Bölüm 2: İşletim Sistemi Yapıları

İşletim Sistemleri

Hizmetler | Süreç, adres alanı, dosya | Çift mod | Sistem çağrıları | POSIX ve Win32

Monolitik | Katmanlı | Mikro çekirdek | Modüler | Hibrit | Exokernel

Sanal makineler | Bağlama ve açılış

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_02_Isletim_Sistemi_Yapilari.html

GİRİŞ

Öğrenme Çıktıları

Hizmetler ve kavramlar

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

Sistem çağrıları

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

Çekirdek yapıları

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

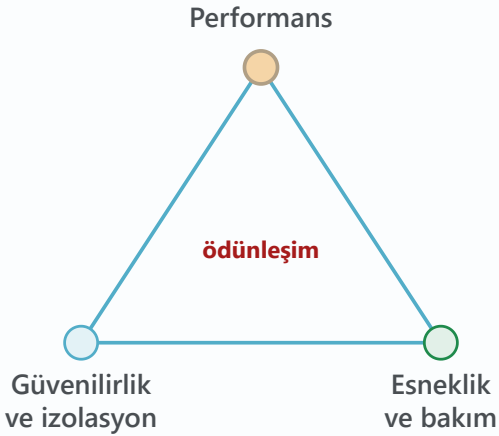
Sanallaştırma ve açılış

Sanal makine ve konteyner kavramlarını, bağlayıcı/yükleyiciyi ve açılış adımlarını tanımak.

GİRİŞ

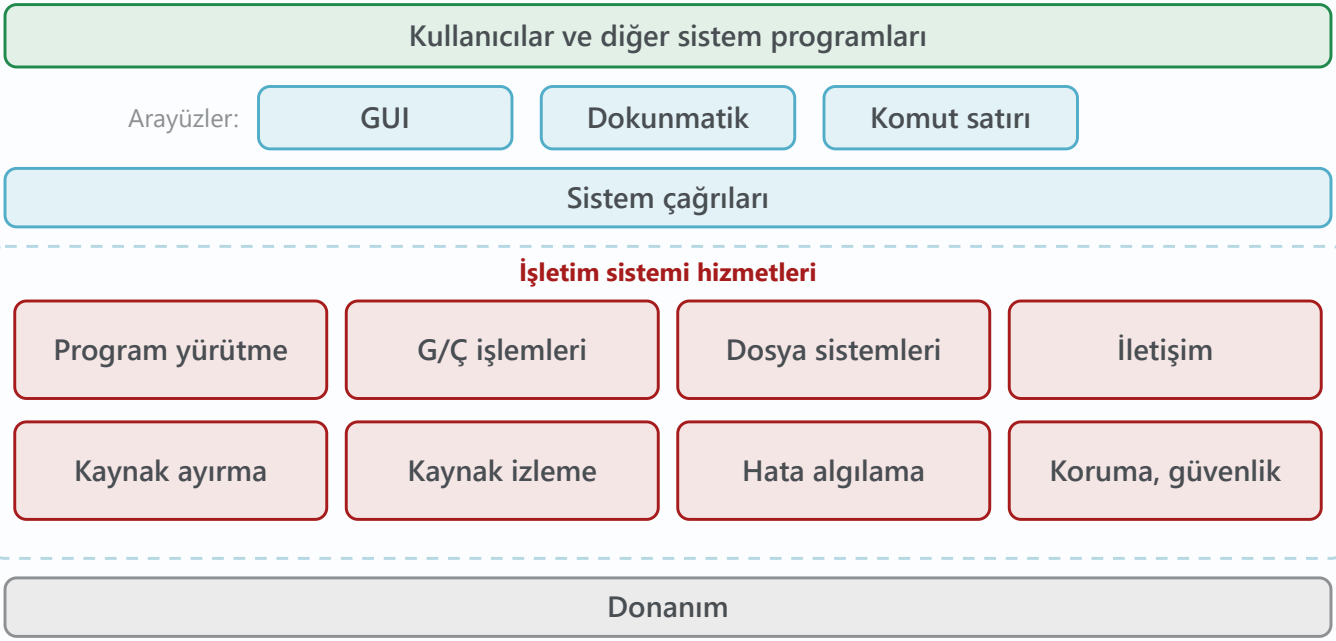
Neden Yapı Önemli?

- İşletim sistemi; süreç, bellek, dosya, G/Ç ve ağ yönetimi gibi çok sayıda bileşenden oluşur.
- Bu bileşenlerin **nerede** çalışacağına (çekirdek / kullanıcı modu) ve **nasıl haberleşeceğine** (fonksiyon çağrısı / ileti) dair stratejiye **işletim sistemi yapısı** denir.
- Yapı; karmaşıklığı yönetmeyi, değişikliği, hata ayıklamayı ve taşınabilirliği doğrudan etkiler.



Tek doğru yapı yoktur: her yapı belirli bir ödünleşim sunar ve belirli kullanım senaryolarına (masaüstü, sunucu, gömülü, güvenlik-kritik) daha uygundur.

İşletim Sistemi Hizmetleri



Hizmetlere kullanıcı arayüzleri ve programlar tarafından sistem çağrıları üzerinden erişilir (Silberschatz, Şekil 2.1'den uyarlanmıştır).

Hizmetler Kime Hizmet Eder?

Kullanıcıya ve programlara yardımcı

- **Arayüz:** komut satırı, GUI, dokunmatik, sesli.
- **Program yürütme:** yükle, çalıştır, normal / hatalı sonlandır.
- **G/Ç işlemleri:** program aygıta doğrudan değil işletim sistemi üzerinden erişir.
- **Dosya sistemi:** oluştur, sil, oku, yaz, ara, izinler.
- **İletişim:** paylaşımlı bellek veya ileti geçirme (yerel / ağ).
- **Hata algılama:** donanım, aygıt ve program hataları.

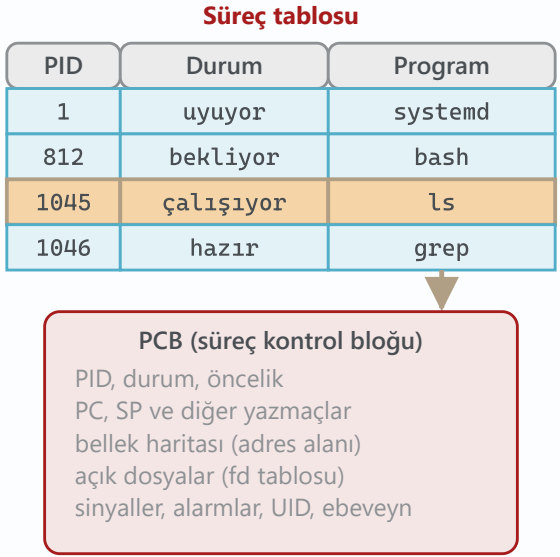
Sistemin verimli ve güvenli çalışması

- **Kaynak ayırma:** CPU, bellek, depolama birden çok sürece paylaştırılır.
- **Kaynak izleme (accounting):** kim, ne kadar kaynak kullandı; günlük kayıtları.
- **Koruma:** süreçler birbirine ve çekirdeğe zarar veremez.
- **Güvenlik:** kimlik doğrulama, yetkilendirme, dış saldırılara karşı savunma.

Kullanıcıların çoğu işletim sistemini sistem çağrılarıyla değil, **sistem programlarıyla** görür: `ls`, `cp`, `ps`, paket yöneticisi, derleyici, dosya yöneticisi…

Süreç (Process)

- Süreç, **yürütülmekte olan bir programdır**; program diskteki pasif dosya, süreç onun etkin örneğidir.
- Her süreçle ilişkili: **adres alanı** (kod, veri, yığın), yazmaçlar (PC, SP), açık dosyalar, alarmlar, ilişkili süreçler ve diğer kaynaklar.
- İşletim sistemi bu bilgileri **süreç tablosunda** tutar; askıya alınan süreç, PCB sayesinde kaldığı yerden devam eder.
- Süreçler birbirleriyle **IPC** (süreçler arası iletişim) ile haberleşebilir.



TEMEL KAVRAMLAR

Süreç Ağacı

Bir süreç `fork()` ile **çocuk süreçler** oluşturabilir; çocuklar da kendi çocuklarını oluşturur. Sonuç bir ağaçtır.

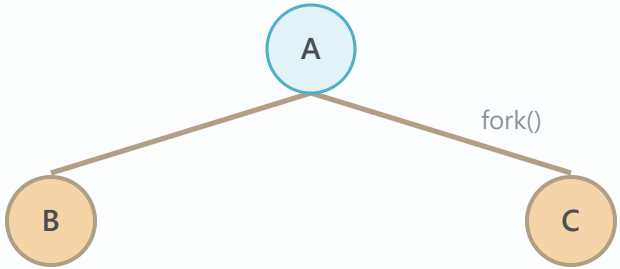


PID 1 (systemd) ağacın köküdür

Başlangıçta yalnızca A süreci vardır. Linux'ta tüm ağacın kökü PID 1 olan `systemd` (veya `init`) sürecidir.

Süreç Ağacı

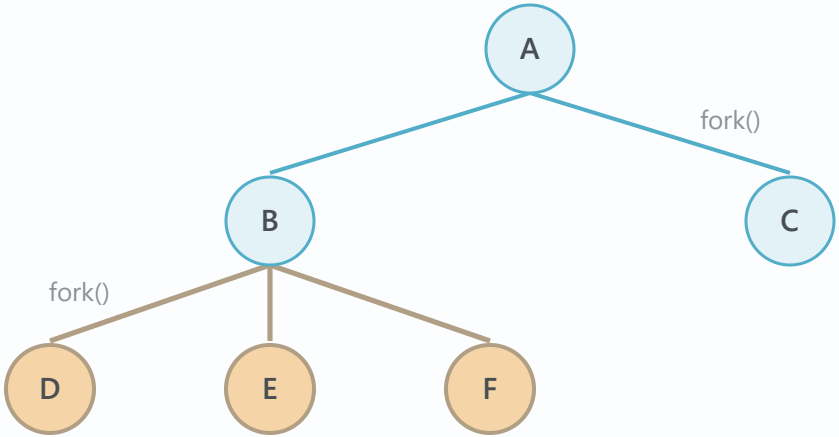
Bir süreç `fork()` ile **çocuk süreçler** oluşturabilir; çocuklar da kendi çocuklarını oluşturur. Sonuç bir ağaçtır.



A süreci iki kez `fork()` çağırır: B ve C, A'nın çocuklarıdır (A onların ebeveynidir).

Süreç Ağacı

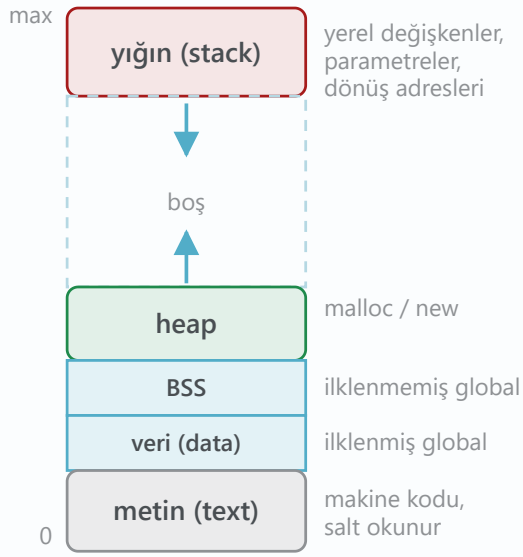
Bir süreç `fork()` ile **çocuk süreçler** oluşturabilir; çocuklar da kendi çocuklarını oluşturur. Sonuç bir ağaçtır.



B süreci D, E ve F olmak üzere üç çocuk oluşturur. `pstree -p` komutu bu ağacı gösterir.

Adres Alanı ve Bellek Kesimleri

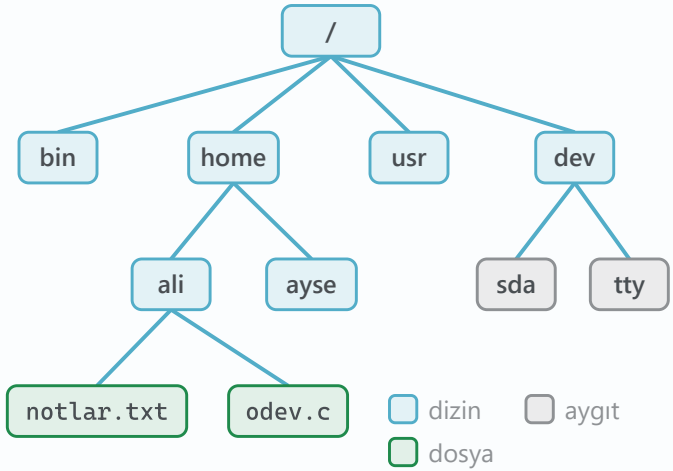
- **Adres alanı:** bir sürecin kullanabildiği adresler kümesi (0 … max). x86-64 Linux'ta kullanıcı alanı 128 TiB'dir.
- Bellekte aynı anda çok sayıda süreç bulunur; işletim sistemi onları birbirinden ve çekirdekten korur.
- **Metin:** makine kodu; salt okunur, aynı programı çalıştıran süreçler ve tüm iş parçacıkları paylaşır.
- **Veri + BSS:** global ve static değişkenler. **Heap:** çalışma anında ayrılan dinamik bellek. İkisi de sürecin iş parçacıklarınca paylaşılır.
- **Yığın:** parametreler, yerel değişkenler, dönüş adresleri; *her iş parçacığının kendi yığını* vardır.
- Fiziksel bellekten büyük adres alanları **sanal bellek** ile mümkündür.



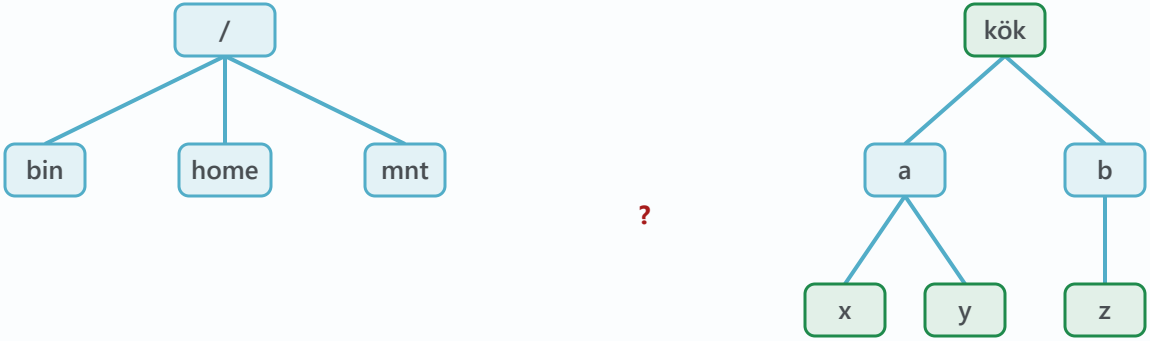
TEMEL KAVRAMLAR

Dosyalar ve Dizin Hiyerarşisi

- **Dosya:** verileri kalıcı olarak saklayan adlandırılmış birim (metin, resim, video, program...).
- İşletim sistemi disk bloklarını ve aygıt ayrıntılarını gizler; temiz bir soyutlama sunar.
- Dosyalar **dizin ağacında** tutulur. Yol adı mutlak (`/home/ali/notlar.txt`) veya çalışma dizinine göre göreli (`ali/notlar.txt`) olabilir.
- Açılan dosyaya küçük bir tam sayı, **dosya tanımlayıcısı** (fd) verilir; 0, 1, 2 = stdin, stdout, stderr.
- **Özel dosyalar:** aygıtlar dosya gibi görünür. Blok (`/dev/sda` , `/dev/nvme0n1`) ve karakter (`/dev/tty` , seri port) aygıtları.



Dosya Sistemini Bağlama (mount)

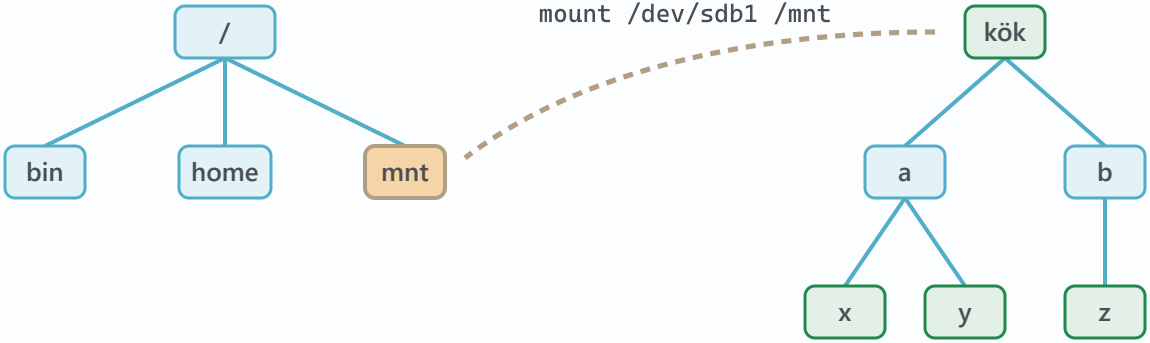


Ana dosya sistemi (/dev/nvme0n1p2)

USB bellek (/dev/sdb1)

Bağlamadan önce USB bellekteki dosya sistemi ayrı bir ağaçtır; ona hiçbir yol adıyla erişilemez.

Dosya Sistemini Bağlama (mount)

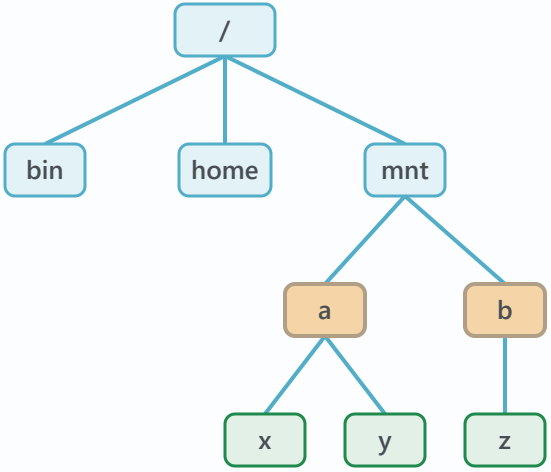


Ana dosya sistemi (/dev/nvme0n1p2)

USB bellek (/dev/sdb1)

`mount`, yeni dosya sisteminin kökünü mevcut ağaçtaki bir **bağlama noktasına** (boş dizin `/mnt`) iliştirir.

Dosya Sistemini Bağlama (mount)



Ana dosya sistemi (/dev/nvme0n1p2)

```
Artık tek ağaç:
/mnt/a/x
/mnt/b/z
```

Bağlandıktan sonra USB'deki dosyalar hiyerarşinin parçasıdır. Windows'ta bunun yerine genellikle sürücü harfleri (C:, D:) kullanılır.

TEMEL KAVRAMLAR

`mount` Komutu ve Seçenekleri

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

`ro` / `rw`

Salt okunur ya da okuma-yazma bağlama. Kurtarma ve adli inceleme için `ro` güvenlidir.

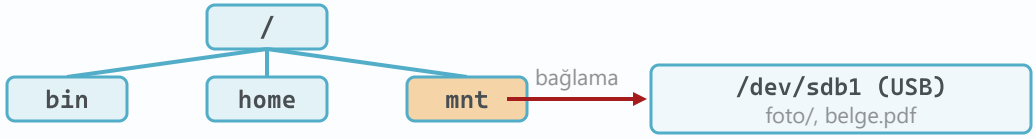
`/etc/fstab`

Açılışta otomatik bağlanacak dosya sistemleri burada (veya systemd mount birimlerinde) tanımlanır.

Masaüstünde

Takılan USB bellek `udisks2` tarafından kendiliğinden `/run/media/…` altına bağlanır.

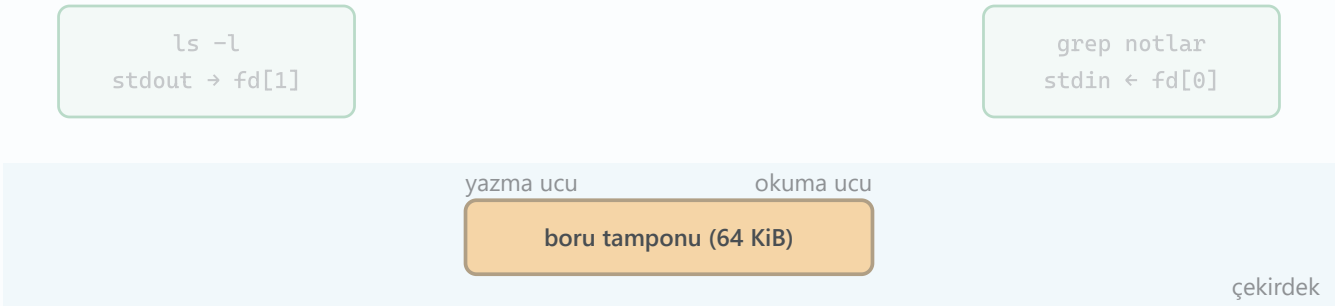
`/mnt/belge.pdf`
artık yol ile erişilir



Boru (pipe)

- İki süreci bağlayan, çekirdekte tamponlanan **tek yönlü** bir bayt kanalıdır; süreçler onu dosya gibi `read` / `write` ile kullanır.
- **Anonim:** `pipe()` ile oluşur, ebeveyn-çocuk süreçler arasında. **Adlandırılmış (FIFO):** `mkfifo` ile dosya sisteminde adı vardır, ilgisiz süreçler kullanabilir.

```
$ ls -l | grep "notlar"
```

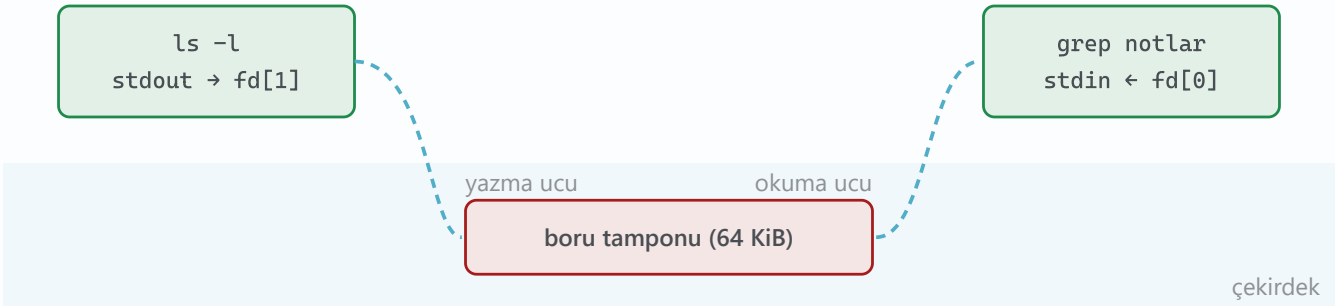


Kabuk önce `pipe(fd)` çağırır: çekirdek bir tampon ve iki uç açar (fd[0] okuma, fd[1] yazma).

Boru (pipe)

- İki süreci bağlayan, çekirdekte tamponlanan **tek yönlü** bir bayt kanalıdır; süreçler onu dosya gibi `read` / `write` ile kullanır.
- **Anonim:** `pipe()` ile oluşur, ebeveyn-çocuk süreçler arasında. **Adlandırılmış (FIFO):** `mkfifo` ile dosya sisteminde adı vardır, ilgisiz süreçler kullanabilir.

```
$ ls -l | grep "notlar"
```

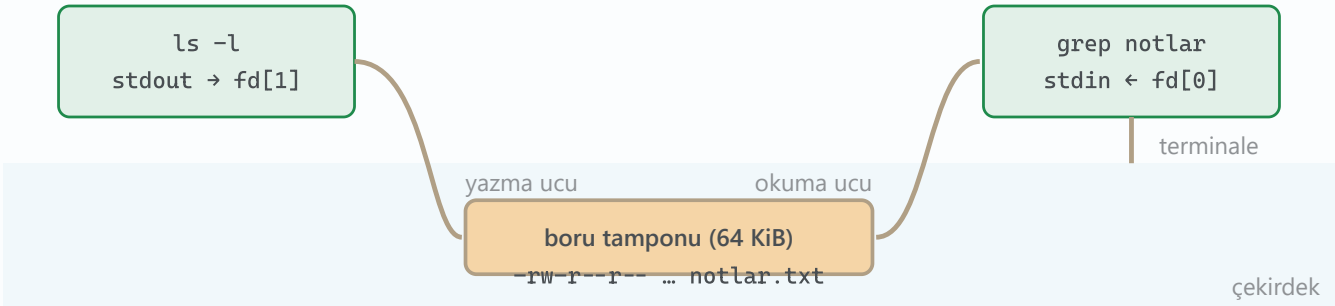


Kabuk iki kez `fork` eder; `dup2` ile ls'in stdout'u yazma ucuna, grep'in stdin'i okuma ucuna yönlendirilir.

Boru (pipe)

- İki süreci bağlayan, çekirdekte tamponlanan **tek yönlü** bir bayt kanalıdır; süreçler onu dosya gibi `read` / `write` ile kullanır.
- **Anonim:** `pipe()` ile oluşur, ebeveyn-çocuk süreçler arasında. **Adlandırılmış (FIFO):** `mkfifo` ile dosya sisteminde adı vardır, ilgisiz süreçler kullanabilir.

```
$ ls -l | grep "notlar"
```

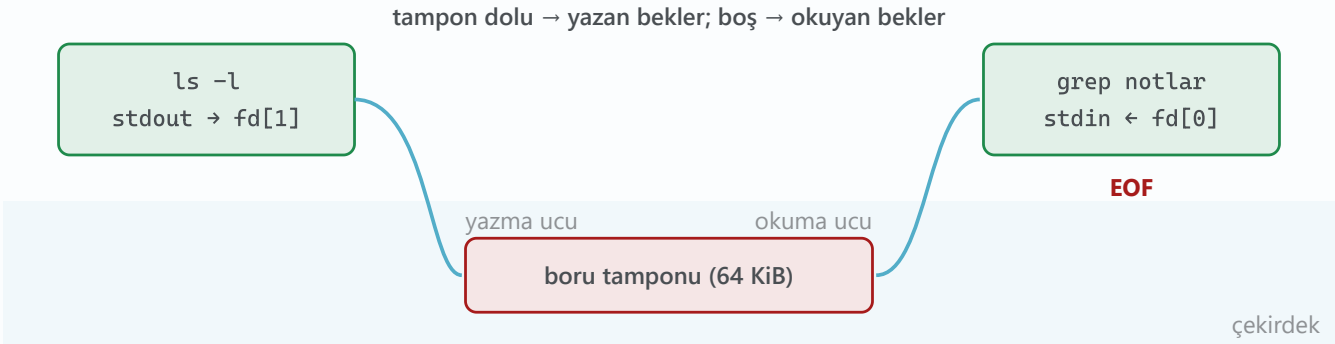


ls listeyi boruya yazar, grep okur ve yalnızca “notlar” içeren satırları terminale yazar.

Boru (pipe)

- İki süreci bağlayan, çekirdekte tamponlanan **tek yönlü** bir bayt kanalıdır; süreçler onu dosya gibi `read` / `write` ile kullanır.
- **Anonim:** `pipe()` ile oluşur, ebeveyn-çocuk süreçler arasında. **Adlandırılmış (FIFO):** `mkfifo` ile dosya sisteminde adı vardır, ilgisiz süreçler kullanabilir.

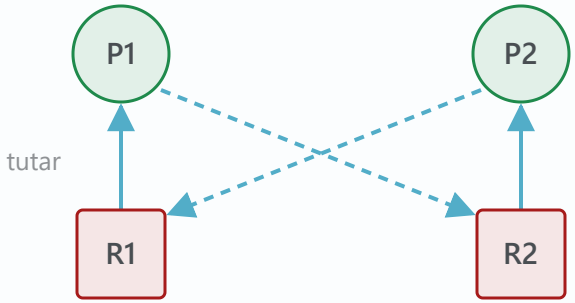
```
$ ls -l | grep "notlar"
```



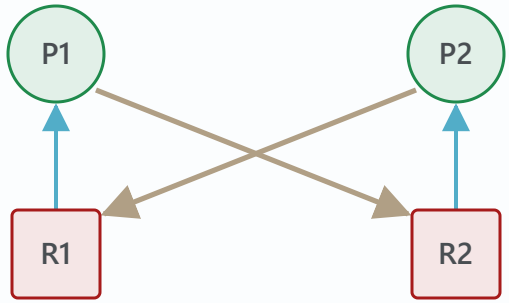
Akış denetimi çekirdekte: tampon dolunca yazan, boşken okuyan bloke olur. ls bitip yazma ucu kapanınca grep dosya sonu (EOF) görür.

Kilitlenme (Deadlock)

- Kümedeki her süreç, **başka bir sürecin tuttuğu** kaynağı beklerse hiçbiri ilerleyemez: kilitlenme.
- Trafikte dört yönden kavşağa girip birbirini bekleyen araçlar gibi.



(a) Potansiyel: istekler henüz yapılmadı



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

Kareden sürece ok: kaynak o süreçte. Süreçten kareye ok: süreç o kaynağı istiyor (kesikli = ileride isteyecek).

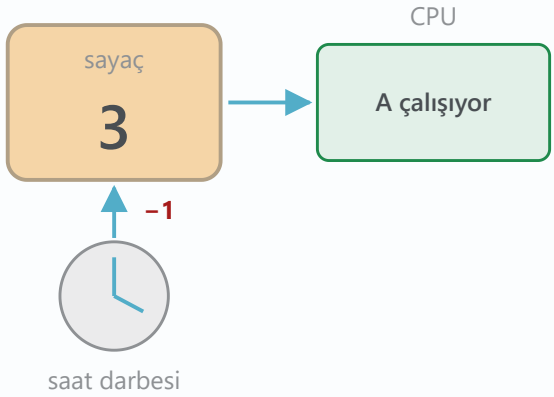
KORUMA

Çift Mod ile Koruma

- İşletim sistemi kendini ve süreçleri donanımın **kullanıcı / çekirdek modu** ayrımıyla korur: G/Ç, zamanlayıcıyı ayarlama, kesmeleri kapatma, sayfa tablolarını değiştirme yalnızca çekirdek modunda yapılabilir.
- Kullanıcı modunda ayrıcalıklı komut denenirse donanım tuzak üretir ve işletim sistemi süreci sonlandırır. Çekirdeğe yalnızca sistem çağrısı, kesme ve istisna yoluyla girilir.
- x86'da halkalar 0–3'tür ama Linux ve Windows yalnızca 0 ve 3'ü kullanır; ARM'da EL0–EL3 vardır; bunlardan EL2 sanallaştırmaya ayrılmıştır.
- Mod ayrımı tek başına yetmez: CPU'yu geri almak için **zamanlayıcı**, süreçlerin belleğini ayırmak için **MMU** gerekir.

Zamanlayıcı (Timer)

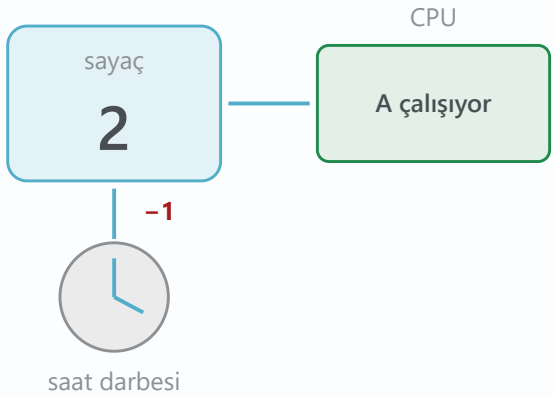
- Amaç: bir sürecin sonsuz döngüye girip CPU'yu **hiç bırakmamasını** önlemek.
- İşletim sistemi sayaca bir değer yükler; bu, **ayrıcalıklı** bir komuttur.
- Fiziksel saatin her darbesinde sayaç azalır; **sıfır olunca kesme** üretilir ve işletim sistemi kontrolü yeniden kazanır.
- İşletim sistemi, zaman dilimini dolduran süreci keser ve CPU'yu başka sürece verir (kesmeli çizelgeleme). Süreci sonlandırmak ancak CPU süresi sınırı (ör. `RLIMIT_CPU`) aşılırsa söz konusudur.
- Linux'ta düzenli tik 100–1000 Hz'dir; boşta tik üretmeyen *tickless* (NO_HZ) mod enerji kazandırır.



İşletim sistemi, A'yı çalıştırmadan önce sayaca 3 yükler.

Zamanlayıcı (Timer)

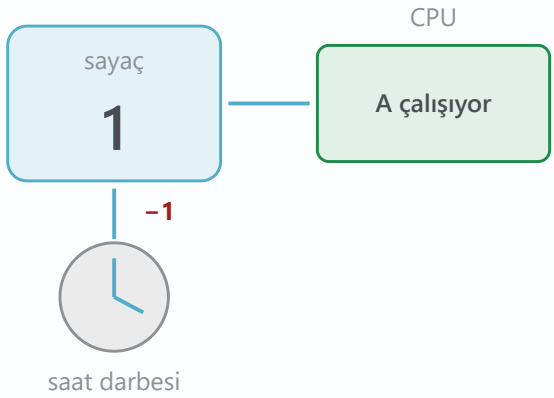
- Amaç: bir sürecin sonsuz döngüye girip CPU'yu **hiç bırakmamasını** önlemek.
- İşletim sistemi sayaca bir değer yükler; bu, **ayrıcalıklı** bir komuttur.
- Fiziksel saatin her darbesinde sayaç azalır; **sıfır olunca kesme** üretilir ve işletim sistemi kontrolü yeniden kazanır.
- İşletim sistemi, zaman dilimini dolduran süreci keser ve CPU'yu başka sürece verir (kesmeli çizelgeleme). Süreci sonlandırmak ancak CPU süresi sınırı (ör. `RLIMIT_CPU`) aşılırsa söz konusudur.
- Linux'ta düzenli tik 100–1000 Hz'dir; boşta tik üretmeyen *tickless* (NO_HZ) mod enerji kazandırır.



Her saat darbesinde sayaç bir azalır: 2.

Zamanlayıcı (Timer)

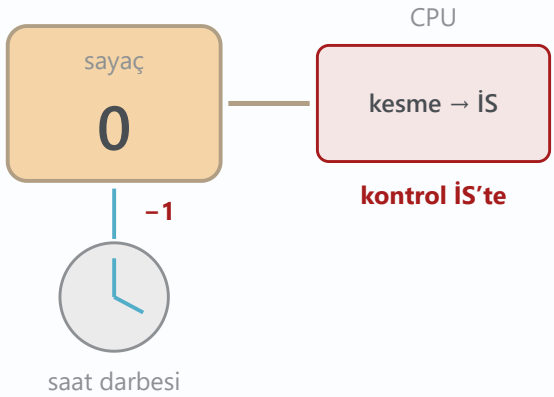
- Amaç: bir sürecin sonsuz döngüye girip CPU'yu **hiç bırakmamasını** önlemek.
- İşletim sistemi sayaca bir değer yükler; bu, **ayrıcalıklı** bir komuttur.
- Fiziksel saatin her darbesinde sayaç azalır; **sıfır olunca kesme** üretilir ve işletim sistemi kontrolü yeniden kazanır.
- İşletim sistemi, zaman dilimini dolduran süreci keser ve CPU'yu başka sürece verir (kesmeli çizelgeleme). Süreci sonlandırmak ancak CPU süresi sınırı (ör. `RLIMIT_CPU`) aşılırsa söz konusudur.
- Linux'ta düzenli tik 100–1000 Hz'dir; boşta tik üretmeyen *tickless* (NO_HZ) mod enerji kazandırır.



Sayaç 1; A hâlâ CPU'da.

Zamanlayıcı (Timer)

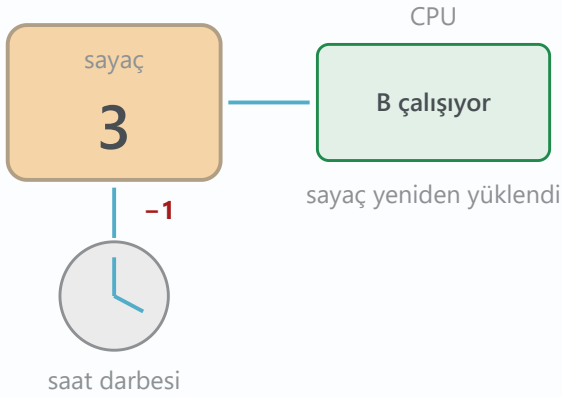
- Amaç: bir sürecin sonsuz döngüye girip CPU'yu **hiç bırakmamasını** önlemek.
- İşletim sistemi sayaca bir değer yükler; bu, **ayrıcalıklı** bir komuttur.
- Fiziksel saatin her darbesinde sayaç azalır; **sıfır olunca kesme** üretilir ve işletim sistemi kontrolü yeniden kazanır.
- İşletim sistemi, zaman dilimini dolduran süreci keser ve CPU'yu başka sürece verir (kesmeli çizelgeleme). Süreci sonlandırmak ancak CPU süresi sınırı (ör. `RLIMIT_CPU`) aşılırsa söz konusudur.
- Linux'ta düzenli tik 100–1000 Hz'dir; boşta tik üretmeyen *tickless* (NO_HZ) mod enerji kazandırır.



Sayaç 0: zamanlayıcı kesmesi. CPU çekirdek moduna geçer, kesme işleyicisi çalışır.

Zamanlayıcı (Timer)

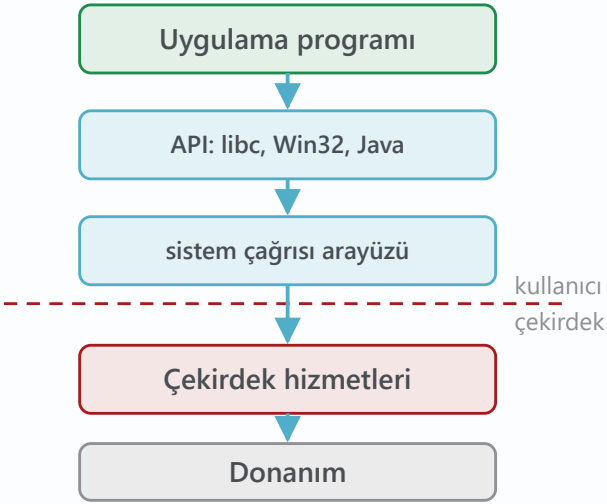
- Amaç: bir sürecin sonsuz döngüye girip CPU'yu **hiç bırakmamasını** önlemek.
- İşletim sistemi sayaca bir değer yükler; bu, **ayrıcalıklı** bir komuttur.
- Fiziksel saatin her darbesinde sayaç azalır; **sıfır olunca kesme** üretilir ve işletim sistemi kontrolü yeniden kazanır.
- İşletim sistemi, zaman dilimini dolduran süreci keser ve CPU'yu başka sürece verir (kesmeli çizelgeleme). Süreci sonlandırmak ancak CPU süresi sınırı (ör. `RLIMIT_CPU`) aşılırsa söz konusudur.
- Linux'ta düzenli tik 100–1000 Hz'dir; boşta tik üretmeyen *tickless* (NO_HZ) mod enerji kazandırır.



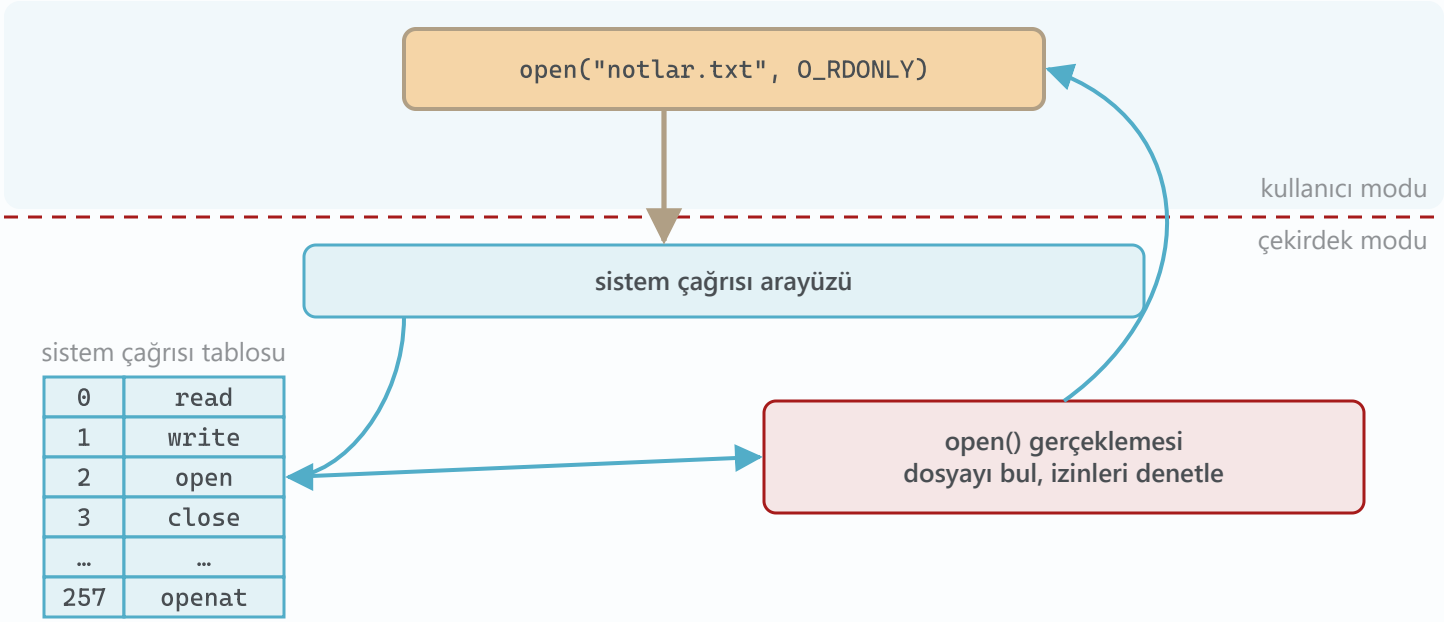
Çizelgeleyici A'nın dilimi dolduğu için B'yi seçer ve sayacı yeniden yükler.

Sistem Çağrısı Nedir?

- Programların işletim sistemi hizmetlerine (dosya, bellek, süreç, zaman, ağ…) erişmek için kullandığı **tanımlı arayüz**.
- Uygulama tarafından *istenir*, çekirdek tarafından *çekirdek modunda yürütülür*.
- Programcı çoğunlukla çağrıları doğrudan değil bir **API** üzerinden kullanır: POSIX (glibc, musl), Windows API, Java API. API taşınabilirlik ve kolaylık sağlar.
- Çağrı kümesi sistemden sisteme değişir: Linux x86-64'te yaklaşık 370 çağrı vardır. Temel kavramlar ise ortaktır.
- `strace` (Linux), `dtruss` (macOS) ve Process Monitor (Windows) bir programın yaptığı çağrıları gösterir.

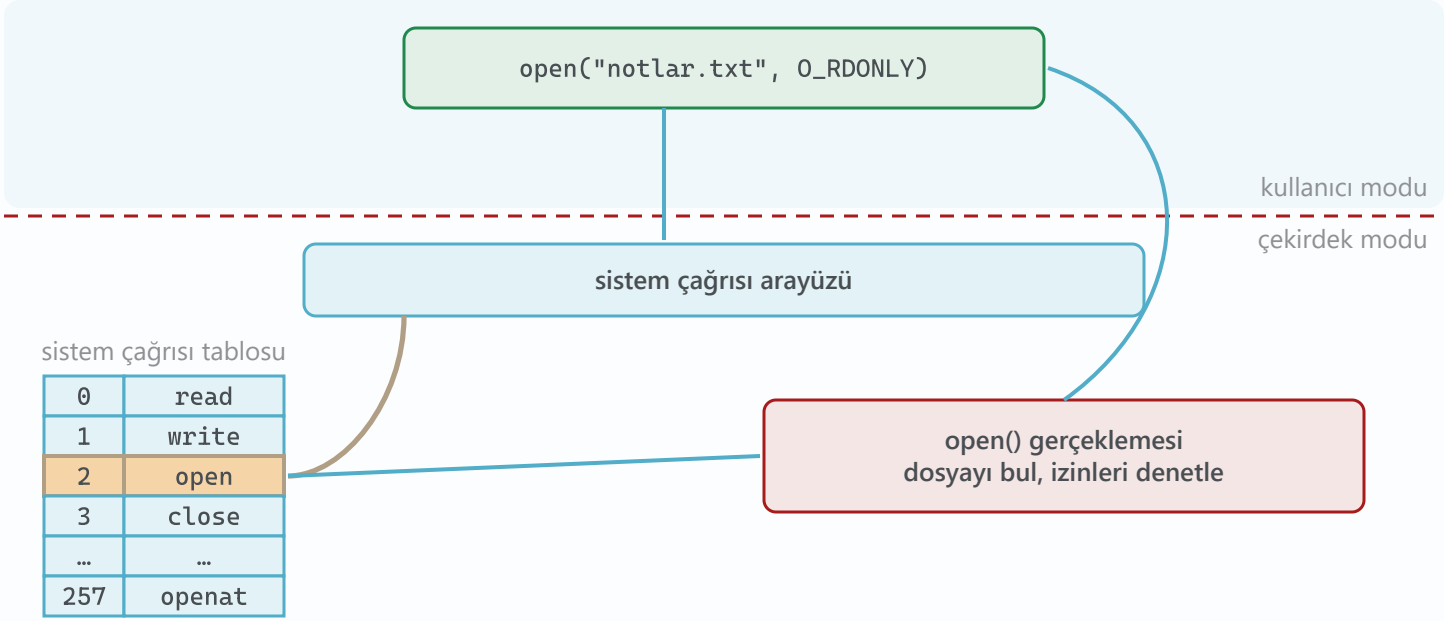


API → Sistem Çağrısı Arayüzü → Çekirdek



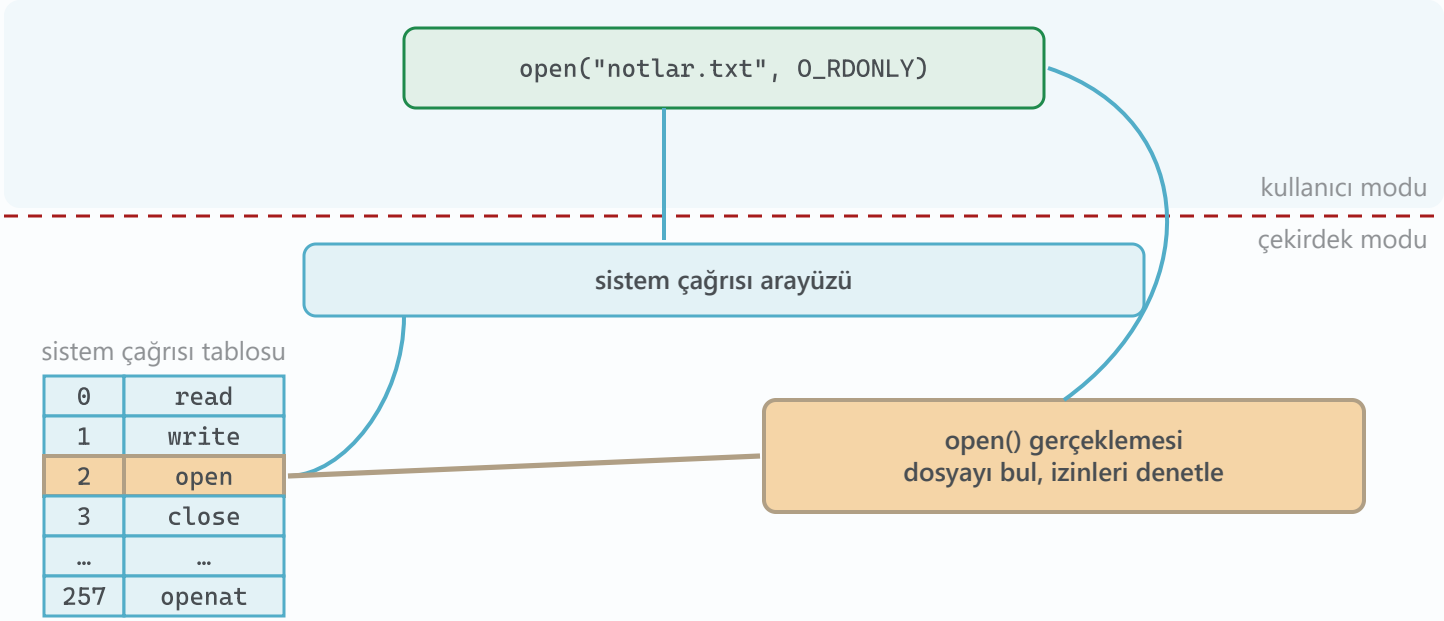
Uygulama libc'deki `open()` sarmalayıcısını çağırır; sarmalayıcı çağrı numarasını yazmaca koyar ve tuzak komutunu çalıştırır.

API → Sistem Çağrısı Arayüzü → Çekirdek



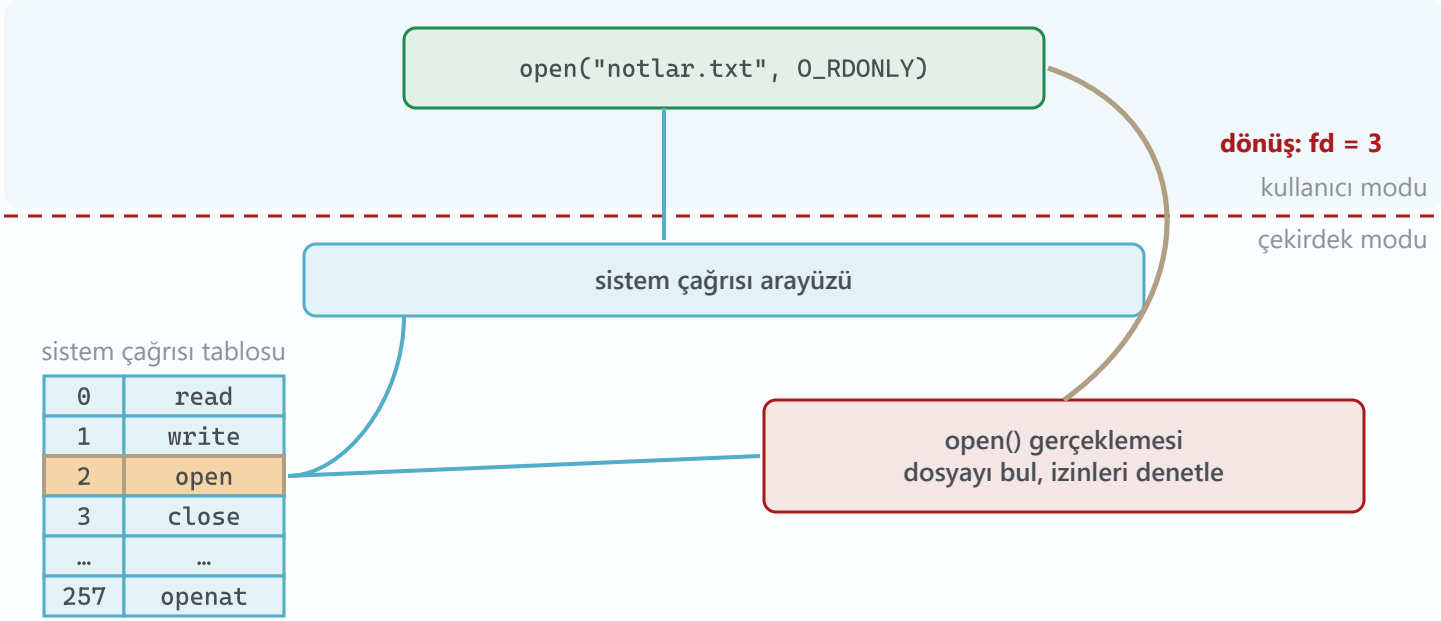
Çekirdekteki sistem çağrısı arayüzü, numarayı **sistem çağrısı tablosunda** indis olarak kullanır (x86-64 Linux: open = 2).

API → Sistem Çağrısı Arayüzü → Çekirdek



Tablodaki işaretçi çekirdekteki gerçeklemeye götürür: yol çözülür, izinler denetlenir, açık dosya kaydı oluşturulur.

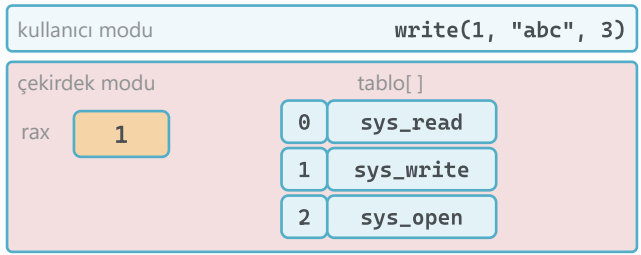
API → Sistem Çağrısı Arayüzü → Çekirdek



Sonuç (ilk boş tanımlayıcı: 0, 1, 2 dolu olduğundan 3) kullanıcıya döner. Not: güncel glibc `open()`'u aslında `openat` (257) ile gerçekleştirir.

Kod İzleme: Sistem Çağrısı Tablosu

```
1 typedef long (*cagri_t)(long, long, long);
2 cagri_t tablo[3] = {sys_read, sys_write,
3                     sys_open};
▶ long syscall_giris(long no, long a,
▶                    long b, long c) {
6   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* geçersiz numara */
8   return tablo[no](a, b, c);
9 }
```



libc sarmalayıcısı çağrı numarasını `rax`'a koyar ve `syscall` çalıştırır. Donanım çekirdek moduna geçip **tek giriş noktasına** atlar: **no = 1** (Linux x86-64'te write).

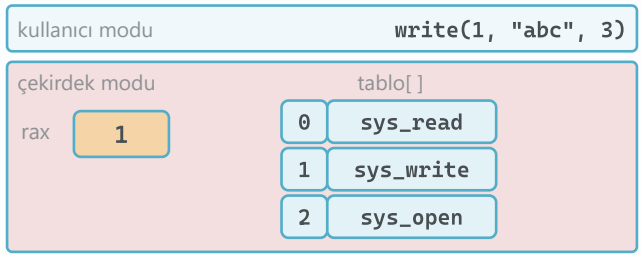
Adım 1 / 9

DEĞİŞKENLER

`no = 1`

Kod İzleme: Sistem Çağrısı Tablosu

```
1 typedef long (*cagri_t)(long, long, long);
2 cagri_t tablo[3] = {sys_read, sys_write,
3                     sys_open};
4 long syscall_giris(long no, long a,
5                     long b, long c) {
▶   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* geçersiz numara */
8   return tablo[no](a, b, c);
9 }
```



0 ≤ 1 < 3: numara geçerli.

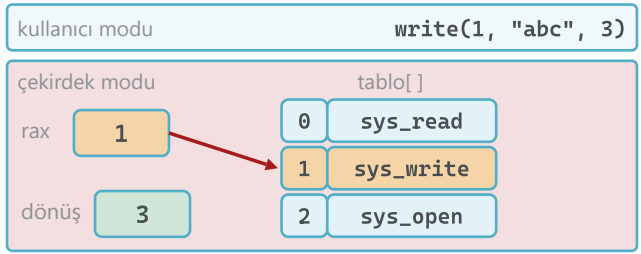
Adım 2 / 9

DEĞİŞKENLER

`no = 1`

Kod İzleme: Sistem Çağrısı Tablosu

```
1 typedef long (*cagri_t)(long, long, long);
2 cagri_t tablo[3] = {sys_read, sys_write,
3                     sys_open};
4 long syscall_giris(long no, long a,
5                     long b, long c) {
6   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* geçersiz numara */
▶   return tablo[no](a, b, c);
9 }
```



`tablo[1]` bir fonksiyon işaretçisi: **sys_write** çağrılır, yazılan bayt sayısı olarak **3** döner.

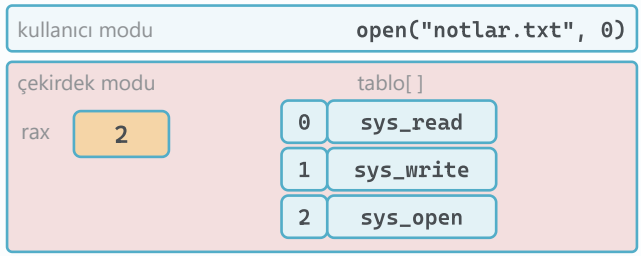
Adım 3 / 9

DEĞİŞKENLER

`no = 1` `dönüş = 3`

Kod İzleme: Sistem Çağrısı Tablosu

```
1 typedef long (*cagri_t)(long, long, long);
2 cagri_t tablo[3] = {sys_read, sys_write,
3                     sys_open};
▶ long syscall_giris(long no, long a,
▶                    long b, long c) {
6   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* geçersiz numara */
8   return tablo[no](a, b, c);
9 }
```



Yeni çağrı: `open("notlar.txt", 0)`, **no = 2**.

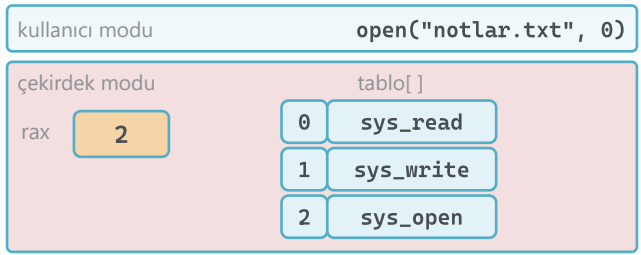
Adım 4 / 9

DEĞİŞKENLER

`no = 2`

Kod İzleme: Sistem Çağrısı Tablosu

```
1 typedef long (*cagri_t)(long, long, long);
2 cagri_t tablo[3] = {sys_read, sys_write,
3                     sys_open};
4 long syscall_giris(long no, long a,
5                     long b, long c) {
▶   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* geçersiz numara */
8   return tablo[no](a, b, c);
9 }
```



0 ≤ 2 < 3: numara geçerli.

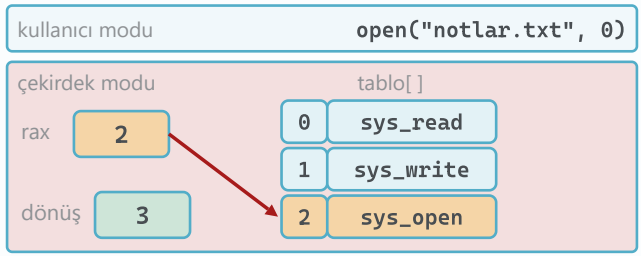
Adım 5 / 9

DEĞİŞKENLER

`no = 2`

Kod İzleme: Sistem Çağrısı Tablosu

```
1 typedef long (*cagri_t)(long, long, long);
2 cagri_t tablo[3] = {sys_read, sys_write,
3                     sys_open};
4 long syscall_giris(long no, long a,
5                     long b, long c) {
6   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* geçersiz numara */
▶   return tablo[no](a, b, c);
9 }
```



`tablo[2]` bir fonksiyon işaretçisi: **sys_open** çağrılır, yeni tanımlayıcı olarak **3** döner.

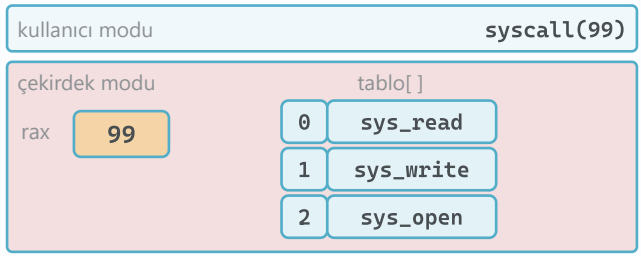
Adım 6 / 9

DEĞİŞKENLER

`no = 2` `dönüş = 3`

Kod İzleme: Sistem Çağrısı Tablosu

```
1 typedef long (*cagri_t)(long, long, long);
2 cagri_t tablo[3] = {sys_read, sys_write,
3                     sys_open};
▶ long syscall_giris(long no, long a,
▶                    long b, long c) {
6   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* geçersiz numara */
8   return tablo[no](a, b, c);
9 }
```



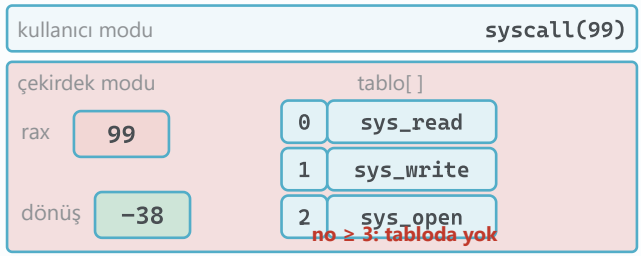
Yeni çağrı: `syscall(99)`, **no = 99**.

DEĞİŞKENLER

`no = 99`

Kod İzleme: Sistem Çağrısı Tablosu

```
1 typedef long (*cagri_t)(long, long, long);
2 cagri_t tablo[3] = {sys_read, sys_write,
3                     sys_open};
4 long syscall_giris(long no, long a,
5                     long b, long c) {
▶   if (no < 0 || no >= 3)
▶     return -ENOSYS;     /* geçersiz numara */
8   return tablo[no](a, b, c);
9 }
```



Numara tablonun dışında. Denetim olmasaydı `tablo[99]` çekirdekte rastgele bir adrese atlamak olurdu. Çağrı **−ENOSYS (−38)** ile reddedilir.

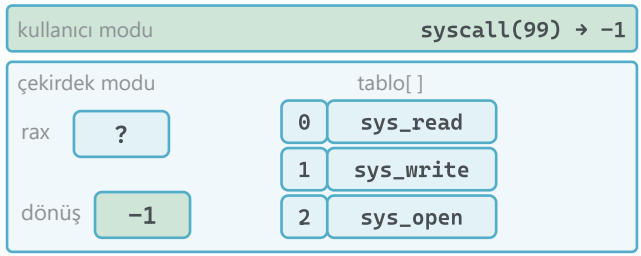
Adım 8 / 9

DEĞİŞKENLER

`no = 99` `dönüş = −38`

Kod İzleme: Sistem Çağrısı Tablosu

```
1 typedef long (*cagri_t)(long, long, long);
2 cagri_t tablo[3] = {sys_read, sys_write,
3                     sys_open};
4 long syscall_giris(long no, long a,
5                     long b, long c) {
6   if (no < 0 || no >= 3)
▶     return -ENOSYS;     /* geçersiz numara */
8   return tablo[no](a, b, c);
9 }
```



libc negatif değeri `errno = ENOSYS` yapar ve programa **−1** döndürür. Çekirdek, kullanıcıdan gelen her değeri **doğrulamadan** kullanmaz.

Adım 9 / 9

DEĞİŞKENLER

`no = 99` `dönüş = −1` `errno = ENOSYS`

Örnek: Bir Dosyayı Başka Dosyaya Kopyalamak

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

Kullanıcıdan ad almak bile iki çağrı ister: ekrana istem yaz (`write`), klavyeden oku (`read`).

Örnek: Bir Dosyayı Başka Dosyaya Kopyalamak

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

Aynı işlem çıktı dosyası için tekrarlanır.

Örnek: Bir Dosyayı Başka Dosyaya Kopyalamak

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

`open` başarısız olursa (dosya yok, izin yok) −1 döner; program hata iletisi yazıp çıkar.

Örnek: Bir Dosyayı Başka Dosyaya Kopyalamak

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

`open(…, O_WRONLY | O_CREAT | O_EXCL)` : dosya zaten varsa hata verir.

Örnek: Bir Dosyayı Başka Dosyaya Kopyalamak

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

Asıl iş: `read` 0 (dosya sonu) dönene kadar okunan baytlar `write` ile yazılır.

Örnek: Bir Dosyayı Başka Dosyaya Kopyalamak

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

Her açık dosya `close` ile kapatılır.

Örnek: Bir Dosyayı Başka Dosyaya Kopyalamak

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

Kullanıcıya bilgi verilir.

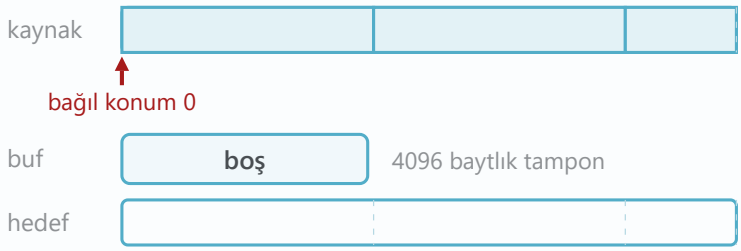
Örnek: Bir Dosyayı Başka Dosyaya Kopyalamak

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

`exit` ile süreç sonlanır. Böyle basit bir program bile onlarca çağrı yapar; `strace -c cp a b` ile sayılabilir.

Kod İzleme: `read`/`write` ile Kopyalama

```
1  char buf[4096];
▶ 2  int in = open("kaynak", O_RDONLY);
3  int out = creat("hedef", 0644);
4  ssize_t n;  long toplam = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    toplam += n;
8  }
9  close(in);  close(out);
10 printf("%ld bayt\n", toplam);
```



çağrılar: open×1

"kaynak" 10.000 baytlık bir dosya. `open` ilk boş tanımlayıcıyı döndürür: 0, 1, 2 dolu olduğundan **in = 3**; bağıl konum 0.

Adım 1 / 12

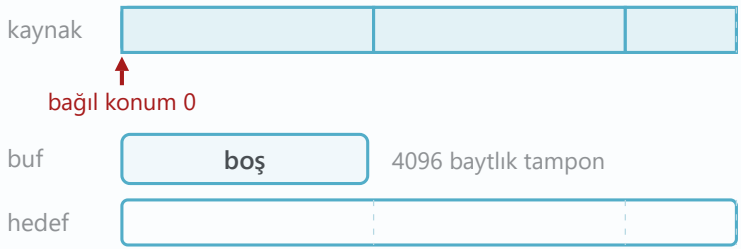
DEĞİŞKENLER

`in = 3` `out = ?` `n = ?` `toplam = 0`

ÇIKTI

Kod İzleme: `read`/`write` ile Kopyalama

```
1  char buf[4096];
2  int in = open("kaynak", O_RDONLY);
▶  int out = creat("hedef", 0644);
4  ssize_t n;  long toplam = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    toplam += n;
8  }
9  close(in);  close(out);
10 printf("%ld bayt\n", toplam);
```



çağrılar: open×1 creat×1

Boş hedef dosya oluşturulur: **out = 4**.

Adım 2 / 12

DEĞİŞKENLER

`in = 3` `out = 4` `n = ?` `toplam = 0`

ÇIKTI

Kod İzleme: `read`/`write` ile Kopyalama

```
1  char buf[4096];
2  int in = open("kaynak", O_RDONLY);
3  int out = creat("hedef", 0644);
4  ssize_t n;  long toplam = 0;
▶ while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    toplam += n;
8  }
9  close(in);  close(out);
10 printf("%ld bayt\n", toplam);
```



çağrılar: open×1 creat×1 read×1

`read` çekirdeğe geçer, en çok 4096 bayt ister: **n = 4096**. Çekirdek dosya bağıl konumunu 4096 ilerletir.

Adım 3 / 12

DEĞİŞKENLER

`in = 3` `out = 4` `n = 4096` `toplam = 0`

ÇIKTI

Kod İzleme: `read`/`write` ile Kopyalama

```
1  char buf[4096];
2  int in = open("kaynak", O_RDONLY);
3  int out = creat("hedef", 0644);
4  ssize_t n;  long toplam = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
▶    write(out, buf, n);
7    toplam += n;
8  }
9  close(in);  close(out);
10 printf("%ld bayt\n", toplam);
```



`çağrılar: open×1 creat×1 read×1 write×1`

Tampondaki **n** bayt (4096 değil, okunan kadar) hedefe yazılır.

Adım 4 / 12

DEĞİŞKENLER

`in = 3` `out = 4` `n = 4096` `toplam = 0`

ÇIKTI

Kod İzleme: `read`/`write` ile Kopyalama

```
1  char buf[4096];
2  int in = open("kaynak", O_RDONLY);
3  int out = creat("hedef", 0644);
4  ssize_t n;  long toplam = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
▶    toplam += n;
8  }
9  close(in);  close(out);
10 printf("%ld bayt\n", toplam);
```



çağrılar: open×1 creat×1 read×1 write×1

toplam = 4096. Her tur iki sistem çağrısı: bir `read`, bir `write`.

Adım 5 / 12

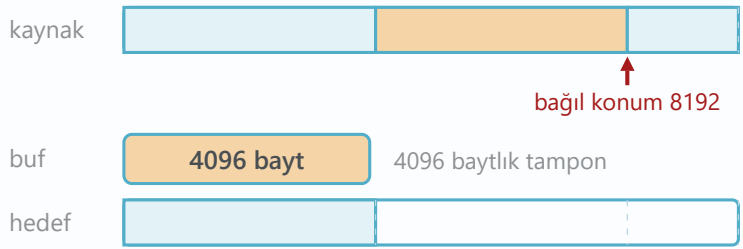
DEĞİŞKENLER

`in = 3` `out = 4` `n = 4096` `toplam = 4096`

ÇIKTI

Kod İzleme: `read`/`write` ile Kopyalama

```
1  char buf[4096];
2  int in = open("kaynak", O_RDONLY);
3  int out = creat("hedef", 0644);
4  ssize_t n;  long toplam = 0;
▶ while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    toplam += n;
8  }
9  close(in);  close(out);
10 printf("%ld bayt\n", toplam);
```



çağrılar: open×1 creat×1 read×2 write×1

Yeniden `read` : **n = 4096**, bağıl konum 8192.

Adım 6 / 12

DEĞİŞKENLER

`in = 3` `out = 4` `n = 4096` `toplam = 4096`

ÇIKTI

Kod İzleme: `read`/`write` ile Kopyalama

```
1  char buf[4096];
2  int in = open("kaynak", O_RDONLY);
3  int out = creat("hedef", 0644);
4  ssize_t n;  long toplam = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
▶    write(out, buf, n);
▶    toplam += n;
8  }
9  close(in);  close(out);
10 printf("%ld bayt\n", toplam);
```



çağrılar: open×1 creat×1 read×2 write×2

4096 bayt yazılır, **toplam = 8192**.

Adım 7 / 12

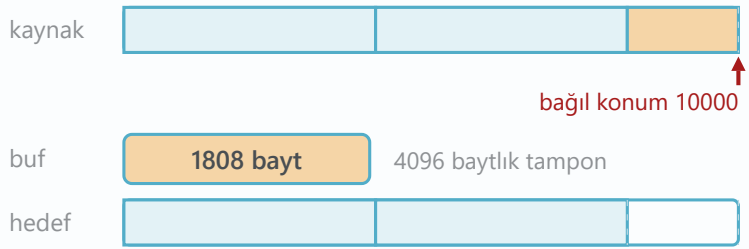
DEĞİŞKENLER

`in = 3` `out = 4` `n = 4096` `toplam = 8192`

ÇIKTI

Kod İzleme: `read`/`write` ile Kopyalama

```
1  char buf[4096];
2  int in = open("kaynak", O_RDONLY);
3  int out = creat("hedef", 0644);
4  ssize_t n;  long toplam = 0;
▶ while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    toplam += n;
8  }
9  close(in);  close(out);
10 printf("%ld bayt\n", toplam);
```



çağrılar: open×1 creat×1 read×3 write×2

Dosyada yalnızca 10.000 − 8192 = 1808 bayt kaldı: `read` istenenden az, **n = 1808** döndürür. Bu hata değildir.

Adım 8 / 12

DEĞİŞKENLER

`in = 3` `out = 4` `n = 1808` `toplam = 8192`

ÇIKTI

Kod İzleme: `read`/`write` ile Kopyalama

```
1  char buf[4096];
2  int in = open("kaynak", O_RDONLY);
3  int out = creat("hedef", 0644);
4  ssize_t n;  long toplam = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
▶    write(out, buf, n);
▶    toplam += n;
8  }
9  close(in);  close(out);
10 printf("%ld bayt\n", toplam);
```



`çağrılar: open×1 creat×1 read×3 write×3`

1808 bayt yazılır, **toplam = 10000**.

Adım 9 / 12

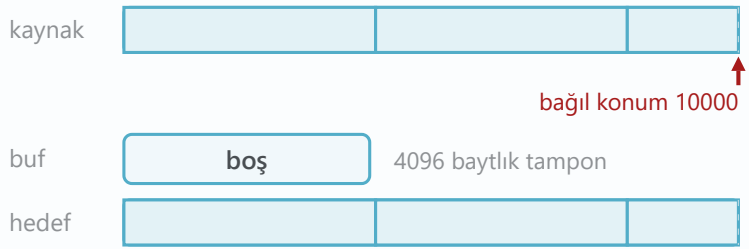
DEĞİŞKENLER

`in = 3` `out = 4` `n = 1808` `toplam = 10000`

ÇIKTI

Kod İzleme: `read`/`write` ile Kopyalama

```
1  char buf[4096];
2  int in = open("kaynak", O_RDONLY);
3  int out = creat("hedef", 0644);
4  ssize_t n;  long toplam = 0;
▶ while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    toplam += n;
8  }
9  close(in);  close(out);
10 printf("%ld bayt\n", toplam);
```



çağrılar: open×1 creat×1 read×4 write×3

Bağıl konum dosya sonunda: `read` **0** döndürür (dosya sonu). Koşul yanlış, döngü biter.

Adım 10 / 12

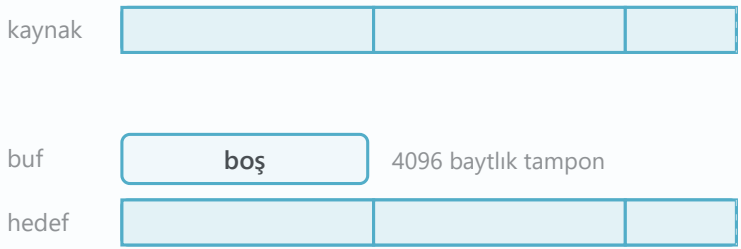
DEĞİŞKENLER

`in = 3` `out = 4` `n = 0` `toplam = 10000`

ÇIKTI

Kod İzleme: `read`/`write` ile Kopyalama

```
1  char buf[4096];
2  int in = open("kaynak", O_RDONLY);
3  int out = creat("hedef", 0644);
4  ssize_t n;  long toplam = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    toplam += n;
8  }
▶  close(in);  close(out);
10 printf("%ld bayt\n", toplam);
```



çağrılar: open×1 creat×1 read×4 write×3 close×2

İki tanımlayıcı da kapatılır; 3 ve 4 yeniden kullanılabilir.

Adım 11 / 12

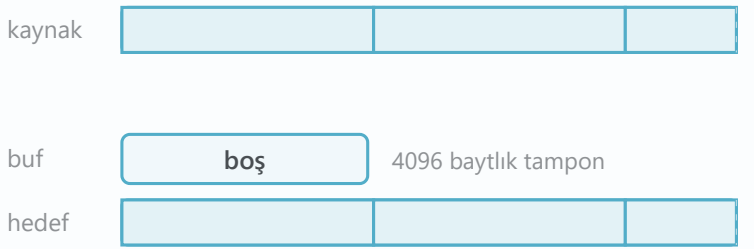
DEĞİŞKENLER

`in = 3` `out = 4` `n = 0` `toplam = 10000`

ÇIKTI

Kod İzleme: `read`/`write` ile Kopyalama

```
1 char buf[4096];
2 int in = open("kaynak", O_RDONLY);
3 int out = creat("hedef", 0644);
4 ssize_t n;  long toplam = 0;
5 while ((n = read(in, buf, 4096)) > 0) {
6   write(out, buf, n);
7   toplam += n;
8 }
9 close(in);  close(out);
▶ printf("%ld bayt\n", toplam);
```



`çağrılar: open×1 creat×1 read×4 write×3 close×2 +write`

Çıktı **10000 bayt**. Toplam 12 sistem çağrısı (printf de bir `write` yapar). Tampon 1 bayt olsaydı yalnızca okuma ve yazma için yaklaşık 20.000 çağrı gerekirdi.

Adım 12 / 12

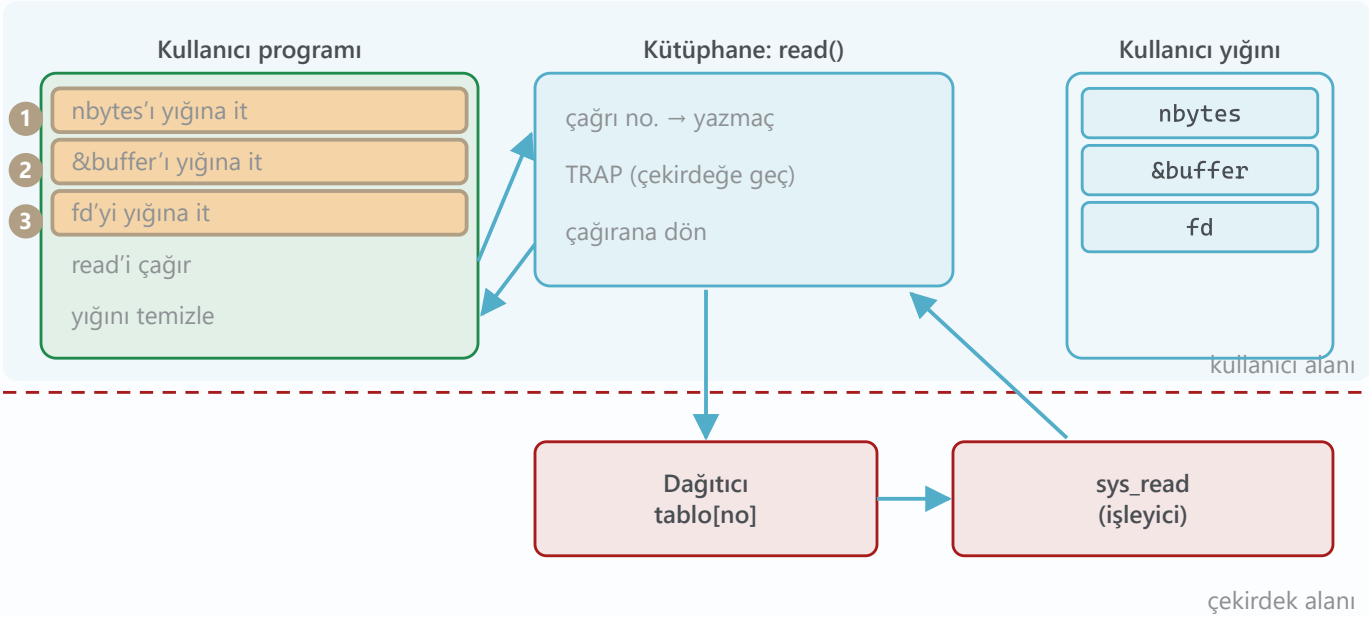
DEĞİŞKENLER

`in = 3` `out = 4` `n = 0` `toplam = 10000`

ÇIKTI

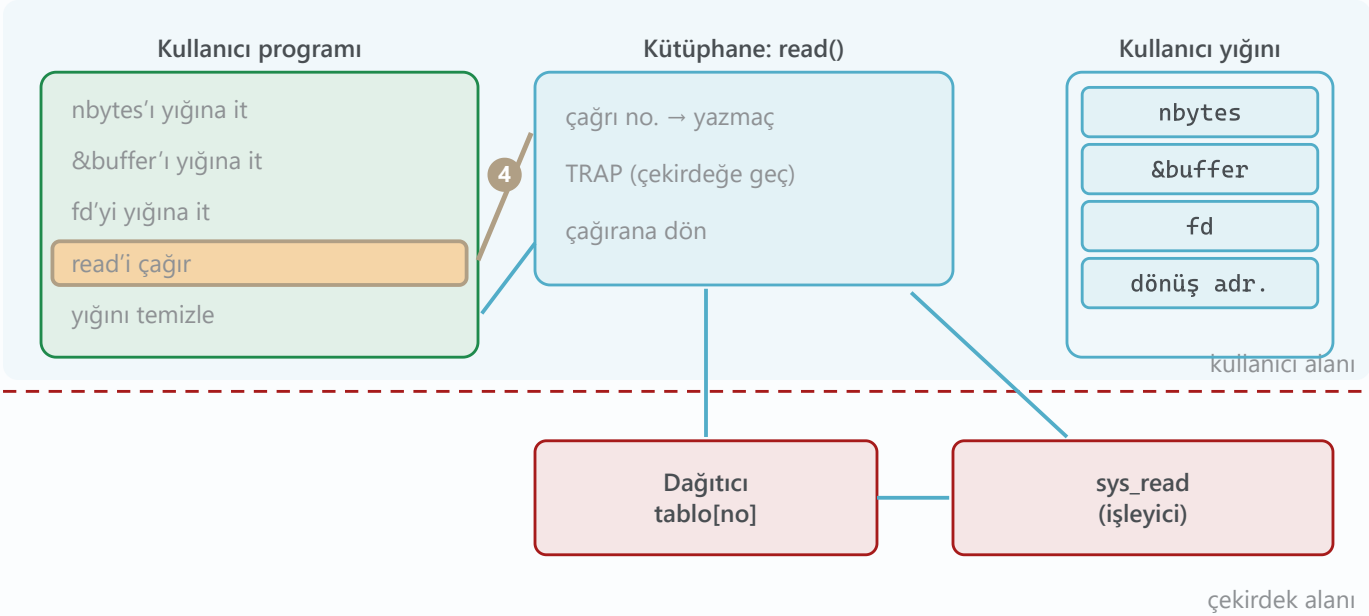
```
10000 bayt
```

`read(fd, buffer, nbytes)` Çağrısının 11 Adımı



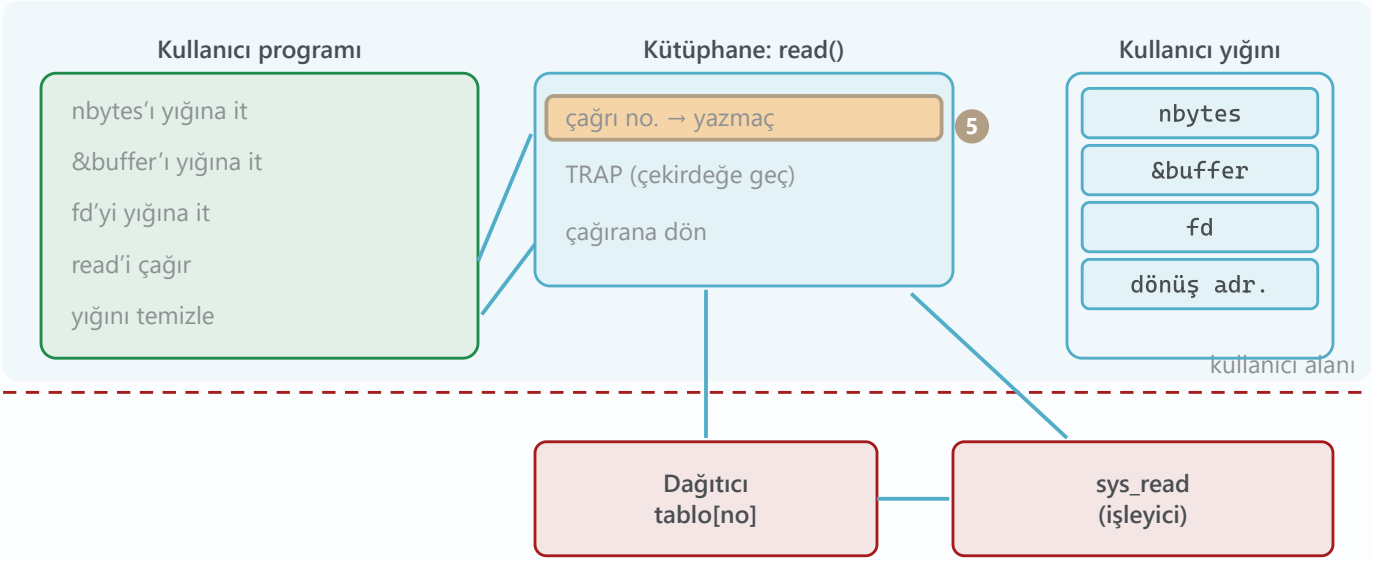
1–3: Parametreler yığına itilir (Tanenbaum'un örneğindeki tarihsel 32-bit x86 kuralı). x86-64'te ilk altı parametre yazmaçlarla aktarılır.

`read(fd, buffer, nbytes)` Çağrısının 11 Adımı



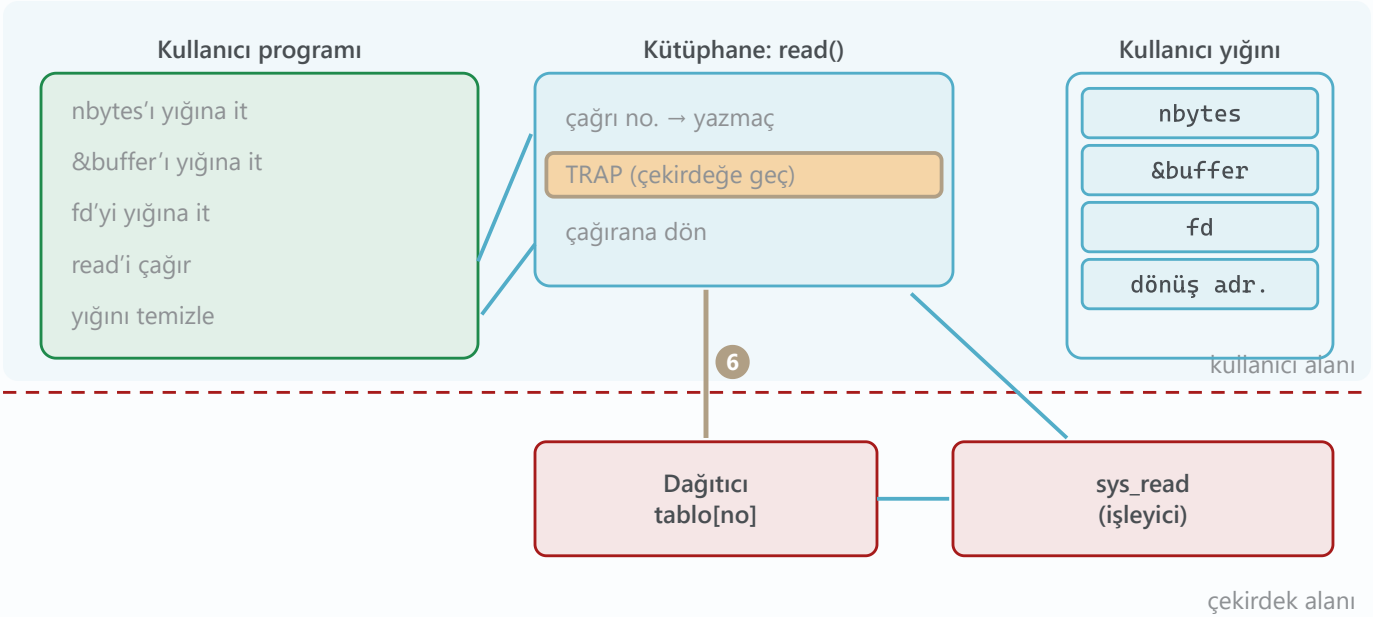
4: Kütüphanedeki `read()` sıradan bir fonksiyon gibi çağrılır; dönüş adresi yığına girer.

`read(fd, buffer, nbytes)` Çağrısının 11 Adımı



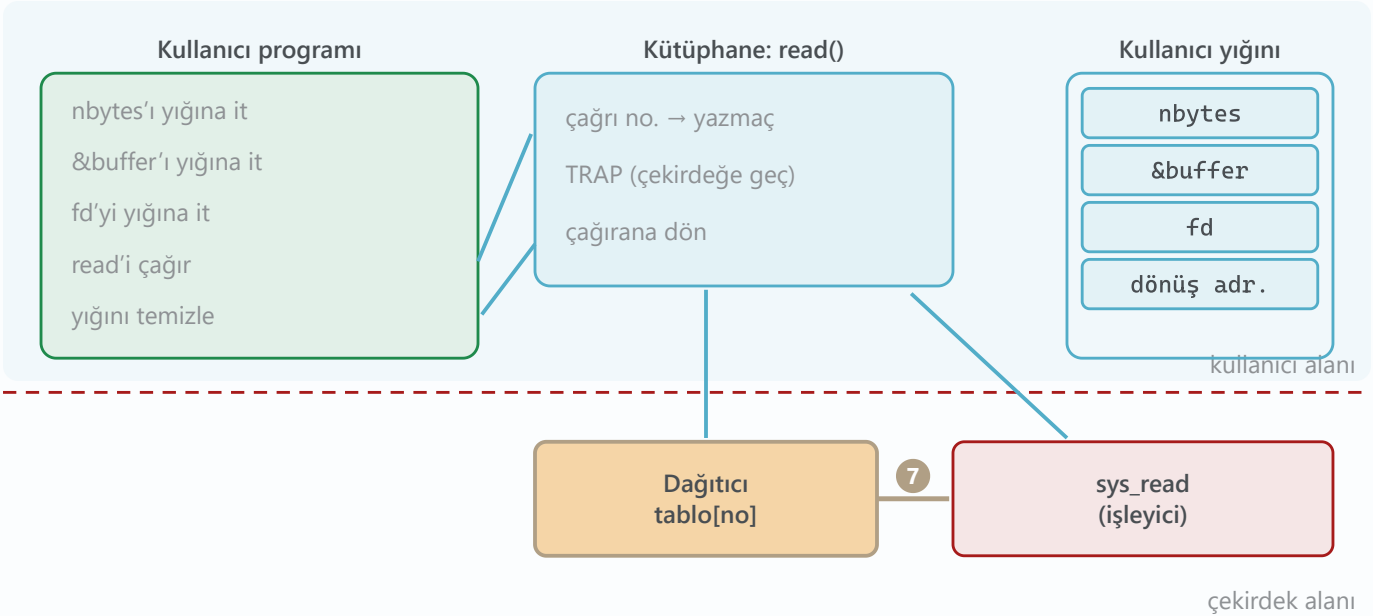
5: Kütüphane çağrı numarasını çekirdeğin beklediği yazmaca koyar (Linux x86-64: `rax = 0`).

`read(fd, buffer, nbytes)` Çağrısının 11 Adımı



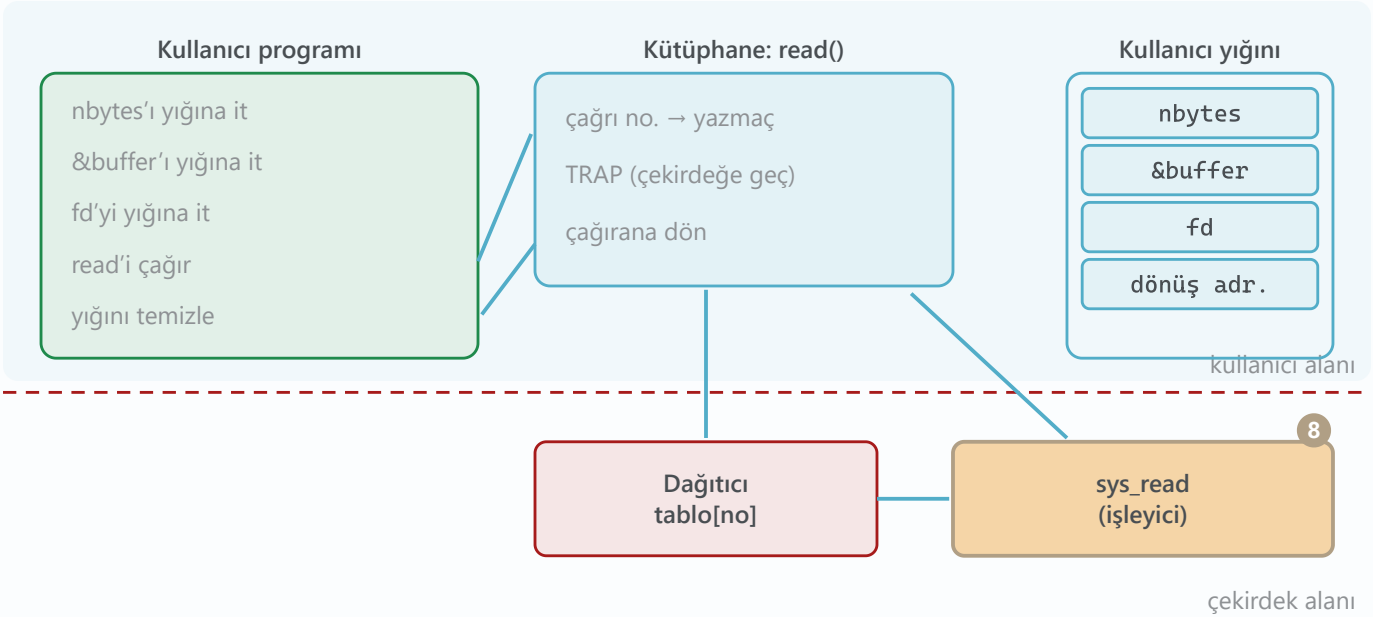
6: Tuzak komutu (x86-64: `syscall`) çalışır; CPU çekirdek moduna geçip çekirdekteki sabit adrese atlar.

`read(fd, buffer, nbytes)` Çağrısının 11 Adımı



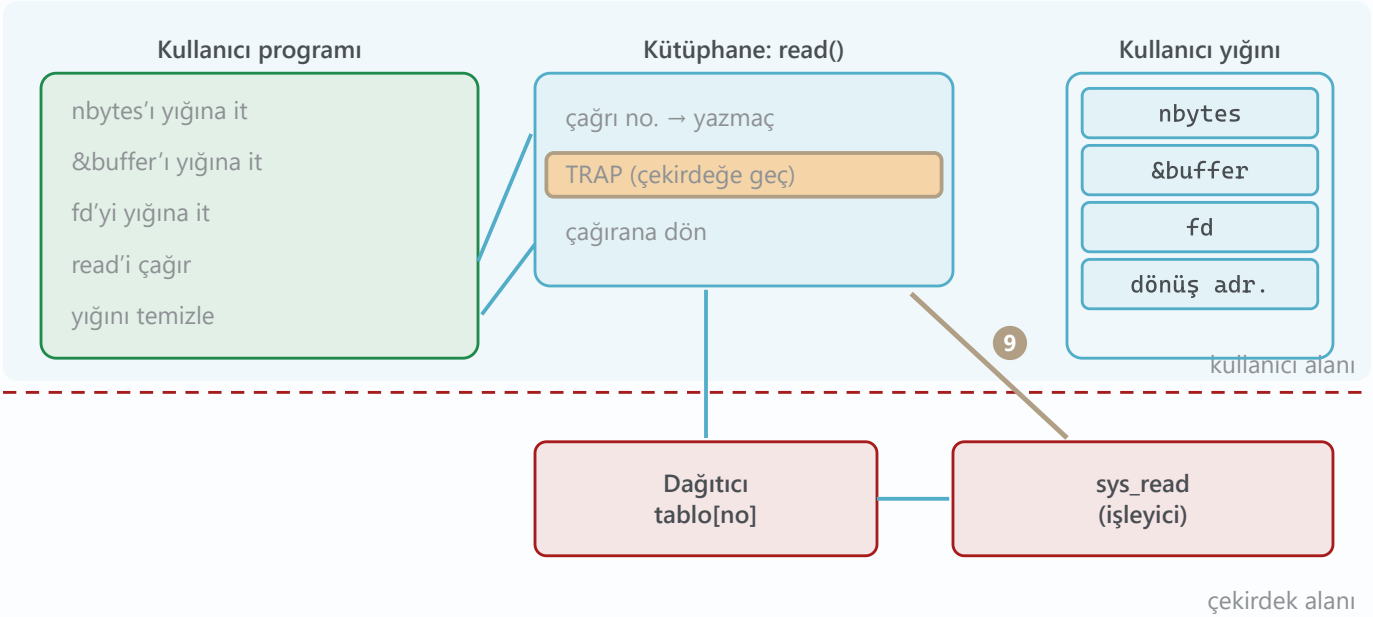
7: Çekirdek çağrı numarasına bakar ve sistem çağrısı tablosu üzerinden doğru işleyiciye dallanır.

`read(fd, buffer, nbytes)` Çağrısının 11 Adımı



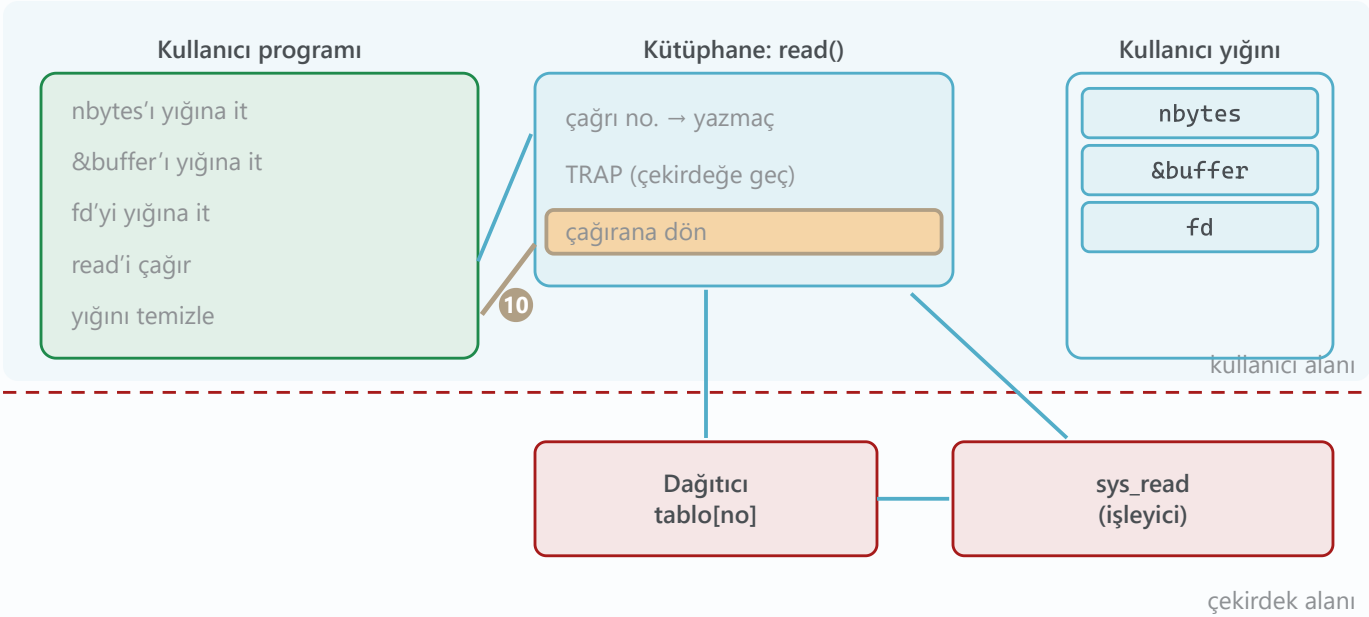
8: İşleyici fd'den buffer'a en çok nbytes bayt kopyalar. Veri hazır değilse süreç bloke olur, CPU başka sürece verilir.

`read(fd, buffer, nbytes)` Çağrısının 11 Adımı



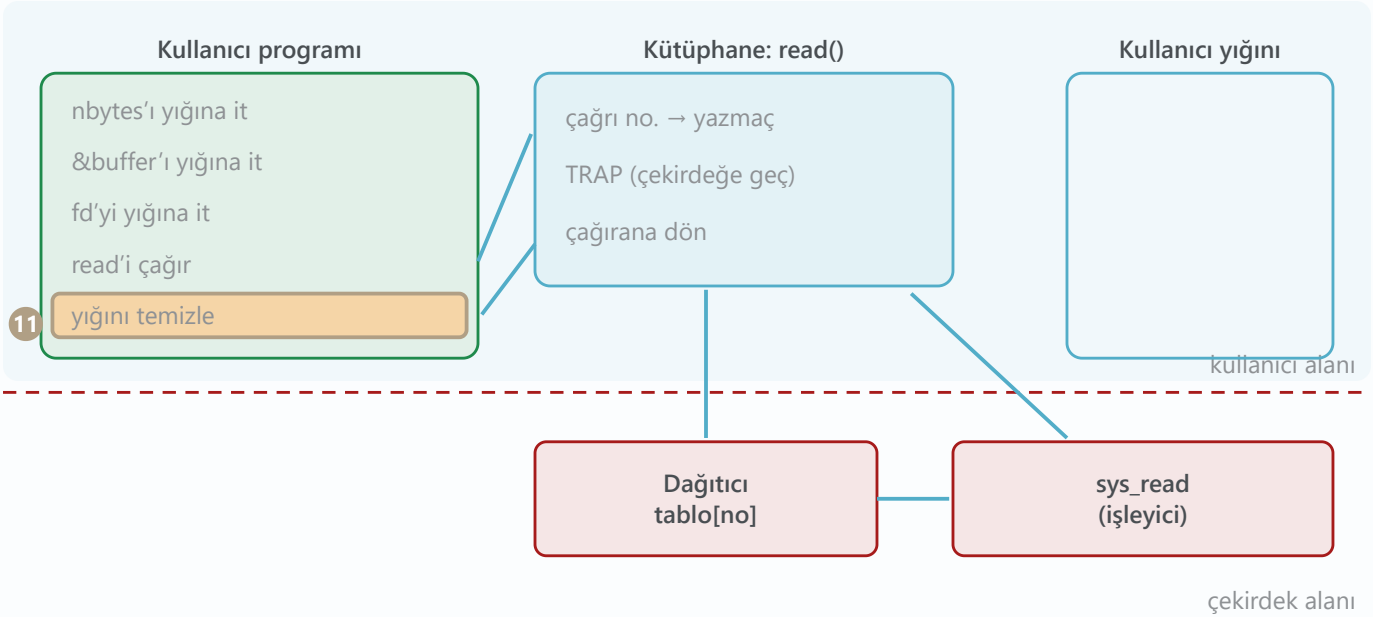
9: İş bitince kontrol kullanıcı moduna, tuzaktan sonraki komuta döner; sonuç (okunan bayt sayısı) yazmaçtadır.

`read(fd, buffer, nbytes)` Çağrısının 11 Adımı



10: Kütüphanedeki `read()` kullanıcı programına döner.

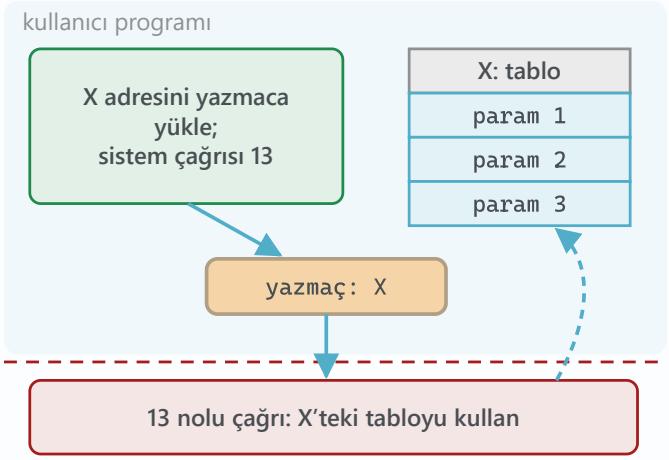
`read(fd, buffer, nbytes)` Çağrısının 11 Adımı



11: Program parametreleri yığından temizler (SP artırılır) ve sonraki komutla devam eder.

Sistem Çağrılarında Parametre Aktarımı

1. **Yazmaçlar:** en hızlı yol; Linux en çok 6 parametreyi böyle aktarır.
2. **Bellekte blok / tablo:** parametreler bellekte, adresi yazmaçta (şekil). Örn. `clone3(struct clone_args *)`.
3. **Yığın:** program parametreleri iter, işletim sistemi çeker.



x86-64 Linux	Görev
`rax`	çağrı numarası; dönüşte sonuç
`rdi, rsi, rdx`	1., 2., 3. parametre
`r10, r8, r9`	4., 5., 6. parametre (C kuralındaki `rcx` yerine `r10`)
`syscall`	çekirdeğe geçiş; `rcx` ve `r11` bozulur
ARM64	`x8` = numara, `x0–x5` parametre, `svc #0`

Hata durumunda çekirdek −1…−4095 arası bir değer (−errno) döndürür; libc bunu `errno`'ya yazar ve çağırana −1 verir.

Kod İzleme: `write()` ile Ekrana Yazmak

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
▶      const char *msg = "Merhaba, Dünya!\n";
6      size_t len = strlen(msg);        // 17 bayt
7      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // hata
10     return 0;
11 }
```

DEĞİŞKENLER

`msg = "Merhaba, Dünya!\n"`

`len = ?` `n = ?`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

`msg`, salt okunur veri kesimindeki dizgiyi gösterir.

Kod İzleme: `write()` ile Ekrana Yazmak

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Merhaba, Dünya!\n";
▶      size_t len = strlen(msg);        // 17 bayt
7      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // hata
10     return 0;
11 }
```

DEĞİŞKENLER

`msg = "Merhaba, Dünya!\n"`

`len = 17` `n = ?`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

“Merhaba, ” 9 + “Dünya” 6 (ü, UTF-8’de 2 bayt) + “!” ve “\n” 2 = **17** bayt. Sabit 15 yazılsaydı çıktı “Merhaba, Dünya” olarak kesilirdi.

Kod İzleme: `write()` ile Ekrana Yazmak

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Merhaba, Dünya!\n";
6      size_t len = strlen(msg);        // 17 bayt
▶      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // hata
10     return 0;
11 }
```

DEĞİŞKENLER

`msg = "Merhaba, Dünya!\n"`

`len = 17` `n = ?`

ÇAĞRI YIĞINI

`write(1, msg, 17) [libc]`

`main()`

ÇIKTI

libc sarmalayıcısı: `rax=1` (write), `rdi=1`, `rsi=msg`, `rdx=17`; ardından `syscall`.

Kod İzleme: `write()` ile Ekrana Yazmak

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Merhaba, Dünya!\n";
6      size_t len = strlen(msg);        // 17 bayt
▶      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // hata
10     return 0;
11 }
```

DEĞİŞKENLER

`msg = "Merhaba, Dünya!\n"`

`len = 17` `n = ?`

ÇAĞRI YIĞINI

- `sys_write [çekirdek]`
- `write [libc]`
- `main()`

ÇIKTI

```
Merhaba, Dünya!
```

Çekirdek modunda `sys_write`, 17 baytı fd 1'in bağlı olduğu terminale aktarır.

Kod İzleme: `write()` ile Ekrana Yazmak

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Merhaba, Dünya!\n";
6      size_t len = strlen(msg);        // 17 bayt
▶      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // hata
10     return 0;
11 }
```

DEĞİŞKENLER

`msg = "Merhaba, Dünya!\n"`

`len = 17` `n = 17`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
Merhaba, Dünya!
```

Kullanıcı moduna dönüş: `n = 17`, yazılan bayt sayısı. Daha küçük olabilir (kısmi yazma).

Kod İzleme: `write()` ile Ekrana Yazmak

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Merhaba, Dünya!\n";
6      size_t len = strlen(msg);        // 17 bayt
7      ssize_t n = write(1, msg, len);  // 1: stdout
▶      if (n == -1)
9          return 1;                    // hata
10     return 0;
11 }
```

DEĞİŞKENLER

`msg = "Merhaba, Dünya!\n"`

`len = 17` `n = 17`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
Merhaba, Dünya!
```

`n` −1 değil, hata yok. Hata olsaydı −1 döner ve `errno` ayarlanırdı.

Kod İzleme: `write()` ile Ekrana Yazmak

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Merhaba, Dünya!\n";
6      size_t len = strlen(msg);        // 17 bayt
7      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // hata
▶      return 0;
11 }
```

DEĞİŞKENLER

`msg = "Merhaba, Dünya!\n"`

`len = 17` `n = 17`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
Merhaba, Dünya!
```

`main` 0 döndürür; libc `exit_group(0)` çağrısıyla süreci sonlandırır.

Kod İzleme: `read()` ve Bloke Olma

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
▶      char buf[100];
6      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("okuma hatası");
9          return 1;
10     }
11     write(1, buf, n);    // tam n bayt
12     return 0;
13 }
```

DEĞİŞKENLER

`buf` = **[100 bayt, ilklenmemiş]**

`n` = **?**

ÇAĞRI YIĞINI

main()

ÇIKTI

`buf` yığında ayrılır; içeriği ilklenmemiştir.

Kod İzleme: `read()` ve Bloke Olma

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
▶      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("okuma hatası");
9          return 1;
10     }
11     write(1, buf, n);    // tam n bayt
12     return 0;
13 }
```

DEĞİŞKENLER

`buf = [100 bayt, ilklenmemiş]`

`n = ?`

ÇAĞRI YIĞINI

`sys_read [çekirdek]`

`read(0, buf, 100)`

`main()`

ÇIKTI

fd 0 = stdin (klavye). Henüz satır yok: süreç **bloke** olur, çizelgeleyici başka süreci çalıştırır.

Kod İzleme: `read()` ve Bloke Olma

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
▶      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("okuma hatası");
9          return 1;
10     }
11     write(1, buf, n);    // tam n bayt
12     return 0;
13 }
```

DEĞİŞKENLER

- `buf = [100 bayt, ilklenmemiş]`
- `n = ?`

ÇAĞRI YIĞINI

- `sys_read [çekirdek]`
- `read(0, buf, 100)`
- `main()`

ÇIKTI

```
selam
```

Kullanıcı “selam” yazıp Enter’a basar; terminal sürücüsü karakterleri ekrana yansıtır ve satırı tamponlar.

Kod İzleme: `read()` ve Bloke Olma

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
▶      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("okuma hatası");
9          return 1;
10     }
11     write(1, buf, n);    // tam n bayt
12     return 0;
13 }
```

DEĞİŞKENLER

`buf = "selam\n…" (6 bayt)`

`n = 6`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
selam
```

Satır tamamlanınca 6 bayt `buf`'a kopyalanır, süreç uyanır, read 6 döndürür. Sona `'\0'` eklenmez.

Kod İzleme: `read()` ve Bloke Olma

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
6      ssize_t n = read(0, buf, sizeof(buf));
▶      if (n == -1) {
8          perror("okuma hatası");
9          return 1;
10     }
11     write(1, buf, n);    // tam n bayt
12     return 0;
13 }
```

DEĞİŞKENLER

`buf` = **"selam\n…" (6 bayt)**

`n` = **6**

ÇAĞRI YIĞINI

main()

ÇIKTI

```
selam
```

`n = 6`, hata yok; `perror` atlanır.

Kod İzleme: `read()` ve Bloke Olma

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
6      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("okuma hatası");
9          return 1;
10     }
▶      write(1, buf, n);     // tam n bayt
12     return 0;
13 }
```

DEĞİŞKENLER

`buf = "selam\n…" (6 bayt)`

`n = 6`

ÇAĞRI YIĞINI

`write(1, buf, 6)`

`main()`

ÇIKTI

```
selam
selam
```

write tam 6 baytı yazar. buf bir C dizgisi olmadığından `printf("%s", buf)` hatalı olurdu.

Kod İzleme: `read()` ve Bloke Olma

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
6      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("okuma hatası");
9          return 1;
10     }
11     write(1, buf, n);    // tam n bayt
▶      return 0;
13 }
```

DEĞİŞKENLER

`buf = "selam\n…" (6 bayt)`

`n = 6`

ÇAĞRI YIĞINI

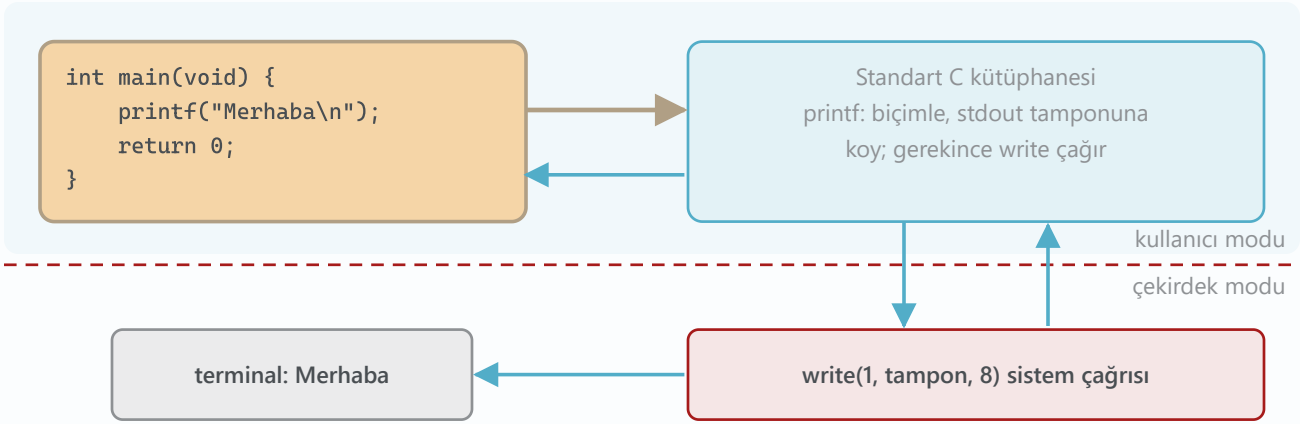
`main()`

ÇIKTI

```
selam
selam
```

Program 0 ile sonlanır.

Standart C Kütüphanesi ve Sistem Çağrıları

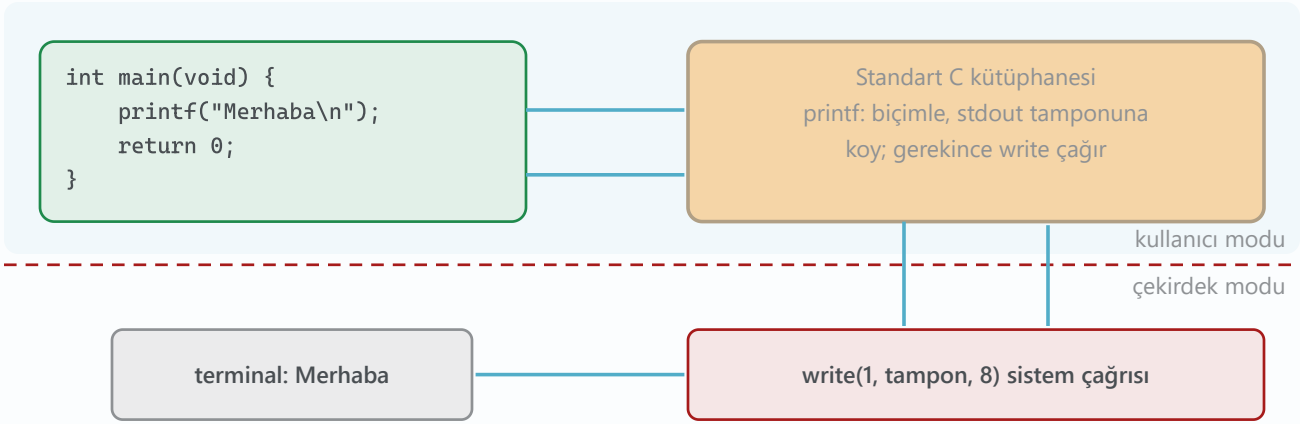


Program `printf` 'i çağırır; bu bir sistem çağrısı değil, kullanıcı modunda çalışan bir kütüphane fonksiyonudur.

Adım 1 / 4

Başlıklar: `stdio.h` (printf, fopen), `stdlib.h` (malloc, exit), `string.h` (strlen, memcpy), `math.h`, `time.h`; POSIX `unistd.h` (fork, read, write). Gerçeklemeler: glibc, musl, Windows'ta UCRT. `strlen` gibi işlevler tamamen kütüphane içindedir, `write` gibileri sistem çağrısını sarmalar.

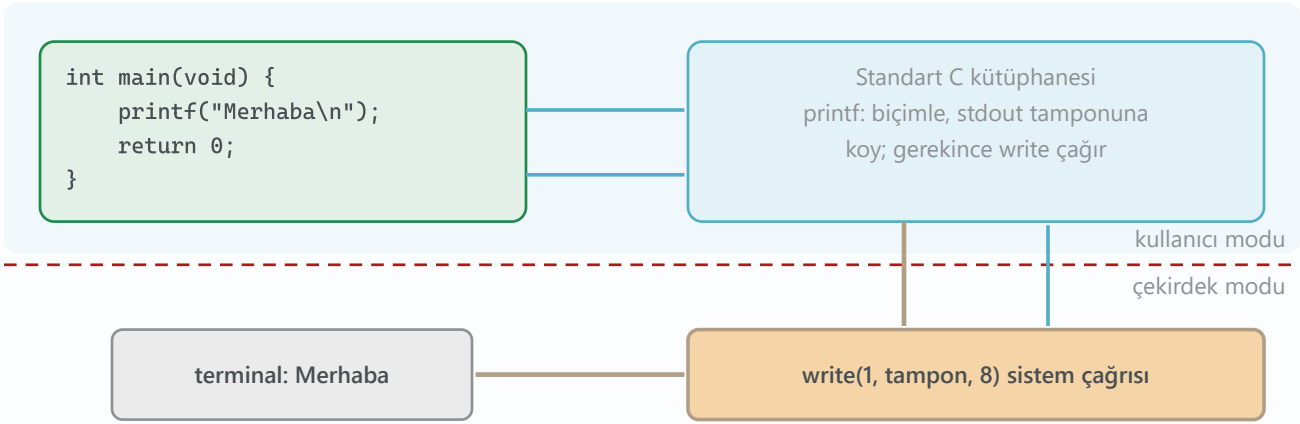
Standart C Kütüphanesi ve Sistem Çağrıları



libc metni biçimler ve stdout tamponuna koyar. Terminalde tampon **satır** tamponludur; dosyaya/boruya yönlendirilince **tam** tamponludur.

Başlıklar: `stdio.h` (printf, fopen), `stdlib.h` (malloc, exit), `string.h` (strlen, memcpy), `math.h`, `time.h`; POSIX `unistd.h` (fork, read, write). Gerçeklemeler: glibc, musl, Windows'ta UCRT. `strlen` gibi işlevler tamamen kütüphane içindedir, `write` gibileri sistem çağrısını sarmalar.

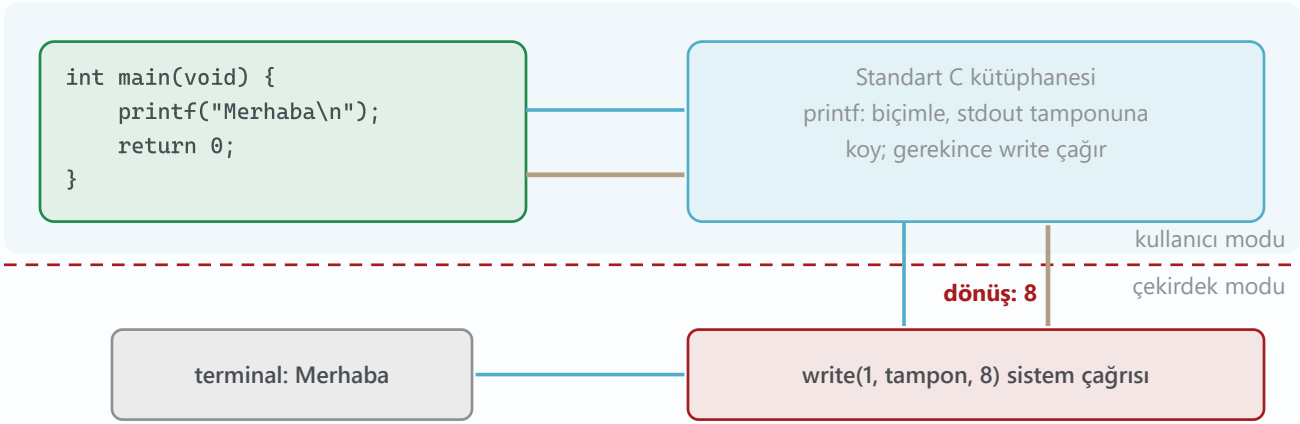
Standart C Kütüphanesi ve Sistem Çağrıları



"\n" geldiği için libc `write(1, tampon, 8)` sistem çağrısını yapar; çekirdek baytları terminale aktarır.

Başlıklar: `stdio.h` (printf, fopen), `stdlib.h` (malloc, exit), `string.h` (strlen, memcpy), `math.h`, `time.h`; POSIX `unistd.h` (fork, read, write). Gerçeklemeler: glibc, musl, Windows'ta UCRT. `strlen` gibi işlevler tamamen kütüphane içindedir, `write` gibileri sistem çağrısını sarmalar.

Standart C Kütüphanesi ve Sistem Çağrıları



write 8 döndürür, printf de çağırana döner. `strace ./merhaba` yalnızca write'ı gösterir, printf'i göstermez.

Başlıklar: `stdio.h` (printf, fopen), `stdlib.h` (malloc, exit), `string.h` (strlen, memcpy), `math.h`, `time.h`; POSIX `unistd.h` (fork, read, write). Gerçeklemeler: glibc, musl, Windows'ta UCRT. `strlen` gibi işlevler tamamen kütüphane içindedir, `write` gibileri sistem çağrısını sarmalar.

Sistem Çağrısı Türleri

Süreç denetimi
oluştur, sonlandır, yükle, bekle, sinyal
`fork, execve, exit, waitpid, kill`

Dosya yönetimi
oluştur, sil, aç, kapat, oku, yaz, öznitelik
`open, read, write, close, stat`

Aygıt yönetimi
aygıt iste / bırak, oku, yaz, denetle
`ioctl, read, write, mmap`

Bilgi bakımı
saat, tarih, sistem ve süreç bilgisi
`getpid, uname, clock_gettime`

İletişim
bağlantı kur, ileti gönder/al, paylaşımlı bellek
`pipe, socket, sendmsg, shm_open`

Koruma
izinleri al / ayarla, kullanıcı kimliği
`chmod, chown, umask, setuid`

Windows karşılıkları: CreateProcess, ReadFile, DeviceIoControl, GetSystemTime, CreatePipe, SetFileSecurity… (Silberschatz, Şekil 2.8)

POSIX Çağrıları: Süreç Yönetimi

Çağrı	Açıklama
`pid = fork()`	Ebeveynin kopyası olan bir çocuk süreç oluşturur.
`pid = waitpid(pid, &statloc, options)`	Bir çocuğun sonlanmasını bekler; çıkış durumunu `statloc` 'a yazar.
`s = execve(name, argv, envp)`	Sürecin bellek görüntüsünü yeni programla değiştirir; başarılıysa geri dönmez.
`exit(status)`	Süreci sonlandırır, durumu ebeveyne iletir.



Hata durumunda çağrılar **−1** döndürür ve `errno` ayarlanır. `pid` : süreç kimliği, `s` : dönüş kodu. Windows'ta tek bir `CreateProcess` çağrısı, fork + execve işini birlikte yapar.

Kod İzleme: Basit Bir Kabuk

```
1  #define TRUE 1
2
3  while (TRUE) {                /* sonsuza dek */
4      type_prompt();            /* istemi yaz */
5      read_command(command, parameters);
6
7      if (fork() != 0) {        /* ebeveyn */
8          waitpid(-1, &status, 0);
9      } else {                  /* çocuk süreç */
10         execve(command, parameters, 0);
11         perror("execve");     /* başarısızsa */
12         _exit(127);
13     }
14 }
```

DEĞİŞKENLER

`command = ?` `status = ?`

ÇAĞRI YIĞINI

`kabuk (812)`

ÇIKTI

```
$
```

Kabuk istemi yazar ve kullanıcıyı bekler.

Kod İzleme: Basit Bir Kabuk

```
1  #define TRUE 1
2
3  while (TRUE) {                  /* sonsuza dek */
4      type_prompt();              /* istemi yaz */
▶      read_command(command, parameters);
6
7      if (fork() != 0) {          /* ebeveyn */
8          waitpid(-1, &status, 0);
9      } else {                    /* çocuk süreç */
10         execve(command, parameters, 0);
11         perror("execve");       /* başarısızsa */
12         _exit(127);
13     }
14 }
```

DEĞİŞKENLER

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = ?`

ÇAĞRI YIĞINI

`kabuk (812)`

ÇIKTI

```
$ ls
```

Kullanıcı `ls` yazar; komut ve parametreler ayrıştırılır.

Kod İzleme: Basit Bir Kabuk

```
1  #define TRUE 1
2
3  while (TRUE) {                /* sonsuza dek */
4      type_prompt();            /* istemi yaz */
5      read_command(command, parameters);
6
▶      if (fork() != 0) {        /* ebeveyn */
8          waitpid(-1, &status, 0);
9      } else {                  /* çocuk süreç */
10         execve(command, parameters, 0);
11         perror("execve");     /* başarısızsa */
12         _exit(127);
13     }
14 }
```

DEĞİŞKENLER

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = ?`

`fork→ebeveyn = 1045` `fork→çocuk = 0`

ÇAĞRI YIĞINI

çocuk (1045)

kabuk (812)

ÇIKTI

```
$ ls
```

`fork` kabuğun kopyasını oluşturur. Ebeveynde sonuç 1045 (≠ 0), çocukta 0'dır.

Kod İzleme: Basit Bir Kabuk

```
1  #define TRUE 1
2
3  while (TRUE) {                /* sonsuza dek */
4      type_prompt();            /* istemi yaz */
5      read_command(command, parameters);
6
7      if (fork() != 0) {        /* ebeveyn */
▶          waitpid(-1, &status, 0);
9      } else {                  /* çocuk süreç */
10         execve(command, parameters, 0);
11         perror("execve");     /* başarısızsa */
12         _exit(127);
13     }
14 }
```

DEĞİŞKENLER

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = ?`

`fork→ebeveyn = 1045` `fork→çocuk = 0`

ÇAĞRI YIĞINI

çocuk (1045)

kabuk (812) [bekliyor]

ÇIKTI

```
$ ls
```

Ebeveyn `waitpid(-1, …)` ile herhangi bir çocuğun bitmesini bekler; bloke olur.

Kod İzleme: Basit Bir Kabuk

```
1  #define TRUE 1
2
3  while (TRUE) {                /* sonsuza dek */
4      type_prompt();            /* istemi yaz */
5      read_command(command, parameters);
6
7      if (fork() != 0) {        /* ebeveyn */
8          waitpid(-1, &status, 0);
9      } else {                  /* çocuk süreç */
▶          execve(command, parameters, 0);
11         perror("execve");     /* başarısızsa */
12         _exit(127);
13     }
14 }
```

DEĞİŞKENLER

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = ?`

`fork→ebeveyn = 1045` `fork→çocuk = 0`

ÇAĞRI YIĞINI

`ls (1045)`

`kabuk (812) [bekliyor]`

ÇIKTI

```
$ ls
notlar.txt  odev.c
```

Çocuk `execve` ile kendini ls programıyla değiştirir (PID aynı kalır); ls çıktısını yazar. Gerçek kabuklar komutu PATH'te arar.

Kod İzleme: Basit Bir Kabuk

```
1  #define TRUE 1
2
3  while (TRUE) {                /* sonsuza dek */
4      type_prompt();            /* istemi yaz */
5      read_command(command, parameters);
6
7      if (fork() != 0) {        /* ebeveyn */
▶          waitpid(-1, &status, 0);
9      } else {                  /* çocuk süreç */
10         execve(command, parameters, 0);
11         perror("execve");     /* başarısızsa */
12         _exit(127);
13     }
14 }
```

DEĞİŞKENLER

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = 0`

ÇAĞRI YIĞINI

`kabuk (812)`

ÇIKTI

```
$ ls
notlar.txt  odev.c
```

ls `exit(0)` ile biter; ebeveynin waitpid'i döner, `status` çıkış durumunu içerir.

Kod İzleme: Basit Bir Kabuk

```
1  #define TRUE 1
2
3  while (TRUE) {                /* sonsuza dek */
4      type_prompt();            /* istemi yaz */
5      read_command(command, parameters);
6
7      if (fork() != 0) {        /* ebeveyn */
8          waitpid(-1, &status, 0);
9      } else {                  /* çocuk süreç */
10         execve(command, parameters, 0);
11         perror("execve");     /* başarısızsa */
12         _exit(127);
13     }
14 }
```

DEĞİŞKENLER

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = 0`

ÇAĞRI YIĞINI

kabuk (812)

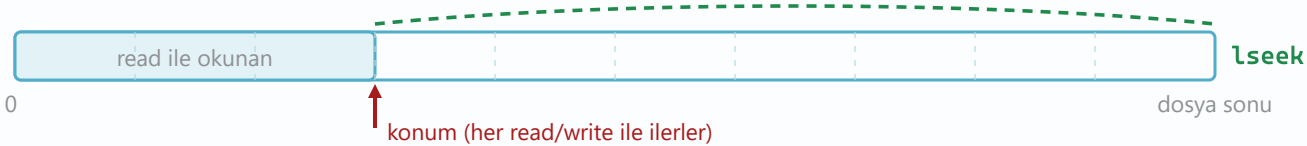
ÇIKTI

```
$ ls
notlar.txt  odev.c
$
```

Döngü başa döner, yeni istem yazılır.

POSIX Çağrıları: Dosya Yönetimi

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



fd: dosya tanımlayıcısı, n: bayt sayısı, position: dosya başına göre konum. Güncel Linux ayrıca `openat` , `pread` / `pwrite` , `statx` sunar.

POSIX Çağrıları: Dizin Yönetimi ve Diğerleri

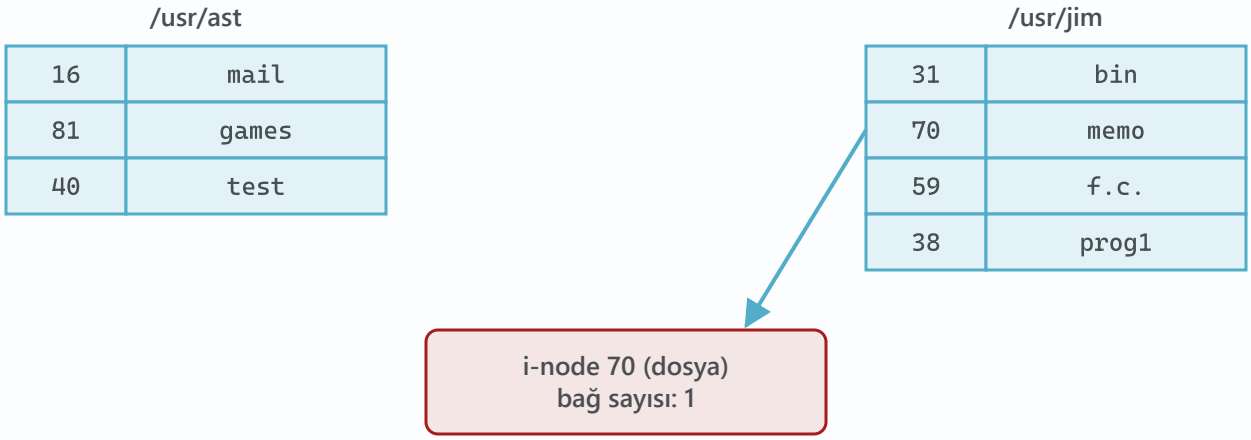
Çağrı	Açıklama
`s = mkdir(name, mode)`	Yeni bir dizin oluşturur.
`s = rmdir(name)`	Boş bir dizini siler.
`s = link(name1, name2)`	name1'in gösterdiği dosyaya name2 adlı yeni bir dizin girdisi oluşturur.
`s = unlink(name)`	Bir dizin girdisini siler.
`s = mount(special, name, flag)`	Bir dosya sistemini bağlar (Linux: `mount(src, hedef, tür, bayrak, veri)`).
`s = umount(special)`	Bir dosya sisteminin bağlantısını kaldırır.
`s = chdir(dirname)`	Çalışma dizinini değiştirir.
`s = chmod(name, mode)`	Dosyanın koruma (izin) bitlerini değiştirir.
`s = kill(pid, signal)`	Bir sürece sinyal gönderir (yalnızca öldürmek için değil).
`seconds = time(&seconds)`	1 Ocak 1970 00:00 UTC'den (Unix epoch) bu yana geçen saniye.

32-bit `time_t` 19 Ocak 2038'de taşar (2038 sorunu); güncel sistemler 64-bit `time_t` kullanır.

SİSTEM ÇAĞRILARI

`link` ve `unlink`: Bir Dosya, İki Ad

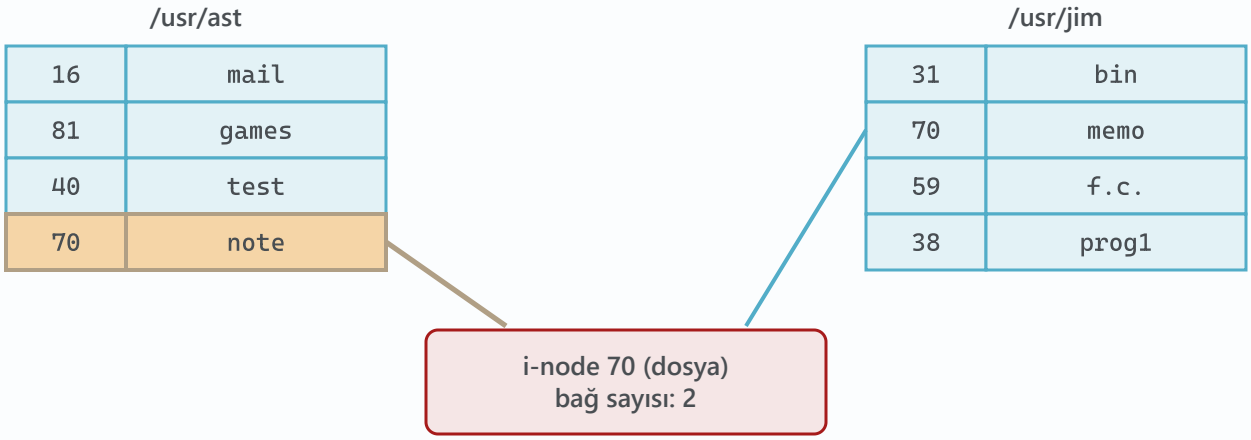
UNIX'te dizin, (i-node numarası, ad) çiftlerinden oluşan bir tablodur. Dosyanın asıl bilgisi i-node'dadır (inode).



Başlangıç: `/usr/jim/memo` 70 numaralı i-node'u gösterir; bağ sayısı 1.

`link` ve `unlink`: Bir Dosya, İki Ad

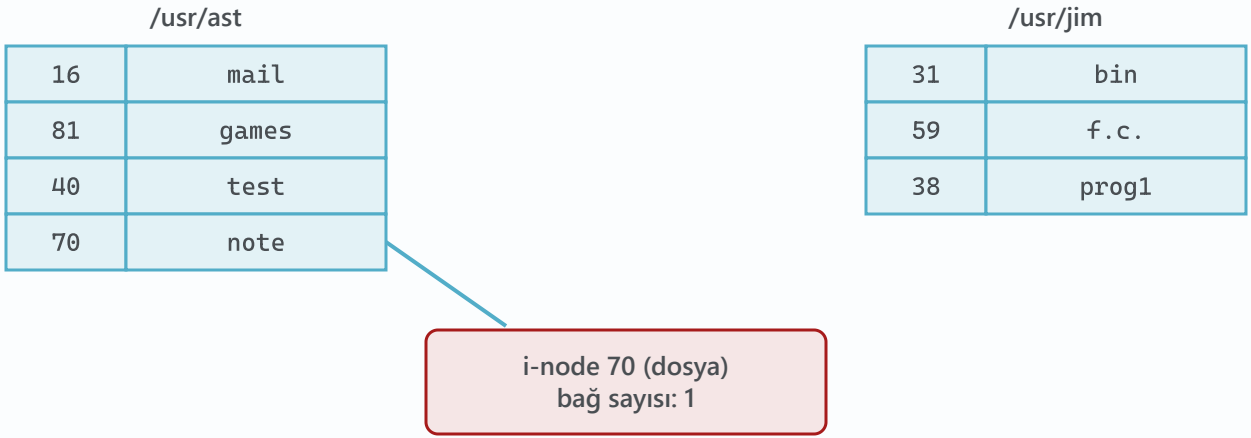
UNIX'te dizin, (i-node numarası, ad) çiftlerinden oluşan bir tablodur. Dosyanın asıl bilgisi i-node'dadır (inode).



`link("/usr/jim/memo", "/usr/ast/note")`: /usr/ast'a aynı i-node'u (70) gösteren yeni girdi eklenir. Veri kopyalanmaz; sayı 2 olur.

`link` ve `unlink`: Bir Dosya, İki Ad

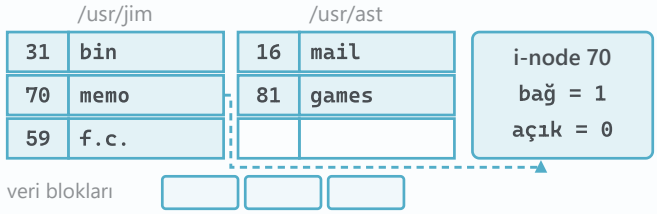
UNIX'te dizin, (i-node numarası, ad) çiftlerinden oluşan bir tablodur. Dosyanın asıl bilgisi i-node'dadır (inode).



`unlink("/usr/jim/memo")` yalnızca girdiyi siler; dosya `note` adıyla yaşar. Sayı 0 olunca (ve açık tutan kalmayınca) blokları serbest kalır.

Kod İzleme: Bağ Sayısı ve Açık Dosya

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



Dizin yalnızca (i-node numarası, ad) çiftlerinden oluşur. `memo` 70 numaralı i-node'u gösteriyor; bağ sayısı 1, dosya 85 bayt.

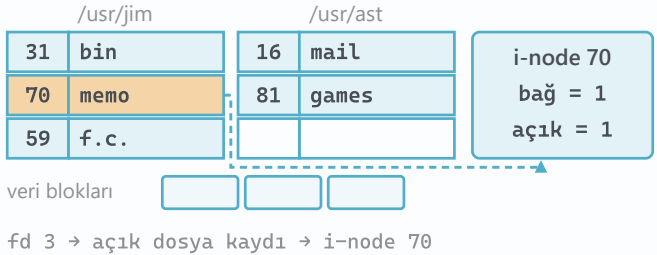
Adım 1 / 7

DEĞİŞKENLER

`fd = ?` `n = ?`

Kod İzleme: Bağ Sayısı ve Açık Dosya

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



`open` yolu çözer ve i-node 70'i açar: **fd = 3**, i-node'un açık sayacı 1.

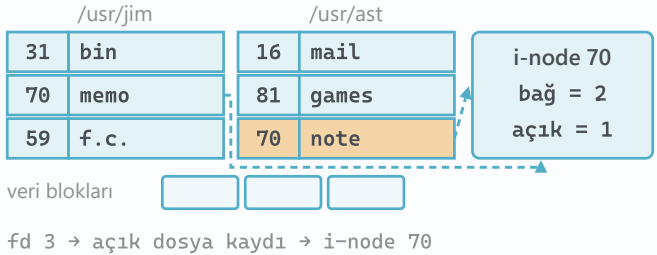
Adım 2 / 7

DEĞİŞKENLER

`fd = 3` `n = ?`

Kod İzleme: Bağ Sayısı ve Açık Dosya

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



`link` /usr/ast'a **aynı i-node'u** gösteren yeni bir girdi ekler. Veri kopyalanmaz; bağ = 2.

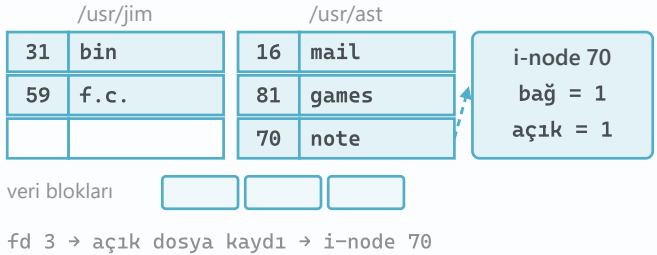
Adım 3 / 7

DEĞİŞKENLER

`fd = 3` `n = ?`

Kod İzleme: Bağ Sayısı ve Açık Dosya

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



`unlink` yalnızca dizin girdisini siler; dosya `note` adıyla yaşar. Bağ = 1.

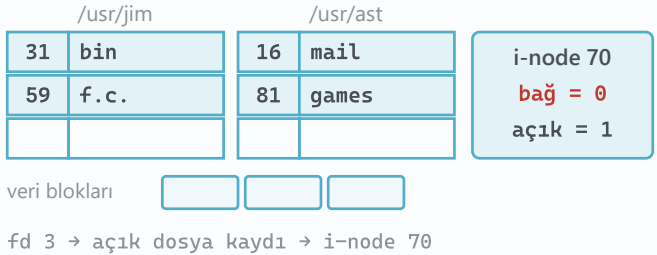
Adım 4 / 7

DEĞİŞKENLER

`fd = 3` `n = ?`

Kod İzleme: Bağ Sayısı ve Açık Dosya

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



Son ad da silindi: **bağ = 0**. Dosyanın artık hiçbir yolu yok ama fd 3 onu açık tuttuğu için i-node ve bloklar **silinmez**.

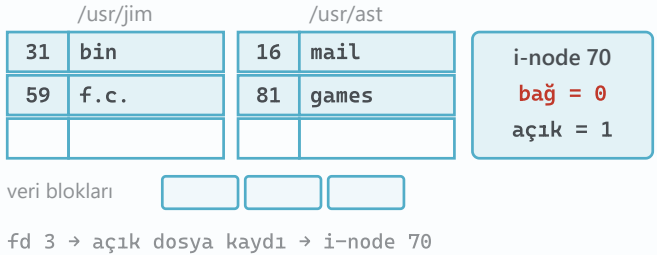
Adım 5 / 7

DEĞİŞKENLER

`fd = 3` `n = ?`

Kod İzleme: Bağ Sayısı ve Açık Dosya

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



`read` tanımlayıcı üzerinden doğrudan i-node'a gider, ad gerekmez: **n = 85**.

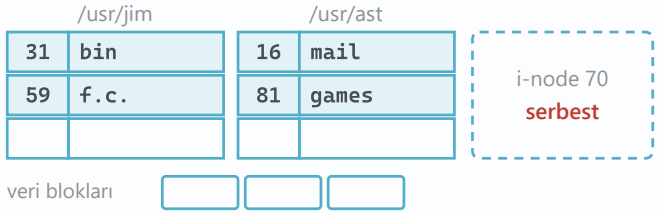
Adım 6 / 7

DEĞİŞKENLER

`fd = 3` `n = 85`

Kod İzleme: Bağ Sayısı ve Açık Dosya

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



Son tanımlayıcı kapandı: bağ = 0 ve açık = 0. Çekirdek ancak şimdi i-node'u ve veri bloklarını **serbest bırakır**. Geçici dosyalar bu yolla açıkken silinir.

Adım 7 / 7

DEĞİŞKENLER

`fd = 3` `n = 85`

UNIX Çağrılarının Win32 API Karşılıkları

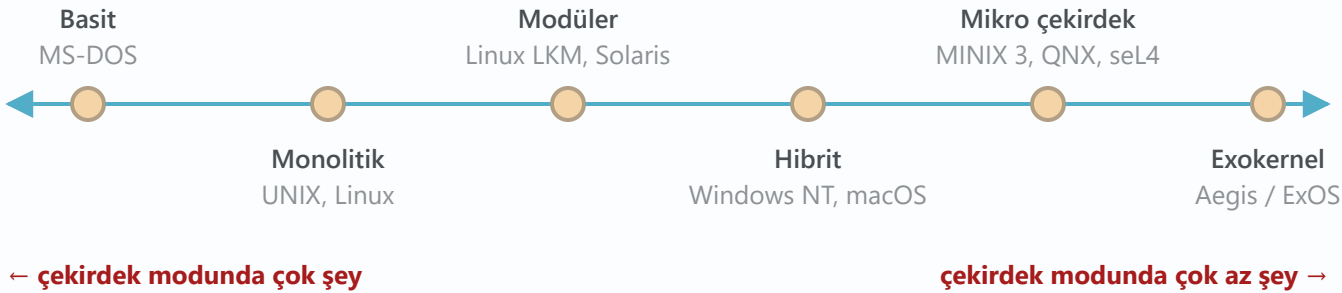
UNIX	Win32
`fork`	`CreateProcess`
`waitpid`	`WaitForSingleObject`
`execve`	(yok: CreateProcess)
`exit`	`ExitProcess`
`open`	`CreateFile`
`close`	`CloseHandle`
`read`	`ReadFile`
`write`	`WriteFile`
`lseek`	`SetFilePointerEx`
`stat`	`GetFileAttributesEx`

UNIX	Win32
`mkdir`	`CreateDirectory`
`rmdir`	`RemoveDirectory`
`link`	`CreateHardLink`
`unlink`	`DeleteFile`
`mount`	`SetVolumeMountPoint`
`umount`	`DeleteVolumeMountPoint`
`chdir`	`SetCurrentDirectory`
`chmod`	(yok: ACL, SetFileSecurity)
`kill`	(yok: TerminateProcess)
`time`	`GetSystemTime`

Win32 API binlerce fonksiyon içerir; asıl çekirdek geçişini `ntdll.dll`'deki `Nt…` fonksiyonları (ör. `NtCreateFile`) yapar. Bu çağrıların numaraları sürümden sürüme değiştiği ve belgelenmediği için programlar API'yi kullanır. Tanenbaum 3e'de "yok" görünen `link` ve `mount` karşılıkları bugün mevcuttur.

YAPILAR

İşletim Sistemi Yapıları: Genel Bakış



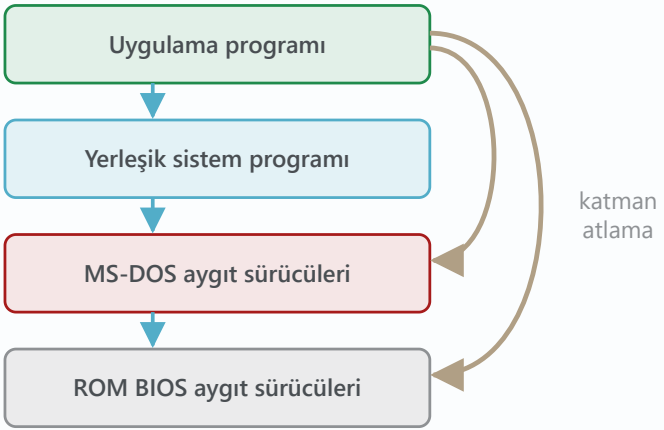
Katmanlı yapı

Bir yerleşim değil, düzenleme ilkesidir: bileşenler hiyerarşik katmanlara ayrılır (THE, MULTICS halkaları).

Sanal makineler

Ayrı bir eksen: donanımın kendisi sanallaştırılır, üzerinde birden çok işletim sistemi çalışır.

Basit Yapı: MS-DOS



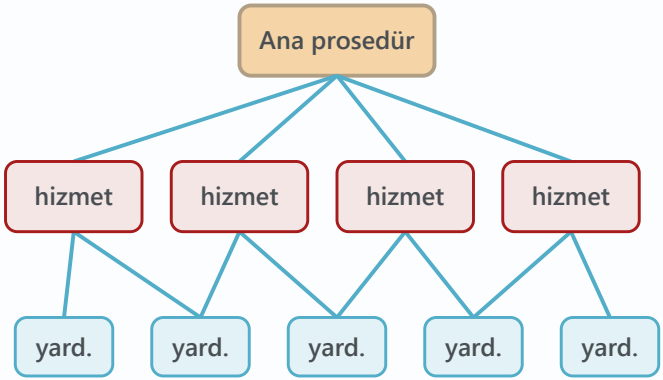
- Belirgin bir yapısı yoktur: küçük, basit ve sınırlı donanım için yazılmıştır.
- Arayüzler ve işlev düzeyleri iyi ayrılmamıştır; uygulamalar BIOS rutinlerine ve donanıma **doğrudan** erişebilir.
- Intel 8088'de çift mod olmadığından donanım koruması yoktu: tek bir hatalı program **tüm sistemi çökertebilir**.
- Modüller arasında net sınır ve veri gizleme (data hiding) yoktur.
- Silberschatz bunu "basit yapı" olarak, monolitik çekirdeklerden ayrı ele alır.

Monolitik Sistem

Tüm işletim sistemi, çekirdek modunda çalışan **tek bir program** olarak bağlanır. Her prosedür diğerini doğrudan çağırabilir.

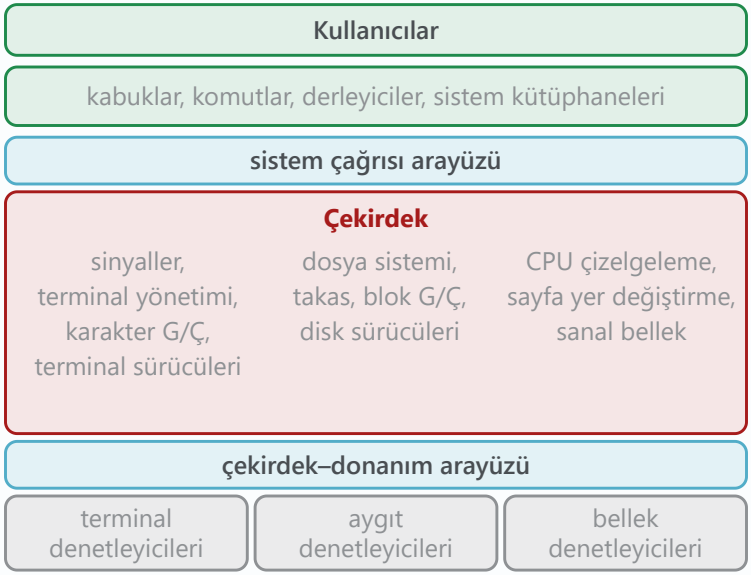
1. **Ana program:** çağrı numarasına bakıp istenen hizmet prosedürünü başlatır.
2. **Hizmet prosedürleri:** her sistem çağrısını gerçekleştirir.
3. **Yardımcı prosedürler:** birden çok hizmetin ortak işleri (ör. kullanıcı belleğinden veri almak).

Çoğu monolitik çekirdek bugün sürücüleri ve dosya sistemlerini çalışma anında **yüklenebilir modül** olarak da alır.



hepsi tek ikili dosya, tek adres alanı, çekirdek modu

Geleneksel UNIX ve Linux: Monolitik Çekirdek



Güçlü yönler

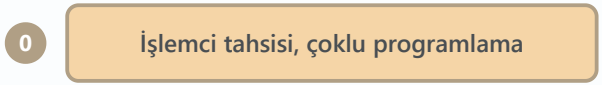
+ Doğrudan fonksiyon çağrıları: IPC ve mod geçişi yok, **hızlı**.
+ Tasarımı ve gerçeklemesi görece basit, bütünleşik.

Zayıf yönler

– Tek bir sürücü hatası **tüm çekirdeği** çökertebilir.
– Dev kod tabanı (Linux 2025'te 40 milyon satırı aştı): hata bulmak ve bakım zor.

Katmanlı Yapı

İşletim sistemi katmanlara bölünür: en alt katman donanıma, en üst katman kullanıcıya en yakındır. Her katman **yalnızca altındakilerin** hizmetlerini kullanır.



Katman 0: kesmelerde ve zamanlayıcı dolunca süreçler arasında geçiş yapar; üstteki katmanlar tek CPU'yu düşünmek zorunda kalmaz.

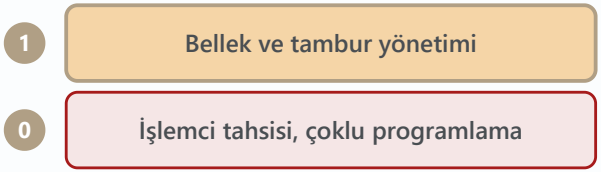
Artılar

+ Hata ayıklama ve doğrulama katman katman yapılır; hata ilgili katmanda aranır.
+ Soyutlama ve veri gizleme kolaydır.

Eksiler

– Katmanları doğru tanımlamak zordur (bağımlılıklar çoğu kez iki yönlüdür).
– Her katman geçişi ek yük getirir. Fikir bugün ağ yığınlarında yaşar.

Katmanlı Yapı



Katman 1: süreçlere bellek ayırır, sayfaları tambur (o günün diski) ile bellek arasında taşır.

İşletim sistemi katmanlara bölünür: en alt katman donanıma, en üst katman kullanıcıya en yakındır. Her katman **yalnızca altındakilerin** hizmetlerini kullanır.

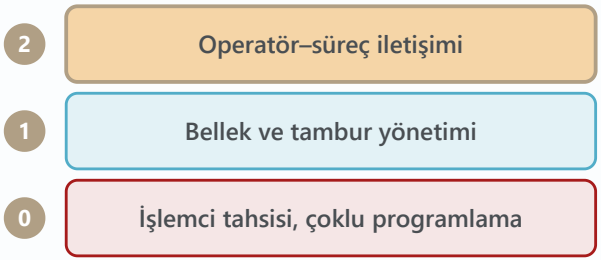
Artılar

+ Hata ayıklama ve doğrulama katman katman yapılır; hata ilgili katmanda aranır.
+ Soyutlama ve veri gizleme kolaydır.

Eksiler

– Katmanları doğru tanımlamak zordur (bağımlılıklar çoğu kez iki yönlüdür).
– Her katman geçişi ek yük getirir. Fikir bugün ağ yığınlarında yaşar.

Katmanlı Yapı



Katman 2: her süreç ile operatör konsolu arasındaki iletişimi yönetir.

İşletim sistemi katmanlara bölünür: en alt katman donanıma, en üst katman kullanıcıya en yakındır. Her katman **yalnızca altındakilerin** hizmetlerini kullanır.

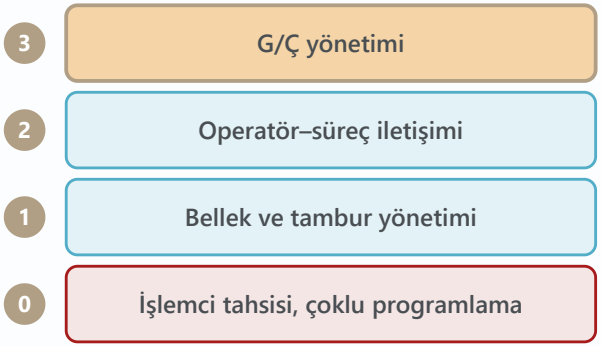
Artılar

+ Hata ayıklama ve doğrulama katman katman yapılır; hata ilgili katmanda aranır.
+ Soyutlama ve veri gizleme kolaydır.

Eksiler

– Katmanları doğru tanımlamak zordur (bağımlılıklar çoğu kez iki yönlüdür).
– Her katman geçişi ek yük getirir. Fikir bugün ağ yığınlarında yaşar.

Katmanlı Yapı



Katman 3: G/Ç aygıtlarını yönetir; üst katmanlar soyut aygıtlarla çalışır.

İşletim sistemi katmanlara bölünür: en alt katman donanıma, en üst katman kullanıcıya en yakındır. Her katman **yalnızca altındakilerin** hizmetlerini kullanır.

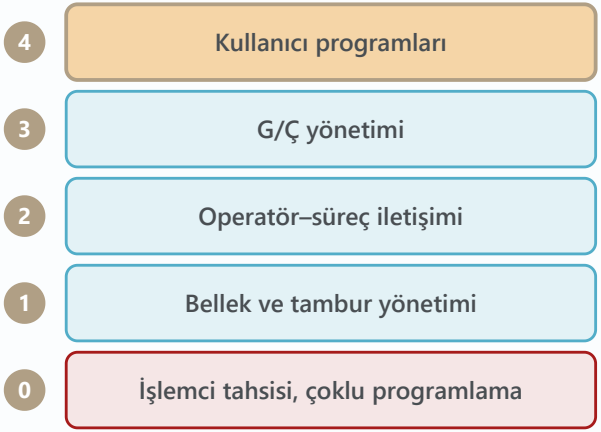
Artılar

+ Hata ayıklama ve doğrulama katman katman yapılır; hata ilgili katmanda aranır.
+ Soyutlama ve veri gizleme kolaydır.

Eksiler

– Katmanları doğru tanımlamak zordur (bağımlılıklar çoğu kez iki yönlüdür).
– Her katman geçişi ek yük getirir. Fikir bugün ağ yığınlarında yaşar.

Katmanlı Yapı



Katman 4: kullanıcı programları; bellek, konsol ve G/Ç ayrıntılarıyla uğraşmaz.

İşletim sistemi katmanlara bölünür: en alt katman donanıma, en üst katman kullanıcıya en yakındır. Her katman **yalnızca altındakilerin** hizmetlerini kullanır.

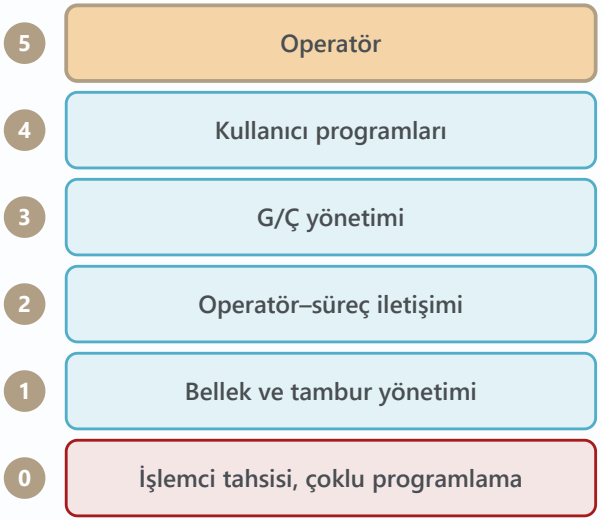
Artılar

+ Hata ayıklama ve doğrulama katman katman yapılır; hata ilgili katmanda aranır.
+ Soyutlama ve veri gizleme kolaydır.

Eksiler

– Katmanları doğru tanımlamak zordur (bağımlılıklar çoğu kez iki yönlüdür).
– Her katman geçişi ek yük getirir. Fikir bugün ağ yığınlarında yaşar.

Katmanlı Yapı



Katman 5: sistem operatörü. Dijkstra'nın THE sistemi (1968) bu altı katmandan oluşuyordu.

İşletim sistemi katmanlara bölünür: en alt katman donanıma, en üst katman kullanıcıya en yakındır. Her katman **yalnızca altındakilerin** hizmetlerini kullanır.

Artılar

+ Hata ayıklama ve doğrulama katman katman yapılır; hata ilgili katmanda aranır.
+ Soyutlama ve veri gizleme kolaydır.

Eksiler

– Katmanları doğru tanımlamak zordur (bağımlılıklar çoğu kez iki yönlüdür).
– Her katman geçişi ek yük getirir. Fikir bugün ağ yığınlarında yaşar.

Mikro Çekirdek (Microkernel)

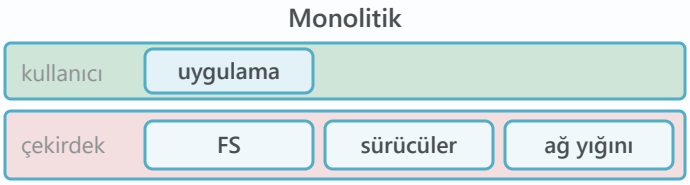
- Gerekli olmayan her şey çekirdekten çıkarılır; çekirdekte yalnızca **IPC (ileti geçirme)**, temel çizelgeleme, adres alanı yönetimi ve kesme işleme kalır.
- Sürücüler, dosya sistemleri ve ağ yığını **kullanıcı modunda sunucu süreçler** olarak çalışır; yeni hizmetler kullanıcı alanına eklenir, çekirdek değişmez.
- Örnekler: Mach, MINIX 3, QNX Neutrino (otomotiv), L4 ailesi, biçimsel olarak doğrulanmış **seL4**, Google Fuchsia'nın **Zircon** çekirdeği. macOS ise Mach'tan türeyen **hibrit** XNU'dur.
- **Mekanizma çekirdekte, politika dışarıda:** çekirdek öncelikli süreçleri çalıştırmayı sağlar; önceliklerin nasıl atanacağına kullanıcı modundaki bileşen karar verir.

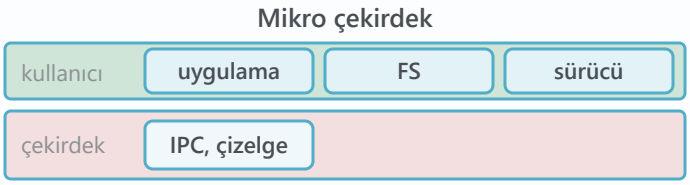
Artılar

+ Çöken sürücü sistemi düşürmez, yeniden başlatılabilir.
+ Küçük çekirdek: test, doğrulama ve taşıma kolay.
+ Daha güvenli: saldırı yüzeyi küçük.

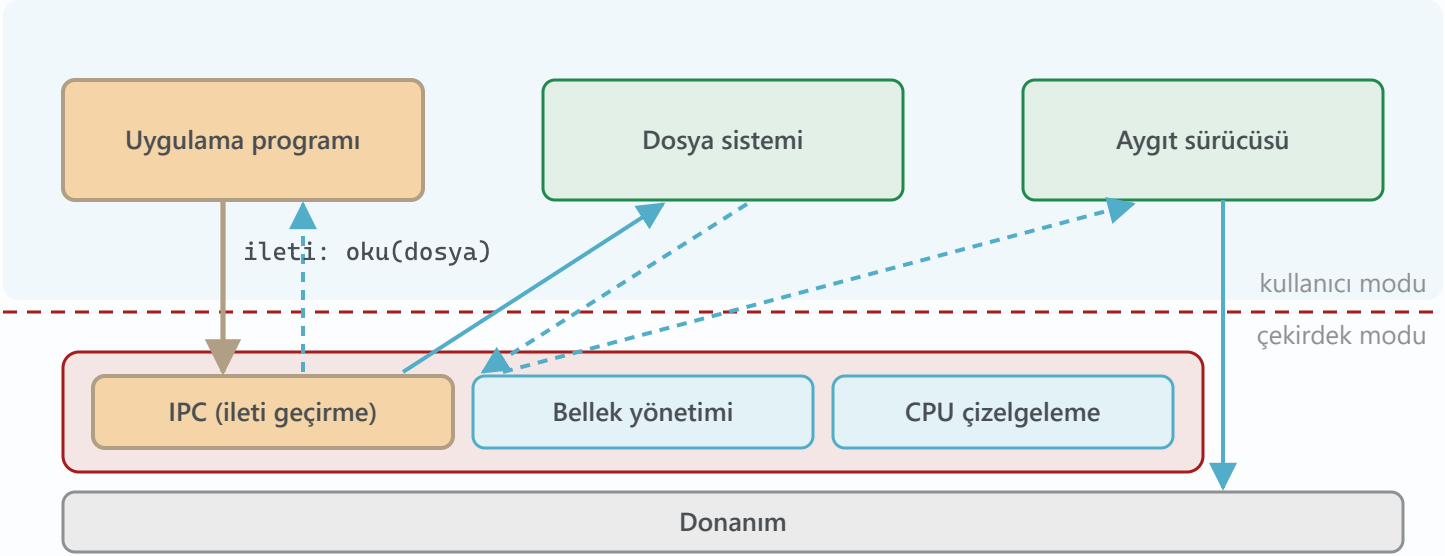
Eksiler

– İletiler ve mod geçişleri ek yük getirir; L4 tasarımları bunu büyük ölçüde azalttı.



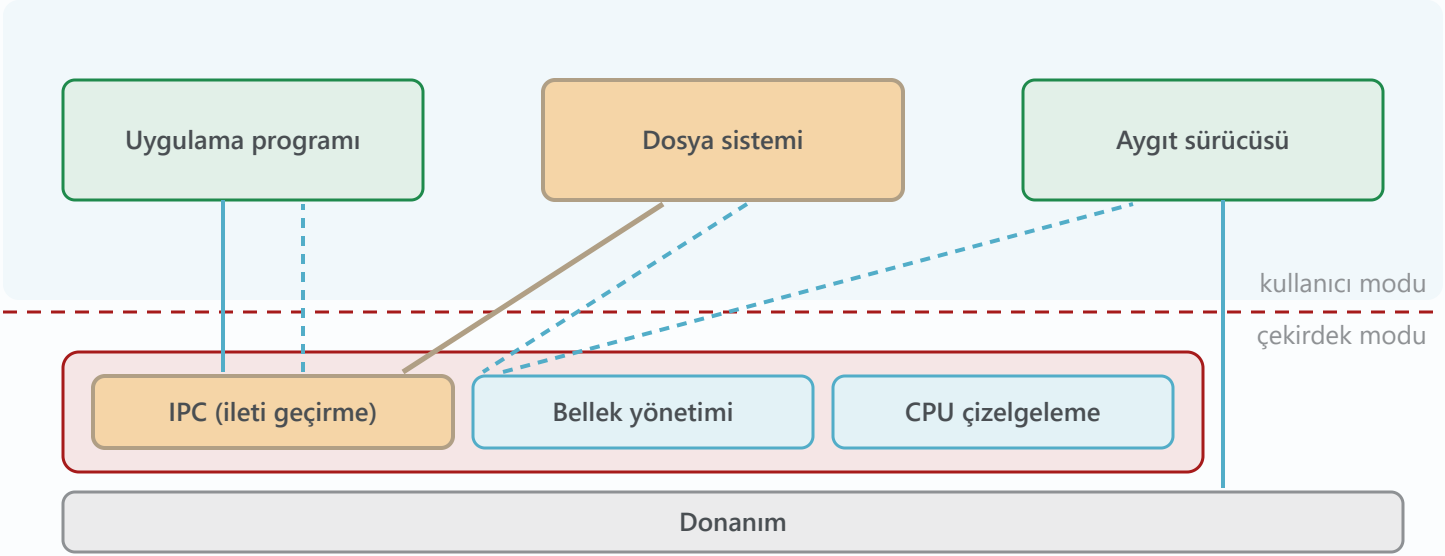


Mikro Çekirdekte Bir Dosya Okuma



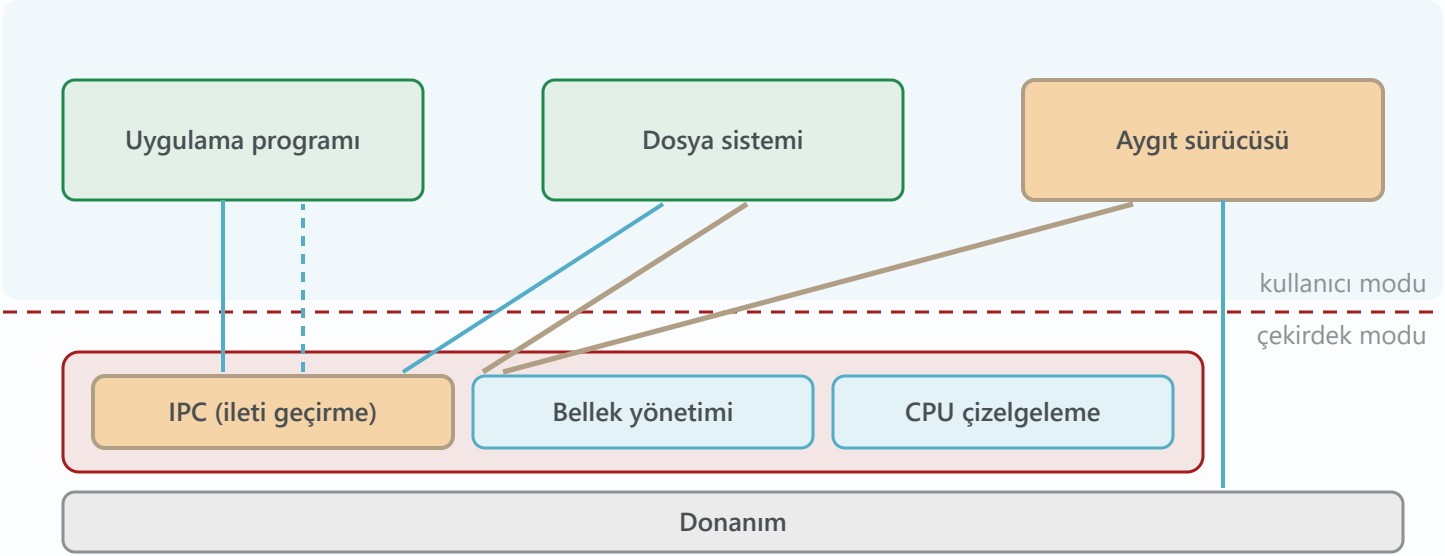
Uygulama, dosya sistemi sunucusuna bir **ileti** gönderir; ileti çekirdeğin IPC mekanizmasından geçer.

Mikro Çekirdekte Bir Dosya Okuma



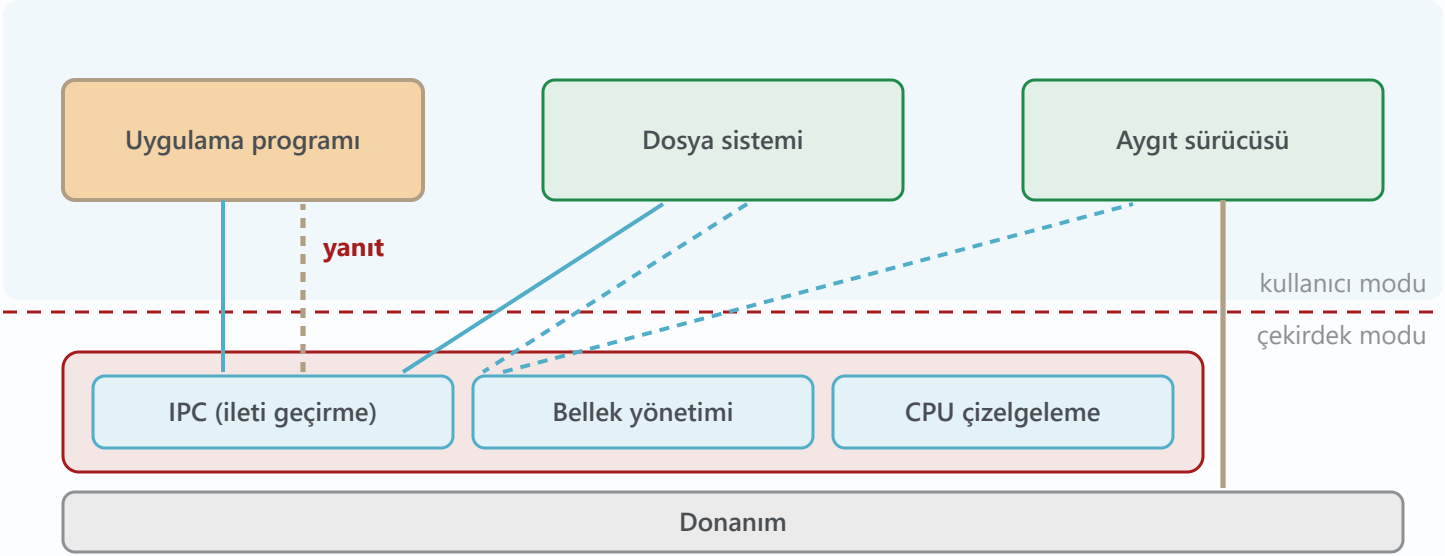
Çekirdek iletiyi, kullanıcı modundaki dosya sistemi sunucusuna teslim eder.

Mikro Çekirdekte Bir Dosya Okuma



Dosya sistemi hangi bloğun okunacağını bulur ve yine çekirdek üzerinden disk sürücüsüne ileti yollar.

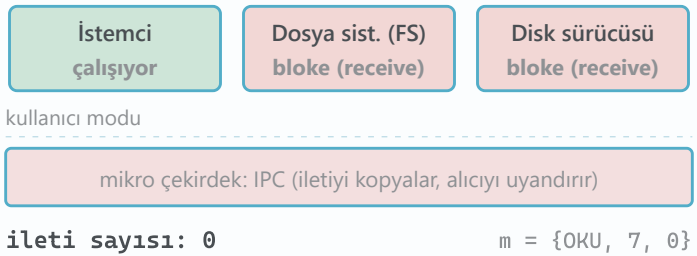
Mikro Çekirdekte Bir Dosya Okuma



Sürücü, çekirdeğin izin verdiği G/Ç ile donanıma erişir; yanıtlar aynı yoldan döner. Monolitik çekirdekte bu iş tek bir fonksiyon çağrı zinciridir.

Kod İzleme: Mikro Çekirdekte `send`/`receive`

```
1  /* istemci */
▶ 2  m.tur = OKU; m.dosya = 7; m.ofset = 0;
3  send(FS, &m);
4  receive(FS, &m);        /* yanıtı bekle */
5  /* dosya sistemi sunucusu (FS) */
6  while (1) {
7    receive(ANY, &m);
8    blok = blok_bul(m.dosya, m.ofset);
9    send(DISK, &blok);
10   receive(DISK, &veri);
11   m.sonuc = veri;
12   send(m.kaynak, &m);
13 }
```



FS ve disk sürücüsü kullanıcı modunda birer süreç; ikisi de `receive`'de iş bekliyor. İstemci isteğini bir **ileti** yapısına yazar.

Adım 1 / 8

DEĞİŞKENLER

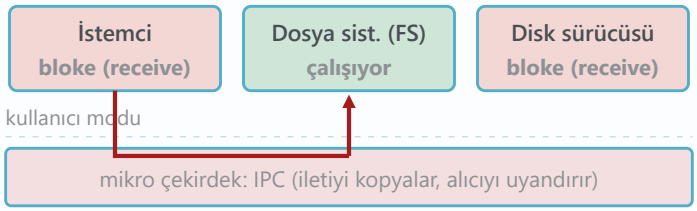
`m.tur = OKU` `m.dosya = 7` `m.ofset = 0`

ÇAĞRI YIĞINI

istemci

Kod İzleme: Mikro Çekirdekte `send`/`receive`

```
1  /* istemci */
2  m.tur = OKU; m.dosya = 7; m.ofset = 0;
▶  send(FS, &m);
▶  receive(FS, &m);        /* yanıtı bekle */
5  /* dosya sistemi sunucusu (FS) */
6  while (1) {
7    receive(ANY, &m);
8    blok = blok_bul(m.dosya, m.ofset);
9    send(DISK, &blok);
10   receive(DISK, &veri);
11   m.sonuc = veri;
12   send(m.kaynak, &m);
13 }
```



`ileti sayısı: 1` `m → FS`

`send` çekirdeğe tuzaktır: çekirdek iletiyi FS'nin tamponuna **kopyalar** ve FS'yi uyandırır. İstemci yanıt için `receive`'de bloke olur.

Adım 2 / 8

DEĞİŞKENLER

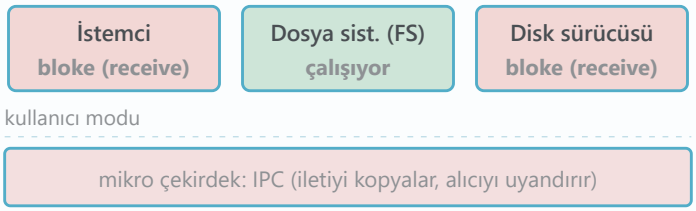
`m.tur = OKU` `m.dosya = 7` `m.ofset = 0`

ÇAĞRI YIĞINI

FS

Kod İzleme: Mikro Çekirdekte `send`/`receive`

```
1  /* istemci */
2  m.tur = OKU; m.dosya = 7; m.ofset = 0;
3  send(FS, &m);
4  receive(FS, &m);        /* yanıtı bekle */
5  /* dosya sistemi sunucusu (FS) */
6  while (1) {
▶    receive(ANY, &m);
▶    blok = blok_bul(m.dosya, m.ofset);
9    send(DISK, &blok);
10   receive(DISK, &veri);
11   m.sonuc = veri;
12   send(m.kaynak, &m);
13 }
```



`ileti sayısı: 1`

FS'nin `receive(ANY)` çağrısı döner; ileti kimden geldiyse `m.kaynak` 'ta yazılıdır. FS, dosya 7'nin bağıl konum 0'ının **blok 120**'de olduğunu bulur.

Adım 3 / 8

DEĞİŞKENLER

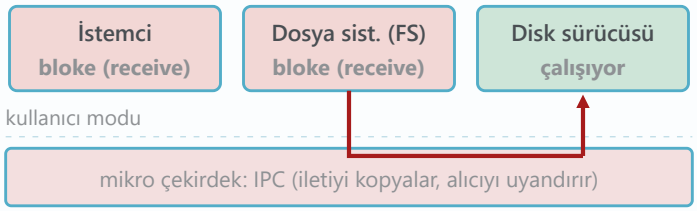
`m.kaynak = istemci` `blok = 120`

ÇAĞRI YIĞINI

FS

Kod İzleme: Mikro Çekirdekte `send`/`receive`

```
1  /* istemci */
2  m.tur = OKU; m.dosya = 7; m.ofset = 0;
3  send(FS, &m);
4  receive(FS, &m);        /* yanıtı bekle */
5  /* dosya sistemi sunucusu (FS) */
6  while (1) {
7    receive(ANY, &m);
8    blok = blok_bul(m.dosya, m.ofset);
▶    send(DISK, &blok);
▶    receive(DISK, &veri);
11   m.sonuc = veri;
12   send(m.kaynak, &m);
13 }
```



ileti sayısı: 2 blok 120 → DISK

FS de doğrudan diske erişemez: sürücüye ikinci iletiyi yollar ve yanıtı bekler.

Adım 4 / 8

DEĞİŞKENLER

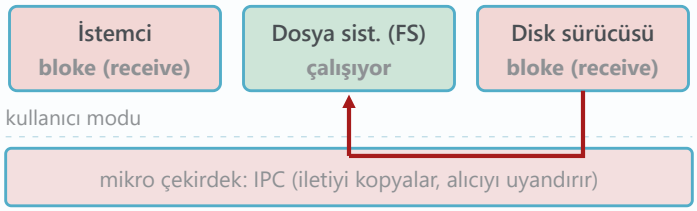
`m.kaynak = istemci` `blok = 120`

ÇAĞRI YIĞINI

disk sürücüsü

Kod İzleme: Mikro Çekirdekte `send`/`receive`

```
1  /* istemci */
2  m.tur = OKU; m.dosya = 7; m.ofset = 0;
3  send(FS, &m);
4  receive(FS, &m);        /* yanıtı bekle */
5  /* dosya sistemi sunucusu (FS) */
6  while (1) {
7    receive(ANY, &m);
8    blok = blok_bul(m.dosya, m.ofset);
9    send(DISK, &blok);
▶    receive(DISK, &veri);
▶    m.sonuc = veri;
12   send(m.kaynak, &m);
13 }
```



`ileti sayısı: 3` `veri → FS`

Sürücü, çekirdeğin izin verdiği G/Ç ile bloğu okur ve **üçüncü** iletiyle veriyi FS'ye döndürür. FS yanıtı hazırlar.

Adım 5 / 8

DEĞİŞKENLER

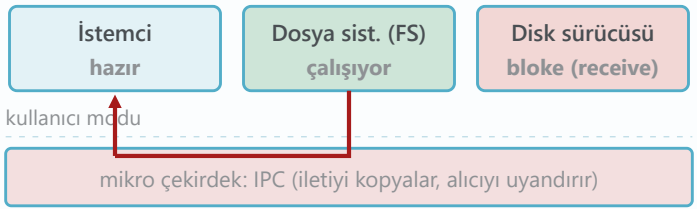
`m.kaynak = istemci` `blok = 120` `m.sonuc = veri`

ÇAĞRI YIĞINI

FS

Kod İzleme: Mikro Çekirdekte `send`/`receive`

```
1  /* istemci */
2  m.tur = OKU; m.dosya = 7; m.ofset = 0;
3  send(FS, &m);
4  receive(FS, &m);        /* yanıtı bekle */
5  /* dosya sistemi sunucusu (FS) */
6  while (1) {
7    receive(ANY, &m);
8    blok = blok_bul(m.dosya, m.ofset);
9    send(DISK, &blok);
10   receive(DISK, &veri);
11   m.sonuc = veri;
▶    send(m.kaynak, &m);
13 }
```



ileti sayısı: 4 `m → istemci`

Dördüncü ileti yanıtı istemciye taşır; istemci **hazır** olur.

Adım 6 / 8

DEĞİŞKENLER

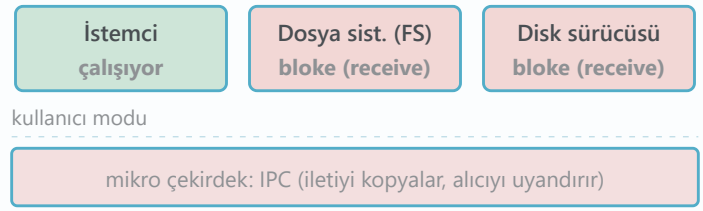
`m.kaynak = istemci` `m.sonuc = veri`

ÇAĞRI YIĞINI

FS

Kod İzleme: Mikro Çekirdekte `send`/`receive`

```
1  /* istemci */
2  m.tur = OKU; m.dosya = 7; m.ofset = 0;
3  send(FS, &m);
4  receive(FS, &m);        /* yanıtı bekle */
5  /* dosya sistemi sunucusu (FS) */
6  while (1) {
▶    receive(ANY, &m);
8    blok = blok_bul(m.dosya, m.ofset);
9    send(DISK, &blok);
10   receive(DISK, &veri);
11   m.sonuc = veri;
12   send(m.kaynak, &m);
13 }
```



`ileti sayısı: 4`

FS döngü başına döner ve yeni iş bekler.

Adım 7 / 8

DEĞİŞKENLER

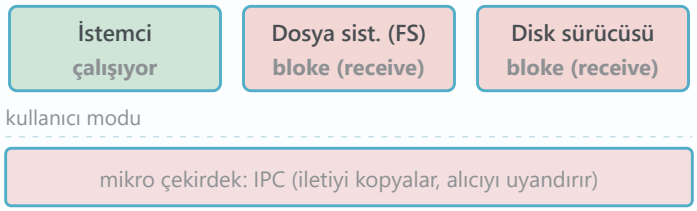
`m.sonuc = veri`

ÇAĞRI YIĞINI

FS

Kod İzleme: Mikro Çekirdekte `send`/`receive`

```
1  /* istemci */
2  m.tur = OKU; m.dosya = 7; m.ofset = 0;
3  send(FS, &m);
▶  receive(FS, &m);        /* yanıtı bekle */
5  /* dosya sistemi sunucusu (FS) */
6  while (1) {
7    receive(ANY, &m);
8    blok = blok_bul(m.dosya, m.ofset);
9    send(DISK, &blok);
10   receive(DISK, &veri);
11   m.sonuc = veri;
12   send(m.kaynak, &m);
13 }
```



`ileti sayısı: 4`

İstemcinin `receive`'i döner: veri `m.sonuc`'ta. Tek okuma için **4 ileti** ve her birinde kullanıcı ↔ çekirdek geçişi oldu; monolitik çekirdekte aynı iş tek bir sistem çağrısı içinde işlev çağrılarıyla biterdi.

Adım 8 / 8

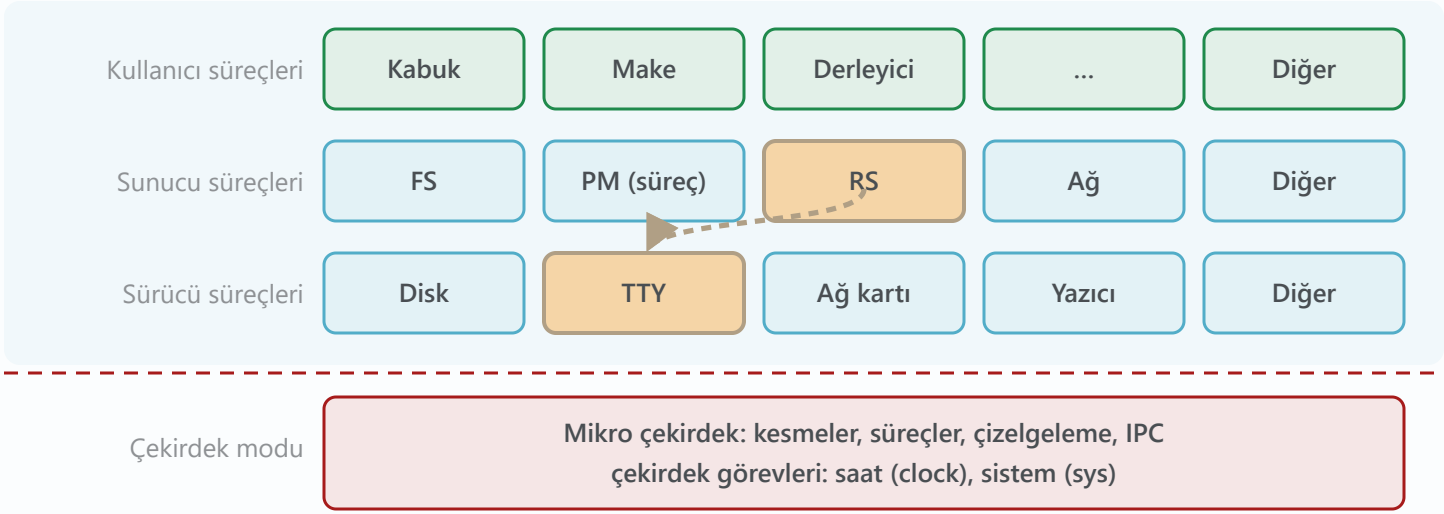
DEĞİŞKENLER

`m.sonuc = veri`

ÇAĞRI YIĞINI

`istemci`

MINIX 3: Bir Mikro Çekirdek Örneği



Reenkarnasyon sunucusu (RS)

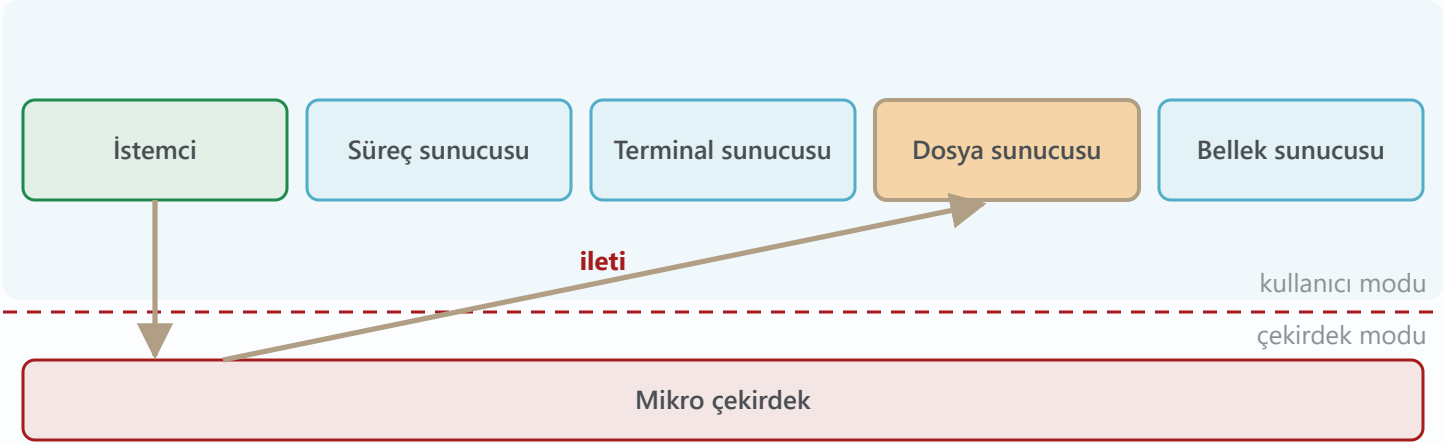
Sürücülerin sağlığını denetler; çöken bir sürücüyü (ör. TTY) kullanıcı fark etmeden yeniden başlatır: *kendini onaran* sistem.

Bölünmüş ayrıcalık

Sürücü kullanıcı modundadır; G/Ç portlarına yalnızca çekirdek çağrılarıyla, izin verilen ölçüde erişir. Hatalı bir sürücü belleği bozamaz.

YAPILAR

İstemci–Sunucu Modeli



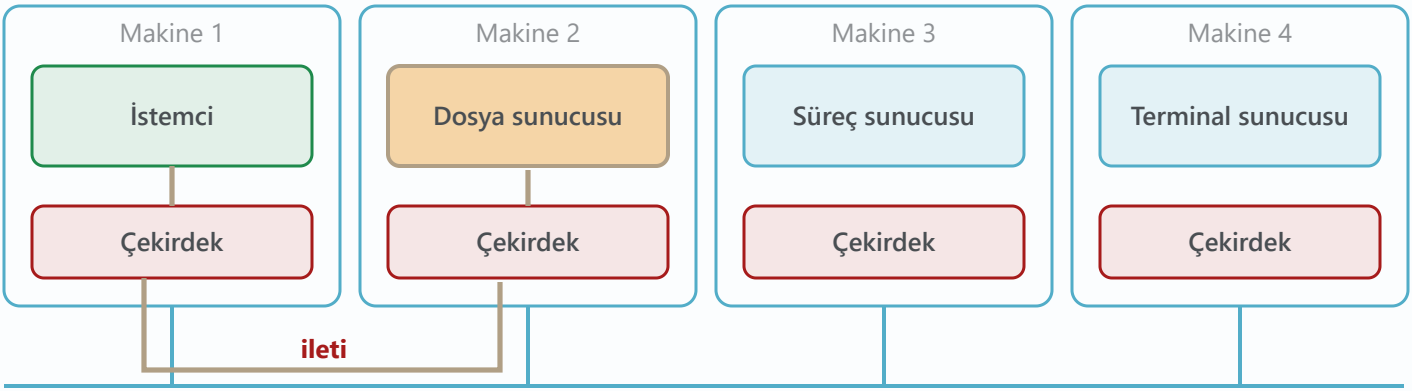
Mikro çekirdek üzerinde süreçler iki sınıfa ayrılır: hizmet isteyen **istemciler** ve hizmet veren **sunucular**. Haberleşme yalnızca iletilerledir.

Adım 1 / 2

İleti alışverişi kendiliğinden güvenli değildir: güvenlik, çekirdeğin hangi sürecin hangi sunucuya ileti gönderebileceğini denetlemesine bağlıdır.

YAPILAR

İstemci–Sunucu Modeli



İstemci, iletinin yerelde mi yoksa ağdaki başka bir makinede mi işlendiğini bilmek zorunda değildir: model doğal olarak **dağıtık sistemlere** genişler (bugünün web ve mikroservis mimarileri).

Adım 2 / 2

İleti alışverişi kendiliğinden güvenli değildir: güvenlik, çekirdeğin hangi sürecin hangi sunucuya ileti gönderebileceğini denetlemesine bağlıdır.

Tanenbaum – Torvalds Tartışması (1992)

Andrew Tanenbaum: mikro çekirdek

- "LINUX is obsolete" başlıklı iletisinde monolitik tasarımı 1970'lerin geride kalmış bir yaklaşımı olarak gördü.
- Mikro çekirdek daha modüler, esnek, taşınabilir; bir bileşendeki hatanın verdiği zarar sınırlı.
- Donanım hızlandıkça ileti alışverişinin ek yükü önemsizleşir.

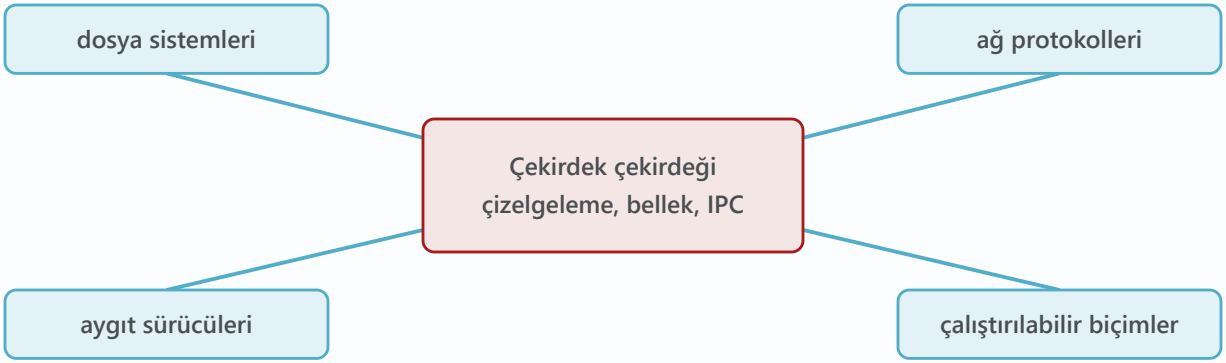
Linus Torvalds: monolitik

- Monolitik çekirdeğin tasarımı ve gerçeklemesi daha basit; doğrudan fonksiyon çağrıları daha hızlı.
- Süreçler arası ileti alışverişine gerek yok; pratikte *çalışan* kod önemli.
- Açık kaynak geliştirme modeli Linux'un hızla gelişmesini sağladı.

Bugün iki taraf da kısmen haklı çıktı: Linux monolitik kaldı ama yüklenebilir modüller, kullanıcı alanı sürücüleri (FUSE, VFIO) ve eBPF ile esneklik kazandı; mikro çekirdekler ise güvenlik-kritik ve gömülü alanda (QNX, seL4) yaygınlaştı.

Modüler Yapı: Yüklenebilir Çekirdek Modülleri

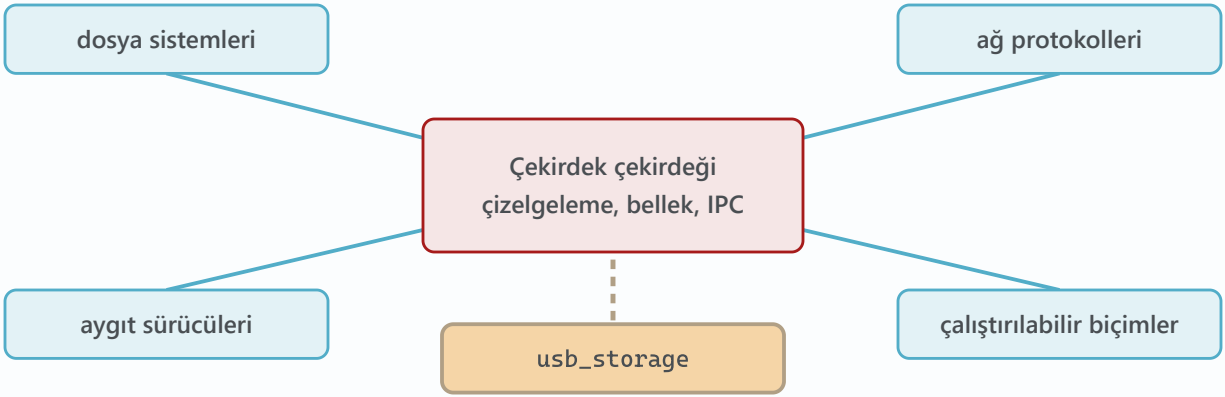
- Çekirdekte yalnızca temel bileşenler bulunur; diğer hizmetler açılışta veya **çalışma anında modül** olarak yüklenir (Linux LKM, FreeBSD, Solaris).
- Katmanlıya benzer (tanımlı arayüzler) ama daha esnektir: her modül diğerini çağırabilir. Mikro çekirdeğe benzer ama ileti gerekmez, modüller **çekirdek modunda** çalışır.



Açılışta çekirdek ve initramfs'teki gerekli modüller yüklüdür. `lsmod` yüklü modülleri listeler.

Modüler Yapı: Yüklenebilir Çekirdek Modülleri

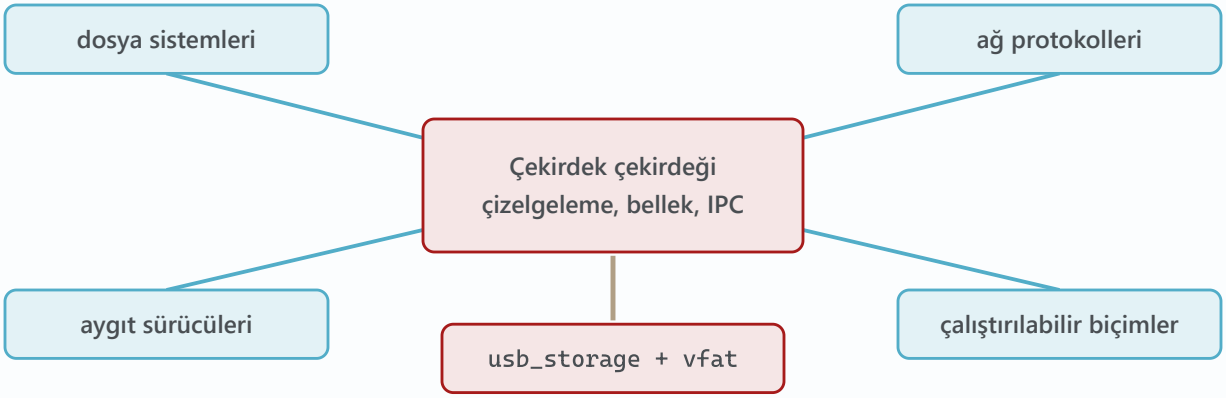
- Çekirdekte yalnızca temel bileşenler bulunur; diğer hizmetler açılışta veya **çalışma anında modül** olarak yüklenir (Linux LKM, FreeBSD, Solaris).
- Katmanlıya benzer (tanımlı arayüzler) ama daha esnektir: her modül diğerini çağırabilir. Mikro çekirdeğe benzer ama ileti gerekmez, modüller **çekirdek modunda** çalışır.



USB bellek takılır: udev olayı `modprobe usb-storage` ile sürücüyü (bağımlılıklarıyla) yükletir; `vfat` modülü USB bellek bağlanırken yüklenir.

Modüler Yapı: Yüklenebilir Çekirdek Modülleri

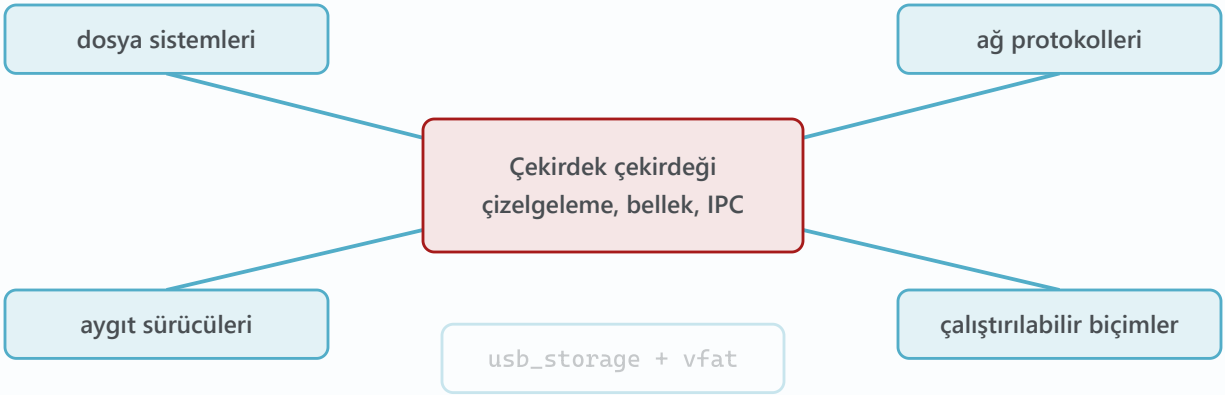
- Çekirdekte yalnızca temel bileşenler bulunur; diğer hizmetler açılışta veya **çalışma anında modül** olarak yüklenir (Linux LKM, FreeBSD, Solaris).
- Katmanlıya benzer (tanımlı arayüzler) ama daha esnektir: her modül diğerini çağırabilir. Mikro çekirdeğe benzer ama ileti gerekmez, modüller **çekirdek modunda** çalışır.



Modül kendini çekirdeğe kaydeder ve doğrudan fonksiyon çağrılarıyla çalışır: hızlı, ama **izolasyon yoktur**; hatalı modül çekirdeği çökertebilir.

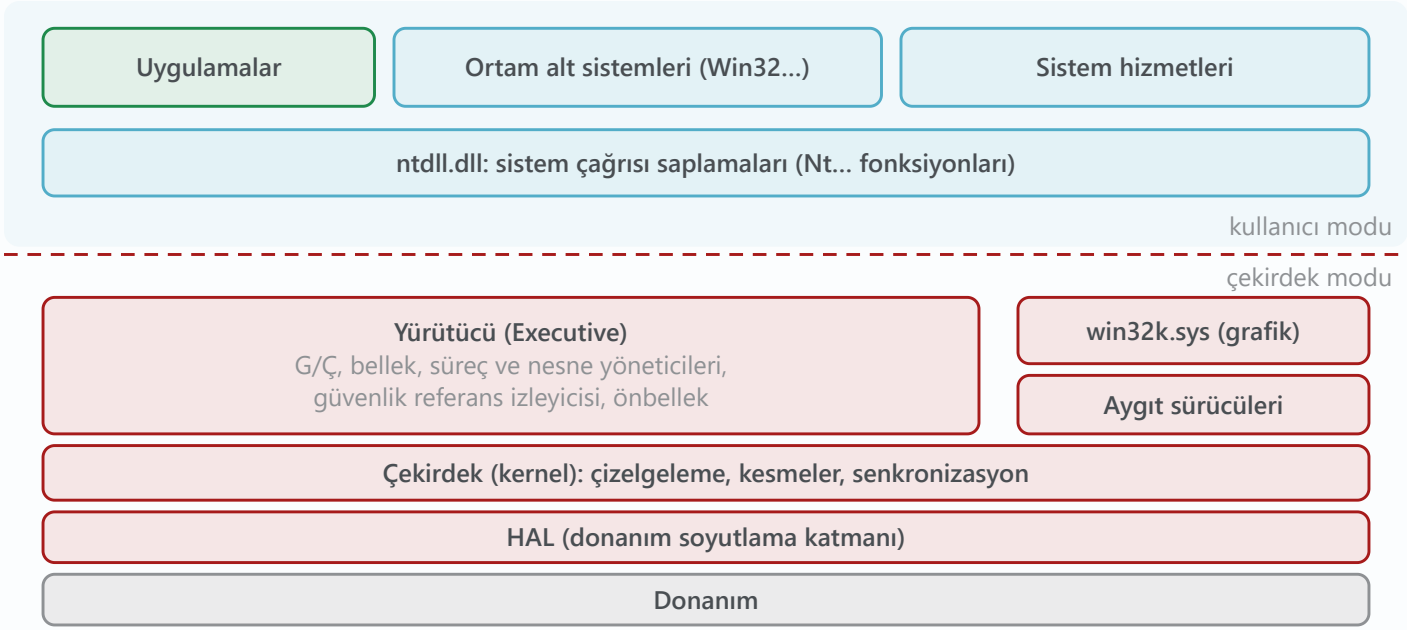
Modüler Yapı: Yüklenebilir Çekirdek Modülleri

- Çekirdekte yalnızca temel bileşenler bulunur; diğer hizmetler açılışta veya **çalışma anında modül** olarak yüklenir (Linux LKM, FreeBSD, Solaris).
- Katmanlıya benzer (tanımlı arayüzler) ama daha esnektir: her modül diğerini çağırabilir. Mikro çekirdeğe benzer ama ileti gerekmez, modüller **çekirdek modunda** çalışır.



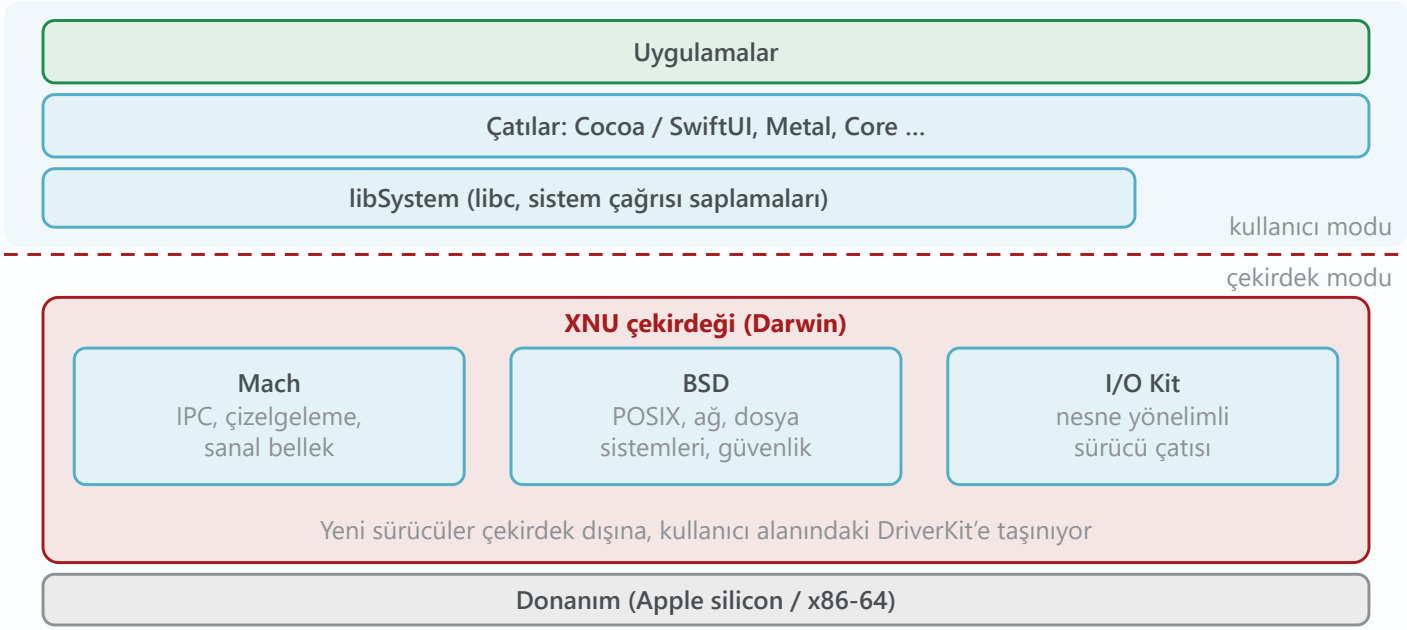
Kullanım sayısı 0 olunca `modprobe -r` (veya `rmmod`) modülü bellekten çıkarır. Modüllerin çekirdek sürümüyle uyumlu olması gerekir.

Hibrit Çekirdek: Windows ve macOS



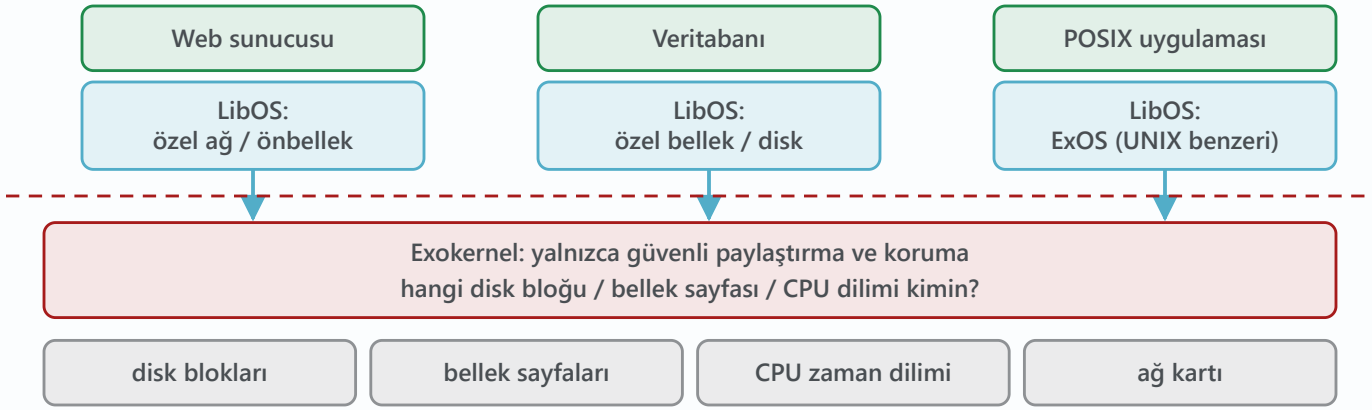
Windows NT: mikro çekirdek fikirleri (alt sistemler, nesne/ileti temelli tasarım) korunmuş, ama performans için yürütücü, grafik ve sürücüler çekirdek moduna alınmıştır.

Hibrit Çekirdek: Windows ve macOS



macOS / iOS XNU: Mach mikro çekirdeği ile BSD katmanı aynı çekirdek adres alanında birleşir: mikro çekirdeğin modülerliği + monolitik hız.

Exo-Çekirdek (Exokernel)



- MIT'de Engler ve Kaashoek'un araştırma mimarisi (1995): çekirdek **Aegis**, kütüphane işletim sistemi **ExOS**. Ticari bir işletim sistemi değildir.
- VM makinenin **kopyasını** alır; exokernel kaynağın bir **bölümünü** verip sahipliği denetler, yeniden eşleme katmanı gerekmez.
- **Yönetimi korumadan ayırır:** çekirdek yalnızca kaynakların kime ait olduğunu izler; soyutlamaları (dosya, sanal bellek) uygulama düzeyindeki **LibOS**'lar sağlar.
- Uygulamaya özgü bellek yönetimi ve önbellekleme ile yüksek başarım; ama uygulama geliştirmek karmaşıktır. Fikir bugün **unikernel**'lerde ve DPDK / SPDK gibi çekirdeği atlayan G/Ç'de yaşar.

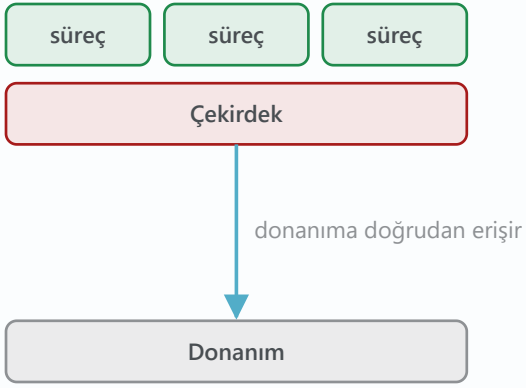
Yapıların Karşılaştırılması

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

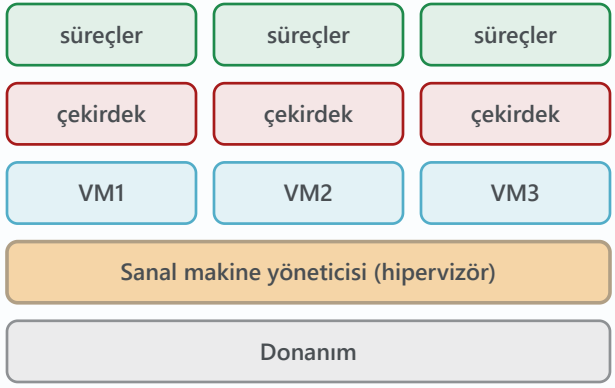
Gerçek sistemler saf değildir: Linux monolitik + modüler, Windows ve macOS hibrittir; hemen hepsi bir ölçüde katmanlıdır.

Sanal Makine Kavramı

- Hipervizör (VMM), tek bir fiziksel donanım üzerinde her biri **bağımsız bir bilgisayar gibi görünen** sanal makineler oluşturur.
- Her misafir işletim sistemi kendi CPU'su, RAM'i, diski ve G/Ç aygıtları olduğuna inanır; farklı işletim sistemleri aynı donanımda eş zamanlı çalışabilir.
- Fikir IBM VM/370'e (1972) dayanır; bugün bulut bilişimin temelidir: sunucu birleştirme, yalıtım, anlık görüntü, canlı taşıma.
- Hipervizör donanımda (**tip 1**) ya da bir ana işletim sistemi üzerinde (**tip 2**) çalışır. **Konteyner** ayrı çekirdek taşımaz, aynı çekirdeği paylaşan süreçleri yalıtır.

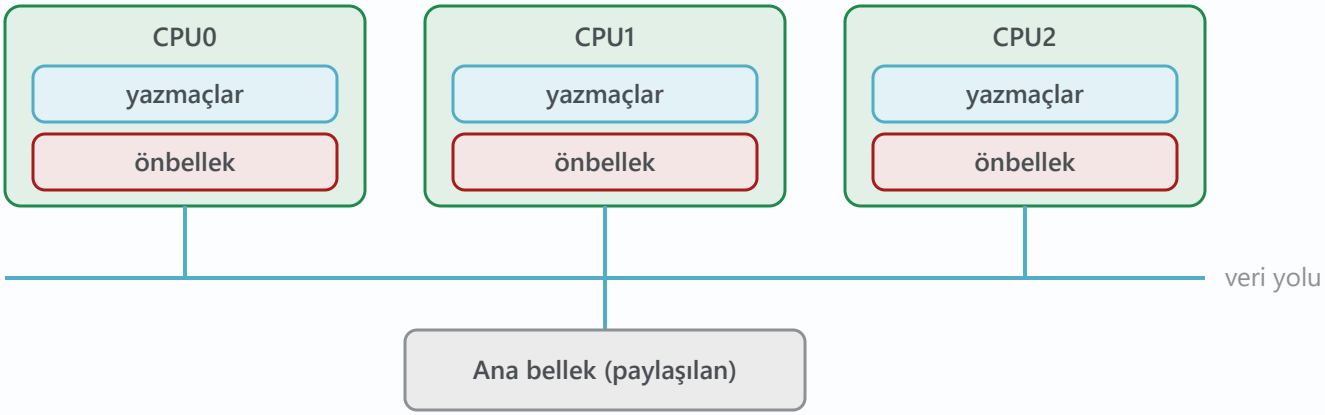


(a) Sanal olmayan makine



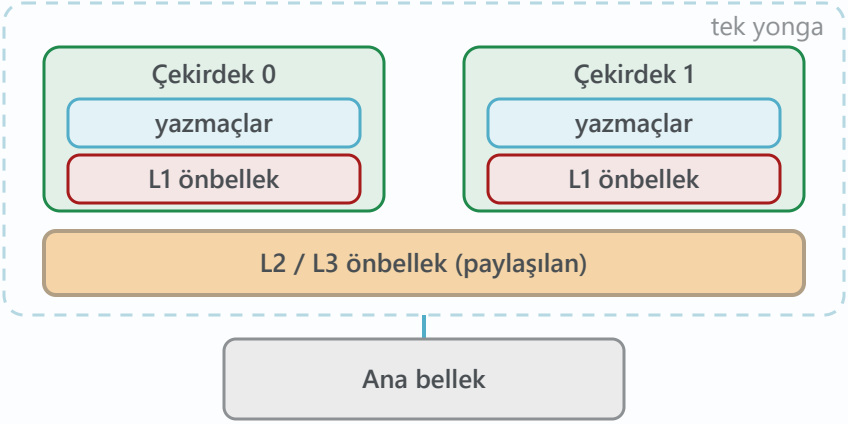
(b) Sanal makineler

Simetrik Çok İşlemcili ve Çok Çekirdekli Mimari



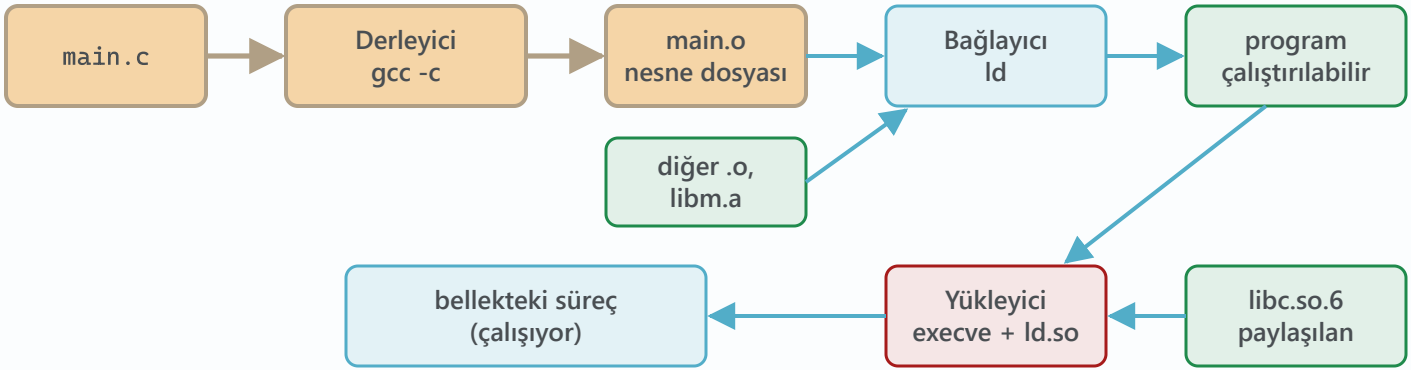
SMP: her CPU kendi yazmaçlarına ve önbelleğine sahiptir, belleği paylaşır; hepsi işletim sistemi dâhil her işi yapabilir. Çekirdek veri yapıları kilitlerle korunmalıdır (Linux'un "büyük çekirdek kilidi" 2011'de tamamen kaldırıldı).

Simetrik Çok İşlemcili ve Çok Çekirdekli Mimari



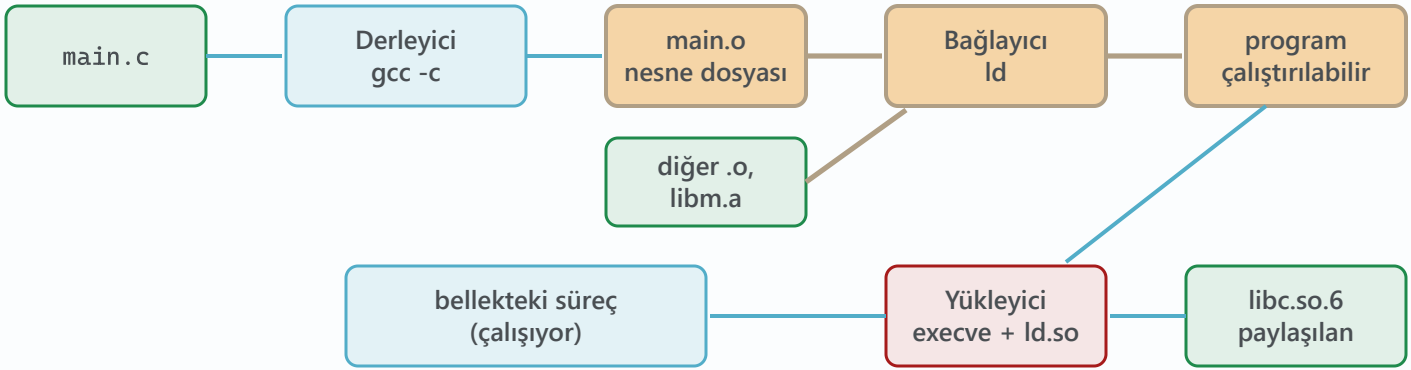
Çok çekirdekli yonga: çekirdekler aynı yongada, yakın önbellek paylaşımıyla daha hızlı ve az enerjiyle haberleşir. 2026'da yaygın olan P/E (performans/verimlilik) çekirdekleri çizelgeleyiciye yeni görevler yükler.

Derleme, Bağlama ve Yükleme



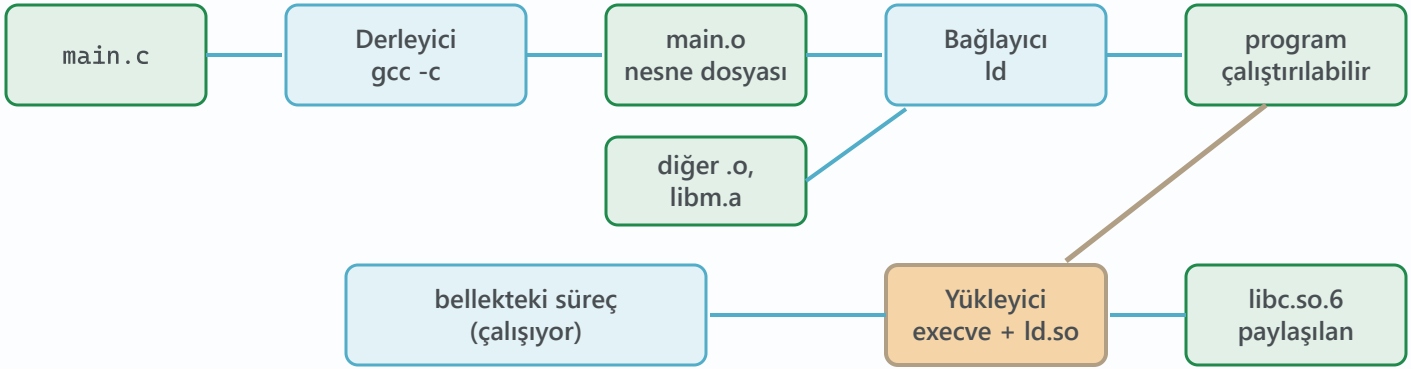
Derleyici kaynak kodu **yeniden konumlandırılabilir nesne dosyasına** çevirir: adresler henüz kesin değildir, dış semboller (ör. `printf`) çözülmemiştir.

Derleme, Bağlama ve Yükleme



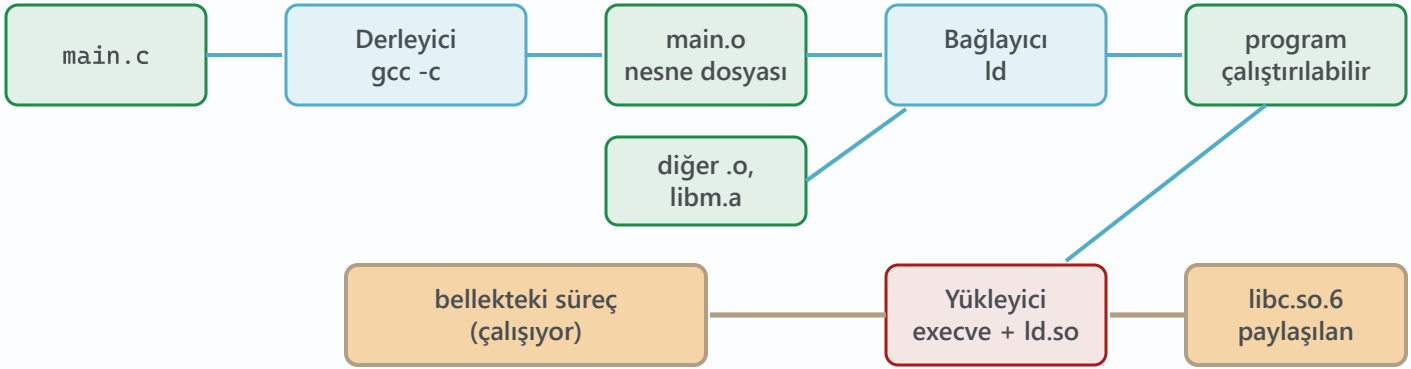
Bağlayıcı nesne dosyalarını ve statik kütüphaneleri birleştirir, sembolleri çözer, adresleri düzeltir ve çalıştırılabilir dosya üretir.

Derleme, Bağlama ve Yükleme



Kabuk `execve` çağırır; çekirdekteki **yükleyici** dosyanın başlığını okur, kesimleri adres alanına eşler ve dinamik bağlayıcıyı (`ld-linux.so`) başlatır.

Derleme, Bağlama ve Yükleme

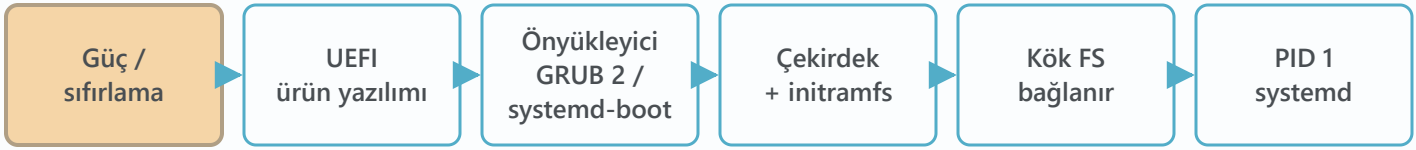


Paylaşımlı kütüphaneler (`libc.so.6`) çalışma anında bağlanır ve süreçler arasında paylaşılır. Biçimler işletim sistemine özgüdür: ELF (Linux), PE (Windows), Mach-O (macOS); bu yüzden programlar işletim sistemleri arasında doğrudan taşınamaz.

BAĞLAMA VE AÇILIŞ

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir. Eski BIOS (1981, 16 bit gerçek mod) diskin ilk sektöründeki MBR kodunu çalıştırırdı; UEFI sektör değil dosya (`.efi`) yükler. Linux'ta `systemd-analyze` her aşamanın süresini gösterir.



CPU sabit bir adresten ürün yazılımını çalıştırmaya başlar.

Güç verilince CPU sıfırlanır ve donanımın belirlediği **sabit bir adresten** flash bellekteki ürün yazılımını yürütmeye başlar.

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir. Eski BIOS (1981, 16 bit gerçek mod) diskin ilk sektöründeki MBR kodunu çalıştırırdı; UEFI sektör değil dosya (`.efi`) yükler. Linux'ta `systemd-analyze` her aşamanın süresini gösterir.

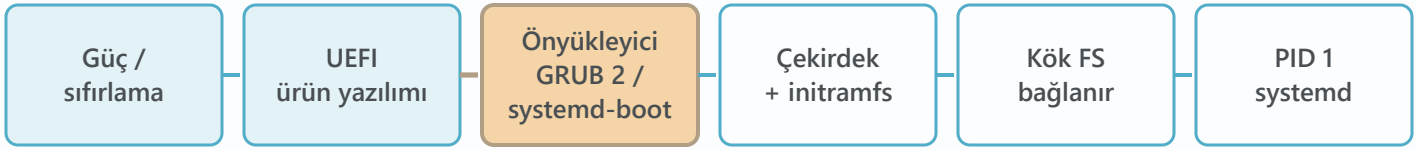


POST ile donanım sınanır ve ilklenir; EFI sistem bölümündeki önyükleyici bulunur.
Secure Boot her aşamanın imzasını doğrular.

UEFI açılış öz-sınamasıyla (**POST**) belleği ve aygıtları dener; NVRAM'deki sıraya göre (NVMe, USB, ağ/PXE) GPT diskteki EFI sistem bölümünden (FAT32) önyükleyiciyi yükler. Güvenli Önyükleme açıksa imzası doğrulanır.

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir. Eski BIOS (1981, 16 bit gerçek mod) diskin ilk sektöründeki MBR kodunu çalıştırırdı; UEFI sektör değil dosya (`.efi`) yükler. Linux'ta `systemd-analyze` her aşamanın süresini gösterir.



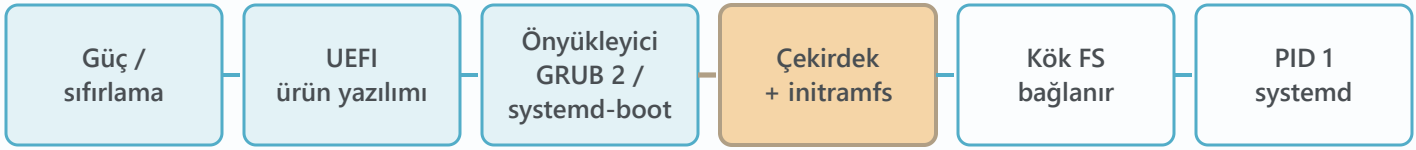
Önyükleyici çekirdek seçimini sunar, çekirdeği ve initramfs'i belleğe yükler.
Eski sistemler: BIOS + MBR.

Önyükleyici menü sunar; seçilen çekirdeği ve başlangıç RAM diskini (initramfs) belleğe koyar ve çekirdeğe atlar.

BAĞLAMA VE AÇILIŞ

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir. Eski BIOS (1981, 16 bit gerçek mod) diskin ilk sektöründeki MBR kodunu çalıştırırdı; UEFI sektör değil dosya (`.efi`) yükler. Linux'ta `systemd-analyze` her aşamanın süresini gösterir.

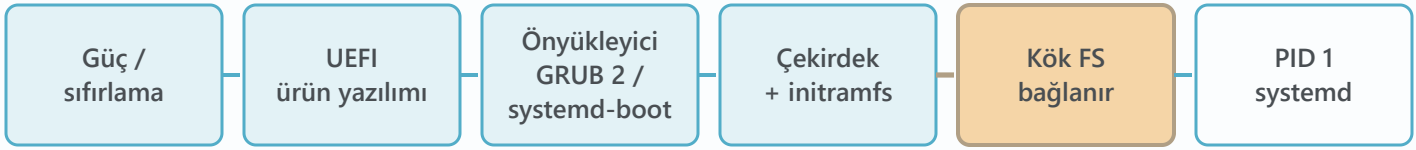


Çekirdek kendini açar, donanımı tanır, sürücüleri yükler;
initramfs geçici bir kök dosya sistemi sağlar.

Çekirdek bellek yönetimini, çizelgeleyiciyi ve sürücüleri başlatır; initramfs içindeki betikler gerekli modülleri yükler.

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir. Eski BIOS (1981, 16 bit gerçek mod) diskin ilk sektöründeki MBR kodunu çalıştırırdı; UEFI sektör değil dosya (`.efi`) yükler. Linux'ta `systemd-analyze` her aşamanın süresini gösterir.



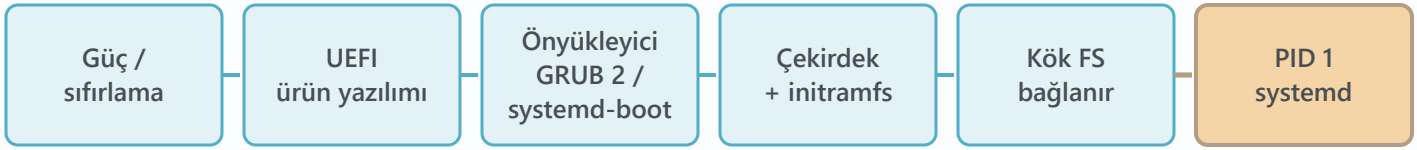
Gerçek kök dosya sistemi (ör. şifreli disk açılarak) bağlanır.

Gerçek kök dosya sistemi bağlanır ve initramfs bırakılır.

BAĞLAMA VE AÇILIŞ

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir. Eski BIOS (1981, 16 bit gerçek mod) diskin ilk sektöründeki MBR kodunu çalıştırırdı; UEFI sektör değil dosya (`.efi`) yükler. Linux'ta `systemd-analyze` her aşamanın süresini gösterir.



İlk kullanıcı süreci başlar; hizmetleri paralel başlatır ve giriş ekranını açar.

Çekirdek PID 1'i (systemd) başlatır; o da hizmetleri, ağı ve giriş ekranını açar. Bundan sonra her süreç PID 1'in soyundandır. Windows'ta karşılığı: winload.efi → ntoskrnl.exe → smss.exe → winlogon.exe.

ÖZET

Kontrol Soruları

1. Kullanıcı modunda çalışan bir program neden G/Ç komutlarını doğrudan çalıştıramaz? Denerse ne olur?
2. `printf` ile `write` arasındaki fark nedir? Hangisi sistem çağrısıdır?
3. `fork()` neden "bir kez çağrılır, iki kez döner"? Ebeveynde ve çocukta dönüş değerleri nedir?
4. Mikro çekirdekte bir sürücü çökerse ne olur? Monolitik çekirdekte?
5. Linux'un yüklenebilir modülleri onu mikro çekirdek yapar mı? Neden?
6. Açılışta çekirdeğin başlattığı ilk kullanıcı süreci hangisidir? Neden önemlidir?

▶ **Cevapları göster**

ÖZET

Kaynaklar

- Silberschatz, Galvin ve Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 (Bölüm 2 ve 18).
- Tanenbaum ve Bos, *Modern Operating Systems*, 4. baskı, Pearson, 2014 (Bölüm 1.5–1.7 ve 7).
- Arpaci-Dusseau ve Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), sürüm 1.10, pages.cs.wisc.edu/~remzi/OSTEP.
- Linux man sayfaları: `syscalls(2)`, `syscall(2)`, `fork(2)`, `pipe(7)`, `mount(8)`.
- Engler, Kaashoek ve O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management", SOSP 1995.

Son

Bölüm 2: İşletim Sistemi Yapıları

Sonraki bölüm: Süreçler · Terim sözlüğü