

Bölüm 2: İşletim Sistemi Yapıları

İşletim Sistemleri



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.

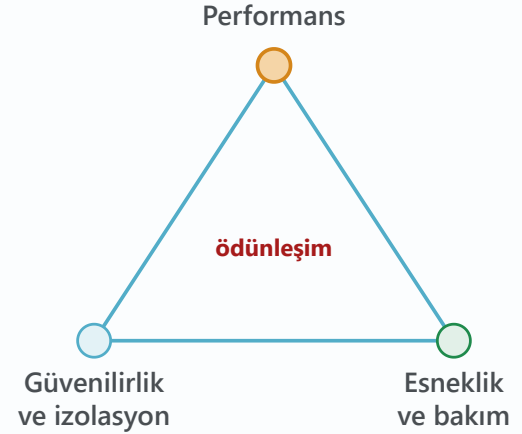
Sanallaştırma ve açılış

Hipervizör türlerini, konteynerleri, bağlayıcı/yükleyiciyi ve önyükleme sürecini tanımak.

GİRİŞ

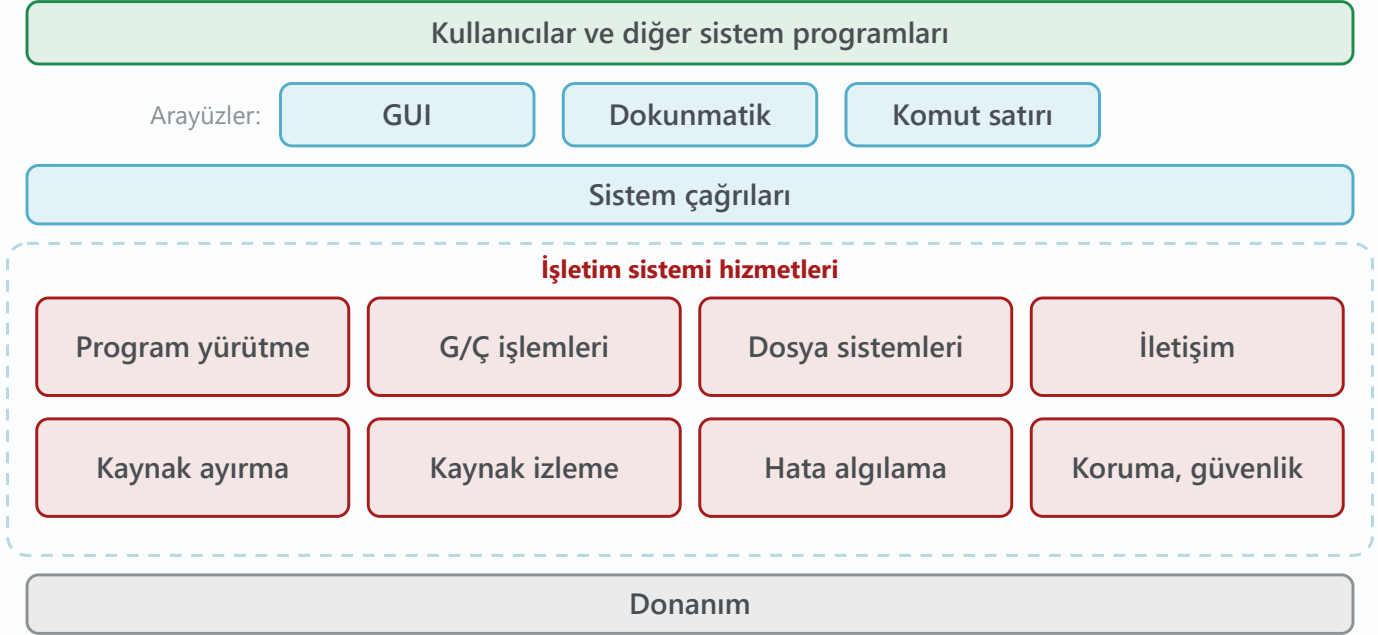
Neden Yapı Önemli?

- 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 ödüneşim 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 aygıtı 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.

Süreç tablosu

PID	Durum	Program
1	uyuyor	systemd
812	bekliyor	bash
1045	çalışıyor	ls
1046	hazır	grep



PCB (süreç denetim bloğu)

PID, durum, öncelik
PC, SP ve diğer yazmaçlar
bellek haritası (adres alanı)
açık dosyalar (fd tablosu)
sinyaller, alarmlar, UID, ebeveyn

Süreç Ağacı

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

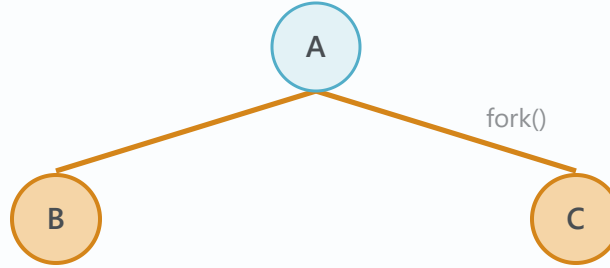


PID 1 (systemd) ağacın köküdür

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

Süreç Ağacı

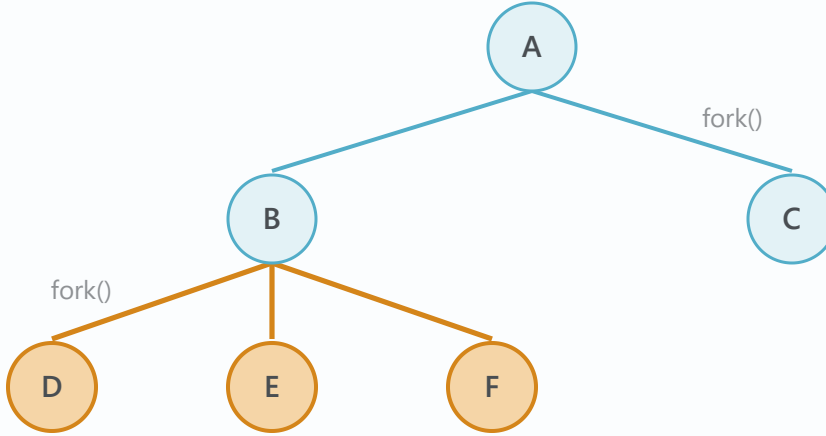
Bir süreç `fork()` ile **çocuk süreçler** oluşturabilir; çocuklar da kendi çocuklarını oluşturur. Sonuç bir ağaçtır.



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

Süreç Ağacı

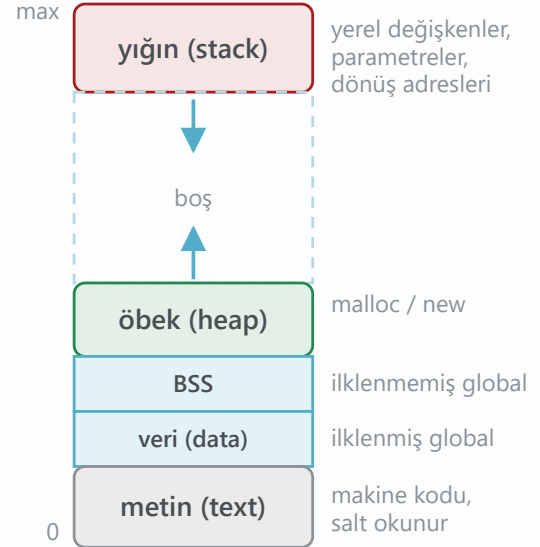
Bir süreç `fork()` ile **çocuk süreçler** oluşturabilir; çocuklar da kendi çocuklarını oluşturur. Sonuç bir ağaçtır.



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

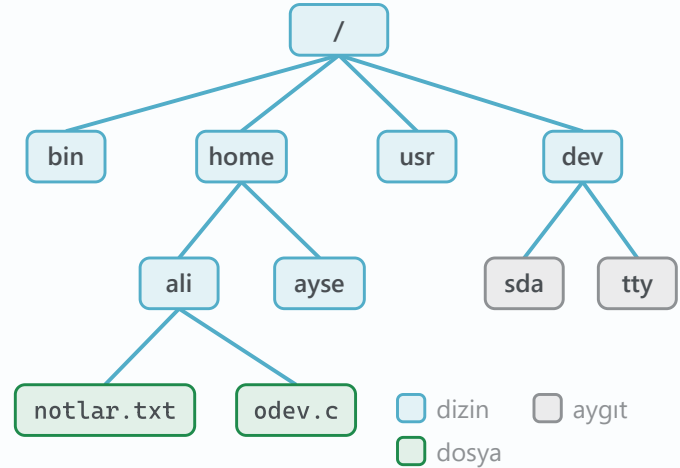
Adres Alanı ve Bellek Kesimleri

- **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. **Öbek:** 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 (Bölüm 9).

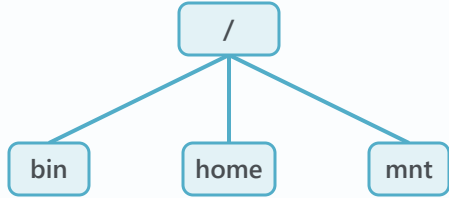


Dosyalar ve Dizin Hiyerarşisi

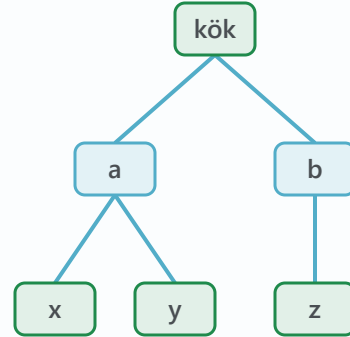
- **Dosya:** verileri kalıcı olarak saklayan adlandırılmış birim (metin, resim, video, program...).
- İşletim sistemi disk bloklarını ve aygıt ayrıntılarını gizler; temiz bir soyutlama sunar.
- Dosyalar **dizin ağacında** tutulur. Yol adı mutlak (`/home/ali/notlar.txt`) veya çalışma dizinine göre görelî (`ali/notlar.txt`) olabilir.
- Açılan dosyaya küçük bir tamsayı, **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)



?

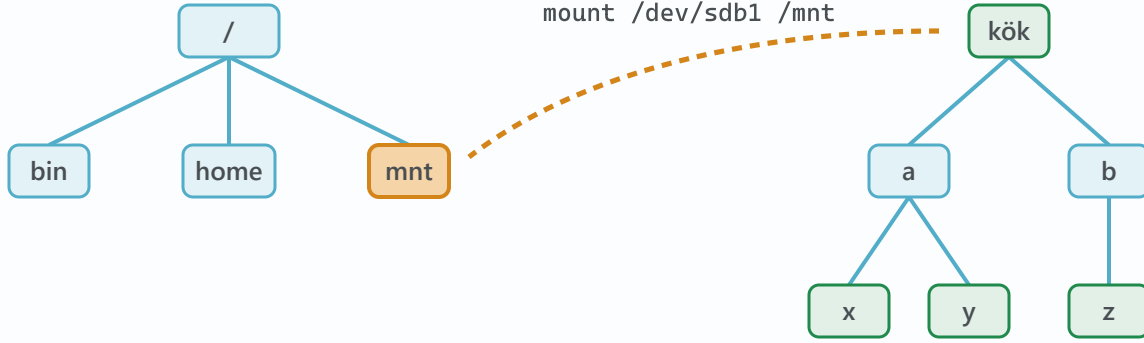


Ana dosya sistemi (/dev/nvme0n1p2)

USB bellek (/dev/sdb1)

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

Dosya Sistemini Bağlama (mount)

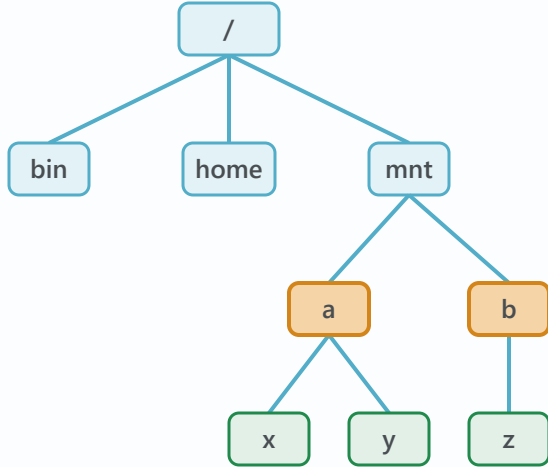


Ana dosya sistemi (/dev/nvme0n1p2)

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.

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"
```

```
ls -l  
stdout → fd[1]
```

```
grep notlar  
stdin ← fd[0]
```

yazma ucu

okuma ucu

boru tamponu (64 KiB)

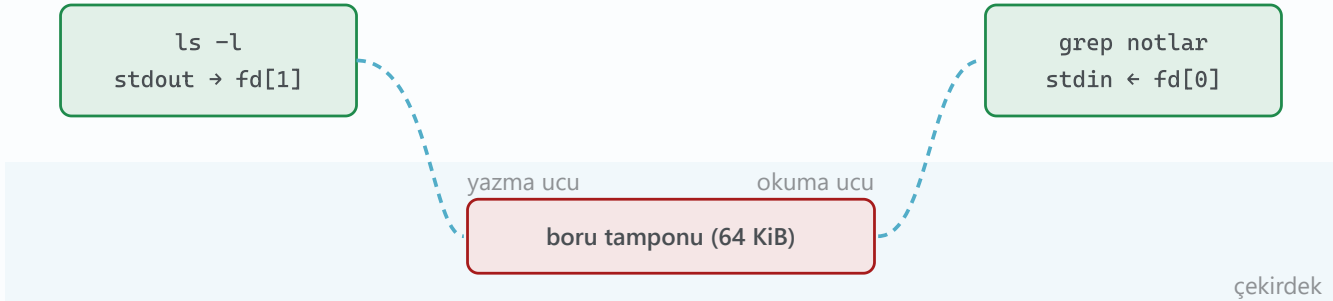
çekirdek

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"
```

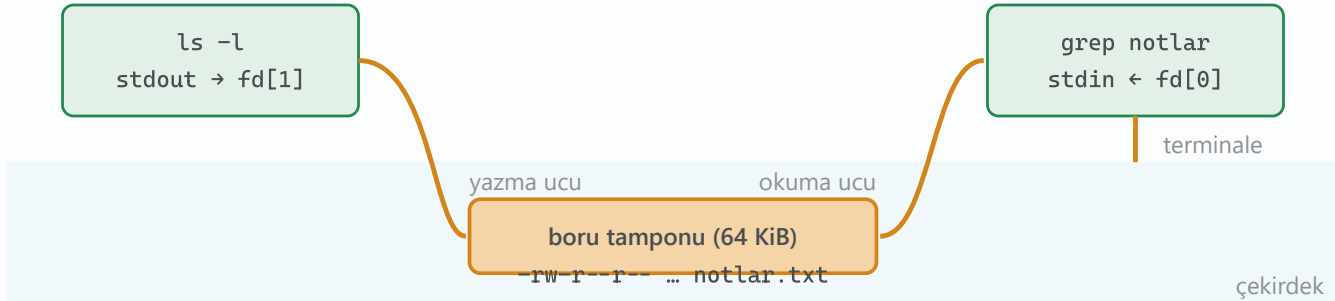


Kabuk iki kez `fork` eder; `dup2` ile `ls`'in `stdout`'u yazma ucuna, `grep`'in `stdin`'i okuma ucuna yönlendirilir.

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"
```

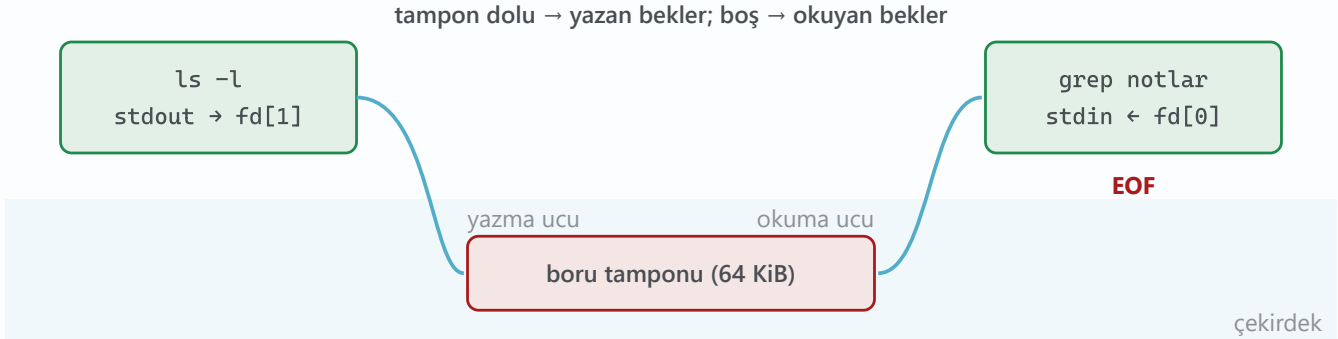


İs listeyi boruya yazar, grep okur ve yalnızca "notlar" içeren satırları terminale yazar.

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.

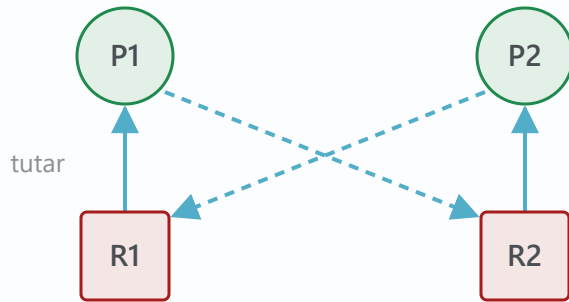
```
$ ls -l | grep "notlar"
```



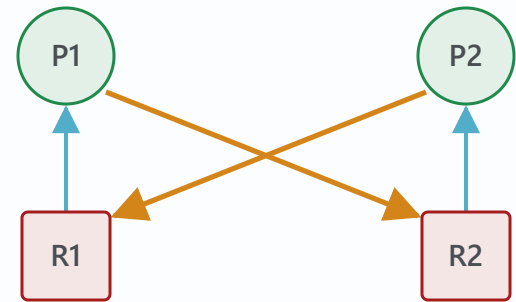
Akış denetimi çekirdekte: tampon dolunca yazan, boşken okuyan bloke olur. `ls` bitip yazma ucu kapanınca `grep` dosya sonu (EOF) görür.

Kilitlenme (Deadlock)

- Kümedeki her süreç, **başka bir sürecin tuttuğu** kaynağı beklerse hiçbiri ilerleyemez: kilitlenme.
- Trafikte dört yönden kavşağa girip birbirini bekleyen araçlar gibi. Ayrıntılar Bölüm 7'de.



(a) Potansiyel: istekler henüz yapılmadı

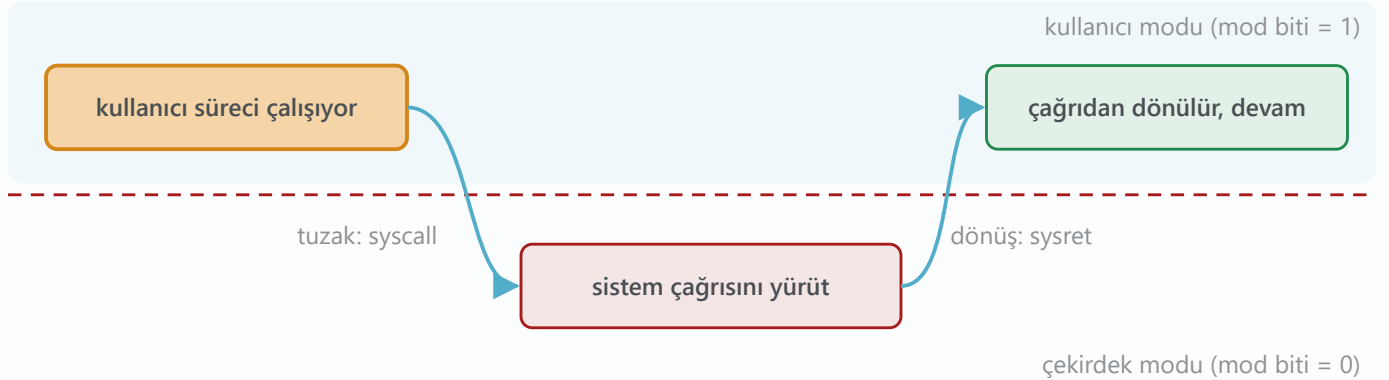


(b) Gerçekleşmiş: döngü $P1 \rightarrow R2 \rightarrow P2 \rightarrow R1 \rightarrow P1$

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

Çift Mod: Kullanıcı ve Çekirdek

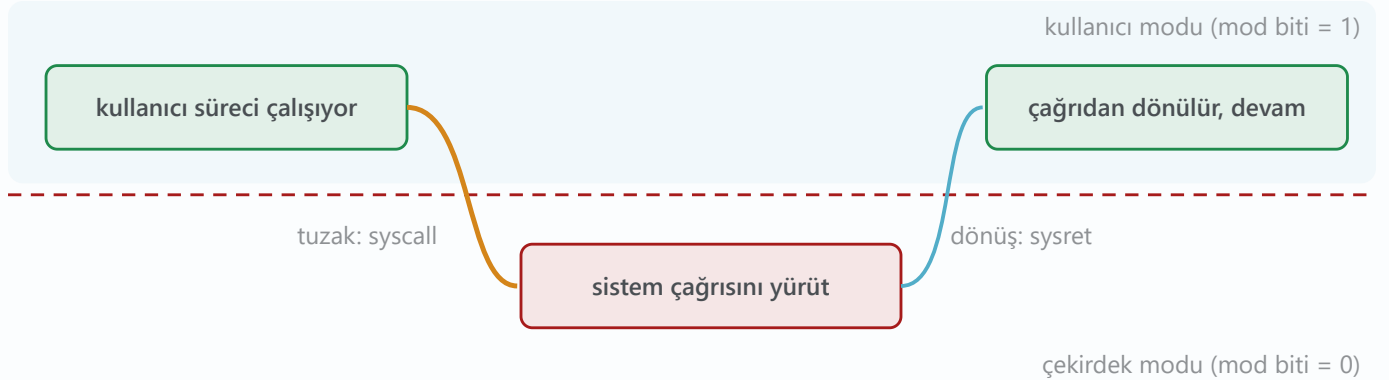
- CPU'daki **mod biti** hangi modda çalışıldığını gösterir. G/Ç, zamanlayıcıyı ayarlama, kesmeleri kapatma, sayfa tablolarını değiştirme gibi **ayrıcalıklı komutlar** yalnızca çekirdek modunda çalışır.
- Kullanıcı modunda ayrıcalıklı komut denirse donanım tuzak üretir, işletim sistemi süreci sonlandırır. x86'da halkalar 0–3 (Linux ve Windows yalnızca 0 ve 3'ü kullanır), ARM'da EL0–EL3; sanallaştırma için ek bir mod vardır.



Kullanıcı süreci kullanıcı modunda kendi kodunu çalıştırır.

Çift Mod: Kullanıcı ve Çekirdek

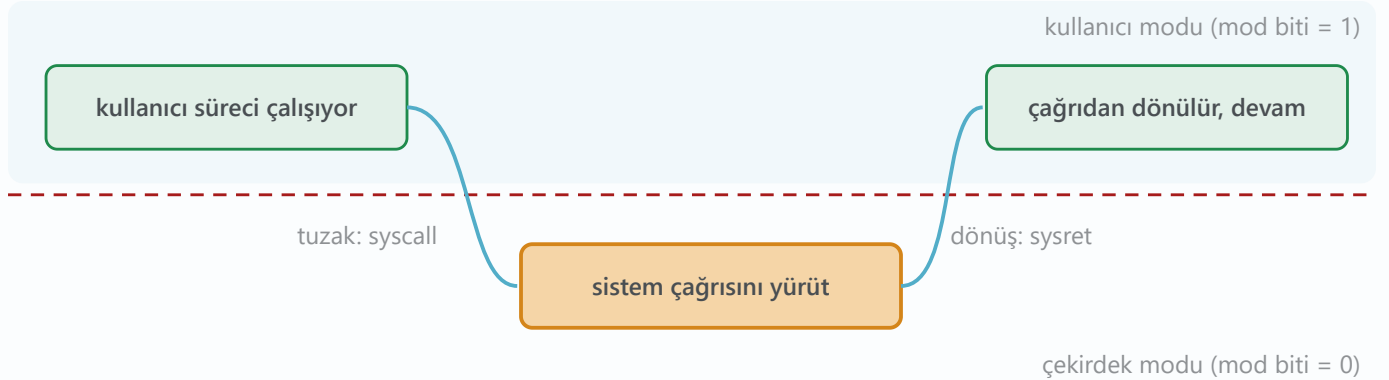
- CPU'daki **mod biti** hangi modda çalışıldığını gösterir. G/Ç, zamanlayıcıyı ayarlama, kesmeleri kapatma, sayfa tablolarını değiştirme gibi **ayrıcalıklı komutlar** yalnızca çekirdek modunda çalışır.
- Kullanıcı modunda ayrıcalıklı komut denenirse donanım tuzak üretir, işletim sistemi süreci sonlandırır. x86'da halkalar 0–3 (Linux ve Windows yalnızca 0 ve 3'ü kullanır), ARM'da EL0–EL3; sanallaştırma için ek bir mod vardır.



Bir sistem çağrısı (veya kesme / istisna) tuzak üretir: mod biti 0 olur, CPU çekirdekteki sabit giriş noktasına atlar.

Çift Mod: Kullanıcı ve Çekirdek

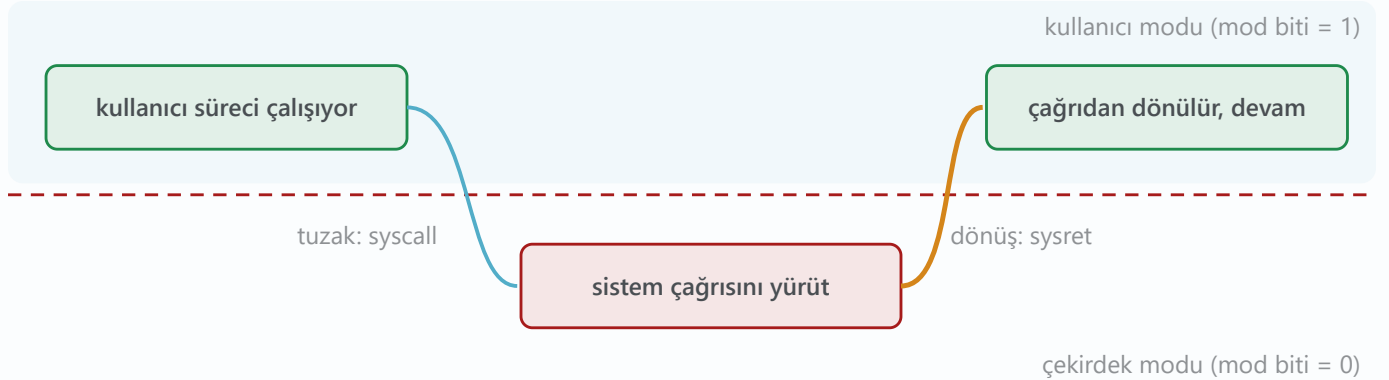
- CPU'daki **mod biti** hangi modda çalışıldığını gösterir. G/Ç, zamanlayıcıyı ayarlama, kesmeleri kapatma, sayfa tablolarını değiştirme gibi **ayrıcalıklı komutlar** yalnızca çekirdek modunda çalışır.
- Kullanıcı modunda ayrıcalıklı komut denenirse donanım tuzak üretir, işletim sistemi süreci sonlandırır. x86'da halkalar 0–3 (Linux ve Windows yalnızca 0 ve 3'ü kullanır), ARM'da EL0–EL3; sanallaştırma için ek bir mod vardır.



Çekirdek, istenen hizmeti tam ayrıcalıkla yürütür.

Çift Mod: Kullanıcı ve Çekirdek

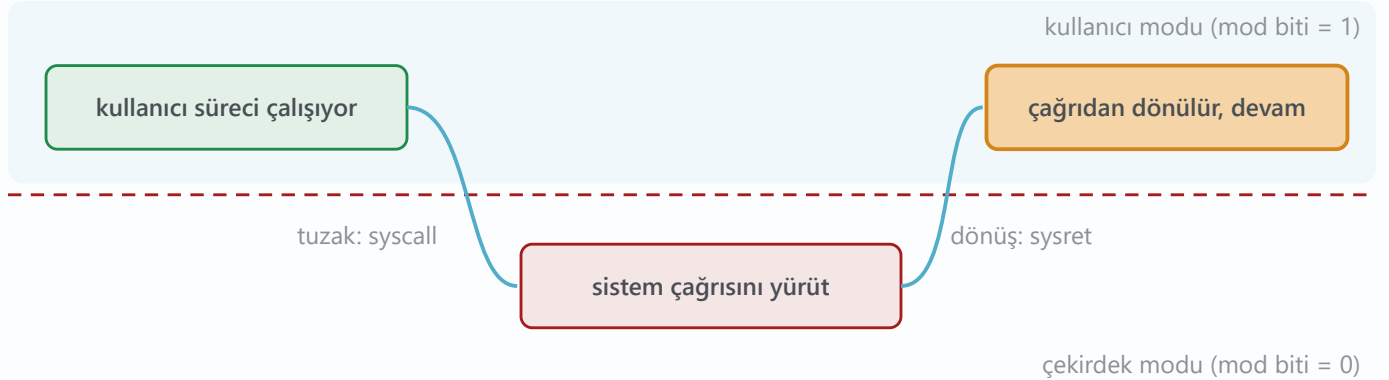
- CPU'daki **mod biti** hangi modda çalışıldığını gösterir. G/Ç, zamanlayıcıyı ayarlama, kesmeleri kapatma, sayfa tablolarını değiştirme gibi **ayrıcalıklı komutlar** yalnızca çekirdek modunda çalışır.
- Kullanıcı modunda ayrıcalıklı komut denenirse donanım tuzak üretir, işletim sistemi süreci sonlandırır. x86'da halkalar 0–3 (Linux ve Windows yalnızca 0 ve 3'ü kullanır), ARM'da EL0–EL3; sanallaştırma için ek bir mod vardır.



İş bitince özel dönüş komutu mod bitini 1 yapar ve kullanıcı koduna döner.

Çift Mod: Kullanıcı ve Çekirdek

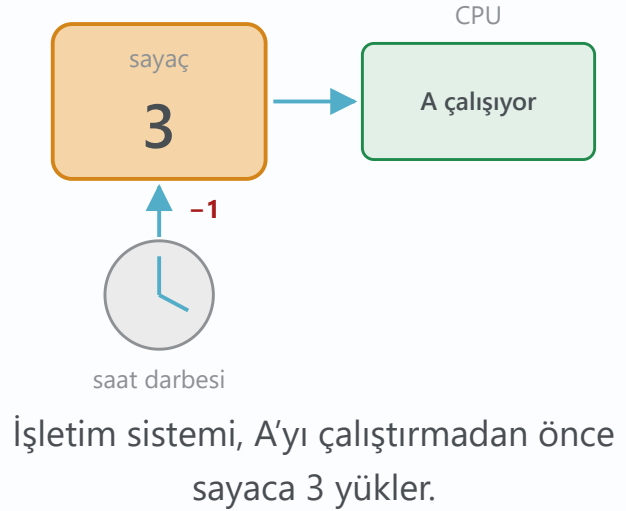
- CPU'daki **mod biti** hangi modda çalışıldığını gösterir. G/Ç, zamanlayıcıyı ayarlama, kesmeleri kapatma, sayfa tablolarını değiştirme gibi **ayrıcalıklı komutlar** yalnızca çekirdek modunda çalışır.
- Kullanıcı modunda ayrıcalıklı komut denenirse donanım tuzak üretir, işletim sistemi süreci sonlandırır. x86'da halkalar 0–3 (Linux ve Windows yalnızca 0 ve 3'ü kullanır), ARM'da EL0–EL3; sanallaştırma için ek bir mod vardır.



Süreç, sistem çağrısından sonraki komuttan devam eder; sonuç bir yazmaçtır.

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`) aşılsa söz konusudur.
- Linux'ta düzenli tik 100–1000 Hz'dir; boşta tik üretmeyen *tickless* (NO_HZ) mod enerji kazandırır.



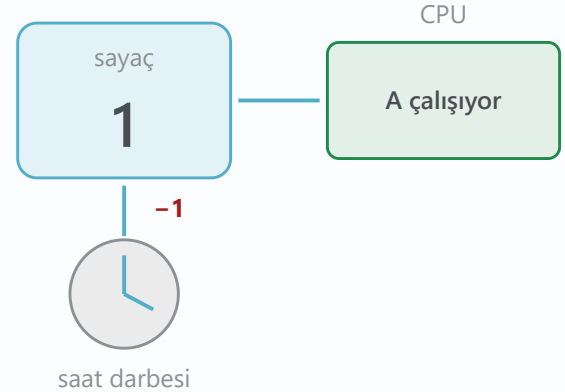
Zamanlayıcı (Timer)

- 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`) aşılsa söz konusudur.
- Linux'ta düzenli tik 100–1000 Hz'dir; boşta tik üretmeyen *tickless* (NO_HZ) mod enerji kazandırır.



Zamanlayıcı (Timer)

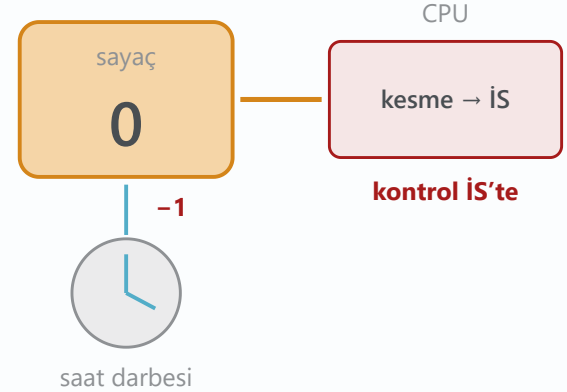
- Amaç: bir sürecin sonsuz döngüye girip CPU'yu **hiç bırakmamasını** önlemek.
- İşletim sistemi sayaca bir değer yükler; bu, **ayrıcalıklı** bir komuttur.
- Fiziksel saatin her darbesinde sayaç azalır; **sıfır olunca kesme** üretilir ve işletim sistemi kontrolü yeniden kazanır.
- İşletim sistemi, zaman dilimini dolduran süreci keser ve CPU'yu başka sürece verir (kesmeli çizelgeleme). Süreci sonlandırmak ancak CPU süresi sınırı (ör. `RLIMIT_CPU`) aşılsa 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)

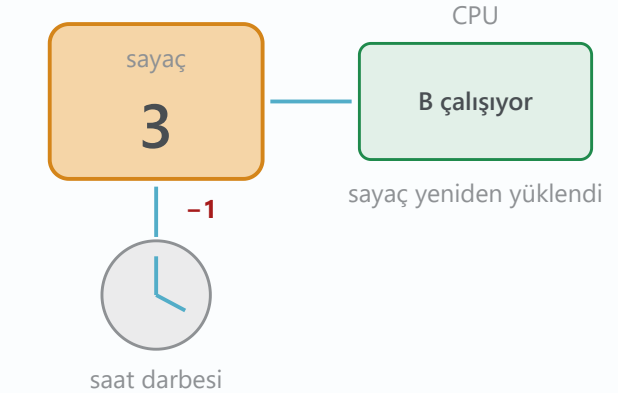
- Amaç: bir sürecin sonsuz döngüye girip CPU'yu **hiç bırakmamasını** önlemek.
- İşletim sistemi sayaca bir değer yükler; bu, **ayrıcılık** bir komuttur.
- Fiziksel saatin her darbesinde sayaç azalır; **sıfır olunca kesme** üretilir ve işletim sistemi kontrolü yeniden kazanır.
- İşletim sistemi, zaman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

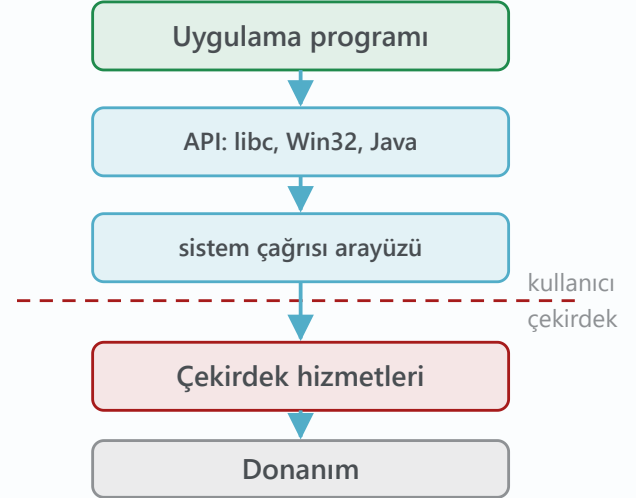


Çizelgeleyici A'nın dilimi dolduğu için B'yi seçer ve sayacı yeniden yükler.

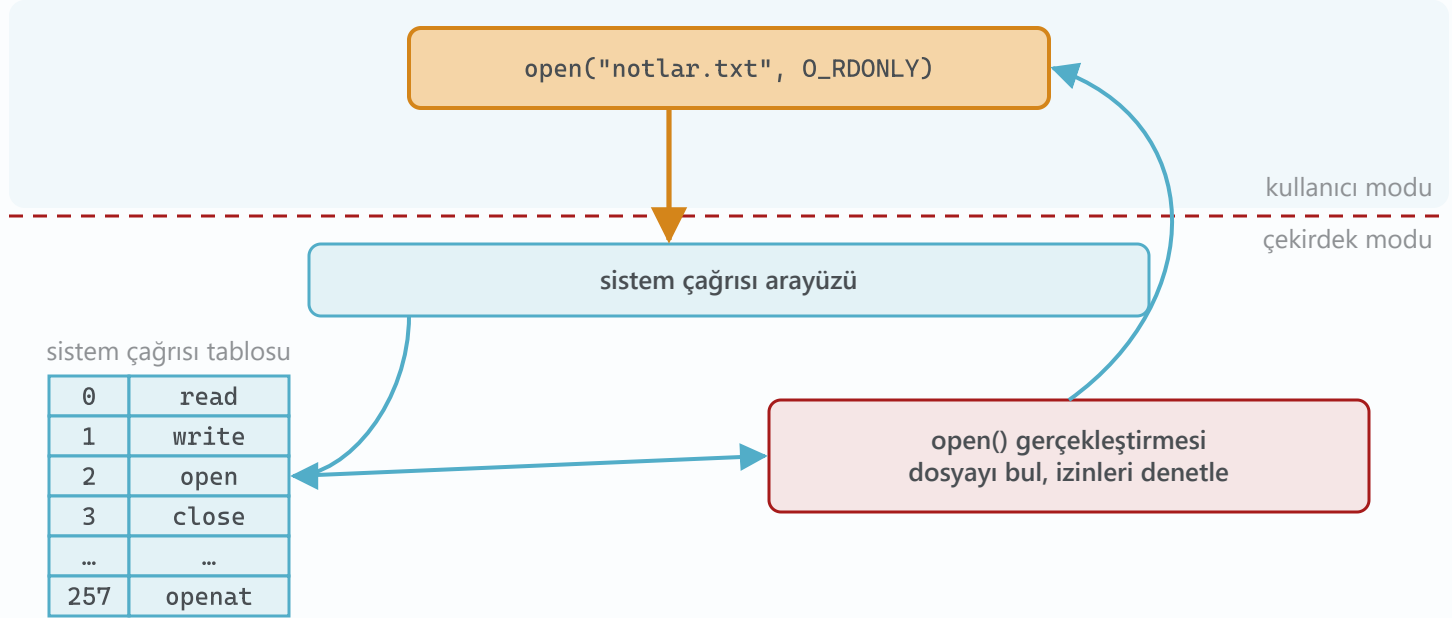
SİSTEM ÇAĞRILARI

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.

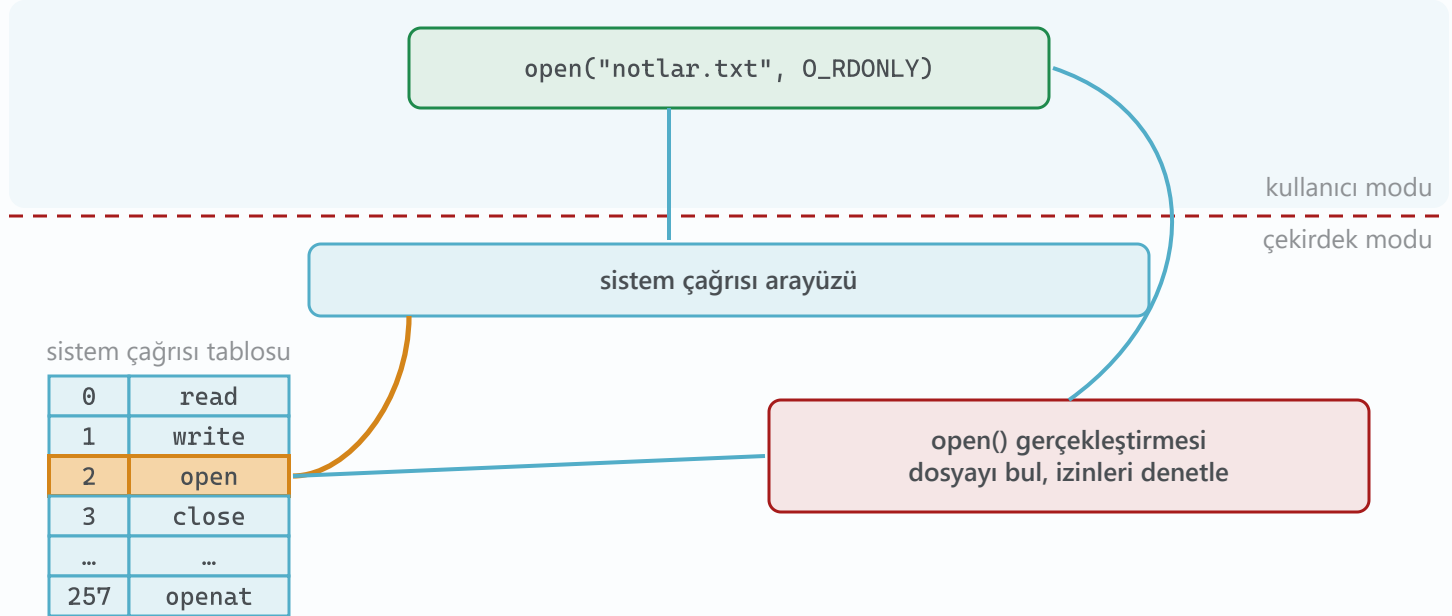


API → Sistem Çağrısı Arayüzü → Çekirdek



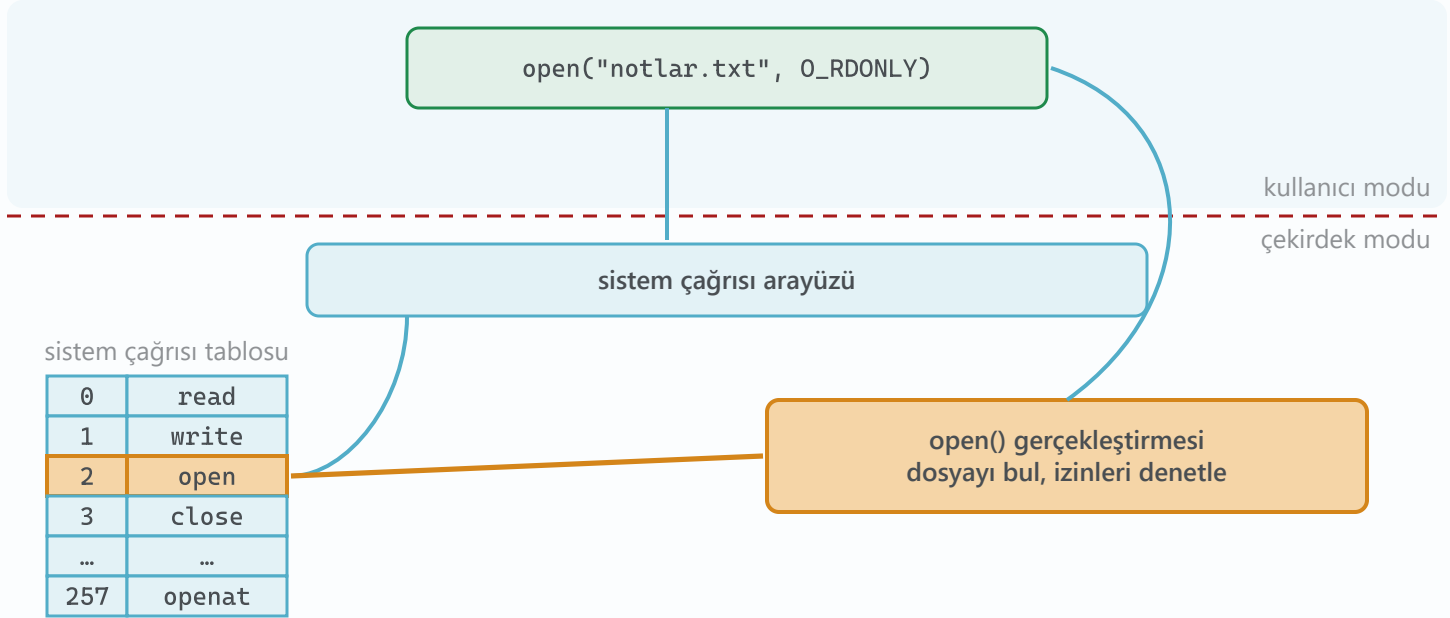
Uygulama libc'deki `open()` sarmalayıcısını çağırır; sarmalayıcı çağrı numarasını yazmaca koyar ve tuzak komutunu çalıştırır.

API → Sistem Çağrısı Arayüzü → Çekirdek



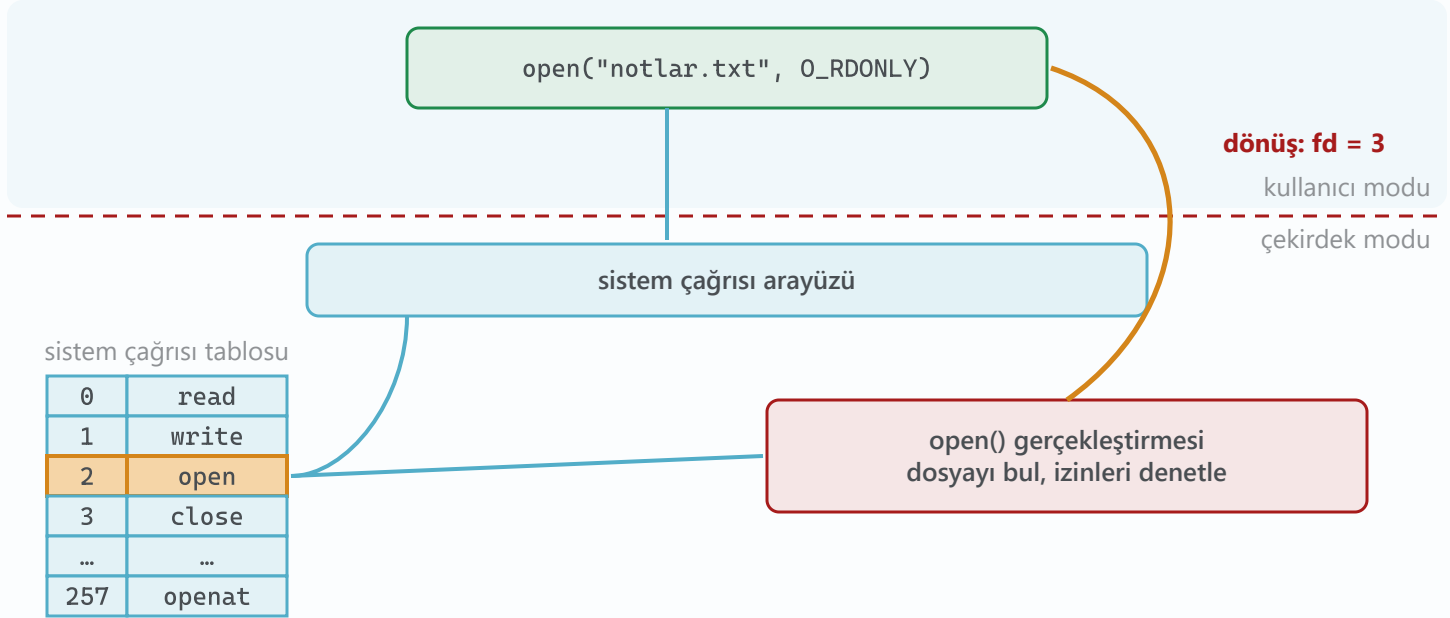
Çekirdekteki sistem çağrısı arayüzü, numarayı **sistem çağrısı tablosunda** indis olarak kullanır (x86-64 Linux: open = 2).

API → Sistem Çağrısı Arayüzü → Çekirdek



Tablodaki gösterici çekirdekteki gerçekleştirmeye 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};
4 ► long syscall_giris(long no, long a,
5 ►                     long b, long c) {
6     if (no < 0 || no >= 3)
7         return -ENOSYS;    /* geçersiz numara */
8     return tablo[no](a, b, c);
9 }
```

kullanıcı modu

write(1, "abc", 3)

çekirdek modu

tablo[]

rax

1

0

sys_read

1

sys_write

2

sys_open

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) {
6     if (no < 0 || no >= 3)
7         return -ENOSYS; /* geçersiz numara */
8     return tablo[no](a, b, c);
9 }
```

kullanıcı modu `write(1, "abc", 3)`

çekirdek modu

tablo[]

rax

1

0

sys_read

1

sys_write

2

sys_open

$0 \leq 1 < 3$: numara geçerli.

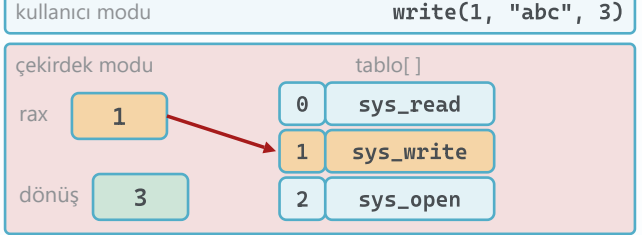
Adım 2 / 9

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 */
8     return tablo[no](a, b, c);
9 }
```



tablo[1] bir işlev göstericisi: **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};
4 ► long syscall_giris(long no, long a,
5 ►                     long b, long c) {
6     if (no < 0 || no >= 3)
7         return -ENOSYS;    /* geçersiz numara */
8     return tablo[no](a, b, c);
9 }
```

kullanıcı modu `open("notlar.txt", 0)`

çekirdek modu

tablo[]

rax

2

0

sys_read

1

sys_write

2

sys_open

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) {
6     if (no < 0 || no >= 3)
7         return -ENOSYS; /* geçersiz numara */
8     return tablo[no](a, b, c);
9 }
```

kullanıcı modu

open("notlar.txt", 0)

çekirdek modu

tablo[]

rax

2

0

sys_read

1

sys_write

2

sys_open

 $0 \leq 2 < 3$: numara geçerli.

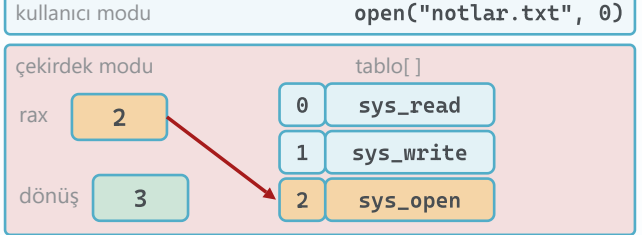
Adım 5 / 9

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 */
8     return tablo[no](a, b, c);
9 }
```



`tablo[2]` bir işlev göstericisi: **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};
4 ► long syscall_giris(long no, long a,
5 ►                     long b, long c) {
6     if (no < 0 || no >= 3)
7         return -ENOSYS;    /* geçersiz numara */
8     return tablo[no](a, b, c);
9 }
```

kullanıcı modu

syscall(99)

çekirdek modu

tablo[]

rax

99

0

sys_read

1

sys_write

2

sys_open

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

Adım 7 / 9

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) {
6     if (no < 0 || no >= 3)
7         return -ENOSYS; /* geçersiz numara */
8     return tablo[no](a, b, c);
9 }
```

kullanıcı modu

syscall(99)

çekirdek modu

tablo[]

rax

99

dönüş

-38

0

sys_read

1

sys_write

2

sys_open

no ≥ 3: tabloda yok

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)
7         return -ENOSYS; /* geçersiz numara */
8     return tablo[no](a, b, c);
9 }
```

kullanıcı modu

syscall(99) → -1

çekirdek modu

tablo[]

rax

?

0

sys_read

1

sys_write

dönüş

-1

2

sys_open

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

SİSTEM ÇAĞRILARI

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`).

SİSTEM ÇAĞRILARI

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 (varsın 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.

SİSTEM ÇAĞRILARI

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.

SİSTEM ÇAĞRILARI

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

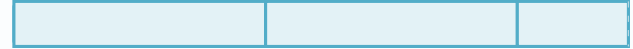
Girdi dosyasının adını iste ve oku	write, read
Çıktı dosyasının adını iste ve oku	write, read
Girdi dosyasını aç (yoksa hata ver, çık)	open
Çıktı dosyasını oluştur (varsın hata, çık)	open + O_CREAT
Döngü: girdiden oku, çıktıya yaz	read, write
Dosyaları kapat	close
Ekrana "tamamlandı" yaz	write
Normal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

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

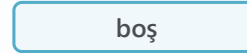
Kod İzleme: `read/write` ile Kopyalama

```
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);
```

kaynak

↑
ofset 0

buf



4096 baytlık tampon

hedef

çağrılar: `open×1`

"kaynak" 10 000 baytlık bir dosya. `open` ilk boş tanımlayıcıyı döndürür: 0, 1, 2 dolu olduğundan **`in = 3`**; ofset 0.

Adım 1 / 12

DEĞİŞKENLER

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

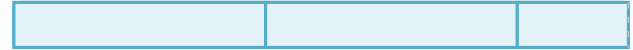
ÇIKTI



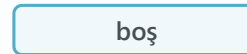
Kod İzleme: `read/write` ile Kopyalama

```
1 char buf[4096];
2 int in = open("kaynak", O_RDONLY);
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);
```

kaynak

↑
offset 0

buf



4096 baytlık tampon

hedef

çağrılar: `open`×1 `creat`×1Boş hedef dosya oluşturulur: **out = 4**.

Adım 2 / 12

DEĞİŞKENLER

in = 3

out = 4

n = ?

toplam = 0

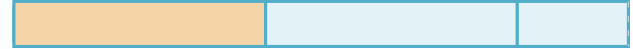
ÇIKTI



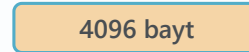
Kod İzleme: `read/write` ile Kopyalama

```
1 char buf[4096];
2 int in = open("kaynak", O_RDONLY);
3 int out = creat("hedef", 0644);
4 ssize_t n; long toplam = 0;
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);
```

kaynak

↑
ofset 4096

buf



4096 baytlık tampon

hedef

çağrılar: `open`×1 `creat`×1 `read`×1

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

Adım 3 / 12

DEĞİŞKENLER

in = 3

out = 4

n = 4096

toplam = 0

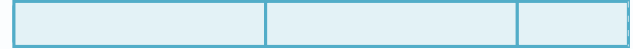
ÇIKTI



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) {
6     write(out, buf, n);
7     toplam += n;
8 }
9 close(in); close(out);
10 printf("%ld bayt\n", toplam);
```

kaynak

↑
ofset 4096

buf



4096 baytlık tampon

hedef

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

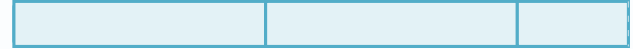
ÇIKTI



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);
7     toplam += n;
8 }
9 close(in); close(out);
10 printf("%ld bayt\n", toplam);
```

kaynak

↑
ofset 4096

buf



4096 baytlık tampon

hedef

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

toplam = 4096. Her tur iki sistem çağırısı: 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;
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);
```

kaynak

↑
ofset 8192

buf



4096 baytlık tampon

hedef



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

Yeniden `read` : **n = 4096**, ofset 8192.

Adım 6 / 12

DEĞİŞKENLER

in = 3

out = 4

n = 4096

toplam = 4096

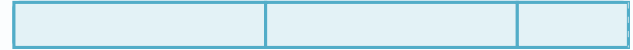
ÇIKTI



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) {
6     write(out, buf, n);
7     toplam += n;
8 }
9 close(in); close(out);
10 printf("%ld bayt\n", toplam);
```

kaynak

↑
offset 8192

buf



4096 baytlık tampon

hedef



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

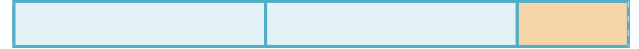
ÇIKTI



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;
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);
```

kaynak



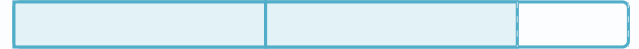
ofset 10000

buf

1808 bayt

4096 baytlık tampon

hedef



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

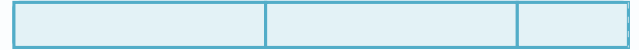
ÇIKTI



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) {
6     write(out, buf, n);
7     toplam += n;
8 }
9 close(in); close(out);
10 printf("%ld bayt\n", toplam);
```

kaynak



offset 10000

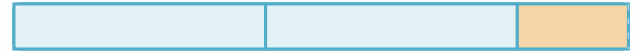


buf

1808 bayt

4096 baytlık tampon

hedef



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

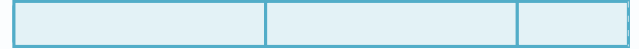
ÇIKTI



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;
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);
```

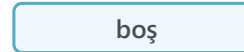
kaynak



offset 10000



buf



4096 baytlık tampon

hedef



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

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

Adım 10 / 12

DEĞİŞKENLER

in = 3

out = 4

n = 0

toplam = 10000

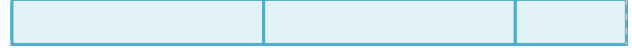
ÇIKTI



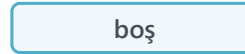
Kod İzleme: `read/write` ile Kopyalama

```
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 }
9 close(in); close(out);
10 printf("%ld bayt\n", toplam);
```

kaynak



buf



4096 baytlık tampon

hedef



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

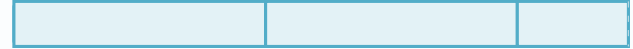
ÇIKTI



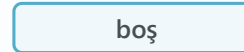
Kod İzleme: read/write ile Kopyalama

```
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);
```

kaynak



buf



4096 baytlık tampon

hedef



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

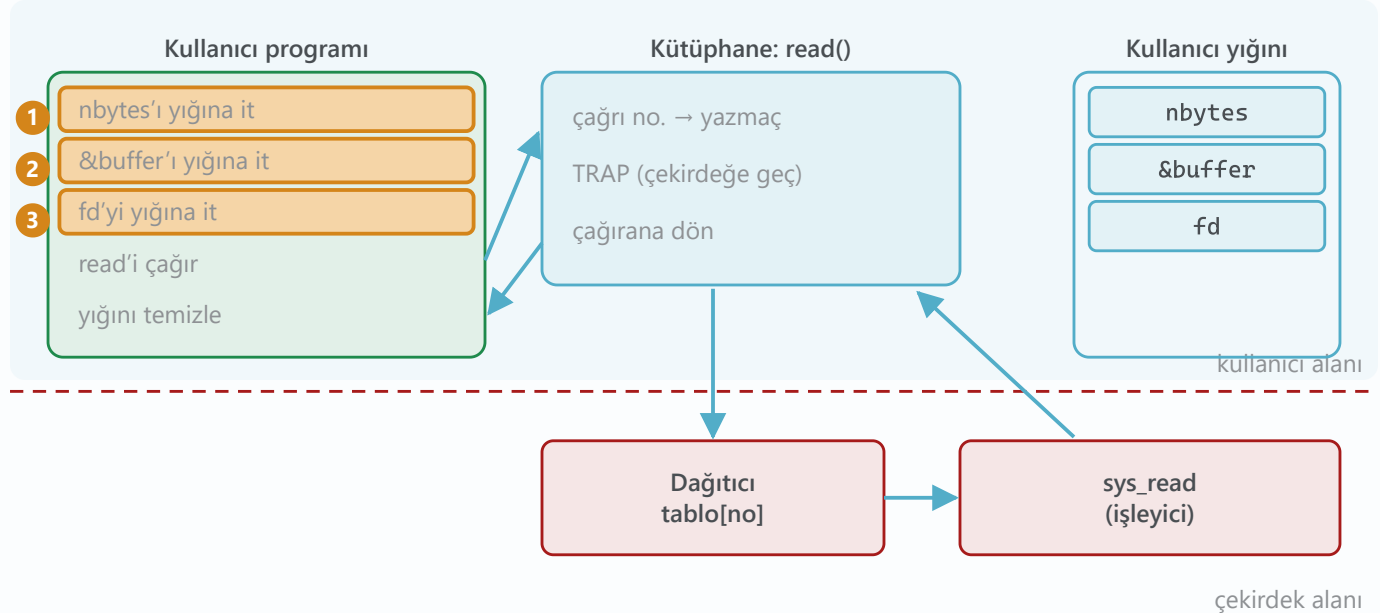
n = 0

toplam = 10000

ÇIKTI

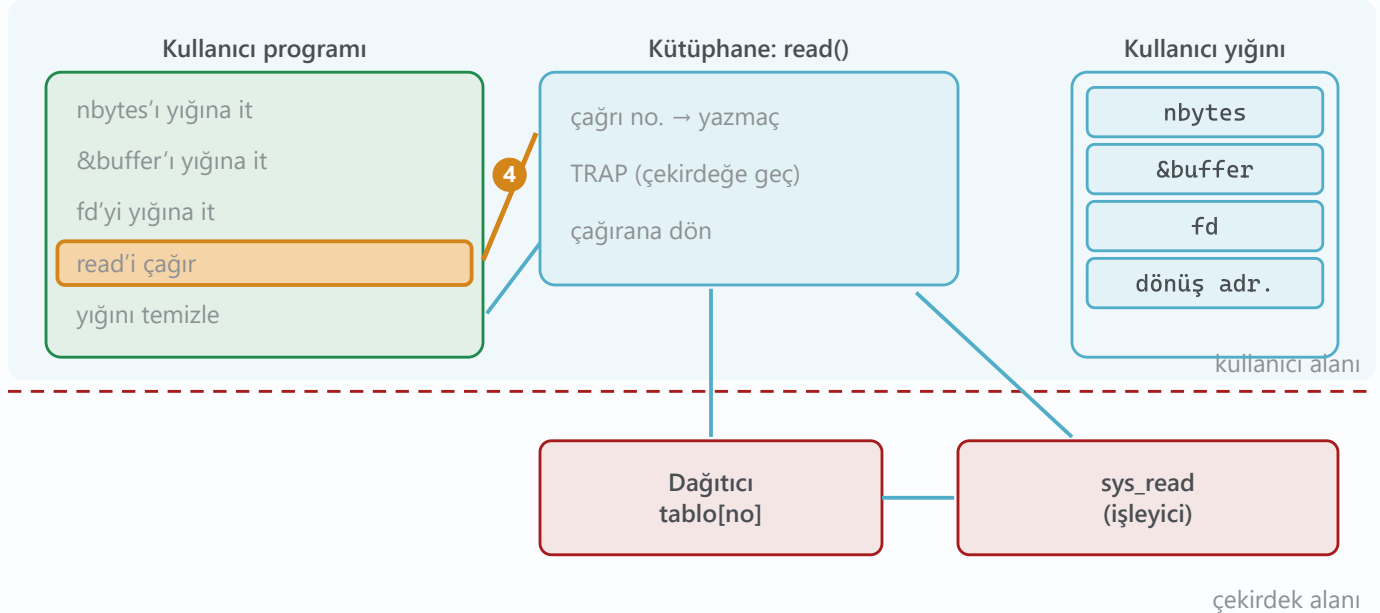
10000 bayt

read(fd, buffer, nbytes) Çağrısının 11 Adımı



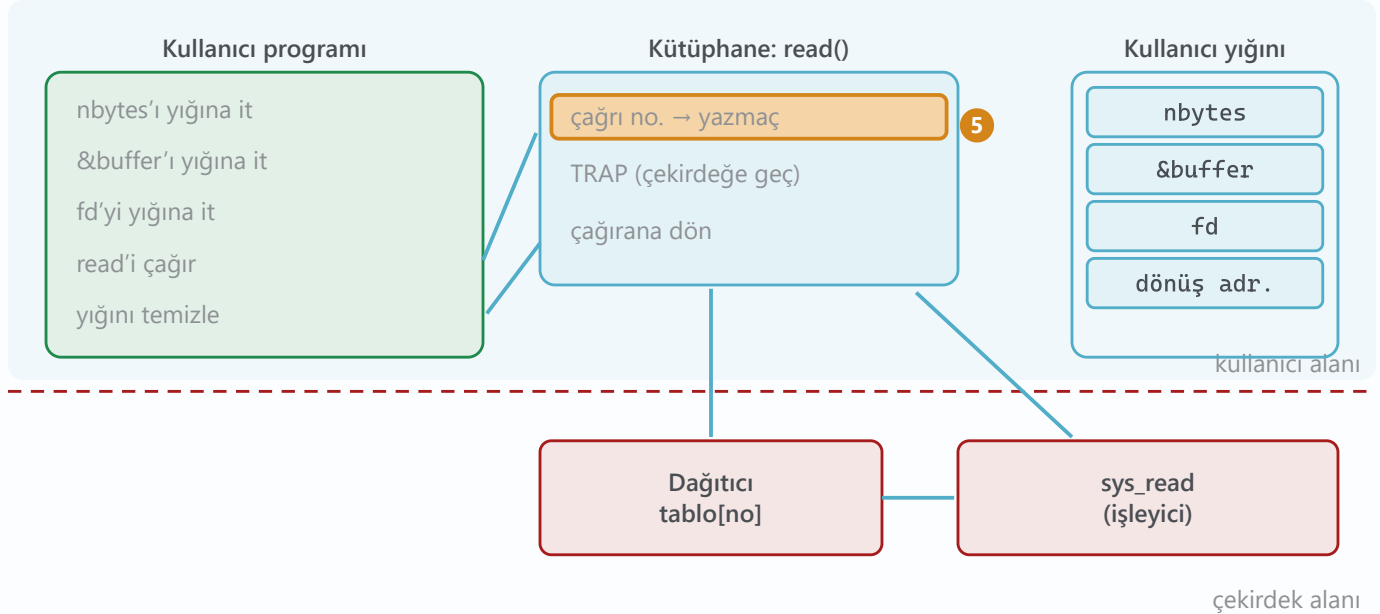
1–3: Parametreler yığına itilir (Tanenbaum'un örneğindeki tarihsel 32-bit x86 kuralı). x86-64'te ilk altı parametre yazmaçlarla aktarılır.

read(fd, buffer, nbytes) Çağrısının 11 Adımı



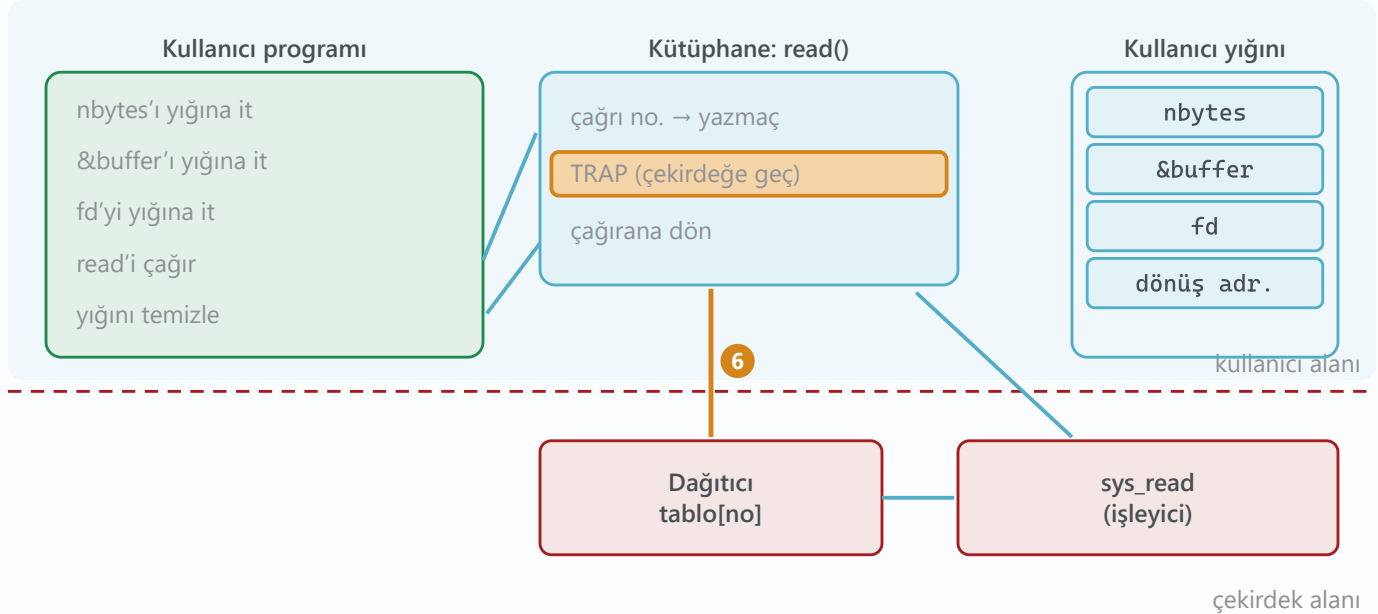
4: Kütüphanedeki `read()` sıradan bir fonksiyon gibi çağrılır; dönüş adresi yığına girer.

read(fd, buffer, nbytes) Çağrısının 11 Adımı



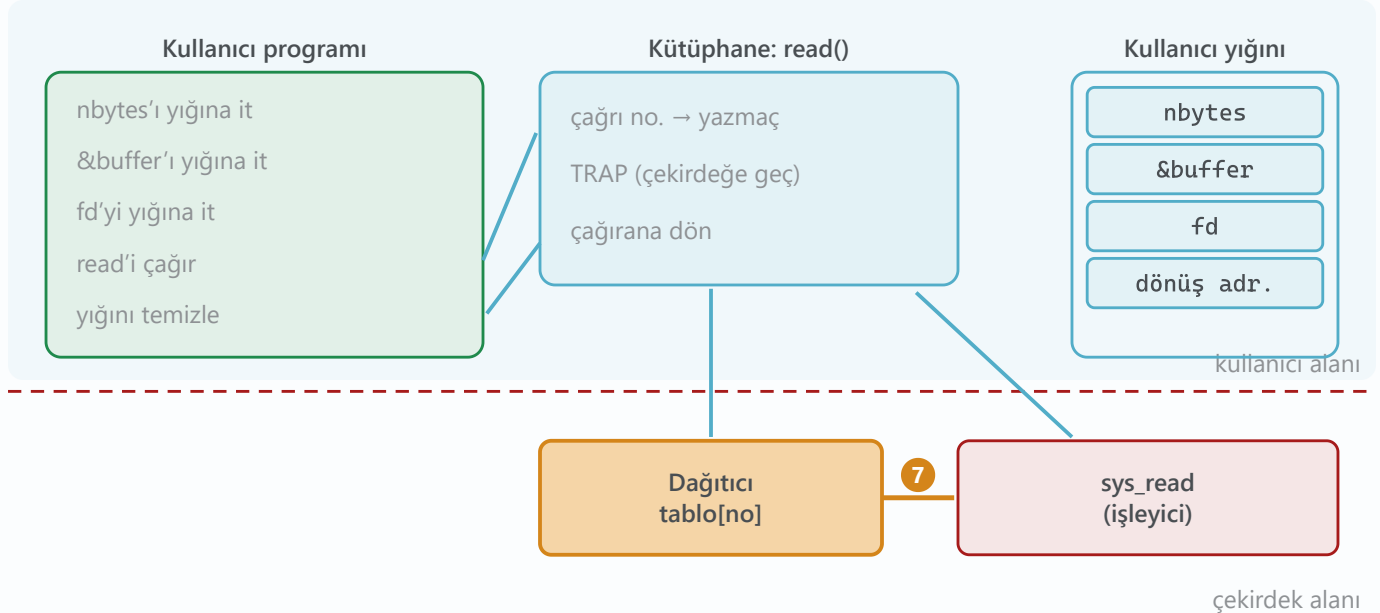
5: Kütüphane çağrı numarasını çekirdeğin beklediği yazmaca koyar (Linux x86-64: `rax = 0`).

read(fd, buffer, nbytes) Çağrısının 11 Adımı

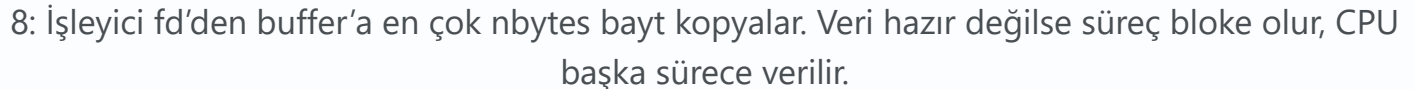


6: Tuzak komutu (x86-64: `syscall`) çalışır; CPU çekirdek moduna geçip çekirdekteki sabit adrese atlar.

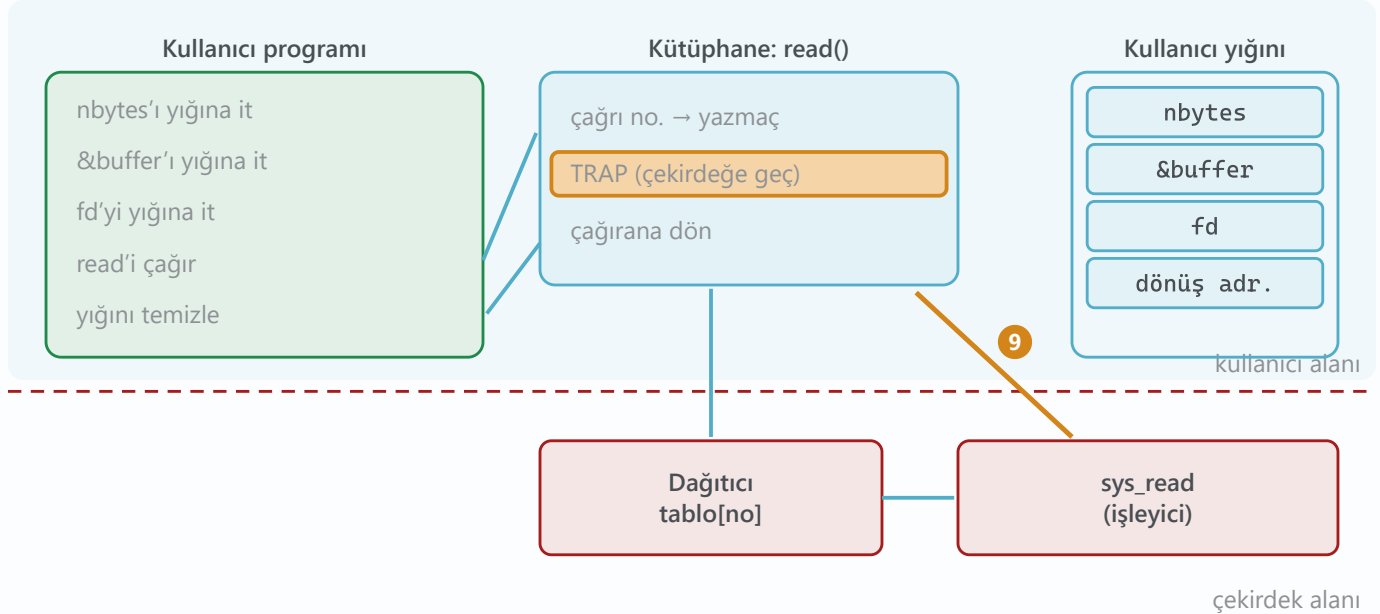
read(fd, buffer, nbytes) Çağrısının 11 Adımı



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

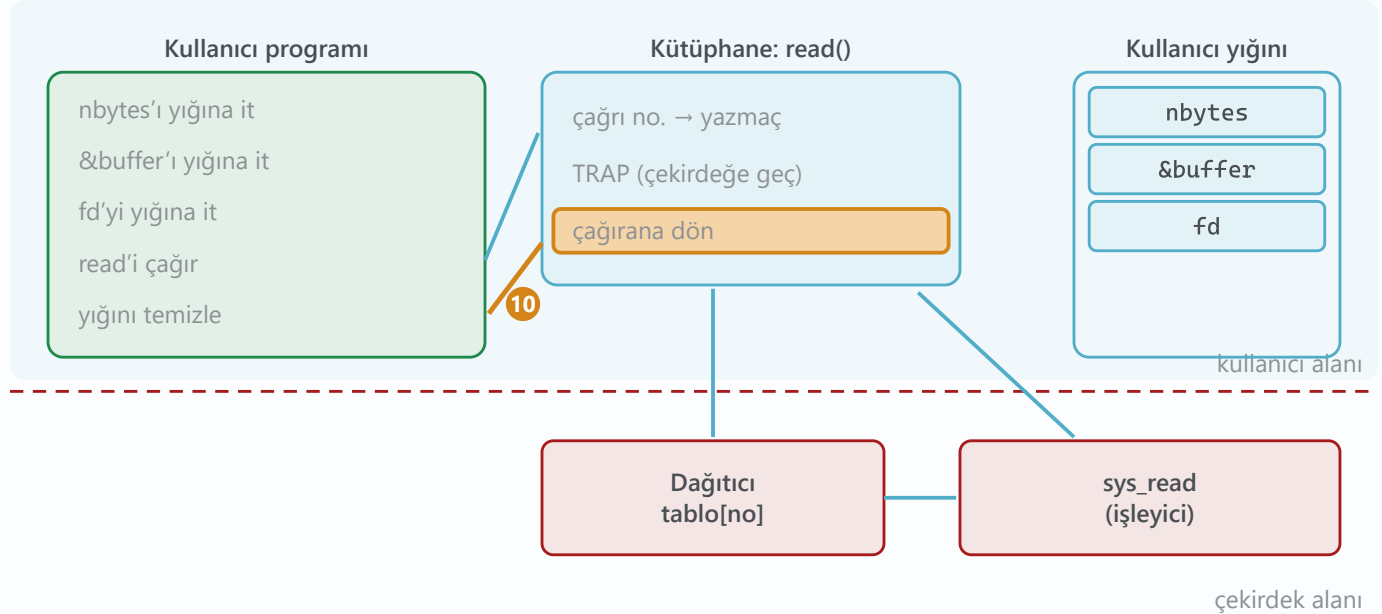


read(fd, buffer, nbytes) Çağrısının 11 Adımı



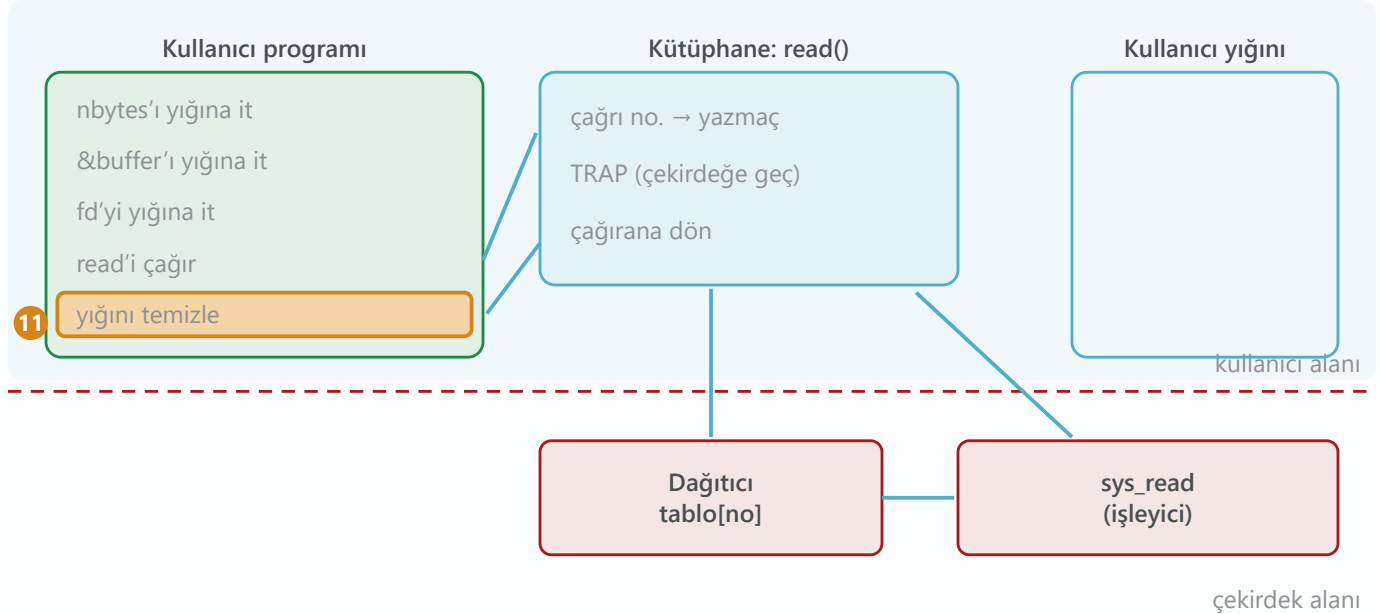
9: İş bitince kontrol kullanıcı moduna, tuzaktan sonraki komuta döner; sonuç (okunan bayt sayısı) yazmaçtadır.

read(fd, buffer, nbytes) Çağrısının 11 Adımı



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

read(fd, buffer, nbytes) Çağrısının 11 Adımı

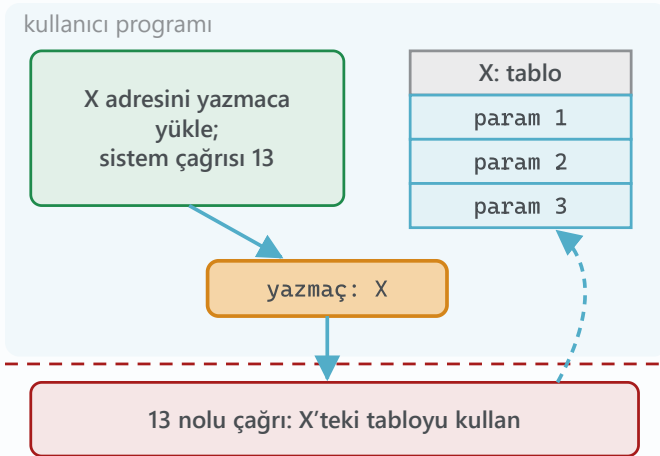


11: Program parametreleri yığından temizler (SP artırılır) ve sonraki komutla devam eder.

SİSTEM ÇAĞRILARI

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

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) {
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
10    return 0;
11 }
```

DEĞİŞKENLER

`msg` = "Merhaba, Dünya!\n"

`len` = ? `n` = ?

ÇAĞRI YIĞINI

`main()`

ÇIKTI

`msg` , salt okunur veri kesimindeki dizgeyi 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";
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 = 17`

`n = ?`

ÇAĞRI YİĞİNİ

`main()`

ÇIKTI

"Merhaba, " 9 + "Dünya" 6 (ü, UTF-8'de 2 bayt) + "!" ve "\n" 2 = **17** bayt. Sabit 15 yazılıydı 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
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 YİĞİNİ

`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
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

`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
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 = 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
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!

`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
10    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) {
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
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];
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 YİĞİNİ

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];
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

`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];
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 = "selam\n..." (6 bayt)`

`n = 6`

ÇAĞRI YİĞİNİ

`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));
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

`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    }
11    ► write(1, buf, n);    // tam n bayt
12    return 0;
13 }
```

DEĞİŞKENLER

`buf = "selam\n..." (6 bayt)`

`n = 6`

ÇAĞRI YİĞİNİ

`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
12    return 0;
13 }
```

DEĞİŞKENLER

`buf = "selam\n..." (6 bayt)`

`n = 6`

ÇAĞRI YİĞİNİ

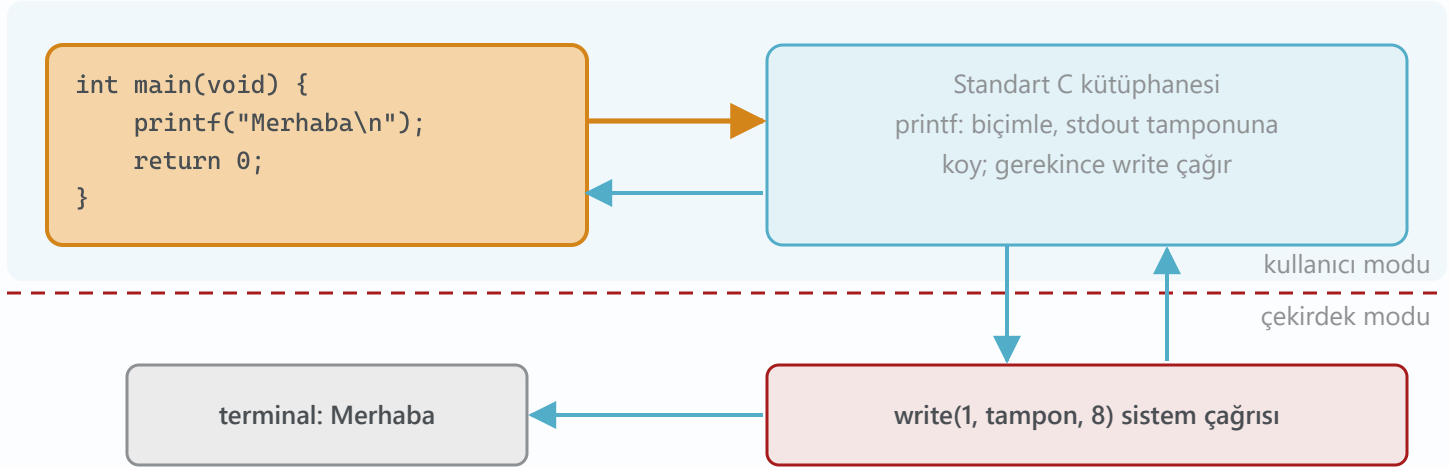
`main()`

ÇIKTI

```
selam
selam
```

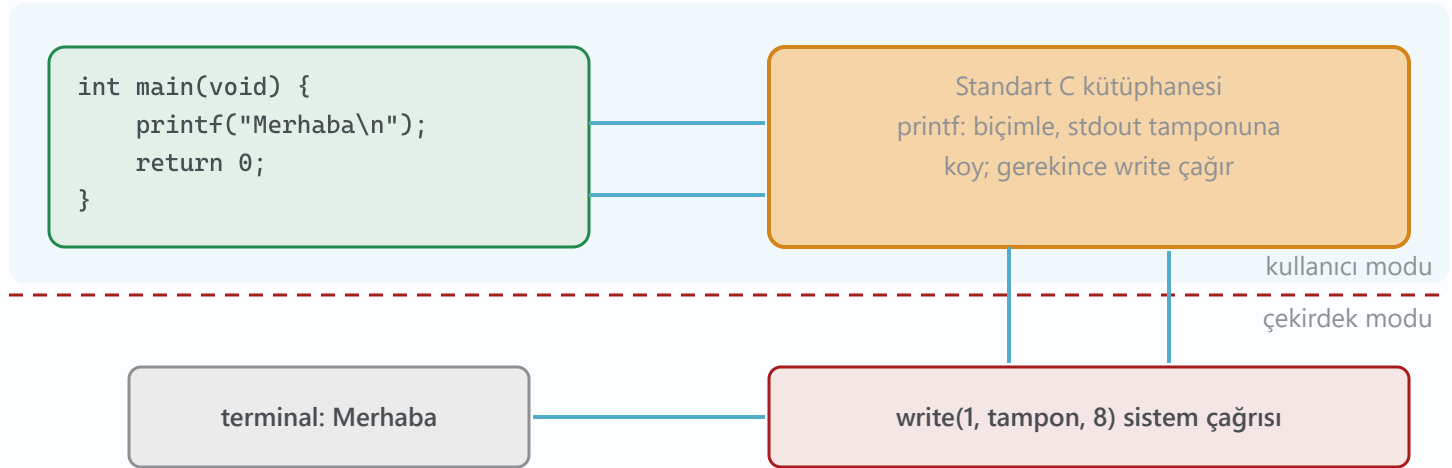
Program 0 ile sonlanır.

Standart C Kütüphanesi ve Sistem Çağrılar



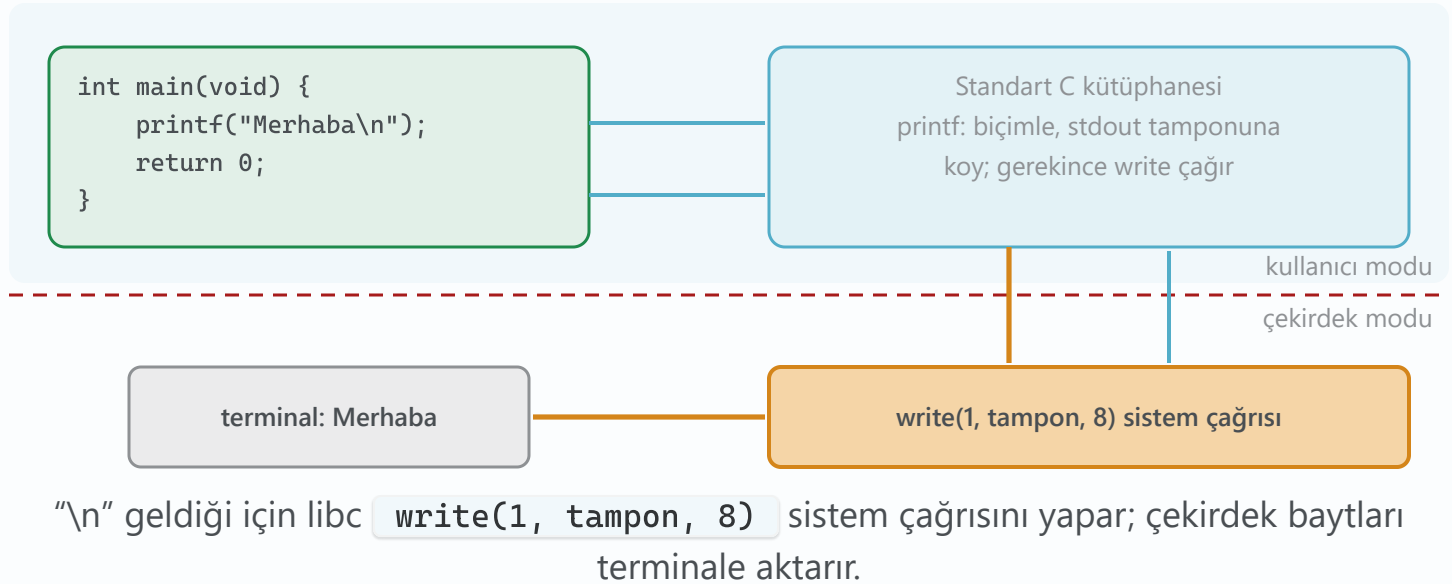
Program `printf` 'i çağırır; bu bir sistem çağırısı değil, kullanıcı modunda çalışan bir kütüphane fonksiyonudur.

Standart C Kütüphanesi ve Sistem Çağrıları

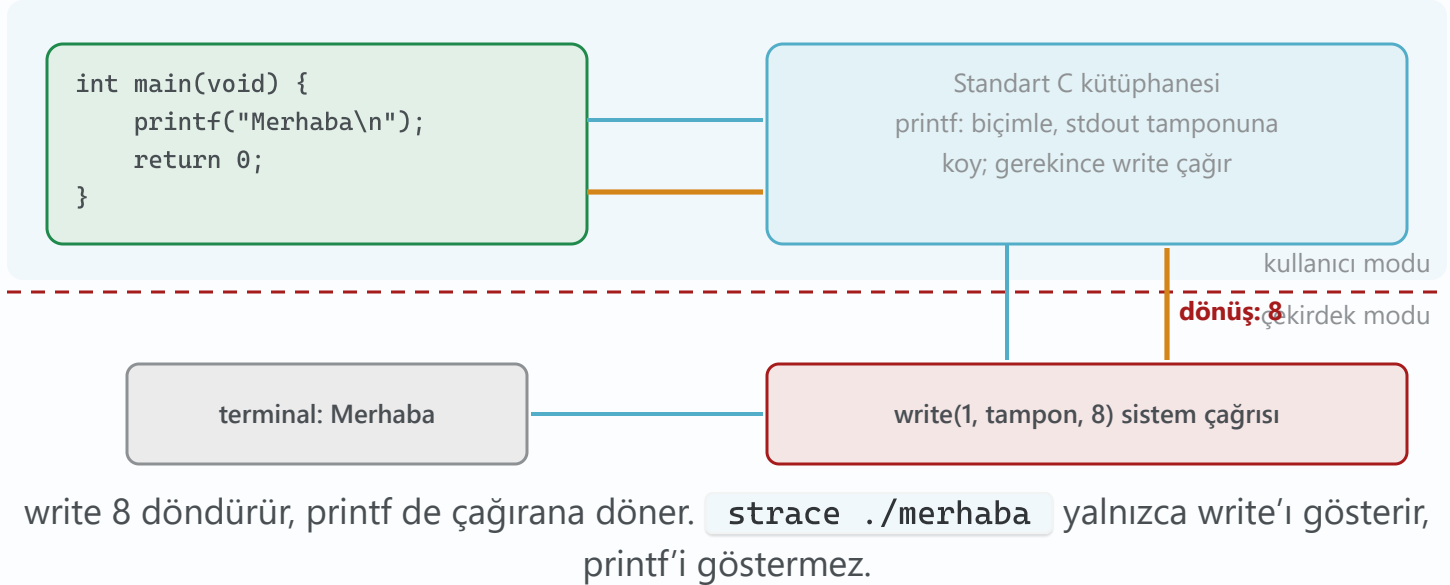


libc metni biçimler ve stdout tamponuna koyar. Terminalde tampon **satır** tamponludur; dosyaya/boruya yönlendirilince **tam** tamponludur.

Standart C Kütüphanesi ve Sistem Çağrıları



Standart C Kütüphanesi ve Sistem Çağrıları



Sistem Çağrısı Türleri

Süreç denetimi

oluştur, sonlandır, yükle,
bekle, sinyale

`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)

SİSTEM ÇAĞRILARI

POSIX Çağrıları: Süreç Yönetimi

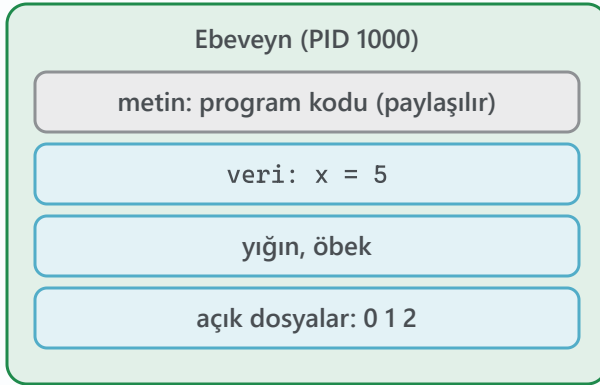
Çağrı	Açıklama
<code>pid = fork()</code>	Ebeveynin kopyası olan bir çocuk süreç oluşturur.
<code>pid = waitpid(pid, &statloc, options)</code>	Bir çocuğun sonlanmasını bekler; çıkış durumunu <code>statloc</code> 'a yazar.
<code>s = execve(name, argv, envp)</code>	Sürecin bellek görüntüsünü yeni programla değiştirir; başarılıysa geri dönmaz.
<code>exit(status)</code>	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.

fork(): Süreç Oluşturma

Çocuk, ebeveynin adres alanının **kopyasını** alır: aynı kod, aynı değişken değerleri, aynı açık dosyalar; ama ayrı bir adres alanında çalışır.

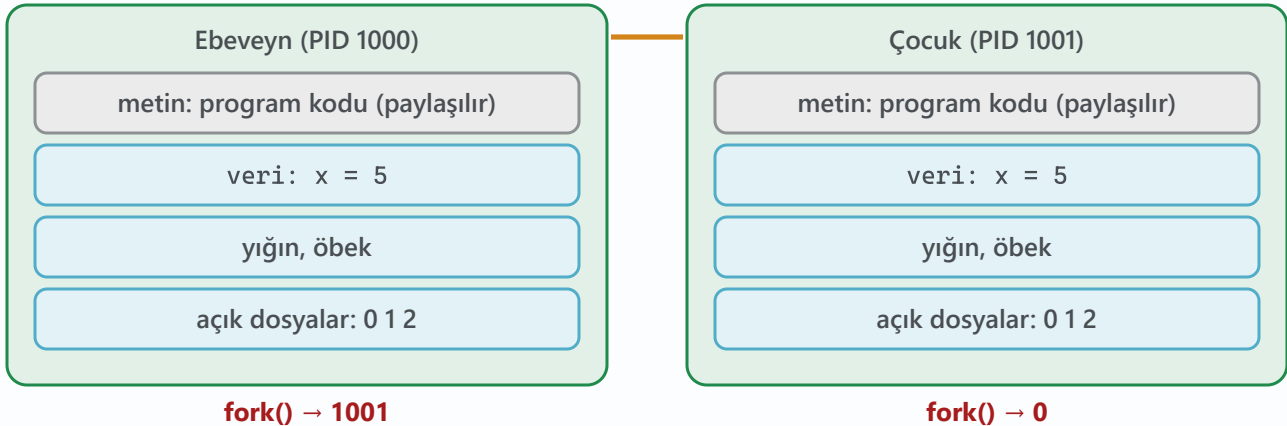


fork() çağırıyor...

Ebeveyn süreçte `x = 5` . Süreç `pid = fork();` satırına gelir.

fork(): Süreç Oluşturma

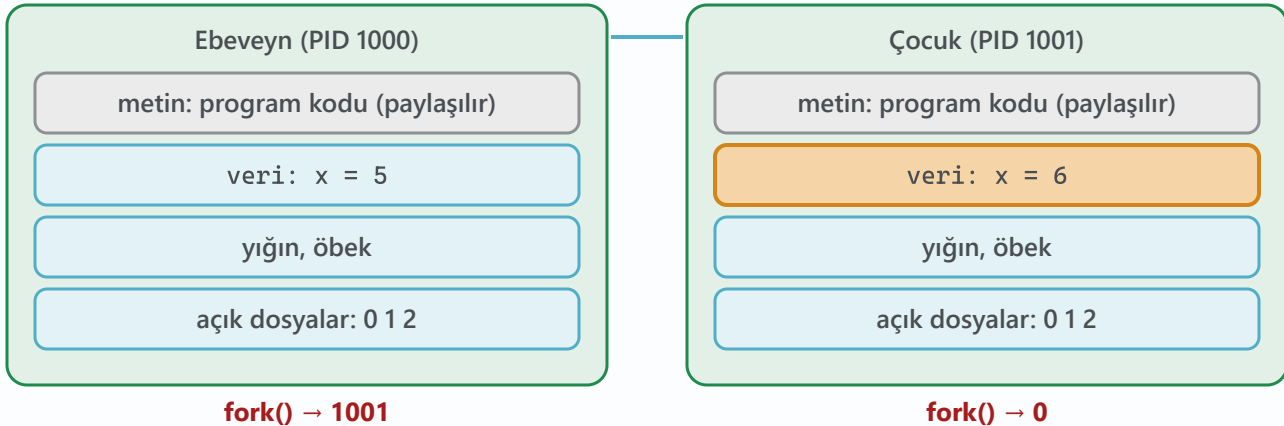
Çocuk, ebeveynin adres alanının **kopyasını** alır: aynı kod, aynı değişken değerleri, aynı açık dosyalar; ama ayrı bir adres alanında çalışır.



`fork` bir kez çağrılır, **iki kez** döner: ebeveynde çocuğun PID'si (1001), çocukta 0. Hata olursa yalnızca ebeveynde -1 döner.

fork(): Süreç Oluşturma

Çocuk, ebeveynin adres alanının **kopyasını** alır: aynı kod, aynı değişken değerleri, aynı açık dosyalar; ama ayrı bir adres alanında çalışır.



Çocuk `x = 6` yazınca yalnızca kendi kopyası değişir; ebeveynde `x` hâlâ 5. Modern işletim sistemleri sayfaları yazılana dek paylaşır (**copy-on-write**), kopyalama ertelenir.

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    }
12 }
```

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 */
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     }
12 }
```

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
7     ▶ if (fork() != 0) {       /* ebeveyn */
8         waitpid(-1, &status, 0);
9     } else {                   /* çocuk süreç */
10         execve(command, parameters, 0);
11     }
12 }
```

DEĞİŞKENLER

command = **"/bin/ls"**

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

fork→ebeveyn = **1045**

fork→çocuk = **0**

ÇAĞRI YİĞİNİ

çocuk (1045)

kabuk (812)

ÇIKTI

\$ ls

fork kabuğun kopyasını oluşturur. Ebeveynde sonuç 1045 ($\neq 0$), 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 */
8         ▶ waitpid(-1, &status, 0);
9     } else {                  /* çocuk süreç */
10         execve(command, parameters, 0);
11     }
12 }
```

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 */
10         ▶ execve(command, parameters, 0);
11     }
12 }
```

DEĞİŞKENLER

command = `"/bin/ls"`

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

fork→ebeveyn = **1045** fork→çocuk = **0**

ÇAĞRI YİĞİNİ

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 */
8         ▶ waitpid(-1, &status, 0);
9     } else {                  /* çocuk süreç */
10         execve(command, parameters, 0);
11     }
12 }
```

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     }
12 }
```

DEĞİŞKENLER

command = `"/bin/ls"`

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

ÇAĞRI YIĞINI

kabuk (812)

ÇIKTI

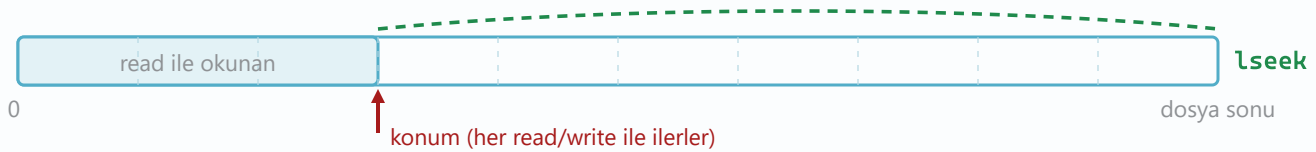
```
$ ls
notlar.txt  odev.c
$
```

Döngü başa döner, yeni istem yazılır.

SİSTEM ÇAĞRILARI

POSIX Çağrıları: Dosya Yönetimi

Çağrı	Açıklama
<code>fd = open(file, how, ...)</code>	Dosyayı okuma, yazma veya ikisi için açar (gerekirse oluşturur).
<code>s = close(fd)</code>	Açık dosyayı kapatır.
<code>n = read(fd, buffer, nbytes)</code>	Dosyadan tampona en çok nbytes bayt okur.
<code>n = write(fd, buffer, nbytes)</code>	Tampondan dosyaya veri yazar.
<code>position = lseek(fd, offset, whence)</code>	Dosya konum göstericisini taşır (<code>SEEK_SET</code> , <code>SEEK_CUR</code> , <code>SEEK_END</code>).
<code>s = stat(name, &buf)</code>	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.

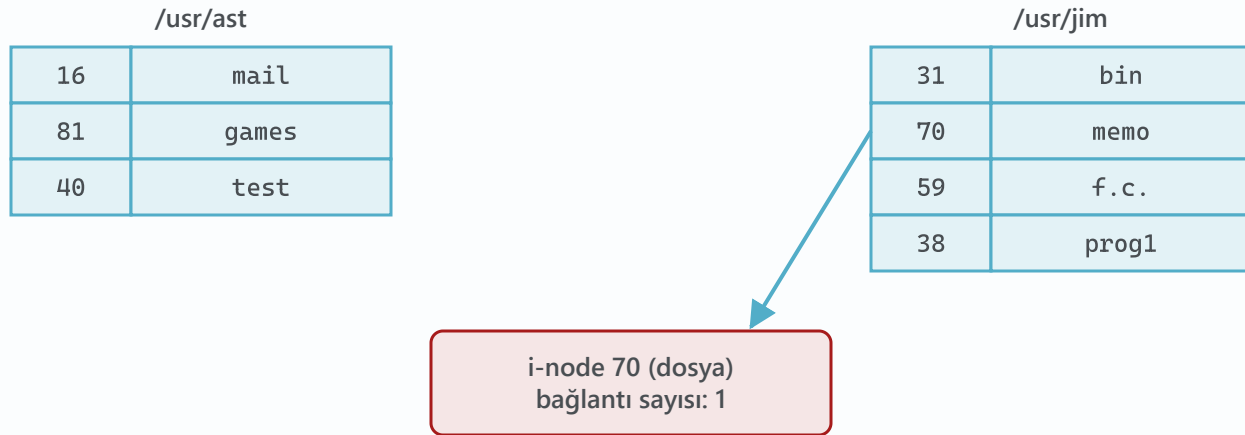
POSIX Çağrıları: Dizin Yönetimi ve Diğerleri

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

link ve unlink: Bir Dosya, İki Ad

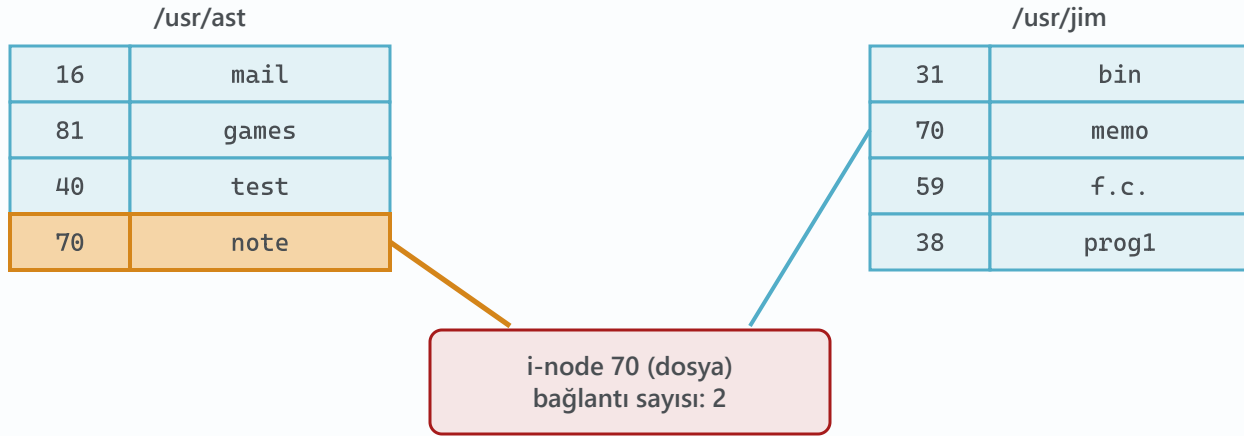
UNIX'te dizin, (i-node numarası, ad) çiftlerinden oluşan bir tablodur. Dosyanın asıl bilgisi i-node'dadır (inode).



Başlangıç: `/usr/jim/memo` 70 numaralı i-node'u gösterir; bağlantı 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).

/usr/ast	
16	mail
81	games
40	test
70	note

/usr/jim	
31	bin
59	f.c.
38	prog1

i-node 70 (dosya)
bağlantı sayısı: 1

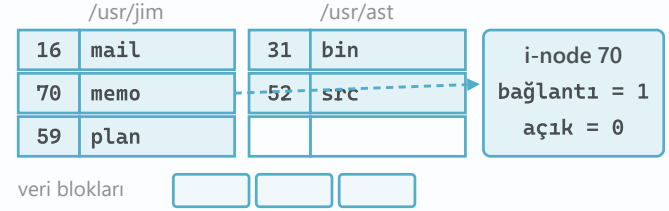
`unlink("/usr/jim/memo")` yalnızca girdiyi siler; dosya `note` adıyla yaşar. Sayı 0 olunca (ve açık tutan kalmayınca) blokları serbest kalır.

Kod İzleme: Bağlantı 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; bağlantı sayısı 1, dosya 85 bayt.

Adım 1 / 7

DEĞİŞKENLER

fd = ?

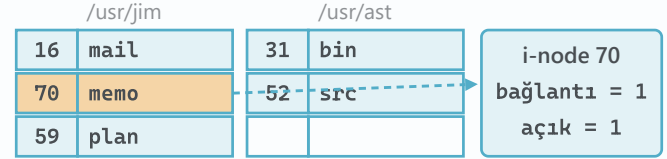
n = ?

Kod İzleme: Bağlantı 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 */
6 close(fd);

```



veri blokları



fd 3 → açık dosya kaydı → i-node 70

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

Adım 2 / 7

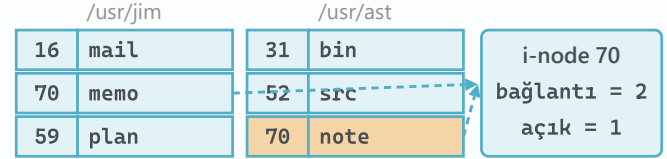
DEĞİŞKENLER

fd = 3

n = ?

Kod İzleme: Bağlantı 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 */
6 close(fd);
```



veri blokları



fd 3 → açık dosya kaydı → i-node 70

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

Adım 3 / 7

DEĞİŞKENLER

fd = 3

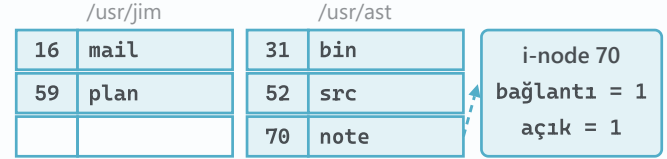
n = ?

Kod İzleme: Bağlantı 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 */
6 close(fd);

```



veri blokları

fd 3 → açık dosya kaydı → i-node 70

unlink yalnızca dizin girdisini siler; dosya

note adıyla yaşar. Bağlantı = 1.

Adım 4 / 7

DEĞİŞKENLER

fd = 3

n = ?

Kod İzleme: Bağlantı 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 */
6 close(fd);
```

/usr/jim		/usr/ast	
16	mail	31	bin
59	plan	52	src

i-node 70
bağlantı = 0
açık = 1

veri blokları

fd 3 → açık dosya kaydı → i-node 70

Son ad da silindi: **bağlantı = 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: Bağlantı 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);
```

/usr/jim		/usr/ast	
16	mail	31	bin
59	plan	52	src

i-node 70
bağlantı = 0
açık = 1

veri blokları

fd 3 → açık dosya kaydı → i-node 70

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

Adım 6 / 7

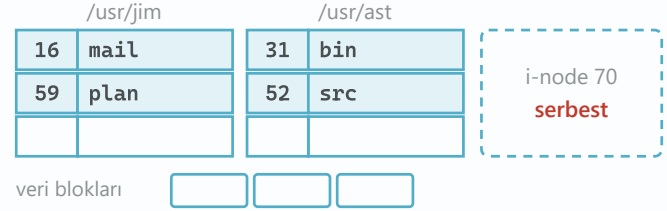
DEĞİŞKENLER

fd = 3

n = 85

Kod İzleme: Bağlantı 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: bağlantı = 0 ve açık = 0.
Çekirdek ancak şimdi i-node'u ve veri bloklarını **serbest bırakır**. Geçici dosyalar bu yolla açırken silinir.

Adım 7 / 7

DEĞİŞKENLER

fd = 3

n = 85

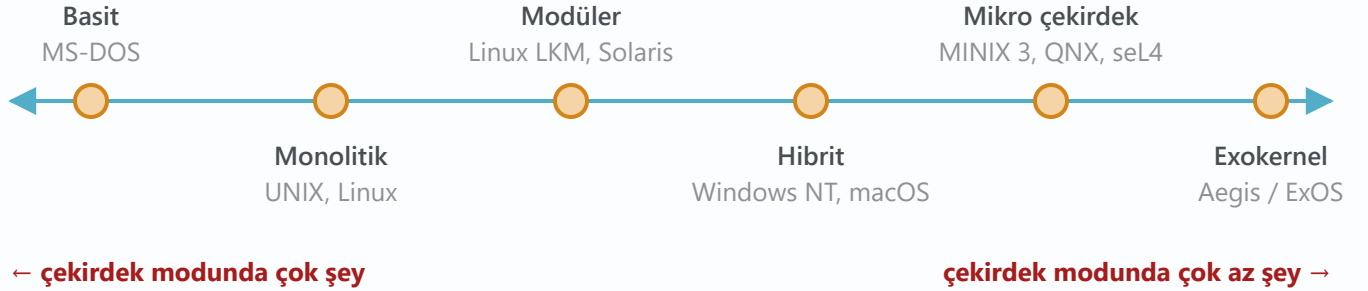
UNIX Çağrılarının Win32 API Karşılıkları

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 çağrılarının numaraları sürümden sürüme değiştiği ve belgelenmediği için programlar API'yi kullanır. Tanenbaum 3e'de "yok" görünen `link` ve `mount` karşılıkları bugün mevcuttur.

İşletim Sistemi Yapıları: Genel Bakış



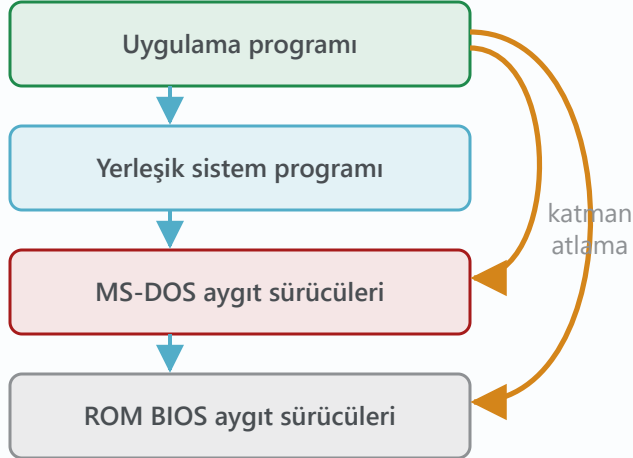
Katmanlı yapı

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



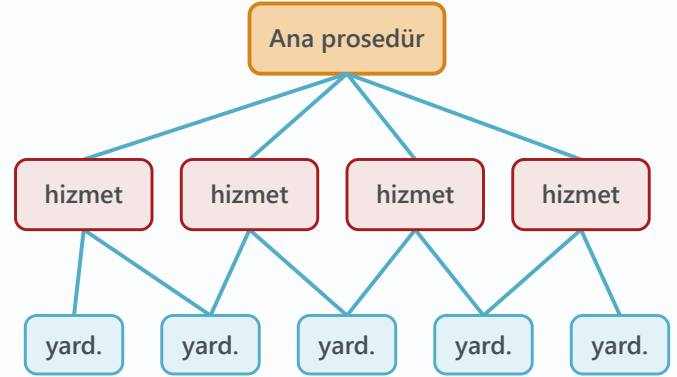
- Belirgin bir yapısı yoktur: küçük, basit ve sınırlı donanım için yazılmıştır.
- Arayüzler ve işlev düzeyleri iyi ayrılmamıştır; uygulamalar BIOS rutinlerine ve donanıma **doğrudan** erişebilir.
- Intel 8088’de çift mod olmadığından donanım koruması yoktu: tek bir hatalı program **tüm sistemi çökertebilir**.
- 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 gerçekleştirmesi 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

0

İşlemci tahsisi, çoklu programlama

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

Adım 1 / 6

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

1

Bellek ve tambur yönetimi

0

İşlemci tahsisi, çoklu programlama

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

Adım 2 / 6

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

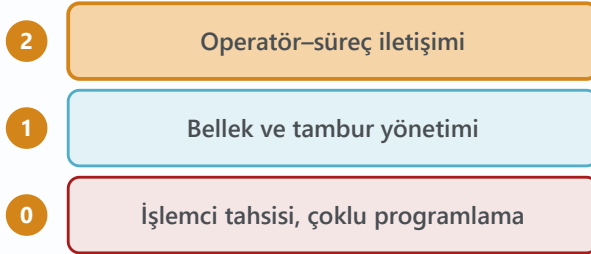
Artılar

- + 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.

Adım 3 / 6

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

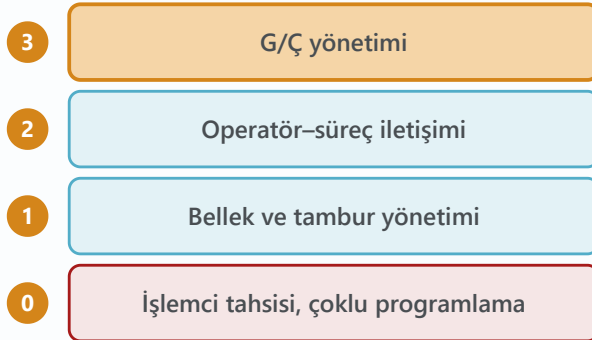
Artılar

- + 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.

Adım 4 / 6

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

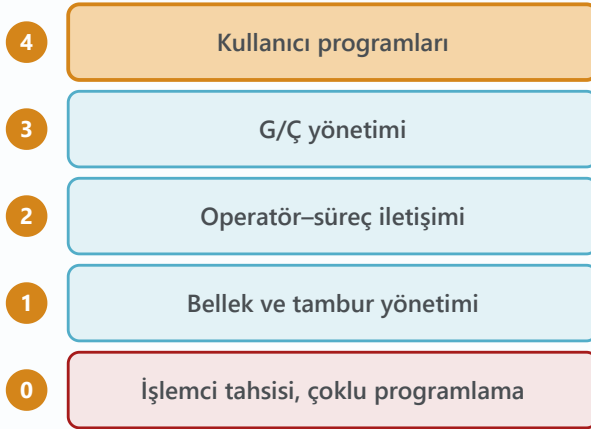
Artılar

- + 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.

Adım 5 / 6

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.

Adım 6 / 6

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: MINIX 3, QNX Neutrino (otomotiv), L4 ailesi, biçimsel olarak doğrulanmış **seL4**. macOS ise saf mikro çekirdek değil, **hibrit** XNU'dur.
- **Mekanizma çekirdekte, politika dışarıda:** çekirdek öncelikli süreçleri çalıştırmayı sağlar; önceliklerin nasıl atanacağına kullanıcı modundaki bileşen karar verir.

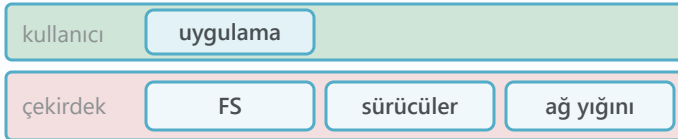
Artılar

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

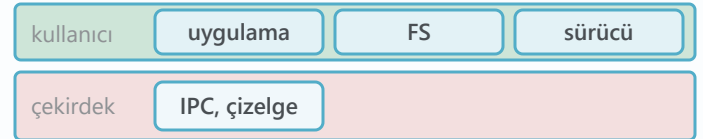
Eksiler

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

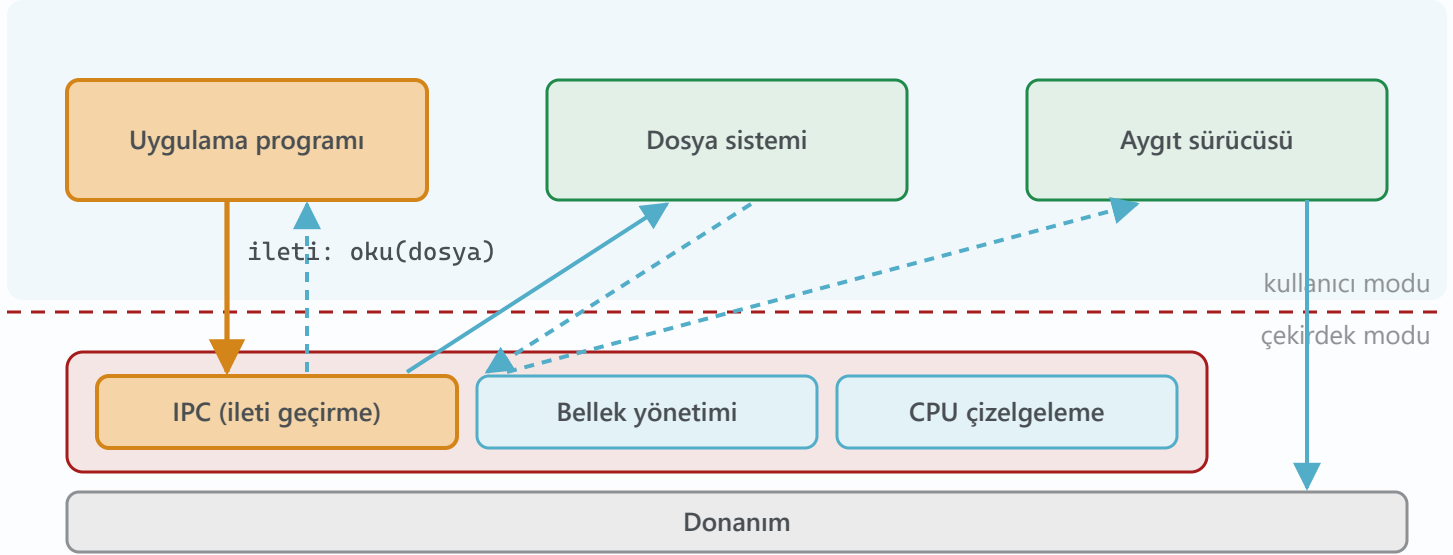
Monolitik



Mikro çekirdek

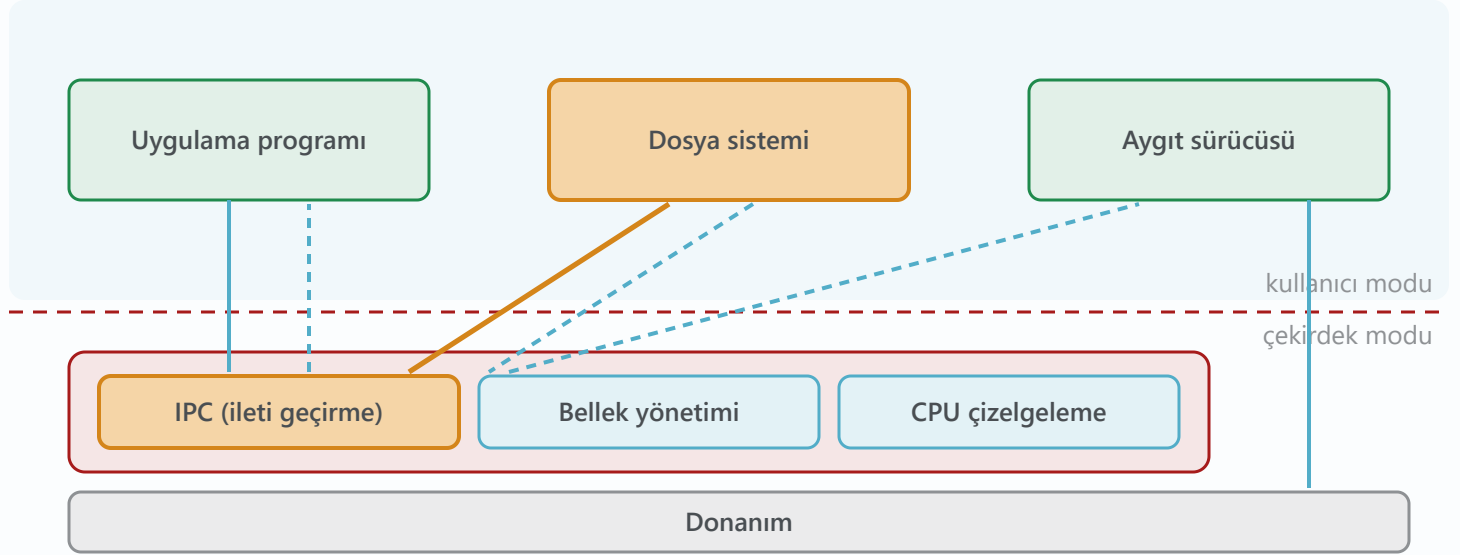


Mikro Çekirdekte Bir Dosya Okuma



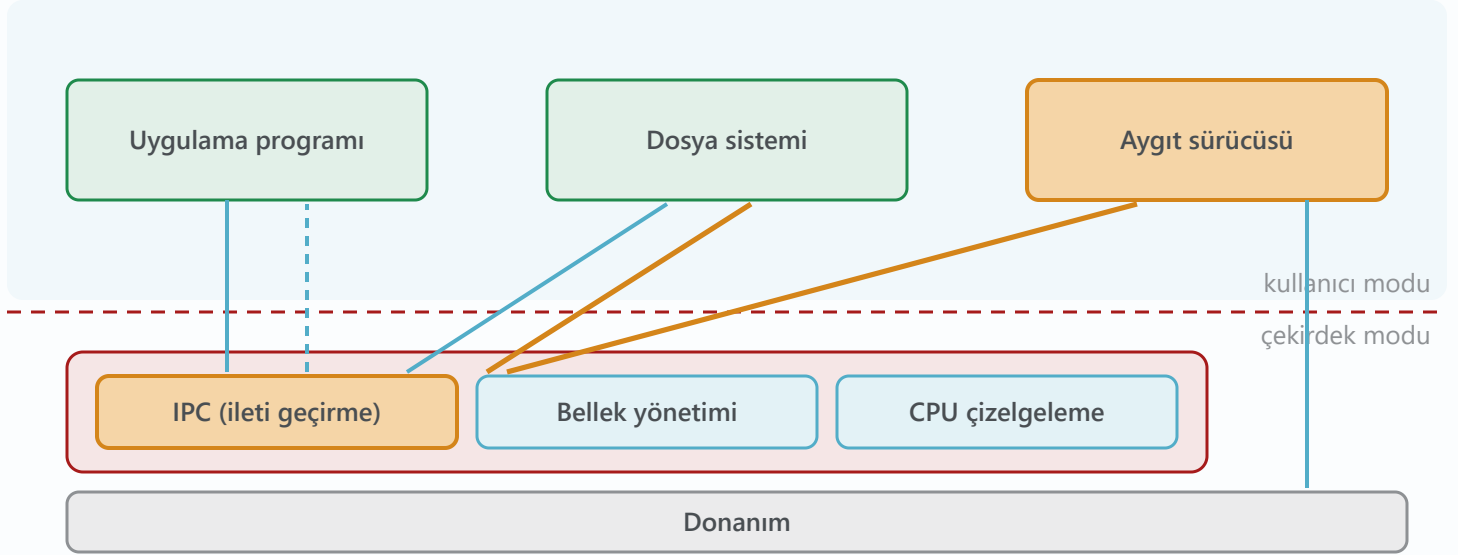
Uygulama, dosya sistemi sunucusuna bir **ileti** gönderir; ileti çekirdeğin IPC mekanizmasından geçer.

Mikro Çekirdekte Bir Dosya Okuma



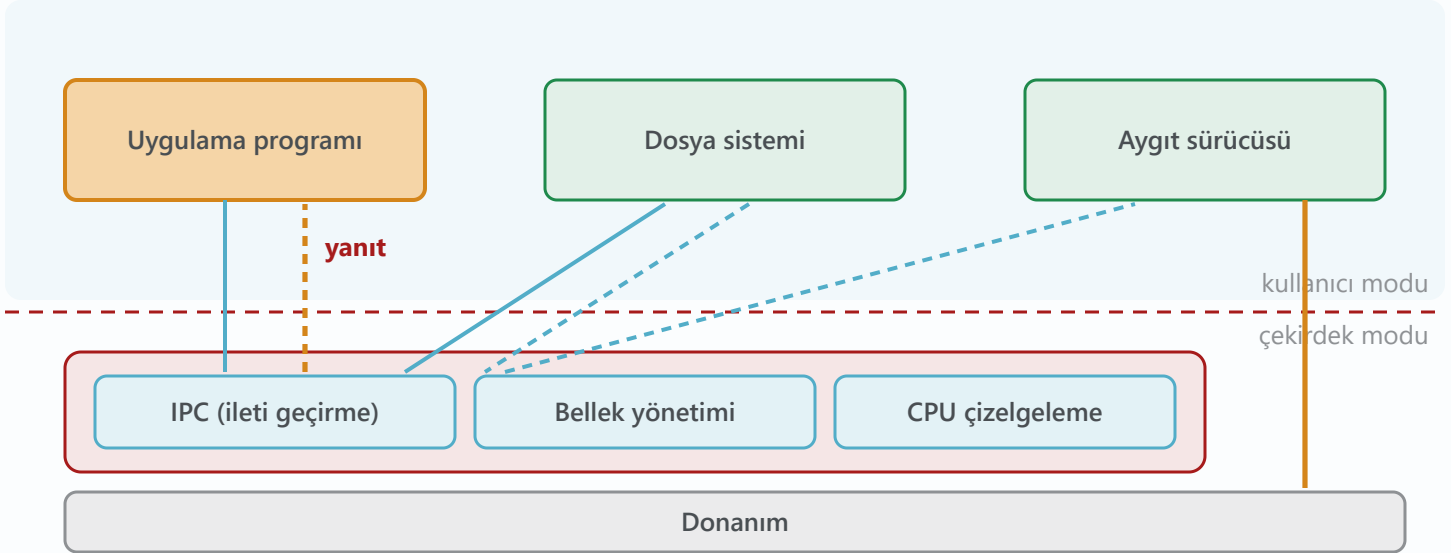
Çekirdek iletiyi, kullanıcı modundaki dosya sistemi sunucusuna teslim eder.

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.offset = 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.offset);
9     send(DISK, &blok);
10    receive(DISK, &veri);
11    m.sonuc = veri;
12    send(m.kaynak, &m);
13 }
```

İstemci
çalışıyor

Dosya sist. (FS)
bloke (receive)

Disk sürücüsü
bloke (receive)

kullanıcı modu

mikro çekirdek: IPC (iletiyi kopyalar, alıcıyı uyandırır)

ileti sayısı: 0

m = {OKU, 7, 0}

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

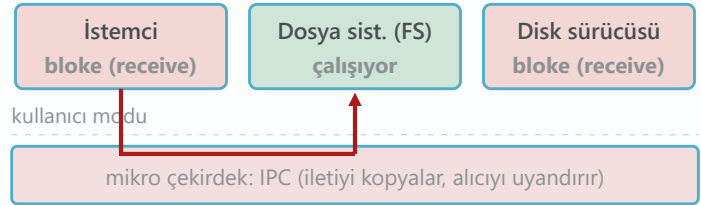
m.offset = 0

ÇAĞRI YIĞINI

istemci

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

```
1 /* istemci */
2 m.tur = OKU; m.dosya = 7; m.offset = 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.offset);
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.offset = 0

ÇAĞRI YİĞİNİ

FS

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

```
1 /* istemci */
2 m.tur = OKU; m.dosya = 7; m.offset = 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.offset);
9     send(DISK, &blok);
10    receive(DISK, &veri);
11    m.sonuc = veri;
12    send(m.kaynak, &m);
13 }
```

İstemci
bloke (receive)

Dosya sist. (FS)
çalışıyor

Disk sürücüsü
bloke (receive)

kullanıcı modu

mikro çekirdek: IPC (iletiyi kopyalar, alıcıyı uyandırır)

ileti sayısı: 1

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

Adım 3 / 8

DEĞİŞKENLER

m.kaynak = istemci

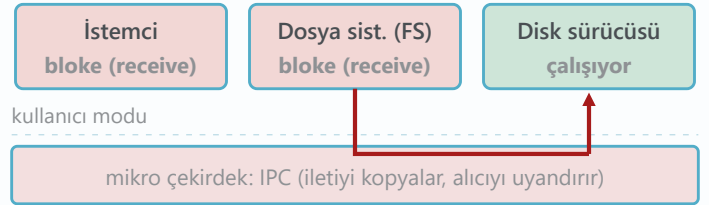
blok = 120

ÇAĞRI YİĞİNİ

FS

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

```
1 /* istemci */
2 m.tur = OKU; m.dosya = 7; m.offset = 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.offset);
9   ▶ send(DISK, &blok);
10  ▶ receive(DISK, &veri);
11  m.sonuc = veri;
12  send(m.kaynak, &m);
13 }
```



ileti sayısı: 2

blok 120 → DISK

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

Adım 4 / 8

DEĞİŞKENLER

m.kaynak = **istemci**

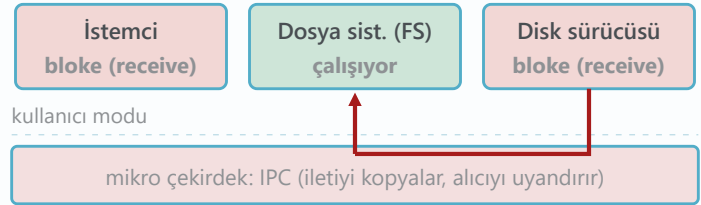
blok = **120**

ÇAĞRI YİĞİNİ

disk sürücüsü

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

```
1 /* istemci */
2 m.tur = OKU; m.dosya = 7; m.offset = 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.offset);
9     send(DISK, &blok);
10    ► receive(DISK, &veri);
11    ► 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 YİĞİNİ

FS

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

```
1 /* istemci */
2 m.tur = OKU; m.dosya = 7; m.offset = 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.offset);
9     send(DISK, &blok);
10    receive(DISK, &veri);
11    m.sonuc = veri;
12    ► send(m.kaynak, &m);
13 }
```



Adım 6 / 8

DEĞİŞKENLER

m.kaynak = **istemci**m.sonuc = **veri**

ÇAĞRI YİĞİNİ

FS

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

```
1 /* istemci */
2 m.tur = OKU; m.dosya = 7; m.offset = 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.offset);
9     send(DISK, &blok);
10    receive(DISK, &veri);
11    m.sonuc = veri;
12    send(m.kaynak, &m);
13 }
```

İstemci
çalışıyor

Dosya sist. (FS)
bloke (receive)

Disk sürücüsü
bloke (receive)

kullanıcı modu

mikro çekirdek: IPC (iletiyi kopyalar, alıcıyı uyandırır)

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.offset = 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.offset);
9     send(DISK, &blok);
10    receive(DISK, &veri);
11    m.sonuc = veri;
12    send(m.kaynak, &m);
13 }
```

İstemci
çalışıyor

Dosya sist. (FS)
bloke (receive)

Disk sürücüsü
bloke (receive)

kullanıcı modu

mikro çekirdek: IPC (iletiyi kopyalar, alıcıyı uyandırır)

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 çağrılılarıyla biterdi.

Adım 8 / 8

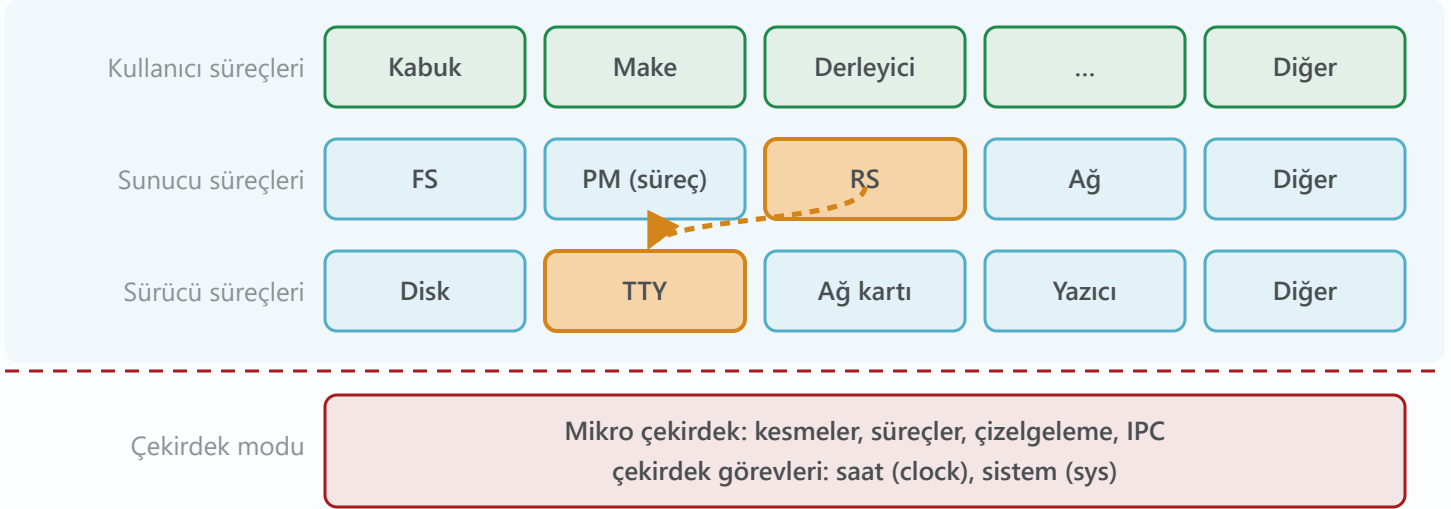
DEĞİŞKENLER

m.sonuc = veri

ÇAĞRI YIĞINI

istemci

MINIX 3: Bir Mikro Çekirdek Örneği



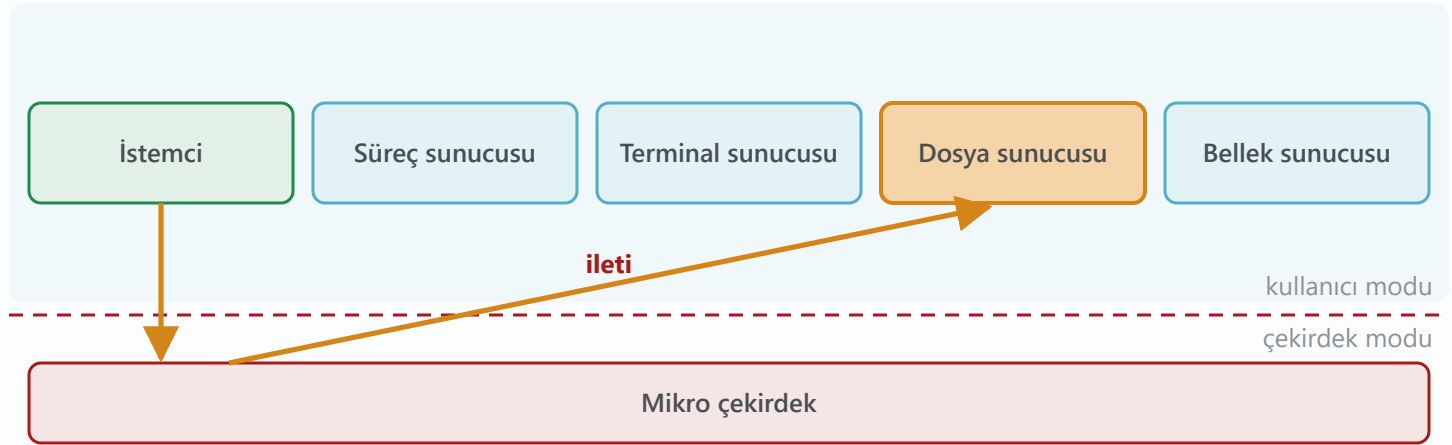
Reenkarnasyon sunucusu (RS)

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

Bölünmüş ayrıcalık

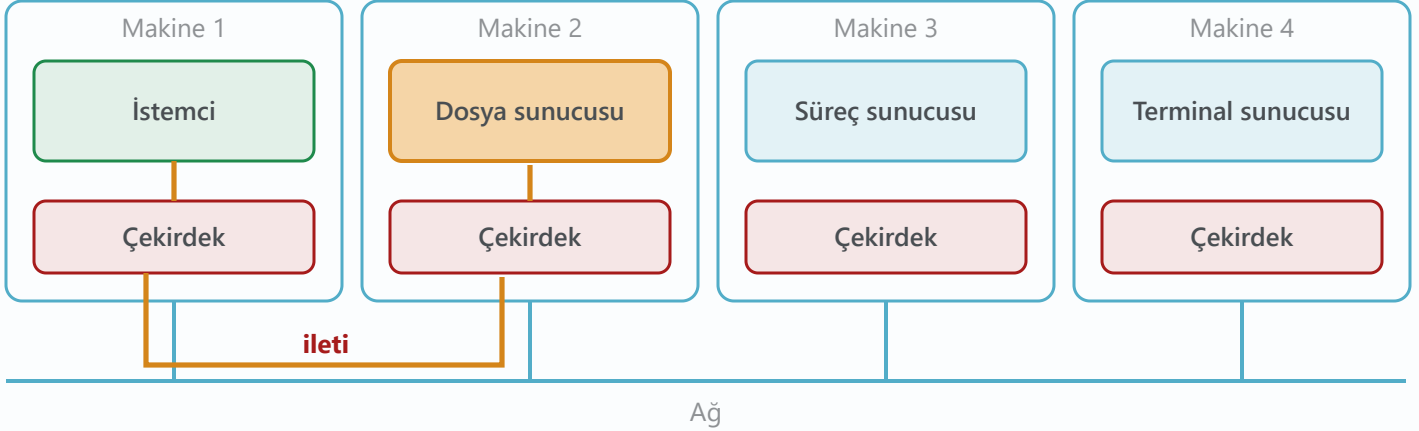
Sürücü kullanıcı modundadır; G/Ç portlarına yalnızca çekirdek çağrısıyla, izin verilen ölçüde erişir. Hatalı bir sürücü belleği bozamaz.

İstemci-Sunucu Modeli



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

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).

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.

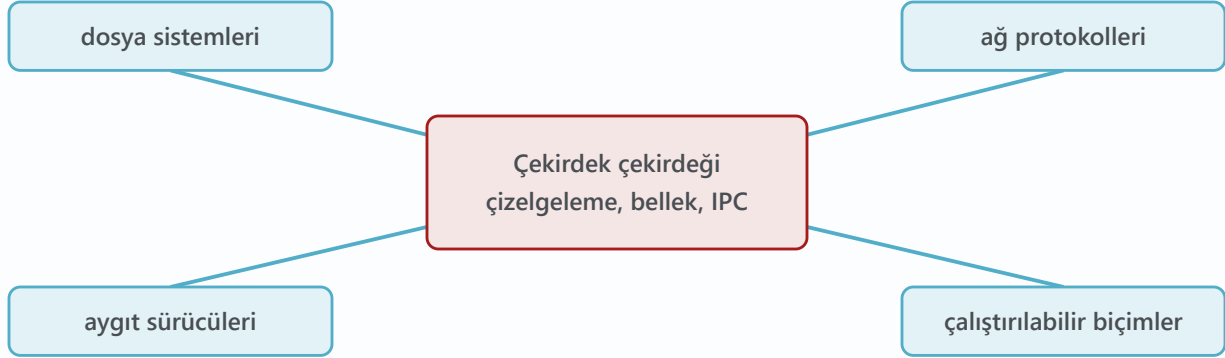
Linus Torvalds: monolitik

- Monolitik çekirdeğin tasarımı ve gerçekleştirmesi 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 sürücüler (FUSE, VFIO) ve eBPF ile esneklik kazandı; mikro çekirdekler ise güvenlik-kritik ve gömülü alanda (QNX, seL4) yaygınlaştı.

Modüler Yapı: Yüklenebilir Çekirdek Modülleri

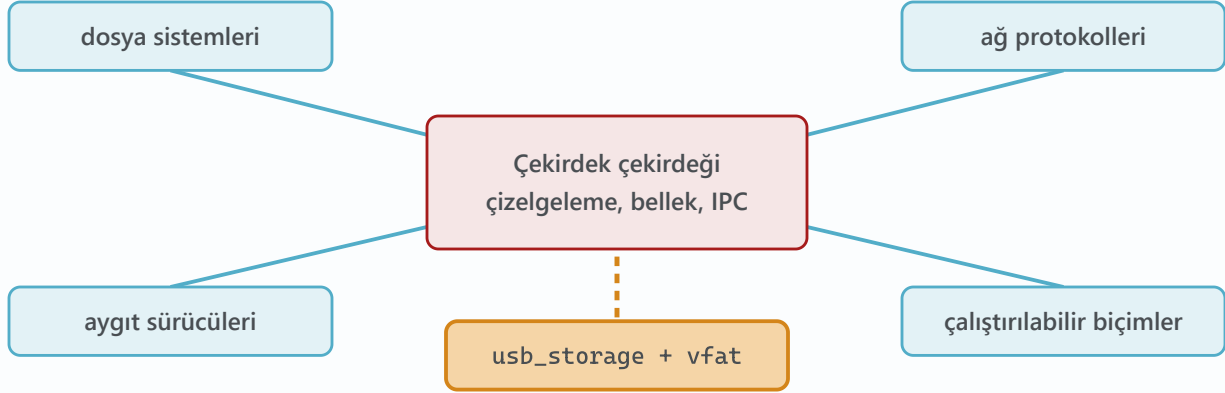
- Çekirdekte yalnızca temel bileşenler bulunur; diğer hizmetler açılıшта veya **çalışma anında modül** olarak yüklenir (Linux LKM, FreeBSD, Solaris).
- Katmanlıya benzer (tanımlı arayüzler) ama daha esnektir: her modül diğerini çağırabilir. Mikro çekirdeğe benzer ama ileti gerekmez, modüller **çekirdek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



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

Modüler Yapı: Yüklenebilir Çekirdek Modülleri

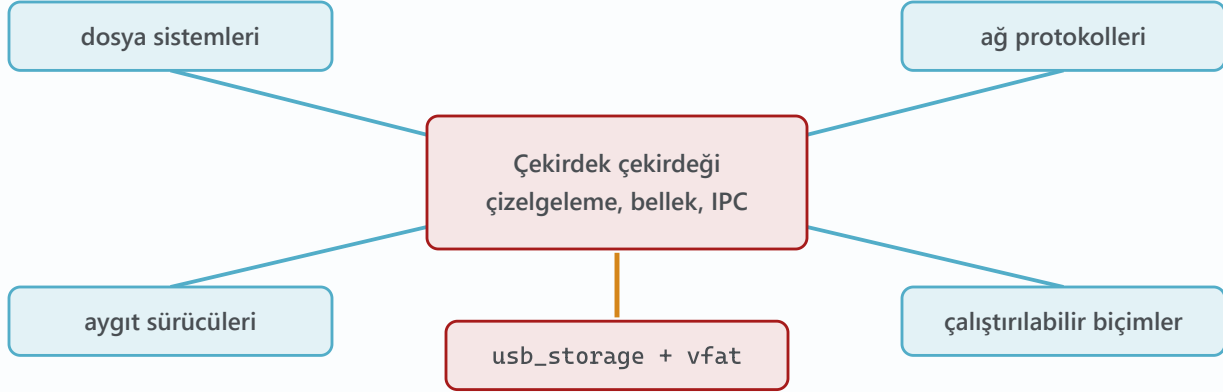
- Çekirdekte yalnızca temel bileşenler bulunur; diğer hizmetler açılıшта veya **çalışma anında modül** olarak yüklenir (Linux LKM, FreeBSD, Solaris).
- Katmanlıya benzer (tanımlı arayüzler) ama daha esnektir: her modül diğerini çağırabilir. Mikro çekirdeğe benzer ama ileti gerekmez, modüller **çekirdek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



USB bellek takılır: udev olayı `modprobe usb-storage` ve `vfat` modüllerini (bağımlılıklarıyla) yükletir.

Modüler Yapı: Yüklenebilir Çekirdek Modülleri

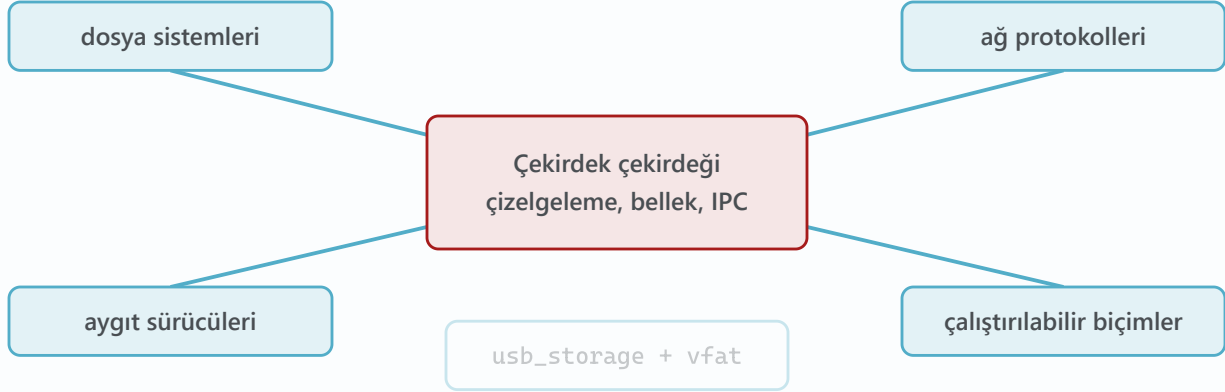
- Çekirdekte yalnızca temel bileşenler bulunur; diğer hizmetler açılıшта veya **çalışma anında modül** olarak yüklenir (Linux LKM, FreeBSD, Solaris).
- Katmanlıya benzer (tanımlı arayüzler) ama daha esnektir: her modül diğerini çağırabilir. Mikro çekirdeğe benzer ama ileti gerekmez, modüller **çekirdek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



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

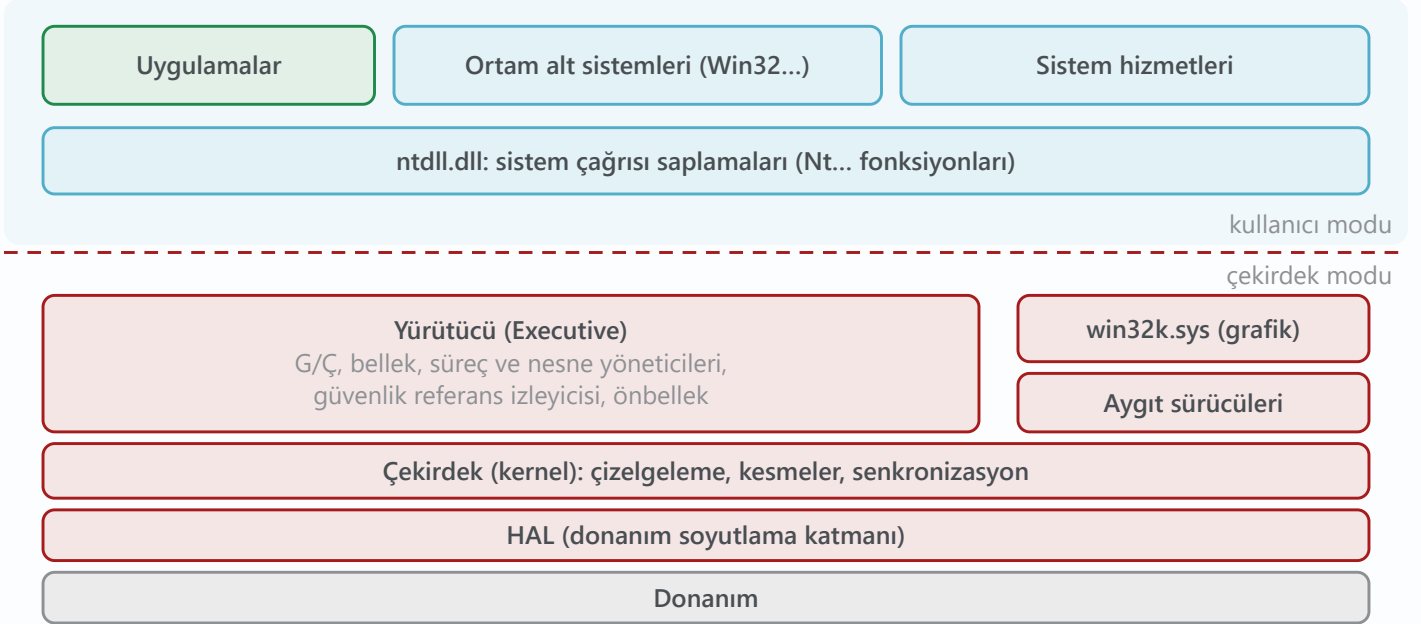
Modüler Yapı: Yüklenebilir Çekirdek Modülleri

- Çekirdekte yalnızca temel bileşenler bulunur; diğer hizmetler açılıшта 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.



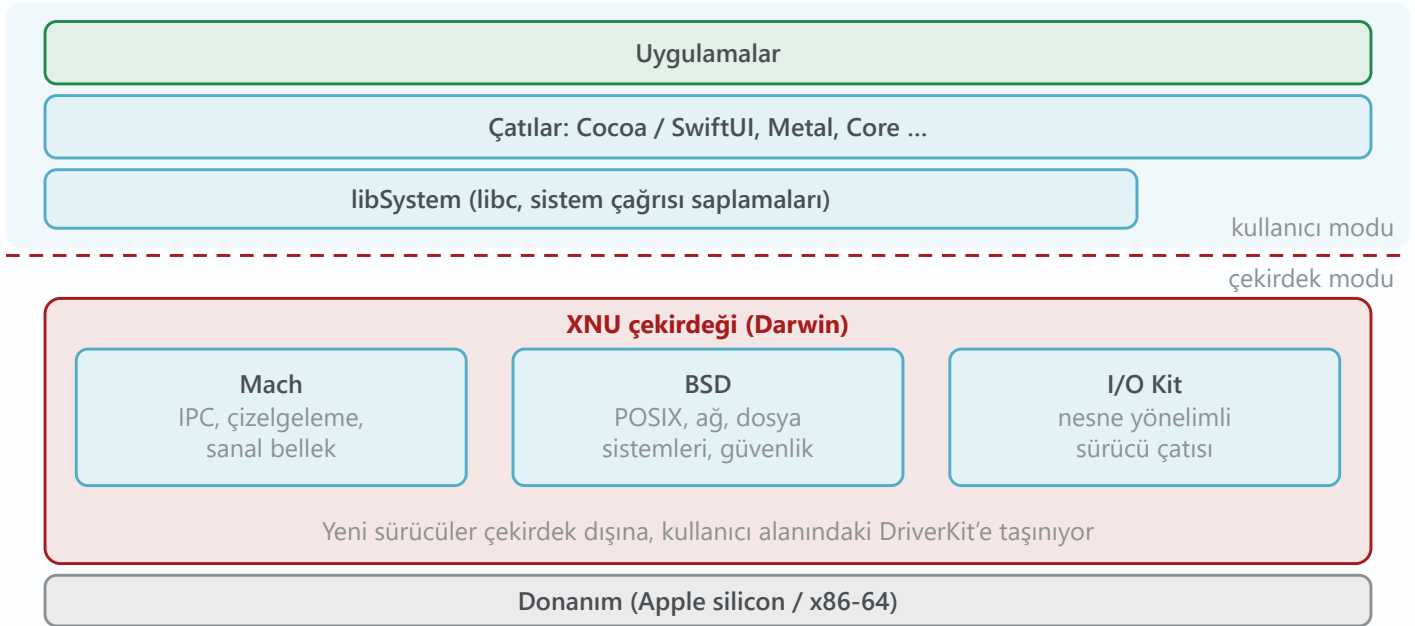
Kullanım sayısı 0 olunca `modprobe -r` (veya `rmmmod`) modülü bellekten çıkarır. Modüllerin çekirdek sürümüyle uyumlu olması gerekir.

Hibrit Çekirdek: Windows ve macOS



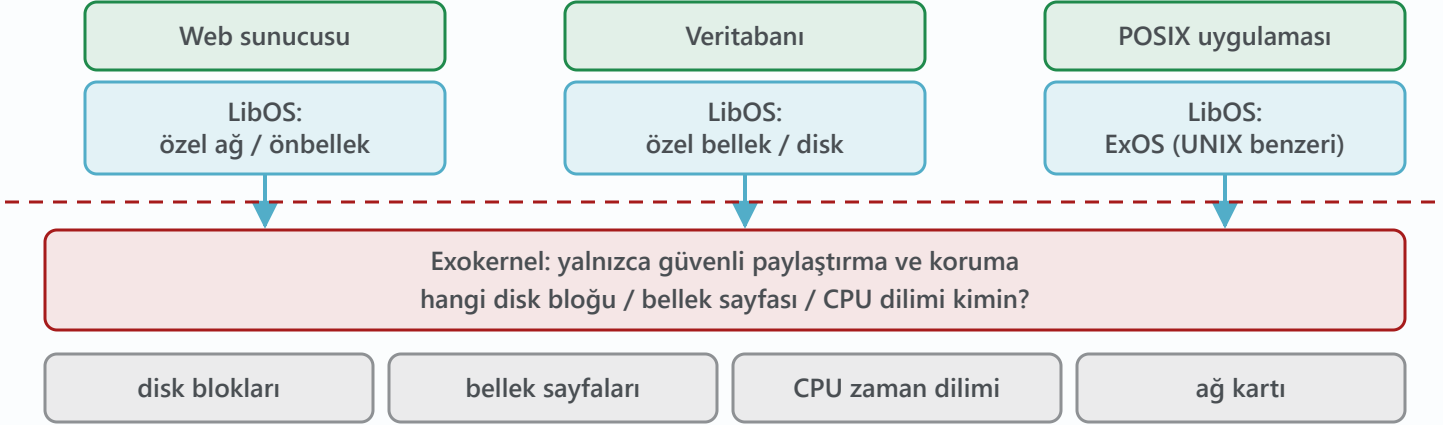
Windows NT: mikro çekirdek fikirleri (alt sistemler, nesne/ileti temelli tasarım) korunmuş, ama performans için yürütücü, grafik ve sürücüler çekirdek moduna alınmıştır.

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.
- **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 başarımlı; ama uygulama geliştirmek karmaşıktır. Fikir bugün **unikernel**'lerde ve DPDK / SPDK gibi çekirdeği atlayan G/Ç'de yaşar.

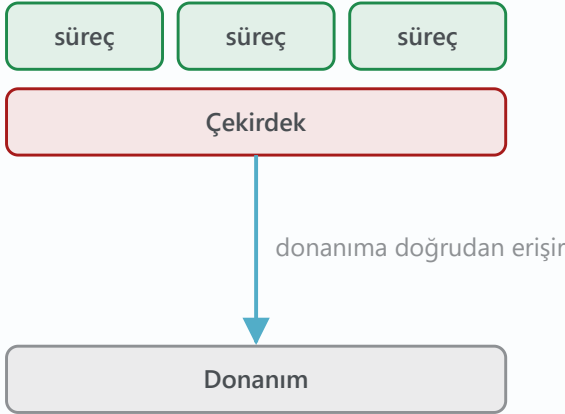
Yapıların Karşılaştırılması

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

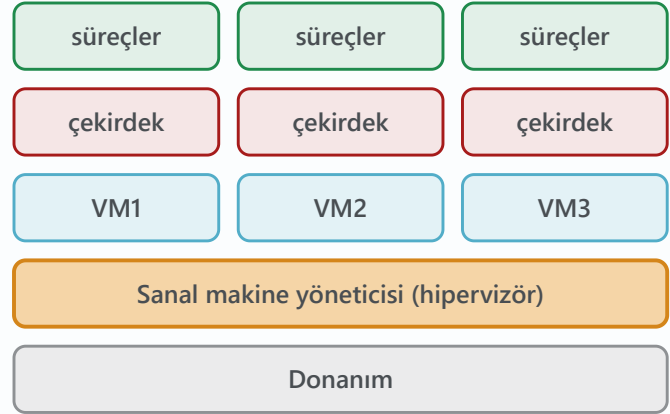
Gerçek sistemler saf değildir: Linux monolitik + modüler, Windows ve macOS hibrittir; hemen hepsi bir ölçüde katmanlıdır.

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.

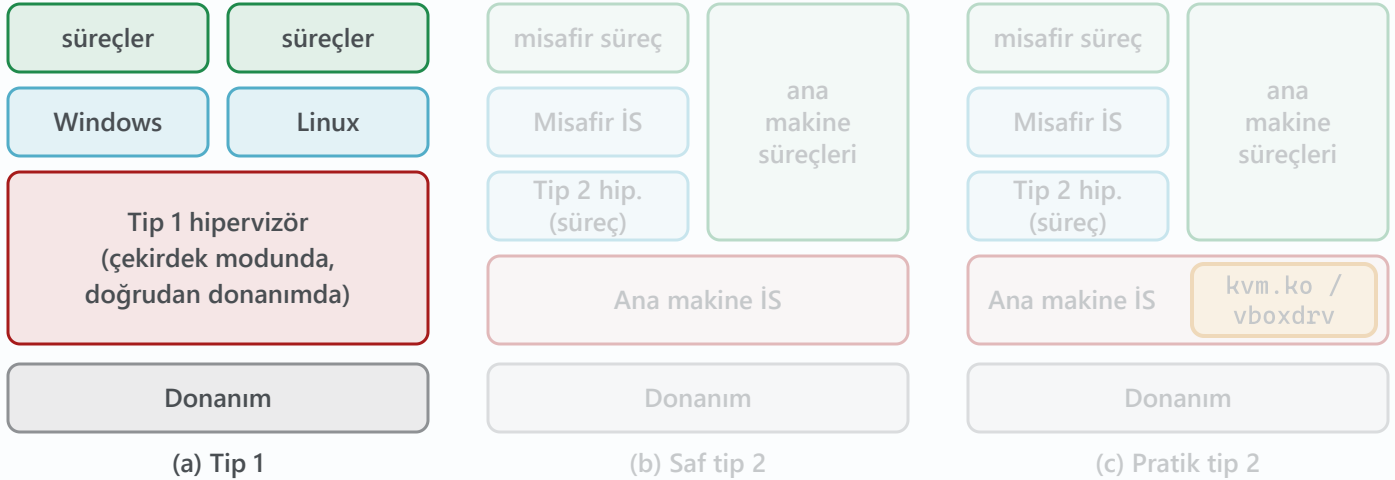


(a) Sanal olmayan makine



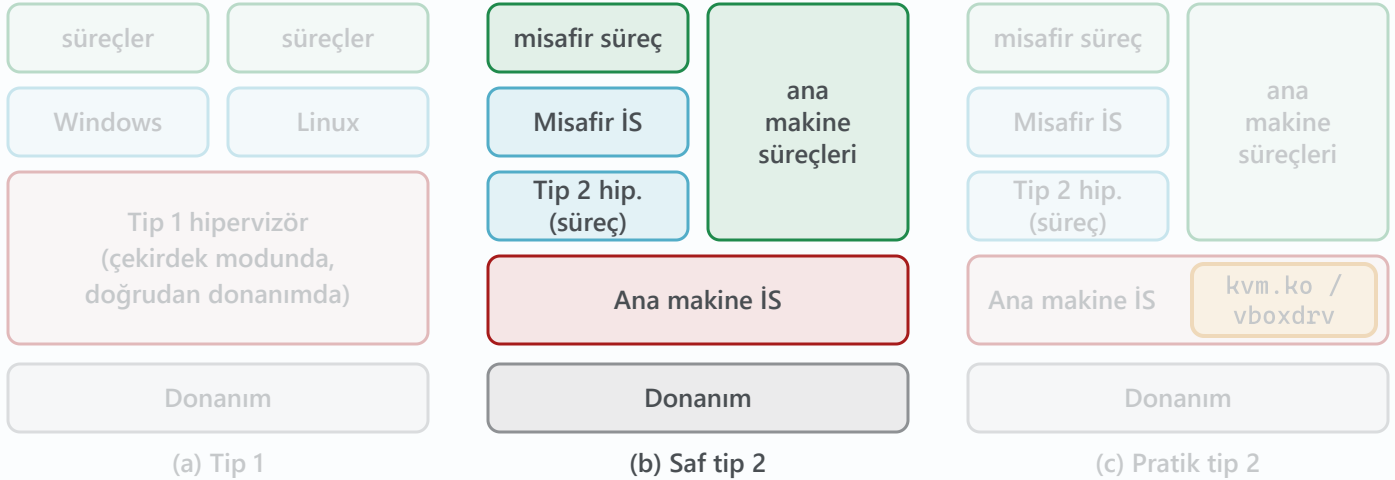
(b) Sanal makineler

Hipervizör Türleri



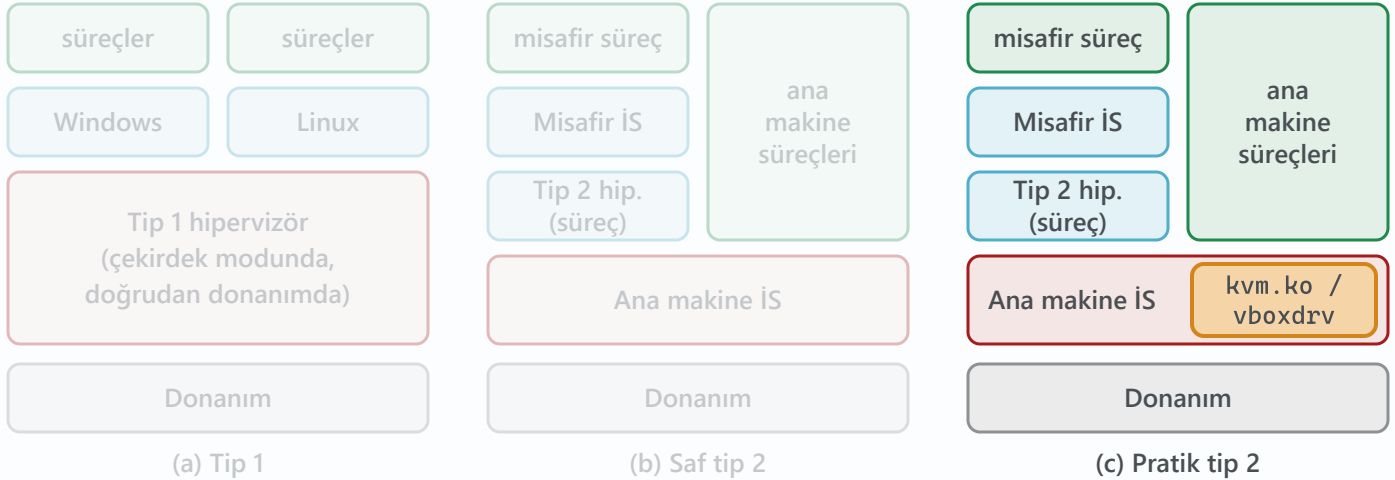
Tip 1 (bare-metal): hipervizör doğrudan donanımda, en ayrıcalıklı modda çalışır; misafir işletim sistemleri onun üzerindedir. Veri merkezlerinin standart çözümüdür.

Hipervizör Türleri



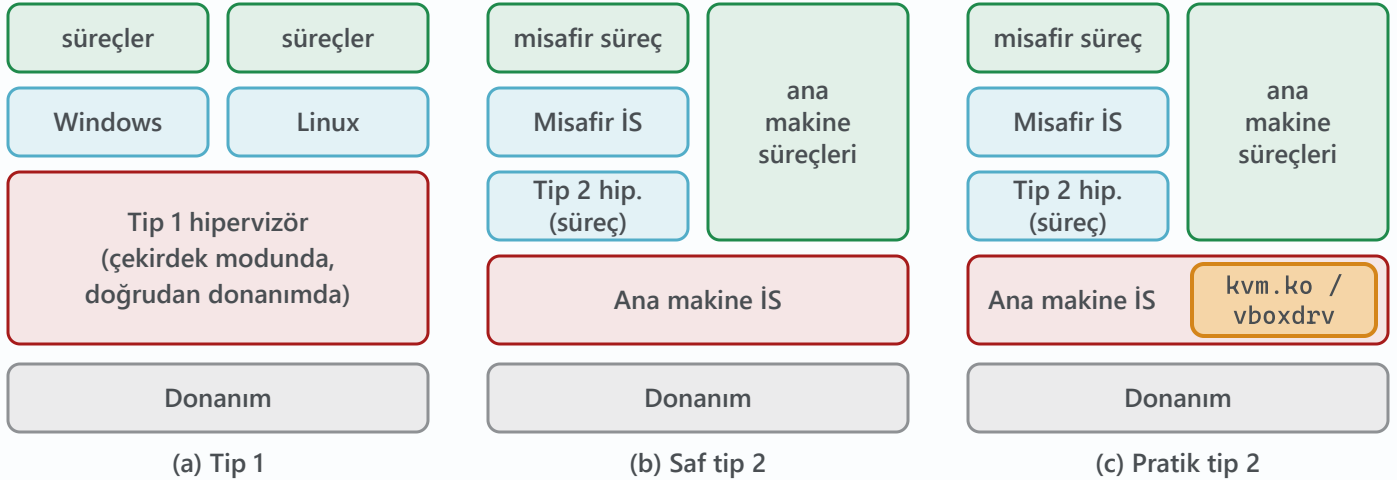
Saf tip 2: hipervizör, ana makine işletim sistemi üzerinde sıradan bir süreçtir; misafir komutlarını yorumlar veya ikili çeviriyle çalıştırır (ör. KVM'siz QEMU). Esnek ama yavaştır.

Hipervizör Türleri



Pratik tip 2: hipervizör yine bir uygulamadır, ancak ana işletim sistemine yüklenen bir çekirdek modülü sayesinde donanım destekli sanallaştırmayı (Intel VT-x, AMD-V, ARM EL2) kullanır.

Hipervizör Türleri



Sınırlar bulanıklaştı: KVM, Linux çekirdeğinin kendisini tip 1 hipervizöre dönüştürür; Hyper-V etkinleşince Windows bile bir kök bölümde (VM) çalışır.

Tip 1 ve Tip 2 Hipervizörler

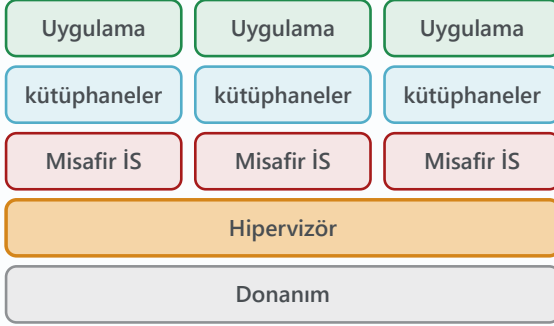
Tip 1: “bare-metal”

- İşletim sisteminden bağımsız, doğrudan donanım üzerinde çalışır.
- Fiziksel sunucuları sanal makinelere bölmek, kaynakları verimli kullanmak ve izlemek için.
- **VMware ESXi**: kurumsal veri merkezleri.
- **Microsoft Hyper-V**: Windows Server rolü (ayrı ücretsiz “Hyper-V Server” 2019’dan sonra sürdürülmedi).
- **Xen** (XCP-ng) ve **KVM**: açık kaynak; büyük bulutların (AWS Nitro, Google Cloud) temeli KVM’dir.

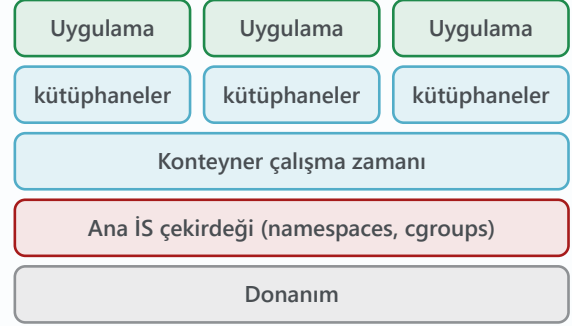
Tip 2: “hosted”

- Bir ana makine işletim sistemi üzerinde uygulama olarak çalışır.
- Kişisel bilgisayarlarda deneme, eğitim ve geliştirme ortamları için.
- **Oracle VirtualBox**: ücretsiz, açık kaynak, çok platformlu.
- **VMware Workstation Pro / Fusion**: 2024’ten beri ücretsiz.
- **Parallels Desktop** ve **UTM** (QEMU): macOS üzerinde.
- G/Ç yolu ana işletim sisteminden geçtiği için ek yük daha fazladır; donanım desteğiyle CPU farkı küçüktür.

Sanal Makine mi, Konteyner mi?



Sanal makineler: her biri kendi çekirdeğiyle



Konteynerler: tek çekirdek paylaşılr

Konteyner

İşletim sistemi düzeyinde sanallaştırma: süreçler namespaces ile yalıtılır, cgroups ile sınırlanır. Saniyenin altında başlar (Docker, Podman, Kubernetes).

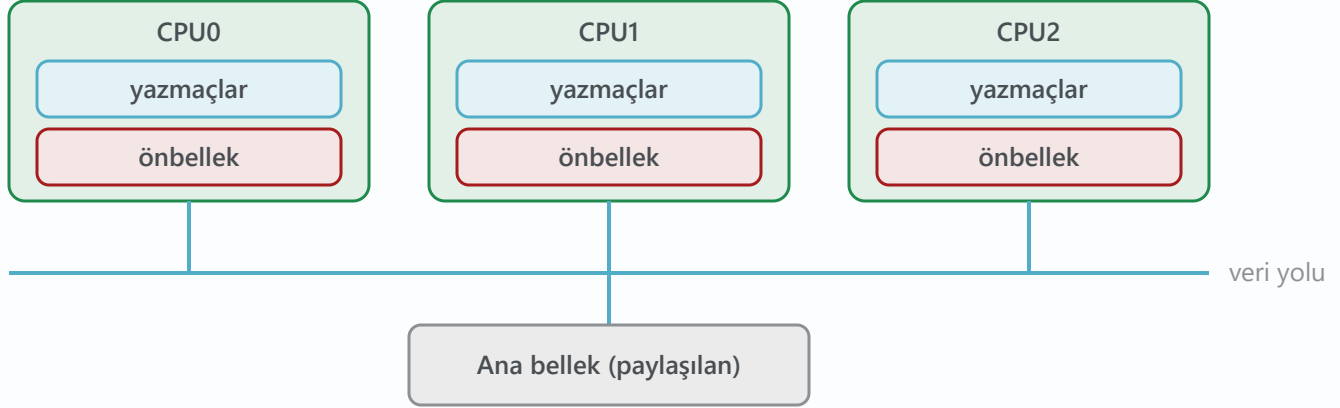
Sanal makine

Tam yalıtım, farklı işletim sistemi çalıştırabilir; ama her VM kendi çekirdeğini ve belleğini taşır.

Ara çözümler

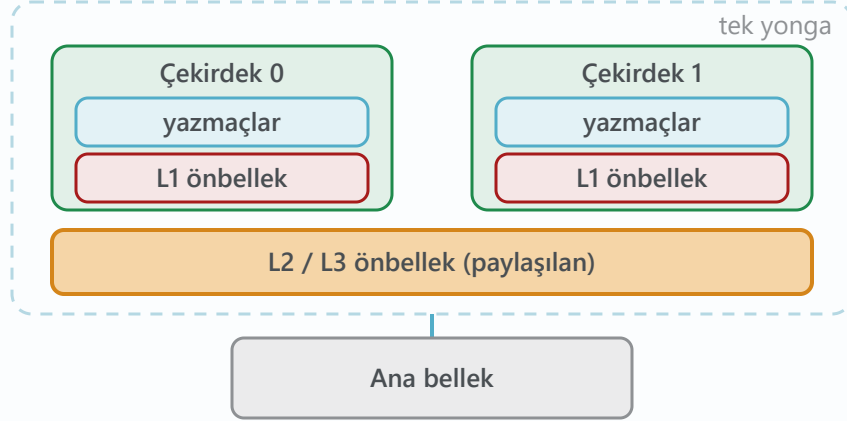
Firecracker microVM, Kata Containers, gVisor; WSL 2 da hafif bir Hyper-V VM'inde gerçek Linux çekirdeği çalıştırır.

Simetrik Çok İşlemcili ve Çok Çekirdekli Mimari



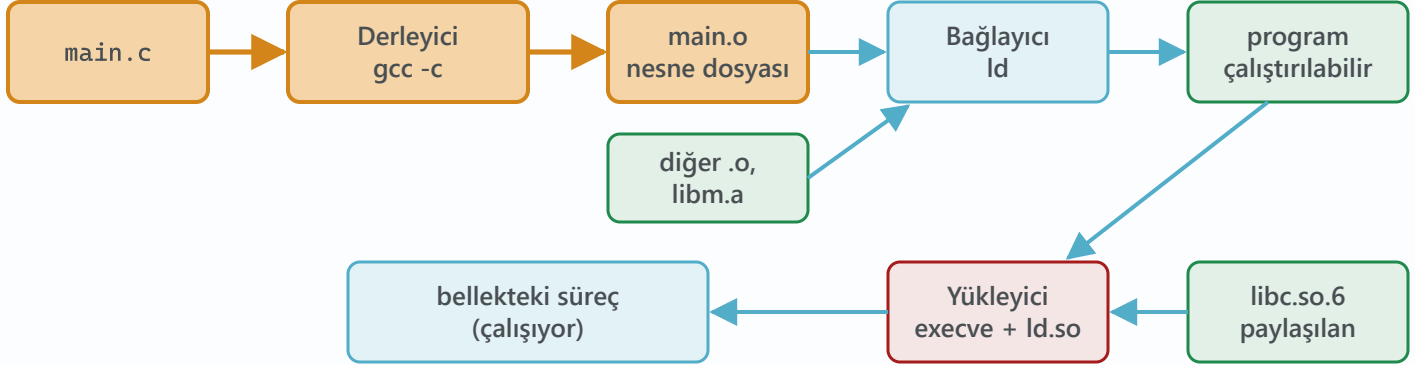
SMP: her CPU kendi yazmaçlarına ve önbelleğine sahiptir, belleği paylaşır; hepsi işletim sistemi dâhil her işi yapabilir. Çekirdek veri yapıları kilitlerle korunmalıdır (Linux'un "büyük çekirdek kilidi" 2011'de tamamen kaldırıldı).

Simetrik Çok İşlemcili ve Çok Çekirdekli Mimari



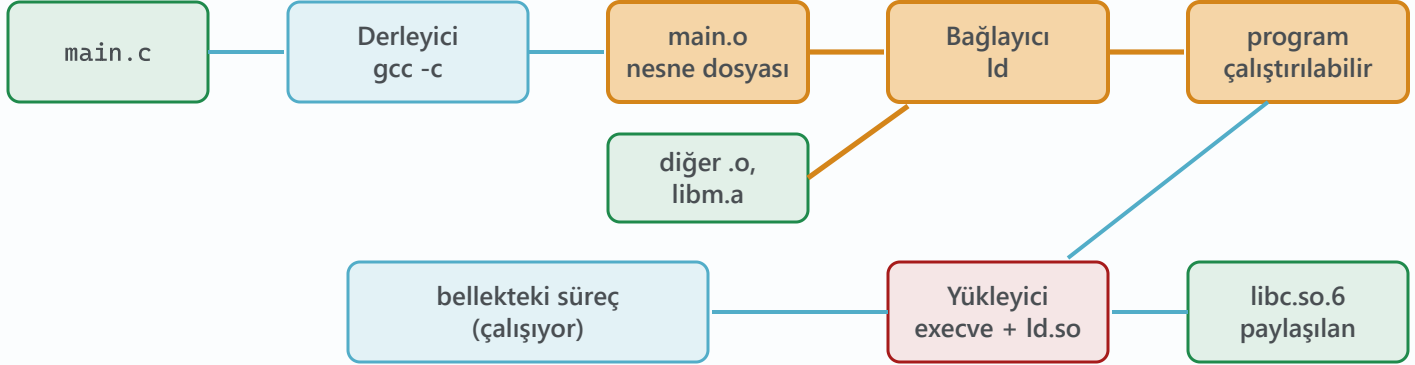
Çok çekirdekli yonga: çekirdekler aynı yongada, yakın önbellek paylaşımıyla daha hızlı ve az enerjiyle haberleşir. 2026'da yaygın olan P/E (performans/verimlilik) çekirdekleri çizelgeleyiciye yeni görevler yükler.

Derleme, Bağlama ve Yükleme



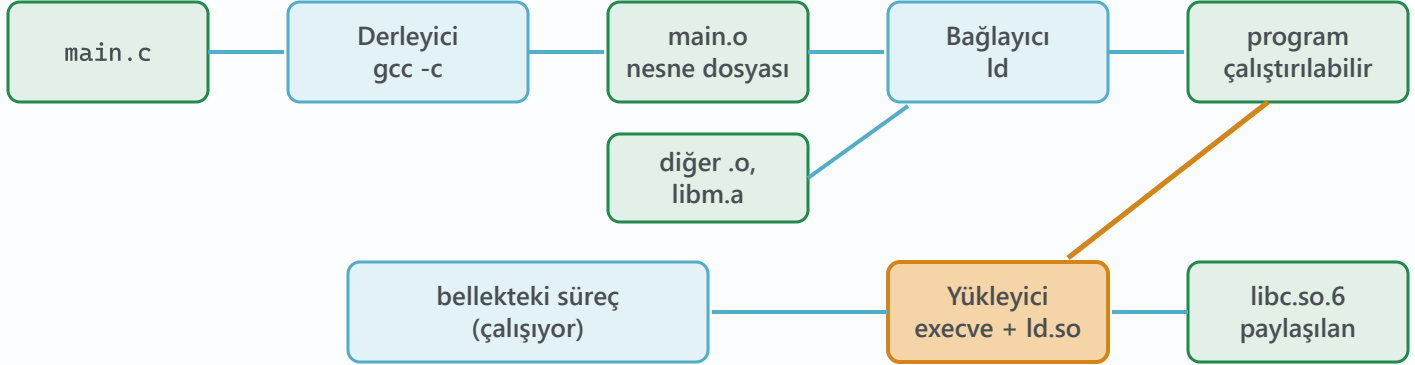
Derleyici kaynak kodu **yeniden konumlandırılabilir nesne dosyasına** çevirir: adresler henüz kesin değildir, dış semboller (ör. `printf`) çözülmemiştir.

Derleme, Bağlama ve Yükleme



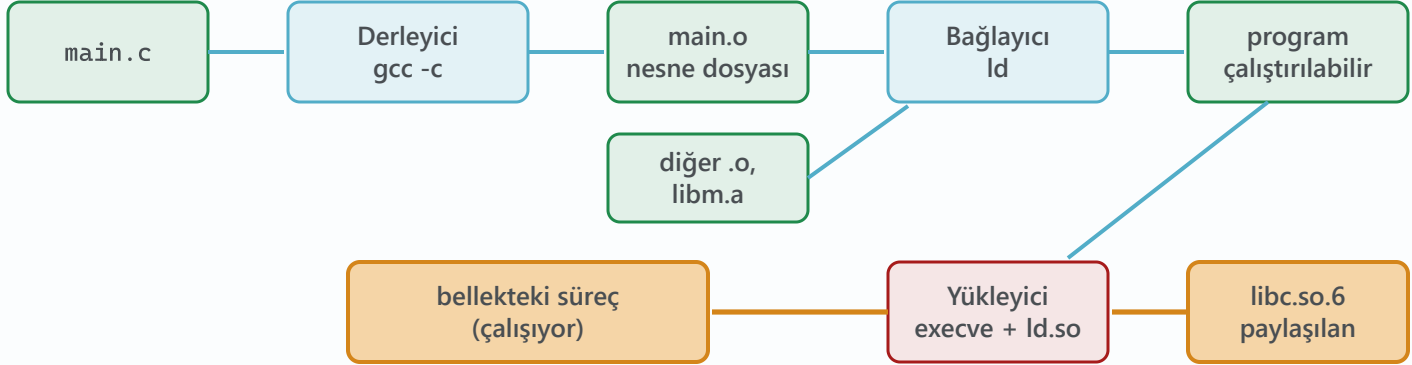
Bağlayıcı nesne dosyalarını ve statik kütüphaneleri birleştirir, sembolleri çözer, adresleri düzeltir ve çalıştırılabilir dosya üretir.

Derleme, Bağlama ve Yükleme



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

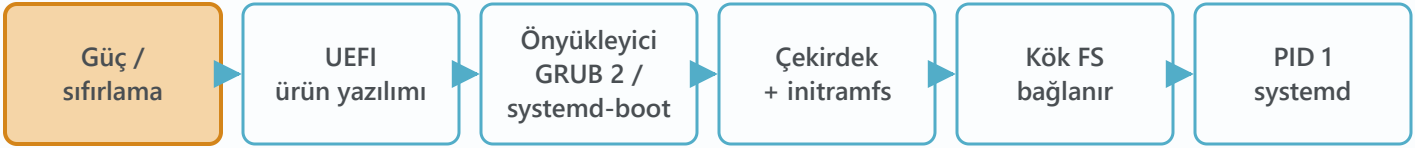
Derleme, Bağlama ve Yükleme



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.

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir. Linux'ta `systemd-analyze` her aşamanın süresini gösterir.



CPU sabit bir adresten ürün yazılımını çalıştırmaya başlar.

Güç verilince CPU sıfırlanır ve kalıcı bellekteki ürün yazılımına atlar.

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir. Linux'ta `systemd-analyze` her aşamanın süresini gösterir.



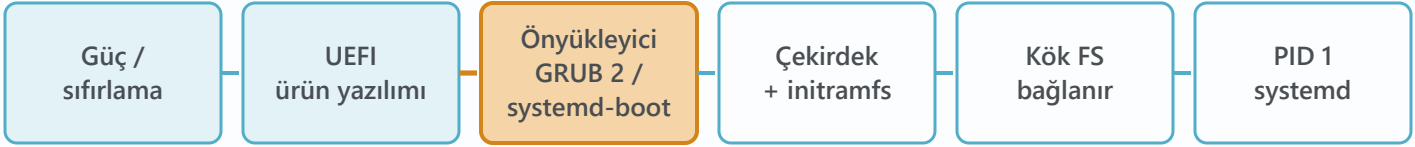
POST ile donanım sınanır ve ilklenir; EFI sistem bölümündeki önyükleyici bulunur.
Secure Boot her aşamanın imzasını doğrular.

UEFI donanımı ilkler; GPT diskteki EFI sistem bölümünden (FAT32) önyükleyiciyi yükler.

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir.

Linux'ta `systemd-analyze` her aşamanın süresini gösterir.



Önyükleyici çekirdek seçimini sunar, çekirdeği ve initramfs'i belleğe yükler.

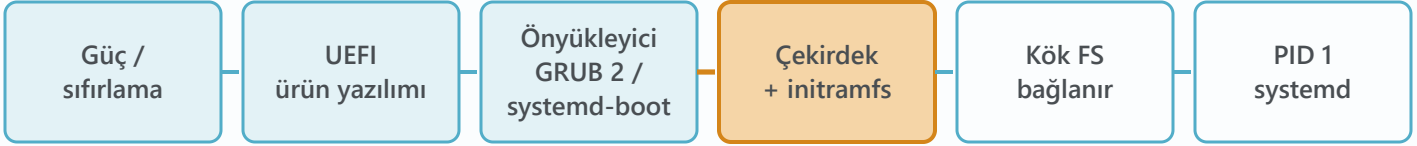
Eski sistemler: BIOS + MBR.

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

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir.

Linux'ta `systemd-analyze` her aşamanın süresini gösterir.

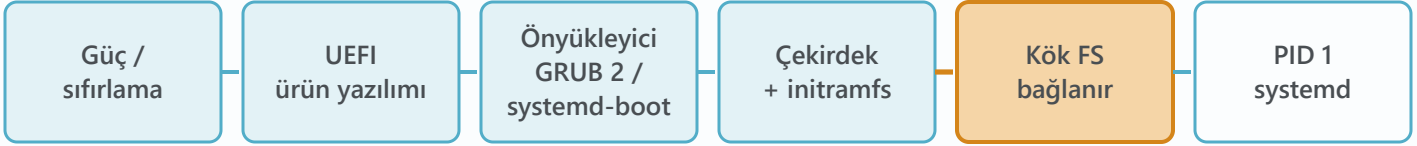


Çekirdek kendini açar, donanımı tanır, sürücülerini yükler;
initramfs geçici bir kök dosya sistemi sağlar.

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

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir. Linux'ta `systemd-analyze` her aşamanın süresini gösterir.

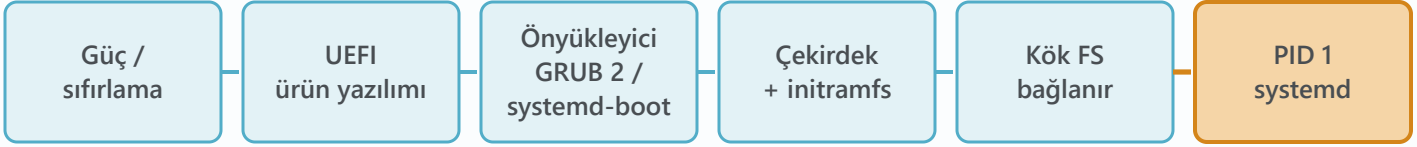


Gerçek kök dosya sistemi (ör. şifreli disk açılarak) bağlanır.

Gerçek kök dosya sistemi bağlanır ve initramfs bırakılır.

İşletim Sisteminin Açılışı (Boot)

Açılış, çekirdeği belleğe yükleyip ilk kullanıcı sürecini başlatana kadar geçen süreçtir. 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.

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. Tip 1 ve tip 2 hipervizörü ayırt eden nedir? KVM nereye konur?

► Cevapları göster

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](#)