Bölüm 11: Giriş/Çıkış

İşletim Sistemleri

G/Ç donanımı ve veri yolları

Aygıt yazmaçları, kapı ve bellek eşlemeli G/Ç

Yoklama (programlanmış G/Ç)

Kesmeler, vektörler, MSI-X

DMA, dağıt-topla, IOMMU

G/Ç yazılım katmanları

Tamponlama, önbellek, biriktirme

Bloke eden, eş zamansız G/Ç, io_uring

Saatler ve zamanlayıcılar

Diskler ve disk çizelgeleme

SSD, NVMe, blk-mq

RAID ve kararlı depolama

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_11_Giris_Cikis.html

GİRİŞ

Öğrenme Çıktıları

Donanım

Denetleyici, aygıt yazmaçları, PCIe/USB/NVMe veri yolları; kapı ve bellek eşlemeli G/Ç'yi açıklamak.

Aktarım yöntemleri

Yoklama, kesme güdümlü G/Ç ve DMA'yı karşılaştırmak; MSI-X ve IOMMU'nun rolünü söylemek.

Yazılım katmanları

Kesme işleyicisi, sürücü, aygıttan bağımsız katman ve kullanıcı kütüphanesini bir `read()` yolunda izlemek.

Arayüzler

Bloke eden, bloke etmeyen ve eş zamansız G/Ç'yi; `epoll` ve `io_uring` 'i ayırt etmek.

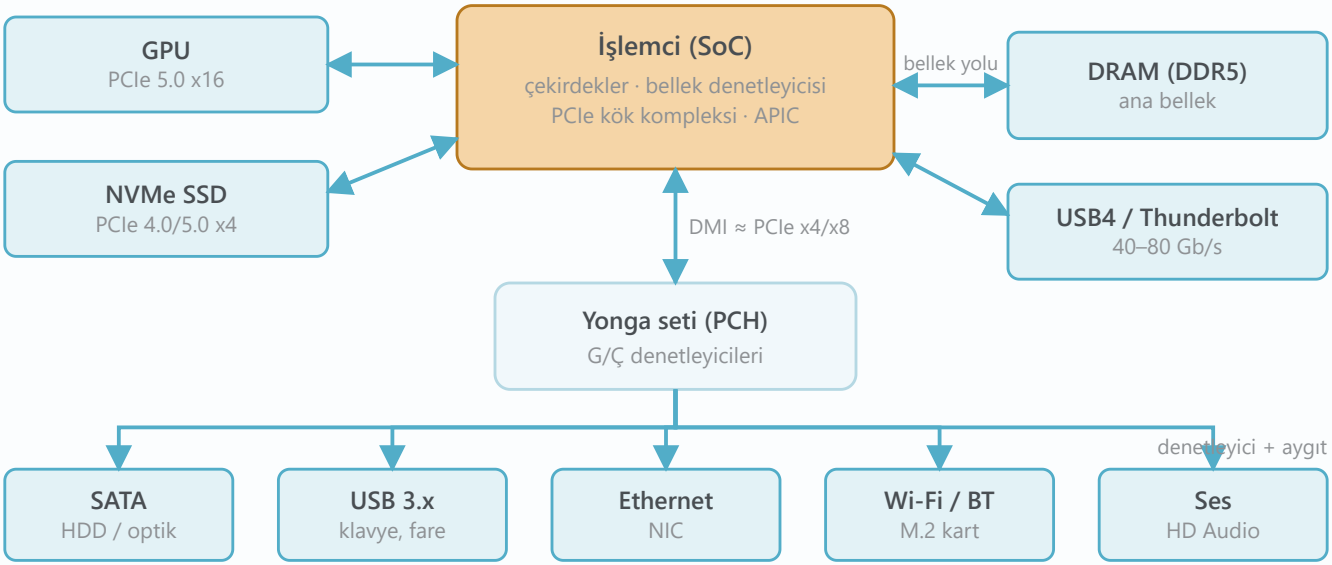
Diskler

Erişim süresini hesaplamak; FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK toplam kafa hareketini bulmak.

SSD ve RAID

FTL, aşınma dengeleme, TRIM ve blk-mq'yu; RAID 0/1/5/6/10 ve XOR eşliği ile kurtarmayı açıklamak.

G/Ç Sistemine Genel Bakış



- İşletim sistemi aygıtlara **komut verir**, **durumlarını okur**, **kesmeleri** yakalar ve **hataları** ele alır.
- Uygulamalara aygıttan bağımsız, katmanlı ve tek tip bir arayüz sunar.
- Çekirdek kodunun büyük bölümü **aygıt sürücüleridir** (Linux kaynak kodunun yaklaşık %60–70'i `drivers/`).

GİRİŞ

Veri Yolları: PCIe, USB4, SATA

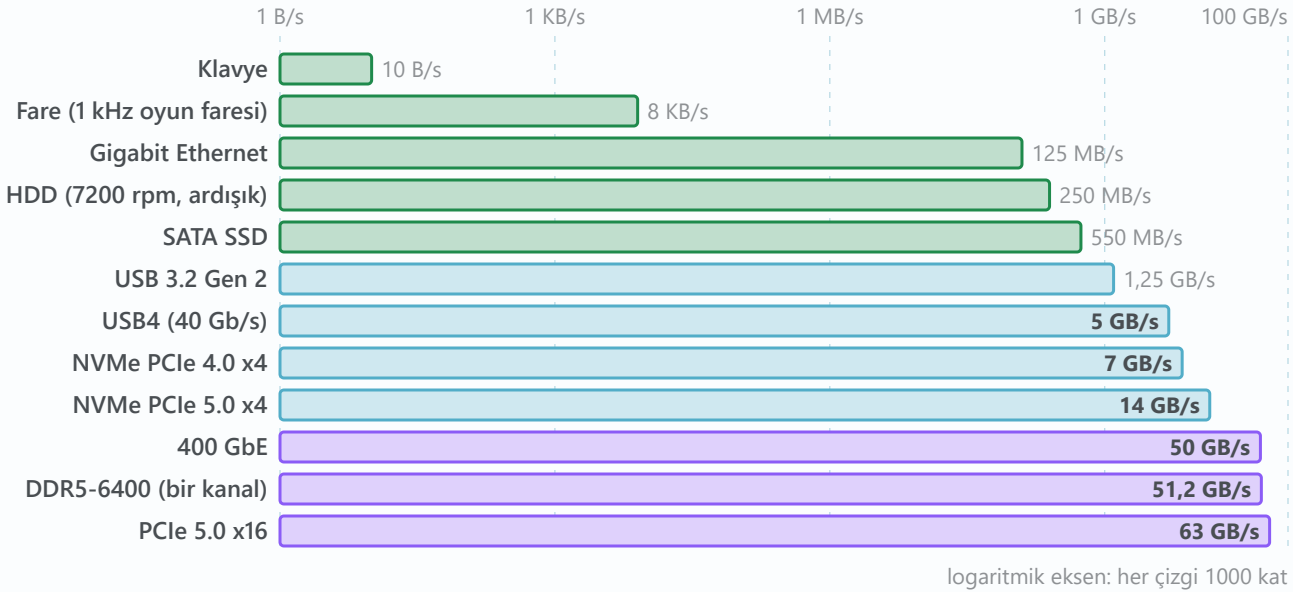
- **Veri yolu**: bileşenler arasında veri, adres ve denetim sinyallerini taşıyan bağlantı. Eski PC'lerde paylaşılan paralel yol (ISA, PCI); bugün **noktadan noktaya seri hatlar** (PCIe, USB, SATA).
- PCIe bir hat (lane) tam çift yönlüdür; bağlantı x1, x4, x8, x16 hat birleştirir. Bant genişliği her yönde ayrı ayrıdır.

Bağlantı	Ham hız	Etkin veri hızı	Not
PCIe 4.0	16 GT/s / hat	≈ 1,97 GB/s / hat; x4 ≈ 7,9 GB/s	128b/130b kodlama; NVMe SSD'ler
PCIe 5.0	32 GT/s / hat	≈ 3,94 GB/s / hat; x16 ≈ 63 GB/s	GPU, 14 GB/s sınıfı SSD
PCIe 6.0	64 GT/s (PAM4)	≈ 7,6 GB/s / hat; x16 ≈ 121 GB/s	FLIT + FEC; sunucularda yaygınlaşıyor
USB 2.0 / 3.2	480 Mb/s / 5–20 Gb/s	≤ 60 MB/s / ≤ 2,5 GB/s	Ana makine yoklamalı protokol
USB4 v1 / v2	40 / 80 Gb/s	PCIe, DisplayPort tünelleme	Thunderbolt 4/5 ile uyumlu
SATA III	6 Gb/s	≈ 600 MB/s (8b/10b)	AHCI: tek kuyruk, 32 komut
Ethernet	1 / 10 / 100 / 400 Gb/s	125 MB/s … 50 GB/s	Veri merkezi: 400–800 GbE

Hesap: PCIe 4.0 hattı 16 GT/s × 128/130 ÷ 8 = 1,97 GB/s. FireWire, 56K modem ve 10/100 Mb/s Ethernet tarihseldir. Pentium dönemindeki kuzey köprüsü kalktı: bellek denetleyicisi CPU'ya taşındı (AMD 2003, Intel 2008); hızlı aygıtlar PCIe ile doğrudan CPU'ya, yavaşlar yonga setine bağlanır.

GİRİŞ

Aygıt Veri Hızları



logaritmik eksen: her çizgi 1000 kat

Klavye ile PCIe 5.0 x16 arasında **≈ 6 × 10^9 kat** fark var: aynı işletim sistemi hem saniyede birkaç tuşu hem saniyede milyonlarca NVMe isteğini yönetmelidir. Değerler tipik üst sınırlardır; gerçek hız sinyal kalitesi, protokol yükü ve aygıta bağlıdır.

GİRİŞ

Aygıt Sınıfları: Blok, Karakter, Ağ

Sınıf	Örnek	Özellik	Tipik arayüz
Blok	HDD, SSD, USB bellek, SD kart	Sabit boyutlu, *adreslenebilir* bloklar (512 B / 4 KiB); her blok bağımsız okunur/yazılır	`read`, `write`, `lseek`; dosya sistemi, `mmap`
Karakter	Klavye, fare, seri port, yazıcı, ses	Bayt *akışı*; adreslenemez, konum (seek) yok	`read` / `write`, `ioctl`; `/dev/ttyS0`
Ağ	Ethernet, Wi-Fi	Paket tabanlı; ne blok ne karakter	Soket API'si (`send`, `recv`); `/dev`'de dosyası yok
Diğer	Saatler, zamanlayıcılar, GPU	Ne akış ne blok: kesme üretir ya da bellek eşlemelidir	`timerfd`, DRM/KMS, `mmap`

```
$ ls -l /dev/nvme0n1 /dev/ttyS0
brw-rw---- 1 root disk    259, 0  nvme0n1   b = blok, ana/yan numara 259,0
crw-rw---- 1 root dialout   4, 64 ttyS0     c = karakter
```

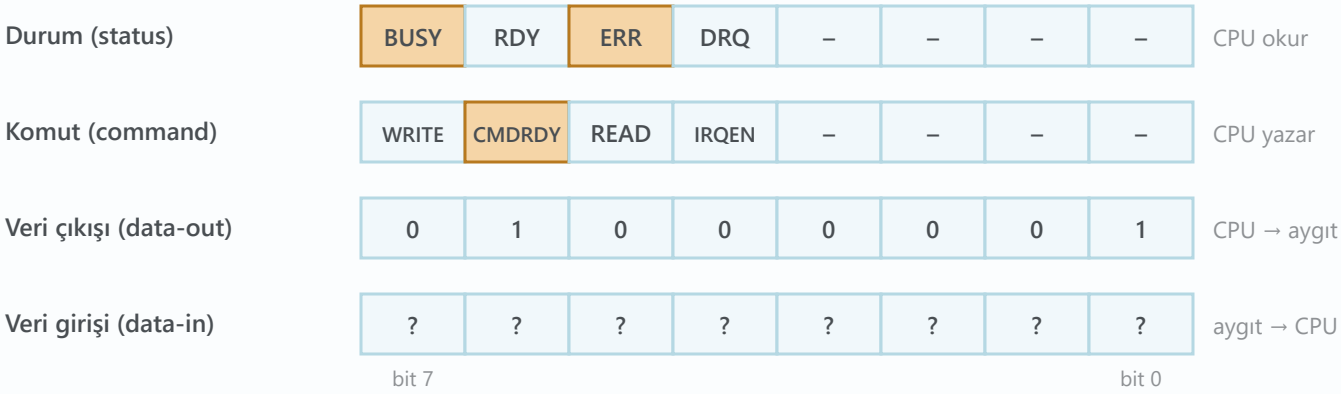
Ağ arabirimleri karakter aygıtı değildir; Unix/Linux'ta ağ ayrı bir sınıftır. Blok boyutu bugün çoğunlukla 512 B (öykünmeli) ya da 4 KiB'dir; sınıflandırma kesin değildir (teyp ve saatler sınıfa tam uymaz).

Aygıt Denetleyicileri



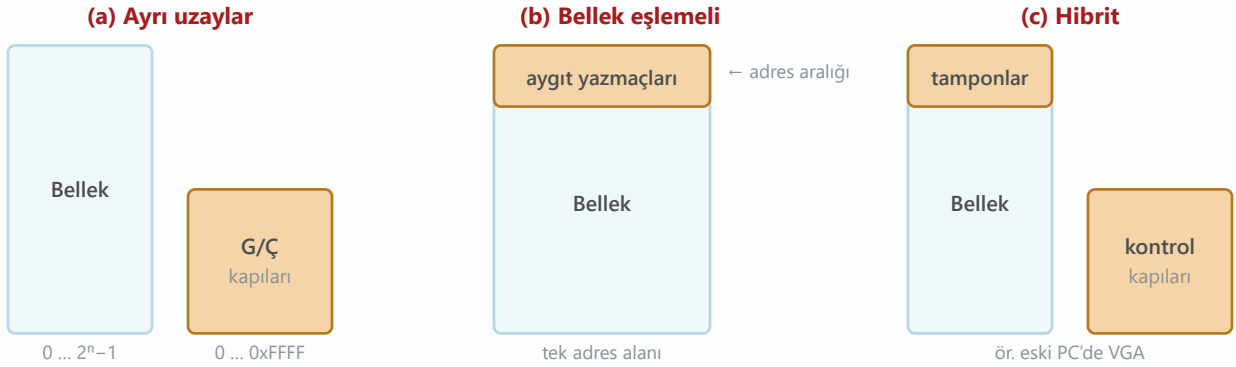
- G/Ç birimi = **mekanik/aygıt** kısmı + **elektronik denetleyici** (yonga ya da kart; bağlayıcı ile aygıta bağlanır).
- **Öncül** (preamble) senkronizasyon deseni ile silindir/sektör numarasını taşır. Denetleyici bit akışını bloğa çevirir, **ECC** ile hatayı düzeltir, bloğu kendi tamponuna koyar; sonra belleğe aktarır.
- Açılışta ve sıcak takmada İS yolları **tarar** (PCIe yapılandırma alanı, USB tanımlayıcıları), aygıt kimliğini okur ve uygun sürücüyü yükler: **tak-çalıştır**.
- İşletim sistemi aygıtla değil **denetleyiciyle** konuşur; SATA/NVMe gibi standart arayüzler sayesinde tek sürücü birçok üreticinin aygıtını sürer.
- Modern diskler 4 KiB fiziksel sektör kullanır (Advanced Format); dışarıya 512 B gösterenlere *512e* denir.

Aygıt Yazmaçları: Durum, Komut, Veri



- Denetleyicinin genellikle dört yazmacı vardır: **durum** (meşgul, hazır, hata), **komut** (ne yapılacağı), **veri girişi** ve **veri çıkışı**.
- Yazmaçlara yazmak = *komut vermek*; okumak = *durum öğrenmek*. Ayrıca büyük veri tamponları (ör. çerçeve tamponu, NIC halkaları) olabilir.
- Yazmaç düzeni aygıta özgüdür; bu ayrıntıyı yalnız **sürücü** bilir. Bit adları burada örnek amaçlıdır.

Kapı Eşlemeli ve Bellek Eşlemeli G/Ç



Kapı (x86)	Aygıt
0x20–0x21	8259 kesme denetleyicisi
0x40–0x43	8254 zamanlayıcı (PIT)
0x60, 0x64	8042 klavye denetleyicisi
0x3F8–0x3FF	COM1 seri port
0xCF8–0xCFF	PCI yapılandırma

```
in  al, 0x60          ; kapı: özel komut
mov eax, [rbx+0x14]   ; MMIO: sıradan yükleme

Linux: inb()/outb()  ·  ioremap() + readl()/writel()
/proc/ioports  ·  /proc/iomem
```

Kapı eşlemeli G/Ç'de CPU özel `IN` / `OUT` komutları ve ayrı bir adres alanı kullanır; bellek eşlemelide aygıt yazmacı sıradan bir bellek adresidir ve adres çözücü erişimi ilgili aygıta yönlendirir. ARM ve RISC-V yalnız MMIO kullanır; PCIe aygıtları da yazmaçlarını BAR'lar ile MMIO olarak sunar.

Bellek Eşlemeli G/Ç: Artılar ve Eksiler

Bellek eşlemeli G/Ç: artılar

+ Özel komut gerekmez; sürücü tamamen C ile yazılabilir (`volatile` işaretçi).
+ Koruma sayfa tablosuyla gelir: yazmaç sayfası kullanıcı uzayına eşlenmezse erişilemez; istenirse tek bir sürece verilebilir (VFIO, DPDK).
+ Her bellek komutu yazmaçla çalışır: `TEST` , `AND` gibi komutlar doğrudan yazmaç üzerinde.

Eksiler

– **Önbellek**: durum yazmacı önbelleğe alınırsa güncel değer görülmez → sayfa *önbelleklenemez* işaretlenir (PAT/MTRR, `ioremap`).

– **Birden çok yol**: bellek ayrı hızlı yolda ise G/Ç aygıtı adresi göremez → yönlendirme köprü/kök kompleks tarafından adres aralıklarıyla yapılır.

– Derleyici erişimleri yeniden sıralayabilir: `volatile` ve bellek bariyerleri gerekir.

(a) tek yol

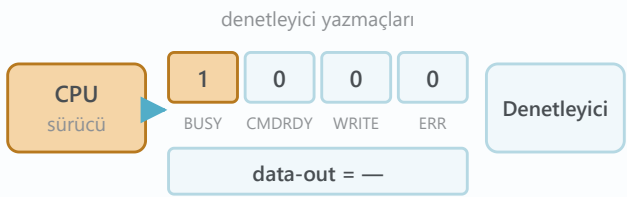


(b) çift yol



Yoklama: El Sıkışma Adım Adım

```
while (status & BUSY)
    ;           /* meşgul bekleme */
command = WRITE;
data_out = 'A';
command |= CMDRDY;
/* --- denetleyici tarafı --- */
/* CMDRDY görüldü → BUSY = 1 */
/* komutu oku, baytı aygıta yaz */
/* CMDRDY = ERR = BUSY = 0 */
```



Sürücü durum yazmacını **tekrar tekrar okur**: denetleyici önceki işi bitirmediği için BUSY = 1. CPU bu sırada başka iş yapamaz.

Adım 1 / 8

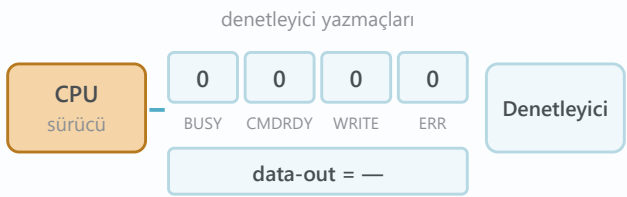
DEĞİŞKENLER

`BUSY = 1` `CMDRDY = 0` `WRITE = 0` `ERR = 0`

Programlanmış G/Ç (PIO): her bayt için CPU bu el sıkışmayı kendisi yürütür. Basittir, ama bekleme boyunca CPU boşa döner; CPU başka sürece geçip bir baytı kaçırırsa (ör. tampon taşması) veri kaybolabilir.

Yoklama: El Sıkışma Adım Adım

```
while (status & BUSY)
    ;           /* meşgul bekleme */
command = WRITE;
data_out = 'A';
command |= CMDRDY;
/* --- denetleyici tarafı --- */
/* CMDRDY görüldü → BUSY = 1 */
/* komutu oku, baytı aygıta yaz */
/* CMDRDY = ERR = BUSY = 0 */
```



BUSY 0 oldu: döngüden çıkılır. Bu bekleme süresi aygıta bağlıdır; birkaç döngüden milisaniyelere kadar sürebilir.

Adım 2 / 8

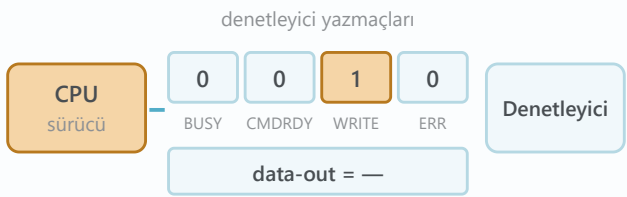
DEĞİŞKENLER

`BUSY = 0` `CMDRDY = 0` `WRITE = 0` `ERR = 0`

Programlanmış G/Ç (PIO): her bayt için CPU bu el sıkışmayı kendisi yürütür. Basittir, ama bekleme boyunca CPU boşa döner; CPU başka sürece geçip bir baytı kaçırırsa (ör. tampon taşması) veri kaybolabilir.

Yoklama: El Sıkışma Adım Adım

```
1 while (status & BUSY)
2     ;           /* meşgul bekleme */
▶ command = WRITE;
4 data_out = 'A';
5 command |= CMDRDY;
6 /* --- denetleyici tarafı --- */
7 /* CMDRDY görüldü → BUSY = 1 */
8 /* komutu oku, baytı aygıta yaz */
9 /* CMDRDY = ERR = BUSY = 0 */
```



Komut yazmacına **WRITE** biti yazılır: yapılacak iş yazma.

Adım 3 / 8

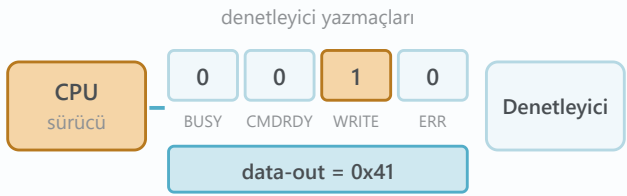
DEĞİŞKENLER

`BUSY = 0` `CMDRDY = 0` `WRITE = 1` `ERR = 0`

Programlanmış G/Ç (PIO): her bayt için CPU bu el sıkışmayı kendisi yürütür. Basittir, ama bekleme boyunca CPU boşa döner; CPU başka sürece geçip bir baytı kaçırırsa (ör. tampon taşması) veri kaybolabilir.

Yoklama: El Sıkışma Adım Adım

```
1 while (status & BUSY)
2     ;           /* meşgul bekleme */
3 command = WRITE;
▶ data_out = 'A';
5 command |= CMDRDY;
6 /* --- denetleyici tarafı --- */
7 /* CMDRDY görüldü → BUSY = 1 */
8 /* komutu oku, baytı aygıta yaz */
9 /* CMDRDY = ERR = BUSY = 0 */
```



Gönderilecek bayt ('A' = 0x41) **veri çıkış** yazmacına konur.

Adım 4 / 8

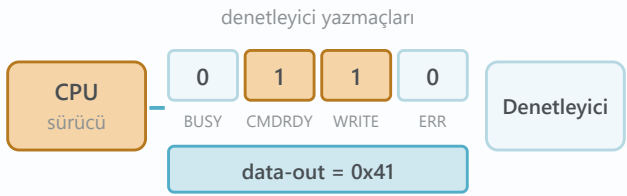
DEĞİŞKENLER

`BUSY = 0` `CMDRDY = 0` `WRITE = 1` `ERR = 0` `data_out = 0x41`

Programlanmış G/Ç (PIO): her bayt için CPU bu el sıkışmayı kendisi yürütür. Basittir, ama bekleme boyunca CPU boşa döner; CPU başka sürece geçip bir baytı kaçırırsa (ör. tampon taşması) veri kaybolabilir.

Yoklama: El Sıkışma Adım Adım

```
1 while (status & BUSY)
2     ;           /* meşgul bekleme */
3 command = WRITE;
4 data_out = 'A';
▶ command |= CMDRDY;
6 /* --- denetleyici tarafı --- */
7 /* CMDRDY görüldü → BUSY = 1 */
8 /* komutu oku, baytı aygıta yaz */
9 /* CMDRDY = ERR = BUSY = 0 */
```



CMDRDY = 1: “komut hazır”. Bu bit denetleyiciye işin başlayabileceğini söyler.

Adım 5 / 8

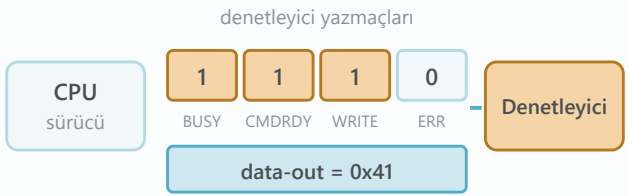
DEĞİŞKENLER

`BUSY = 0` `CMDRDY = 1` `WRITE = 1` `ERR = 0` `data_out = 0x41`

Programlanmış G/Ç (PIO): her bayt için CPU bu el sıkışmayı kendisi yürütür. Basittir, ama bekleme boyunca CPU boşa döner; CPU başka sürece geçip bir baytı kaçırırsa (ör. tampon taşması) veri kaybolabilir.

Yoklama: El Sıkışma Adım Adım

```
1 while (status & BUSY)
2     ;           /* meşgul bekleme */
3 command = WRITE;
4 data_out = 'A';
5 command |= CMDRDY;
6 /* --- denetleyici tarafı --- */
▶ /* CMDRDY görüldü → BUSY = 1 */
8 /* komutu oku, baytı aygıta yaz */
9 /* CMDRDY = ERR = BUSY = 0 */
```



Denetleyici CMDRDY'yi görür ve **BUSY = 1** yapar: artık meşgul olduğunu ana makineye bildirir.

Adım 6 / 8

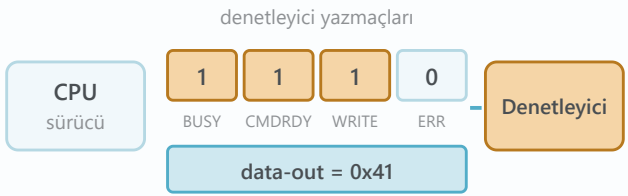
DEĞİŞKENLER

`BUSY = 1` `CMDRDY = 1` `WRITE = 1` `ERR = 0` `data_out = 0x41`

Programlanmış G/Ç (PIO): her bayt için CPU bu el sıkışmayı kendisi yürütür. Basittir, ama bekleme boyunca CPU boşa döner; CPU başka sürece geçip bir baytı kaçırırsa (ör. tampon taşması) veri kaybolabilir.

Yoklama: El Sıkışma Adım Adım

```
1 while (status & BUSY)
2     ;           /* meşgul bekleme */
3 command = WRITE;
4 data_out = 'A';
5 command |= CMDRDY;
6 /* --- denetleyici tarafı --- */
7 /* CMDRDY görüldü → BUSY = 1 */
▶ /* komutu oku, baytı aygıta yaz */
9 /* CMDRDY = ERR = BUSY = 0 */
```



Denetleyici komut yazmacını okur (WRITE), veri çıkış yazmacındaki baytı alır ve aygıta gönderir.

Adım 7 / 8

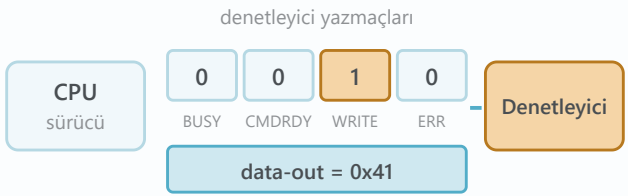
DEĞİŞKENLER

`BUSY = 1` `CMDRDY = 1` `WRITE = 1` `ERR = 0` `data_out = 0x41`

Programlanmış G/Ç (PIO): her bayt için CPU bu el sıkışmayı kendisi yürütür. Basittir, ama bekleme boyunca CPU boşa döner; CPU başka sürece geçip bir baytı kaçırırsa (ör. tampon taşması) veri kaybolabilir.

Yoklama: El Sıkışma Adım Adım

```
while (status & BUSY)
    ;          /* meşgul bekleme */
command = WRITE;
data_out = 'A';
command |= CMDRDY;
/* --- denetleyici tarafı --- */
/* CMDRDY görüldü → BUSY = 1 */
/* komutu oku, baytı aygıta yaz */
/* CMDRDY = ERR = BUSY = 0 */
```



İş bitti: CMDRDY, ERR ve BUSY temizlenir (ERR = 0 → başarılı). Sonraki bayt için döngü baştan başlar.

Adım 8 / 8

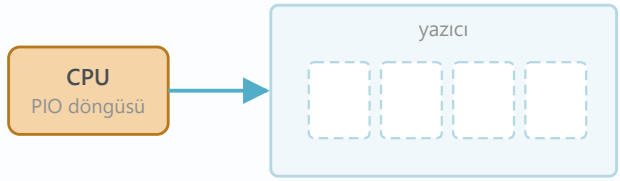
DEĞİŞKENLER

`BUSY = 0` `CMDRDY = 0` `WRITE = 1` `ERR = 0` `data_out = 0x41`

Programlanmış G/Ç (PIO): her bayt için CPU bu el sıkışmayı kendisi yürütür. Basittir, ama bekleme boyunca CPU boşa döner; CPU başka sürece geçip bir baytı kaçırırsa (ör. tampon taşması) veri kaybolabilir.

Programlanmış G/Ç: Yazıcı Örneği

```
▶ copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* meşgul bekleme */
5     *printer_data_register = p[i];
6 }
7 return_to_user();
```



Kullanıcının “ABCD” dizisi çekirdek tamponuna kopyalanır (`copy_from_user`); kullanıcı belleği doğrudan kullanılmaz.

Adım 1 / 10

DEĞİŞKENLER

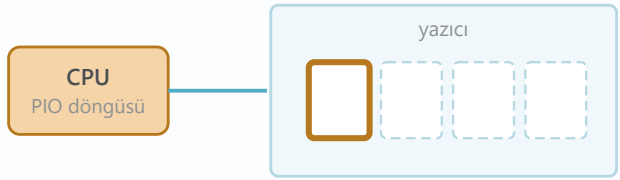
`i = –` `count = 4` `status = READY`

ÇIKTI

Tanenbaum’un yazıcı örneği. Yazıcı saniyede 100 karakter basıyorsa her karakter için CPU ≈ 10 ms bekler; bu sürede milyonlarca komut çalıştırabilirdi. Gömülü sistemlerde CPU’nun başka işi yoksa PIO yine de mantıklıdır.

Programlanmış G/Ç: Yazıcı Örneği

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
▶     while (*printer_status_reg != READY)
▶         ;   /* meşgul bekleme */
5     *printer_data_register = p[i];
6 }
7 return_to_user();
```



i = 0: durum READY olana kadar döngü döner.

Adım 2 / 10

DEĞİŞKENLER

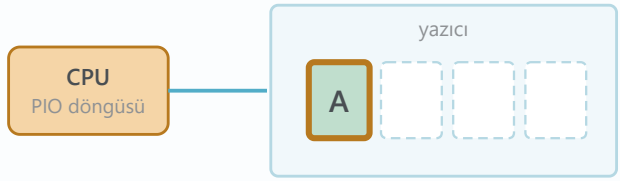
`i = 0` `count = 4` `status = BUSY`

ÇIKTI

Tanenbaum'un yazıcı örneği. Yazıcı saniyede 100 karakter basıyorsa her karakter için CPU ≈ 10 ms bekler; bu sürede milyonlarca komut çalıştırabilirdi. Gömülü sistemlerde CPU'nun başka işi yoksa PIO yine de mantıklıdır.

Programlanmış G/Ç: Yazıcı Örneği

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* meşgul bekleme */
▶     *printer_data_register = p[i];
6 }
7 return_to_user();
```



'A' veri yazmacına yazılır; yazıcı karakteri basar ve yeniden meşgul olur.

Adım 3 / 10

DEĞİŞKENLER

`i = 0` `count = 4` `status = READY`

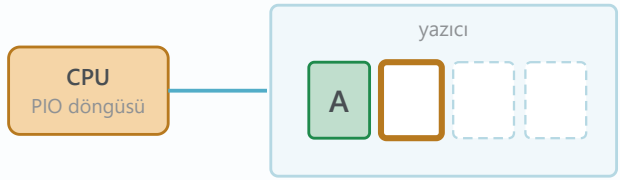
ÇIKTI

```
A
```

Tanenbaum'un yazıcı örneği. Yazıcı saniyede 100 karakter basıyorsa her karakter için CPU ≈ 10 ms bekler; bu sürede milyonlarca komut çalıştırabilirdi. Gömülü sistemlerde CPU'nun başka işi yoksa PIO yine de mantıklıdır.

Programlanmış G/Ç: Yazıcı Örneği

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
▶     while (*printer_status_reg != READY)
▶         ;   /* meşgul bekleme */
5     *printer_data_register = p[i];
6 }
7 return_to_user();
```



i = 1: yazıcı meşgul, CPU durum yazmacını **yoklar**.

Adım 4 / 10

DEĞİŞKENLER

`i = 1` `count = 4` `status = BUSY`

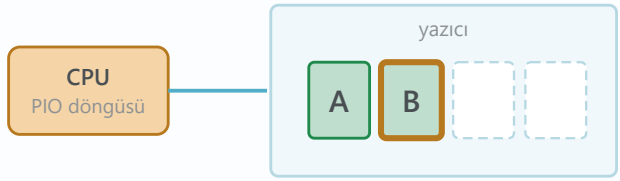
ÇIKTI

```
A
```

Tanenbaum'un yazıcı örneği. Yazıcı saniyede 100 karakter basıyorsa her karakter için CPU ≈ 10 ms bekler; bu sürede milyonlarca komut çalıştırabilirdi. Gömülü sistemlerde CPU'nun başka işi yoksa PIO yine de mantıklıdır.

Programlanmış G/Ç: Yazıcı Örneği

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* meşgul bekleme */
▶     *printer_data_register = p[i];
6 }
7 return_to_user();
```



'B' veri yazmacına yazılır; yazıcı karakteri basar ve yeniden meşgul olur.

Adım 5 / 10

DEĞİŞKENLER

`i = 1` `count = 4` `status = READY`

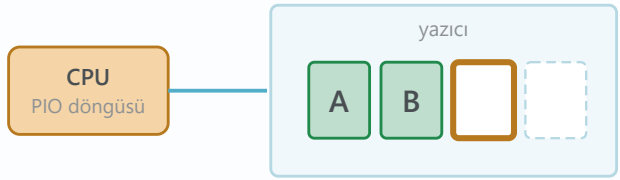
ÇIKTI

```
AB
```

Tanenbaum'un yazıcı örneği. Yazıcı saniyede 100 karakter basıyorsa her karakter için CPU ≈ 10 ms bekler; bu sürede milyonlarca komut çalıştırabilirdi. Gömülü sistemlerde CPU'nun başka işi yoksa PIO yine de mantıklıdır.

Programlanmış G/Ç: Yazıcı Örneği

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
▶     while (*printer_status_reg != READY)
▶         ;   /* meşgul bekleme */
5     *printer_data_register = p[i];
6 }
7 return_to_user();
```



i = 2: yazıcı meşgul, CPU durum yazmacını **yoklar**.

Adım 6 / 10

DEĞİŞKENLER

`i = 2` `count = 4` `status = BUSY`

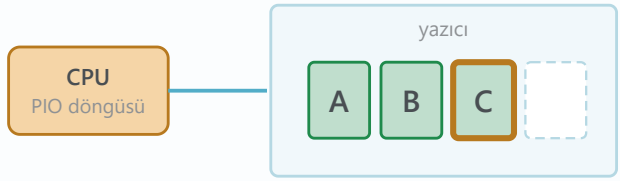
ÇIKTI

```
AB
```

Tanenbaum'un yazıcı örneği. Yazıcı saniyede 100 karakter basıyorsa her karakter için CPU ≈ 10 ms bekler; bu sürede milyonlarca komut çalıştırabilirdi. Gömülü sistemlerde CPU'nun başka işi yoksa PIO yine de mantıklıdır.

Programlanmış G/Ç: Yazıcı Örneği

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* meşgul bekleme */
▶     *printer_data_register = p[i];
6 }
7 return_to_user();
```



'C' veri yazmacına yazılır; yazıcı karakteri basar ve yeniden meşgul olur.

Adım 7 / 10

DEĞİŞKENLER

`i = 2` `count = 4` `status = READY`

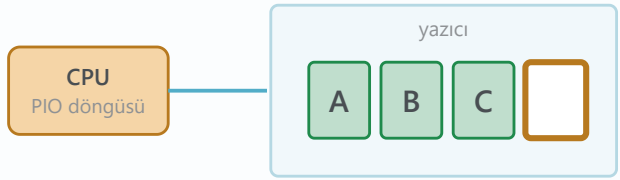
ÇIKTI

```
ABC
```

Tanenbaum'un yazıcı örneği. Yazıcı saniyede 100 karakter basıyorsa her karakter için CPU ≈ 10 ms bekler; bu sürede milyonlarca komut çalıştırabilirdi. Gömülü sistemlerde CPU'nun başka işi yoksa PIO yine de mantıklıdır.

Programlanmış G/Ç: Yazıcı Örneği

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
▶     while (*printer_status_reg != READY)
▶         ;   /* meşgul bekleme */
5     *printer_data_register = p[i];
6 }
7 return_to_user();
```



i = 3: yazıcı meşgul, CPU durum yazmacını **yoklar**.

Adım 8 / 10

DEĞİŞKENLER

`i = 3` `count = 4` `status = BUSY`

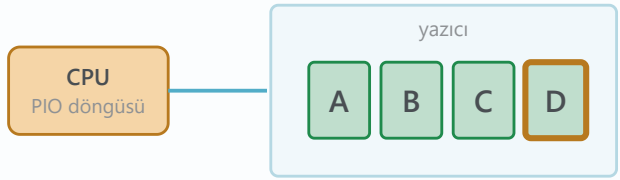
ÇIKTI

```
ABC
```

Tanenbaum'un yazıcı örneği. Yazıcı saniyede 100 karakter basıyorsa her karakter için CPU ≈ 10 ms bekler; bu sürede milyonlarca komut çalıştırabilirdi. Gömülü sistemlerde CPU'nun başka işi yoksa PIO yine de mantıklıdır.

Programlanmış G/Ç: Yazıcı Örneği

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* meşgul bekleme */
▶     *printer_data_register = p[i];
6 }
7 return_to_user();
```



'D' veri yazmacına yazılır; yazıcı karakteri basar ve yeniden meşgul olur.

Adım 9 / 10

DEĞİŞKENLER

`i = 3` `count = 4` `status = READY`

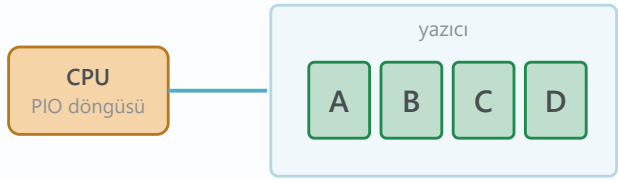
ÇIKTI

```
ABCD
```

Tanenbaum'un yazıcı örneği. Yazıcı saniyede 100 karakter basıyorsa her karakter için CPU ≈ 10 ms bekler; bu sürede milyonlarca komut çalıştırabilirdi. Gömülü sistemlerde CPU'nun başka işi yoksa PIO yine de mantıklıdır.

Programlanmış G/Ç: Yazıcı Örneği

```
1 copy_from_user(buffer, p, count);
2 for (i = 0; i < count; i++) {
3     while (*printer_status_reg != READY)
4         ;   /* meşgul bekleme */
5     *printer_data_register = p[i];
6 }
▶ return_to_user();
```



Döngü bitti; sürece dönülür. CPU tüm süre boyunca bu sürece bağlı kaldı: 4 karakterlik iş, yazıcının hızıyla sınırlı.

Adım 10 / 10

DEĞİŞKENLER

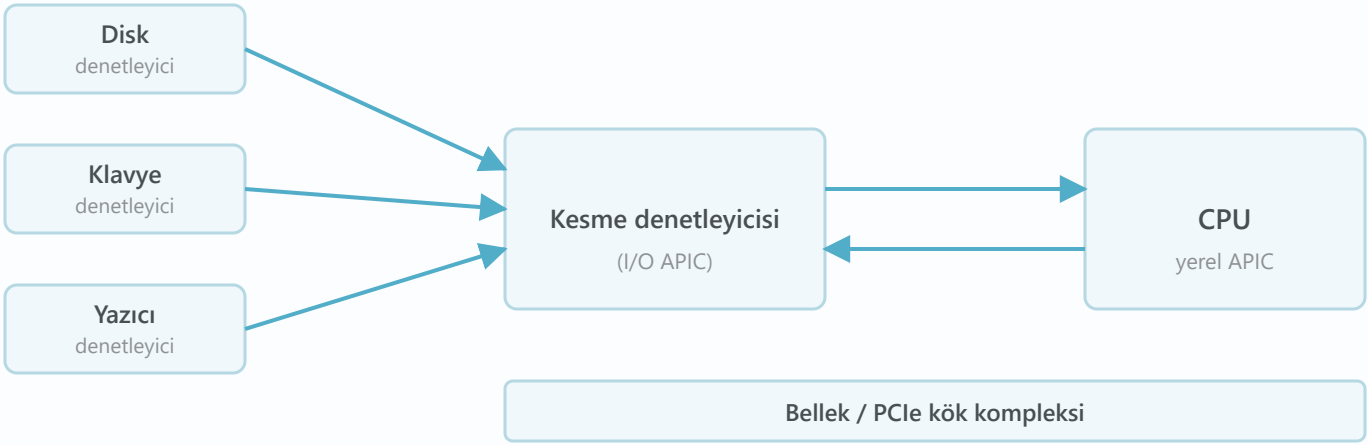
`i = 4` `count = 4` `status = READY`

ÇIKTI

```
ABCD
```

Tanenbaum'un yazıcı örneği. Yazıcı saniyede 100 karakter basıyorsa her karakter için CPU ≈ 10 ms bekler; bu sürede milyonlarca komut çalıştırabilirdi. Gömülü sistemlerde CPU'nun başka işi yoksa PIO yine de mantıklıdır.

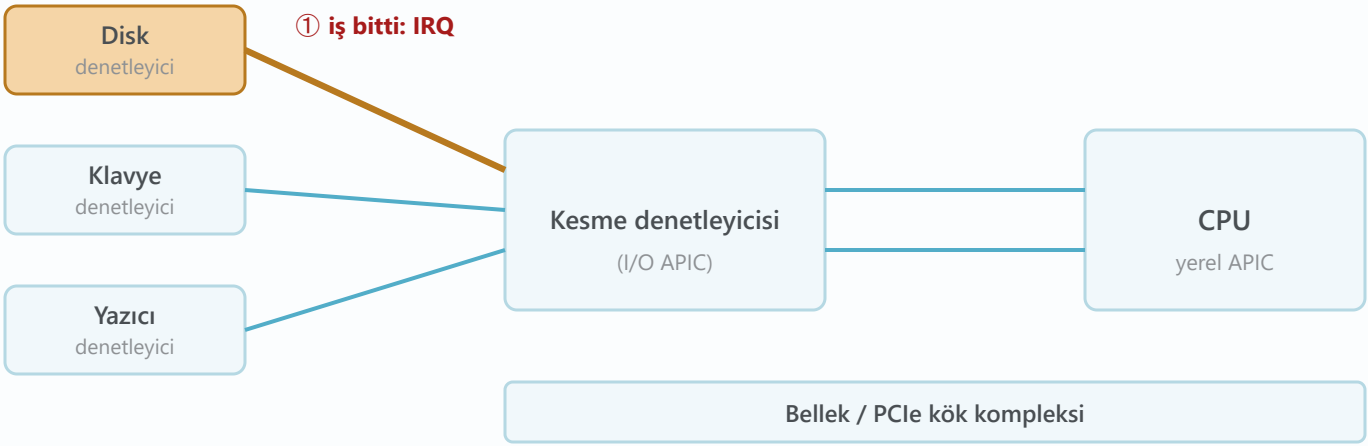
Kesme Donanımı: Hattan MSI'ya



Aygıtların her biri kesme denetleyicisine bağlıdır; denetleyici CPU'nun tek *kesme girişini* paylaştırır ve öncelik belirler.

Eski PC: iki 8259 PIC, 15 IRQ hattı. Bugün x86'da I/O APIC + çekirdek başına yerel APIC (x2APIC), ARM'da GIC. Denetleyicinin "adres satırına sayı koyması" anlatımı yalnızca bu tarihsel PIC düzeni için geçerlidir.

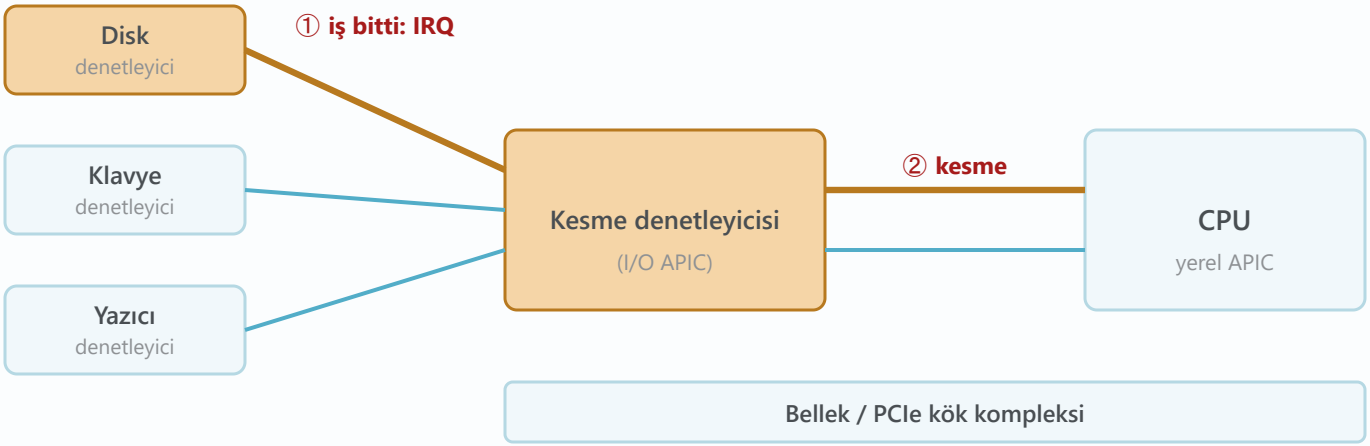
Kesme Donanımı: Hattan MSI'ya



① Disk denetleyicisi işini bitirince kendi **kesme hattına** (IRQ) sinyal verir.

Eski PC: iki 8259 PIC, 15 IRQ hattı. Bugün x86'da I/O APIC + çekirdek başına yerel APIC (x2APIC), ARM'da GIC. Denetleyicinin "adres satırına sayı koyması" anlatımı yalnızca bu tarihsel PIC düzeni için geçerlidir.

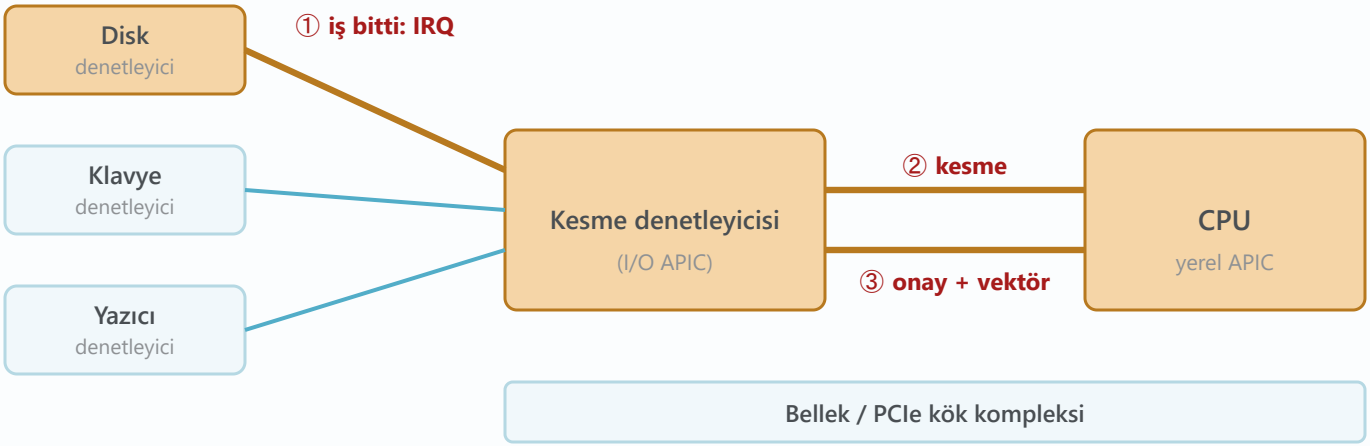
Kesme Donanımı: Hattan MSI'ya



② Denetleyici başka kesme işlenmiyorsa ya da bu daha öncelikliyse CPU'ya kesme iletir; değilse istek bekletilir (CPU onaylayana kadar hat etkin kalır).

Eski PC: iki 8259 PIC, 15 IRQ hattı. Bugün x86'da I/O APIC + çekirdek başına yerel APIC (x2APIC), ARM'da GIC. Denetleyicinin "adres satırına sayı koyması" anlatımı yalnızca bu tarihsel PIC düzeni için geçerlidir.

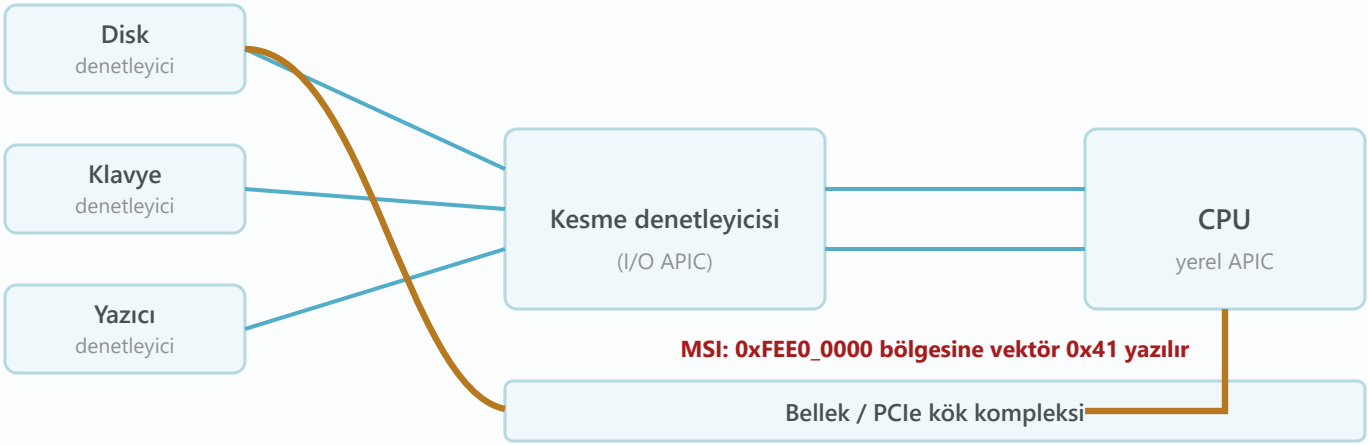
Kesme Donanımı: Hattan MSI'ya



③ CPU kesmeyi kabul eder; denetleyici **vektör numarasını** yola koyar. CPU bu sayıyı kesme vektör tablosunda indis olarak kullanır.

Eski PC: iki 8259 PIC, 15 IRQ hattı. Bugün x86'da I/O APIC + çekirdek başına yerel APIC (x2APIC), ARM'da GIC. Denetleyicinin "adres satırına sayı koyması" anlatımı yalnızca bu tarihsel PIC düzeni için geçerlidir.

Kesme Donanımı: Hattan MSI'ya



Modern PCIe: **MSI/MSI-X**. Kablo yok; aygıt özel bir adrese (x86'da 0xFEE0_0000 bölgesi) vektörü içeren bir **bellek yazması** gönderir, yerel APIC bunu kesmeye çevirir.

Eski PC: iki 8259 PIC, 15 IRQ hattı. Bugün x86'da I/O APIC + çekirdek başına yerel APIC (x2APIC), ARM'da GIC. Denetleyicinin "adres satırına sayı koyması" anlatımı yalnızca bu tarihsel PIC düzeni için geçerlidir.

Vektörler, Öncelik, NMI ve İstisnalar

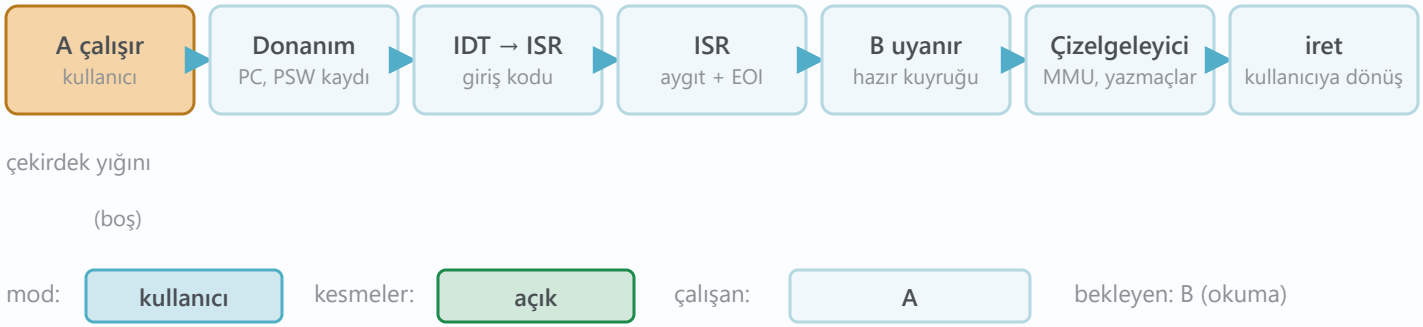
Vektör	Olay	Tür
0	Bölme hatası	hata (fault)
1 / 3	Hata ayıklama / kesme noktası	tuzak (trap)
2	NMI (maskelenemez kesme)	kesme
6	Geçersiz işlem kodu	hata
13	Genel koruma hatası	hata
14	Sayfa hatası	hata
18	Makine denetimi	iptal (abort)
32–255	Aygıt kesmeleri, IPI, APIC zamanlayıcı	maskelenebilir



- **Kesme vektör tablosu** (x86: IDT, 256 giriş): vektör → işleyici adresi. 0–31 işlemci istisnalarına ayrılmıştır.
- **Maskelenebilir** kesmeler kritik bölgede ertelenebilir (`cli` / `sti`); **NMI** ertelenemez: donanım hatası, bellek ECC, izleme köpeği.
- **Öncelik**: APIC vektör sınıfına göre sıralar; yüksek öncelikli kesme düşük öncelikli işleyiciyi kesebilir.
- **Kesme zinciri**: aygıt sayısı vektörden fazlaysa her girdi bir işleyici listesine işaret eder (paylaşılan IRQ).
- **Hata** (sayfa hatası): komut tamamlanamaz, işleyiciden sonra *aynı komut* yeniden çalışır. **Tuzak** (sistem çağrısı, `int3`): sonraki komuttan devam edilir.
- **MSI-X**: işlev başına 2048 vektöre kadar; NVMe ve NIC her kuyruk için ayrı vektör alır ve kesmeyi kuyruğun çekirdeğine yönlendirir (`/proc/interrupts` , `irqbalance`).

KESMELER · ADIM ADIM

Kesme İşleme Adım Adım



Süreç A çalışıyor; B bir disk okumasının bitmesini bekliyor. Disk kesme isteği gelir. CPU **o anki komutu bitirir**, sonra kesmeyi kabul eder.

Adım 1 / 7

Bu sıra Tanenbaum'un kesme işleyicisi adımlarını izler. Çizelgeleyici A'yı da seçebilir; seçim politikaya bağlıdır. Ayrıntılı benzetim: Kesme, DMA ve G/Ç Akışı Simülatörü.

KESMELER · ADIM ADIM

Kesme İşleme Adım Adım



Donanım PC (RIP), PSW (RFLAGS) ve yığın işaretçisini **çekirdek yığınına** iter, çekirdek moduna geçer, kesmeleri maskeler.

Adım 2 / 7

Bu sıra Tanenbaum'un kesme işleyicisi adımlarını izler. Çizelgeleyici A'yı da seçebilir; seçim politikaya bağlıdır. Ayrıntılı benzetim: Kesme, DMA ve G/Ç Akışı Simülatörü.

KESMELER · ADIM ADIM

Kesme İşleme Adım Adım



Vektör numarası IDT'de indis olur → işleyici adresi. Giriş kodu (assembly) donanımın kaydetmediği yazmaçları kaydeder; ISR için bağlam ve yığın hazırdır.

Adım 3 / 7

Bu sıra Tanenbaum'un kesme işleyicisi adımlarını izler. Çizelgeleyici A'yı da seçebilir; seçim politikaya bağlıdır. Ayrıntılı benzetim: Kesme, DMA ve G/Ç Akışı Simülatörü.

KESMELER · ADIM ADIM

Kesme İşleme Adım Adım



ISR aygıtın durumunu okur, veriyi alır ve denetleyiciye **onay (EOI)** gönderir; yeni kesmeler artık gelebilir. Uzun iş alt yarıya ertelenir.

Adım 4 / 7

Bu sıra Tanenbaum'un kesme işleyicisi adımlarını izler. Çizelgeleyici A'yı da seçebilir; seçim politikaya bağlıdır. Ayrıntılı benzetim: Kesme, DMA ve G/Ç Akışı Simülatörü.

Kesme İşleme Adım Adım



Bekleyen süreç B **bloke** durumundan **hazır** kuyruğuna alınır.

Bu sıra Tanenbaum'un kesme işleyicisi adımlarını izler. Çizelgeleyici A'yı da seçebilir; seçim politikaya bağlıdır. Ayrıntılı benzetim: Kesme, DMA ve G/Ç Akışı Simülatörü.

Kesme İşleme Adım Adım

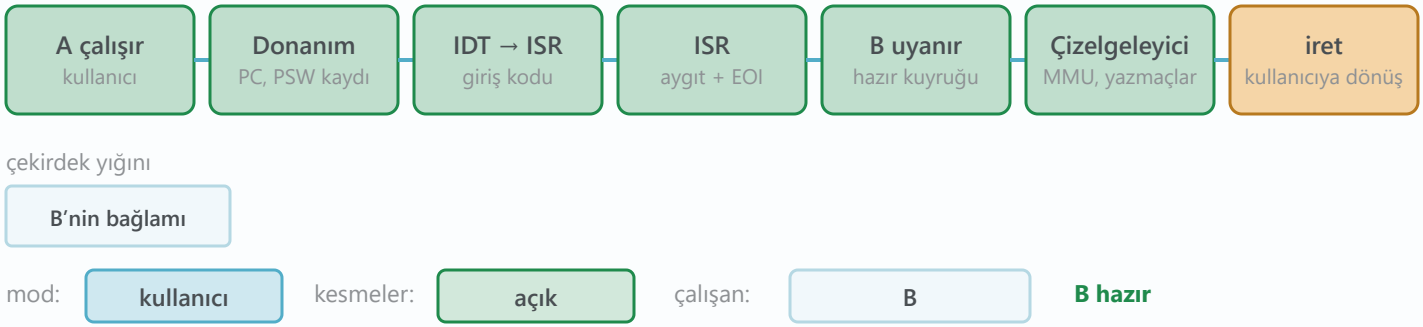


Çizelgeleyici sıradakini seçer (burada B). B'nin sayfa tablosu MMU'ya yüklenir (x86: CR3), yazmaçları süreç tablosu/yığından geri alınır.

Bu sıra Tanenbaum'un kesme işleyicisi adımlarını izler. Çizelgeleyici A'yı da seçebilir; seçim politikaya bağlıdır. Ayrıntılı benzetim: Kesme, DMA ve G/Ç Akışı Simülatörü.

KESMELER · ADIM ADIM

Kesme İşleme Adım Adım



`iret` : kaydedilen PC ve PSW geri yüklenir, kullanıcı moduna dönülür, kesmeler açılır. B kaldığı komuttan devam eder.

Adım 7 / 7

Bu sıra Tanenbaum'un kesme işleyicisi adımlarını izler. Çizelgeleyici A'yı da seçebilir; seçim politikaya bağlıdır. Ayrıntılı benzetim: [Kesme, DMA ve G/Ç Akışı Simülatörü].

Kesin ve Kesin Olmayan Kesmeler

(a) Kesin kesme: PC = 316



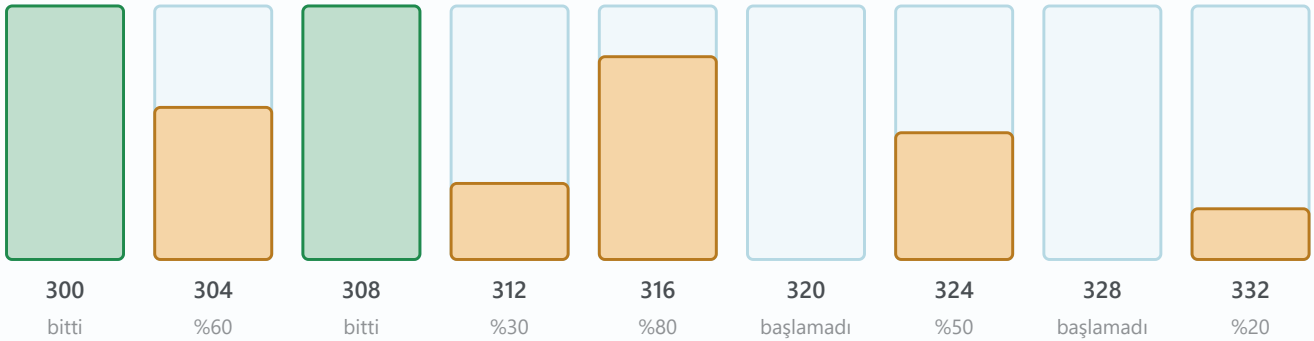
Kesin kesme: PC bilinen yere kaydedilir; PC'den önceki her komut **tamamen** bitmiş, sonrakilerin **hiçbiri** başlamamıştır; PC'deki komutun durumu bilinir.

Adım 1 / 2

- Boru hattında aynı anda birçok komut farklı aşamalardadır; üstün ölçekli işlemcide komutlar sıra dışı tamamlanır. "PC" tek başına durumu anlatmaz.
- Modern x86, ARM ve RISC-V işlemciler **kesin kesme** sağlar: yeniden sıralama arabelleği (ROB) komutları program sırasıyla emekliye ayırır; kesmede sonraki komutların etkileri atılır. Bedeli yonga alanı ve karmaşıklıktır.

Kesin ve Kesin Olmayan Kesmeler

(b) Kesin olmayan: komutlar farklı aşamalarda



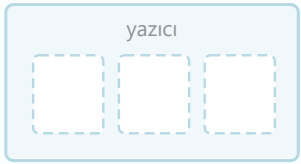
Kesin olmayan kesme: boru hattı / sıra dışı yürütmede komutlar yarım kalmıştır. Yeniden başlatmak için ya donanım geri alır ya da işletim sistemi çok miktarda iç durumu kaydedip çözümlemelidir.

Adım 2 / 2

- Boru hattında aynı anda birçok komut farklı aşamalardadır; üstün ölçekli işlemcide komutlar sıra dışı tamamlanır. “PC” tek başına durumu anlatmaz.
- Modern x86, ARM ve RISC-V işlemciler **kesin kesme** sağlar: yeniden sıralama arabelleği (ROB) komutları program sırasıyla emekliye ayırır; kesmede sonraki komutların etkileri atılır. Bedeli yonga alanı ve karmaşıklıktır.

Kesme Güdümlü G/Ç: Yazıcı Örneği

```
1  /* sistem çağrısı */
▶ copy_from_user(buffer, p, count);
▶ enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* süreç bloke */
8  /* kesme hizmet yordamı */
9  if (count == 0) {
10     unblock_user();
11 } else {
12     *printer_data_register = p[i];
13     count = count - 1;
14     i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



"ABC" çekirdek tamponuna kopyalanır, yazıcı kesmeleri açılır.

Adım 1 / 7

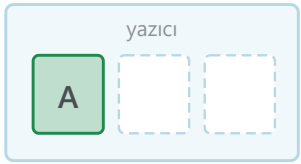
DEĞİŞKENLER

`i = –` `count = 3`

Her karakter için bir kesme: 3 karakter → 3 kesme. Kesme yordamındaki 13. ve 14. satırlar `i` ve `count`'u her karakterden sonra günceller. Bedel: kesme başına bağlam kaydı ve işleyici süresi (≈ 1–2 µs).

Kesme Güdümlü G/Ç: Yazıcı Örneği

```
1  /* sistem çağrısı */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
▶ while (*printer_status_reg != READY) ;
▶ *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* süreç bloke */
8  /* kesme hizmet yordamı */
9  if (count == 0) {
10     unblock_user();
11 } else {
12     *printer_data_register = p[i];
13     count = count - 1;
14     i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



süreç: çalışıyor

CPU: süreç

Yazıcı hazır olunca **yalnız ilk karakter** yazılır.

Adım 2 / 7

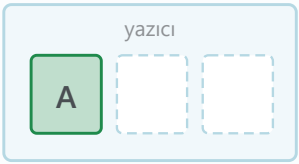
DEĞİŞKENLER

`i = –` `count = 3`

Her karakter için bir kesme: 3 karakter → 3 kesme. Kesme yordamındaki 13. ve 14. satırlar `i` ve `count`'u her karakterden sonra günceller. Bedel: kesme başına bağlam kaydı ve işleyici süresi (≈ 1–2 µs).

Kesme Güdümlü G/Ç: Yazıcı Örneği

```
1  /* sistem çağrısı */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
▶  i = 1; count = count - 1;
▶  scheduler();      /* süreç bloke */
8  /* kesme hizmet yordamı */
9  if (count == 0) {
10     unblock_user();
11 } else {
12     *printer_data_register = p[i];
13     count = count - 1;
14     i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



süreç: bloke

CPU: başka süreç

Sayaçlar güncellenir ve çizelgeleyici çağrılır: süreç **bloke**, CPU başka bir süreci çalıştırır.

Adım 3 / 7

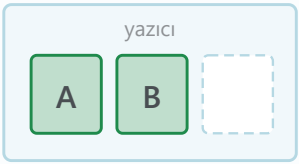
DEĞİŞKENLER

`i = 1` `count = 2`

Her karakter için bir kesme: 3 karakter → 3 kesme. Kesme yordamındaki 13. ve 14. satırlar `i` ve `count`'u her karakterden sonra günceller. Bedel: kesme başına bağlam kaydı ve işleyici süresi (≈ 1–2 µs).

Kesme Güdümlü G/Ç: Yazıcı Örneği

```
1  /* sistem çağrısı */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* süreç bloke */
8  /* kesme hizmet yordamı */
9  if (count == 0) {
10     unblock_user();
▶  } else {
▶      *printer_data_register = p[i];
▶      count = count - 1;
▶      i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



1. kesme: 'A' basıldı. count ≠ 0 → p[1] = 'B' yazılır.

Adım 4 / 7

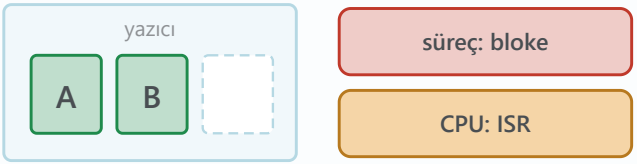
DEĞİŞKENLER

`i = 2` `count = 1`

Her karakter için bir kesme: 3 karakter → 3 kesme. Kesme yordamındaki 13. ve 14. satırlar `i` ve `count`'u her karakterden sonra günceller. Bedel: kesme başına bağlam kaydı ve işleyici süresi (≈ 1–2 µs).

Kesme Güdümlü G/Ç: Yazıcı Örneği

```
1  /* sistem çağrısı */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* süreç bloke */
8  /* kesme hizmet yordamı */
9  if (count == 0) {
10     unblock_user();
11 } else {
12     *printer_data_register = p[i];
13     count = count - 1;
14     i = i + 1;
15 }
   acknowledge_interrupt();
   return_from_interrupt();
```



Kesme onaylanır ve kesilen sürece dönülür. CPU yazıcı beklerken boşa dönmedi.

Adım 5 / 7

DEĞİŞKENLER

`i = 2` `count = 1`

Her karakter için bir kesme: 3 karakter → 3 kesme. Kesme yordamındaki 13. ve 14. satırlar `i` ve `count`'u her karakterden sonra günceller. Bedel: kesme başına bağlam kaydı ve işleyici süresi (≈ 1–2 µs).

Kesme Güdümlü G/Ç: Yazıcı Örneği

```
1  /* sistem çağrısı */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* süreç bloke */
8  /* kesme hizmet yordamı */
9  if (count == 0) {
10     unblock_user();
11 } else {
▶      *printer_data_register = p[i];
▶      count = count - 1;
▶      i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



2. kesme: 'B' basıldı → p[2] = 'C' yazılır.

Adım 6 / 7

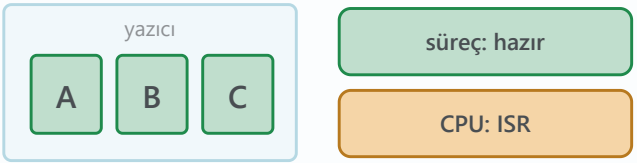
DEĞİŞKENLER

`i = 3` `count = 0`

Her karakter için bir kesme: 3 karakter → 3 kesme. Kesme yordamındaki 13. ve 14. satırlar `i` ve `count`'u her karakterden sonra günceller. Bedel: kesme başına bağlam kaydı ve işleyici süresi (≈ 1–2 µs).

Kesme Güdümlü G/Ç: Yazıcı Örneği

```
1  /* sistem çağrısı */
2  copy_from_user(buffer, p, count);
3  enable_interrupts();
4  while (*printer_status_reg != READY) ;
5  *printer_data_register = p[0];
6  i = 1; count = count - 1;
7  scheduler();      /* süreç bloke */
8  /* kesme hizmet yordamı */
▶  if (count == 0) {
▶      unblock_user();
11 } else {
12     *printer_data_register = p[i];
13     count = count - 1;
14     i = i + 1;
15 }
16 acknowledge_interrupt();
17 return_from_interrupt();
```



3. kesme: count == 0 → tüm karakterler basıldı, kullanıcı süreci **hazır** kuyruğuna alınır.

Adım 7 / 7

DEĞİŞKENLER

`i = 3` `count = 0`

Her karakter için bir kesme: 3 karakter → 3 kesme. Kesme yordamındaki 13. ve 14. satırlar `i` ve `count`'u her karakterden sonra günceller. Bedel: kesme başına bağlam kaydı ve işleyici süresi (≈ 1–2 µs).

KESMELER

Üst Yarı, Alt Yarı ve Hibrit Yoklama

Üst yarı (hardirq)

Gerçek ISR: kısa tutulur, kesmeler kısmen kapalıdır. Aygıtı onaylar, veriyi tampona alır, alt yarıyı planlar.

Alt yarı

Ertelenmiş iş: Linux'ta *softirq*, *tasklet*, *workqueue* ya da iş parçacıklı IRQ (`request_threaded_irq`); Windows'ta ISR + DPC.

Kesme birleştirme

NIC ve NVMe birkaç tamamlanmayı tek kesmede bildirir (coalescing): daha az kesme, biraz daha fazla gecikme.

NAPI: kesme + yoklama

Yoğun ağ trafiğinde sürücü kesmeyi kapatıp kuyruğu *yoklar*, trafik azalınca kesmeye döner. Hibrit en iyisidir.

Saf yoklama ne zaman?

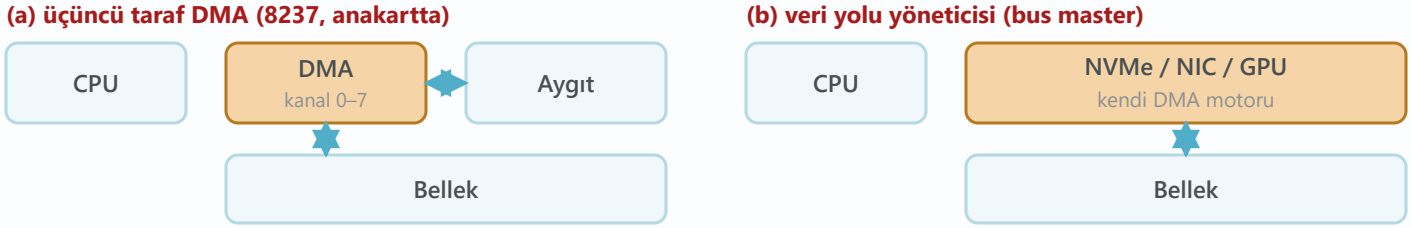
Olay hızı çok yüksek ve gecikme kritikse: DPDK/SPDK, NVMe yoklamalı kuyruklar, `io_uring` IOPOLL. Bir çekirdek tamamen yoklamaya ayrılır.

Kesme fırtınası

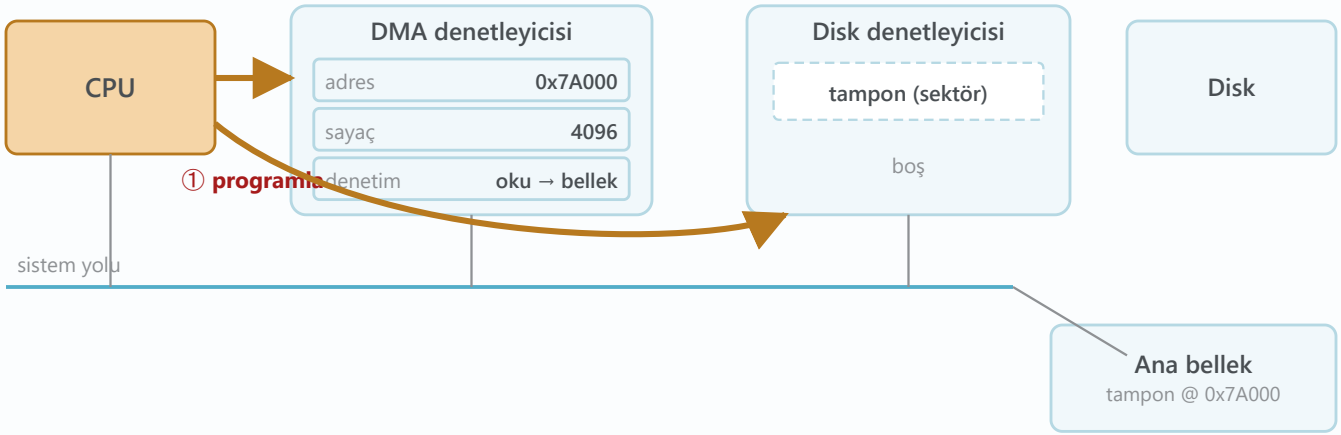
Hatalı aygıt sürekli kesme üretirse sistem kilitlenir; çekirdek "nobody cared" ile hattı kapatır, kesme hızını sınırlar.

Doğrudan Bellek Erişimi (DMA)

- Bayt bayt aktarımda CPU zamanı boşa gider; **DMA** denetleyicisi blok aktarımını CPU olmadan yapar. CPU yalnız kurulum ve bitiş kesmesiyle uğraşır.
- Yazmaçlar: **bellek adresi**, **bayt sayısı**, **denetim** (G/Ç kapısı, yön: okuma/yazma, birim: bayt/sözcük, patlama boyutu). Çok kanallı denetleyici aynı anda birçok aktarımı yönetir.
- Anakartta tek bir DMA denetleyicisi düzeni yalnız tarihsel 8237 için geçerlidir. Bugün aygıtların çoğunda **kendi DMA motoru** vardır ve belleğe doğrudan PCIe yazması yapar.



DMA Aktarımı Adım Adım

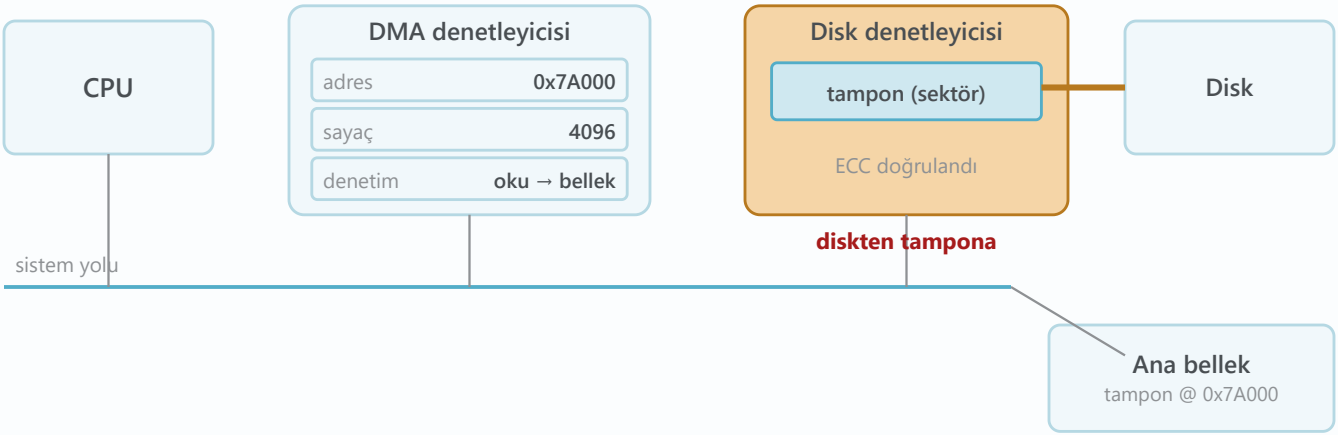


① CPU DMA denetleyicisini (adres = 0x7A000, sayaç = 4096, yön) ve disk denetleyicisini (sektörü oku) programlar, sonra başka işe döner.

Adım 1 / 6

Döngü çalma (cycle stealing): DMA her sözcük için yolu kısa süre alır, CPU belleğe erişirken biraz yavaşlar. **Patlama** (burst) modunda yol bir blok boyunca tutulur. Geçişsiz (fly-by) modda veri aygıttan belleğe doğrudan gider; aksi halde DMA önce okuyup sonra yazar.

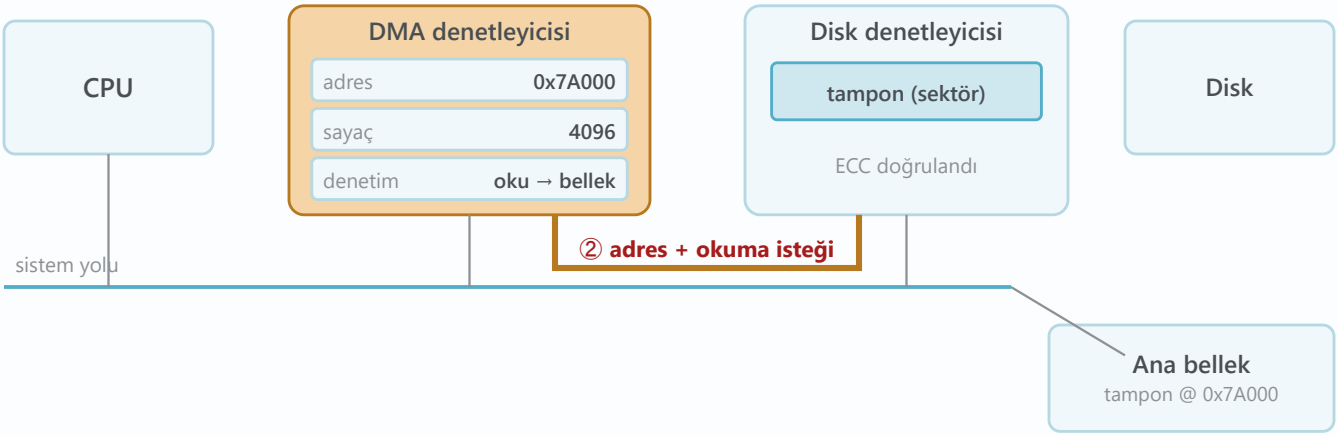
DMA Aktarımı Adım Adım



Disk denetleyicisi sektörü kendi tamponuna okur ve ECC ile doğrular. Tampon sayesinde yol meşgulken veri kaybolmaz.

Döngü çalma (cycle stealing): DMA her sözcük için yolu kısa süre alır, CPU belleğe erişirken biraz yavaşlar. **Patlama** (burst) modunda yol bir blok boyunca tutulur. Geçişsiz (fly-by) modda veri aygıttan belleğe doğrudan gider; aksi halde DMA önce okuyup sonra yazar.

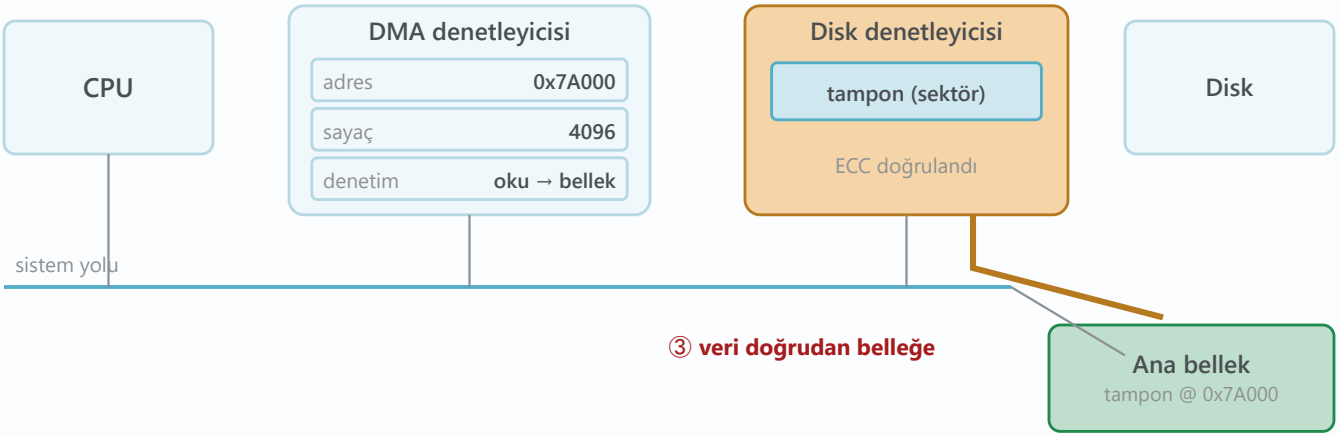
DMA Aktarımı Adım Adım



② DMA denetleyicisi yolu ister, bellek adresini yola koyar ve disk denetleyicisine okuma isteği gönderir.

Döngü çalma (cycle stealing): DMA her sözcük için yolu kısa süre alır, CPU belleğe erişirken biraz yavaşlar. **Patlama** (burst) modunda yol bir blok boyunca tutulur. Geçişsiz (fly-by) modda veri aygıttan belleğe doğrudan gider; aksi halde DMA önce okuyup sonra yazar.

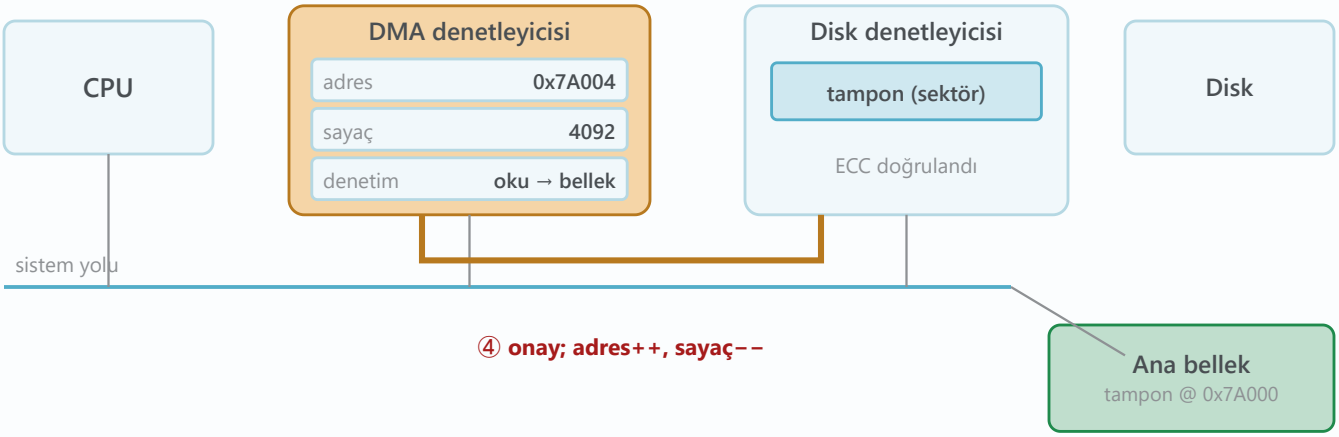
DMA Aktarımı Adım Adım



③ Disk denetleyicisi sözcüğü **doğrudan belleğe** yazar; CPU veri yolunda yer almaz.

Döngü çalma (cycle stealing): DMA her sözcük için yolu kısa süre alır, CPU belleğe erişirken biraz yavaşlar. **Patlama** (burst) modunda yol bir blok boyunca tutulur. Geçişsiz (fly-by) modda veri aygıttan belleğe doğrudan gider; aksi halde DMA önce okuyup sonra yazar.

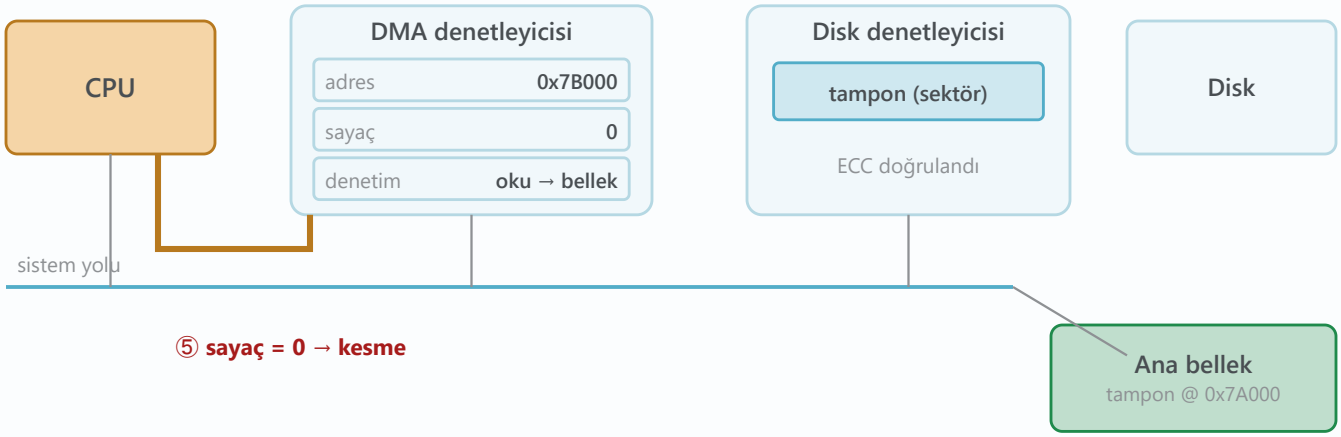
DMA Aktarımı Adım Adım



④ Disk denetleyicisi onay gönderir; DMA adresi artırır, sayacı azaltır. Sayaç 0 olana kadar ②–④ tekrarlanır.

Döngü çalma (cycle stealing): DMA her sözcük için yolu kısa süre alır, CPU belleğe erişirken biraz yavaşlar. **Patlama** (burst) modunda yol bir blok boyunca tutulur. Geçişsiz (fly-by) modda veri aygıttan belleğe doğrudan gider; aksi halde DMA önce okuyup sonra yazar.

DMA Aktarımı Adım Adım

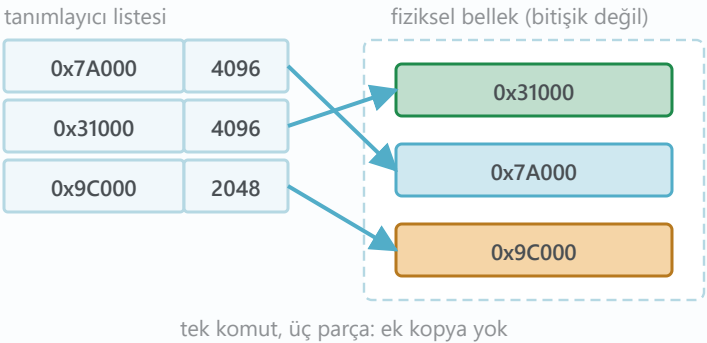


⑤ Sayaç 0: DMA denetleyicisi CPU'yu **kesmeyle** uyarır. İşletim sistemi açtığında veri zaten bellektedir.

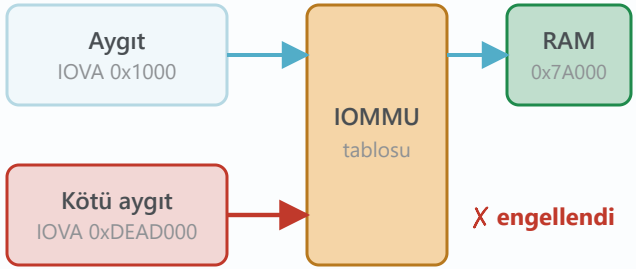
Döngü çalma (cycle stealing): DMA her sözcük için yolu kısa süre alır, CPU belleğe erişirken biraz yavaşlar. **Patlama** (burst) modunda yol bir blok boyunca tutulur. Geçişsiz (fly-by) modda veri aygıttan belleğe doğrudan gider; aksi halde DMA önce okuyup sonra yazar.

Dağıt-Topla ve IOMMU

Dağıt-topla (scatter-gather)



IOMMU (Intel VT-d, AMD-Vi, Arm SMMU)



Dağıt-topla

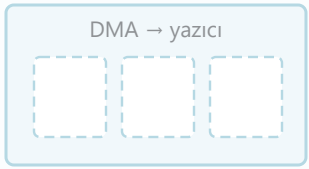
Aygıta (adres, uzunluk) çiftleri listesi verilir; aygıt bitişik olmayan sayfalara tek işlemde yazar (`readv`, ağ paketleri, NVMe PRP/SGL).

IOMMU

Aygıtın gördüğü adresi (IOVA) sayfa tablosuyla fiziksele çevirir: *koruma* (Thunderbolt DMA saldırıları), sanal makineye aygıt atama (VFIO), 32 bit aygıtlara yüksek bellek.

DMA ile G/Ç: Yazıcı Örneği

```
1 /* sistem çağrısı */
▶ copy_from_user(buffer, p, count);
3 set_up_DMA_controller();
4 scheduler();     /* süreç bloke */
5
6 /* kesme hizmet yordamı */
7 acknowledge_interrupt();
8 unblock_user();
9 return_from_interrupt();
```



süreç: çalışıyor

CPU: süreç

Veri çekirdek tamponuna kopyalanır.

Adım 1 / 6

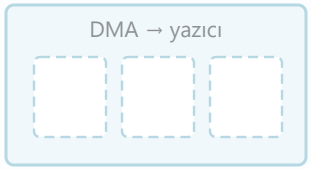
DEĞİŞKENLER

`DMA sayaç = –` `kesme = 0`

DMA, CPU'yu G/Ç'den kurtarır, ama DMA denetleyicisi genellikle CPU'dan yavaştır. Aygıt çok yavaşsa ya da CPU'nun yapacak başka işi yoksa kesme güdümlü G/Ç ya da PIO daha basit ve yeterli olabilir (Tanenbaum).

DMA ile G/Ç: Yazıcı Örneği

```
1 /* sistem çağrısı */
2 copy_from_user(buffer, p, count);
▶ set_up_DMA_controller();
4 scheduler();      /* süreç bloke */
5
6 /* kesme hizmet yordamı */
7 acknowledge_interrupt();
8 unblock_user();
9 return_from_interrupt();
```



süreç: çalışıyor

CPU: süreç

DMA denetleyicisi programlanır: tampon adresi, sayaç = 3, hedef yazıcı kapısı.

Adım 2 / 6

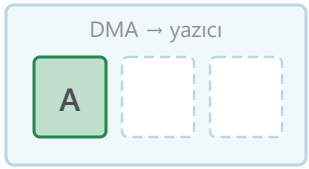
DEĞİŞKENLER

`DMA sayaç = 3` `kesme = 0`

DMA, CPU'yu G/Ç'den kurtarır, ama DMA denetleyicisi genellikle CPU'dan yavaştır. Aygıt çok yavaşsa ya da CPU'nun yapacak başka işi yoksa kesme güdümlü G/Ç ya da PIO daha basit ve yeterli olabilir (Tanenbaum).

DMA ile G/Ç: Yazıcı Örneği

```
1 /* sistem çağrısı */
2 copy_from_user(buffer, p, count);
3 set_up_DMA_controller();
▶ scheduler();     /* süreç bloke */
5
6 /* kesme hizmet yordamı */
7 acknowledge_interrupt();
8 unblock_user();
9 return_from_interrupt();
```



Süreç bloke edilir; CPU başka iş yapar. DMA karakterleri **CPU'suz** yazıcıya iletir.

Adım 3 / 6

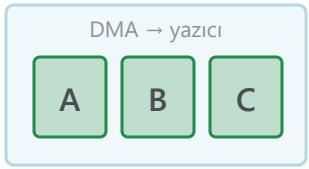
DEĞİŞKENLER

`DMA sayaç = 2` `kesme = 0`

DMA, CPU'yu G/Ç'den kurtarır, ama DMA denetleyicisi genellikle CPU'dan yavaştır. Aygıt çok yavaşsa ya da CPU'nun yapacak başka işi yoksa kesme güdümlü G/Ç ya da PIO daha basit ve yeterli olabilir (Tanenbaum).

DMA ile G/Ç: Yazıcı Örneği

```
1 /* sistem çağrısı */
2 copy_from_user(buffer, p, count);
3 set_up_DMA_controller();
▶ scheduler();     /* süreç bloke */
5
6 /* kesme hizmet yordamı */
7 acknowledge_interrupt();
8 unblock_user();
9 return_from_interrupt();
```



Tüm karakterler aktarıldı; bu arada CPU'ya hiç kesme gelmedi.

Adım 4 / 6

DEĞİŞKENLER

`DMA sayaç = 0` `kesme = 0`

DMA, CPU'yu G/Ç'den kurtarır, ama DMA denetleyicisi genellikle CPU'dan yavaştır. Aygıt çok yavaşsa ya da CPU'nun yapacak başka işi yoksa kesme güdümlü G/Ç ya da PIO daha basit ve yeterli olabilir (Tanenbaum).

DMA ile G/Ç: Yazıcı Örneği

```
1 /* sistem çağrısı */
2 copy_from_user(buffer, p, count);
3 set_up_DMA_controller();
4 scheduler();      /* süreç bloke */
5
6 /* kesme hizmet yordamı */
▶ acknowledge_interrupt();
8 unblock_user();
9 return_from_interrupt();
```



DMA tek bir kesme üretir: işleyici onaylar.

Adım 5 / 6

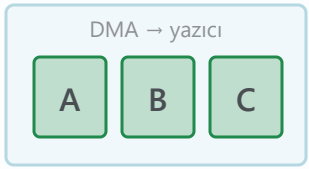
DEĞİŞKENLER

`DMA sayaç = 0` `kesme = 1`

DMA, CPU'yu G/Ç'den kurtarır, ama DMA denetleyicisi genellikle CPU'dan yavaştır. Aygıt çok yavaşsa ya da CPU'nun yapacak başka işi yoksa kesme güdümlü G/Ç ya da PIO daha basit ve yeterli olabilir (Tanenbaum).

DMA ile G/Ç: Yazıcı Örneği

```
1 /* sistem çağrısı */
2 copy_from_user(buffer, p, count);
3 set_up_DMA_controller();
4 scheduler();     /* süreç bloke */
5
6 /* kesme hizmet yordamı */
7 acknowledge_interrupt();
  unblock_user();
  return_from_interrupt();
```



Kullanıcı süreci uyandırılır ve kesmeden dönülür. Karakter başına değil, **tampon başına bir kesme**.

Adım 6 / 6

DEĞİŞKENLER

`DMA sayaç = 0` `kesme = 1`

DMA, CPU'yu G/Ç'den kurtarır, ama DMA denetleyicisi genellikle CPU'dan yavaştır. Aygıt çok yavaşsa ya da CPU'nun yapacak başka işi yoksa kesme güdümlü G/Ç ya da PIO daha basit ve yeterli olabilir (Tanenbaum).

Üç Yöntemin Maliyeti

Yazıcı saniyede **1000 karakter** basıyor; 4096 karakter yazdırılacak. Kesme işleme maliyeti **2 µs**, DMA kurulumu **5 µs**.

Yoklama: CPU her karakteri bekler: 4096 × 1 ms = **4,096 s** boyunca meşgul.

Programlanmış G/Ç'de CPU, aygıt hızına bağlıdır.

Etkileşimli simülatör: Kesme, DMA ve G/Ç Akışı Simülatörü: üç yöntem aynı aktarımla çalıştırılır, CPU kullanımları karşılaştırılır.

Üç Yöntemin Maliyeti

Yazıcı saniyede **1000 karakter** basıyor; 4096 karakter yazdırılacak. Kesme işleme maliyeti **2 µs**, DMA kurulumu **5 µs**.

Yoklama: CPU her karakteri bekler: 4096 × 1 ms = **4,096 s** boyunca meşgul.

Kesme güdümlü: 4096 kesme × 2 µs = **8,19 ms** (işin süresinin %0,2'si).

Bekleme ortadan kalkar; bedel kesme sayısıdır.

Etkileşimli simülatör: Kesme, DMA ve G/Ç Akışı Simülatörü: üç yöntem aynı aktarımla çalıştırılır, CPU kullanımları karşılaştırılır.

Üç Yöntemin Maliyeti

Yazıcı saniyede **1000 karakter** basıyor; 4096 karakter yazdırılacak. Kesme işleme maliyeti **2 µs**, DMA kurulumu **5 µs**.

Yoklama: CPU her karakteri bekler: 4096 × 1 ms = **4,096 s** boyunca meşgul.

Kesme güdümlü: 4096 kesme × 2 µs = **8,19 ms** (işin süresinin %0,2'si).

DMA: 5 µs kurulum + 1 kesme × 2 µs = **≈ 7 µs**; yaklaşık 1170 kat daha az.

Ama DMA yol döngüsü çalar ve kurulumu küçük aktarımlarda kendini geri ödemez.

Etkileşimli simülatör: Kesme, DMA ve G/Ç Akışı Simülatörü: üç yöntem aynı aktarımla çalıştırılır, CPU kullanımları karşılaştırılır.

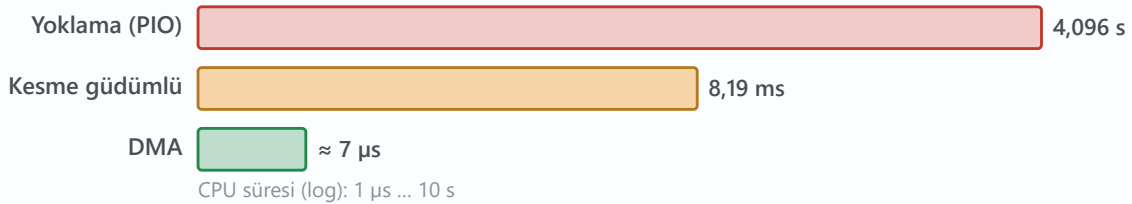
Üç Yöntemin Maliyeti

Yazıcı saniyede **1000 karakter** basıyor; 4096 karakter yazdırılacak. Kesme işleme maliyeti **2 µs**, DMA kurulumu **5 µs**.

Yoklama: CPU her karakteri bekler: 4096 × 1 ms = **4,096 s** boyunca meşgul.

Kesme güdümlü: 4096 kesme × 2 µs = **8,19 ms** (işin süresinin %0,2'si).

DMA: 5 µs kurulum + 1 kesme × 2 µs = **≈ 7 µs**; yaklaşık 1170 kat daha az.



Hız farkı 4 basamaktan fazla. Kısa aktarım, yavaş DMA ya da boşta bekleyen CPU varsa DMA'nın kazancı kaybolur.

Etkileşimli simülatör: Kesme, DMA ve G/Ç Akışı Simülatörü: üç yöntem aynı aktarımla çalıştırılır, CPU kullanımları karşılaştırılır.

G/Ç Yazılımının Hedefleri

Aygıt bağımsızlığı
Program aygıtı bilmeden çalışmalı: `sort < girdi` dosya, SSD ya da USB bellekte aynı çalışır.

Tek tip adlandırma
Ad aygıt tipine bağlı olmamalı: `/dev/nvme0n1`, `/mnt/usb/rapor.txt`; bağlama (mount) ile tek ağaç.

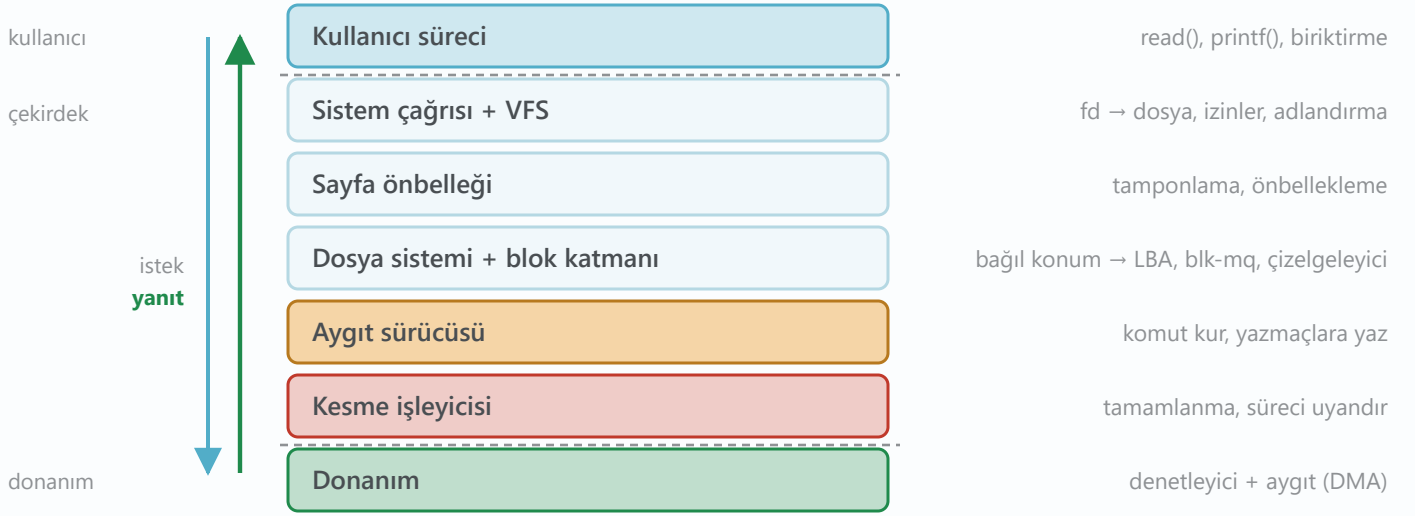
Hata ele alma
Hata donanıma olabildiğince yakın çözülmeli: denetleyici ECC ile düzeltir, sürücü yeniden dener; üst katman ancak gerekirse bilir.

Eş zamanlı / eş zamansız
Donanımda aktarım çoğunlukla eş zamansızdır (kesme güdümlü); kullanıcıya ise *bloke eden* arayüz sunmak programlamayı kolaylaştırır.

Tamponlama
Gelen paket nereye gideceği belli olmadan bir yere konmalı; ses gibi gerçek zamanlı aygıtlar da tampona dayanır.

Paylaşılan / adanmış
Disk birçok kullanıcıya aynı anda hizmet verir; yazıcı ya da tarayıcı tek kullanıcıya *adanır* (kilitlenme riski doğar).

G/Ç Yazılım Katmanları



Her katmanın iyi tanımlı bir görevi ve üst katmana arayüzü vardır. Tanenbaum dört katman sayar: kullanıcı düzeyi G/Ç yazılımı, aygıttan bağımsız yazılım, aygıt sürücüleri, kesme işleyicileri. Linux'ta "aygıttan bağımsız" kısım VFS, sayfa önbelleği, dosya sistemi ve blok katmanına ayrılır.

Bir read() Çağrısının Yolu



n = read(fd, buf, 4096)

Süreç `read(fd, buf, 4096)` çağırır; libc sarmalayıcısı `syscall` komutuyla çekirdeğe geçer.

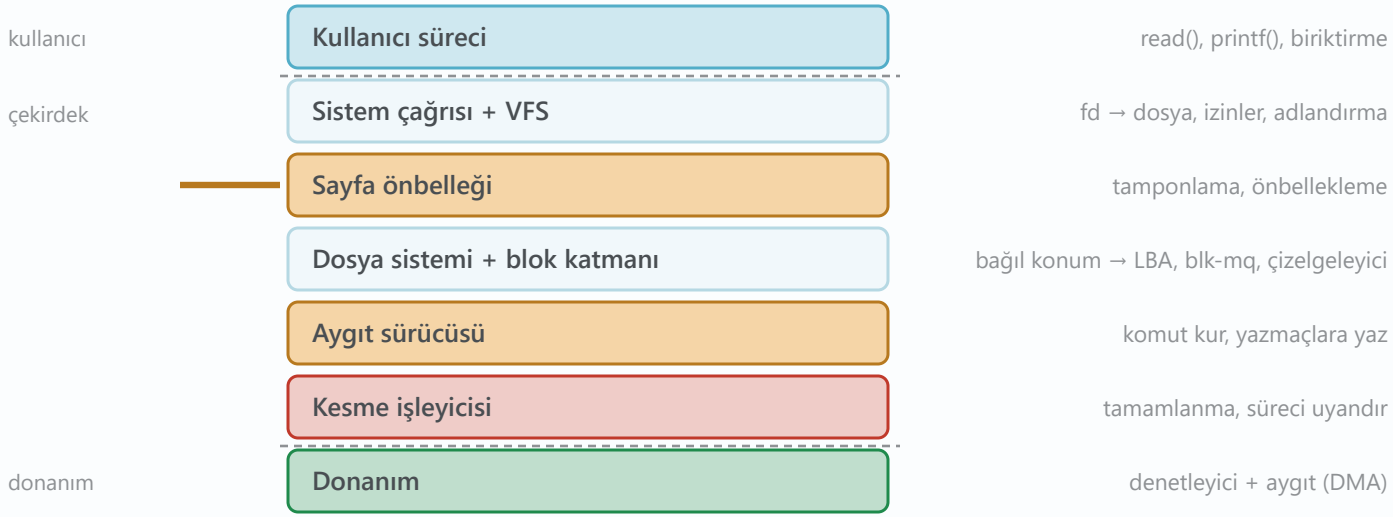
Bir read() Çağrısının Yolu



fd → struct file → inode

VFS tanımlayıcıyı açık dosyaya, dosyayı inode'a çevirir; erişim izni açılışta zaten denetlenmiştir.

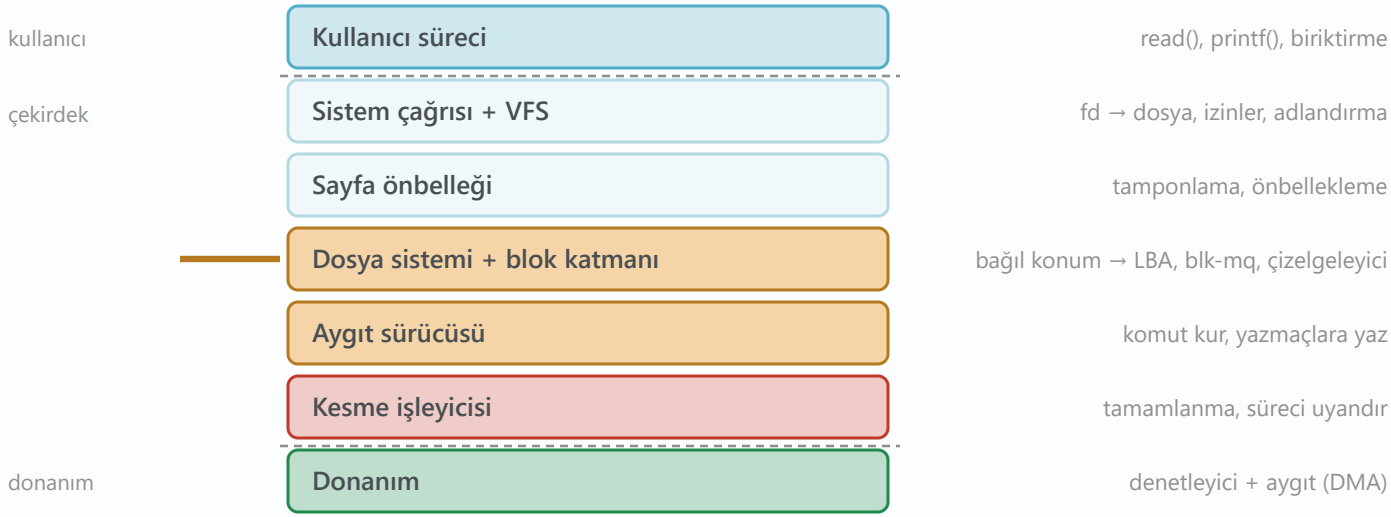
Bir read() Çağrısının Yolu



önbellekte yok: ıska

Sayfa önbelleğine bakılır: istenen sayfa **yok**. Önbellek sayfası ayrılır, okuma başlatılır.

Bir read() Çağrısının Yolu



bağıl konum 0 → LBA 2048

Dosya sistemi dosya bağıl konumunu disk bloğuna (LBA) çevirir; blok katmanı bir `bio` /istek oluşturup donanım kuyruğuna verir.

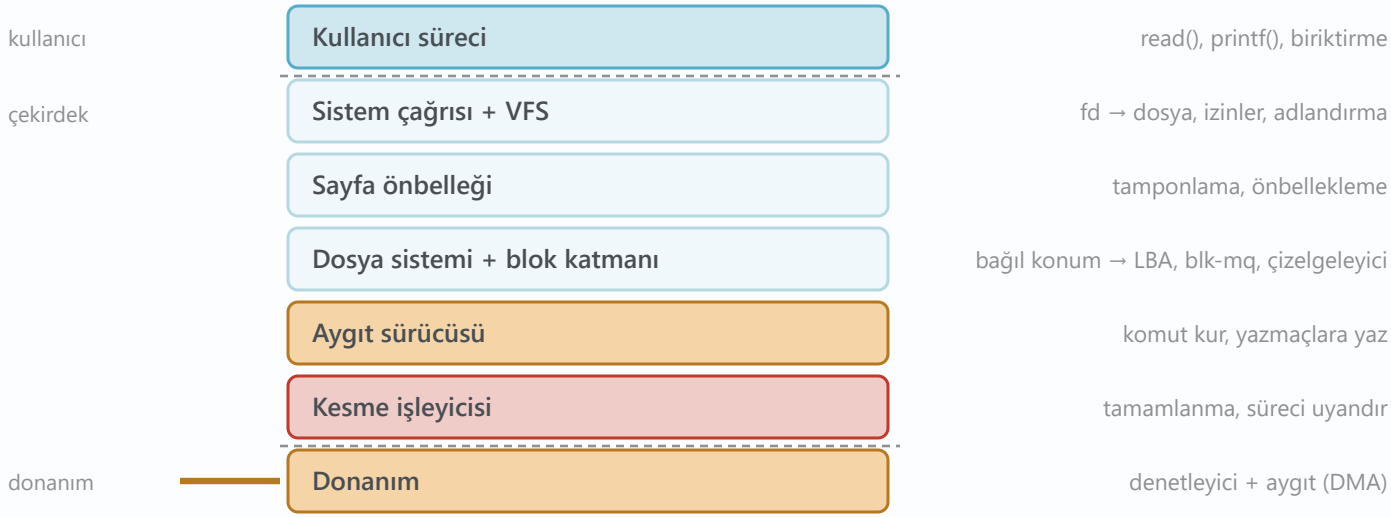
Bir read() Çağrısının Yolu



SQ'ya komut + kapı zili

NVMe sürücüsü okuma komutunu gönderim kuyruğuna yazar, kapı zili (doorbell) yazmacına yazar. Süreç **bloke** olur (D durumu).

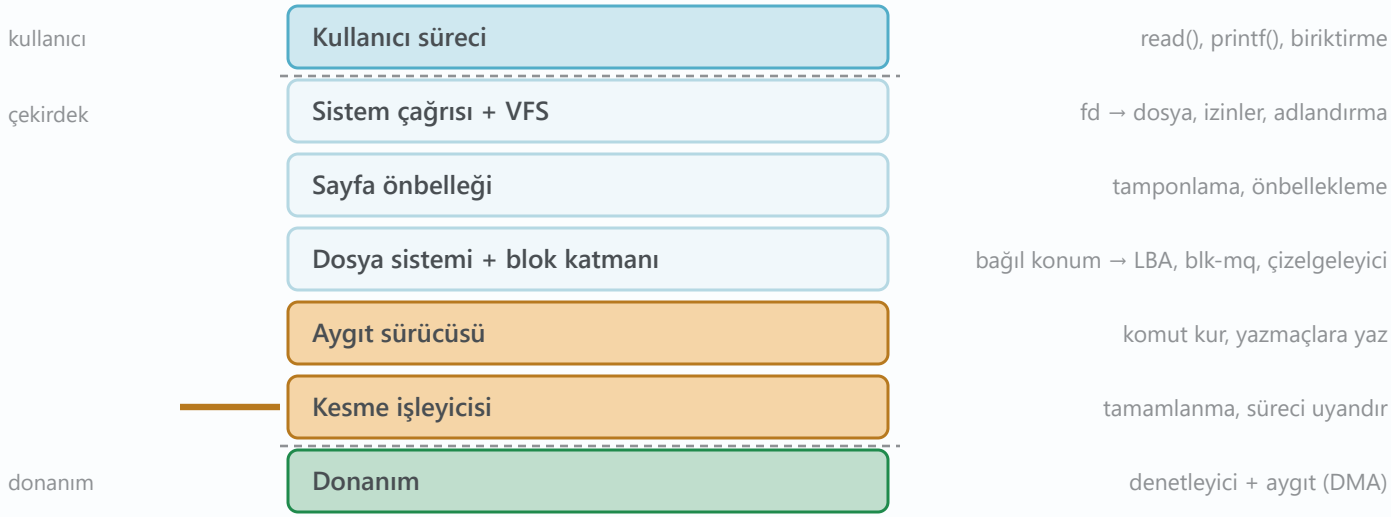
Bir read() Çağrısının Yolu



DMA → önbellek sayfası, MSI-X

SSD veriyi DMA ile önbellek sayfasına yazar, tamamlanma kuyruğuna kayıt ekler ve MSI-X kesmesi üretir.

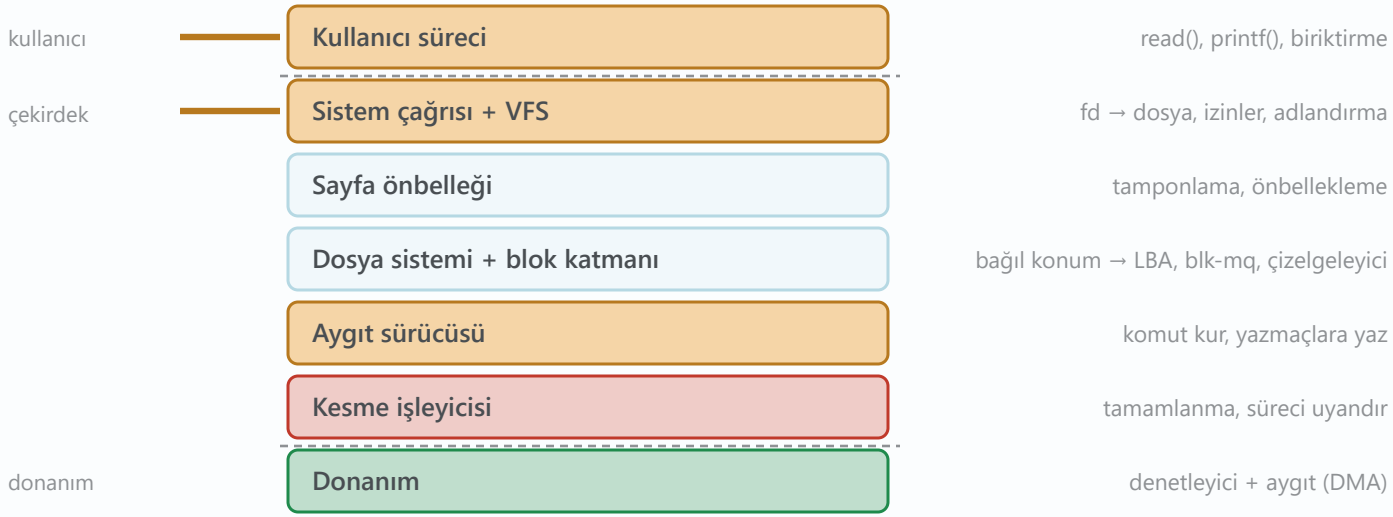
Bir read() Çağrısının Yolu



tamamlandı: süreci uyandır

Kesme işleyicisi tamamlanmayı işler: sayfa “güncel” işaretlenir, bekleyen süreç **hazır** olur.

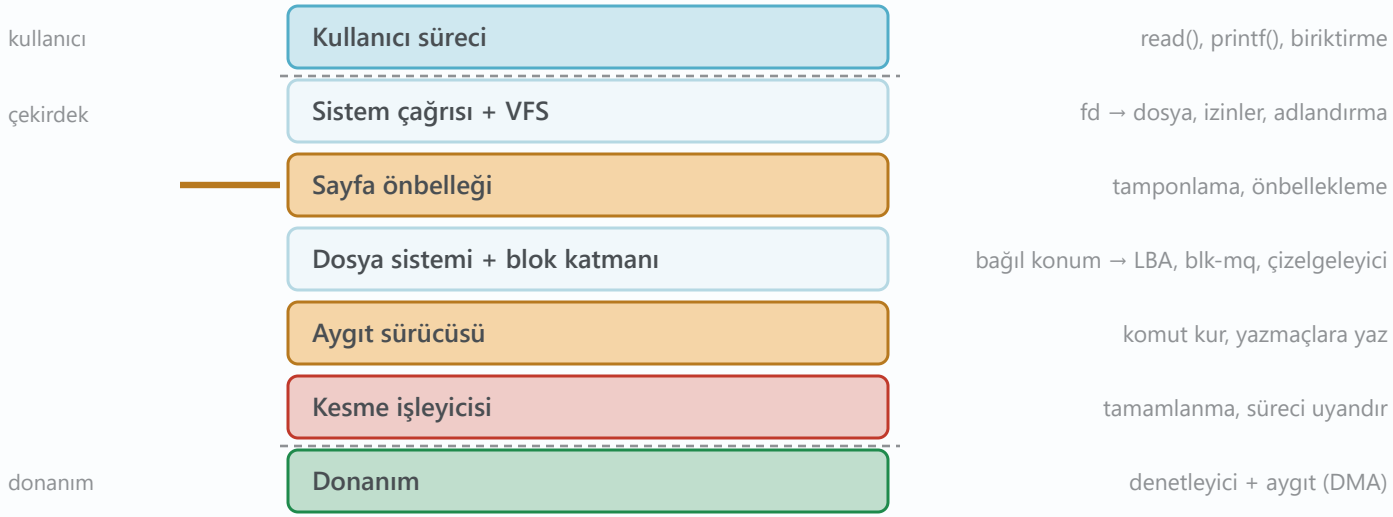
Bir read() Çağrısının Yolu



copy_to_user → n = 4096

Süreç çalışınca VFS veriyi önbellekten kullanıcı tamponuna kopyalar ve 4096 döndürür.

Bir read() Çağrısının Yolu

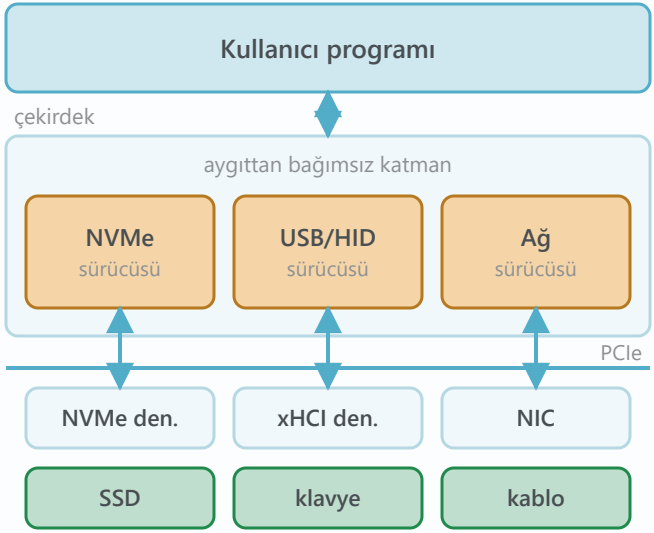


ikinci okuma: isabet

Aynı veri yeniden okunursa sayfa önbelleğinde **isabet**: aygıta hiç gidilmez, işlem mikrosaniyeler sürer.

Aygıt Sürücüleri

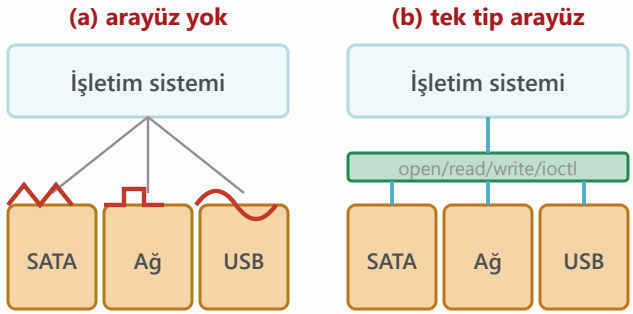
- **Sürücü** aygıta özgü kodu kapsüller: yazmaçları, komutları ve tuhaflıklarını yalnız o bilir. Linux'ta çoğu sürücü çekirdek ağacındadır (*in-tree*); bazıları üreticiden gelir.
- Görevleri: parametreleri doğrula, isteği aygıt komutuna çevir (LBA, kuyruk girdisi), gerekirse aygıtı başlat, yazmaçlara yaz, kesmeyi bekle (bloke), hatayı denetle, sonucu döndür.
- Yeniden girilebilir (reentrant) olmalı: işlerken yeni istek ya da kesme gelebilir. Sıcak tak (hot-plug) ile aygıt çalışırken kaybolabilir.
- Çekirdekteki hatalı sürücü tüm sistemi çökertir → **kullanıcı alanı sürücüleri**: MINIX 3, FUSE, VFIO/UIO, DPDK/SPDK; Windows UMDF. Yüklenebilir modüller: `modprobe`, `lsmod`.



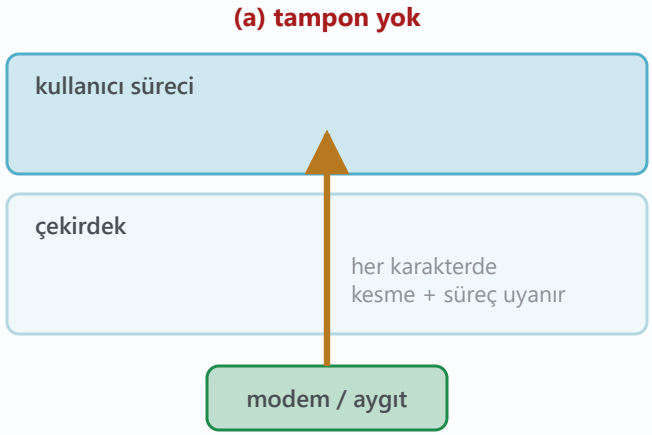
Aygıttan Bağımsız G/Ç Yazılımı

- Tüm sürücüler için **tek tip arayüz**: her sınıf aynı işlevleri sunar; çekirdek bir *fonksiyon işaretçisi tablosu* üzerinden çağırır.
- **Adlandırma**: `/dev/nvme0n1` → ana numara (sürücü) + yan numara (birim). **Koruma**: aygıt dosyasının izinleri.
- Aygıttan bağımsız **blok boyutu**, tamponlama, hata raporlama, adanmış aygıtların ayrılması ve bırakılması bu katmandadır.

```
struct file_operations nvme_fops = {
  .open           = nvme_open,
  .release        = nvme_release,
  .read_iter      = nvme_read,
  .unlocked_ioctl = nvme_ioctl,
  .poll           = nvme_poll,
};
```



Ara Belleğe Alma (Tamponlama)



Tamponsuz: her karakter için kesme gelir ve süreç uyandırılır. Çok sayıda bağlam değiştirme, verimsiz.

Adım 1 / 5

Tamponlamanın üç nedeni (Silberschatz): üretici ile tüketici arasındaki **hız farkı**, **aktarım boyutu farkı** (paketlerin birleştirilmesi) ve **kopya anlamı** (`write` döndükten sonra kullanıcı tamponunu değiştirebilmek).

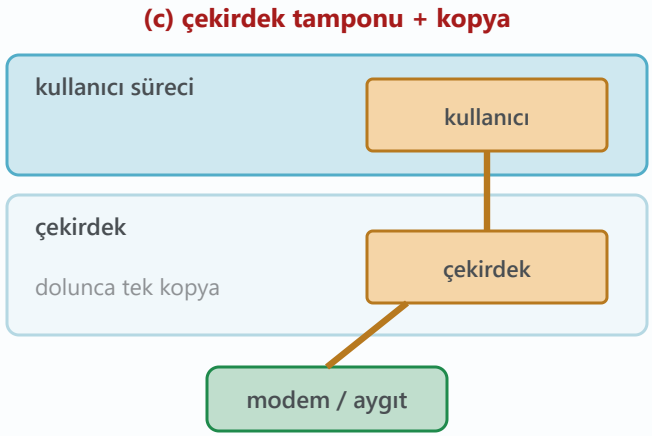
Ara Belleğe Alma (Tamponlama)



Kullanıcı tamponu: kesme işleyicisi karakterleri doğrudan süreç tamponuna yazar. Tampon sayfası diske atılırsa? Sayfa kilitlenmeli; çok süreç kilitlerse bellek azalır.

Tamponlamanın üç nedeni (Silberschatz): üretici ile tüketici arasındaki **hız farkı**, **aktarım boyutu farkı** (paketlerin birleştirilmesi) ve **kopya anlamı** (`write` döndükten sonra kullanıcı tamponunu değiştirebilmek).

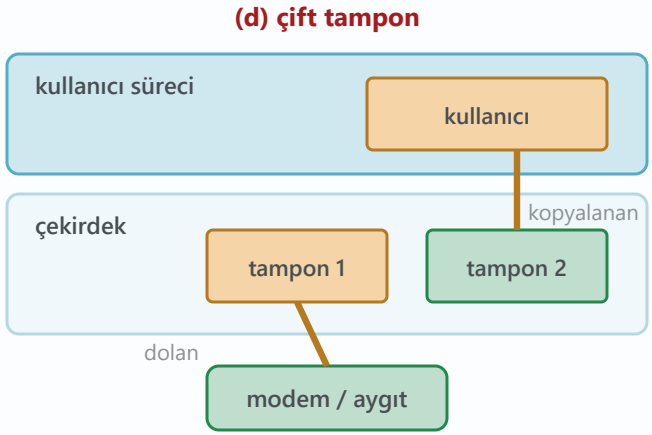
Ara Belleğe Alma (Tamponlama)



Çekirdek tamponu: karakterler önce çekirdekte birikir, dolunca tek seferde kullanıcıya kopyalanır. Ama kopyalanırken gelen veri nereye konacak?

Tamponlamanın üç nedeni (Silberschatz): üretici ile tüketici arasındaki **hız farkı**, **aktarım boyutu farkı** (paketlerin birleştirilmesi) ve **kopya anlamı** (`write` döndükten sonra kullanıcı tamponunu değiştirebilmek).

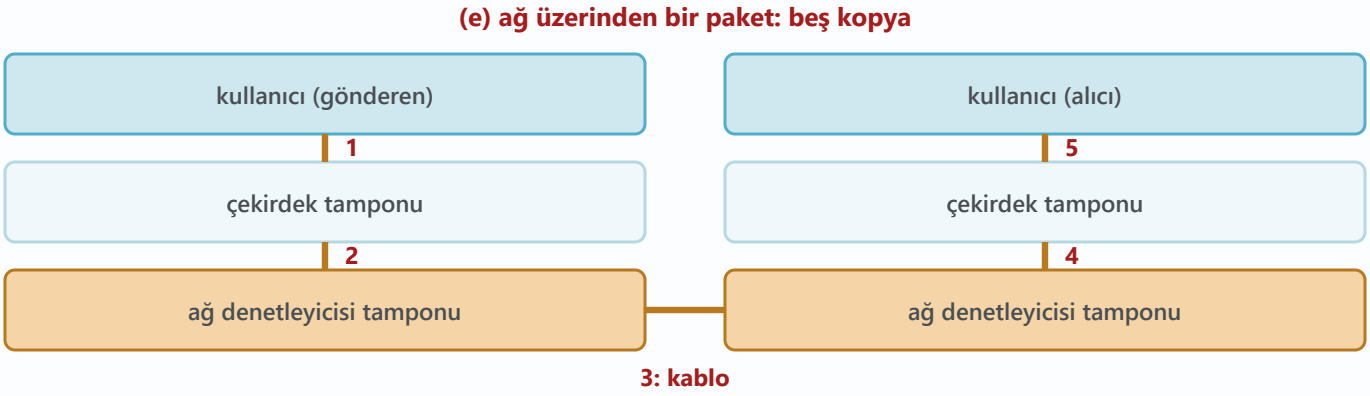
Ara Belleğe Alma (Tamponlama)



Çift tampon: biri dolarken diğeri kopyalanır; roller sırayla değişir. Daha genel biçimi **dairesel tampondur**.

Tamponlamanın üç nedeni (Silberschatz): üretici ile tüketici arasındaki **hız farkı**, **aktarım boyutu farkı** (paketlerin birleştirilmesi) ve **kopya anlamı** (`write` döndükten sonra kullanıcı tamponunu değiştirebilmek).

Ara Belleğe Alma (Tamponlama)

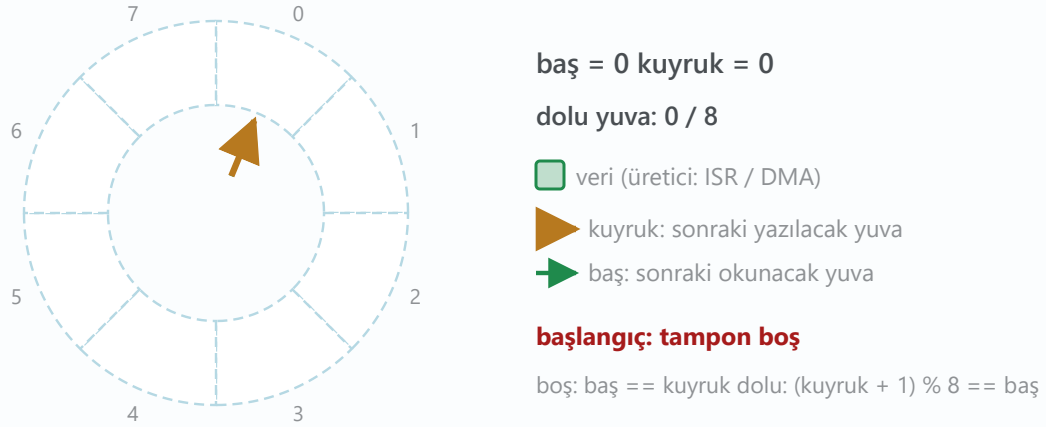


sıfır kopya: sendfile, splice, MSG_ZEROCOPY, io_uring

Ağda bir paket gönderilirken 5 kez kopyalanabilir; her kopya bellek bant genişliği ve gecikme demektir. Bu yüzden sıfır kopya teknikleri geliştirilmiştir.

Tamponlamanın üç nedeni (Silberschatz): üretici ile tüketici arasındaki **hız farkı**, **aktarım boyutu farkı** (paketlerin birleştirilmesi) ve **kopya anlamı** (`write` döndükten sonra kullanıcı tamponunu değiştirebilmek).

Dairesel Tampon

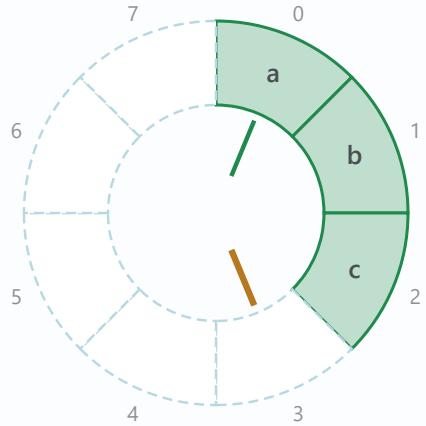


Baş ve kuyruk aynı yuvayı gösteriyor: tampon **boş**.

Adım 1 / 5

Dairesel tamponlar her yerdedir: klavye ve seri port sürücüleri, NIC alma/gönderme halkaları, NVMe gönderim/tamamlanma kuyrukları, `io_uring`. Tek üretici–tek tüketicide kilit gerekmez; yalnız bellek bariyerleri gerekir.

Dairesel Tampon



baş = 0 kuyruk = 3

dolu yuva: 3 / 8

veri (üretici: ISR / DMA)

kuyruk: sonraki yazılacak yuva

baş: sonraki okunacak yuva

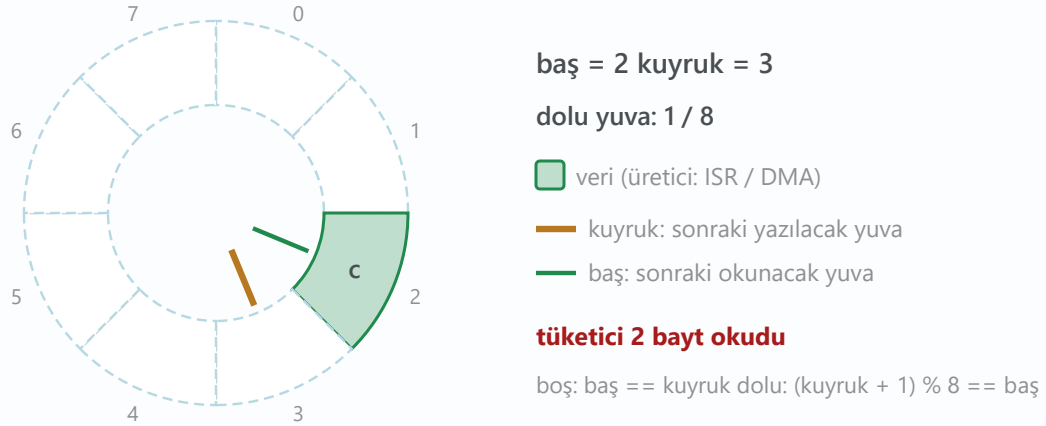
üretici 3 bayt yazdı

boş: baş == kuyruk dolu: (kuyruk + 1) % 8 == baş

Kesme işleyicisi gelen 3 baytı kuyruğa yazar; kuyruk ilerler. Tüketici henüz okumadı.

Dairesel tamponlar her yerdedir: klavye ve seri port sürücüleri, NIC alma/gönderme halkaları, NVMe gönderim/tamamlanma kuyrukları, `io_uring`. Tek üretici–tek tüketicide kilit gerekmez; yalnız bellek bariyerleri gerekir.

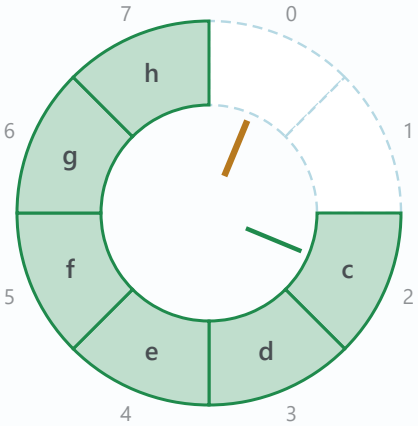
Dairesel Tampon



Süreç 2 bayt okur; baş ilerler. Okunan yuvalar yeniden kullanılabilir.

Dairesel tamponlar her yerdedir: klavye ve seri port sürücüleri, NIC alma/gönderme halkaları, NVMe gönderim/tamamlanma kuyrukları, `io_uring`. Tek üretici–tek tüketicide kilit gerekmez; yalnız bellek bariyerleri gerekir.

Dairesel Tampon



baş = 2 kuyruk = 0

dolu yuva: 6 / 8

veri (üretici: ISR / DMA)

kuyruk: sonraki yazılacak yuva

baş: sonraki okunacak yuva

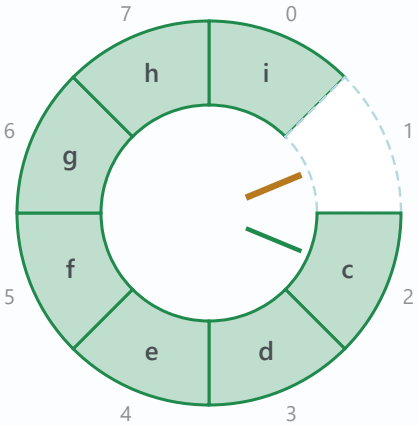
sarma: kuyruk 7 → 0

boş: baş == kuyruk dolu: (kuyruk + 1) % 8 == baş

Üretici 5 bayt daha yazar; kuyruk sona gelince **başa sarar** (mod 8): kuyruk = 0.

Dairesel tamponlar her yerdedir: klavye ve seri port sürücüleri, NIC alma/gönderme halkaları, NVMe gönderim/tamamlanma kuyrukları, `io_uring`. Tek üretici–tek tüketicide kilit gerekmez; yalnız bellek bariyerleri gerekir.

Dairesel Tampon



baş = 2 kuyruk = 1

dolu yuva: 7 / 8

veri (üretici: ISR / DMA)

kuyruk: sonraki yazılacak yuva

baş: sonraki okunacak yuva

bir bayt daha: tampon dolu

boş: baş == kuyruk dolu: (kuyruk + 1) % 8 == baş

'i' yazılınca kuyruk = 1 ve (kuyruk + 1) mod 8 == baş: tampon **dolu**. Bir yuva boş bırakılarak boş/dolu ayırt edilir. Dolu tamponda üretici bekler ya da veriyi düşürür (ağ paketi kaybı).

Dairesel tamponlar her yerdedir: klavye ve seri port sürücüleri, NIC alma/gönderme halkaları, NVMe gönderim/tamamlanma kuyrukları, `io_uring`. Tek üretici–tek tüketicide kilit gerekmez; yalnız bellek bariyerleri gerekir.

Dairesel Tampon: Kod İzleme

```
▶ int koy(char c) {          /* üretici */
2   int s = (kuy + 1) % N;
3   if (s == bas) return -1; /* dolu */
4   buf[kuy] = c;
5   kuy = s;
6   return 0;
7 }
8 int al(char *c) {          /* tüketici */
9   if (bas == kuy) return -1; /* boş */
10  *c = buf[bas];
11  bas = (bas + 1) % N;
12  return 0;
13 }
```



Tampon boş: bas = kuy = 0. Kesme işleyicisi `koy` ile yazar, süreç `al` ile okur; her biri yalnızca kendi indisini değiştirir.

Adım 1 / 9

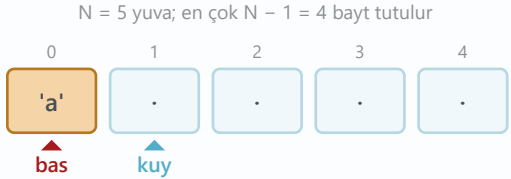
DEĞİŞKENLER

`işlem = –` `bas = 0` `kuy = 0` `dolu = 0`

ÇIKTI

Dairesel Tampon: Kod İzleme

```
1  int koy(char c) {          /* üretici */
2    int s = (kuy + 1) % N;
3    if (s == bas) return -1; /* dolu */
4    buf[kuy] = c;
5    kuy = s;
6    return 0;
7  }
8  int al(char *c) {          /* tüketici */
9    if (bas == kuy) return -1; /* boş */
10   *c = buf[bas];
11   bas = (bas + 1) % N;
12   return 0;
13 }
```



`koy('a')` : buf[0] = 'a', kuy = 1.

Adım 2 / 9

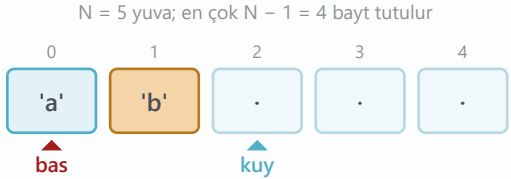
DEĞİŞKENLER

işlem = **koy('a')** | bas = **0** | kuy = **1** | dolu = **1**

ÇIKTI

Dairesel Tampon: Kod İzleme

```
1  int koy(char c) {          /* üretici */
2    int s = (kuy + 1) % N;
3    if (s == bas) return -1; /* dolu */
4    buf[kuy] = c;
5    kuy = s;
6    return 0;
7  }
8  int al(char *c) {          /* tüketici */
9    if (bas == kuy) return -1; /* boş */
10   *c = buf[bas];
11   bas = (bas + 1) % N;
12   return 0;
13 }
```



`koy('b')` : buf[1] = 'b', kuy = 2.

Adım 3 / 9

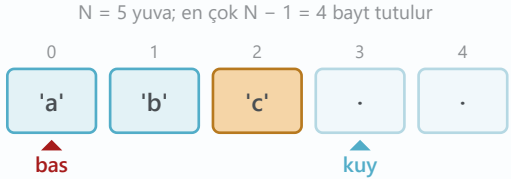
DEĞİŞKENLER

işlem = **koy('b')** | bas = **0** | kuy = **2** | dolu = **2**

ÇIKTI

Dairesel Tampon: Kod İzleme

```
1  int koy(char c) {          /* üretici */
2    int s = (kuy + 1) % N;
3    if (s == bas) return -1; /* dolu */
4    buf[kuy] = c;
5    kuy = s;
6    return 0;
7  }
8  int al(char *c) {          /* tüketici */
9    if (bas == kuy) return -1; /* boş */
10   *c = buf[bas];
11   bas = (bas + 1) % N;
12   return 0;
13 }
```



`koy('c')` : buf[2] = 'c', kuy = 3.

Adım 4 / 9

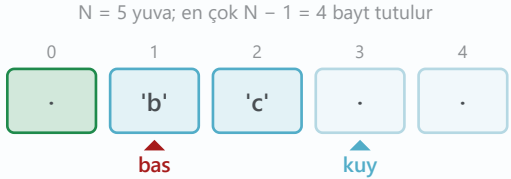
DEĞİŞKENLER

işlem = **koy('c')** | bas = **0** | kuy = **3** | dolu = **3**

ÇIKTI

Dairesel Tampon: Kod İzleme

```
1  int koy(char c) {          /* üretici */
2    int s = (kuy + 1) % N;
3    if (s == bas) return -1; /* dolu */
4    buf[kuy] = c;
5    kuy = s;
6    return 0;
7  }
8  int al(char *c) {          /* tüketici */
▶    if (bas == kuy) return -1; /* boş */
▶    *c = buf[bas];
▶    bas = (bas + 1) % N;
12   return 0;
13 }
```



`al` : bas ≠ kuy, tampon boş değil. buf[0] = 'a' okunur, bas = 1. Yuva 0 yeniden kullanılabilir.

Adım 5 / 9

DEĞİŞKENLER

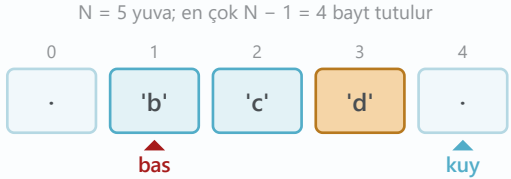
`işlem = al` `bas = 1` `kuy = 3` `dolu = 2`

ÇIKTI

```
al: a
```

Dairesel Tampon: Kod İzleme

```
1  int koy(char c) {          /* üretici */
2    int s = (kuy + 1) % N;
3    if (s == bas) return -1; /* dolu */
4    buf[kuy] = c;
5    kuy = s;
6    return 0;
7  }
8  int al(char *c) {          /* tüketici */
9    if (bas == kuy) return -1; /* boş */
10   *c = buf[bas];
11   bas = (bas + 1) % N;
12   return 0;
13 }
```



`koy('d')` : buf[3] = 'd', kuy = 4.

Adım 6 / 9

DEĞİŞKENLER

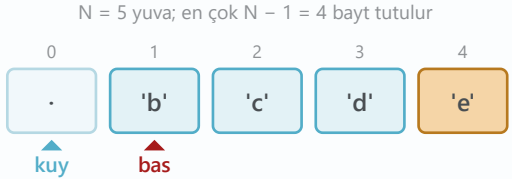
işlem = **koy('d')** | bas = **1** | kuy = **4** | dolu = **3**

ÇIKTI

```
al: a
```

Dairesel Tampon: Kod İzleme

```
1  int koy(char c) {          /* üretici */
2    int s = (kuy + 1) % N;
3    if (s == bas) return -1; /* dolu */
4    buf[kuy] = c;
5    kuy = s;
6    return 0;
7  }
8  int al(char *c) {          /* tüketici */
9    if (bas == kuy) return -1; /* boş */
10   *c = buf[bas];
11   bas = (bas + 1) % N;
12   return 0;
13 }
```



`koy('e')` : buf[4] = 'e', kuy = 0. kuy sona gelip **başa sardı** (mod 5).

Adım 7 / 9

DEĞİŞKENLER

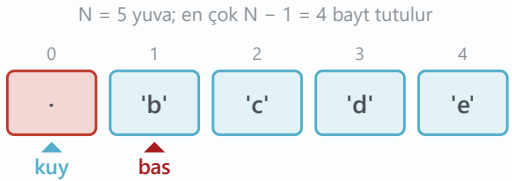
`işlem = koy('e')` `bas = 1` `kuy = 0` `dolu = 4`

ÇIKTI

```
al: a
```

Dairesel Tampon: Kod İzleme

```
1  int koy(char c) {          /* üretici */
▶    int s = (kuy + 1) % N;
▶    if (s == bas) return -1; /* dolu */
4    buf[kuy] = c;
5    kuy = s;
6    return 0;
7  }
8  int al(char *c) {          /* tüketici */
9    if (bas == kuy) return -1; /* boş */
10   *c = buf[bas];
11   bas = (bas + 1) % N;
12   return 0;
13 }
```



(kuy + 1) mod 5 = 1 = bas → dolu

`koy('f')` : s = (0 + 1) mod 5 = 1 = bas. Bir yuva hep boş bırakılır; aksi hâlde bas == kuy hem boş hem dolu demek olurdu. Bayt reddedilir (−1).

Adım 8 / 9

DEĞİŞKENLER

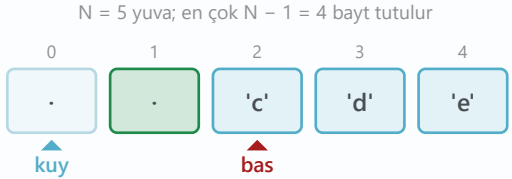
işlem = **koy('f')** | bas = **1** | kuy = **0** | dolu = **4**

ÇIKTI

```
al: a
koy('f'): dolu
```

Dairesel Tampon: Kod İzleme

```
1  int koy(char c) {          /* üretici */
2    int s = (kuy + 1) % N;
3    if (s == bas) return -1; /* dolu */
4    buf[kuy] = c;
5    kuy = s;
6    return 0;
7  }
8  int al(char *c) {          /* tüketici */
▶    if (bas == kuy) return -1; /* boş */
▶    *c = buf[bas];
▶    bas = (bas + 1) % N;
12   return 0;
13 }
```



`al` : bas ≠ kuy, tampon boş değil. buf[1] = 'b' okunur, bas = 2. Yuva 1 yeniden kullanılabilir.

Adım 9 / 9

DEĞİŞKENLER

işlem = **al** | bas = **2** | kuy = **0** | dolu = **3**

ÇIKTI

```
al: a
koy('f'): dolu
al: b
```

G/Ç YAZILIMI

Önbellek, Biriktirme ve Hata Ele Alma

Önbellek ≠ tampon

Tampon verinin *tek* kopyasını geçici tutar; önbellek başka yerde duran verinin hızlı *kopyasıdır*. Linux sayfa önbelleği ikisini birleştirir; boş RAM önbelleğe gider (`free -h` : buff/cache).

Biriktirme (spooling)

Adanmış aygıt için kuyruk: kullanıcı dosyayı biriktirme dizinine koyar, tek bir arka plan süreci (CUPS, `/var/spool/cups`) sırayla yazdırır. Hiçbir süreç yazıcıyı kilitlemez.

Hata ele alma

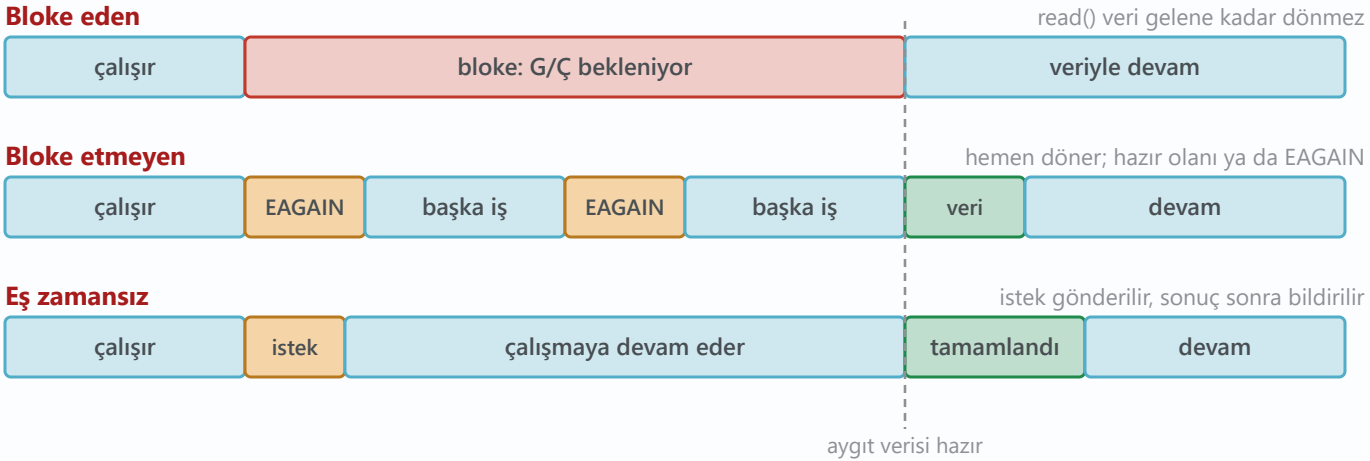
Programlama hatası (geçersiz fd) → hata kodu (`EBADF` , `EINVAL`). Gerçek G/Ç hatası → sürücü yeniden dener; kalıcıysa `EIO` ve günlük kaydı (`dmesg`).

Ayırma ve koruma

Adanmış aygıt `open` ile ayrılır, meşgulse istek kuyruğa alınır ya da `EBUSY` döner. G/Ç komutları ayrıcalıklıdır; kullanıcı yalnız sistem çağrısıyla G/Ç yapar.

Kullanıcı alanındaki G/Ç yazılımı: `printf` / `scanf` gibi kütüphane yordamları biçimlendirme ve kullanıcı tamponlaması yapar (`stdio` satır/blok tamponu), sonra `write` / `read` sistem çağrılarını kullanır.

Bloke Eden, Bloke Etmeyen ve Eş Zamansız G/Ç



Arayüz	Model	Not
`select` / `poll`	hazır olma bildirimi	her çağrıda tüm fd listesi taranır: O(n)
`epoll` (Linux), `kqueue` (BSD)	hazır olma bildirimi	ilgi listesi çekirdekte; yalnız hazır olanlar döner
`io_uring`, Windows IOCP	tamamlanma bildirimi	gerçek eş zamansız G/Ç; disk dahil

Olay Döngüsü: epoll

```
▶ int ep = epoll_create1(0);
2 for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
3     epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4 for (;;) {
5   int n = epoll_wait(ep, evs, 64, -1);
6   for (int i = 0; i < n; i++) {
7     int fd = evs[i].data.fd;
8     ssize_t r = read(fd, buf, sizeof buf);
9     if (r < 0 && errno == EAGAIN) continue;
10    handle(fd, buf, r);
11  }
12 }
```

ilgi listesi (çekirdekte)

hazır listesi

(boş)

Bir epoll örneği oluşturulur: çekirdekte ilgi ve hazır listeleri tutan bir nesne.

Adım 1 / 6

DEĞİŞKENLER

`ep = 3` `n = –` `fd = –` `r = –`

Tek iş parçacığı on binlerce bağlantıyı yönetir (nginx, Node.js, Redis). `epoll` "okuyabilirsin" der, işi yine `read` yapar; disk dosyaları her zaman "hazır" görünür, bu yüzden disk için gerçek eş zamansızlık `io_uring` ile gelir.

Olay Döngüsü: epoll

```
1  int ep = epoll_create1(0);
▶  for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
▶      epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4  for (;;) {
5    int n = epoll_wait(ep, evs, 64, -1);
6    for (int i = 0; i < n; i++) {
7      int fd = evs[i].data.fd;
8      ssize_t r = read(fd, buf, sizeof buf);
9      if (r < 0 && errno == EAGAIN) continue;
10     handle(fd, buf, r);
11   }
12 }
```

ilgi listesi (çekirdekte)

fd 5 | fd 6 | fd 7

hazır listesi

(boş)

Üç soket (5, 6, 7) **bloke etmeyen** modda ilgi listesine eklenir. Bu bir kez yapılır; `select`'teki gibi her turda tekrar verilmez.

Adım 2 / 6

DEĞİŞKENLER

`ep = 3` `n = –` `fd = –` `r = –`

Tek iş parçacığı on binlerce bağlantıyı yönetir (nginx, Node.js, Redis). `epoll` "okuyabilirsin" der, işi yine `read` yapar; disk dosyaları her zaman "hazır" görünür, bu yüzden disk için gerçek eş zamansızlık `io_uring` ile gelir.

Olay Döngüsü: epoll

```
1  int ep = epoll_create1(0);
2  for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
3      epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4  for (;;) {
▶    int n = epoll_wait(ep, evs, 64, -1);
6    for (int i = 0; i < n; i++) {
7      int fd = evs[i].data.fd;
8      ssize_t r = read(fd, buf, sizeof buf);
9      if (r < 0 && errno == EAGAIN) continue;
10     handle(fd, buf, r);
11   }
12 }
```

ilgi listesi (çekirdekte)

fd 5 | fd 6 | fd 7

hazır listesi

(boş)

`epoll_wait` hiçbir soket hazır değilken süreci **bloke eder**; CPU boşa dönmez.

Adım 3 / 6

DEĞİŞKENLER

`ep = 3` `n = –` `fd = –` `r = –`

Tek iş parçacığı on binlerce bağlantıyı yönetir (nginx, Node.js, Redis). `epoll` "okuyabilirsin" der, işi yine `read` yapar; disk dosyaları her zaman "hazır" görünür, bu yüzden disk için gerçek eş zamansızlık `io_uring` ile gelir.

Olay Döngüsü: epoll

```
1  int ep = epoll_create1(0);
2  for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
3      epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4  for (;;) {
▶    int n = epoll_wait(ep, evs, 64, -1);
6    for (int i = 0; i < n; i++) {
7      int fd = evs[i].data.fd;
8      ssize_t r = read(fd, buf, sizeof buf);
9      if (r < 0 && errno == EAGAIN) continue;
10     handle(fd, buf, r);
11   }
12 }
```

ilgi listesi (çekirdekte)

fd 5 | fd 6 | fd 7

hazır listesi

fd 7

Soket 7'ye paket geldi: NIC kesmesi → ağ yığını → fd 7 hazır listesine eklenir, süreç uyanır. n = 1.

Adım 4 / 6

DEĞİŞKENLER

`ep = 3` `n = 1` `fd = –` `r = –`

Tek iş parçacığı on binlerce bağlantıyı yönetir (nginx, Node.js, Redis). `epoll` "okuyabilirsin" der, işi yine `read` yapar; disk dosyaları her zaman "hazır" görünür, bu yüzden disk için gerçek eş zamansızlık `io_uring` ile gelir.

Olay Döngüsü: epoll

```
1  int ep = epoll_create1(0);
2  for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
3      epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4  for (;;) {
5    int n = epoll_wait(ep, evs, 64, -1);
6    for (int i = 0; i < n; i++) {
▶      int fd = evs[i].data.fd;
▶      ssize_t r = read(fd, buf, sizeof buf);
9      if (r < 0 && errno == EAGAIN) continue;
10     handle(fd, buf, r);
11   }
12 }
```

ilgi listesi (çekirdekte)

fd 5 | fd 6 | fd 7

hazır listesi

fd 7

Yalnız hazır olan fd işlenir: 512 bayt okunur. Maliyet hazır fd sayısıyla orantılıdır, toplamla değil.

Adım 5 / 6

DEĞİŞKENLER

`ep = 3` `n = 1` `fd = 7` `r = 512`

Tek iş parçacığı on binlerce bağlantıyı yönetir (nginx, Node.js, Redis). `epoll` "okuyabilirsin" der, işi yine `read` yapar; disk dosyaları her zaman "hazır" görünür, bu yüzden disk için gerçek eş zamansızlık `io_uring` ile gelir.

Olay Döngüsü: epoll

```
1  int ep = epoll_create1(0);
2  for (k = 0; k < nsock; k++)   /* O_NONBLOCK */
3      epoll_ctl(ep, EPOLL_CTL_ADD, s[k], &ev);
4  for (;;) {
5    int n = epoll_wait(ep, evs, 64, -1);
6    for (int i = 0; i < n; i++) {
7      int fd = evs[i].data.fd;
8      ssize_t r = read(fd, buf, sizeof buf);
▶      if (r < 0 && errno == EAGAIN) continue;
▶      handle(fd, buf, r);
11   }
12 }
```

ilgi listesi (çekirdekte)

fd 5 | fd 6 | fd 7

hazır listesi

(boş)

Veri işlendi; bir sonraki `read` EAGAIN dönerse beklenecek veri kalmamıştır. Döngü yeniden `epoll_wait`'e gider.

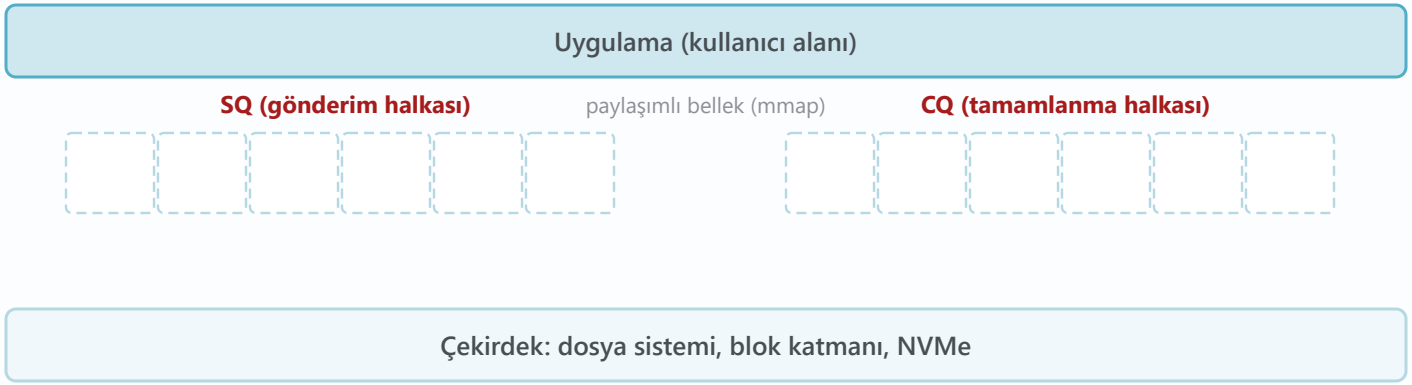
Adım 6 / 6

DEĞİŞKENLER

`ep = 3` `n = 1` `fd = 7` `r = -1 (EAGAIN)`

Tek iş parçacığı on binlerce bağlantıyı yönetir (nginx, Node.js, Redis). `epoll` "okuyabilirsin" der, işi yine `read` yapar; disk dosyaları her zaman "hazır" görünür, bu yüzden disk için gerçek eş zamansızlık `io_uring` ile gelir.

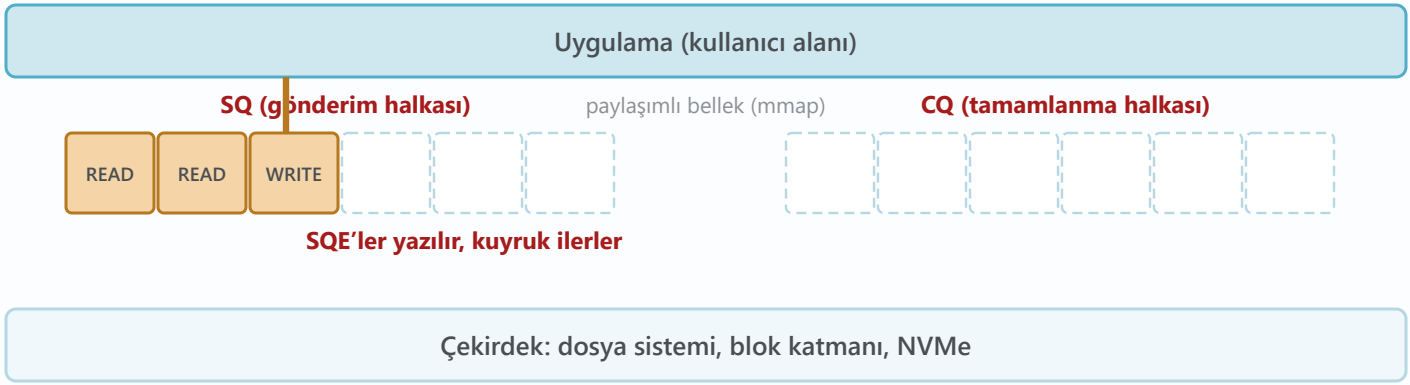
io_uring: Paylaşılan Halkalar



io_uring (Linux 5.1, 2019): uygulama ile çekirdek iki halkayı **paylaşımlı bellekte** görür; istek ve sonuç kopyalanmaz.

Bu bir **tamamlanma modelidir** (Windows IOCP gibi): "işin bitti" bildirilir. Ağ, dosya, `accept`, zamanlayıcı dahil yüzlerce işlem türü vardır. Güvenlik açıkları yüzünden bazı ortamlarda (ör. Android, bazı konteynerler) kapalıdır.

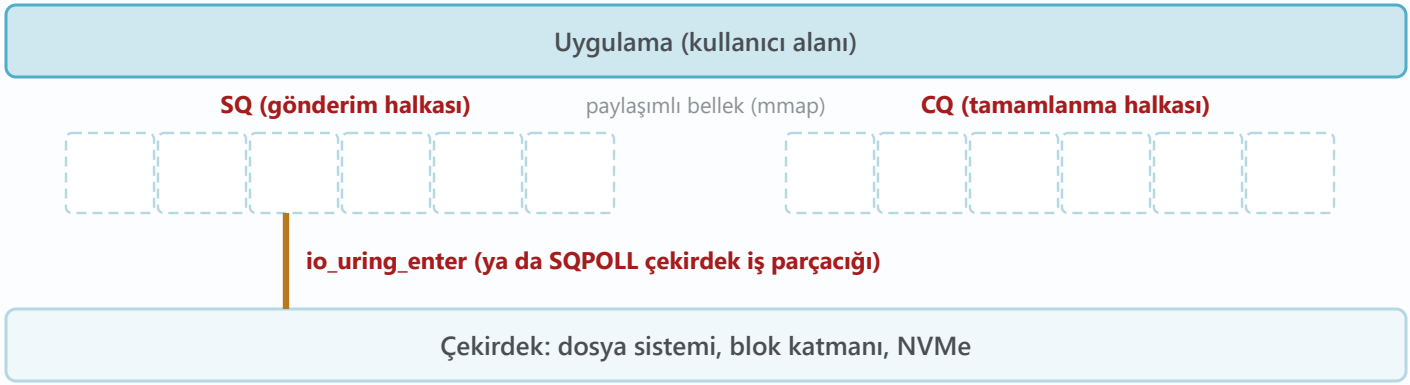
io_uring: Paylaşılan Halkalar



Uygulama üç gönderim kaydı (SQE) yazar: iki okuma, bir yazma. Her biri işlem kodu, fd, tampon, bağıl konum taşır.

Bu bir **tamamlanma modelidir** (Windows IOCP gibi): "işin bitti" bildirilir. Ağ, dosya, `accept`, zamanlayıcı dahil yüzlerce işlem türü vardır. Güvenlik açıkları yüzünden bazı ortamlarda (ör. Android, bazı konteynerler) kapalıdır.

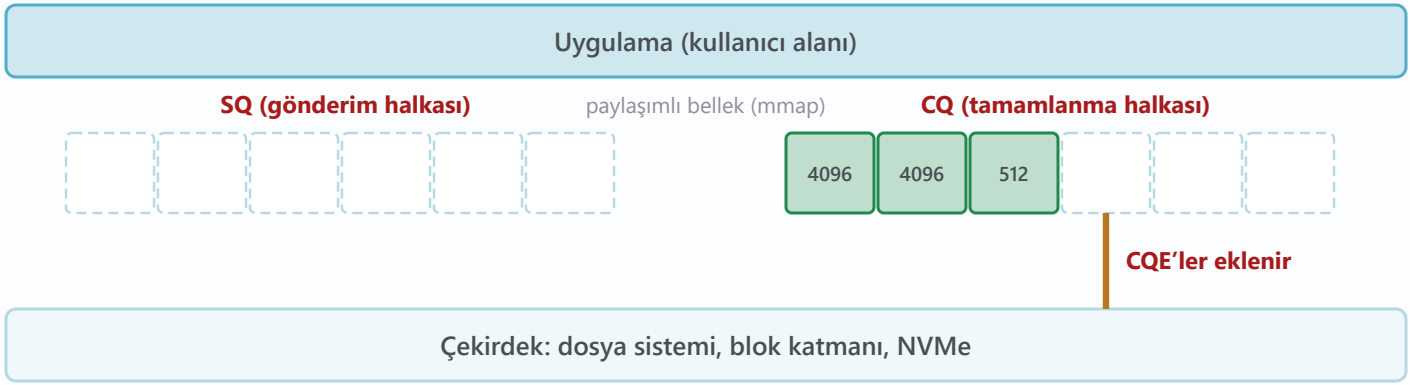
io_uring: Paylaşılan Halkalar



Tek bir `io_uring_enter` çağrısı üç isteği birden gönderir. SQPOLL modunda çekirdek iş parçacığı halkayı yoklar: **hiç sistem çağrısı** gerekmez.

Bu bir **tamamlanma modelidir** (Windows IOCP gibi): "işin bitti" bildirilir. Ağ, dosya, `accept`, zamanlayıcı dahil yüzlerce işlem türü vardır. Güvenlik açıkları yüzünden bazı ortamlarda (ör. Android, bazı konteynerler) kapalıdır.

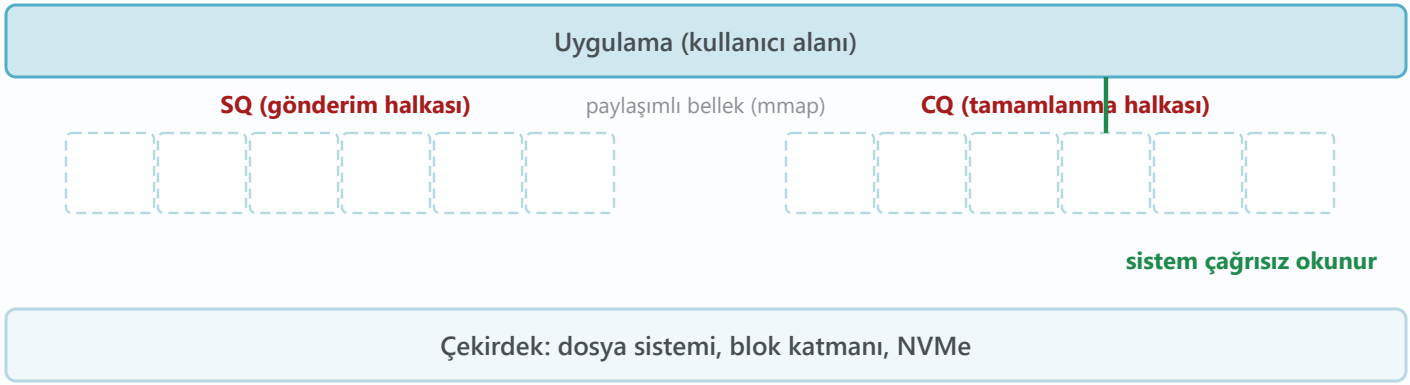
io_uring: Paylaşılan Halkalar



Aygıtlar tamamlandıkça çekirdek tamamlanma kayıtlarını (CQE) halkaya ekler; sonuç alanı aktarılan bayt ya da −errno'dur.

Bu bir **tamamlanma modelidir** (Windows IOCP gibi): "işin bitti" bildirilir. Ağ, dosya, `accept`, zamanlayıcı dahil yüzlerce işlem türü vardır. Güvenlik açıkları yüzünden bazı ortamlarda (ör. Android, bazı konteynerler) kapalıdır.

io_uring: Paylaşılan Halkalar



Uygulama CQE'leri doğrudan bellekten okur ve başı ilerletir. Toplu gönderim + toplu toplama → çok az bağlam değiştirme.

Bu bir **tamamlanma modelidir** (Windows IOCP gibi): "işin bitti" bildirilir. Ağ, dosya, `accept`, zamanlayıcı dahil yüzlerce işlem türü vardır. Güvenlik açıkları yüzünden bazı ortamlarda (ör. Android, bazı konteynerler) kapalıdır.

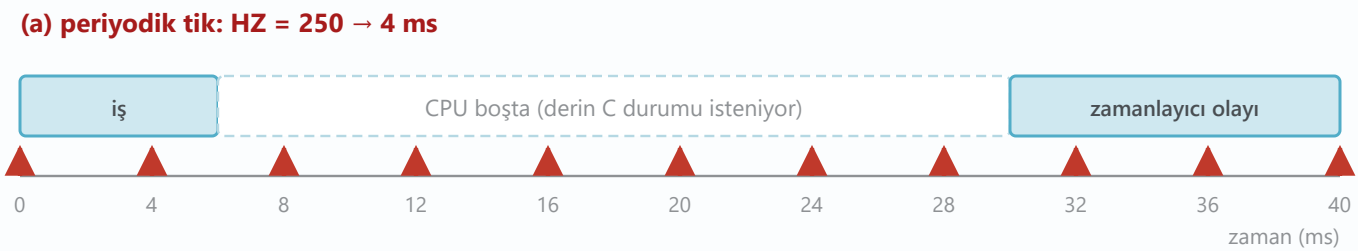
Saat Donanımı

modlar: tek atım (one-shot) · kare dalga (periyodik)



- Örnek (8254 PIT): 1 kHz kesme için sayaç = 1.193.182 / 1000 ≈ **1193** → gerçek frekans 1000,15 Hz.
- Modern x86: **TSC** (zaman damgası sayacı), **HPET**, **yerel APIC zamanlayıcısı** (TSC-deadline modu); ARM: genel zamanlayıcı. Pil destekli **RTC** kapalıyken tarihi tutar.
- Saat yazılımı: günün saati, zaman dilimi (quantum) tükenmesi, CPU kullanım muhasebesi, `alarm` /zamanlayıcılar, izleme köpeği (watchdog), profil çıkarma.
- Günün saati 64 bit sayaçla tutulmalı: 32 bit `time_t` 19 Ocak 2038'de taşar.

Saat Yazılımı: Tik ve Tiksiz Çekirdek



boşta da 6 gereksiz uyanma: her biri CPU'yu derin uykudan çıkarır

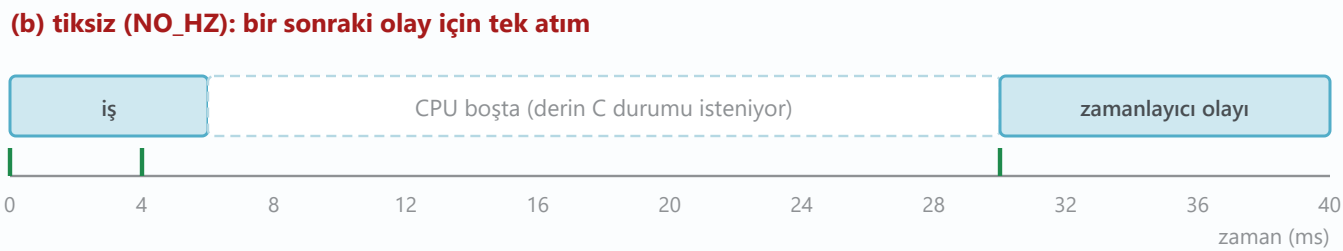
hrtimer: ns çözünürlüklü zamanlayıcılar, kırmızı-siyah ağaçta sıralı

Periyodik tik: zamanlayıcı her 1/HZ saniyede kesme üretir (Linux'ta HZ = 100, 250 ya da 1000 → 10, 4, 1 ms). İşleyici süreyi artırır, quantum'u azaltır, süresi dolan zamanlayıcıları çalıştırır.

Adım 1 / 3

Tiksiz çalışma dizüstü ve sanal makinelerde enerji ile işlemci zamanı kazandırır. Kısa süreli (µs) bekleme için ise *yumuşak zamanlayıcılar* ve yoklama birlikte kullanılır.

Saat Yazılımı: Tik ve Tiksiz Çekirdek



boşta tik yok: zamanlayıcı 30 ms'de bir kez programlanır

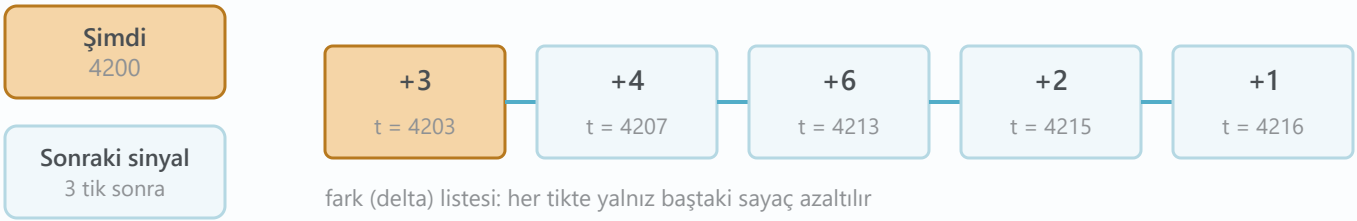
hrtimer: ns çözünürlüklü zamanlayıcılar, kırmızı-siyah ağaçta sıralı

Tiksiz çekirdek (NO_HZ_IDLE, 2007): CPU boşken tik durdurulur, saat bir sonraki olaya tek atım kurulur. `nohz_full` tek süreçli çekirdekte çalışırken bile tiki kaldırır (HPC, gerçek zamanlı).

Tiksiz çalışma dizüstü ve sanal makinelerde enerji ile işlemci zamanı kazandırır. Kısa süreli (µs) bekleme için ise *yumuşak zamanlayıcılar* ve yoklama birlikte kullanılır.

Saat Yazılımı: Tik ve Tiksiz Çekirdek

(c) zamanlayıcı listesi: şimdiki zaman 4200



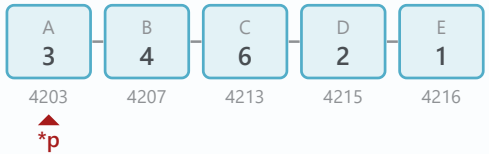
Tanenbaum'un örneği: bekleyen sinyaller 4203, 4207, 4213, 4215, 4216'da. Liste yalnız farkları tutar; baştaki 0'a inince olay tetiklenir.

Tiksiz çalışma dizüstü ve sanal makinelerde enerji ile işlemci zamanı kazandırır. Kısa süreli (µs) bekleme için ise *yumuşak zamanlayıcılar* ve yoklama birlikte kullanılır.

Zamanlayıcı Listesi: Kod İzleme

```
▶ void ekle(Zam *z, int t) {  /* t: tik */
▶   Zam **p = &bas;
3   while (*p && (*p)->fark <= t) {
4     t -= (*p)->fark;
5     p = &(*p)->sonraki;
6   }
7   z->fark = t; z->sonraki = *p;
8   if (*p) (*p)->fark -= t;
9   *p = z;
10 }
11 void tik(void) {
12   if (bas) bas->fark--;
13   while (bas && bas->fark == 0)
14     bas = calistir(bas);   /* çıkar */
15 }
```

şimdi = 4200 — kutuda: fark (önceki olaydan sonra)



alt satır: olayın mutlak zamanı

Bekleyen olaylar 4203, 4207, 4213, 4215, 4216'da; liste yalnız farkları tutar: 3, 4, 6, 2, 1. Yeni zamanlayıcı X 10 tik sonra (4210): `ekle(X, 10)`.

Adım 1 / 8

DEĞİŞKENLER

`şimdi = 4200` `t = 10` `baş.fark = 3`

ÇIKTI

Zamanlayıcı Listesi: Kod İzleme

```
1  void ekle(Zam *z, int t) {  /* t: tik */
2    Zam **p = &bas;
▶    while (*p && (*p)->fark <= t) {
▶      t -= (*p)->fark;
▶      p = &(*p)->sonraki;
6    }
7    z->fark = t; z->sonraki = *p;
8    if (*p) (*p)->fark -= t;
9    *p = z;
10 }
11 void tik(void) {
12   if (bas) bas->fark--;
13   while (bas && bas->fark == 0)
14     bas = calistir(bas);   /* çıkar */
15 }
```



A'nın farkı 3 ≤ t: X ondan sonra gelir. t = 7, *p sonraki düğüme geçer.

Adım 2 / 8

DEĞİŞKENLER

`şimdi = 4200` `t = 7` `baş.fark = 3`

ÇIKTI

Zamanlayıcı Listesi: Kod İzleme

```
1  void ekle(Zam *z, int t) {  /* t: tik */
2    Zam **p = &bas;
▶    while (*p && (*p)->fark <= t) {
▶      t -= (*p)->fark;
▶      p = &(*p)->sonraki;
6    }
7    z->fark = t; z->sonraki = *p;
8    if (*p) (*p)->fark -= t;
9    *p = z;
10 }
11 void tik(void) {
12   if (bas) bas->fark--;
13   while (bas && bas->fark == 0)
14     bas = calistir(bas);   /* çıkar */
15 }
```



B'nin farkı 4 ≤ t: X ondan sonra gelir. t = 3, *p sonraki düğüme geçer.

Adım 3 / 8

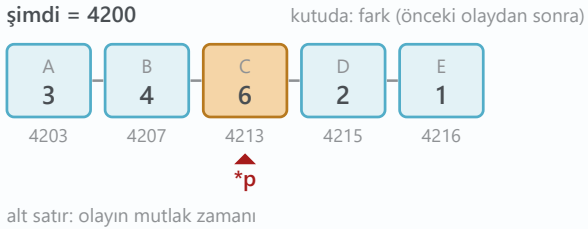
DEĞİŞKENLER

`şimdi = 4200` `t = 3` `baş.fark = 3`

ÇIKTI

Zamanlayıcı Listesi: Kod İzleme

```
1  void ekle(Zam *z, int t) {  /* t: tik */
2    Zam **p = &bas;
▶    while (*p && (*p)->fark <= t) {
4      t -= (*p)->fark;
5      p = &(*p)->sonraki;
6    }
7    z->fark = t; z->sonraki = *p;
8    if (*p) (*p)->fark -= t;
9    *p = z;
10 }
11 void tik(void) {
12   if (bas) bas->fark--;
13   while (bas && bas->fark == 0)
14     bas = calistir(bas);   /* çıkar */
15 }
```



C'nin farkı 6 > t = 3: X, C'den **önce** girmeli. Döngü biter.

Adım 4 / 8

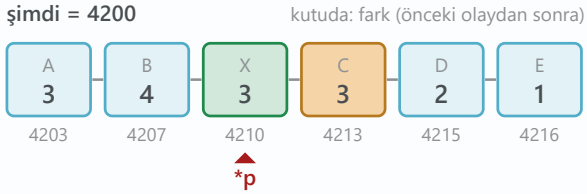
DEĞİŞKENLER

`şimdi = 4200` `t = 3` `baş.fark = 3`

ÇIKTI

Zamanlayıcı Listesi: Kod İzleme

```
1  void ekle(Zam *z, int t) {  /* t: tik */
2    Zam **p = &bas;
3    while (*p && (*p)->fark <= t) {
4      t -= (*p)->fark;
5      p = &(*p)->sonraki;
6    }
▶    z->fark = t; z->sonraki = *p;
▶    if (*p) (*p)->fark -= t;
▶    *p = z;
10 }
11 void tik(void) {
12   if (bas) bas->fark--;
13   while (bas && bas->fark == 0)
14     bas = calistir(bas);   /* çıkar */
15 }
```



X.fark = 3 olarak araya girer; C'nin farkı 6 − 3 = 3 olur. C'nin mutlak zamanı (4213) değişmedi; yalnız bir düğüm güncellendi.

Adım 5 / 8

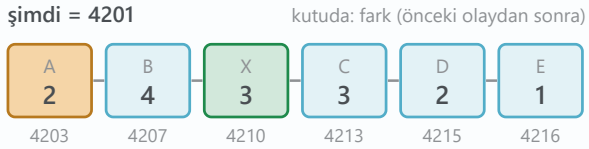
DEĞİŞKENLER

`şimdi = 4200` `t = 3` `baş.fark = 3`

ÇIKTI

Zamanlayıcı Listesi: Kod İzleme

```
1  void ekle(Zam *z, int t) {  /* t: tik */
2    Zam **p = &bas;
3    while (*p && (*p)->fark <= t) {
4      t -= (*p)->fark;
5      p = &(*p)->sonraki;
6    }
7    z->fark = t; z->sonraki = *p;
8    if (*p) (*p)->fark -= t;
9    *p = z;
10 }
11 void tik(void) {
▶    if (bas) bas->fark--;
13   while (bas && bas->fark == 0)
14     bas = calistir(bas);   /* çıkar */
15 }
```



alt satır: olayın mutlak zamanı

Tik: yalnız baştaki fark azalır → 2.

Adım 6 / 8

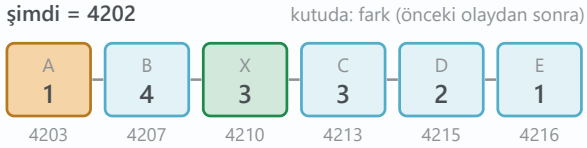
DEĞİŞKENLER

`şimdi = 4201` `t = 3` `baş.fark = 2`

ÇIKTI

Zamanlayıcı Listesi: Kod İzleme

```
 1 void ekle(Zam *z, int t) {  /* t: tik */
 2   Zam **p = &bas;
 3   while (*p && (*p)->fark <= t) {
 4     t -= (*p)->fark;
 5     p = &(*p)->sonraki;
 6   }
 7   z->fark = t; z->sonraki = *p;
 8   if (*p) (*p)->fark -= t;
 9   *p = z;
10 }
11 void tik(void) {
▶   if (bas) bas->fark--;
13   while (bas && bas->fark == 0)
14     bas = calistir(bas);   /* çıkar */
15 }
```



alt satır: olayın mutlak zamanı

Tik: yalnız baştaki fark azalır → 1.

Adım 7 / 8

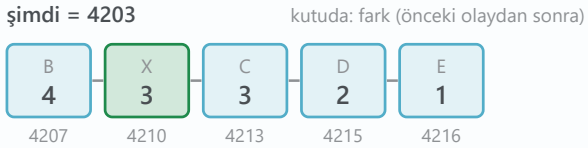
DEĞİŞKENLER

`şimdi = 4202` `t = 3` `baş.fark = 1`

ÇIKTI

Zamanlayıcı Listesi: Kod İzleme

```
 1 void ekle(Zam *z, int t) {  /* t: tik */
 2   Zam **p = &bas;
 3   while (*p && (*p)->fark <= t) {
 4     t -= (*p)->fark;
 5     p = &(*p)->sonraki;
 6   }
 7   z->fark = t; z->sonraki = *p;
 8   if (*p) (*p)->fark -= t;
 9   *p = z;
10 }
11 void tik(void) {
▶   if (bas) bas->fark--;
▶   while (bas && bas->fark == 0)
▶     bas = calistir(bas);   /* çıkar */
15 }
```



alt satır: olayın mutlak zamanı

4203: baştaki fark 0 oldu; A çalıştırılıp listeden çıkarılır. Tik başına iş yalnız **baştaki** sayacı azaltmaktır: liste ne kadar uzun olursa olsun O(1).

Adım 8 / 8

DEĞİŞKENLER

`şimdi = 4203` `t = 3` `baş.fark = 4`

ÇIKTI

```
4203: A çalıştı
```

Terminaller, Giriş Aygıtları ve Güç Yönetimi

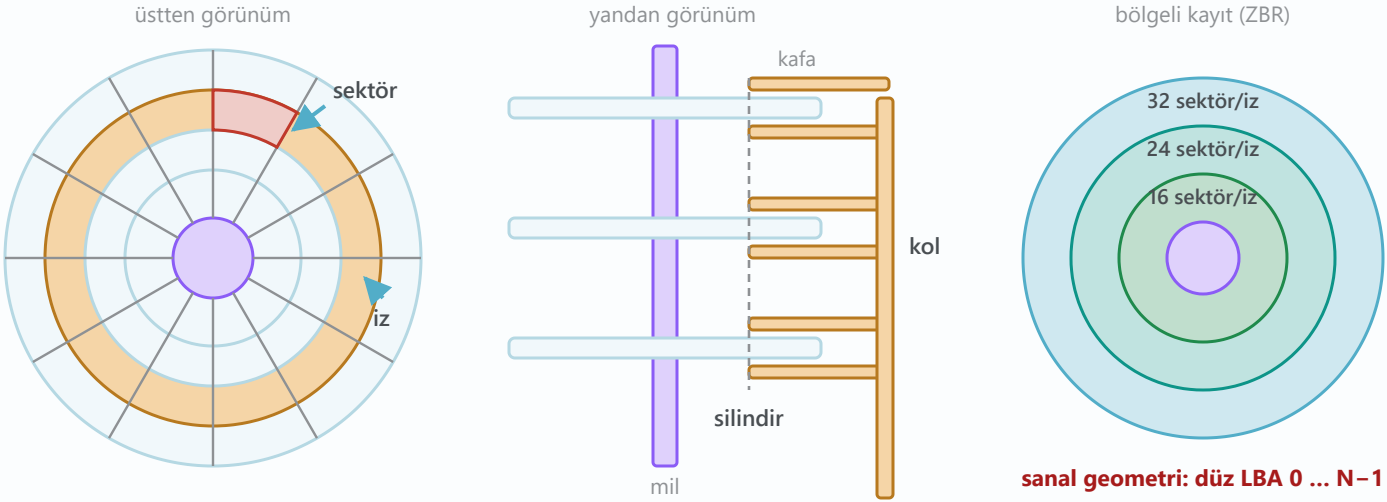
Terminaller ve GUI girişi

- **tty** + **satır disiplini**: kanonik modda satır düzenlenir (geri silme, Enter'da teslim); ham modda her tuş hemen gelir (`stty -icanon` , editörler).
- **Sözde terminal** (pty): terminal öykünücüsü, `ssh` ; ekran denetimi ANSI kaçış dizileriyle.
- Klavye *tarama kodu* gönderir (basma/bırakma); sürücü tuş eşlemesiyle karaktere çevirir. USB HID aygıtları ana makine tarafından 125–1000 Hz *yoklanır*.
- Linux: input alt sistemi → `/dev/input/event*` (evdev) → libinput → Wayland birleştiricisi → pencere.

Güç yönetimi

- ACPI sistem durumları: **S0** çalışır (S0ix modern bekleme), **S3** RAM'e askı, **S4** hazırda bekleme (diske), **S5** kapalı.
- CPU: **C durumları** (boşta uyku derinliği), **P durumları** (frekans/gerilim; Linux `schedutil`).
- Aygıtlar: **D0–D3** durumları, çalışma zamanı güç yönetimi; NVMe APST, PCIe ASPM, Wi-Fi güç tasarrufu, ekranın kararması.
- İşletim sistemi aygıtları doğru sırayla uyutur/uyandırır ve uyandırma kaynaklarını (klavye, ağ, saat) belirler. Pil ömrünün büyük kısmı boşta geçen süreye bağlıdır.

Manyetik Disk Geometrisi



- Plakalar 5400–7200 rpm (sunucu: 10–15 bin) döner; her yüzeye bir kafa. Aynı yarıçaptaki izler **silindiri** oluşturur. Dış bölgelerde iz başına daha çok sektör vardır.
- İşletim sistemi CHS (silindir, kafa, sektör) yerine **LBA** görür; disk içindeki denetleyici bunu gerçek geometriye, yedek sektörlere ve önbelleğe kendisi çevirir. Eski CHS sınırı: 1024 × 255 × 63 × 512 B ≈ 8,4 GB.

Erişim Süresi: Örnek

Erişim süresi = **arama** (seek) + **dönme gecikmesi** + **aktarım**. Disk: 7200 rpm, ortalama arama 8 ms, ortam hızı 200 MB/s; blok 4 KiB.

Aktarım süresi de dahil üç bileşen birlikte sayılmalıdır.

Erişim Süresi: Örnek

Erişim süresi = **arama** (seek) + **dönme gecikmesi** + **aktarım**. Disk: 7200 rpm, ortalama arama 8 ms, ortam hızı 200 MB/s; blok 4 KiB.

Bir tur = 60.000 / 7200 = **8,33 ms**; ortalama dönme gecikmesi = yarım tur = **4,17 ms**.

İstenen sektör ortalamada yarım tur uzaktadır.

Erişim Süresi: Örnek

Erişim süresi = **arama** (seek) + **dönme gecikmesi** + **aktarım**. Disk: 7200 rpm, ortalama arama 8 ms, ortam hızı 200 MB/s; blok 4 KiB.

Bir tur = 60.000 / 7200 = **8,33 ms**; ortalama dönme gecikmesi = yarım tur = **4,17 ms**.

Aktarım = 4096 B ÷ 200 MB/s = **0,020 ms**. Toplam = 8 + 4,17 + 0,02 = **12,19 ms**.

Mekanik gecikme baskın: süre neredeyse tamamen arama ve dönmede geçer. CPU bu sürede milyonlarca komut çalıştırabilir; İS beklemez, başka süreç çalıştırır.

Erişim Süresi: Örnek

Erişim süresi = **arama** (seek) + **dönme gecikmesi** + **aktarım**. Disk: 7200 rpm, ortalama arama 8 ms, ortam hızı 200 MB/s; blok 4 KiB.

Bir tur = 60.000 / 7200 = **8,33 ms**; ortalama dönme gecikmesi = yarım tur = **4,17 ms**.

Aktarım = 4096 B ÷ 200 MB/s = **0,020 ms**. Toplam = 8 + 4,17 + 0,02 = **12,19 ms**.

Rastgele 4 KiB: 1000 / 12,19 ≈ **82 IOPS** → 82 × 4 KiB ≈ **0,34 MB/s**. Ardışık 1 MiB: 8 + 4,17 + 5,24 = 17,41 ms → **60 MB/s**.

Aynı disk, erişim düzenine göre 180 kat farklı hız verir: yerellik ve çizelgeleme bu yüzden önemli.

Erişim Süresi: Örnek

Erişim süresi = **arama** (seek) + **dönme gecikmesi** + **aktarım**. Disk: 7200 rpm, ortalama arama 8 ms, ortam hızı 200 MB/s; blok 4 KiB.

Bir tur = 60.000 / 7200 = **8,33 ms**; ortalama dönme gecikmesi = yarım tur = **4,17 ms**.

Aktarım = 4096 B ÷ 200 MB/s = **0,020 ms**. Toplam = 8 + 4,17 + 0,02 = **12,19 ms**.

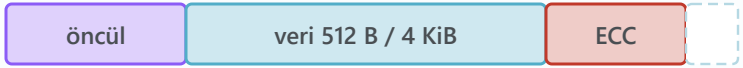
Rastgele 4 KiB: 1000 / 12,19 ≈ **82 IOPS** → 82 × 4 KiB ≈ **0,34 MB/s**. Ardışık 1 MiB: 8 + 4,17 + 5,24 = 17,41 ms → **60 MB/s**.



Karşılaştırma: NVMe SSD'de 4 KiB rastgele okuma ≈ 50–100 µs ve paralel kuyruklarla 1 milyonun üzerinde IOPS.

Disk Biçimlendirme ve Silindir Eğriliği

sektör biçimi (düşük düzey format)



eğrilik (skew): izler arası geçişte sektör 0 kaçırılmasın

iz 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
iz 1	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12
iz 2	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9

- **Düşük düzey format** fabrikada yapılır: izler, sektörler, öncül, ECC ve yedek sektörler. Biçimlendirilmiş kapasite ham kapasiteden azdır. **Üst düzey format** ise bir bölüme boş dosya sistemi yazar (önyükleme bloğu, boş alan yönetimi, kök dizin).
- Eğrilik hesabı (Tanenbaum): 10.000 rpm → tur 6 ms; 300 sektör/iz → sektör başına 20 µs. İz değiştirme 800 µs ise eğrilik = 800 / 20 = **40 sektör**.
- **Serpiştirme** (interleaving) yavaş denetleyiciler için ardışık sektörleri aralıklı dizerdi; bugün denetleyici tüm izi tampona okuduğundan gereksizdir.
- Bugünkü diskler: 4 KiB fiziksel sektör (Advanced Format); 512e diskler 512 B öykünür, hizasız yazma okuma-değiştir-yaz gerektirir.

Optik Diskler (Kısaca)

	CD (1982)	DVD (1996)	Blu-ray (2006)
Kapasite	650–700 MB	4,7 / 8,5 / 9,4 / 17 GB	25 / 50 GB; BDXL 100–128 GB
Lazer	780 nm (kızılötesi)	650 nm (kırmızı)	405 nm (mavi-mor)
En küçük çukur	0,83 µm	0,40 µm	0,15 µm
İz aralığı	1,6 µm	0,74 µm	0,32 µm

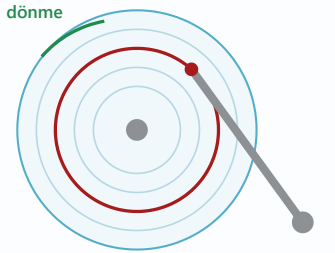
- Tek bir **sarmal iz**; çukur ile düzlük arasındaki *geçiş* 1 bitidir. Basılı disklerde çukurlar kalıpla polikarbonata basılır; lazerle yakılmaz. CD-R/DVD-R'de lazer organik boya katmanını ısıtıp yansıtıcılığı değiştirir.
- CD-ROM'da her bayt **EFM** ile 14 bitlik kanal sembolüne çevrilir (bu bir modülasyon kodudur, ECC değildir); hata düzeltme **CIRC** ile yapılır. 98 çerçeve × 24 veri baytı = 2352 B; mod 1'de bunun 2048 B'ı kullanıcı verisi, kalanı senkron, başlık ve ek ECC/EDC'dir.
- DVD'nin iki katmanı yarı saydamdır; lazer odağını değiştirerek alttakini okur. Optik ortam bugün çoğunlukla arşiv ve dağıtım içindir; yoğunlukta manyetik ve flash ortamın gerisindedir.



optik: tek sarmal iz manyetik: eş merkezli

Disk Kolu Çizelgeleme

- Amaç kol hareketini (arama süresi) azaltıp verimi artırmak; sürücü (silindir, istek zamanı) listesini tutar. Ölçüt: **toplam kafa hareketi** (silindir).
- Bugün LBA → fiziksel eşleme diskin içindedir; diskin kendisi **NCQ** ile 32 isteği dönme konumuna göre yeniden sıralar. İşletim sistemi yaklaşık olarak "LBA yakınlığı" ile çizelgeler.



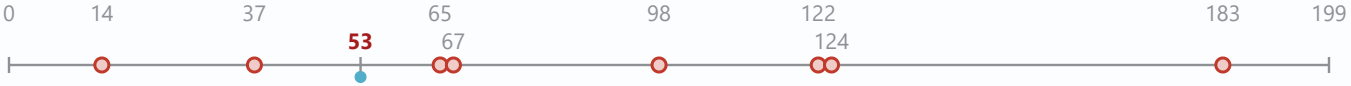
arama: kol silindir değiştirir
kırmızı: kafanın silindiri

Algoritma	Fikir	Sorun
FCFS	Geliş sırası	Adil ama kol çok gezer
SSTF	Kafaya en yakın istek (açgözlü)	Uçtaki istekler *açlık* çekebilir
SCAN (asansör)	Bir yönde uca kadar git, dön	Uçlar ve orta eşit beklemez
C-SCAN	Tek yönde tara, uçtan başa sıçra	Dönüş hareketi boşa
LOOK / C-LOOK	SCAN/C-SCAN, ama son isteğe kadar	Pratikte kullanılan biçimler

Örnekler Silberschatz'ın kuyruğunu kullanır: kafa 53, istekler 98, 183, 37, 122, 14, 124, 65, 67. Silindirler 0–199.

FCFS

İlk gelen ilk hizmet alır: istekler geliş sırasıyla.



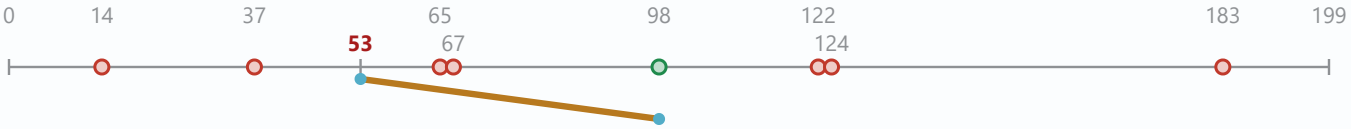
hizmet verildi bekliyor **toplam: 0 silindir**

Kafa **53**'te. Kuyruk: 98, 183, 37, 122, 14, 124, 65, 67 (silindir 0–199). Kırmızı: bekleyen, yeşil: hizmet verilmiş.

Toplam 640 silindir. Adil ve açlıksız, ama 122 → 14 → 124 gibi gidip gelmeler çok pahalı.

FCFS

İlk gelen ilk hizmet alır: istekler geliş sırasıyla.



hizmet verildi bekliyor

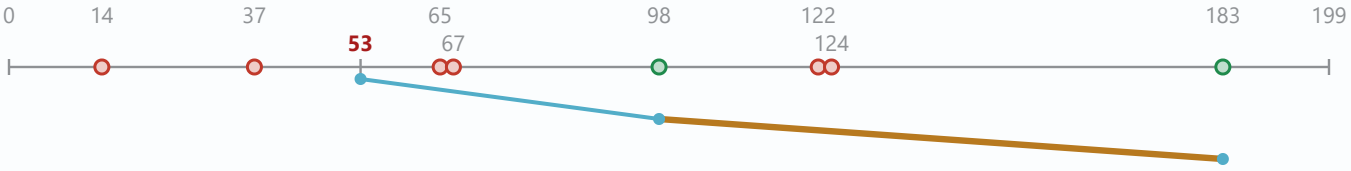
toplam: 45 silindir

53 → 98: **45** silindir · toplam **45**

Toplam 640 silindir. Adil ve açlıksız, ama 122 → 14 → 124 gibi gidip gelmeler çok pahalı.

FCFS

İlk gelen ilk hizmet alır: istekler geliş sırasıyla.



hizmet verildi bekliyor

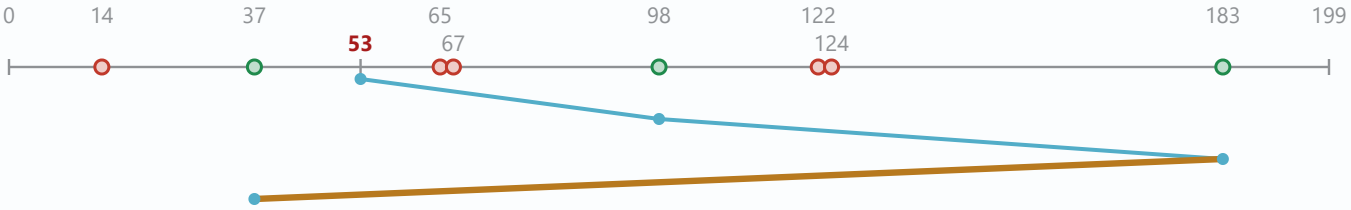
toplam: 130 silindir

98 → 183: **85** silindir · toplam **130**

Toplam 640 silindir. Adil ve açlıksız, ama 122 → 14 → 124 gibi gidip gelmeler çok pahalı.

FCFS

İlk gelen ilk hizmet alır: istekler geliş sırasıyla.

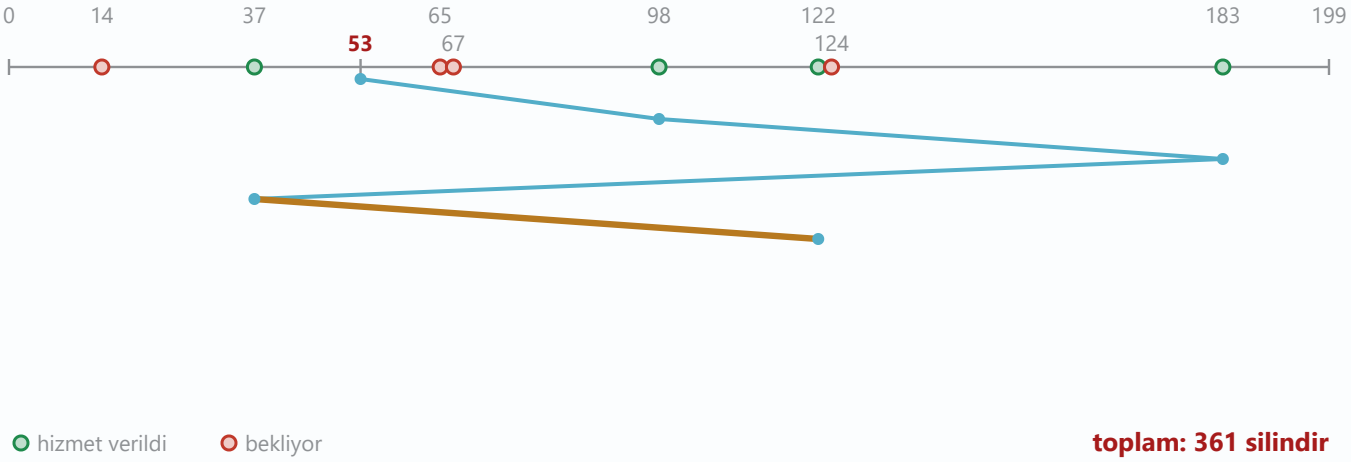


183 → 37: **146** silindir · toplam **276**

Toplam 640 silindir. Adil ve açlıksız, ama 122 → 14 → 124 gibi gidip gelmeler çok pahalı.

FCFS

İlk gelen ilk hizmet alır: istekler geliş sırasıyla.

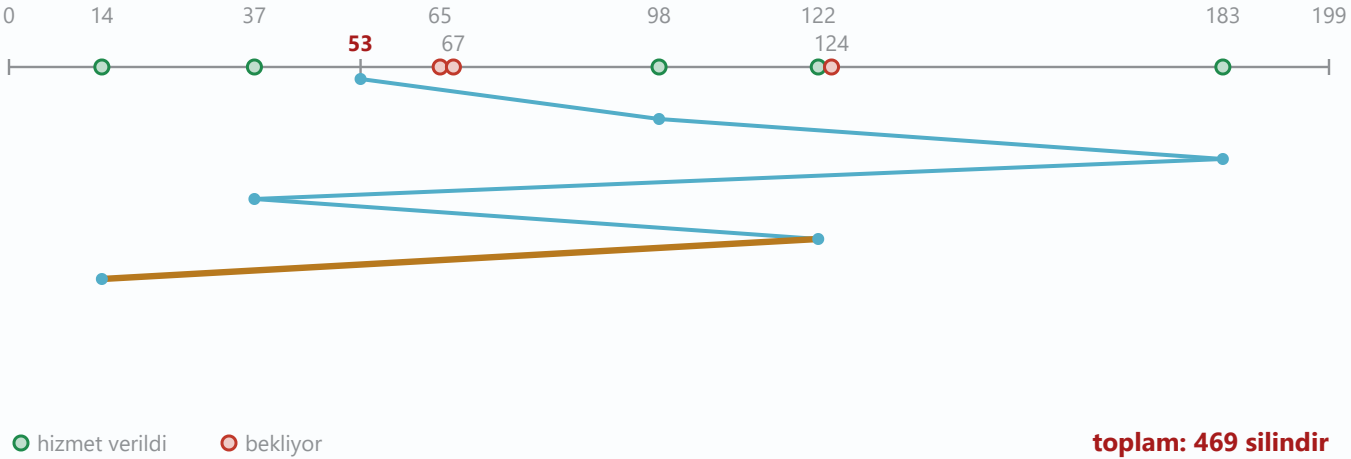


37 → 122: **85** silindir · toplam **361**

Toplam 640 silindir. Adil ve açlıksız, ama 122 → 14 → 124 gibi gidip gelmeler çok pahalı.

FCFS

İlk gelen ilk hizmet alır: istekler geliş sırasıyla.

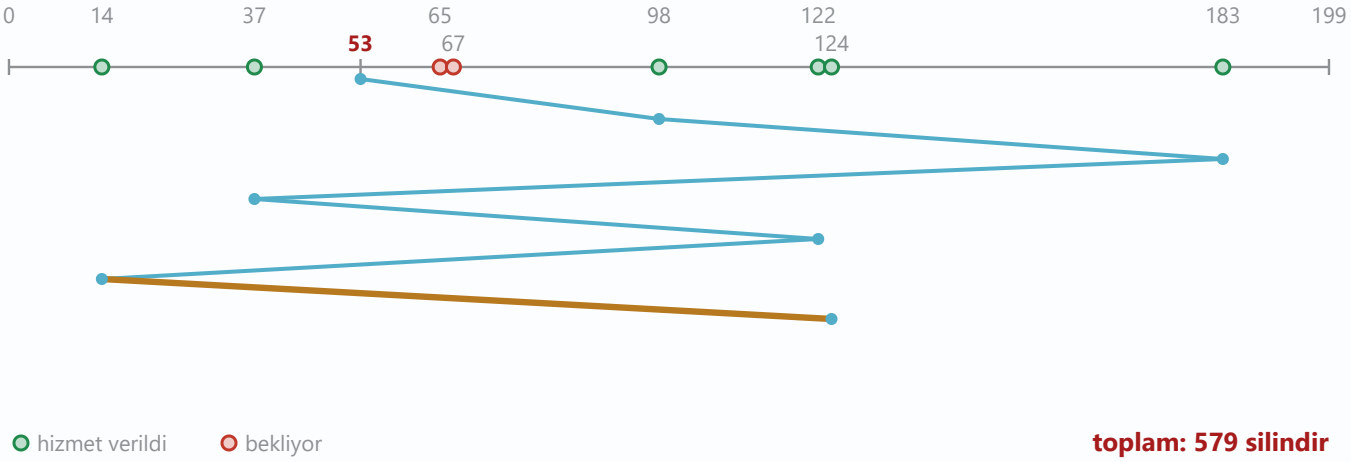


122 → 14: **108** silindir · toplam **469**

Toplam 640 silindir. Adil ve açlıksız, ama 122 → 14 → 124 gibi gidip gelmeler çok pahalı.

FCFS

İlk gelen ilk hizmet alır: istekler geliş sırasıyla.

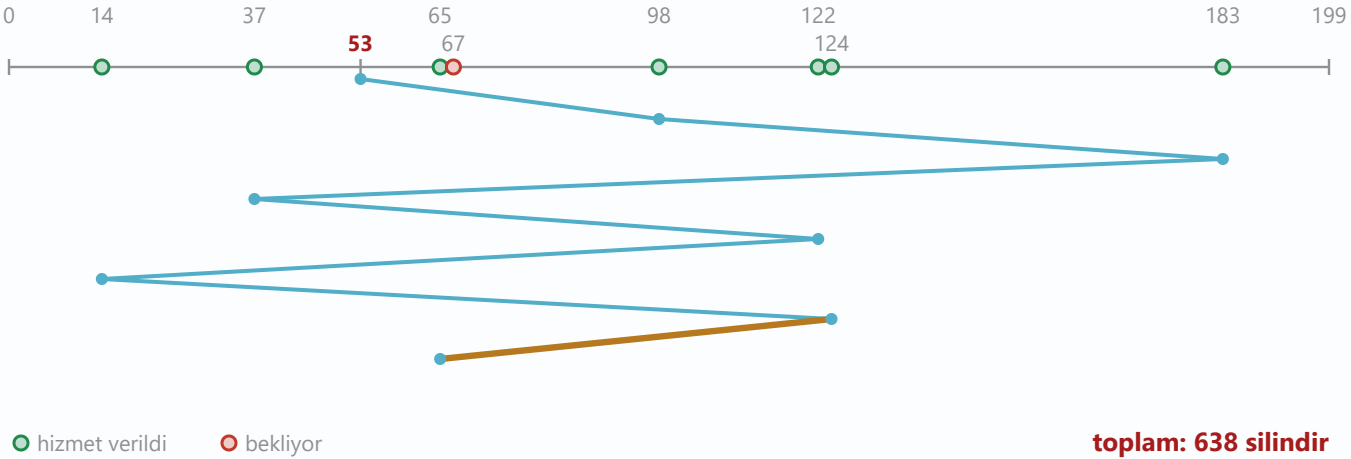


14 → 124: **110** silindir · toplam **579**

Toplam 640 silindir. Adil ve açlıksız, ama 122 → 14 → 124 gibi gidip gelmeler çok pahalı.

FCFS

İlk gelen ilk hizmet alır: istekler geliş sırasıyla.

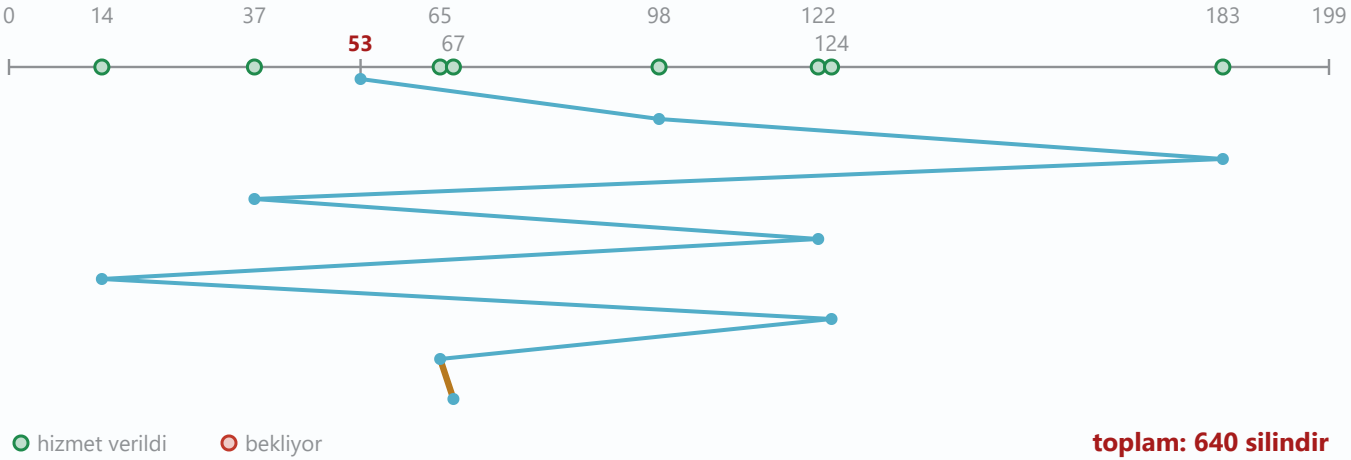


124 → 65: **59** silindir · toplam **638**

Toplam 640 silindir. Adil ve açlıksız, ama 122 → 14 → 124 gibi gidip gelmeler çok pahalı.

FCFS

İlk gelen ilk hizmet alır: istekler geliş sırasıyla.

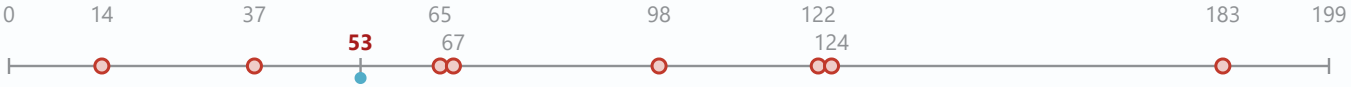


65 → 67: **2** silindir · toplam **640**

Toplam 640 silindir. Adil ve açlıksız, ama 122 → 14 → 124 gibi gidip gelmeler çok pahalı.

SSTF

En kısa arama süresi önce: her adımda kafaya en yakın istek.



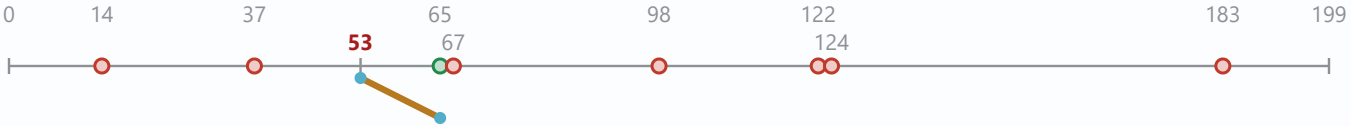
hizmet verildi bekliyor **toplam: 0 silindir**

Kafa **53**'te. Kuyruk: 98, 183, 37, 122, 14, 124, 65, 67 (silindir 0–199). Kırmızı: bekleyen, yeşil: hizmet verilmiş.

Toplam 236 (FCFS'nin %37'si). Açgözlüdür ve en iyi sonucu garanti etmez: 53 → 37 → 14 → 65 … daha kısa (208). Yoğun yükte merkezdeki istekler uçtakileri *aç bırakabilir*.

SSTF

En kısa arama süresi önce: her adımda kafaya en yakın istek.



hizmet verildi bekliyor

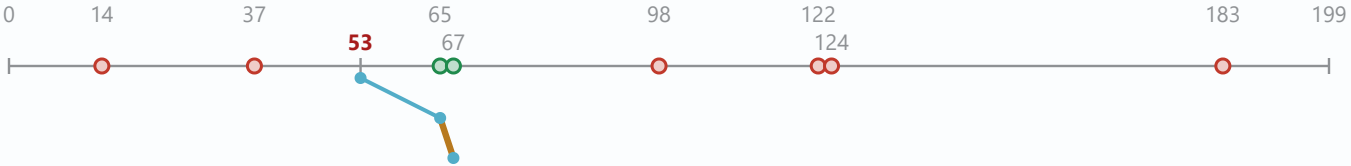
toplam: 12 silindir

53 → 65: **12** silindir · toplam **12**

Toplam 236 (FCFS'nin %37'si). Açgözlüdür ve en iyi sonucu garanti etmez: 53 → 37 → 14 → 65 … daha kısa (208). Yoğun yükte merkezdeki istekler uçtakileri *aç bırakabilir*.

SSTF

En kısa arama süresi önce: her adımda kafaya en yakın istek.



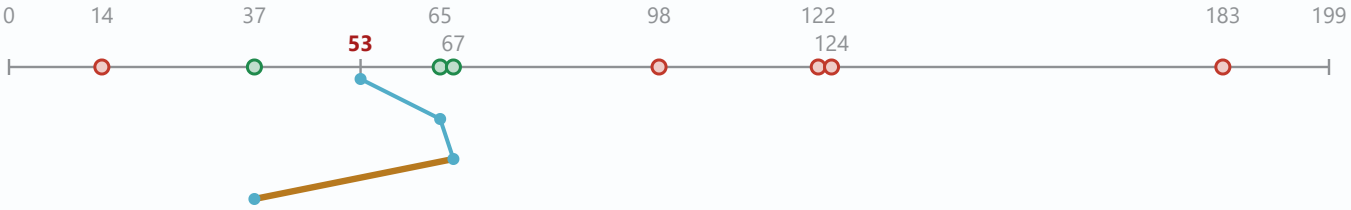
hizmet verildi bekliyor **toplam: 14 silindir**

65 → 67: **2** silindir · toplam **14**

Toplam 236 (FCFS'nin %37'si). Açgözlüdür ve en iyi sonucu garanti etmez: 53 → 37 → 14 → 65 … daha kısa (208). Yoğun yükte merkezdeki istekler uçtakileri *aç bırakabilir*.

SSTF

En kısa arama süresi önce: her adımda kafaya en yakın istek.

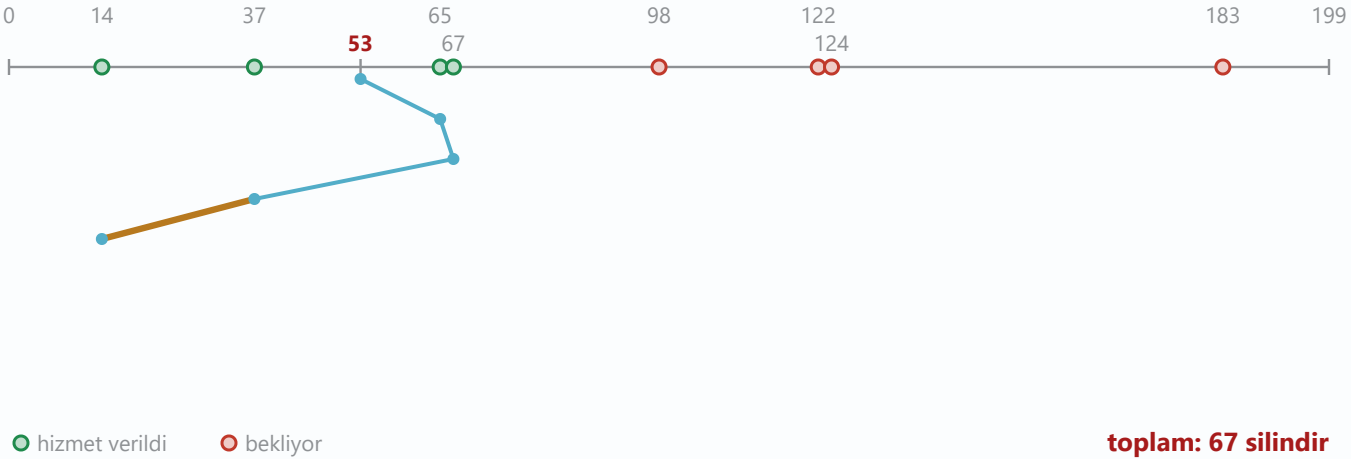


67 → 37: **30** silindir · toplam **44**

Toplam 236 (FCFS'nin %37'si). Açgözlüdür ve en iyi sonucu garanti etmez: 53 → 37 → 14 → 65 … daha kısa (208). Yoğun yükte merkezdeki istekler uçtakileri *aç bırakabilir*.

SSTF

En kısa arama süresi önce: her adımda kafaya en yakın istek.

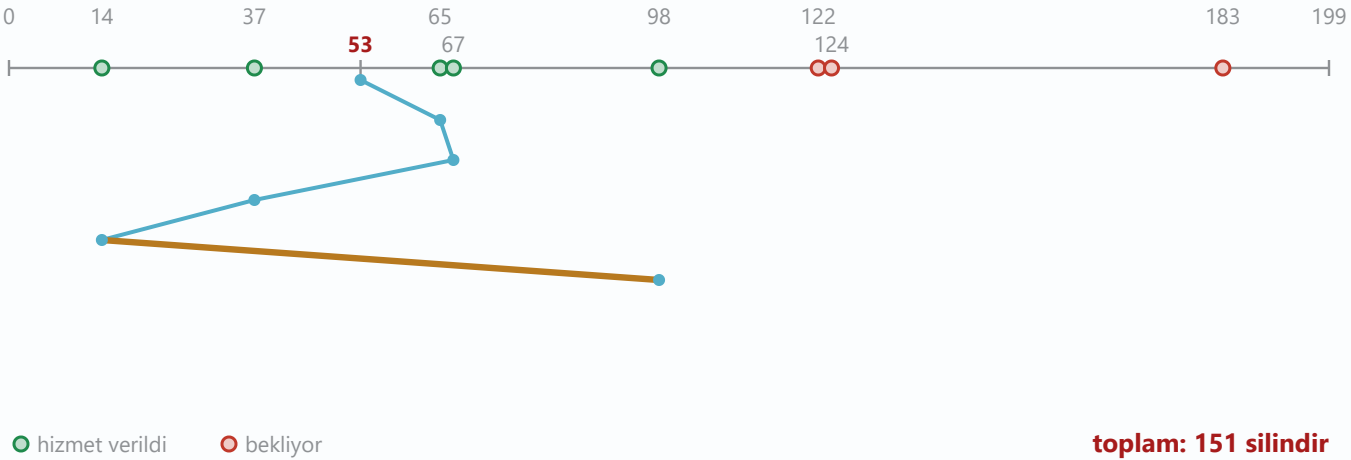


37 → 14: **23** silindir · toplam **67**

Toplam 236 (FCFS'nin %37'si). Açgözlüdür ve en iyi sonucu garanti etmez: 53 → 37 → 14 → 65 … daha kısa (208). Yoğun yükte merkezdeki istekler uçtakileri *aç bırakabilir*.

SSTF

En kısa arama süresi önce: her adımda kafaya en yakın istek.



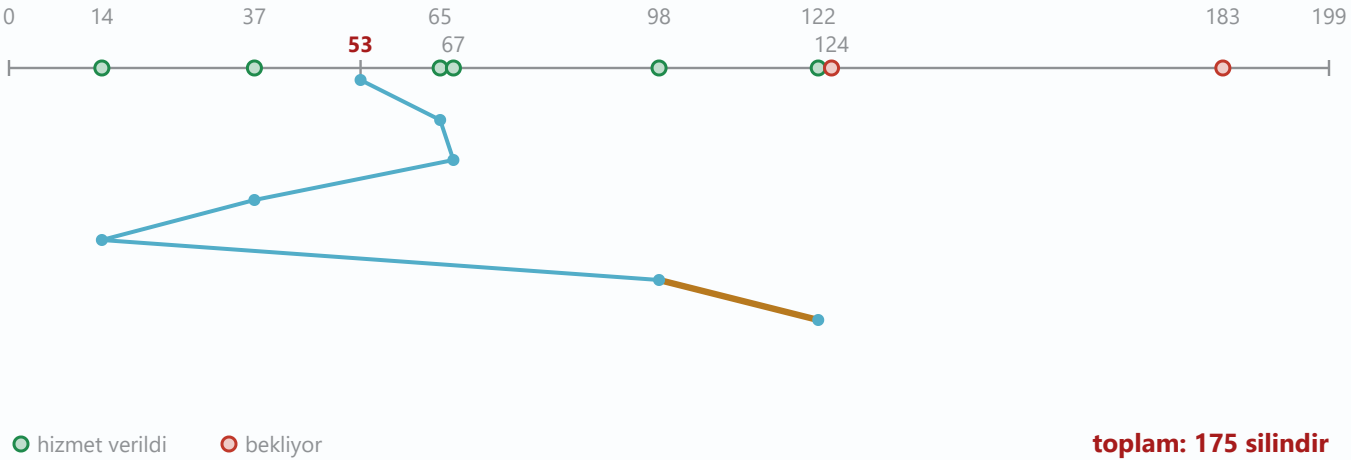
14 → 98: **84** silindir · toplam **151**

Adım 6 / 9

Toplam 236 (FCFS'nin %37'si). Açgözlüdür ve en iyi sonucu garanti etmez: 53 → 37 → 14 → 65 … daha kısa (208). Yoğun yükte merkezdeki istekler uçtakileri *aç bırakabilir*.

SSTF

En kısa arama süresi önce: her adımda kafaya en yakın istek.

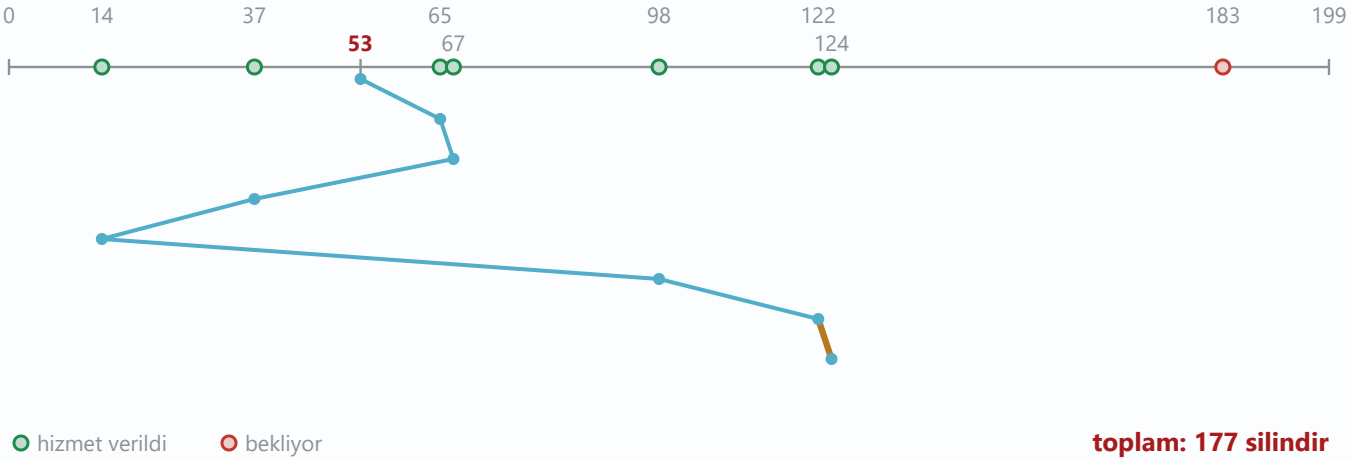


98 → 122: **24** silindir · toplam **175**

Toplam 236 (FCFS'nin %37'si). Açgözlüdür ve en iyi sonucu garanti etmez: 53 → 37 → 14 → 65 … daha kısa (208). Yoğun yükte merkezdeki istekler uçtakileri *aç bırakabilir*.

SSTF

En kısa arama süresi önce: her adımda kafaya en yakın istek.

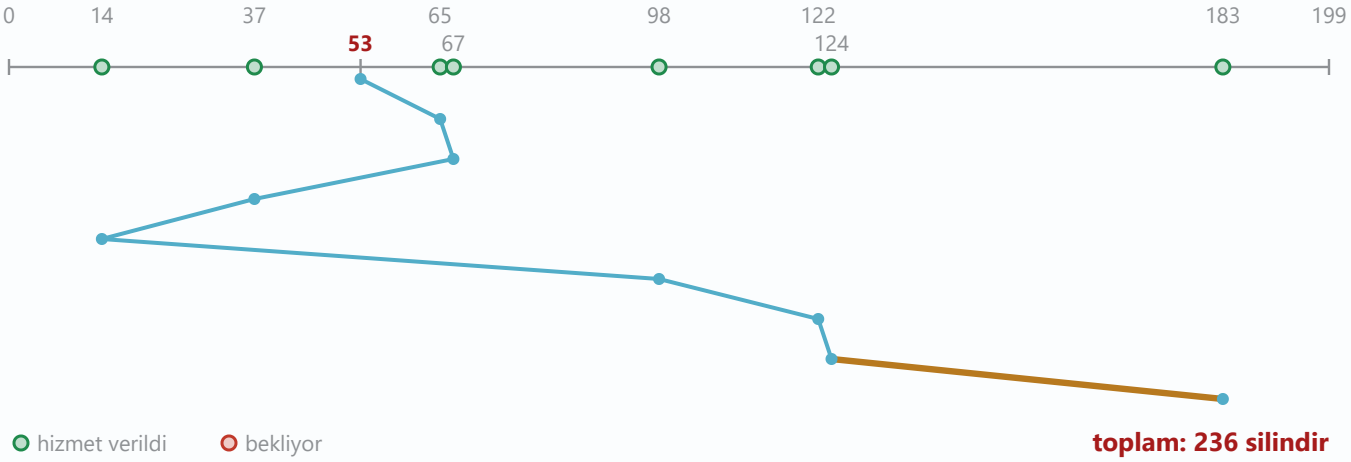


122 → 124: **2** silindir · toplam **177**

Toplam 236 (FCFS'nin %37'si). Açgözlüdür ve en iyi sonucu garanti etmez: 53 → 37 → 14 → 65 … daha kısa (208). Yoğun yükte merkezdeki istekler uçtakileri *aç bırakabilir*.

SSTF

En kısa arama süresi önce: her adımda kafaya en yakın istek.

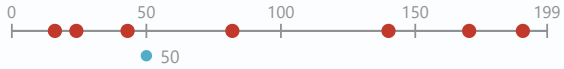


124 → 183: **59** silindir · toplam **236**

Toplam 236 (FCFS'nin %37'si). Açgözlüdür ve en iyi sonucu garanti etmez: 53 → 37 → 14 → 65 … daha kısa (208). Yoğun yükte merkezdeki istekler uçtakileri *aç bırakabilir*.

SSTF: Kod İzleme

```
▶ int sstf(int bas, int q[], int n) {
▶   int top = 0;
3   for (int k = 0; k < n; k++) {
4     int en = -1;
5     for (int i = 0; i < n; i++)
6       if (!bitti[i] && (en < 0 ||
7         abs(q[i]-bas) < abs(q[en]-bas)))
8         en = i;            /* en yakın */
9     top += abs(q[en] - bas);
10    bas = q[en];
11    bitti[en] = 1;
12  }
13  return top;
14 }
```



toplam = 0 silindir

Kafa silindir 50'de; kuyruk (geliş sırası): 82, 170, 43, 140, 24, 16, 190. Her turda iç döngü bekleyen isteklerin hepsine bakar.

Adım 1 / 9

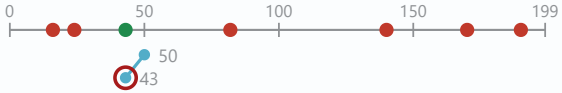
DEĞİŞKENLER

`bas = 50` `k = –` `en = –` `top = 0`

ÇIKTI

SSTF: Kod İzleme

```
1  int sstf(int bas, int q[], int n) {
2    int top = 0;
3    for (int k = 0; k < n; k++) {
▶      int en = -1;
▶      for (int i = 0; i < n; i++)
▶        if (!bitti[i] && (en < 0 ||
▶          abs(q[i]-bas) < abs(q[en]-bas)))
▶          en = i;            /* en yakın */
▶      top += abs(q[en] - bas);
▶      bas = q[en];
▶      bitti[en] = 1;
12   }
13   return top;
14 }
```



toplam = 7 silindir

k = 0: kafa 50, en yakın bekleyen **43** (uzaklık 7; sonraki aday 24, uzaklık 26). top = 7.

Adım 2 / 9

DEĞİŞKENLER

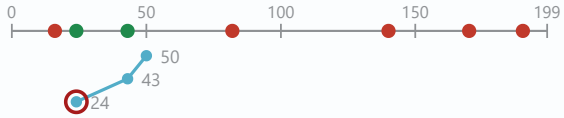
`bas = 43` `k = 0` `en = 2` `top = 7`

ÇIKTI

```
43
```

SSTF: Kod İzleme

```
1  int sstf(int bas, int q[], int n) {
2    int top = 0;
3    for (int k = 0; k < n; k++) {
4      int en = -1;
5      for (int i = 0; i < n; i++)
6        if (!bitti[i] && (en < 0 ||
7          abs(q[i]-bas) < abs(q[en]-bas)))
▶          en = i;            /* en yakın */
▶      top += abs(q[en] - bas);
▶      bas = q[en];
▶      bitti[en] = 1;
12   }
13   return top;
14 }
```



toplam = 26 silindir

k = 1: kafa 43, en yakın bekleyen **24** (uzaklık 19; sonraki aday 16, uzaklık 27). top = 26.

Adım 3 / 9

DEĞİŞKENLER

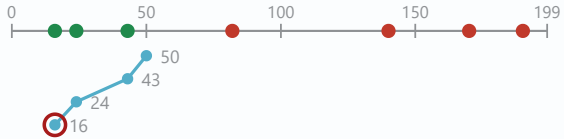
`bas = 24` `k = 1` `en = 4` `top = 26`

ÇIKTI

```
43 24
```

SSTF: Kod İzleme

```
1  int sstf(int bas, int q[], int n) {
2    int top = 0;
3    for (int k = 0; k < n; k++) {
4      int en = -1;
5      for (int i = 0; i < n; i++)
6        if (!bitti[i] && (en < 0 ||
7          abs(q[i]-bas) < abs(q[en]-bas)))
▶          en = i;            /* en yakın */
▶      top += abs(q[en] - bas);
▶      bas = q[en];
▶      bitti[en] = 1;
12   }
13   return top;
14 }
```



toplam = 34 silindir

k = 2: kafa 24, en yakın bekleyen **16** (uzaklık 8; sonraki aday 82, uzaklık 58). top = 34.

Adım 4 / 9

DEĞİŞKENLER

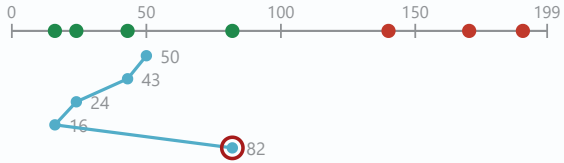
`bas = 16` `k = 2` `en = 5` `top = 34`

ÇIKTI

```
43 24 16
```

SSTF: Kod İzleme

```
1  int sstf(int bas, int q[], int n) {
2    int top = 0;
3    for (int k = 0; k < n; k++) {
4      int en = -1;
5      for (int i = 0; i < n; i++)
6        if (!bitti[i] && (en < 0 ||
7          abs(q[i]-bas) < abs(q[en]-bas)))
▶          en = i;            /* en yakın */
▶      top += abs(q[en] - bas);
▶      bas = q[en];
▶      bitti[en] = 1;
12   }
13   return top;
14 }
```



toplam = 100 silindir

k = 3: kafa 16, en yakın bekleyen **82** (uzaklık 66; sonraki aday 140, uzaklık 124). top = 100. Aşağıdaki istekler bitti, kol uzun bir yolu geri döner.

Adım 5 / 9

DEĞİŞKENLER

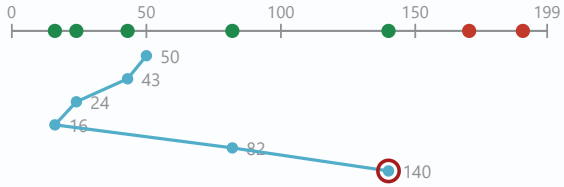
`bas = 82` `k = 3` `en = 0` `top = 100`

ÇIKTI

```
43 24 16 82
```

SSTF: Kod İzleme

```
1  int sstf(int bas, int q[], int n) {
2    int top = 0;
3    for (int k = 0; k < n; k++) {
4      int en = -1;
5      for (int i = 0; i < n; i++)
6        if (!bitti[i] && (en < 0 ||
7          abs(q[i]-bas) < abs(q[en]-bas)))
▶          en = i;            /* en yakın */
▶      top += abs(q[en] - bas);
▶      bas = q[en];
▶      bitti[en] = 1;
12   }
13   return top;
14 }
```



toplam = 158 silindir

k = 4: kafa 82, en yakın bekleyen **140** (uzaklık 58; sonraki aday 170, uzaklık 88). top = 158.

Adım 6 / 9

DEĞİŞKENLER

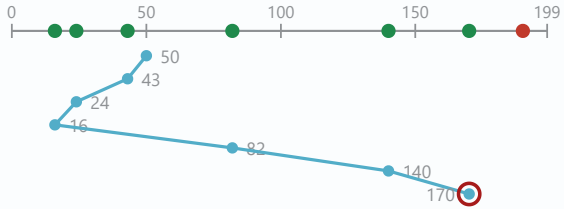
`bas = 140` `k = 4` `en = 3` `top = 158`

ÇIKTI

```
43 24 16 82 140
```

SSTF: Kod İzleme

```
1  int sstf(int bas, int q[], int n) {
2    int top = 0;
3    for (int k = 0; k < n; k++) {
4      int en = -1;
5      for (int i = 0; i < n; i++)
6        if (!bitti[i] && (en < 0 ||
7          abs(q[i]-bas) < abs(q[en]-bas)))
▶          en = i;            /* en yakın */
▶      top += abs(q[en] - bas);
▶      bas = q[en];
▶      bitti[en] = 1;
12   }
13   return top;
14 }
```



toplam = 188 silindir

k = 5: kafa 140, en yakın bekleyen **170** (uzaklık 30; sonraki aday 190, uzaklık 50). top = 188.

Adım 7 / 9

DEĞİŞKENLER

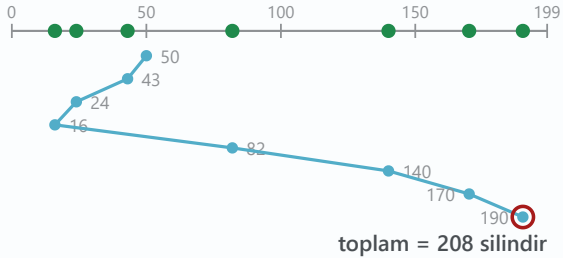
`bas = 170` `k = 5` `en = 1` `top = 188`

ÇIKTI

```
43 24 16 82 140 170
```

SSTF: Kod İzleme

```
1  int sstf(int bas, int q[], int n) {
2    int top = 0;
3    for (int k = 0; k < n; k++) {
4      int en = -1;
5      for (int i = 0; i < n; i++)
6        if (!bitti[i] && (en < 0 ||
7          abs(q[i]-bas) < abs(q[en]-bas)))
▶          en = i;            /* en yakın */
▶      top += abs(q[en] - bas);
▶      bas = q[en];
▶      bitti[en] = 1;
12   }
13   return top;
14 }
```



toplam = 208 silindir

k = 6: kafa 170, en yakın bekleyen **190** (uzaklık 20). top = 208.

Adım 8 / 9

DEĞİŞKENLER

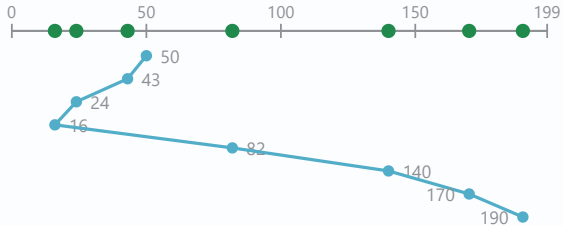
`bas = 190` `k = 6` `en = 6` `top = 208`

ÇIKTI

```
43 24 16 82 140 170 190
```

SSTF: Kod İzleme

```
1  int sstf(int bas, int q[], int n) {
2    int top = 0;
3    for (int k = 0; k < n; k++) {
4      int en = -1;
5      for (int i = 0; i < n; i++)
6        if (!bitti[i] && (en < 0 ||
7          abs(q[i]-bas) < abs(q[en]-bas)))
8          en = i;            /* en yakın */
9      top += abs(q[en] - bas);
10     bas = q[en];
11     bitti[en] = 1;
12   }
▶  13  return top;
14 }
```



toplam = 208 silindir

Toplam **208** silindir (FCFS bu kuyrukta 642). Açgözlü seçim her adımda en kısa hareketi alır; ortadaki istekler sürekli gelirse uçtakiler bekleyebilir.

Adım 9 / 9

DEĞİŞKENLER

`bas = 190` `k = 7` `en = -` `top = 208`

ÇIKTI

```
43 24 16 82 140 170 190
```

SCAN (Asansör)

SCAN: kafa bir yönde (burada 0'a doğru) uca kadar gider, sonra döner; asansör gibi.



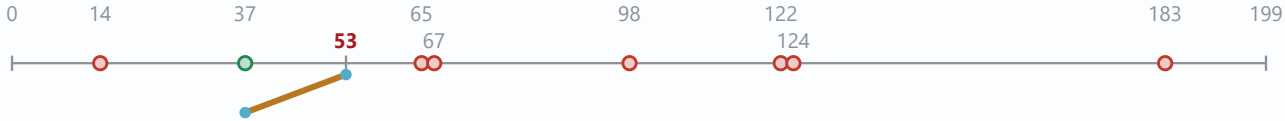
hizmet verildi bekliyor **toplam: 0 silindir**

Kafa **53**'te. Kuyruk: 98, 183, 37, 122, 14, 124, 65, 67 (silindir 0–199). Kırmızı: bekleyen, yeşil: hizmet verilmiş.

Toplam 236. "Açgözlü, kafa bir bölgede sıkışabilir" özelliği SSTF'ye aittir; SCAN açlığı önler. Uca kadar gidip dönmek (14 → 0 → 14) boşa harcanan 28 silindir demektir.

SCAN (Asansör)

SCAN: kafa bir yönde (burada 0'a doğru) uca kadar gider, sonra döner; asansör gibi.



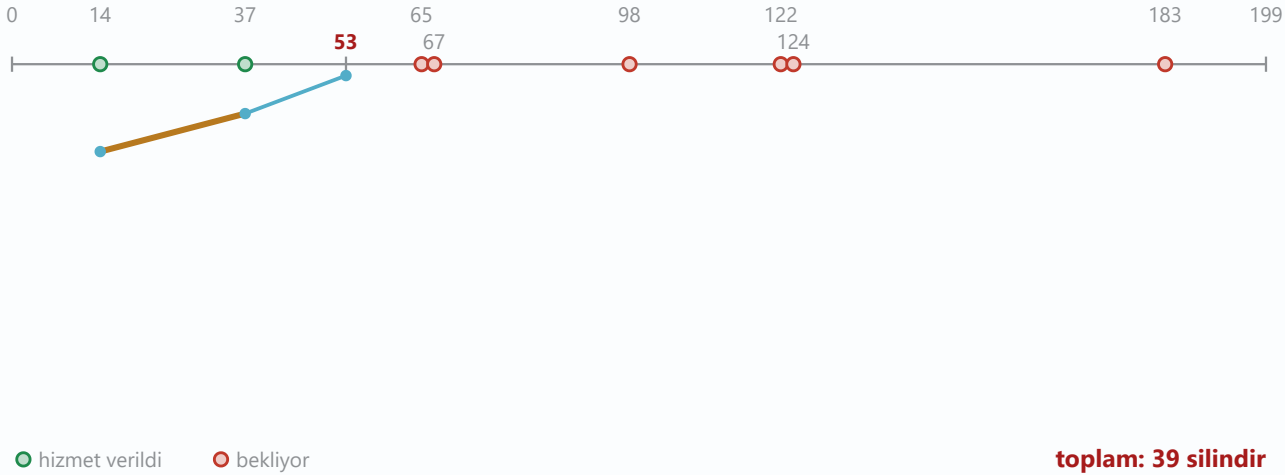
hizmet verildi bekliyor **toplam: 16 silindir**

53 → 37: **16** silindir · toplam **16**

Toplam 236. "Açgözlü, kafa bir bölgede sıkışabilir" özelliği SSTF'ye aittir; SCAN açlığı önler. Uca kadar gidip dönmek (14 → 0 → 14) boşa harcanan 28 silindir demektir.

SCAN (Asansör)

SCAN: kafa bir yönde (burada 0'a doğru) uca kadar gider, sonra döner; asansör gibi.

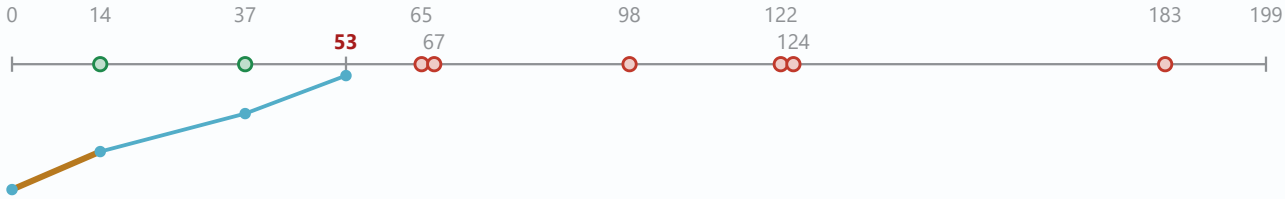


37 → 14: **23** silindir · toplam **39**

Toplam 236. "Açgözlü, kafa bir bölgede sıkışabilir" özelliği SSTF'ye aittir; SCAN açlığı önler. Uca kadar gidip dönmek (14 → 0 → 14) boşa harcanan 28 silindir demektir.

SCAN (Asansör)

SCAN: kafa bir yönde (burada 0'a doğru) uca kadar gider, sonra döner; asansör gibi.



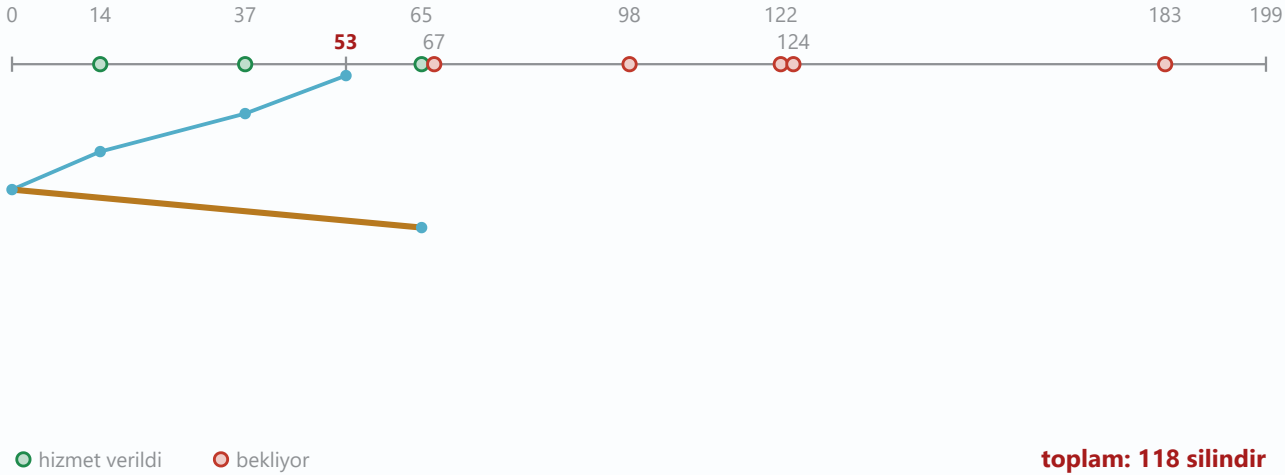
hizmet verildi bekliyor **toplam: 53 silindir**

14 → 0: **14** silindir · toplam **53** (istek yok, diskin ucuna kadar gidilir)

Toplam 236. "Açgözlü, kafa bir bölgede sıkışabilir" özelliği SSTF'ye aittir; SCAN açlığı önler. Uca kadar gidip dönmek (14 → 0 → 14) boşa harcanan 28 silindir demektir.

SCAN (Asansör)

SCAN: kafa bir yönde (burada 0'a doğru) uca kadar gider, sonra döner; asansör gibi.

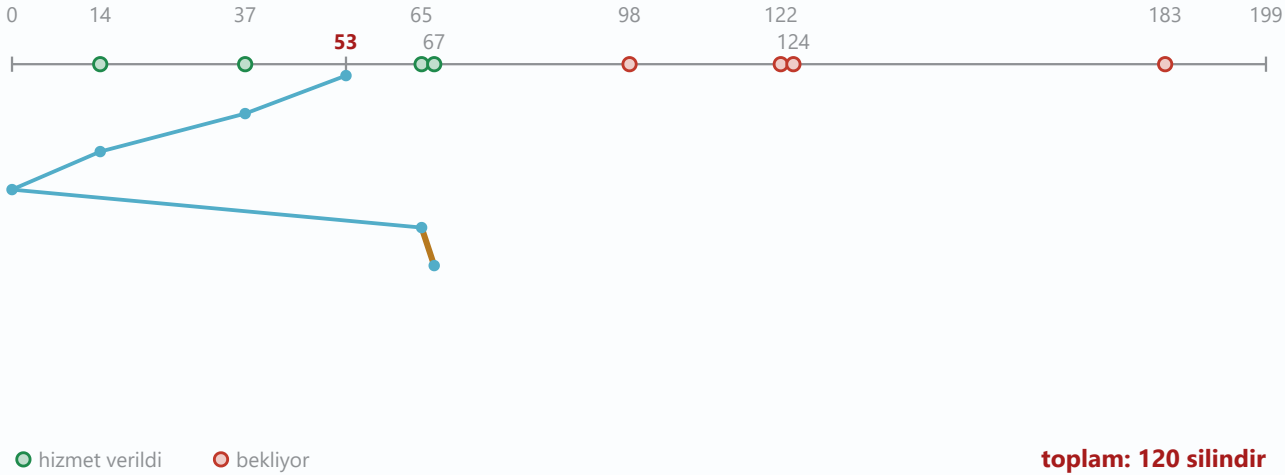


0 → 65: **65** silindir · toplam **118**

Toplam 236. "Açgözlü, kafa bir bölgede sıkışabilir" özelliği SSTF'ye aittir; SCAN açlığı önler. Uca kadar gidip dönmek (14 → 0 → 14) boşa harcanan 28 silindir demektir.

SCAN (Asansör)

SCAN: kafa bir yönde (burada 0'a doğru) uca kadar gider, sonra döner; asansör gibi.

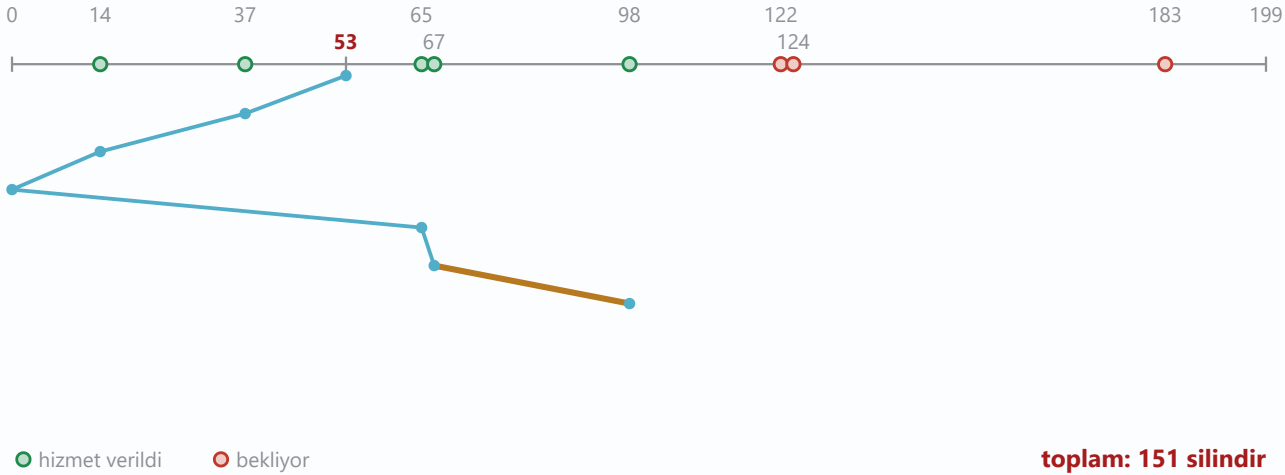


65 → 67: **2** silindir · toplam **120**

Toplam 236. "Açgözlü, kafa bir bölgede sıkışabilir" özelliği SSTF'ye aittir; SCAN açlığı önler. Uca kadar gidip dönmek (14 → 0 → 14) boşa harcanan 28 silindir demektir.

SCAN (Asansör)

SCAN: kafa bir yönde (burada 0'a doğru) uca kadar gider, sonra döner; asansör gibi.

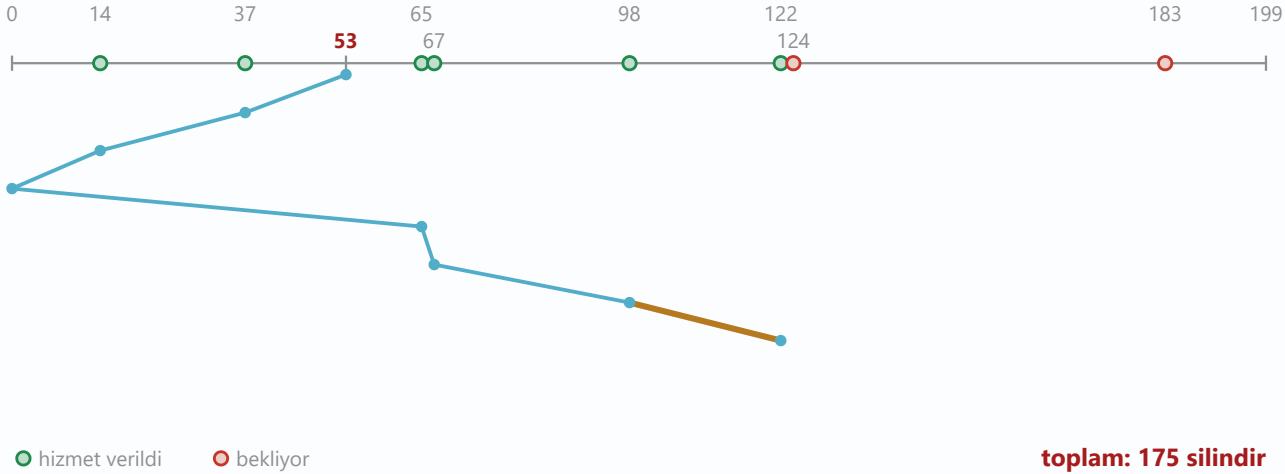


67 → 98: **31** silindir · toplam **151**

Toplam 236. "Açgözlü, kafa bir bölgede sıkışabilir" özelliği SSTF'ye aittir; SCAN açlığı önler. Uca kadar gidip dönmek (14 → 0 → 14) boşa harcanan 28 silindir demektir.

SCAN (Asansör)

SCAN: kafa bir yönde (burada 0'a doğru) uca kadar gider, sonra döner; asansör gibi.

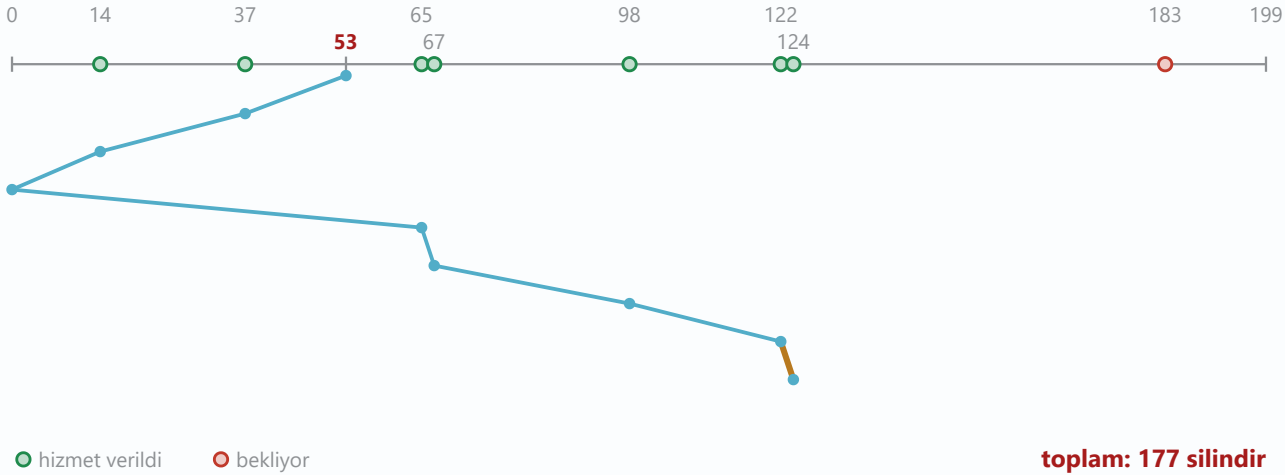


98 → 122: **24** silindir · toplam **175**

Toplam 236. "Açgözlü, kafa bir bölgede sıkışabilir" özelliği SSTF'ye aittir; SCAN açlığı önler. Uca kadar gidip dönmek (14 → 0 → 14) boşa harcanan 28 silindir demektir.

SCAN (Asansör)

SCAN: kafa bir yönde (burada 0'a doğru) uca kadar gider, sonra döner; asansör gibi.

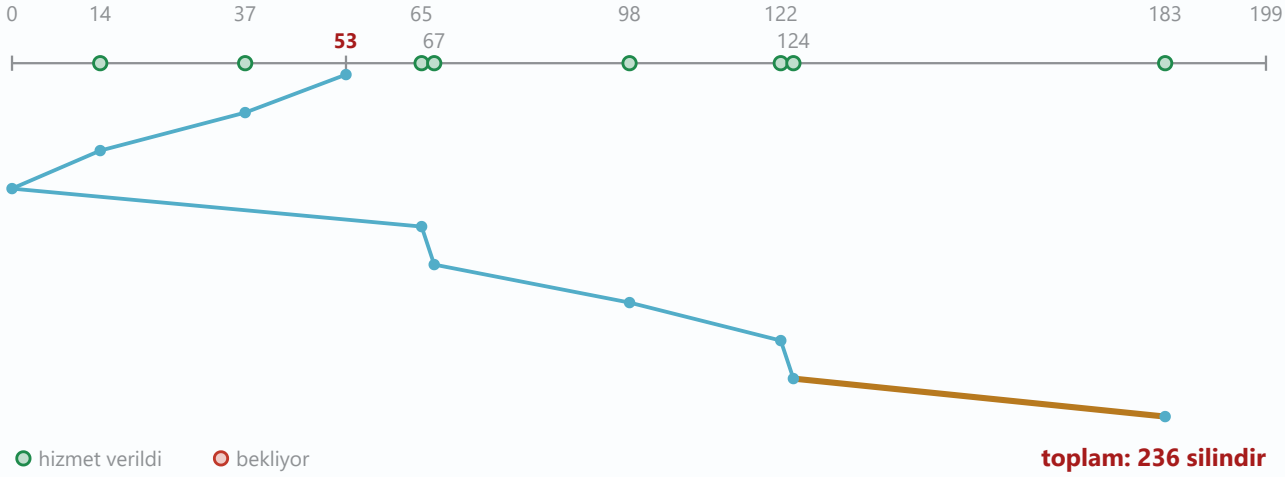


122 → 124: **2** silindir · toplam **177**

Toplam 236. "Açgözlü, kafa bir bölgede sıkışabilir" özelliği SSTF'ye aittir; SCAN açlığı önler. Uca kadar gidip dönmek (14 → 0 → 14) boşa harcanan 28 silindir demektir.

SCAN (Asansör)

SCAN: kafa bir yönde (burada 0'a doğru) uca kadar gider, sonra döner; asansör gibi.



124 → 183: **59** silindir · toplam **236**

Toplam 236. "Açgözlü, kafa bir bölgede sıkışabilir" özelliği SSTF'ye aittir; SCAN açlığı önler. Uca kadar gidip dönmek (14 → 0 → 14) boşa harcanan 28 silindir demektir.

C-SCAN

Dairesel SCAN: yalnız yukarı yönde hizmet; uca varınca hizmet vermeden başa döner.



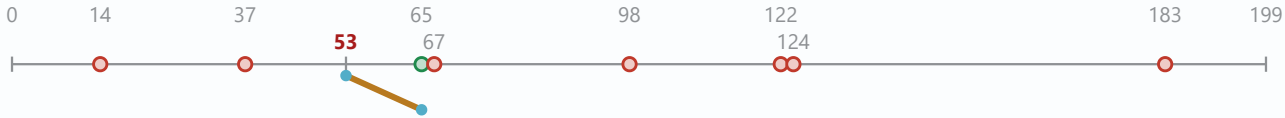
hizmet verildi bekliyor **toplam: 0 silindir**

Kafa **53**'te. Kuyruk: 98, 183, 37, 122, 14, 124, 65, 67 (silindir 0–199). Kırmızı: bekleyen, yeşil: hizmet verilmiş.

Toplam 382 (dönüş sayılmazsa 183). Disk bir halka gibi görülür; bekleme süreleri SCAN'e göre daha eşittir.

C-SCAN

Dairesel SCAN: yalnız yukarı yönde hizmet; uca varınca hizmet vermeden başa döner.



hizmet verildi bekliyor

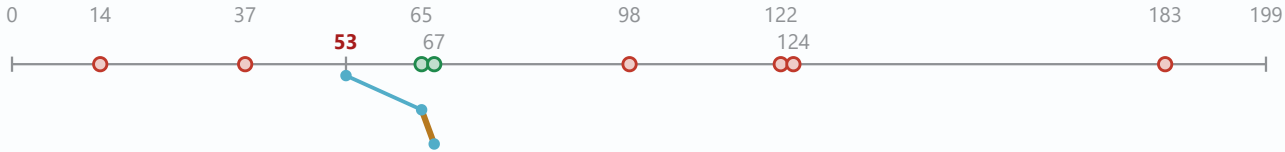
toplam: 12 silindir

53 → 65: **12** silindir · toplam **12**

Toplam 382 (dönüş sayılmazsa 183). Disk bir halka gibi görülür; bekleme süreleri SCAN'e göre daha eşittir.

C-SCAN

Dairesel SCAN: yalnız yukarı yönde hizmet; uca varınca hizmet vermeden başa döner.



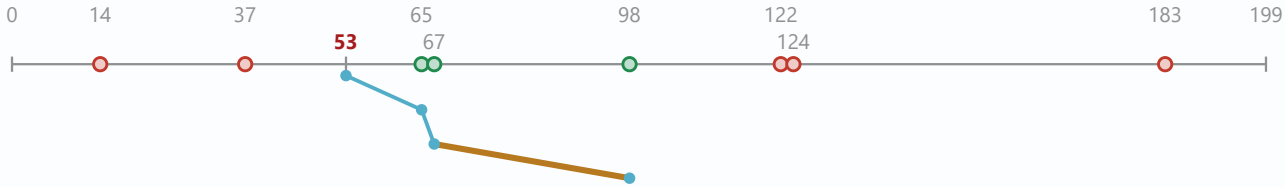
hizmet verildi bekliyor **toplam: 14 silindir**

65 → 67: **2** silindir · toplam **14**

Toplam 382 (dönüş sayılmazsa 183). Disk bir halka gibi görülür; bekleme süreleri SCAN'e göre daha eşittir.

C-SCAN

Dairesel SCAN: yalnız yukarı yönde hizmet; uca varınca hizmet vermeden başa döner.



hizmet verildi bekliyor

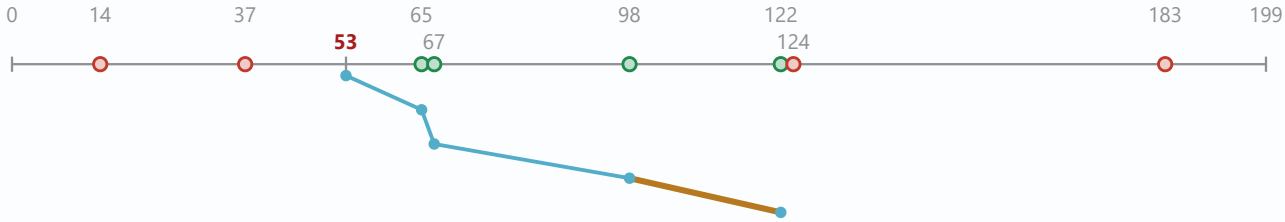
toplam: 45 silindir

67 → 98: **31** silindir · toplam **45**

Toplam 382 (dönüş sayılmazsa 183). Disk bir halka gibi görülür; bekleme süreleri SCAN'e göre daha eşittir.

C-SCAN

Dairesel SCAN: yalnız yukarı yönde hizmet; uca varınca hizmet vermeden başa döner.



hizmet verildi bekliyor

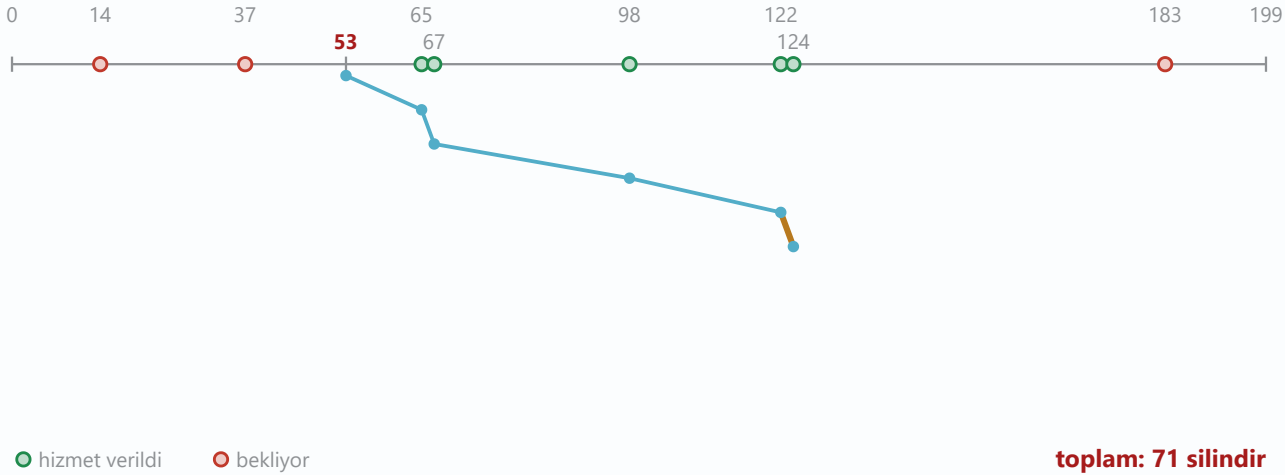
toplam: 69 silindir

98 → 122: **24** silindir · toplam **69**

Toplam 382 (dönüş sayılmazsa 183). Disk bir halka gibi görülür; bekleme süreleri SCAN'e göre daha eşittir.

C-SCAN

Dairesel SCAN: yalnız yukarı yönde hizmet; uca varınca hizmet vermeden başa döner.

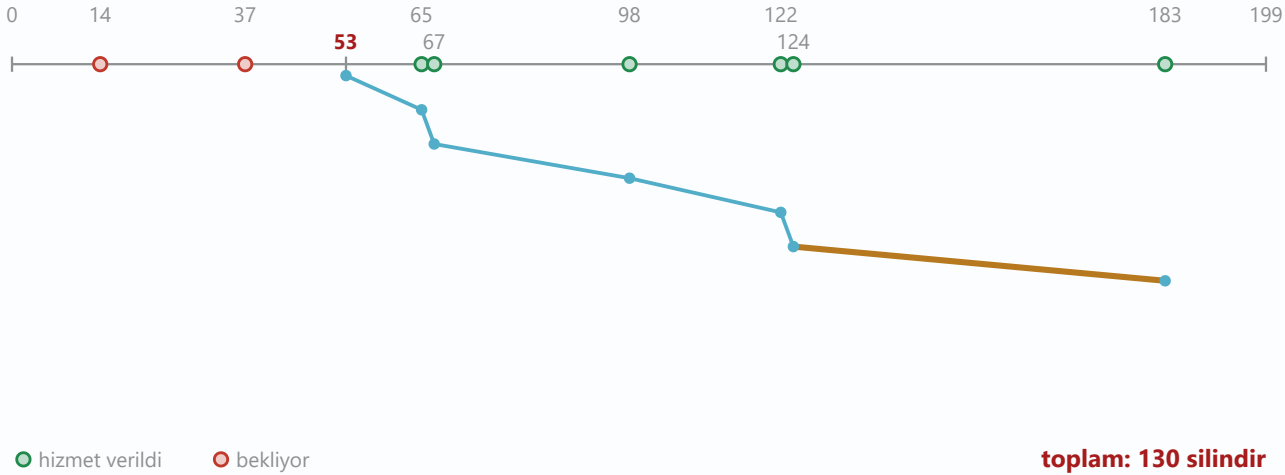


122 → 124: **2** silindir · toplam **71**

Toplam 382 (dönüş sayılmazsa 183). Disk bir halka gibi görülür; bekleme süreleri SCAN'e göre daha eşittir.

C-SCAN

Dairesel SCAN: yalnız yukarı yönde hizmet; uca varınca hizmet vermeden başa döner.

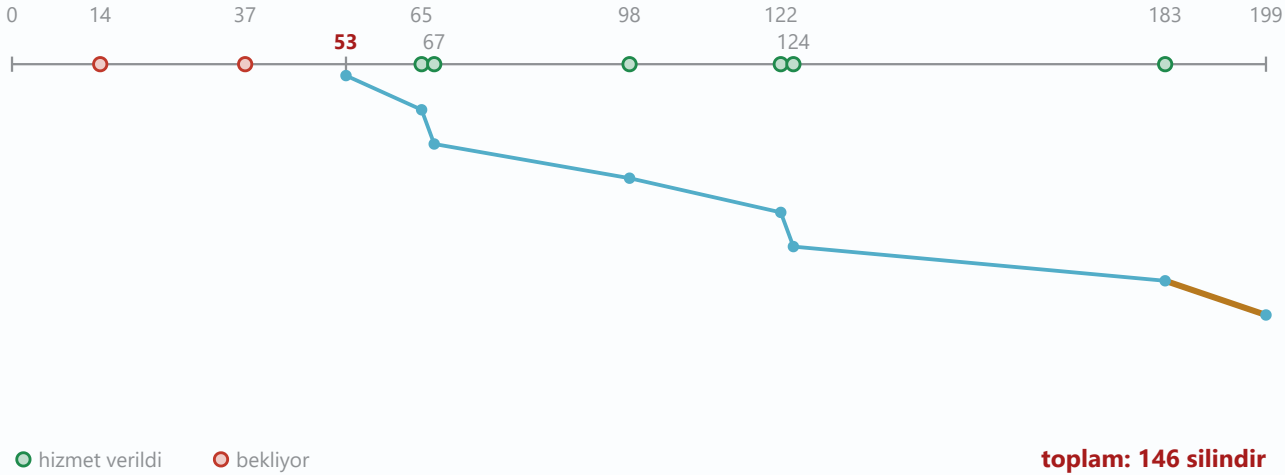


124 → 183: **59** silindir · toplam **130**

Toplam 382 (dönüş sayılmazsa 183). Disk bir halka gibi görülür; bekleme süreleri SCAN'e göre daha eşittir.

C-SCAN

Dairesel SCAN: yalnız yukarı yönde hizmet; uca varınca hizmet vermeden başa döner.

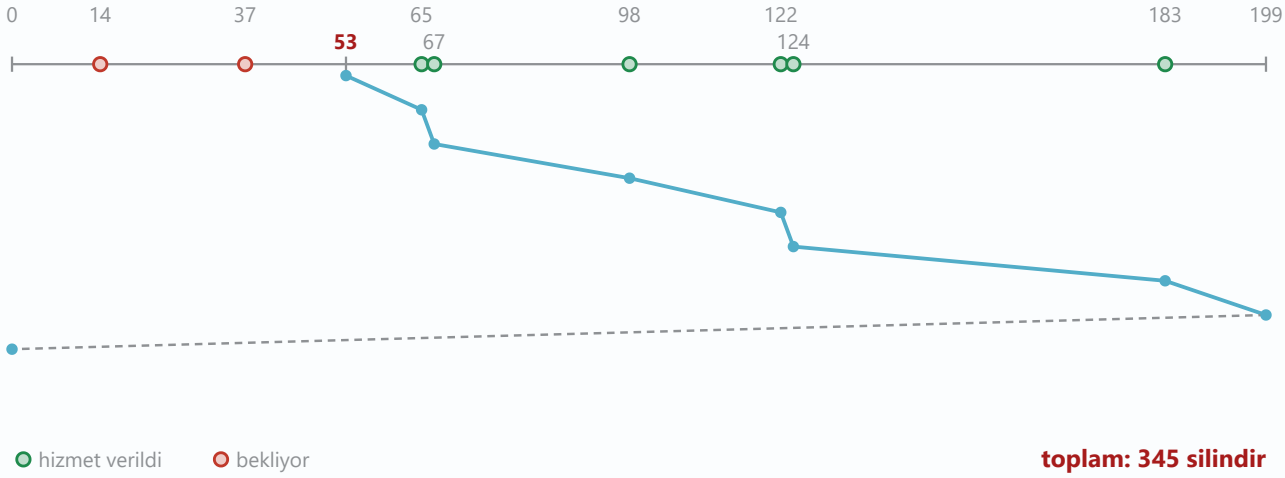


183 → 199: **16** silindir · toplam **146** (istek yok, diskin ucuna kadar gidilir)

Toplam 382 (dönüş sayılmazsa 183). Disk bir halka gibi görülür; bekleme süreleri SCAN'e göre daha eşittir.

C-SCAN

Dairesel SCAN: yalnız yukarı yönde hizmet; uca varınca hizmet vermeden başa döner.

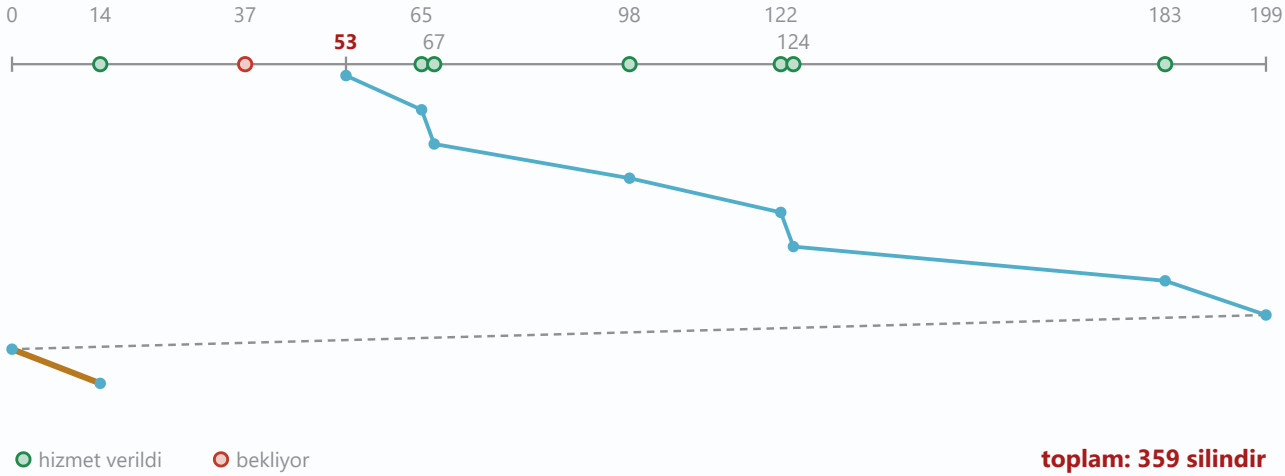


199 → 0: geri dönüş (sıçrama, 199 silindir), istek hizmeti yok. Burada sayılıyor · toplam **345**

Toplam 382 (dönüş sayılmazsa 183). Disk bir halka gibi görülür; bekleme süreleri SCAN'e göre daha eşittir.

C-SCAN

Dairesel SCAN: yalnız yukarı yönde hizmet; uca varınca hizmet vermeden başa döner.

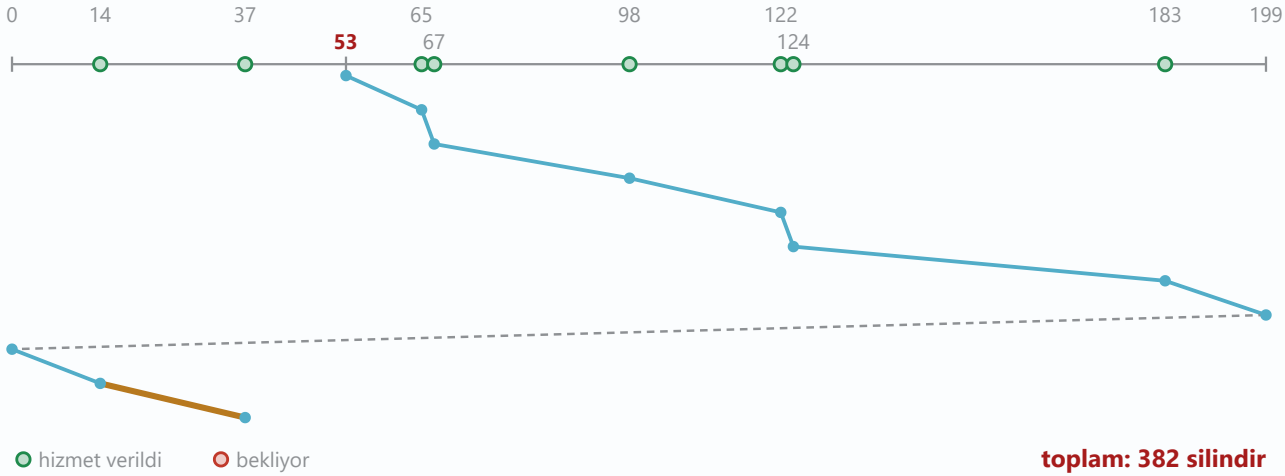


0 → 14: **14** silindir · toplam **359**

Toplam 382 (dönüş sayılmazsa 183). Disk bir halka gibi görülür; bekleme süreleri SCAN'e göre daha eşittir.

C-SCAN

Dairesel SCAN: yalnız yukarı yönde hizmet; uca varınca hizmet vermeden başa döner.



14 → 37: **23** silindir · toplam **382**

Toplam 382 (dönüş sayılmazsa 183). Disk bir halka gibi görülür; bekleme süreleri SCAN'e göre daha eşittir.

LOOK ve C-LOOK



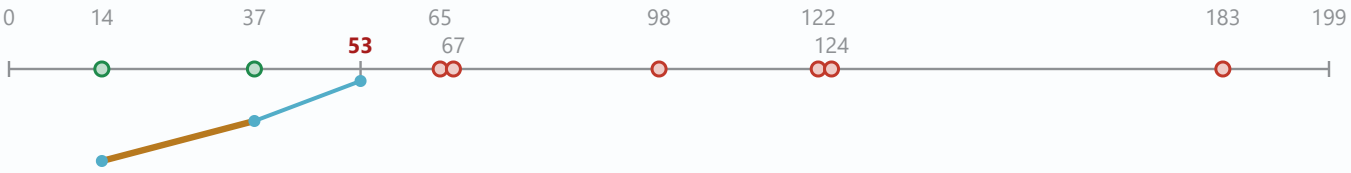
hizmet verildi bekliyor **toplam: 0 silindir**

LOOK (aşağı yönde): SCAN gibi, ama uca kadar gitmez; o yöndeki son istekte döner.

Adım 1 / 6

Linux'un eski *deadline* ve *CFQ* çizelgeleyicileri ile günümüzdeki *mq-deadline* LBA'ya göre sıralı gönderir; bu C-LOOK'a benzer bir davranıştır.

LOOK ve C-LOOK

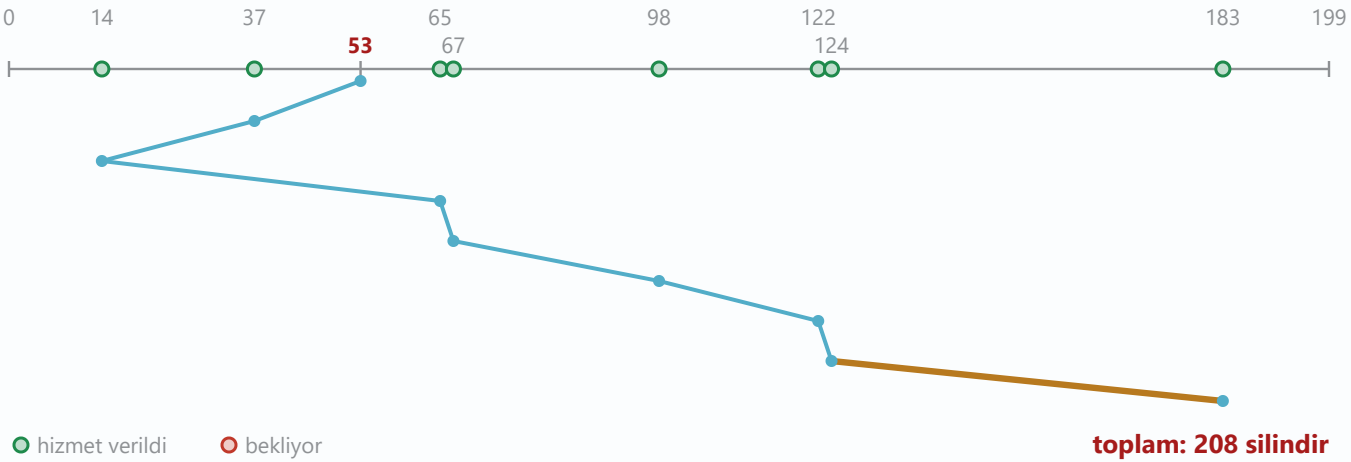


hizmet verildi bekliyor **toplam: 39 silindir**

53 → 37 → 14: aşağıdaki istekler bitti, 0'a gidilmez. Toplam 39.

Linux'un eski *deadline* ve *CFQ* çizelgeleyicileri ile günümüzdeki *mq-deadline* LBA'ya göre sıralı gönderir; bu C-LOOK'a benzer bir davranıştır.

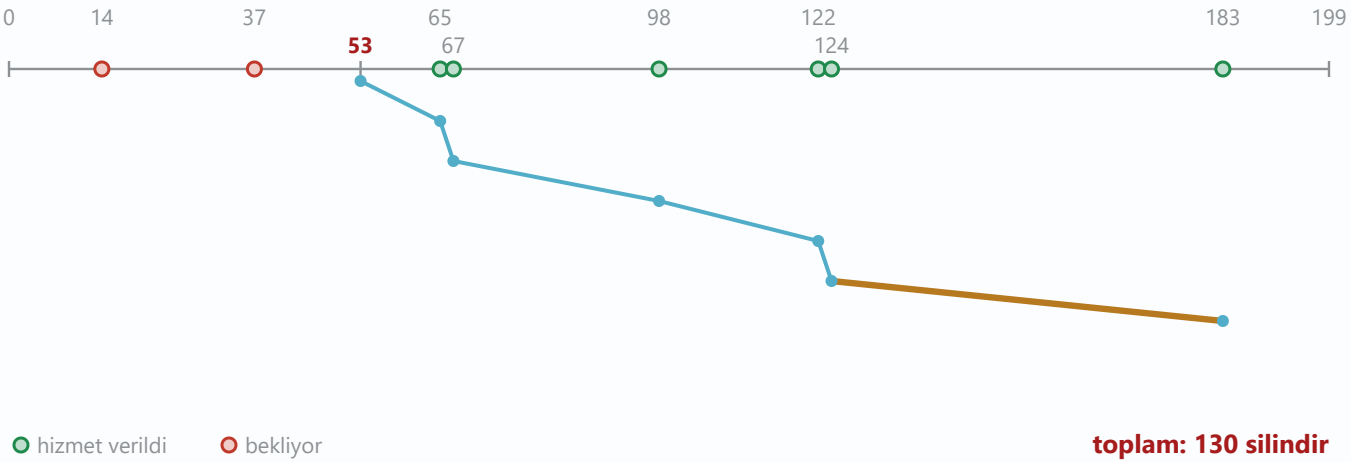
LOOK ve C-LOOK



14 → 65 … 183: yön değişir. **Toplam 208** (SCAN'e göre 28 az). Uygulamada "asansör" denen davranış çoğunlukla bu LOOK'tur.

Linux'un eski *deadline* ve *CFQ* çizelgeleyicileri ile günümüzdeki *mq-deadline* LBA'ya göre sıralı gönderir; bu C-LOOK'a benzer bir davranıştır.

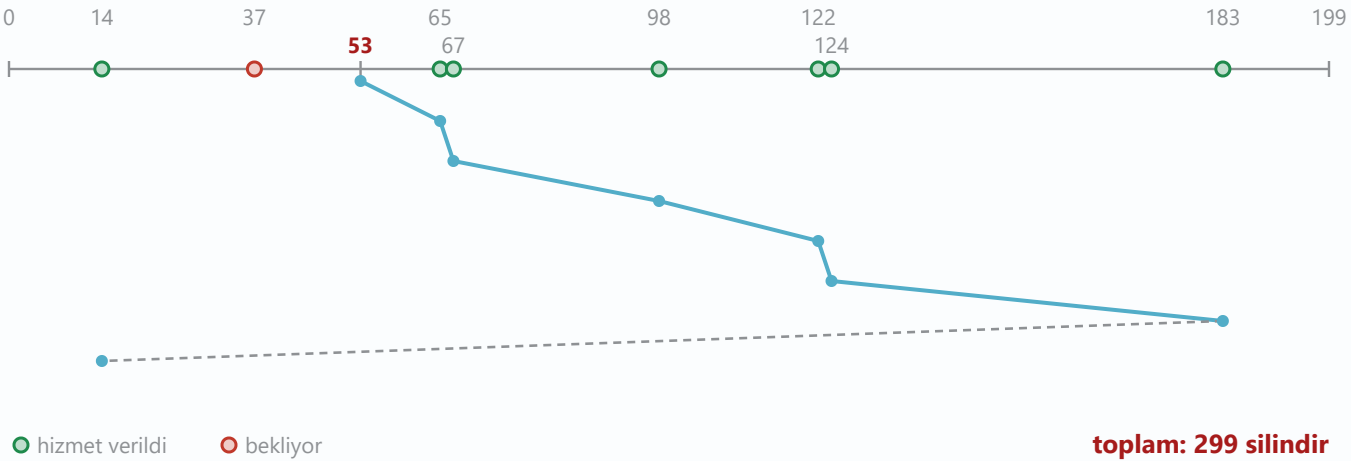
LOOK ve C-LOOK



C-LOOK (yukarı yönde): 53 → 65 … 183 yukarıdaki son istek. Toplam 130.

Linux'un eski *deadline* ve *CFQ* çizelgeleyicileri ile günümüzdeki *mq-deadline* LBA'ya göre sıralı gönderir; bu C-LOOK'a benzer bir davranıştır.

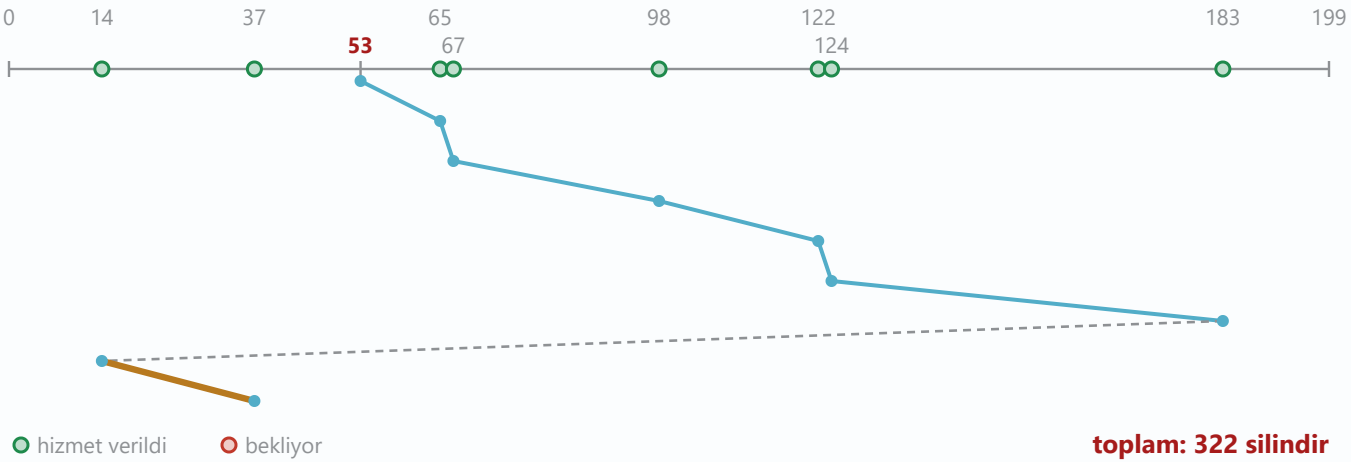
LOOK ve C-LOOK



183 → 14: en küçük bekleyen isteğe sıçrar (169 silindir), 199 ve 0'a gidilmez.

Linux'un eski *deadline* ve *CFQ* çizelgeleyicileri ile günümüzdeki *mq-deadline* LBA'ya göre sıralı gönderir; bu C-LOOK'a benzer bir davranıştır.

LOOK ve C-LOOK

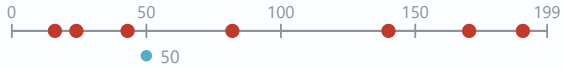


14 → 37: **toplam 322** (sıçrama sayılmazsa 153). Tek yönlü tarama bekleme sürelerini eşitler.

Linux'un eski *deadline* ve *CFQ* çizelgeleyicileri ile günümüzdeki *mq-deadline* LBA'ya göre sıralı gönderir; bu C-LOOK'a benzer bir davranıştır.

LOOK: Kod İzleme

```
1  /* s[] sıralı; önce yukarı, sonra aşağı */
2  int look(int bas, int s[], int n) {
3    int i = 0, top = 0;
4    while (i < n && s[i] < bas) i++;
5    for (int j = i; j < n; j++) {
6      top += s[j] - bas; bas = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      top += bas - s[j]; bas = s[j];
10   }
11   return top;
12 }
```



toplam = 0 silindir

Aynı kuyruk sıralanmış: s = 16, 24, 43, 82, 140, 170, 190. Kafa 50'de ve **yukarı** yönde hareket ediyor.

Adım 1 / 10

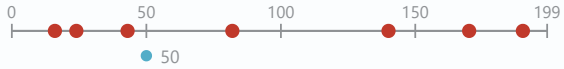
DEĞİŞKENLER

`bas = 50` `i = 0` `j = –` `top = 0`

ÇIKTI

LOOK: Kod İzleme

```
1  /* s[] sıralı; önce yukarı, sonra aşağı */
2  int look(int bas, int s[], int n) {
3    int i = 0, top = 0;
4    while (i < n && s[i] < bas) i++;
5    for (int j = i; j < n; j++) {
6      top += s[j] - bas; bas = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      top += bas - s[j]; bas = s[j];
10   }
11   return top;
12 }
```



toplam = 0 silindir

i, 50'den büyük ilk isteğin indisi: s[3] = 82. Altındaki 16, 24, 43 dönüşte alınacak.

Adım 2 / 10

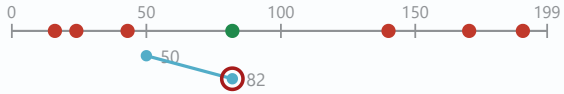
DEĞİŞKENLER

`bas = 50` `i = 3` `j = –` `top = 0`

ÇIKTI

LOOK: Kod İzleme

```
1  /* s[] sıralı; önce yukarı, sonra aşağı */
2  int look(int bas, int s[], int n) {
3    int i = 0, top = 0;
4    while (i < n && s[i] < bas) i++;
▶    for (int j = i; j < n; j++) {
▶      top += s[j] - bas; bas = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      top += bas - s[j]; bas = s[j];
10   }
11   return top;
12 }
```



toplam = 32 silindir

j = 3: 82 (+32), top = 32.

Adım 3 / 10

DEĞİŞKENLER

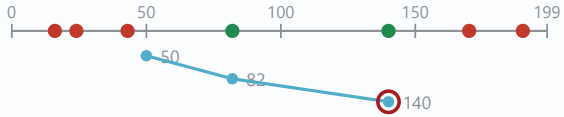
`bas = 82` `i = 3` `j = 3` `top = 32`

ÇIKTI

```
82
```

LOOK: Kod İzleme

```
1  /* s[] sıralı; önce yukarı, sonra aşağı */
2  int look(int bas, int s[], int n) {
3    int i = 0, top = 0;
4    while (i < n && s[i] < bas) i++;
▶    for (int j = i; j < n; j++) {
▶      top += s[j] - bas; bas = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      top += bas - s[j]; bas = s[j];
10   }
11   return top;
12 }
```



toplam = 90 silindir

j = 4: 140 (+58), top = 90.

Adım 4 / 10

DEĞİŞKENLER

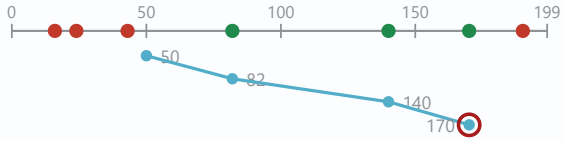
bas = 140 | i = 3 | j = 4 | top = 90

ÇIKTI

```
82 140
```

LOOK: Kod İzleme

```
1  /* s[] sıralı; önce yukarı, sonra aşağı */
2  int look(int bas, int s[], int n) {
3    int i = 0, top = 0;
4    while (i < n && s[i] < bas) i++;
▶    for (int j = i; j < n; j++) {
▶      top += s[j] - bas; bas = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      top += bas - s[j]; bas = s[j];
10   }
11   return top;
12 }
```



toplam = 120 silindir

j = 5: 170 (+30), top = 120.

Adım 5 / 10

DEĞİŞKENLER

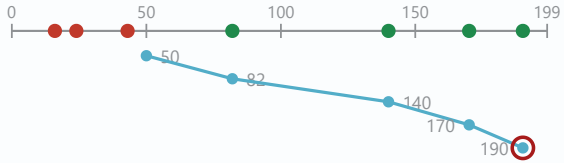
bas = **170** | i = **3** | j = **5** | top = **120**

ÇIKTI

```
82 140 170
```

LOOK: Kod İzleme

```
1  /* s[] sıralı; önce yukarı, sonra aşağı */
2  int look(int bas, int s[], int n) {
3    int i = 0, top = 0;
4    while (i < n && s[i] < bas) i++;
▶    for (int j = i; j < n; j++) {
▶      top += s[j] - bas; bas = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      top += bas - s[j]; bas = s[j];
10   }
11   return top;
12 }
```



toplam = 140 silindir

j = 6: 190 (+20), top = 140. Yukarıdaki son istek: kol 199'a **gitmeden** döner (SCAN'den farkı).

Adım 6 / 10

DEĞİŞKENLER

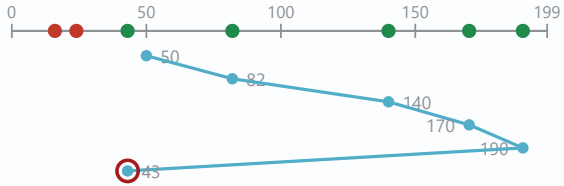
`bas = 190` `i = 3` `j = 6` `top = 140`

ÇIKTI

```
82 140 170 190
```

LOOK: Kod İzleme

```
1  /* s[] sıralı; önce yukarı, sonra aşağı */
2  int look(int bas, int s[], int n) {
3    int i = 0, top = 0;
4    while (i < n && s[i] < bas) i++;
5    for (int j = i; j < n; j++) {
6      top += s[j] - bas; bas = s[j];
7    }
▶    for (int j = i - 1; j >= 0; j--) {
▶      top += bas - s[j]; bas = s[j];
10   }
11   return top;
12 }
```



toplam = 287 silindir

Yön değişti, j = 2: 190 → 43 (+147), top = 287. Dönüşte en uzun hareket.

Adım 7 / 10

DEĞİŞKENLER

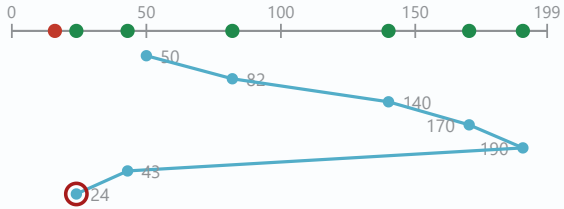
`bas = 43` `i = 3` `j = 2` `top = 287`

ÇIKTI

```
82 140 170 190 43
```

LOOK: Kod İzleme

```
1  /* s[] sıralı; önce yukarı, sonra aşağı */
2  int look(int bas, int s[], int n) {
3    int i = 0, top = 0;
4    while (i < n && s[i] < bas) i++;
5    for (int j = i; j < n; j++) {
6      top += s[j] - bas; bas = s[j];
7    }
▶    for (int j = i - 1; j >= 0; j--) {
▶      top += bas - s[j]; bas = s[j];
10   }
11   return top;
12 }
```



toplam = 306 silindir

j = 1: 24 (+19), top = 306.

Adım 8 / 10

DEĞİŞKENLER

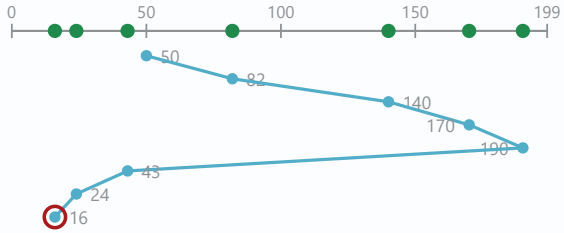
bas = 24 | i = 3 | j = 1 | top = 306

ÇIKTI

```
82 140 170 190 43 24
```

LOOK: Kod İzleme

```
1  /* s[] sıralı; önce yukarı, sonra aşağı */
2  int look(int bas, int s[], int n) {
3    int i = 0, top = 0;
4    while (i < n && s[i] < bas) i++;
5    for (int j = i; j < n; j++) {
6      top += s[j] - bas; bas = s[j];
7    }
▶    for (int j = i - 1; j >= 0; j--) {
▶      top += bas - s[j]; bas = s[j];
10   }
11   return top;
12 }
```



toplam = 314 silindir

j = 0: 16 (+8), top = 314.

Adım 9 / 10

DEĞİŞKENLER

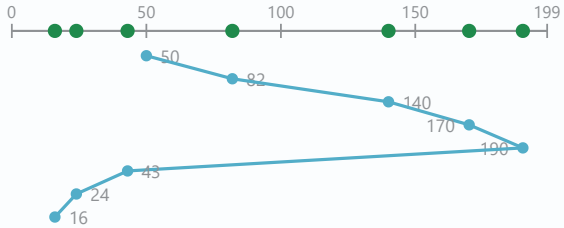
bas = 16 | i = 3 | j = 0 | top = 314

ÇIKTI

```
82 140 170 190 43 24 16
```

LOOK: Kod İzleme

```
1  /* s[] sıralı; önce yukarı, sonra aşağı */
2  int look(int bas, int s[], int n) {
3    int i = 0, top = 0;
4    while (i < n && s[i] < bas) i++;
5    for (int j = i; j < n; j++) {
6      top += s[j] - bas; bas = s[j];
7    }
8    for (int j = i - 1; j >= 0; j--) {
9      top += bas - s[j]; bas = s[j];
10   }
▶    return top;
12 }
```



toplam = 314 silindir

Toplam **314**: aynı kuyrukta SSTF 208. Kafa aşağı yönde olsaydı LOOK da 34 + 174 = 208 yapardı; yön seçimi sonucu belirler ama hiçbir istek aç kalmaz.

Adım 10 / 10

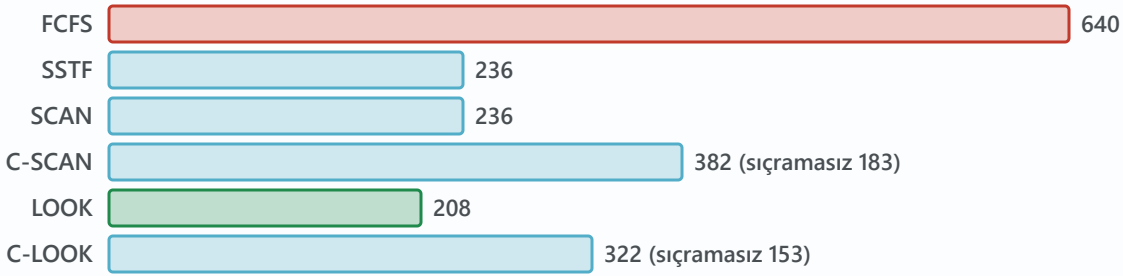
DEĞİŞKENLER

`bas = 16` `i = 3` `j = −1` `top = 314`

ÇIKTI

```
82 140 170 190 43 24 16
```

Algoritmaların Karşılaştırması

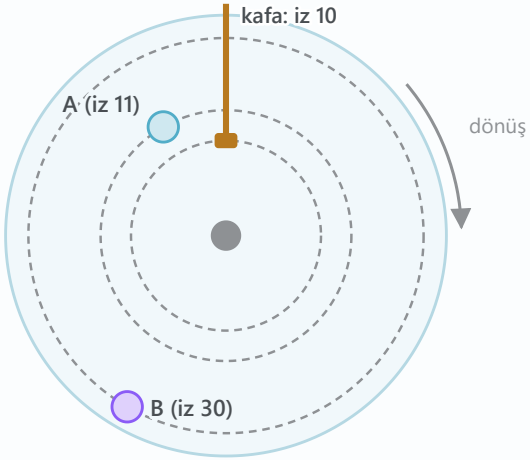


Tanenbaum örneği: kafa 11; 1, 36, 16, 34, 9, 12	**Hareket**
FCFS: 10 + 35 + 20 + 18 + 25 + 3	**111**
SSTF: 12, 9, 16, 1, 34, 36 → 1 + 3 + 7 + 15 + 33 + 2	**61**
Asansör (yukarı): 12, 16, 34, 36, 9, 1 → 1 + 4 + 18 + 2 + 27 + 8	**60**

Etkileşimli simülatör: Disk Kolu Çizelgeleme Simülatörü: farklı istek kuyrukları ve başlangıç yönleriyle altı algoritmanın karşılaştırması.

Grafikteki kuyrukta SCAN yukarı yönde başlasaydı 331 olurdu: yön seçimi sonucu değiştirir. Tanenbaum'un asansörü uca gitmediği için aslında LOOK'tur.

SPTF: Dönme Gecikmesini de Saymak



7200 rpm: 1 tur = 8,33 ms, 1 ms'de disk 43,2° döner

A: arama 1 ms, sektör kafaya 30° sonra gelir
B: arama 3 ms, sektör kafaya 150° sonra gelir

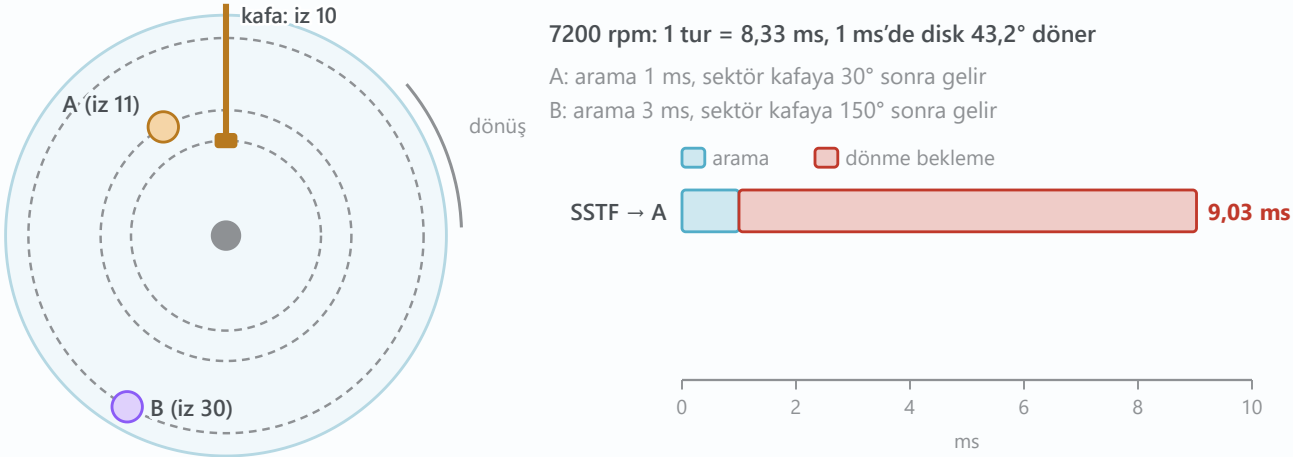


Kafa iz 10'da, iki istek bekliyor. Yalnız **iz farkına** bakan SSTF için A açık ara yakın: 1 iz ötede. Ama süreyi arama değil **arama + dönme** belirler.

Adım 1 / 3

SPTF (shortest positioning time first) konumlandırma süresini en aza indirir, ama İS sektörlerin açısını bilmez; bu yüzden diskin içinde NCQ ile uygulanır. Açlığa karşı **F-SCAN**: tarama başlarken kuyruk dondurulur, yeni istekler bir sonraki taramayı bekler.

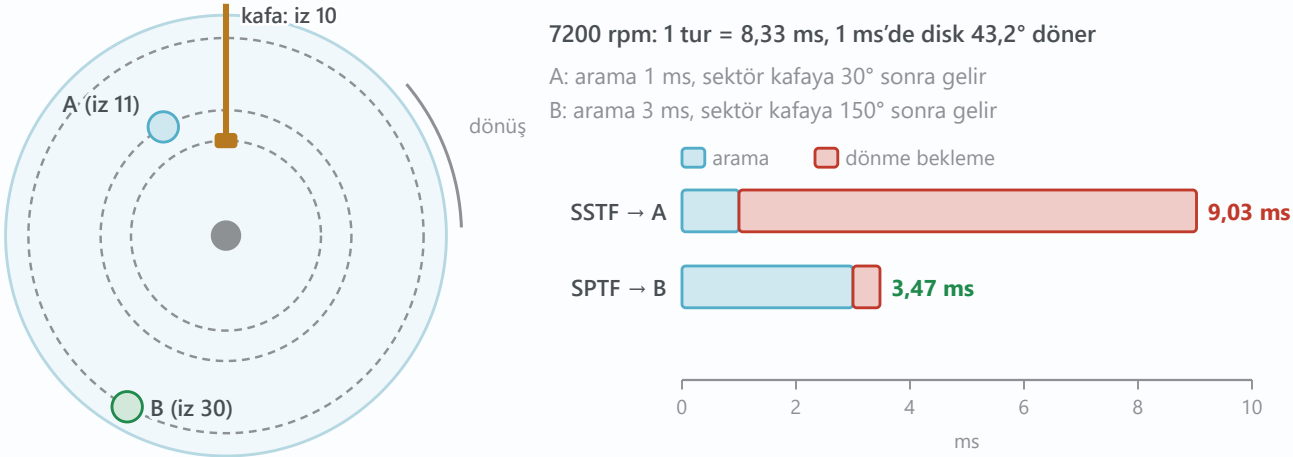
SPTF: Dönme Gecikmesini de Saymak



SSTF A'yı seçer: 1 ms aramada disk 43,2° döner, kafaya 30° kalan sektör arama bitince kafayı 13,2° geçmiş olur. Neredeyse bir tur beklenir: 1 + 8,03 = **9,03 ms**.

SPTF (shortest positioning time first) konumlandırma süresini en aza indirir, ama İS sektörlerin açısını bilmez; bu yüzden diskin içinde NCQ ile uygulanır. Açlığa karşı **F-SCAN**: tarama başlarken kuyruk dondurulur, yeni istekler bir sonraki taramayı bekler.

SPTF: Dönme Gecikmesini de Saymak



SPTF B'yi seçer: 3 ms aramada disk 129,6° döner, B'ye 20,4° kalır: 3 + 0,47 = **3,47 ms**. Uzak iz, yakın izden 2,6 kat çabuk sonuç verdi.

Adım 3 / 3

SPTF (shortest positioning time first) konumlandırma süresini en aza indirir, ama İS sektörlerin açısını bilmez; bu yüzden diskin içinde NCQ ile uygulanır. Açlığa karşı **F-SCAN**: tarama başlarken kuyruk dondurulur, yeni istekler bir sonraki taramayı bekler.

Denetleyici Önbelleği ve Bozuk Sektörler

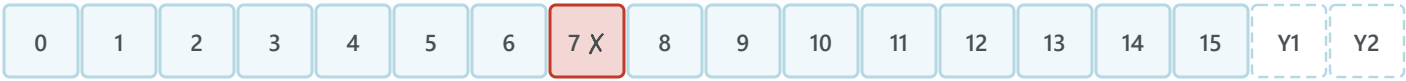
Denetleyici önbelleği

Disk kendi DRAM'inde (≈ 256 MB) iz önbelleği tutar: kafanın altından geçen, istenmemiş blokları da okur. İşletim sistemi önbelleğinden ayrıdır.

Yazma önbelleği ve NCQ

Yazma önbellekte bekleyebilir: kalıcılık için `FLUSH` /FUA gerekir. NCQ 32 isteği dönme konumuna göre sıralar.

(a) iz: sektör 7 bozuk, iki yedek



Üretimde ya da kullanımda bozulan sektör: yazılan veri geri okunamaz. Denetleyici fabrika listesini ve kullanımda bulduklarını izler (SMART: *Reallocated Sectors*).

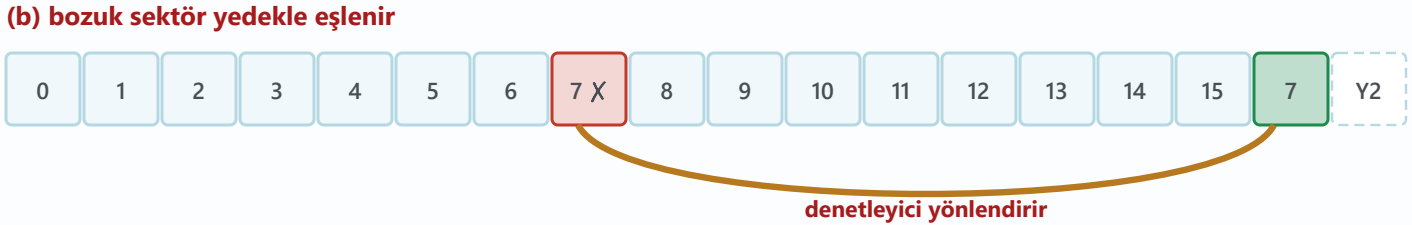
Denetleyici Önbelleği ve Bozuk Sektörler

Denetleyici önbelleği

Disk kendi DRAM'inde (≈ 256 MB) iz önbelleği tutar: kafanın altından geçen, istenmemiş blokları da okur. İşletim sistemi önbelleğinden ayrıdır.

Yazma önbelleği ve NCQ

Yazma önbellekte bekleyebilir: kalıcılık için `FLUSH` /FUA gerekir. NCQ 32 isteği dönme konumuna göre sıralar.



Yöntem 1: bozuk sektör bir yedekle **eşlenir**. Basit ama sektör 7'ye giden ardışık okuma ize bir tur daha ödeyebilir.

Denetleyici Önbelleği ve Bozuk Sektörler

Denetleyici önbelleği

Disk kendi DRAM'inde (≈ 256 MB) iz önbelleği tutar: kafanın altından geçen, istenmemiş blokları da okur. İşletim sistemi önbelleğinden ayrıdır.

Yazma önbelleği ve NCQ

Yazma önbellekte bekleyebilir: kalıcılık için `FLUSH` /FUA gerekir. NCQ 32 isteği dönme konumuna göre sıralar.

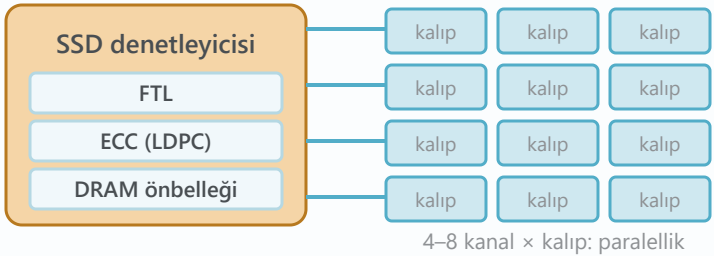
(c) sektörler kaydırılır, bozuk atlanır



8 … 15 numaralı sektörler bir kaydı, bir yedek kullanıldı; sıralı okuma bozulmaz

Yöntem 2: sektörler **kaydırılır**; tüm iz yeniden yazılmalıdır ama sıra korunur. İşletim sistemi de bozuk blok listesi tutabilir (`badblocks` , dosya sistemi).

SSD: NAND Flash

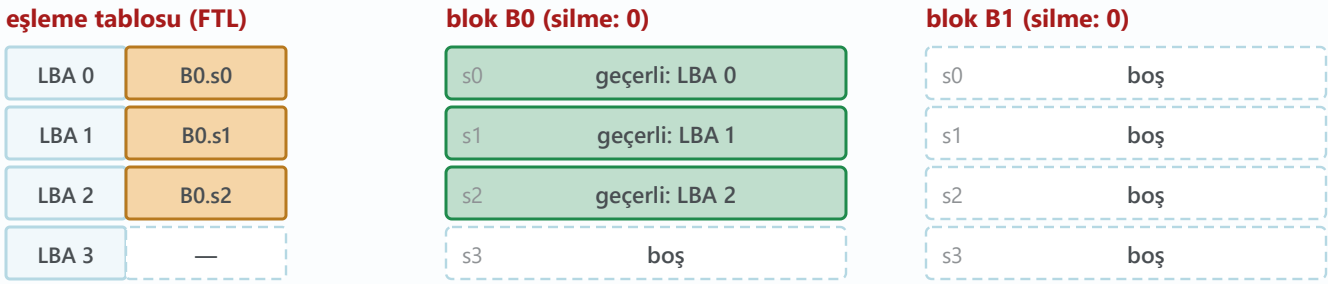




Hücre	Bit/hücre	Tipik P/E döngüsü	Kullanım
SLC	1	≈ 50–100 bin	önbellek bölgesi, endüstriyel
MLC	2	≈ 3–10 bin	eski kurumsal
TLC	3	≈ 1–3 bin	çoğu tüketici ve sunucu SSD'si
QLC	4	≈ 0,1–1 bin	yüksek kapasite, okuma ağırlıklı

- Hareketli parça yok: **arama ve dönme gecikmesi yok**, rastgele erişim ≈ ardışık erişim.
- Yerinde güncelleme yok: *sil, sonra yaz*; her hücrenin silme ömrü sınırlıdır. 3D NAND 200–300+ katman.
- Fiyat/GB farkı azaldı; HDD yine de arşivde ve kapasitede ucuzdur.

FTL, Çöp Toplama, TRIM, Aşınma Dengeleme



Üç mantıksal blok yazılır; FTL onları B0'ın boş sayfalarına koyar ve tabloya kaydeder.

Adım 1 / 6

Windows "İyileştir" komutu SSD'de birleştirme yapmaz, TRIM gönderir; SSD'yi birleştirmek yalnızca yazma ve aşınma ekler. Bölümleri 1 MiB sınırına hizalamak, mantıksal blokların flaş sayfalarına bölünmeden oturmasını sağlar.

FTL, Çöp Toplama, TRIM, Aşınma Dengeleme



LBA 1 güncellendi: yerinde değil

yalnız silinmiş (boş) sayfaya yazılabilir; silme blok düzeyinde

LBA 1 yeniden yazılır: eski sayfanın üzerine yazılamaz. Yeni veri boş sayfaya gider, eskisi **geçersiz** olur (yer dışı güncelleme).

Windows “İyileştir” komutu SSD’de birleştirme yapmaz, TRIM gönderir; SSD’yi birleştirmek yalnızca yazma ve aşınma ekler. Bölümleri 1 MiB sınırına hizalamak, mantıksal blokların flaş sayfalarına bölünmeden oturmasını sağlar.

FTL, Çöp Toplama, TRIM, Aşınma Dengeleme



Dosya silinince dosya sistemi **TRIM** (NVMe: Deallocate) gönderir; SSD sayfanın gereksiz olduğunu öğrenir. TRIM olmasa bu veri çöp toplamada boşuna kopyalanırdı.

Windows “İyileştir” komutu SSD’de birleştirme yapmaz, TRIM gönderir; SSD’yi birleştirmek yalnızca yazma ve aşınma ekler. Bölümleri 1 MiB sınırına hizalamak, mantıksal blokların flaş sayfalarına bölünmeden oturmasını sağlar.

FTL, Çöp Toplama, TRIM, Aşınma Dengeleme

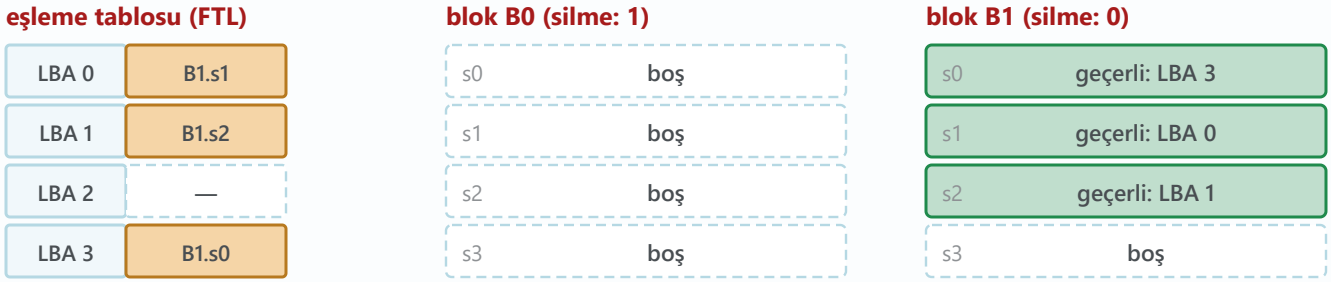


yalnız silinmiş (boş) sayfaya yazılabilir; silme blok düzeyinde

B0'da boş sayfa kalmadı; yeni yazmalar B1'e gider. B0'da yarısı geçersiz.

Windows "İyileştir" komutu SSD'de birleştirme yapmaz, TRIM gönderir; SSD'yi birleştirmek yalnızca yazma ve aşınma ekler. Bölümleri 1 MiB sınırına hizalamak, mantıksal blokların flaş sayfalarına bölünmeden oturmasını sağlar.

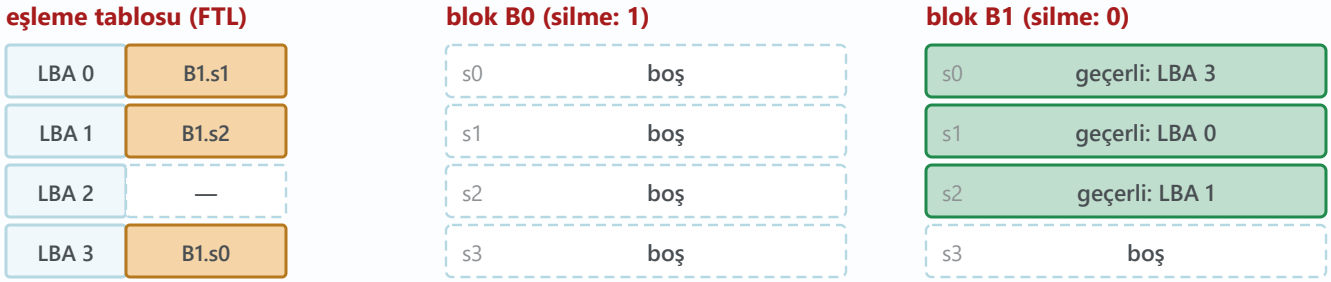
FTL, Çöp Toplama, TRIM, Aşınma Dengeleme



Çöp toplama: B0'daki geçerli sayfalar (LBA 0, 1) B1'e kopyalanır, B0 silinir. Ek 2 sayfa yazması = **yazma büyütmesi**.

Windows "İyileştir" komutu SSD'de birleştirme yapmaz, TRIM gönderir; SSD'yi birleştirmek yalnızca yazma ve aşınma ekler. Bölümleri 1 MiB sınırına hizalamak, mantıksal blokların flaş sayfalarına bölünmeden oturmasını sağlar.

FTL, Çöp Toplama, TRIM, Aşınma Dengeleme



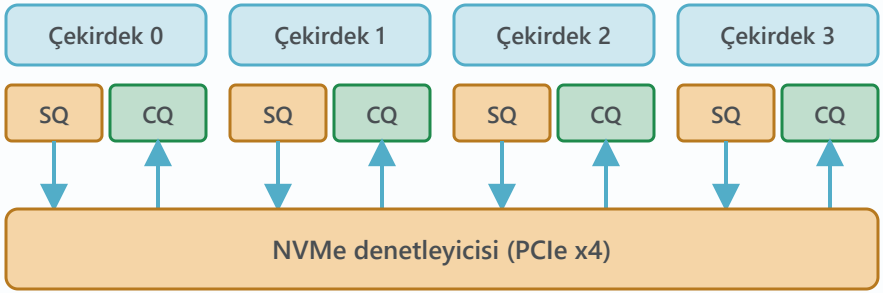
aşınma dengeleme: az silinen bloklar da kullanılır

yalnız silinmiş (boş) sayfaya yazılabilir; silme blok düzeyinde

Aşınma dengeleme: FTL silme sayılarını dengeler; uzun süre değişmeyen (soğuk) veriyi de taşıyarak tüm blokları eşit yaşlandırır. Yedek alan (over-provisioning) çöp toplamayı kolaylaştırır.

Windows “İyileştir” komutu SSD’de birleştirme yapmaz, TRIM gönderir; SSD’yi birleştirmek yalnızca yazma ve aşınma ekler. Bölümleri 1 MiB sınırına hizalamak, mantıksal blokların flaş sayfalarına bölünmeden oturmasını sağlar.

NVMe ve Çoklu Kuyruk



AHCI (SATA)

tek kuyruk · 32 komut · kilit

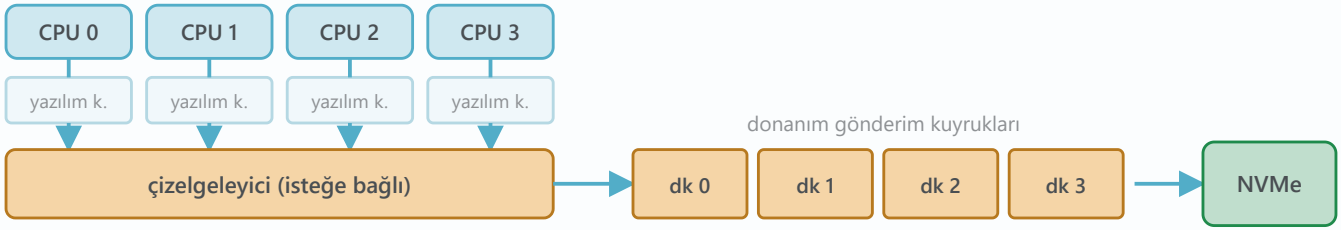
NVMe

65.535 kuyruk × 65.536 komut

çekirdek başına kuyruk: kilitsiz gönderim

- NVMe, PCIe üzerindeki flash için tasarlanmış **protokoldür**; M.2 ise bir *form faktörüdür* (M.2 kart SATA da olabilir). U.2/U.3 ve E1.S/E3.S kablolu/yuvalı sunucu biçimleridir.
- Kuyruk kapasitesi: 64 K kuyruk × kuyruk başına 64 K komut. Sürücü desteği 2013'ten beri tüm işletim sistemlerinde yerleşiktir; AHCI ile uyumsuzluk yalnız protokol düzeyindedir.
- Gecikme ≈ 10–100 µs; PCIe 5.0 x4 sürücüler ≈ 14 GB/s ve > 2 milyon IOPS. NVMe-oF ile ağ üzerinden, ZNS ile bölgeli ad alanları.

Linux blk-mq ve G/Ç Çizelgeleyicileri



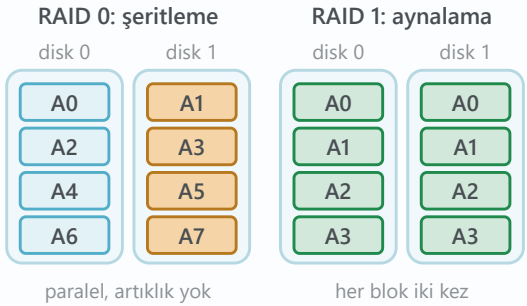
Çizelgeleyici	Fikir	Uygun
`none`	Sıralama yok, doğrudan gönder	NVMe (varsayılan), hızlı SSD
`mq-deadline`	LBA'ya göre sıralı + okuma/yazma son tarihi	SATA HDD/SSD (çoğu dağıtımda varsayılan)
`bfq`	Süreç başına bütçe, adil paylaşım	Masaüstü, yavaş aygıtlarda etkileşim
`kyber`	Okuma/yazma gecikme hedefleriyle sınırlama	Hızlı çok kuyruklu aygıtlar

```
$ cat /sys/block/nvme0n1/queue/scheduler
[none] mq-deadline kyber bfq
$ echo bfq | sudo tee /sys/block/sda/queue/scheduler
```

CPU başına yazılım kuyrukları, milyonlarca IOPS'de tek kuyruk kilidinin çekişmesini önler; Linux 5.0'da tek kuyruklu eski blok katmanı (noop, deadline, CFQ) kaldırıldı. SSD'de arama olmadığından çizelgeleyicinin işi sıralamak değil, adaleti ve gecikmeyi korumaktır.

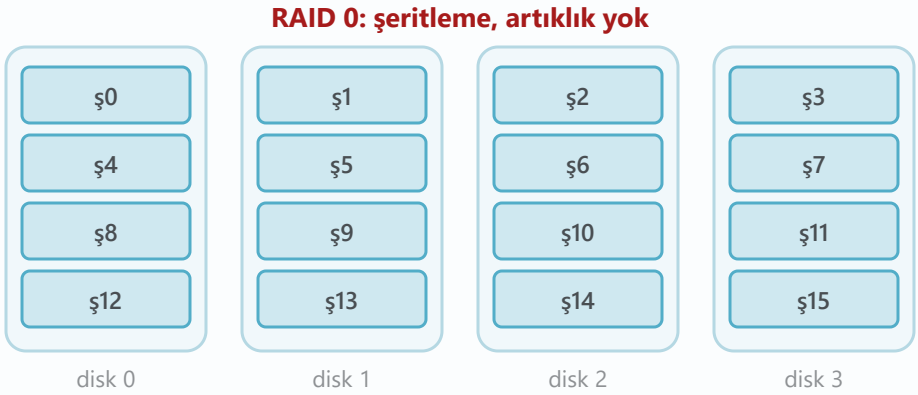
RAID: Diskleri Birleştirmek

- **RAID** (Patterson, Gibson, Katz, 1988): birçok diski işletim sistemine *tek disk* gibi gösterir; **şeritleme** paralellik, **aynalama** ve **eşlik** artıklık sağlar. Karşıtı SLED: tek, büyük, pahalı disk.
- Açılımı bugün *Redundant Array of **Independent** Disks*. SCSI diskler pahalıydı; bugün SAS genişleticiler yüzlerce disk bağlar, standart düzeyler 0–6'dır.
- Donanım RAID kartı ya da yazılım RAID (Linux `mdadm`, ZFS RAID-Z, Btrfs, Windows Depolama Alanları). **RAID yedek değildir**: silinen dosya her diskte silinir.



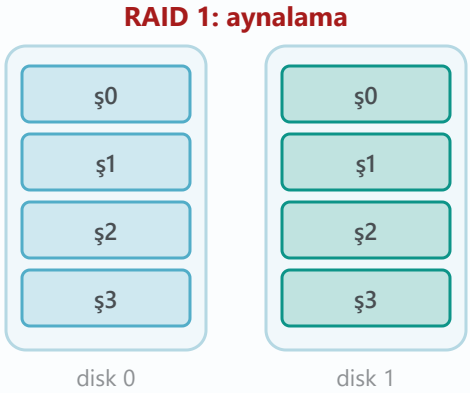
4 × 4 TB disk	Kullanılabilir	Dayanır	Küçük rastgele yazma maliyeti
RAID 0 (şeritleme)	16 TB	0 disk	1 G/Ç
RAID 1 (4 kopya ayna)	4 TB	3 disk	4 G/Ç (her kopyaya)
RAID 10 (ayna + şerit)	8 TB	1 (çiftler farklıysa 2)	2 G/Ç
RAID 5 (dağıtılmış eşlik)	12 TB	1 disk	4 G/Ç (2 oku + 2 yaz)
RAID 6 (çift eşlik)	8 TB	2 disk	6 G/Ç (3 oku + 3 yaz)

RAID Düzenleri



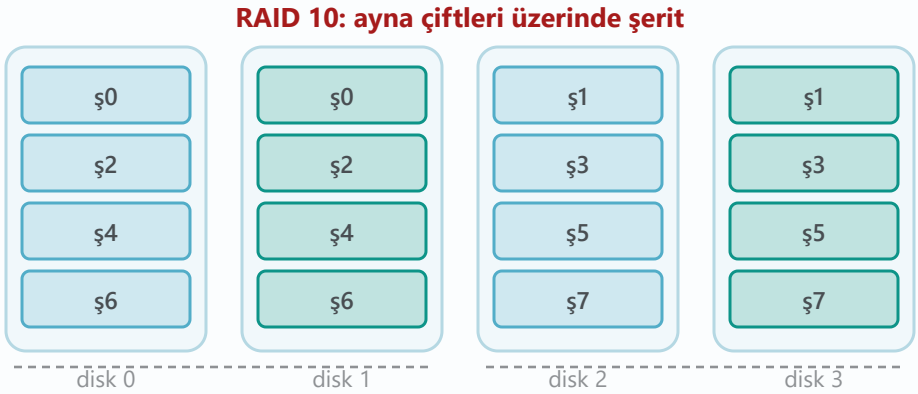
RAID 0: ardışık şeritler farklı disklere; büyük isteklerde n kat bant genişliği. Bir disk bozulursa **her şey** kaybolur.

RAID Düzenleri



RAID 1: her yazma iki kez yapılır; okuma iki diskten birinden (yük dağıtılır). Kapasitenin yarısı artıklık.

RAID Düzenleri



RAID 10: önce aynala, sonra şeritle. Veritabanlarında yaygın: hızlı yazma ve hızlı yeniden kurma.

RAID Düzenleri



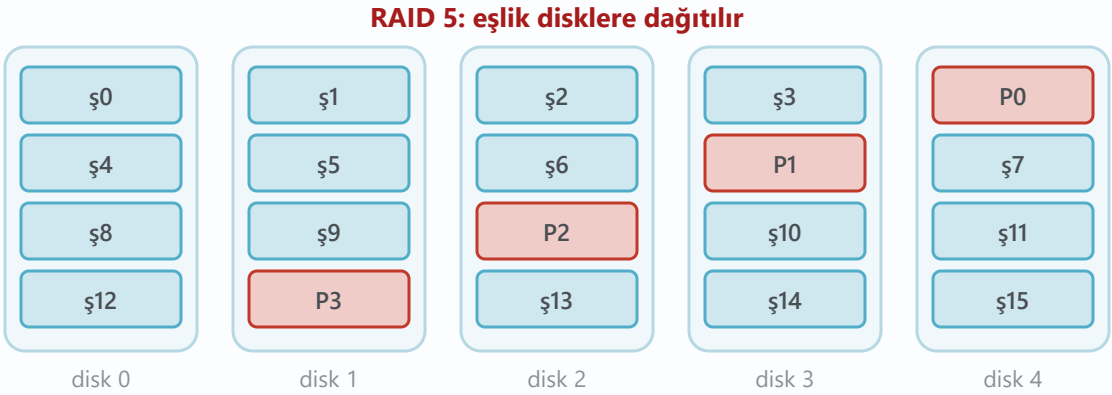
RAID 2 sözcük + Hamming kodu, **RAID 3** bayt + tek eşlik diski kullanır; disklerin eş zamanlı dönmesi gerekir. Bugün kullanılmaz (disk zaten ECC yapar).

RAID Düzenleri



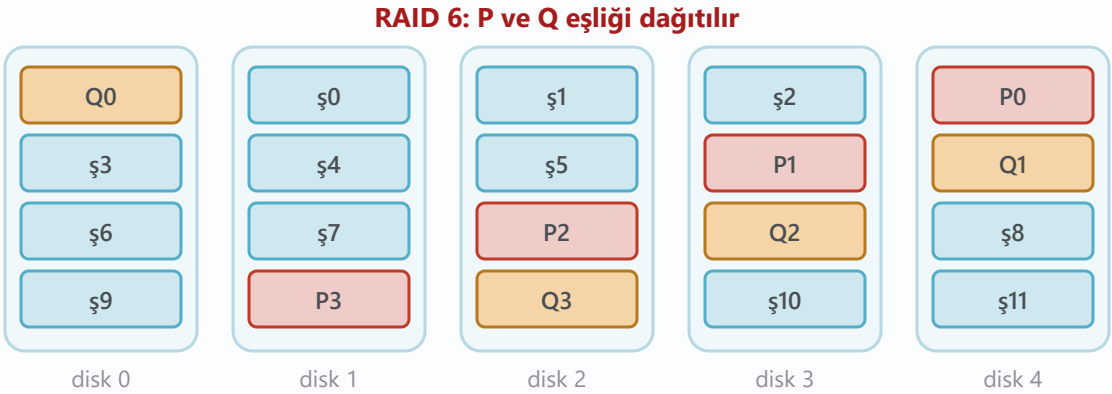
RAID 4: blok şeritleri + ayrı eşlik diski. Her küçük yazma eşlik diskini günceller: eşlik diski **darboğaz**.

RAID Düzenleri



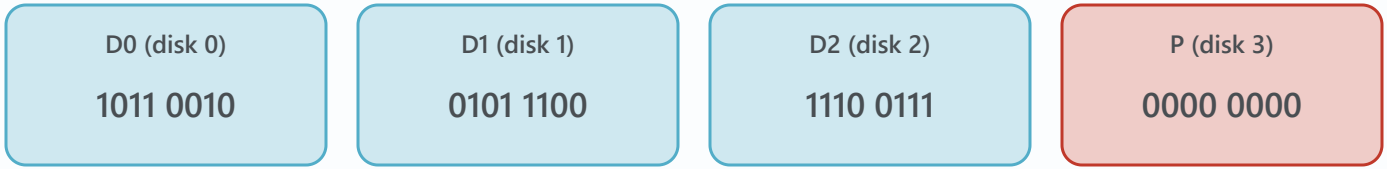
RAID 5: eşlik blokları disklere sırayla dağıtılır; darboğaz kalkar. n − 1 disk kapasitesi, bir disk kaybına dayanır.

RAID Düzenleri



RAID 6: P (XOR) + Q (Reed–Solomon, GF(2^8)) eşliği; iki disk kaybına dayanır. Büyük disklerde yeniden kurma saatler sürdüğü için tercih edilir.

Eşlik ve XOR: Örnek



Bir şeritte üç veri bloğu (burada 8 bitlik örnek). Eşlik bloğu henüz hesaplanmadı.

Adım 1 / 5

Yeniden kurma riski: tüketici diskte düzeltilemeyen okuma hatası (URE) ≈ 10^{14} bitte 1. 12 TB'lık yeniden kurmada beklenen hata $12\times10^{12}\times8 / 10^{14} \approx$ **0,96**; en az bir hata olasılığı ≈ %62: RAID 5'te yeniden kurma büyük olasılıkla takılır, bu yüzden büyük dizilerde RAID 6 seçilir.

Eşlik ve XOR: Örnek

D0 (disk 0)	D1 (disk 1)	D2 (disk 2)	P (disk 3)
1011 0010	0101 1100	1110 0111	0000 1001

P = D0 ⊕ D1 ⊕ D2 = 0000 1001

Eşlik bit bit XOR'dur: her bit konumunda 1'lerin sayısı çift olacak biçimde tamamlanır.

Yeniden kurma riski: tüketici diskte düzeltilemeyen okuma hatası (URE) ≈ 10^{14} bitte 1. 12 TB'lık yeniden kurmada beklenen hata $12\times10^{12}\times8 / 10^{14} \approx$ **0,96**; en az bir hata olasılığı ≈ %62: RAID 5'te yeniden kurma büyük olasılıkla takılır, bu yüzden büyük dizilerde RAID 6 seçilir.

Eşlik ve XOR: Örnek

D0 (disk 0)	D1 (disk 1)	D2 (disk 2)	P (disk 3)
1011 0010	✗ bozuldu	1110 0111	0000 1001

D1 = D0 ⊕ D2 ⊕ P = ?

Disk 1 bozuldu. XOR'un özelliği: x ⊕ x = 0, dolayısıyla kayıp blok kalanların XOR'udur.

Yeniden kurma riski: tüketici diskte düzeltilemeyen okuma hatası (URE) ≈ 10^{14} bitte 1. 12 TB'lık yeniden kurmada beklenen hata $12\times10^{12}\times8 / 10^{14} \approx$ **0,96**; en az bir hata olasılığı ≈ %62: RAID 5'te yeniden kurma büyük olasılıkla takılır, bu yüzden büyük dizilerde RAID 6 seçilir.

Eşlik ve XOR: Örnek

D0 (disk 0)	D1 (disk 1)	D2 (disk 2)	P (disk 3)
1011 0010	0101 1100	1110 0111	0000 1001

D1 = 1011 0010 ⊕ 1110 0111 ⊕ 0000 1001 = 0101 1100

Kurtarılan değer 0101 1100: özgün D1 ile aynı. Yeni diske tüm şeritler böyle yeniden yazılır (yeniden kurma).

Yeniden kurma riski: tüketici diskte düzeltilemeyen okuma hatası (URE) ≈ 10^{14} bitte 1. 12 TB'lık yeniden kurmada beklenen hata $12\times10^{12}\times8 / 10^{14} \approx$ **0,96**; en az bir hata olasılığı ≈ %62: RAID 5'te yeniden kurma büyük olasılıkla takılır, bu yüzden büyük dizilerde RAID 6 seçilir.

Eşlik ve XOR: Örnek

D0 (disk 0)	D1 (disk 1)	D2 (disk 2)	P (disk 3)
1011 0010	0000 1111	1110 0111	0101 1010

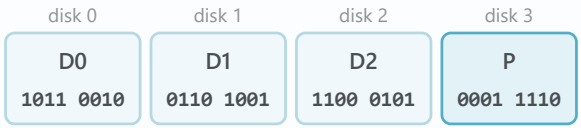
P' = P ⊕ D1 ⊕ D1' = 0101 1010

Küçük yazma: D1' = 0000 1111. Eski D1 ve P okunur, P' hesaplanır, D1' ve P' yazılır: **4 G/Ç** (oku-değiştir-yaz).

Yeniden kurma riski: tüketici diskte düzeltilemeyen okuma hatası (URE) ≈ 10^{14} bitte 1. 12 TB'lık yeniden kurmada beklenen hata $12\times10^{12}\times8 / 10^{14} \approx$ **0,96**; en az bir hata olasılığı ≈ %62: RAID 5'te yeniden kurma büyük olasılıkla takılır, bu yüzden büyük dizilerde RAID 6 seçilir.

RAID 5 Küçük Yazma ve Kurtarma: Kod İzleme

```
▶ void kucuk_yaz(int d, uint8_t yeni) {
2   uint8_t eski = oku(d);    /* G/Ç 1 */
3   uint8_t p = oku(P);       /* G/Ç 2 */
4   p = p ^ eski ^ yeni;
5   yaz(d, yeni);             /* G/Ç 3 */
6   yaz(P, p);                /* G/Ç 4 */
7 }
8 uint8_t kur(int bozuk) {
9   uint8_t x = 0;
10  for (int d = 0; d <= P; d++)
11    if (d != bozuk)
12      x ^= oku(d);          /* P dahil */
13  return x;
14 }
```



disk G/Ç: 0

Şeritte P = D0 ⊕ D1 ⊕ D2 = 0001 1110. D1 yerine 0011 0011 yazılacak: `kucuk_yaz(1, ...)`.

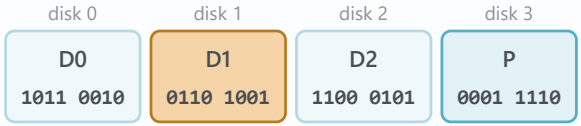
Adım 1 / 11

DEĞİŞKENLER

`d = 1` `yeni = 0011 0011` `eski = –` `p = –`

RAID 5 Küçük Yazma ve Kurtarma: Kod İzleme

```
1  void kucuk_yaz(int d, uint8_t yeni) {
▶    uint8_t eski = oku(d);    /* G/Ç 1 */
3    uint8_t p = oku(P);       /* G/Ç 2 */
4    p = p ^ eski ^ yeni;
5    yaz(d, yeni);             /* G/Ç 3 */
6    yaz(P, p);                /* G/Ç 4 */
7  }
8  uint8_t kur(int bozuk) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bozuk)
12       x ^= oku(d);          /* P dahil */
13   return x;
14 }
```



disk G/Ç: 1

Eski D1 okunur. Diğer veri diskleri (D0, D2) okunmaz: şeritte 3 değil yalnız 1 veri diski değişiyor.

Adım 2 / 11

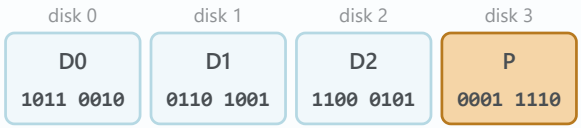
DEĞİŞKENLER

`d = 1` `yeni = 0011 0011`

`eski = 0110 1001` `p = –`

RAID 5 Küçük Yazma ve Kurtarma: Kod İzleme

```
1  void kucuk_yaz(int d, uint8_t yeni) {
2    uint8_t eski = oku(d);    /* G/Ç 1 */
▶    uint8_t p = oku(P);       /* G/Ç 2 */
4    p = p ^ eski ^ yeni;
5    yaz(d, yeni);             /* G/Ç 3 */
6    yaz(P, p);                /* G/Ç 4 */
7  }
8  uint8_t kur(int bozuk) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bozuk)
12       x ^= oku(d);          /* P dahil */
13   return x;
14 }
```



disk G/Ç: 2

Eski eşlik okunur: p = 0001 1110.

Adım 3 / 11

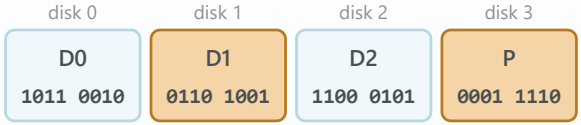
DEĞİŞKENLER

`d = 1` `yeni = 0011 0011`

`eski = 0110 1001` `p = 0001 1110`

RAID 5 Küçük Yazma ve Kurtarma: Kod İzleme

```
1  void kucuk_yaz(int d, uint8_t yeni) {
2    uint8_t eski = oku(d);    /* G/Ç 1 */
3    uint8_t p = oku(P);       /* G/Ç 2 */
▶    p = p ^ eski ^ yeni;
5    yaz(d, yeni);             /* G/Ç 3 */
6    yaz(P, p);                /* G/Ç 4 */
7  }
8  uint8_t kur(int bozuk) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bozuk)
12       x ^= oku(d);          /* P dahil */
13   return x;
14 }
```



0001 1110 ⊕ 0110 1001 ⊕ 0011 0011 = 0100 0100

disk G/Ç: 2

p ⊕ eski, eski D1'in etkisini eşlikten çıkarır (x ⊕ x = 0); ⊕ yeni, yeni değeri ekler. p = 0100 0100.

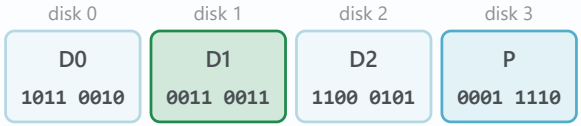
Adım 4 / 11

DEĞİŞKENLER

`d = 1` `yeni = 0011 0011` `eski = 0110 1001` `p = 0100 0100`

RAID 5 Küçük Yazma ve Kurtarma: Kod İzleme

```
1  void kucuk_yaz(int d, uint8_t yeni) {
2    uint8_t eski = oku(d);    /* G/Ç 1 */
3    uint8_t p = oku(P);       /* G/Ç 2 */
4    p = p ^ eski ^ yeni;
▶    yaz(d, yeni);             /* G/Ç 3 */
6    yaz(P, p);                /* G/Ç 4 */
7  }
8  uint8_t kur(int bozuk) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bozuk)
12       x ^= oku(d);          /* P dahil */
13   return x;
14 }
```



disk G/Ç: 3

Yeni D1 yazılır.

Adım 5 / 11

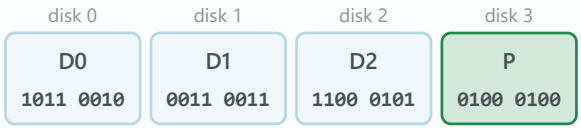
DEĞİŞKENLER

`d = 1` `yeni = 0011 0011`

`eski = 0110 1001` `p = 0100 0100`

RAID 5 Küçük Yazma ve Kurtarma: Kod İzleme

```
1  void kucuk_yaz(int d, uint8_t yeni) {
2    uint8_t eski = oku(d);    /* G/Ç 1 */
3    uint8_t p = oku(P);       /* G/Ç 2 */
4    p = p ^ eski ^ yeni;
5    yaz(d, yeni);             /* G/Ç 3 */
▶    yaz(P, p);                /* G/Ç 4 */
7  }
8  uint8_t kur(int bozuk) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bozuk)
12       x ^= oku(d);          /* P dahil */
13   return x;
14 }
```



disk G/Ç: 4

Yeni P yazılır: **4 G/Ç** (2 okuma + 2 yazma). Bu iki yazma arasında çökme olursa şerit tutarsız kalır (“yazma deliği”).

Adım 6 / 11

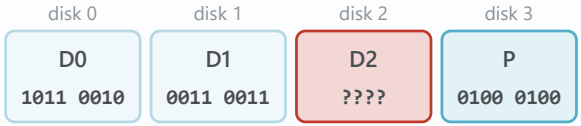
DEĞİŞKENLER

`d = 1` `yeni = 0011 0011`

`eski = 0110 1001` `p = 0100 0100`

RAID 5 Küçük Yazma ve Kurtarma: Kod İzleme

```
1  void kucuk_yaz(int d, uint8_t yeni) {
2    uint8_t eski = oku(d);    /* G/Ç 1 */
3    uint8_t p = oku(P);       /* G/Ç 2 */
4    p = p ^ eski ^ yeni;
5    yaz(d, yeni);             /* G/Ç 3 */
6    yaz(P, p);                /* G/Ç 4 */
7  }
▶ uint8_t kur(int bozuk) {
▶   uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bozuk)
12       x ^= oku(d);          /* P dahil */
13   return x;
14 }
```



disk G/Ç: 0

Disk 2 bozuldu: D2 okunamıyor. `kur(2)` onu kalanların XOR'u olarak hesaplar.

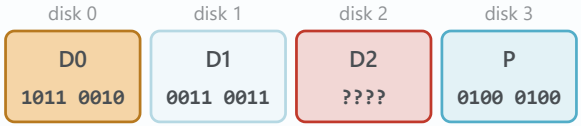
Adım 7 / 11

DEĞİŞKENLER

`bozuk = 2` `d = –` `x = 0000 0000`

RAID 5 Küçük Yazma ve Kurtarma: Kod İzleme

```
1  void kucuk_yaz(int d, uint8_t yeni) {
2    uint8_t eski = oku(d);    /* G/Ç 1 */
3    uint8_t p = oku(P);       /* G/Ç 2 */
4    p = p ^ eski ^ yeni;
5    yaz(d, yeni);             /* G/Ç 3 */
6    yaz(P, p);                /* G/Ç 4 */
7  }
8  uint8_t kur(int bozuk) {
9    uint8_t x = 0;
▶    for (int d = 0; d <= P; d++)
▶      if (d != bozuk)
▶        x ^= oku(d);          /* P dahil */
13   return x;
14 }
```



x = 1011 0010

disk G/Ç: 1

d = 0: x ^= D0 → x = 1011 0010.

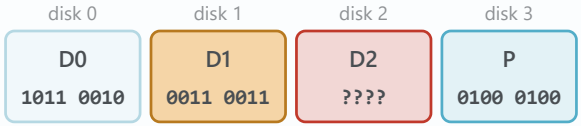
Adım 8 / 11

DEĞİŞKENLER

bozuk = **2** | d = **0** | x = **1011 0010**

RAID 5 Küçük Yazma ve Kurtarma: Kod İzleme

```
1  void kucuk_yaz(int d, uint8_t yeni) {
2    uint8_t eski = oku(d);    /* G/Ç 1 */
3    uint8_t p = oku(P);       /* G/Ç 2 */
4    p = p ^ eski ^ yeni;
5    yaz(d, yeni);             /* G/Ç 3 */
6    yaz(P, p);                /* G/Ç 4 */
7  }
8  uint8_t kur(int bozuk) {
9    uint8_t x = 0;
▶    for (int d = 0; d <= P; d++)
▶      if (d != bozuk)
▶        x ^= oku(d);          /* P dahil */
13   return x;
14 }
```



`x = 1000 0001`

disk G/Ç: 2

d = 1: x ^= D1 → x = 1000 0001.

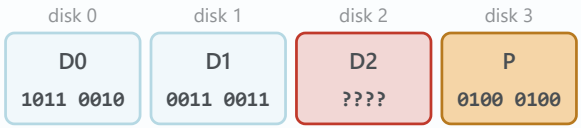
Adım 9 / 11

DEĞİŞKENLER

`bozuk = 2` `d = 1` `x = 1000 0001`

RAID 5 Küçük Yazma ve Kurtarma: Kod İzleme

```
1  void kucuk_yaz(int d, uint8_t yeni) {
2    uint8_t eski = oku(d);    /* G/Ç 1 */
3    uint8_t p = oku(P);       /* G/Ç 2 */
4    p = p ^ eski ^ yeni;
5    yaz(d, yeni);             /* G/Ç 3 */
6    yaz(P, p);                /* G/Ç 4 */
7  }
8  uint8_t kur(int bozuk) {
9    uint8_t x = 0;
▶    for (int d = 0; d <= P; d++)
▶      if (d != bozuk)
▶        x ^= oku(d);          /* P dahil */
13   return x;
14 }
```



x = 1100 0101

disk G/Ç: 3

d = 3: x ^= P → x = 1100 0101. (d = 2 atlandı.)

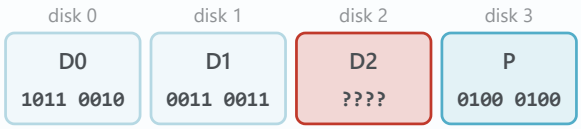
Adım 10 / 11

DEĞİŞKENLER

bozuk = **2** | d = **3** | x = **1100 0101**

RAID 5 Küçük Yazma ve Kurtarma: Kod İzleme

```
1  void kucuk_yaz(int d, uint8_t yeni) {
2    uint8_t eski = oku(d);    /* G/Ç 1 */
3    uint8_t p = oku(P);       /* G/Ç 2 */
4    p = p ^ eski ^ yeni;
5    yaz(d, yeni);             /* G/Ç 3 */
6    yaz(P, p);                /* G/Ç 4 */
7  }
8  uint8_t kur(int bozuk) {
9    uint8_t x = 0;
10   for (int d = 0; d <= P; d++)
11     if (d != bozuk)
12       x ^= oku(d);          /* P dahil */
▶    return x;
14 }
```



D2 = 1100 0101

disk G/Ç: 3

x = 1100 0101, kaybolan D2 ile aynı. Yeni diske her şerit için 3 okuma ile yeniden kurulur; bu sürede dizi korumasızdır.

Adım 11 / 11

DEĞİŞKENLER

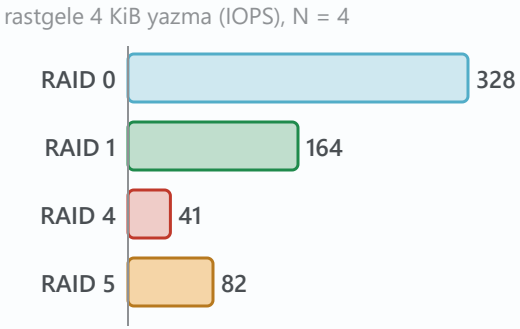
`bozuk = 2` `d = 4` `x = 1100 0101`

RAID Performansı: Örnek

Tek disk (7200 rpm, 8 ms arama, 200 MB/s): rastgele 4 KiB = 12,19 ms → **R ≈ 82 IOPS** (0,34 MB/s); ardışık 10 MiB parça = 8 + 4,17 + 52,43 = 64,6 ms → **S ≈ 162 MB/s**. Dizi: N = 4 disk.

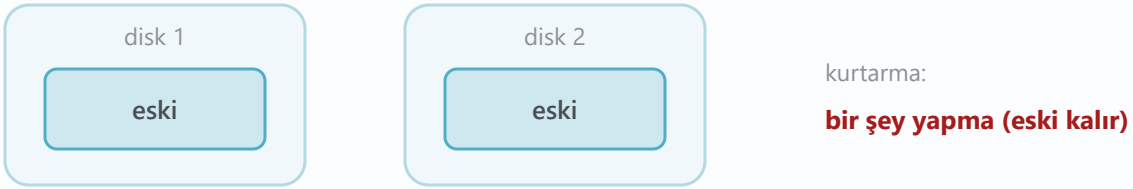
Düzen	Kapasite	Ardışık okuma/yazma	Rastgele okuma	Rastgele yazma	Yazma gecikmesi
RAID 0	N = 4	N·S ≈ 649 MB/s	N·R ≈ 328	N·R ≈ 328	D
RAID 1 (ikili ayna)	N/2 = 2	(N/2)·S ≈ 325 MB/s	N·R ≈ 328	(N/2)·R ≈ 164	D (yavaş kopya)
RAID 4	N − 1 = 3	(N−1)·S ≈ 487 MB/s	(N−1)·R ≈ 246	**R/2 ≈ 41**	2D
RAID 5	N − 1 = 3	(N−1)·S ≈ 487 MB/s	N·R ≈ 328	(N/4)·R ≈ 82	2D

- **Aynada ardışık okuma** yalnız (N/2)·S: iki kopya blokları paylaşır, ama her disk atladığı blokların da altından dönerek geçer.
- **RAID 4**: her küçük yazma eşlik diskinde 1 okuma + 1 yazma ister; dizi tek diskin yarı hızına iner ve disk eklemek işe yaramaz.
- **RAID 5**: aynı 4 G/Ç tüm disklere yayılır, N·R/4 disk sayısıyla büyür. Tam şerit yazmada eski veri okunmaz: (N−1)·S.



Kararlı Depolama

- **Kararlı depolama**: bir yazma ya doğru biçimde tamamlanır ya da eski veri yerinde kalır. Varsayımlar: iki diskte aynı bloğun birden bozulma olasılığı ihmal edilir; hatalı veri ECC ile fark edilir; CPU çökerse süren yazma durur.
- **Kararlı yazma**: disk 1'e yaz, geri oku, doğrula (n kez dene, gerekirse yedek sektör); sonra aynısını disk 2'ye yap. **Kararlı okuma**: disk 1'den geçerli ECC ile okunamazsa disk 2'den oku.



(a) Çökme disk 1'e yazmadan önce: iki kopya da eski. Yazma hiç olmamış sayılır.

RAID bozulan sektörlere karşı korur, yazma sırasında çökmeye karşı korumaz. Bugün aynı fikir günlükleme (journaling), yazma bariyerleri/FUA ve kurumsal SSD'lerdeki güç kaybı koruma kapasitörleriyle uygulanır.

Kararlı Depolama

- **Kararlı depolama**: bir yazma ya doğru biçimde tamamlanır ya da eski veri yerinde kalır. Varsayımlar: iki diskte aynı bloğun birden bozulma olasılığı ihmal edilir; hatalı veri ECC ile fark edilir; CPU çökerse süren yazma durur.
- **Kararlı yazma**: disk 1'e yaz, geri oku, doğrula (n kez dene, gerekirse yedek sektör); sonra aynısını disk 2'ye yap. **Kararlı okuma**: disk 1'den geçerli ECC ile okunamazsa disk 2'den oku.



(b) Çökme disk 1'e yazarken: disk 1 bozuk (ECC). Eski değer disk 2'den geri getirilir.

RAID bozulan sektörlere karşı korur, yazma sırasında çökmeye karşı korumaz. Bugün aynı fikir günlükleme (journaling), yazma bariyerleri/FUA ve kurumsal SSD'lerdeki güç kaybı koruma kapasitörleriyle uygulanır.

Kararlı Depolama

- **Kararlı depolama**: bir yazma ya doğru biçimde tamamlanır ya da eski veri yerinde kalır. Varsayımlar: iki diskte aynı bloğun birden bozulma olasılığı ihmal edilir; hatalı veri ECC ile fark edilir; CPU çökerse süren yazma durur.
- **Kararlı yazma**: disk 1'e yaz, geri oku, doğrula (n kez dene, gerekirse yedek sektör); sonra aynısını disk 2'ye yap. **Kararlı okuma**: disk 1'den geçerli ECC ile okunamazsa disk 2'den oku.

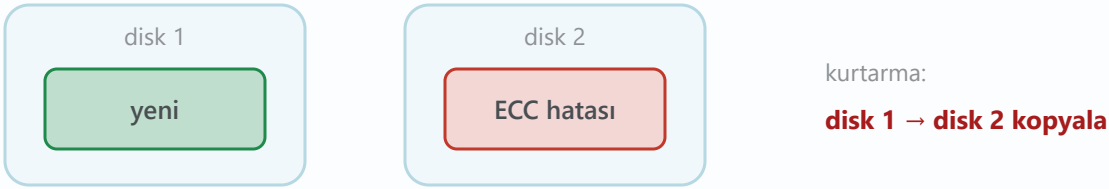


(c) Disk 1 yazıldı, disk 2 yazılmadan çöktü: yeni değer tamamlanır.

RAID bozulan sektörlere karşı korur, yazma sırasında çökmeye karşı korumaz. Bugün aynı fikir günlükleme (journaling), yazma bariyerleri/FUA ve kurumsal SSD'lerdeki güç kaybı koruma kapasitörleriyle uygulanır.

Kararlı Depolama

- **Kararlı depolama**: bir yazma ya doğru biçimde tamamlanır ya da eski veri yerinde kalır. Varsayımlar: iki diskte aynı bloğun birden bozulma olasılığı ihmal edilir; hatalı veri ECC ile fark edilir; CPU çökerse süren yazma durur.
- **Kararlı yazma**: disk 1'e yaz, geri oku, doğrula (n kez dene, gerekirse yedek sektör); sonra aynısını disk 2'ye yap. **Kararlı okuma**: disk 1'den geçerli ECC ile okunamazsa disk 2'den oku.

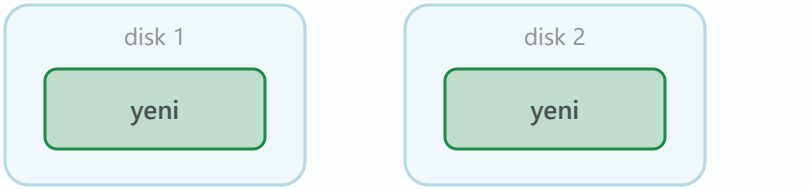


(d) Disk 2'ye yazarken çöktü: disk 2 bozuk, yeni değer disk 1'den kopyalanır.

RAID bozulan sektörlere karşı korur, yazma sırasında çökmeye karşı korumaz. Bugün aynı fikir günlükleme (journaling), yazma bariyerleri/FUA ve kurumsal SSD'lerdeki güç kaybı koruma kapasitörleriyle uygulanır.

Kararlı Depolama

- **Kararlı depolama**: bir yazma ya doğru biçimde tamamlanır ya da eski veri yerinde kalır. Varsayımlar: iki diskte aynı bloğun birden bozulma olasılığı ihmal edilir; hatalı veri ECC ile fark edilir; CPU çökerse süren yazma durur.
- **Kararlı yazma**: disk 1'e yaz, geri oku, doğrula (n kez dene, gerekirse yedek sektör); sonra aynısını disk 2'ye yap. **Kararlı okuma**: disk 1'den geçerli ECC ile okunamazsa disk 2'den oku.



(e) İki kopya da yeni: yazma tamamlanmış. Her durumda sonuç ya tamamen eski ya tamamen yenidir.

RAID bozulan sektörlere karşı korur, yazma sırasında çökmeye karşı korumaz. Bugün aynı fikir günlükleme (journaling), yazma bariyerleri/FUA ve kurumsal SSD'lerdeki güç kaybı koruma kapasitörleriyle uygulanır.

Özet

Konu	Ana fikir	Bugün
Donanım erişimi	Denetleyici yazmaçları; kapı ya da bellek eşlemeli	PCIe BAR ile MMIO
Aktarım	Yoklama → kesme → DMA	MSI-X, bus-master DMA, IOMMU
Yazılım katmanları	Kullanıcı / bağımsız / sürücü / kesme	VFS + sayfa önbelleği + blk-mq
Arayüz	Bloke eden, bloke etmeyen, eş zamansız	epoll, io_uring
Tamponlama	Hız/boyut farkı, çift ve dairesel tampon	NIC/NVMe halkaları, sıfır kopya
HDD	Arama + dönme + aktarım; çizelgeleme	NCQ, mq-deadline, BFQ
SSD	Arama yok; FTL, çöp toplama, TRIM	NVMe, çok kuyruk, `none`
Güvenilirlik	RAID 0/1/5/6/10, kararlı depolama	mdadm, ZFS, günlükleme

G/Ç tasarımının özü: yavaş ve çeşitli aygıtları hızlı CPU'dan ayırmak (tampon, kesme, DMA), farkları tek tip arayüzün arkasına gizlemek (sürücü, VFS) ve sırayı akıllıca seçmek (çizelgeleme, toplu gönderim).

ÖZET

Kontrol Soruları

1. Kapı eşlemeli ve bellek eşlemeli G/Ç'de aygıt yazmacına hangi komutla erişilir? MMIO'da önbellek sorunu nasıl çözülür?
2. 4096 karakter, saniyede 1000 karakter basan yazıcıya kesme güdümlü G/Ç ile gönderilecek; kesme başına 2 µs. CPU maliyeti nedir?
3. Döngü çalma nedir? DMA hangi durumlarda kendini geri ödemez?
4. 7200 rpm, ortalama arama 8 ms, ortam hızı 200 MB/s olan diskte 4 KiB rastgele okuma kaç ms sürer?
5. Kafa 53, kuyruk 98, 183, 37, 122, 14, 124, 65, 67 (0–199). SSTF ve C-LOOK (yukarı) toplam hareketi nedir?
6. Kafa 11, istekler 1, 36, 16, 34, 9, 12. FCFS, SSTF ve asansör (yukarı) kaç silindir hareket eder?
7. RAID 5'te D0 = 1011 0010, D2 = 1110 0111, P = 0000 1001. Bozulan D1 nedir? Tek disk 100 IOPS ise 6 diskli RAID 5 ve RAID 4 rastgele küçük yazmada kaç IOPS verir?
8. SSD'de TRIM neden gerekir? Linux NVMe aygıtlarında varsayılan G/Ç çizelgeleyicisi hangisidir, neden?
9. `epoll` ile `io_uring` arasındaki temel model farkı nedir?

▶ **Cevaplar**

ÖZET

Kaynaklar

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 11: Mass-Storage Structure; Bölüm 12: I/O Systems.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 5: Input/Output.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "I/O Devices", "Hard Disk Drives", "RAID", "Flash-based SSDs".
- D. A. Patterson, G. Gibson, R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", SIGMOD 1988.
- NVM Express Base Specification 2.x; PCI-SIG PCIe 5.0/6.0; USB4 Specification; UEFI Specification (GPT).
- Linux çekirdek belgeleri: "Multi-Queue Block IO Queueing Mechanism (blk-mq)", "NO_HZ: Reducing Scheduling-Clock Ticks", `io_uring` (J. Axboe, "Efficient IO with io_uring"), "Linux Device Drivers" (Corbet, Rubini, Kroah-Hartman, 3. baskı).

Son

Bölüm 11: Giriş/Çıkış

Sonraki bölüm: Çok İşlemcili Sistemler · Terim sözlüğü