Bölüm 1: Giriş

İşletim Sistemleri

İşletim sistemi nedir? | Kullanıcı ve çekirdek modu | Genişletilmiş makine | Kaynak yöneticisi | Tarihçe | İşlemci ve yazmaçlar | Bellek hiyerarşisi | Çok çekirdekli yongalar | Yoklama, kesme, DMA | İS çeşitleri | Politika ve mekanizma | Programdan sürece

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_01_Giris.html

GİRİŞ

Öğrenme Çıktıları

Tanım ve konum

İşletim sisteminin donanım ile uygulamalar arasındaki yerini ve iki temel rolünü (genişletilmiş makine, kaynak yöneticisi) açıklamak.

Koruma

Çekirdek modu ile kullanıcı modunu ve aralarındaki kontrollü geçişi (sistem çağrısı, kesme, istisna) açıklamak.

Tarihsel gelişim

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

Donanım temelleri

İşlemci ve yazmaçların, bellek hiyerarşisinin ve çok çekirdekli yongaların İS'yi nasıl etkilediğini yorumlamak.

G/Ç yöntemleri

Yoklama, kesme ve DMA arasındaki temel farkı söylemek; kesme ile tuzağı ayırmak.

Kavramlar

İS çeşitlerini, politika–mekanizma ayrımını ve programın çalıştırılabilir dosyaya dönüşmesini örneklerle açıklamak.

GİRİŞ

İşletim Sistemi Nedir?

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

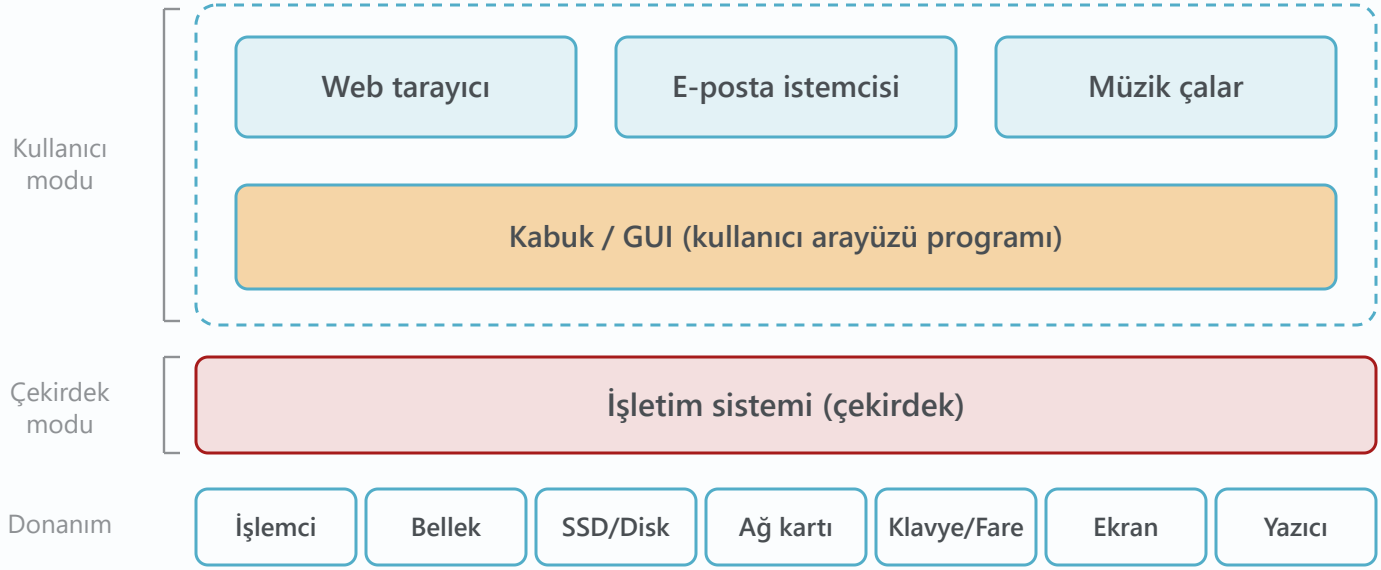
Masaüstü
Windows 11, macOS, Linux dağıtımları (Ubuntu, Fedora), ChromeOS

Mobil
Android, iOS / iPadOS, HarmonyOS

Sunucu ve bulut
Linux, Windows Server, FreeBSD

Gömülü ve IoT
Zephyr, FreeRTOS, Contiki-NG, VxWorks, QNX

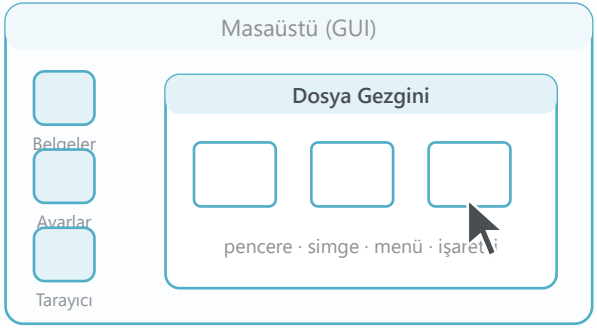
Bilgisayar Sisteminin Bileşenleri



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

Kabuk ve GUI





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

GİRİŞ

Kabuk mu, GUI mi?

Kabuk

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

GUI

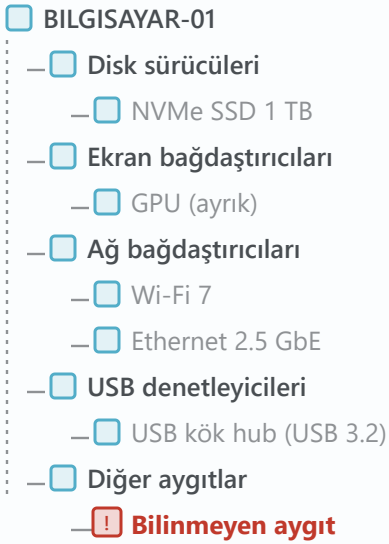
+ **Sezgisel** ve keşfedilebilir: komut ezberlemek gerekmez.
+ Görsel içerik (fotoğraf, harita, tasarım) için doğal.
+ Geniş bir kullanıcı kitlesine ulaşır; erişilebilirlik araçları.
+ Dokunmatik ve sesli arayüzler de bu ailededir.

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

GİRİŞ

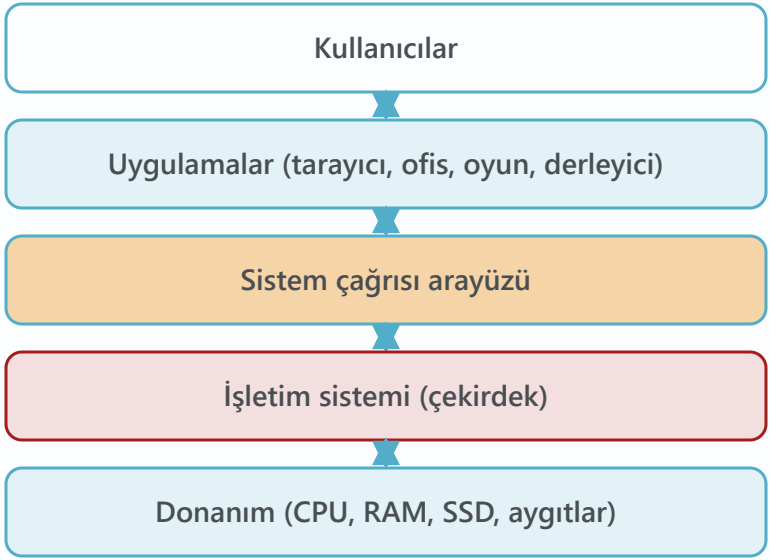
Aygıt Yöneticisi

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



İşletim Sisteminin Konumu

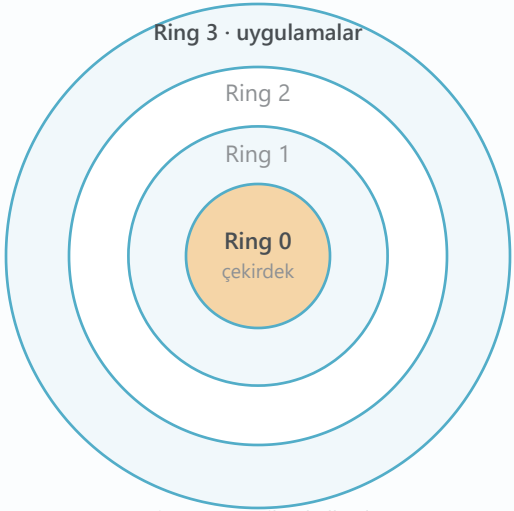
- İS, **donanım** (fiziksel bileşenler: CPU, RAM, disk) ile **yazılım** (bilgisayarın yapacağı işleri tarif eden kod) arasındaki arayüzdür.
- Uygulamalar donanıma doğrudan değil, İS'nin sunduğu **sistem çağrıları** üzerinden erişir.



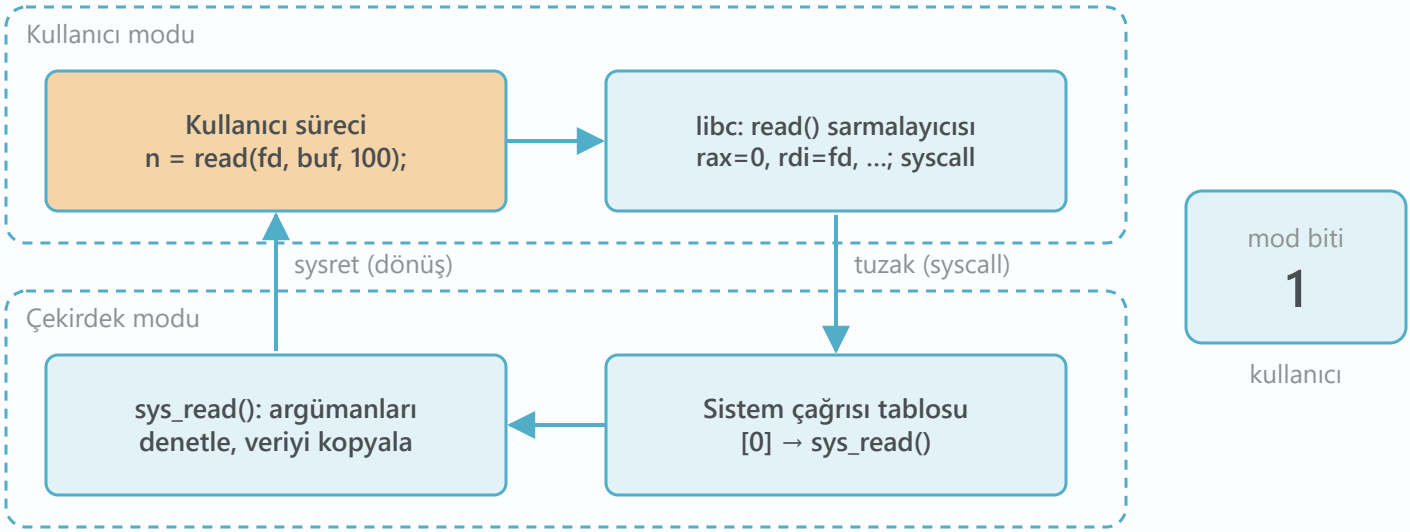
- yazılımın donanımı kullanmasını kontrol eder
- donanım kullanımını optimize eder
- sistem güvenliğini ve yalıtımı sağlar

Çekirdek Modu ve Kullanıcı Modu

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

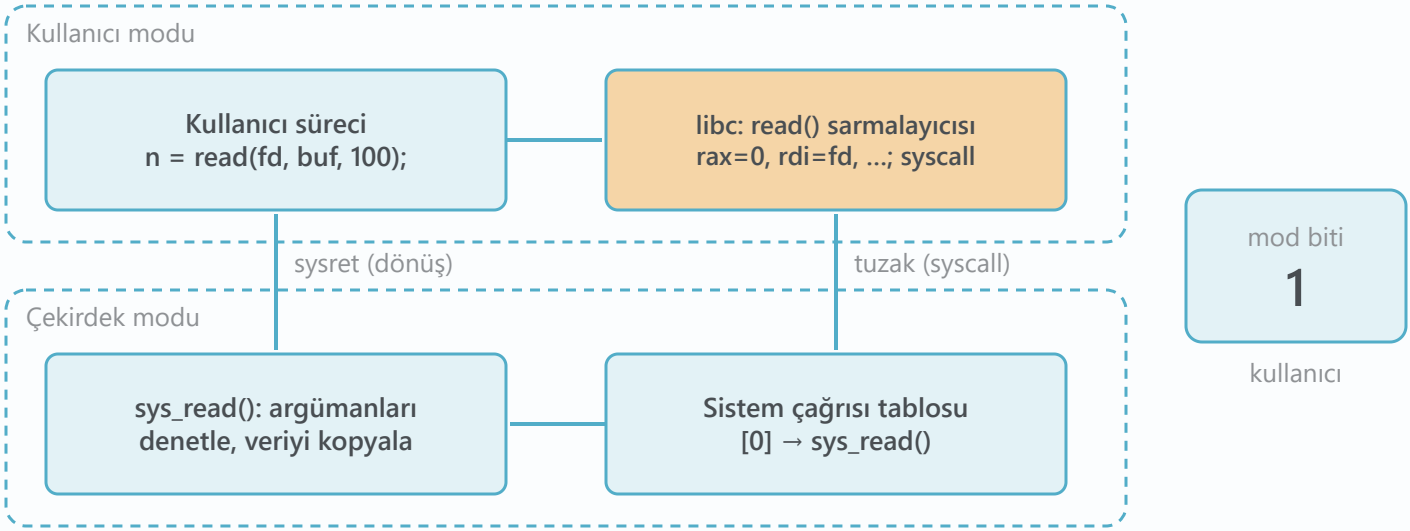


Mod Geçişi Adım Adım: `read()` Sistem Çağrısı



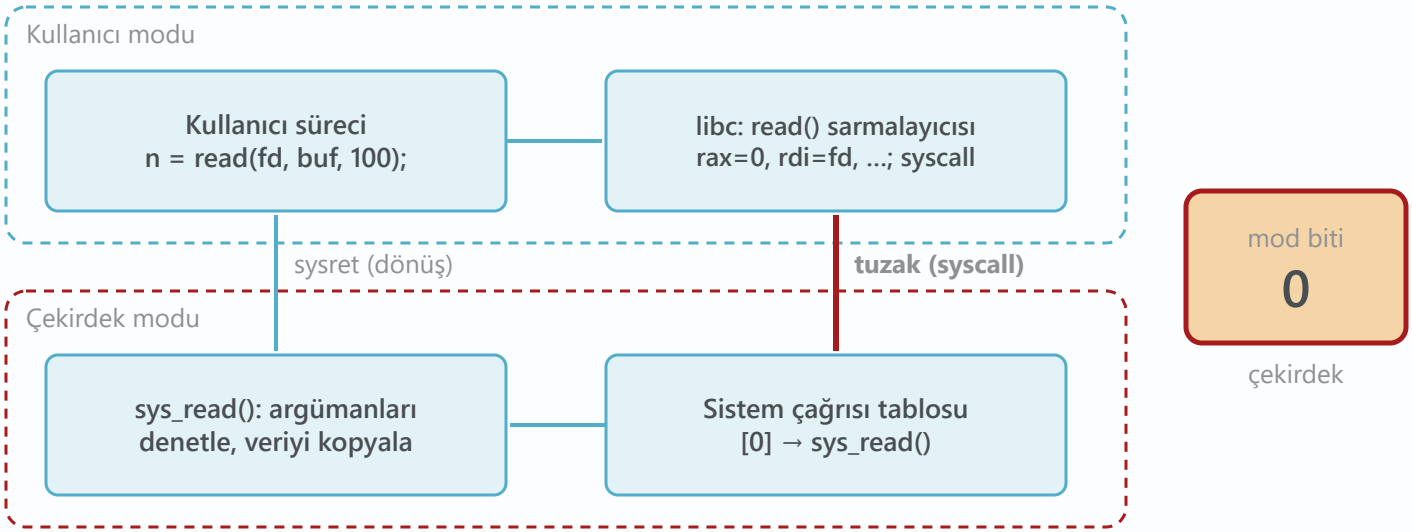
Süreç **kullanıcı modunda** çalışıyor (mod biti = 1). Bir dosyadan okumak istiyor; diske kendisi erişemez.

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



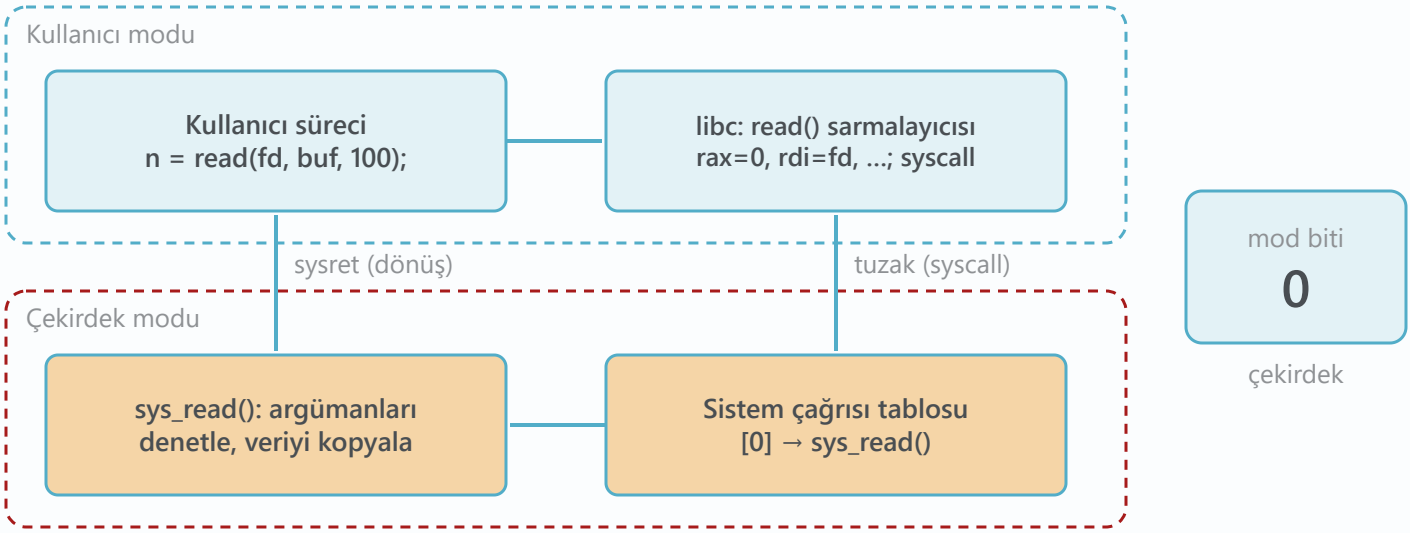
C kütüphanesindeki `read()` sarmalayıcısı çağrı numarasını (x86-64 Linux: read = **0**, `rax`) ve argümanları yazmaçlara koyar, `syscall` komutunu çalıştırır.

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



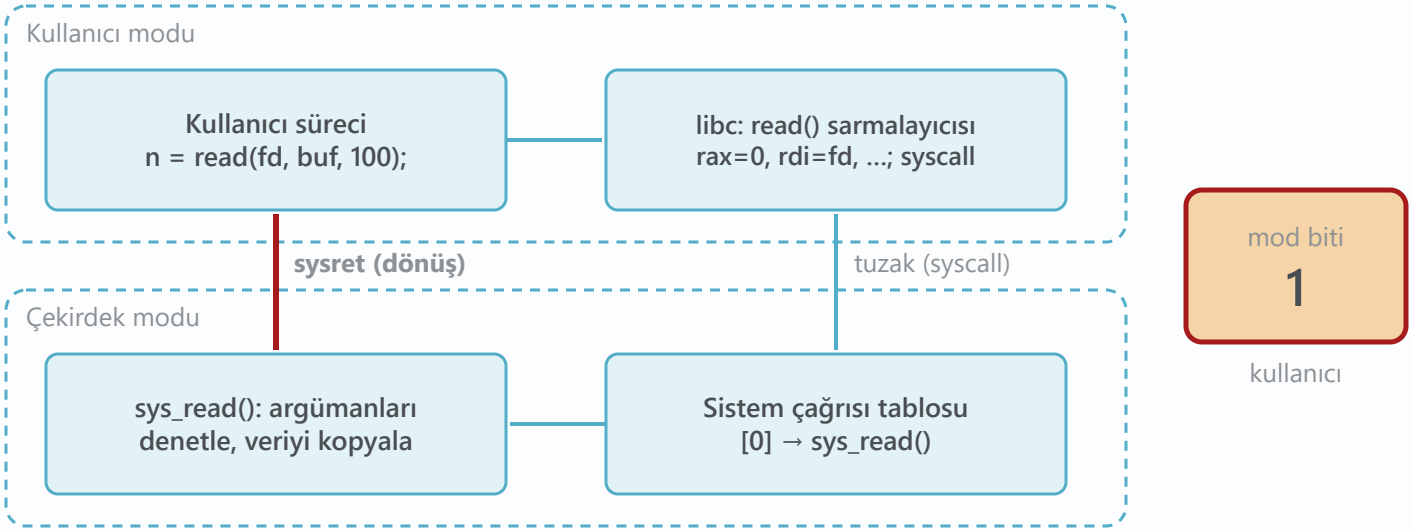
Tuzak: donanım mod bitini **0** yapar, dönüş adresini kaydeder ve çekirdeğin **sabit** giriş noktasına atlar. Süreç nereye atlanacağını seçemez.

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



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

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



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

Fiziksel Aygıtlar, Mikro Mimari, Makine Dili



Fiziksel aygıtlar

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

Mikro mimari

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

Makine dili (ISA)

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

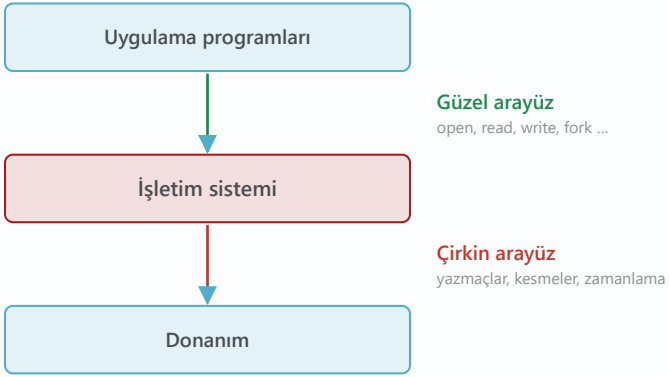
İS'nin işi

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

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

Genişletilmiş Makine Olarak İS

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



İS'NİN ROLLERİ

Soyutlamalar ve Roller

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

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

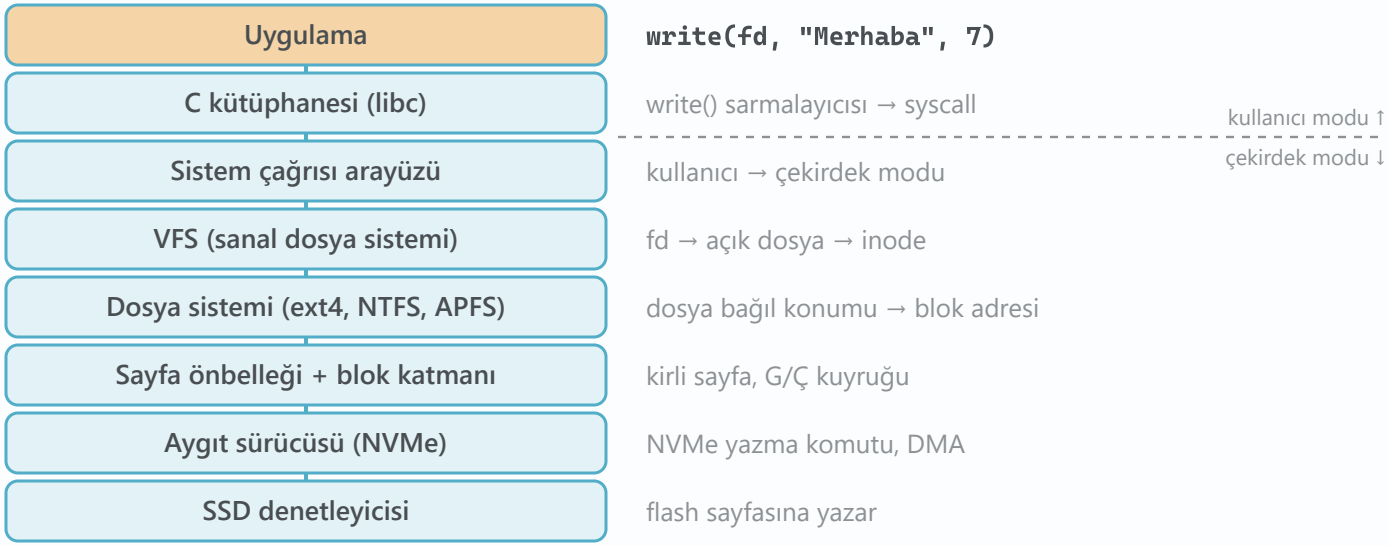
Donanım tasarımcısı
ISA'yı, aygıtları ve arayüzlerini tasarlar.

Çekirdek geliştirici
Soyutlamaları donanım üzerinde gerçekleştirir.

Uygulama geliştirici
Soyutlamaları sistem çağrıları ve API'lerle kullanır.

Son kullanıcı
Yalnızca uygulamaları ve arayüzü görür.

Bir `write()` Çağrısının Yolculuğu



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

Bir `write()` Çağrısının Yolculuğu



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

Bir `write()` Çağrısının Yolculuğu



VFS: fd, sürecin açık dosya tablosundan bir dosya nesnesine ve **i-node**'una çözülür. Tüm dosya sistemleri için ortak katman.

Bir `write()` Çağrısının Yolculuğu



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

Bir `write()` Çağrısının Yolculuğu



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

Bir `write()` Çağrısının Yolculuğu

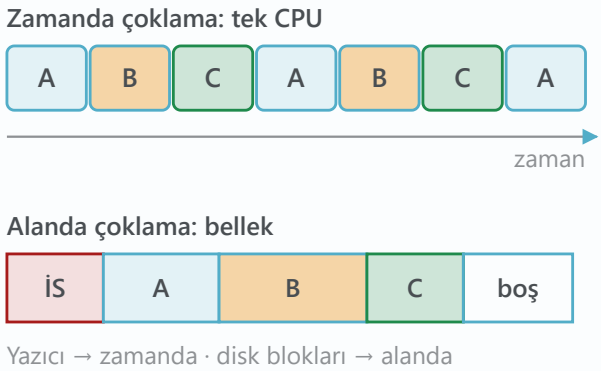


Sürücü bir **NVMe komutu** hazırlar; SSD veriyi DMA ile bellekten alıp flash'a yazar ve bitince kesme üretir.

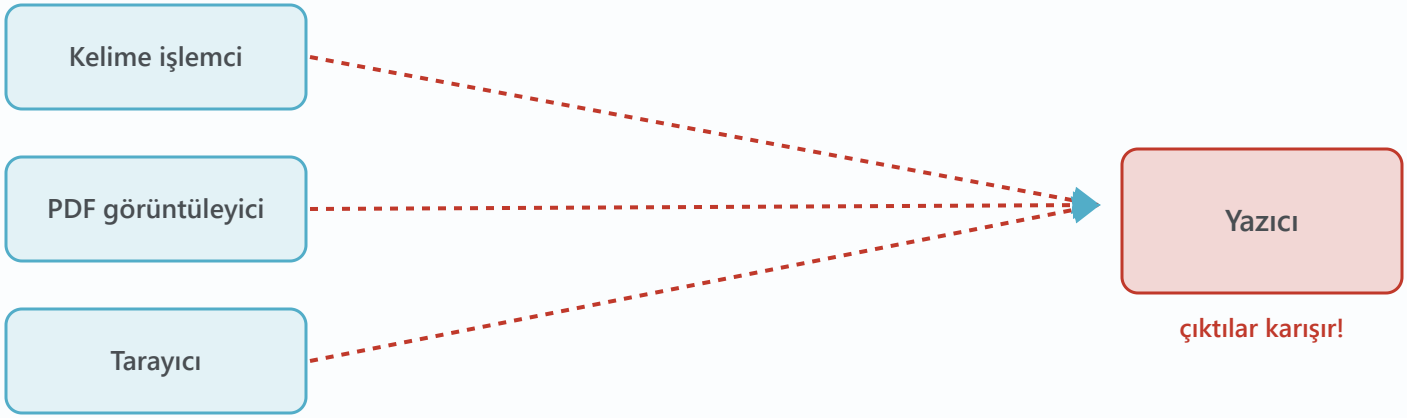
Kaynak Yöneticisi Olarak İS

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

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

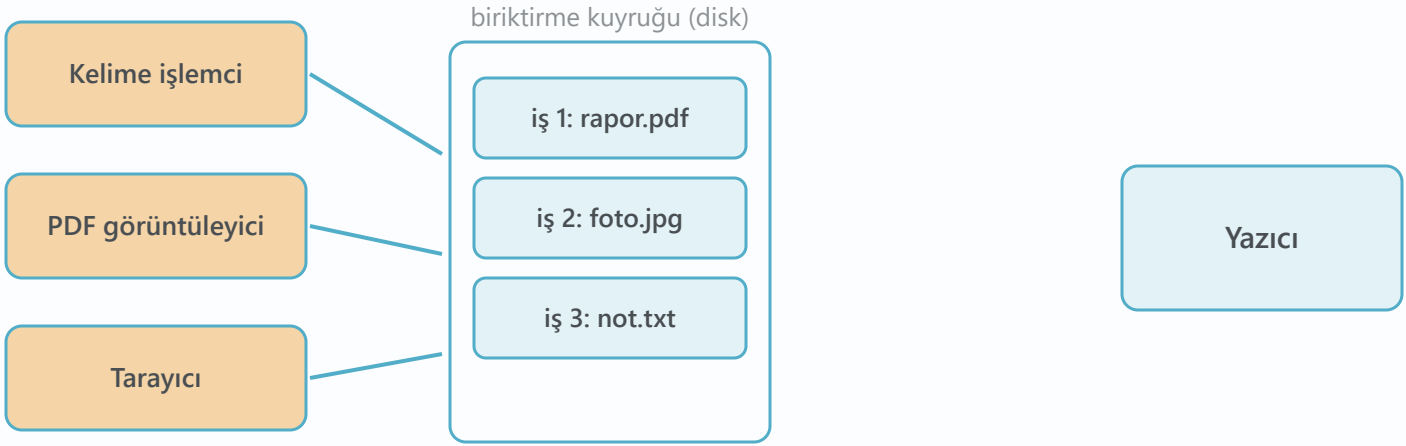


Birçok Program Aynı Anda Yazdırmak İsterse



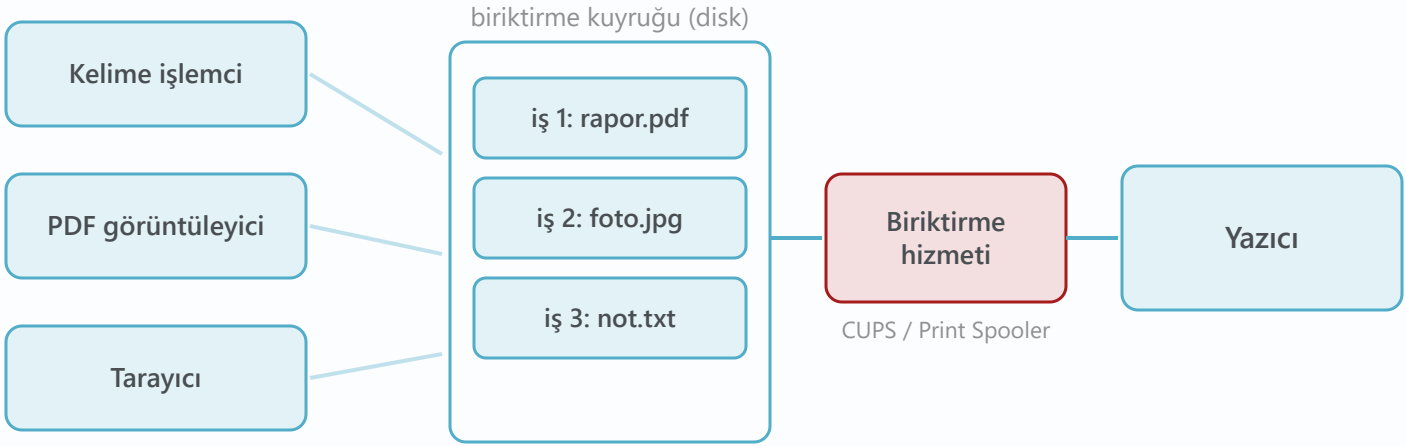
Üç program neredeyse aynı anda yazdırmak istiyor; yazıcı tek. Hepsi yazıcıya **doğrudan** yazabilseydi sayfalar birbirine karışırdı.

Birçok Program Aynı Anda Yazdırmak İsterse



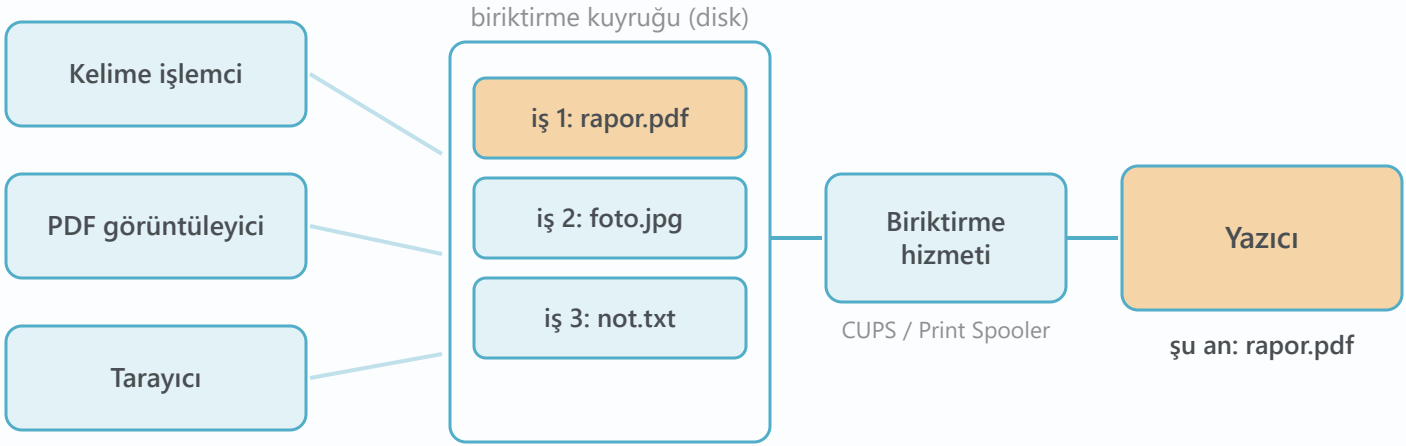
Programlar işlerini yazıcıya değil, İS'nin diskteki **biriktirme (spool) kuyruğuna** yazar ve hemen kendi işlerine döner.

Birçok Program Aynı Anda Yazdırmak İsterse



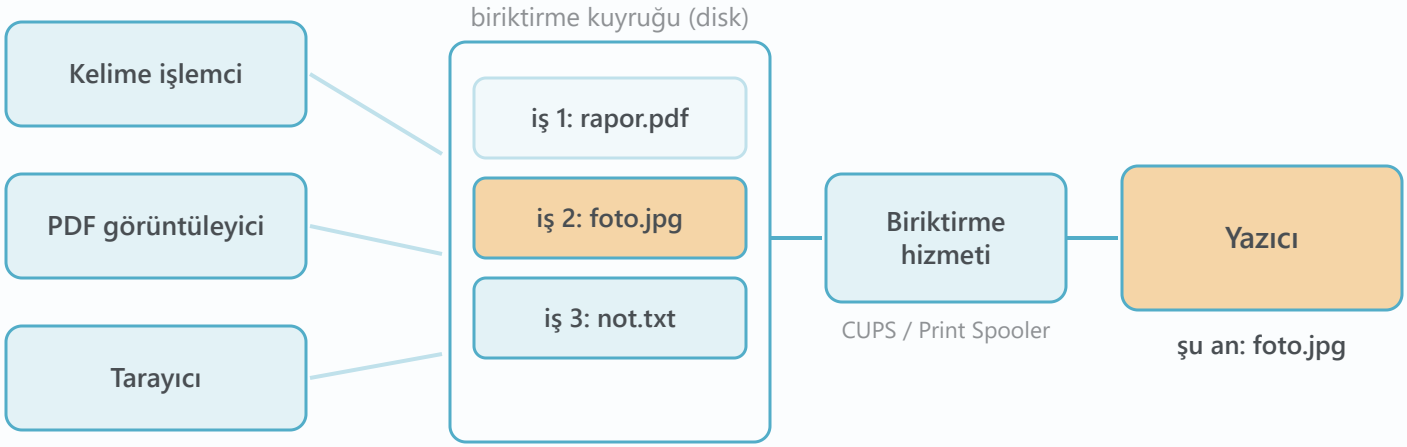
Yazıcıyı kullanan **tek süreç** biriktirme hizmetidir (Linux: CUPS, Windows: Print Spooler); karışıklık bu tekellikle önlenir.

Birçok Program Aynı Anda Yazdırmak İsterse



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

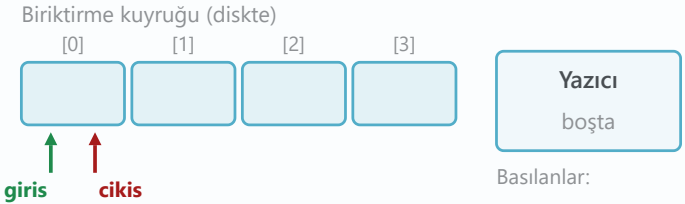
Birçok Program Aynı Anda Yazdırmak İsterse



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

Kod İzleme: Biriktirme Kuyruğu

```
▶ char *kuyruk[4];
▶ int giris = 0, cikis = 0, adet = 0;
3  void ekle(char *dosya) {   /* programlar */
4    kuyruk[giris] = dosya;
5    giris = (giris + 1) % 4;
6    adet++;
7  }
8  void biriktirme_hizmeti(void) {
9    while (adet > 0) {
10     yazici_gonder(kuyruk[cikis]);
11     kuyruk[cikis] = NULL;   /* yuva boşalır */
12     cikis = (cikis + 1) % 4;
13     adet--;
14   }
15 }
```



`adet = 0`

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

Adım 1 / 11

DEĞİŞKENLER

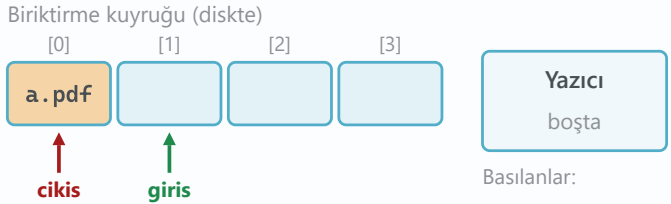
`giris = 0` `cikis = 0` `adet = 0`

ÇAĞRI YIĞINI

boş

Kod İzleme: Biriktirme Kuyruğu

```
1  char *kuyruk[4];
2  int giris = 0, cikis = 0, adet = 0;
3  void ekle(char *dosya) {   /* programlar */
▶    kuyruk[giris] = dosya;
▶    giris = (giris + 1) % 4;
▶    adet++;
7  }
8  void biriktirme_hizmeti(void) {
9    while (adet > 0) {
10     yazici_gonder(kuyruk[cikis]);
11     kuyruk[cikis] = NULL;   /* yuva boşalır */
12     cikis = (cikis + 1) % 4;
13     adet--;
14   }
15 }
```



`adet = 1`

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

Adım 2 / 11

DEĞİŞKENLER

`giris = 1` `cikis = 0` `adet = 1`

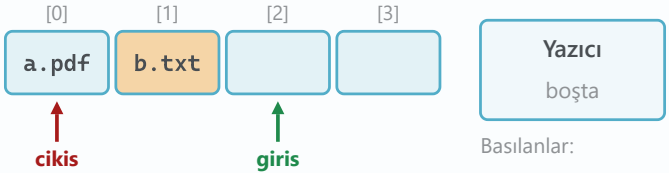
ÇAĞRI YIĞINI

`ekle("a.pdf")`

Kod İzleme: Biriktirme Kuyruğu

```
1  char *kuyruk[4];
2  int giris = 0, cikis = 0, adet = 0;
3  void ekle(char *dosya) {   /* programlar */
▶    kuyruk[giris] = dosya;
▶    giris = (giris + 1) % 4;
▶    adet++;
7  }
8  void biriktirme_hizmeti(void) {
9    while (adet > 0) {
10     yazici_gonder(kuyruk[cikis]);
11     kuyruk[cikis] = NULL;   /* yuva boşalır */
12     cikis = (cikis + 1) % 4;
13     adet--;
14   }
15 }
```

Biriktirme kuyruğu (diskte)



`adet = 2`

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

Adım 3 / 11

DEĞİŞKENLER

`giris = 2` `cikis = 0` `adet = 2`

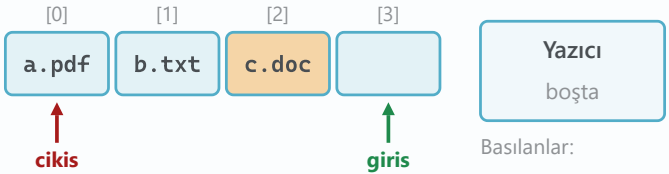
ÇAĞRI YIĞINI

`ekle("b.txt")`

Kod İzleme: Biriktirme Kuyruğu

```
1  char *kuyruk[4];
2  int giris = 0, cikis = 0, adet = 0;
3  void ekle(char *dosya) {   /* programlar */
▶    kuyruk[giris] = dosya;
▶    giris = (giris + 1) % 4;
▶    adet++;
7  }
8  void biriktirme_hizmeti(void) {
9    while (adet > 0) {
10     yazici_gonder(kuyruk[cikis]);
11     kuyruk[cikis] = NULL;   /* yuva boşalır */
12     cikis = (cikis + 1) % 4;
13     adet--;
14   }
15 }
```

Biriktirme kuyruğu (diskte)



`adet = 3`

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

Adım 4 / 11

DEĞİŞKENLER

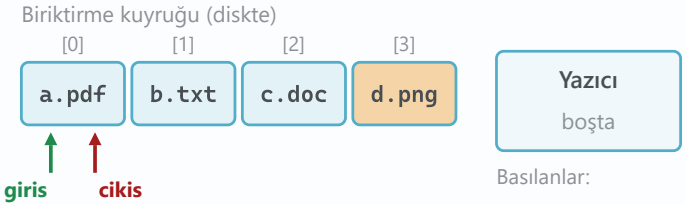
`giris = 3` `cikis = 0` `adet = 3`

ÇAĞRI YIĞINI

`ekle("c.doc")`

Kod İzleme: Biriktirme Kuyruğu

```
1  char *kuyruk[4];
2  int giris = 0, cikis = 0, adet = 0;
3  void ekle(char *dosya) {   /* programlar */
▶    kuyruk[giris] = dosya;
▶    giris = (giris + 1) % 4;
▶    adet++;
7  }
8  void biriktirme_hizmeti(void) {
9    while (adet > 0) {
10     yazici_gonder(kuyruk[cikis]);
11     kuyruk[cikis] = NULL;   /* yuva boşalır */
12     cikis = (cikis + 1) % 4;
13     adet--;
14   }
15 }
```



`adet = 4`

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

Adım 5 / 11

DEĞİŞKENLER

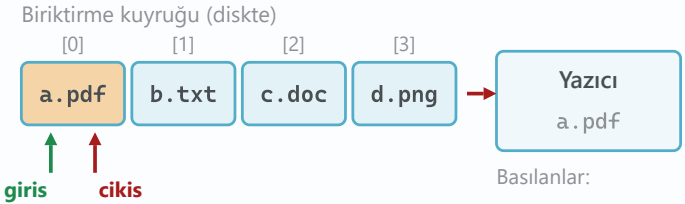
`giris = 0` `cikis = 0` `adet = 4`

ÇAĞRI YIĞINI

`ekle("d.png")`

Kod İzleme: Biriktirme Kuyruğu

```
1  char *kuyruk[4];
2  int giris = 0, cikis = 0, adet = 0;
3  void ekle(char *dosya) {   /* programlar */
4    kuyruk[giris] = dosya;
5    giris = (giris + 1) % 4;
6    adet++;
7  }
8  void biriktirme_hizmeti(void) {
▶    while (adet > 0) {
▶      yazici_gonder(kuyruk[cikis]);
11     kuyruk[cikis] = NULL;   /* yuva boşalır */
12     cikis = (cikis + 1) % 4;
13     adet--;
14   }
15 }
```



`adet = 4`

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

Adım 6 / 11

DEĞİŞKENLER

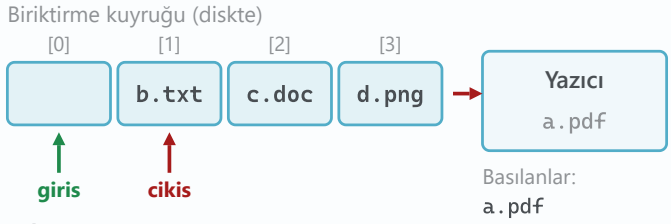
`giris = 0` `cikis = 0` `adet = 4`

ÇAĞRI YIĞINI

`biriktirme_hizmeti()`

Kod İzleme: Biriktirme Kuyruğu

```
1  char *kuyruk[4];
2  int giris = 0, cikis = 0, adet = 0;
3  void ekle(char *dosya) {   /* programlar */
4    kuyruk[giris] = dosya;
5    giris = (giris + 1) % 4;
6    adet++;
7  }
8  void biriktirme_hizmeti(void) {
9    while (adet > 0) {
10     yazici_gonder(kuyruk[cikis]);
▶      kuyruk[cikis] = NULL;   /* yuva boşalır */
▶      cikis = (cikis + 1) % 4;
▶      adet--;
14   }
15 }
```



adet = 3

Yuva boşaltılır (kuyruk[0] = NULL): cikis = 1, adet = 3. Yazıcı yavaştır; programlar bu sırada başka iş yapar.

Adım 7 / 11

DEĞİŞKENLER

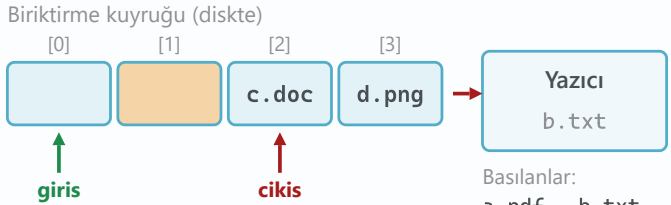
`giris = 0` `cikis = 1` `adet = 3`

ÇAĞRI YIĞINI

`biriktirme_hizmeti()`

Kod İzleme: Biriktirme Kuyruğu

```
1  char *kuyruk[4];
2  int giris = 0, cikis = 0, adet = 0;
3  void ekle(char *dosya) {   /* programlar */
4    kuyruk[giris] = dosya;
5    giris = (giris + 1) % 4;
6    adet++;
7  }
8  void biriktirme_hizmeti(void) {
▶    while (adet > 0) {
▶      yazici_gonder(kuyruk[cikis]);
▶      kuyruk[cikis] = NULL;   /* yuva boşalır */
▶      cikis = (cikis + 1) % 4;
▶      adet--;
14   }
15 }
```



`adet = 2`

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

Adım 8 / 11

DEĞİŞKENLER

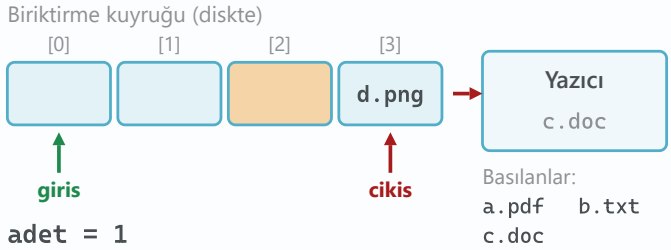
`giris = 0` `cikis = 2` `adet = 2`

ÇAĞRI YIĞINI

`biriktirme_hizmeti()`

Kod İzleme: Biriktirme Kuyruğu

```
1  char *kuyruk[4];
2  int giris = 0, cikis = 0, adet = 0;
3  void ekle(char *dosya) {   /* programlar */
4    kuyruk[giris] = dosya;
5    giris = (giris + 1) % 4;
6    adet++;
7  }
8  void biriktirme_hizmeti(void) {
▶    while (adet > 0) {
▶      yazici_gonder(kuyruk[cikis]);
▶      kuyruk[cikis] = NULL;   /* yuva boşalır */
▶      cikis = (cikis + 1) % 4;
▶      adet--;
14   }
15 }
```



adet = 1

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

Adım 9 / 11

DEĞİŞKENLER

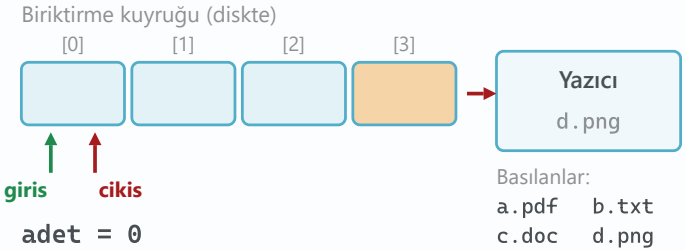
`giris = 0` `cikis = 3` `adet = 1`

ÇAĞRI YIĞINI

`biriktirme_hizmeti()`

Kod İzleme: Biriktirme Kuyruğu

```
1  char *kuyruk[4];
2  int giris = 0, cikis = 0, adet = 0;
3  void ekle(char *dosya) {   /* programlar */
4    kuyruk[giris] = dosya;
5    giris = (giris + 1) % 4;
6    adet++;
7  }
8  void biriktirme_hizmeti(void) {
▶    while (adet > 0) {
▶      yazici_gonder(kuyruk[cikis]);
▶      kuyruk[cikis] = NULL;   /* yuva boşalır */
▶      cikis = (cikis + 1) % 4;
▶      adet--;
14   }
15 }
```



`adet = 0`

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

Adım 10 / 11

DEĞİŞKENLER

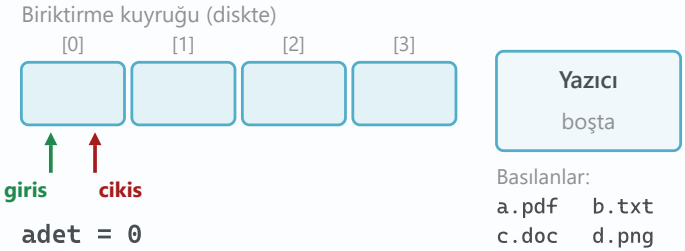
`giris = 0` `cikis = 0` `adet = 0`

ÇAĞRI YIĞINI

`biriktirme_hizmeti()`

Kod İzleme: Biriktirme Kuyruğu

```
1  char *kuyruk[4];
2  int giris = 0, cikis = 0, adet = 0;
3  void ekle(char *dosya) {   /* programlar */
4    kuyruk[giris] = dosya;
5    giris = (giris + 1) % 4;
6    adet++;
7  }
8  void biriktirme_hizmeti(void) {
▶    while (adet > 0) {
10     yazici_gonder(kuyruk[cikis]);
11     kuyruk[cikis] = NULL;   /* yuva boşalır */
12     cikis = (cikis + 1) % 4;
13     adet--;
14   }
15 }
```



`adet = 0`

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

Adım 11 / 11

DEĞİŞKENLER

`giris = 0` `cikis = 0` `adet = 0`

ÇAĞRI YIĞINI

`biriktirme_hizmeti()`

Kaynak Kullanımını Ölçmek

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

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

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

İS'NİN ROLLERİ

Adalet ve Verimlilik

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

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

Önceliklendirme
`nice` değerleri, Windows öncelik sınıfları, gerçek zamanlı sınıflar.

Kotalar ve sınırlar
Linux **cgroups v2** (konteynerler), `ulimit`, disk kotaları, Windows Job nesneleri.

Ödünleşim
Verim ↔ tepki süresi ↔ adalet: biri iyileşirken diğeri kötüleşebilir.

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

Üç Büyük Fikir: Dört Küçük Deney

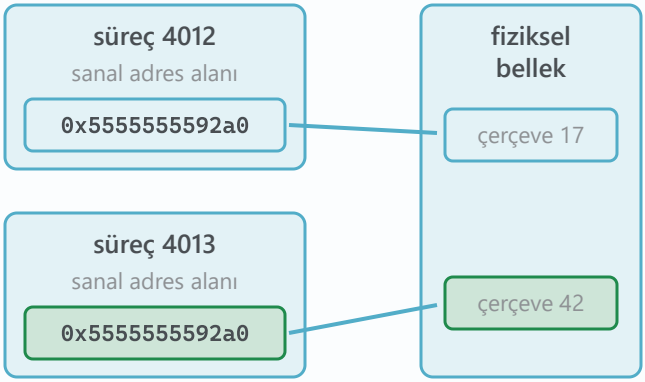
```
$ ./yaz X & ./yaz Y & ./yaz Z &
[1] 3101 [2] 3102 [3] 3103
X
Y
Z
Z
X
Y
…
```



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

Üç Büyük Fikir: Dört Küçük Deney

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



aynı sanal adres, farklı fiziksel çerçeve

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

Üç Büyük Fikir: Dört Küçük Deney

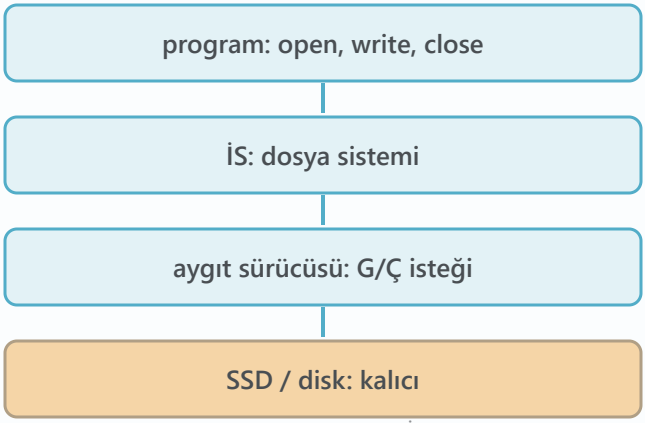
```
$ ./artir 1000
sonuç: 2000
$ ./artir 1000000
sonuç: 1318442
$ ./artir 1000000
sonuç: 1407715
(beklenen: 2000000)
```



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

Üç Büyük Fikir: Dört Küçük Deney

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

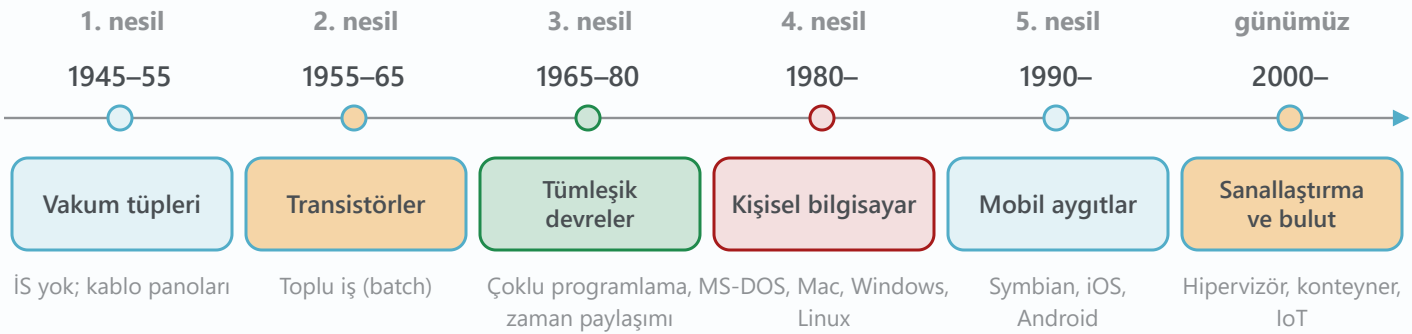


blok seçimi, meta veri, sıralama: hepsi İS'de

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

TARİHÇE

İşletim Sistemlerinin Tarihi



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

TARİHÇE

1. ve 2. Nesil

1. nesil (1945–55): vakum tüpleri

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

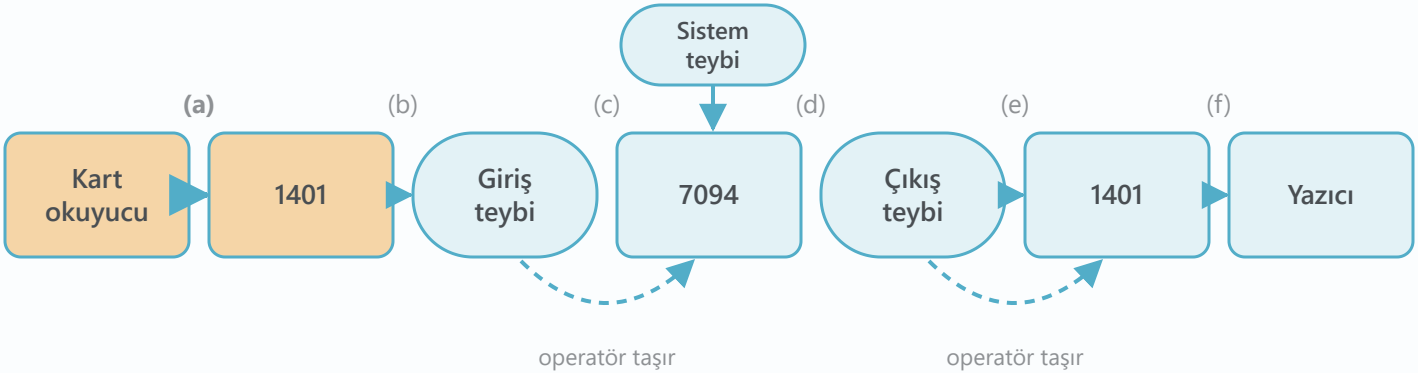
2. nesil (1955–65): transistör ve toplu iş

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

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

TARİHÇE · ANİMASYON

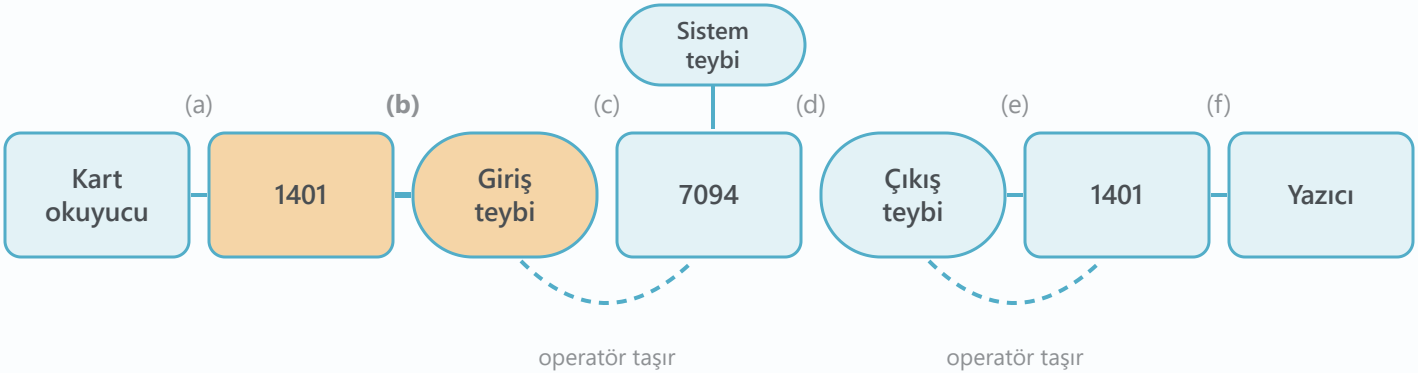
Erken Bir Toplu İş Sistemi



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

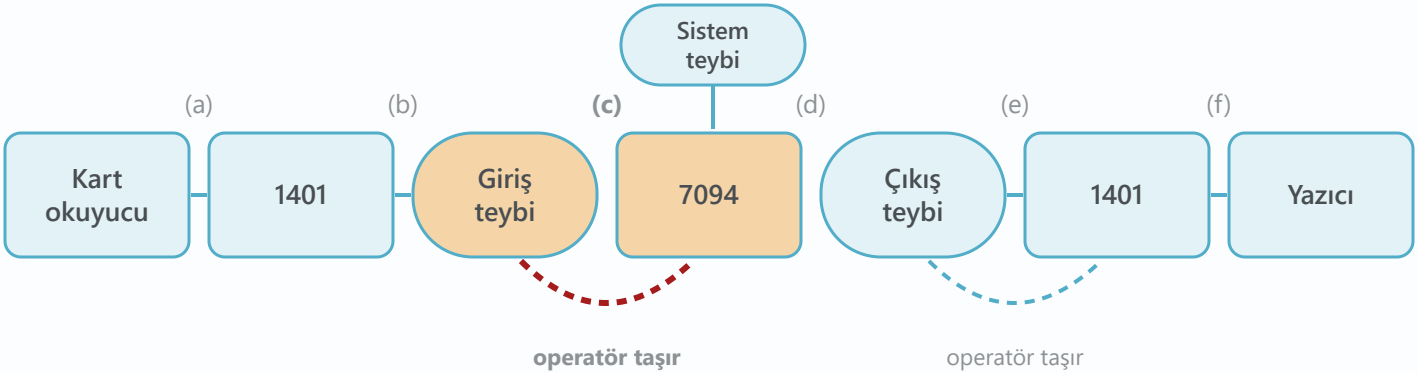
TARİHÇE · ANİMASYON

Erken Bir Toplu İş Sistemi



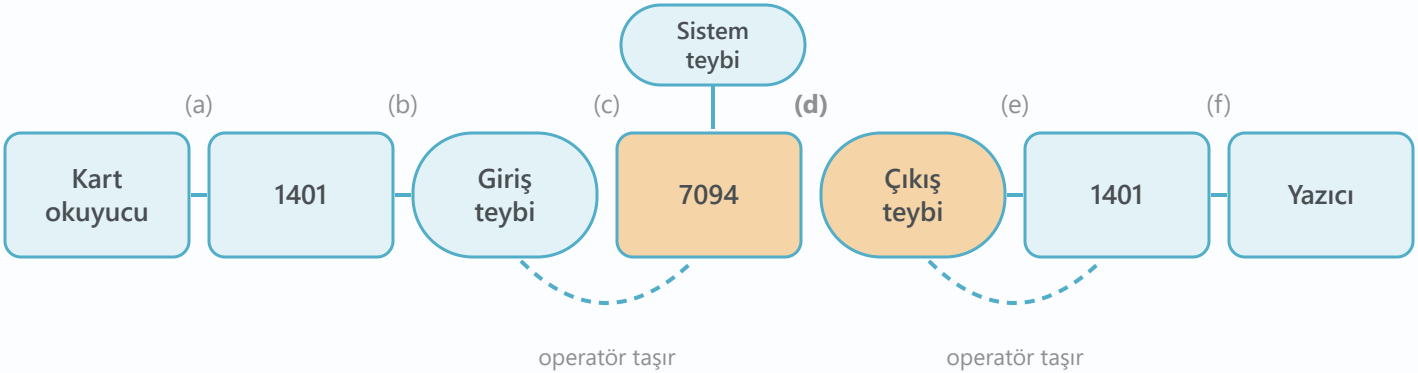
(b) Ucuz 1401, bir grup işin kartlarını okuyup **giriş teybine** yazar (yavaş kart okumayı pahalı makineden ayırır).

Erken Bir Toplu İş Sistemi



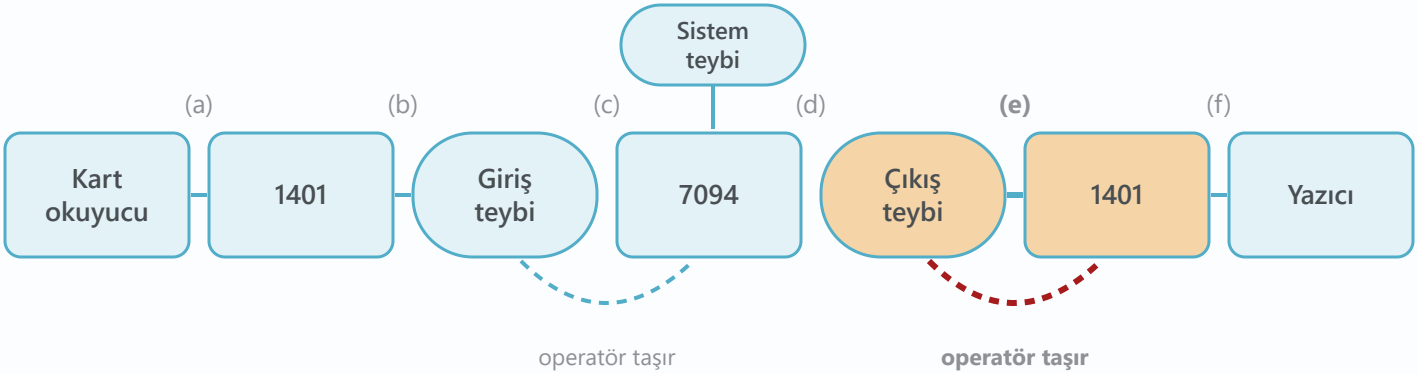
(c) Operatör giriş teybini pahalı **7094**'e taşır.

Erken Bir Toplu İş Sistemi



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

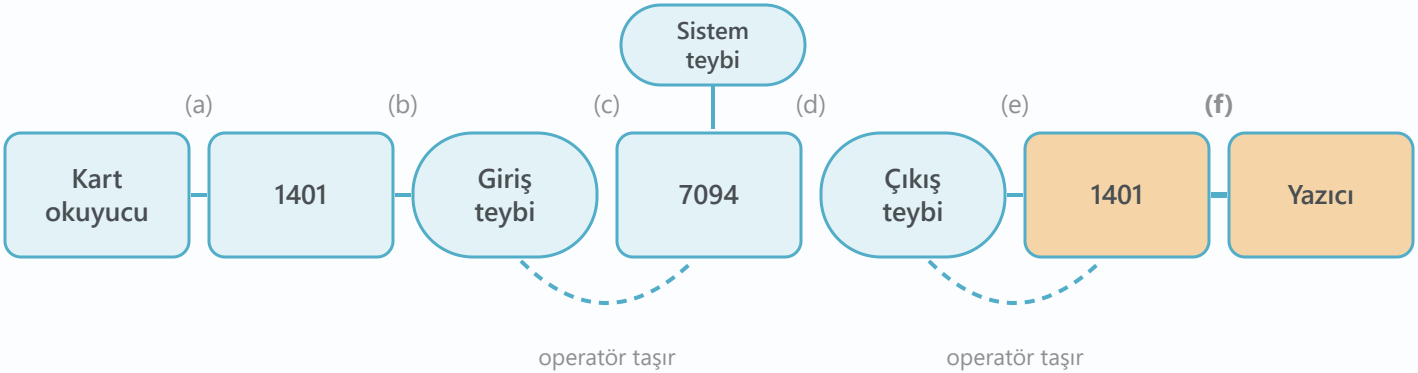
Erken Bir Toplu İş Sistemi



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

TARİHÇE · ANİMASYON

Erken Bir Toplu İş Sistemi



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

TARİHÇE

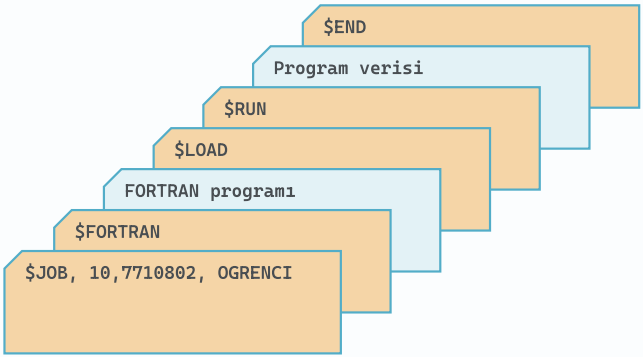
IBM 1401, 7094 ve System/360

Makine	Yıl	Özellik
IBM 1401	1959	Ticari veri işleme; karakter tabanlı, görece ucuz. Kart okuma, teybe kopyalama, yazdırma gibi G/Ç işlerinde de kullanıldı. 1960'ların en yaygın bilgisayarlarından.
IBM 7094	1962	Bilimsel hesaplama; 36 bit sözcüklü, **1401'den tamamen farklı** bir mimari. IBSYS / FMS altında toplu iş çalıştırdı. MIT'de değiştirilmiş bir 7094 üzerinde **CTSS** ile zaman paylaşımı denendi.
System/360	1964	Küçükten büyüğe **aynı mimariyi** paylaşan bir aile. İşletim sistemi **OS/360**: çoklu programlama, biriktirme; milyonlarca satırlık, hatalarla dolu dev bir proje (Brooks, *The Mythical Man-Month*).

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

TARİHÇE

Tipik Bir FMS İşi

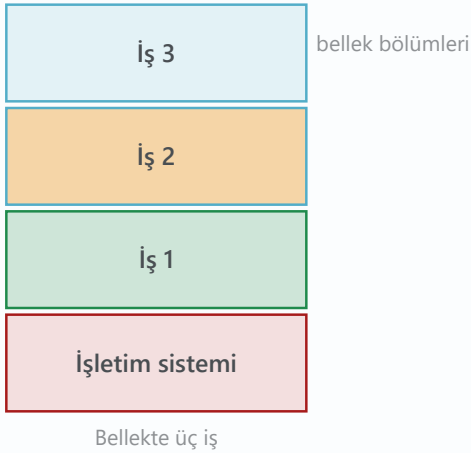


- **FMS = Fortran Monitor System**: IBM 709/7094 için bir toplu iş izleme (monitör) programı.
- `$` ile başlayan **denetim kartları** monitöre ne yapacağını söyler: işi başlat, derleyiciyi yükle, programı bağla, çalıştır, bitir.
- Bu kartlar bugünkü **kabuk betiklerinin** ve IBM JCL'nin atasıdır.

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

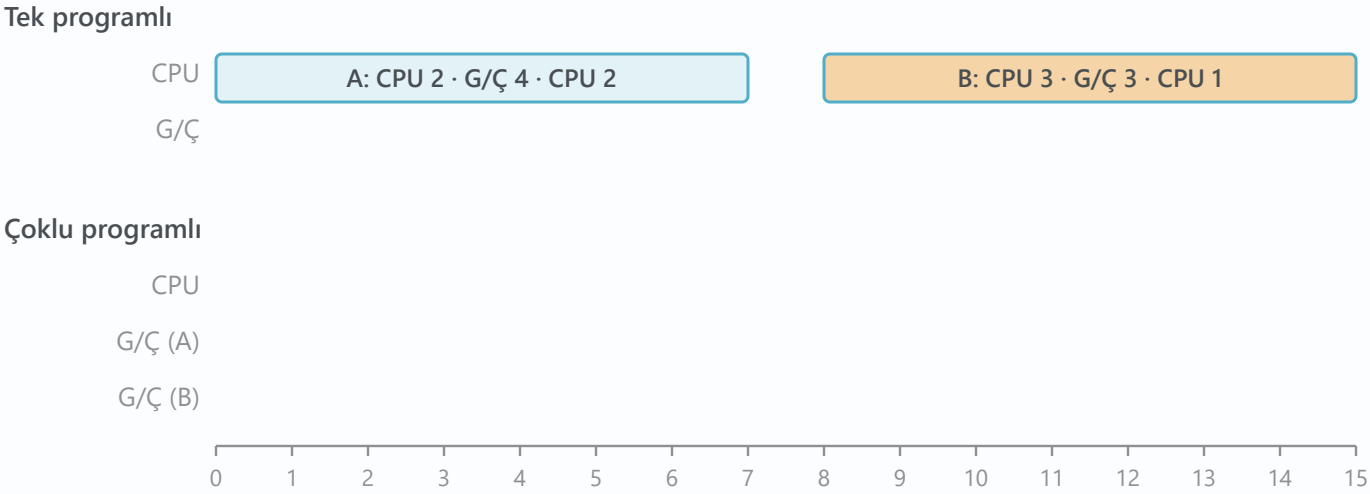
3. Nesil: Tümleşik Devreler ve Çoklu Programlama

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



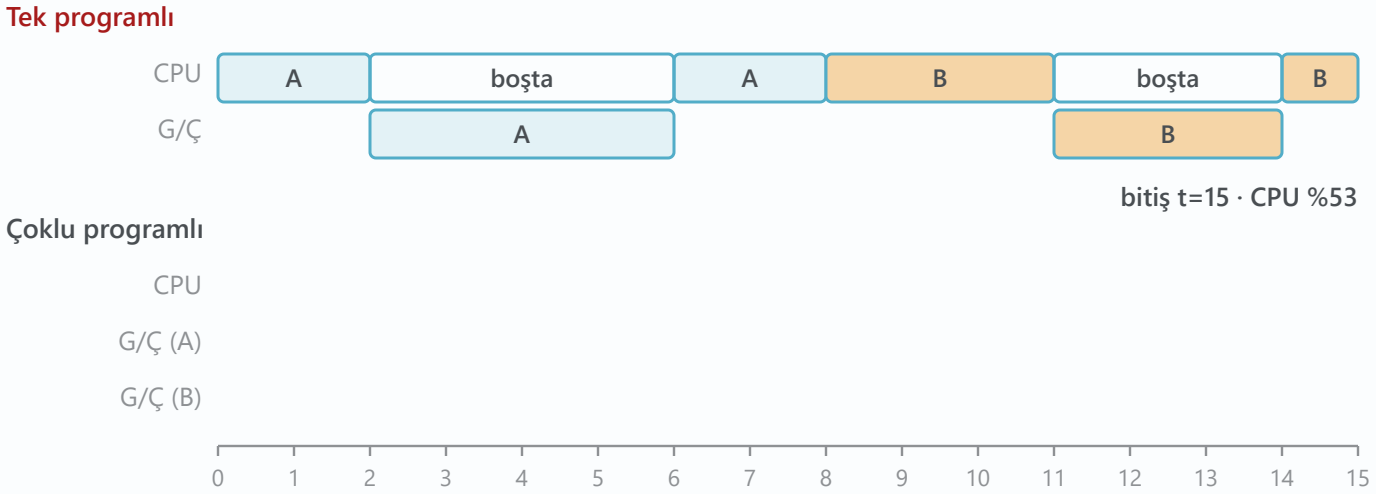
TARİHÇE · ANİMASYON

Çoklu Programlama: CPU Neden Boş Kalmaz?



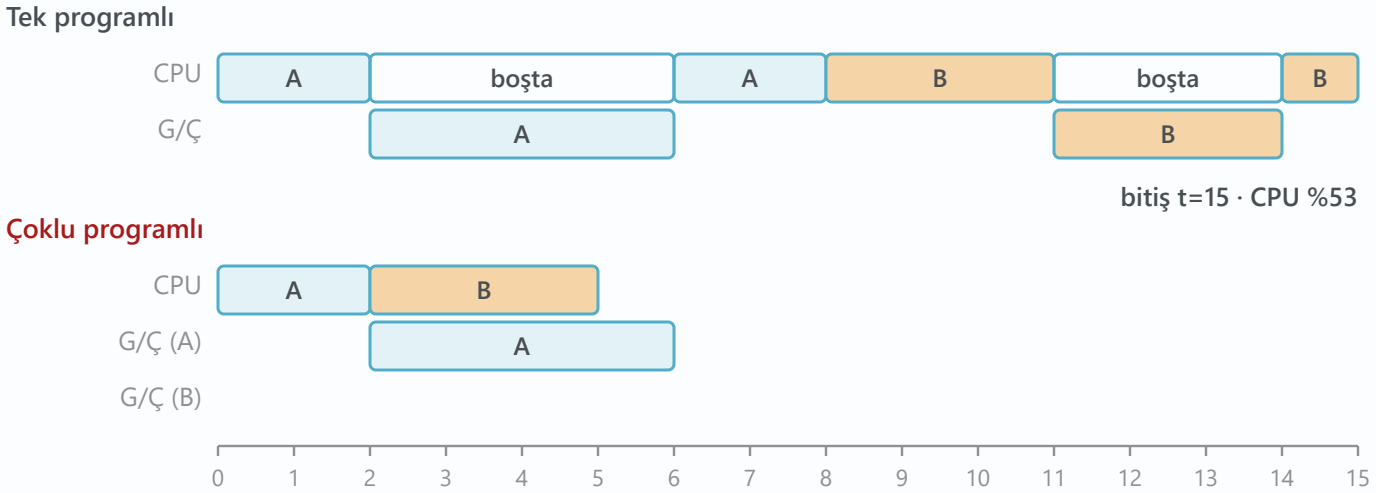
İki iş (birim zaman): **A** = CPU 2, G/Ç 4, CPU 2 · **B** = CPU 3, G/Ç 3, CPU 1. Toplam CPU ihtiyacı 8 birim.

Çoklu Programlama: CPU Neden Boş Kalmaz?



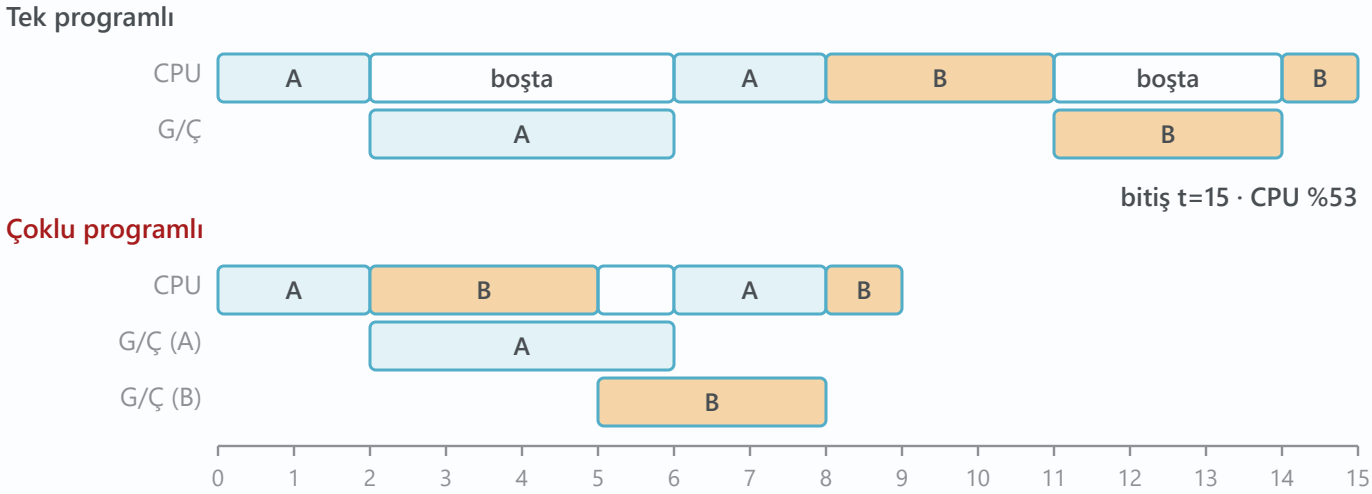
Tek programlı: önce A (t = 0–8), sonra B (t = 8–15). G/Ç sırasında CPU **boşta**: kullanım 8 / 15 ≈ **%53**.

Çoklu Programlama: CPU Neden Boş Kalmaz?



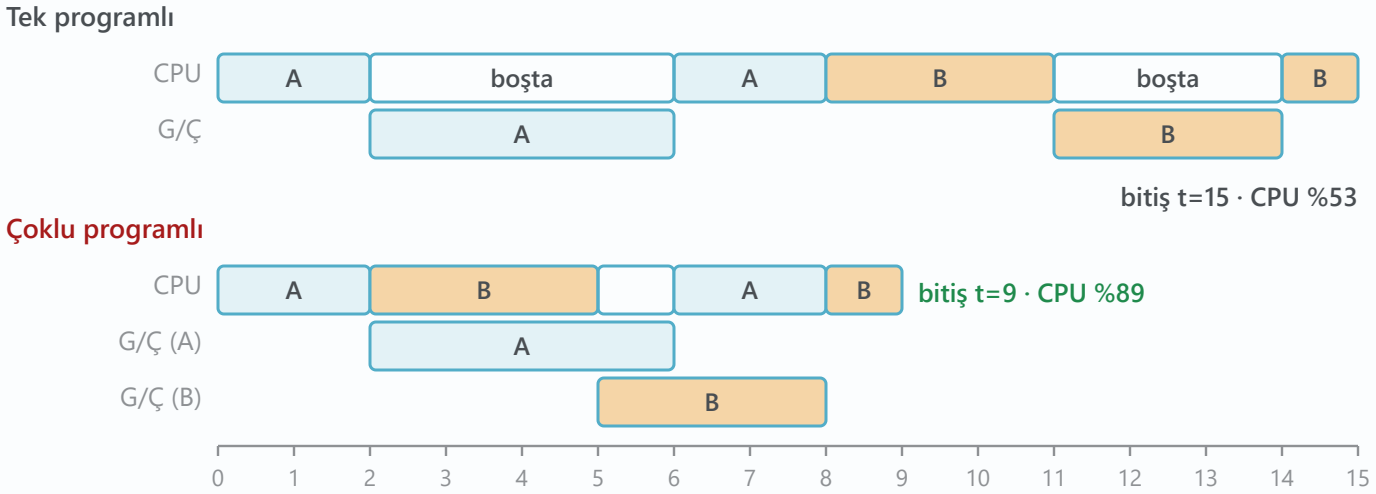
Çoklu programlı: A t = 2'de G/Ç'ye gidince CPU hemen **B**'ye geçer (t = 2–5). A'nın G/Ç'si arka planda sürer.

Çoklu Programlama: CPU Neden Boş Kalmaz?



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

Çoklu Programlama: CPU Neden Boş Kalmaz?



Aynı iki iş **15 yerine 9** birimde bitti; CPU kullanımı 8 / 9 ≈ **%89**. Çoklu programlamanın kazancı, CPU ile G/Ç'nin örtüşmesidir.

TARİHÇE

4. Nesil: Kişisel Bilgisayarlar

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

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

TARİHÇE

5. Nesil: Mobil ve Günümüz

Mobil aygıtlar (1990–)

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

Sanallaştırma ve bulut

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

Konteynerler

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

Heterojen donanım

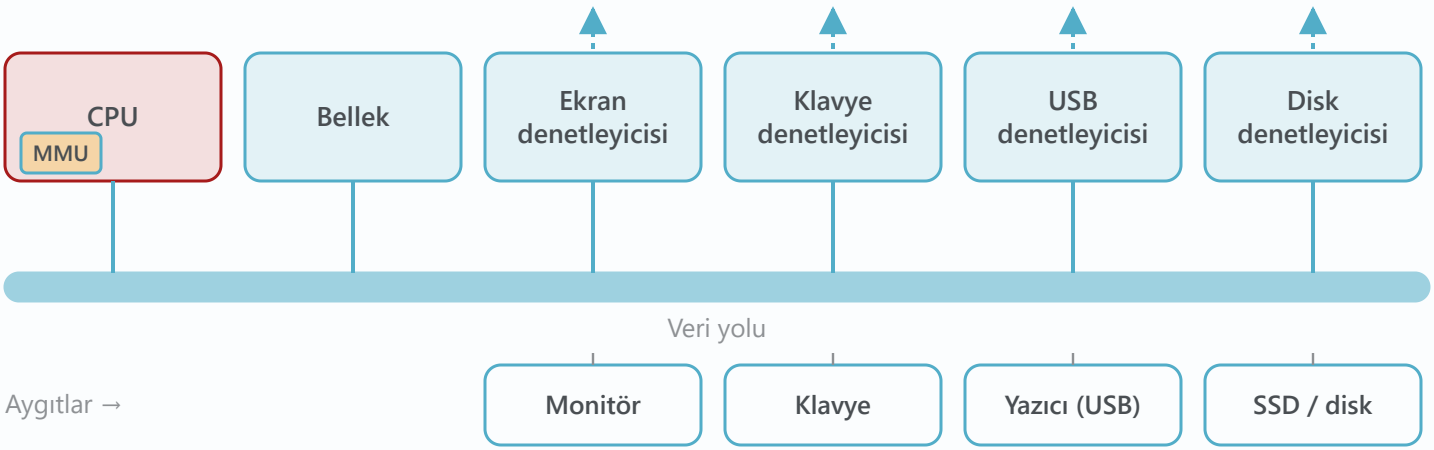
CPU + GPU + NPU; büyük ve küçük çekirdekli hibrit işlemciler.

Güvenlik

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

DONANIM

Kişisel Bilgisayarın Bileşenleri



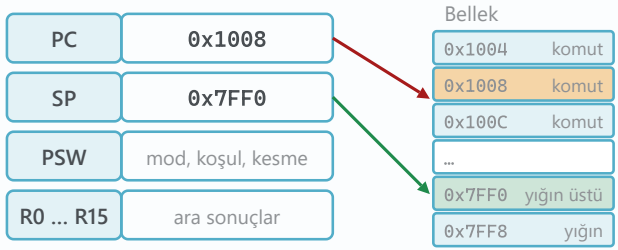
- CPU, bellek ve G/Ç aygıtları bir **veri yolu** üzerinden haberleşir; her aygıtın bir **denetleyicisi** vardır.
- **MMU** (bellek yönetim birimi) CPU içindedir ve sanal adresleri fiziksel adrese çevirir.
- Gerçek sistemlerde tek ortak hat yerine noktadan noktaya bağlantılar (PCIe) kullanılır; ama kavramsal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

DONANIM

İşlemci ve Yazmaçlar

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

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



Çok Çekirdekli Yongalar



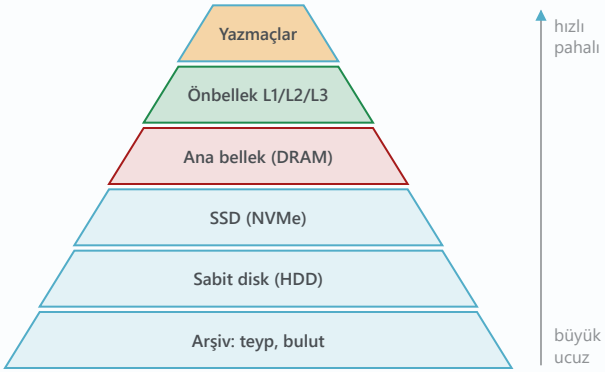
Çok çekirdek
Bir yongada birden çok bağımsız CPU; İS her birini ayrı işlemci olarak görür ve çizelgeler.

SMT / Hyper-Threading
Bir çekirdek iki donanım iş parçacığı çalıştırır; İS'ye iki "mantıksal CPU" görünür ama kaynakları paylaşırlar.

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

İS açısından sonuç: paralellik, senkronizasyon ve önbellek tutarlılığı sorunları.

Bellek Hiyerarşisi



Düzey	Erişim (yakl.)	Kapasite (tipik)
Yazmaçlar	< 1 ns	< 1 KB
L1 önbellek	~1 ns	32–64 KB
L2 / L3	3–20 ns	1–100+ MB
DRAM	60–100 ns	8–64 GB
NVMe SSD	20–100 µs	0,5–4 TB
HDD	5–10 ms	2–24 TB

DRAM ve üstü **uçucudur** (güç gidince silinir); SSD ve HDD kalıcıdır. Değerler 2026 itibarıyla yaklaşık büyüklüklerdir. Her düzey altındakinin önbelleği gibi çalışır: **yerellik** sayesinde erişimlerin çoğu üstteki hızlı düzeylerde karşılanır.

G/Ç

Üç G/Ç Yöntemi

Her G/Ç aygıtının bir **denetleyicisi** vardır; çekirdekteki **aygıt sürücüsü** denetleyicinin yazmaçlarına komut yazar, durumunu okur. İşin bitip bitmediği üç yoldan öğrenilir:

Yöntem	Temel fark
Yoklama (polling)	CPU, aygıt hazır olana kadar durum yazmacını döngüde okur: beklerken başka iş yapamaz (meşgul bekleme).
Kesme (interrupt)	Aygıt her iş biriminin sonunda CPU'ya sinyal gönderir; CPU arada başka süreci çalıştırır.
DMA (doğrudan bellek erişimi)	Denetleyici bütün bloğu belleğe kendisi taşır; CPU yalnızca başta (programlama) ve sonda (tek kesme) devreye girer.

Aygıttan gelen **kesme** ile yürütülen komutun kendisinden doğan **tuzak** (sıfıra bölme, sayfa hatası, sistem çağrısı) aynı yolla çekirdeğe geçer: işlemci çekirdek moduna girer ve tablodaki işleyiciye atlar.

İşletim Sistemi Çeşitleri (1)

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

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

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

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

İşletim Sistemi Çeşitleri (2)

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

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

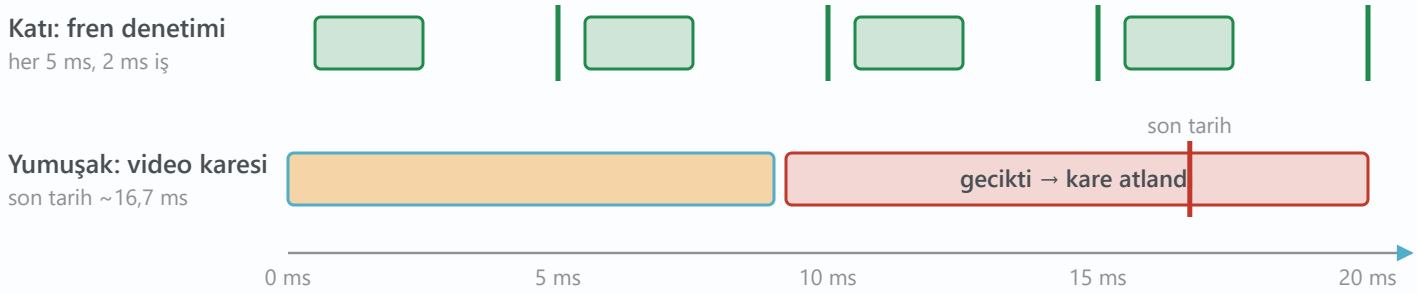
Algılayıcı düğümü (sensor node)
Birkaç on KB RAM, pil ile yıllarca çalışma, olay güdümlü yapı, düşük güçlü radyo (802.15.4, BLE). IoT ve M2M ağları. *Contiki-NG, RIOT, Zephyr; TinyOS artık tarihsel.*

Akıllı kart (smart card)
Çok kısıtlı CPU ve bellek; kimlik doğrulama, ödeme, kriptografi, güvenli saklama. Banka kartı, SIM/eSIM, kimlik kartı. *Java Card, MULTOS*.



İS ÇEŞİTLERİ

Gerçek Zamanlı İşletim Sistemleri



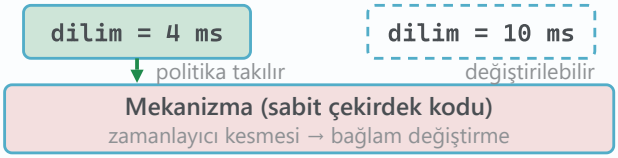
- Önemli olan **hız değil, öngörülebilirlik**: iş belirli bir **son tarihten** (deadline) önce bitmelidir.
- **Katı (hard)**: son tarihi kaçırmak felakettir (fren, hava yastığı, uçuş kontrolü, endüstriyel robot).
- **Yumuşak (soft)**: ara sıra kaçırmak kabul edilir, kalite düşer (ses/video akışı, oyun).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

Politika ve Mekanizma

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

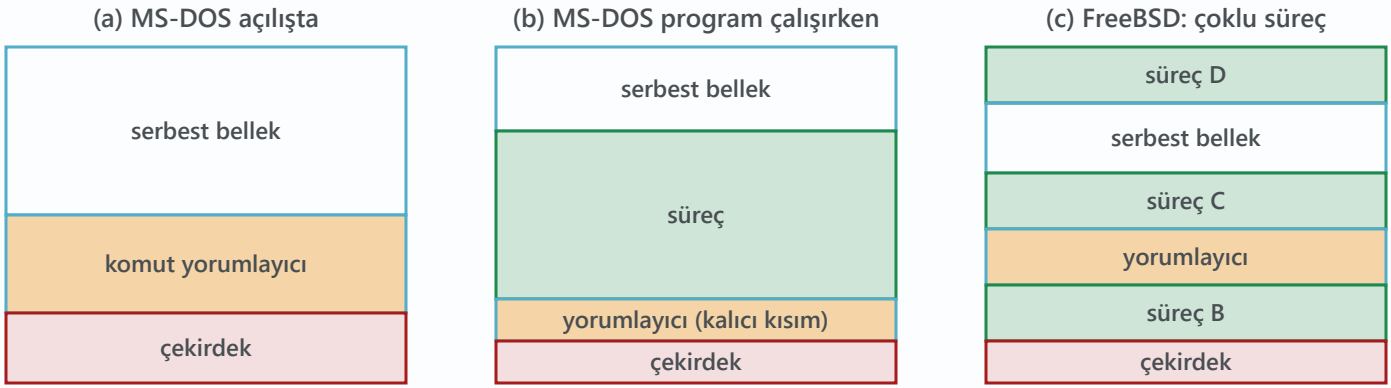
Örnek: Linux'ta çizelgeleme sınıfı `sched_setscheduler()` ile, G/Ç çizelgeleyicisi `/sys` üzerinden çalışırken değiştirilebilir; çekirdek kodu değişmez.

Mekanizma	Politika
Zamanlayıcı kesmesi	Zaman dilimi kaç ms olmalı?
Öncelikli çizelgeleyici	Hangi süreç yüksek öncelikli?
Dosya izin bitleri (rwx)	Kime hangi izin verilecek?
Sayfa yer değiştirme altyapısı	Hangi sayfa dışarı atılacak?
cgroups	Hangi konteynere ne kadar CPU/bellek?



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

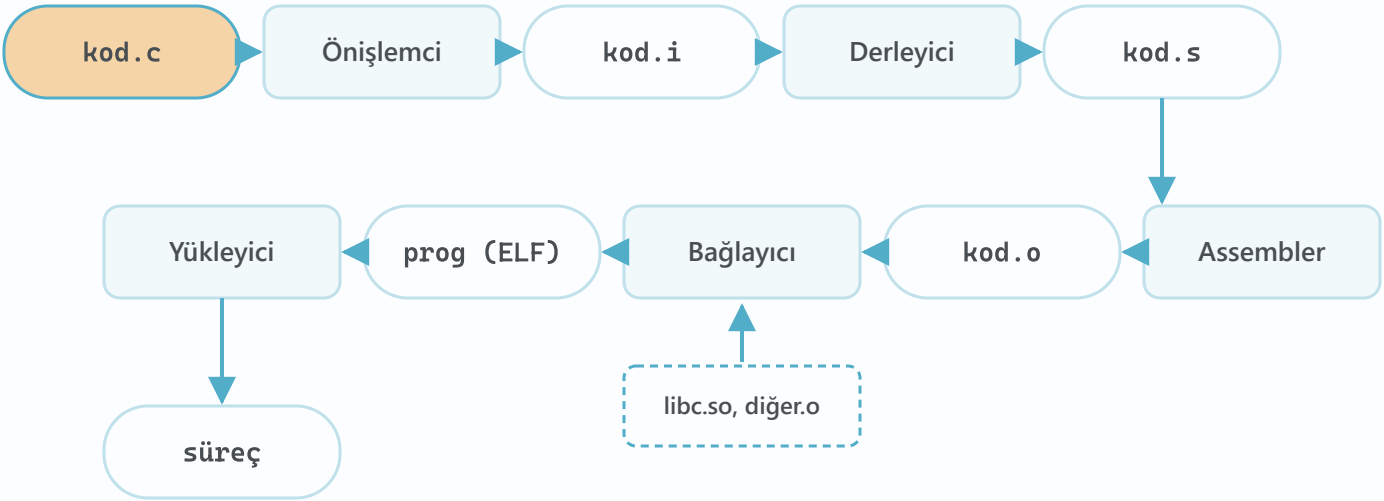
MS-DOS ve FreeBSD



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

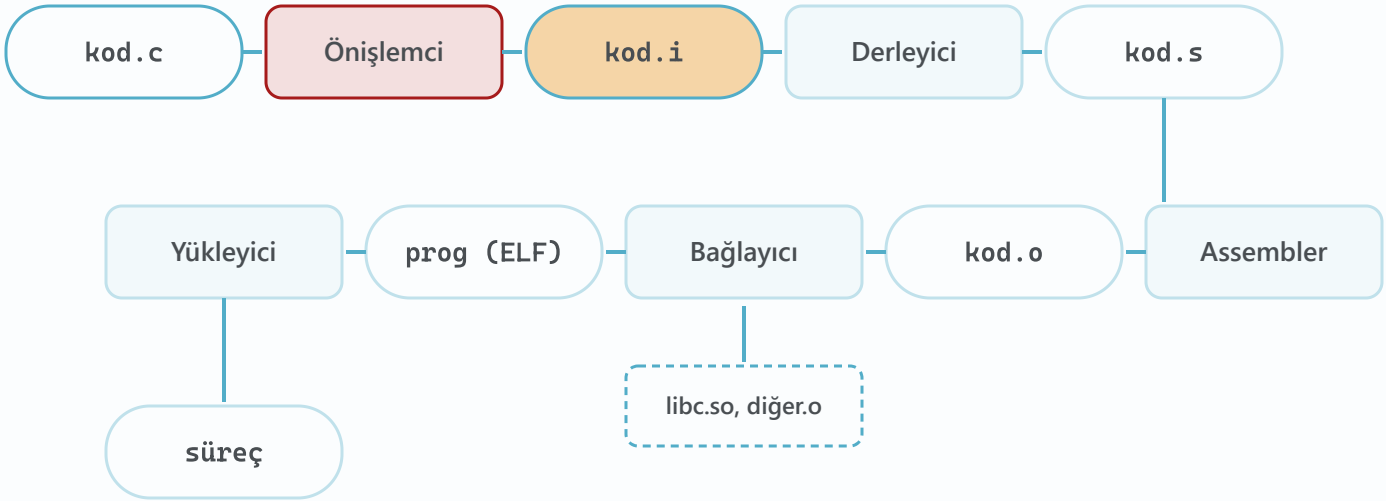
FreeBSD (UNIX türevi): çok görevli. Kullanıcı giriş yapınca kabuk başlar; kabuk `fork()` ile yeni süreç yaratır, `exec()` ile programı o sürece yükler; sonra programı bekler ya da (`&` ile) arka planda çalıştırıp komut almaya devam eder.

Çalıştırılabilir Dosya Oluşturma



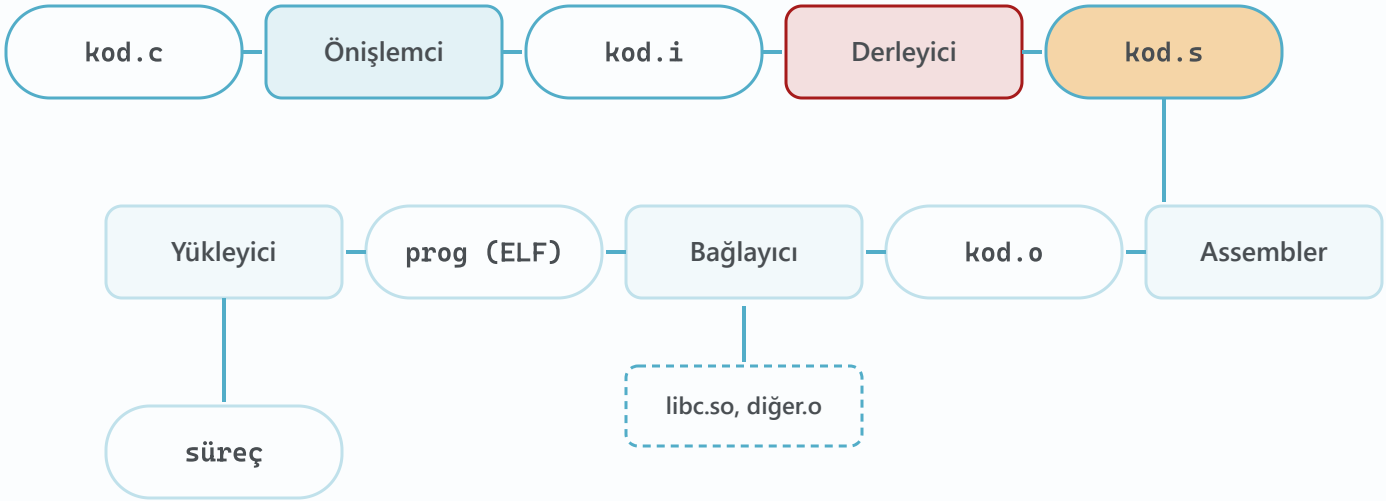
Kaynak dosya `kod.c` : insanın okuduğu C kodu.

Çalıştırılabilir Dosya Oluşturma



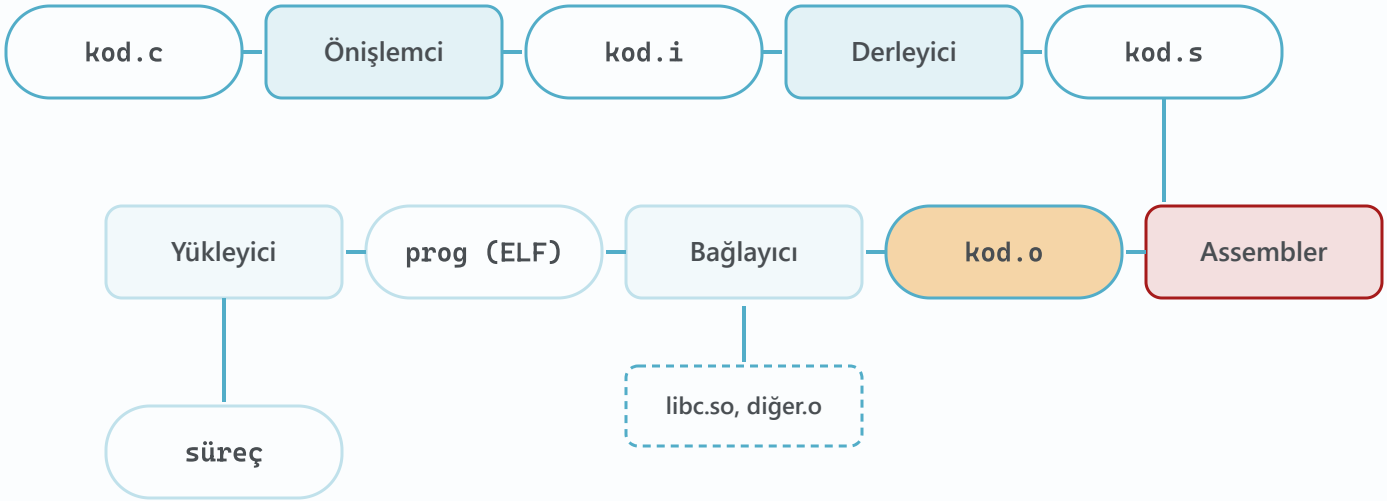
Önişlemci (`gcc -E`): `#include` ile başlık dosyalarını ekler, makroları genişletir, `#ifdef` ile koşullu derlemeyi çözer → `kod.i` .

Çalıştırılabilir Dosya Oluşturma



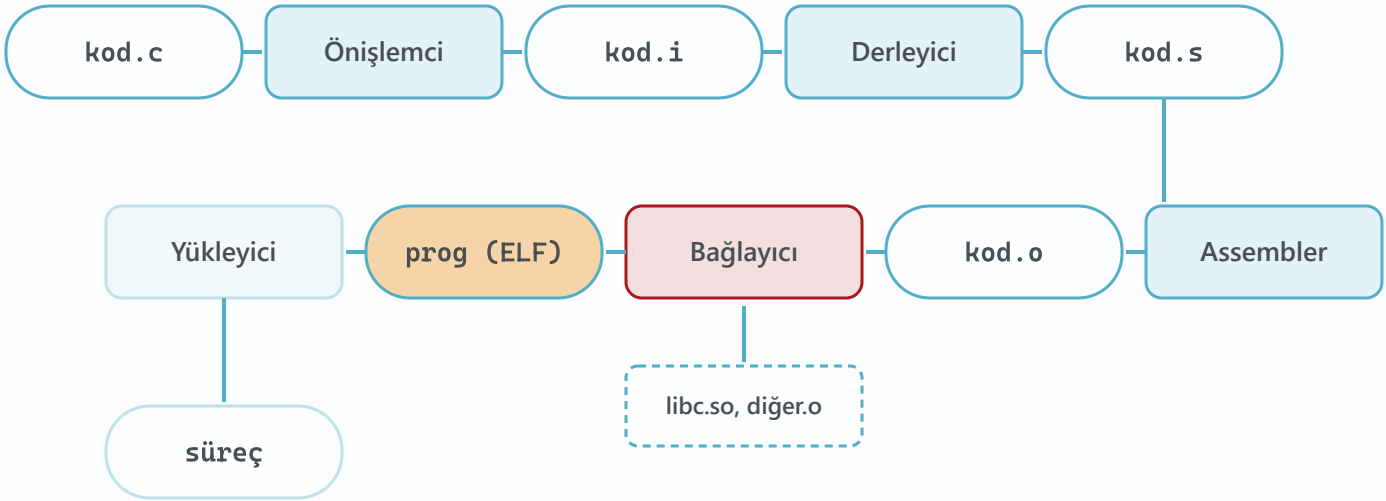
Derleyici (`gcc -S`): C kodunu hedef işlemcinin **assembly** diline çevirir, en iyileme yapar → `kod.s` .

Çalıştırılabilir Dosya Oluşturma



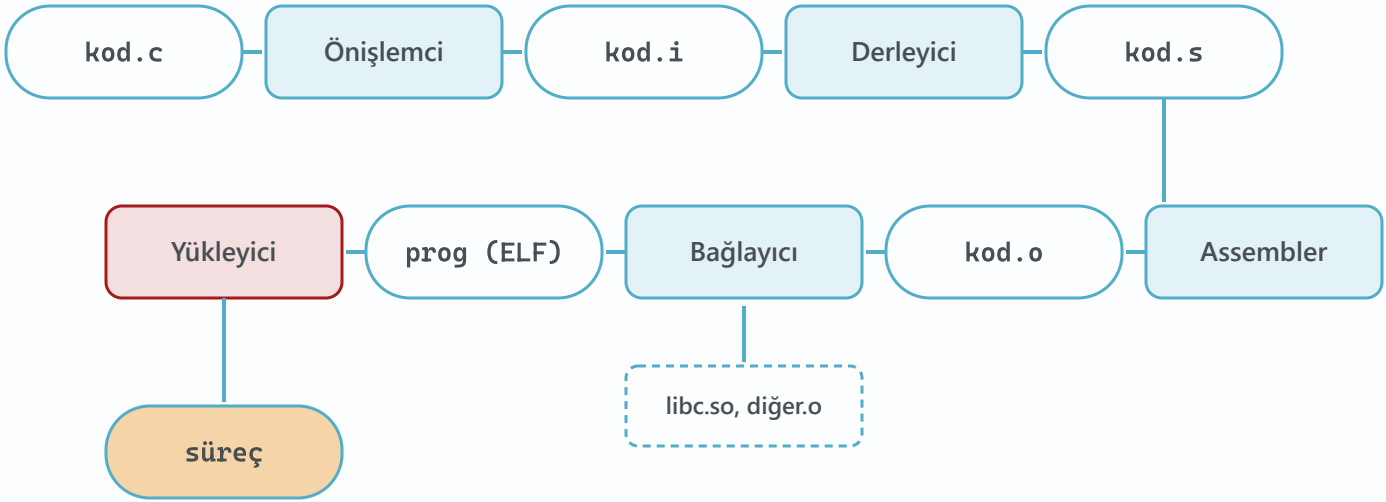
Assembler (`gcc -c`): assembly'yi makine koduna çevirir → **nesne dosyası** `kod.o`. Dış işlevlerin (printf) adresleri henüz belli değildir.

Çalıştırılabilir Dosya Oluşturma



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

Çalıştırılabilir Dosya Oluşturma



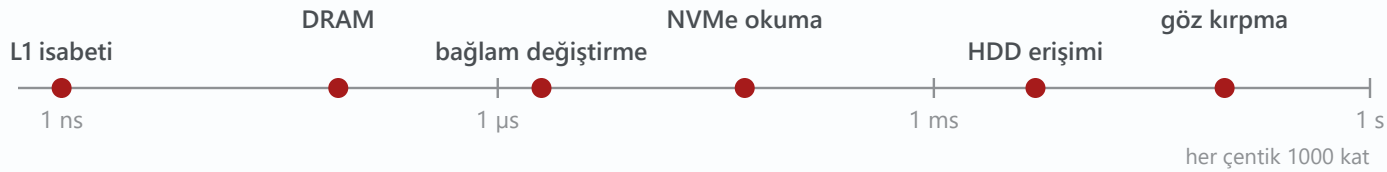
`./prog` : kabuk fork + exec yapar; çekirdeğin **yükleyicisi** ELF'i belleğe eşler, dinamik bağlayıcı libc.so'yu bağlar ve program bir **süreç** olarak çalışır.

Metrik Birimler

Üs	Ad	Sembol	Üs	Ad	Sembol
10^{-3}	mili	m	10^{3}	kilo	k
10^{-6}	mikro	µ	10^{6}	mega	M
10^{-9}	nano	n	10^{9}	giga	G
10^{-12}	piko	p	10^{12}	tera	T
10^{-15}	femto	f	10^{15}	peta	P
10^{-18}	atto	a	10^{18}	eksa	E

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

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



ÖZET

Kontrol Soruları

1. Kullanıcı programı ayrıcalıklı bir komut çalıştırırsa ne olur?
2. Çoklu programlama CPU kullanımını neden artırır? Gerçek paralellikten farkı nedir?
3. 4 KB'lık bir disk bloğunu belleğe almak için yoklama, kesme ve DMA'dan hangisi uygundur? Neden?
4. Bellek hiyerarşisinde üst düzeyler neden küçük, alt düzeyler neden büyüktür? Hiyerarşi neden işe yarar?
5. Bağlam değiştirmede İS neden tüm yazmaçları kaydetmek zorundadır?
6. Politika ile mekanizmayı bir örnekle ayırın.

▶ **Cevapları göster**

ÖZET

Kaynaklar

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

Son

Bölüm 1: Giriş

Sonraki bölüm: İşletim Sistemi Yapıları · Terim sözlüğü