | wc -l`'yi nasıl kurar? Ebeveyn boru uçlarını kapatmazsa ne olur?
3. Linux'ta süreç ile iş parçacığı arasındaki fark nedir? clone bayraklarıyla açıklayın.
4. Linux 6.6'da adil sınıfta ne değişti, ne aynı kaldı?
5. `0x00007f3a1c2b5e88` adresinin 4 düzeyli tablodaki dört dizini ve bağıl konumu nedir?
6. SLUB, `task_struct` gibi nesneler için doğrudan sayfa ayırmaya göre neden daha iyidir?
7. `umask 027` iken oluşturulan dosya ve dizinin izinleri nedir?
8. Katı bağ ile sembolik bağ arasındaki fark nedir? Hedef silinince ne olur?
9. ext4 sıralı günlüklemede çökmeden sonra tutarlılık nasıl sağlanır?
10. Konteyner = ad alanları + cgroups: her biri neyi sağlar? VM'den farkı nedir?

▶ **Cevaplar**

Kaynaklar

- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 10: Case Study 1: Unix, Linux and Android.
- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 20: The Linux System.
- R. Love, *Linux Kernel Development*, 3. baskı, Addison-Wesley, 2010.
- M. Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010; man7.org kılavuz sayfaları.
- The Linux Kernel Documentation, docs.kernel.org — scheduler (EEVDF, sched-ext), admin-guide/mm (MGLRU, zswap), block (blk-mq), filesystems (vfs, ext4), userspace-api (seccomp, landlock).
- Android Open Source Project, source.android.com — mimari, Binder, uygulama korumalı alanı.
- D. M. Ritchie, K. Thompson, "The UNIX Time-Sharing System", *Communications of the ACM*, 1974.

Son

Bölüm 15: Linux

Dersin son bölümü — İşletim Sistemleri dersi burada tamamlanıyor. · Terim sözlüğü