

# Bölüm 10: Dosya Sistemleri

## İşletim Sistemleri

Dosya kavramı ve öznitelikler

Dosya işlemleri

Açık dosya tabloları ve fd

Erişim yöntemleri

Dizin yapıları ve yol çözümleme

Katı ve sembolik bağlar

Bağlama ve koruma (rwx, ACL)

MBR, GPT ve süperblok

Bitişik, bağlı, FAT, i-node

Boş alan yönetimi

Önbellek ve performans

fsck, günlükleme, CoW, LFS

NTFS, ext4, FAT32, exFAT

VFS, NFS ve SSD

## Öğrenme Çıktıları

### Dosya soyutlaması

Dosyayı adı, türü, öznitelikleri ve işlemleriyle tanımlamak; açık dosya tablolarını ve dosya tanımlayıcılarını izlemek.

### Dizinler

Dizin yapılarını karşılaştırmak, bir yolu adım adım çözmek, katı/sembolik bağ ve bağlamayı (mount) açıklamak.

### Koruma

rwX izin bitlerini ve özel bitleri okumak; POSIX ve NTFS ACL'lerini karşılaştırmak.

### Tahsis

Bitişik, bağlı liste, FAT ve i-node tahsisini aynı dosya üzerinde uygulamak; en büyük dosya boyutunu hesaplamak.

### Boş alan ve performans

Biteşlem, liste, gruplama ve sayma yöntemlerini hesaplamak; tampon/sayfa önbelleğini ve önden okumayı açıklamak.

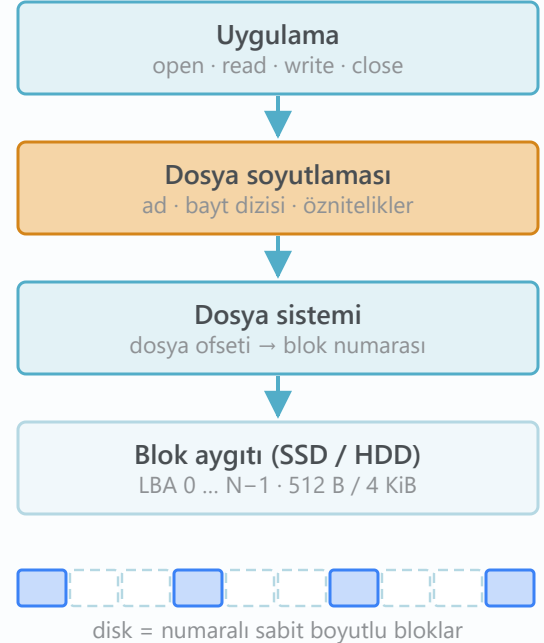
### Güvenilirlik ve gerçek sistemler

fsck, günlükleme, CoW ve LFS'i karşılaştırmak; NTFS, ext4, FAT32/exFAT, VFS, NFS ve SSD TRIM'i tanımak.

# GİRİŞ

## Neden Dosya Sistemi?

- Üç gereksinim: **çok büyük** veri (adres alanına sığmaz), **kalıcılık** (süreç bitince de kalır), **eş zamanlı erişim** (birçok süreç aynı veriyi kullanır).
- Disk yalnızca numaralı bloklar sunar: “*k. bloğu oku / yaz*”. Dosya sistemi bunun üzerine **dosya** ve **dizin** soyutlamasını kurar.
- Dosya, bellekteki adres alanına benzeyen **mantıksal** bir birimdir; süreçler oluşturur, işletim sistemi yönetir: adlandırma, yapı, erişim, koruma, yerleşim.
- İki bakış açısı:
  - **Kullanıcı**: ad, tür, izin, işlem — ne görünüyor?
  - **Gerçekleştirici**: bloklar nasıl tahsis edilir, boş alan nasıl izlenir, çökme sonrası tutarlılık nasıl sağlanır?



# Dosya Adlandırma

| Dosya sistemi | Ad uzunluğu            | Büyük/küçük harf             | Not                                     |
|---------------|------------------------|------------------------------|-----------------------------------------|
| FAT (MS-DOS)  | 8.3 (ör. RAPOR~1.TXT ) | duyarsız                     | VFAT uzun adı ek girdilerde: 255 UTF-16 |
| NTFS          | 255 UTF-16 birimi      | korur, duyarsız              | \ / : * ? " < >   yasak                 |
| ext4, XFS     | 255 bayt               | duyarlı                      | yalnız / ve NUL yasak                   |
| APFS          | 255 UTF-8 karakter     | macOS'ta varsayılan duyarsız | Unicode normalizasyonu                  |

### Uzantı (sonek)

Ad içinde noktadan sonraki kısım: `tez.pdf` .  
UNIX'te yalnızca *gelenektir*; çekirdek anlam yüklemes, ama programlar yükleyebilir  
( `gcc` `.c` ile `.o` 'yu ayırır).

### Windows

Uzantı, kayıt defterinde (registry) bir programla **ilişkilendirilir**; çift tıklayınca o program açılır. Gizli uzantı tuzağı:  
`fatura.pdf.exe` .

Eski Windows sürümleri FAT16/FAT32, güncel Windows NTFS kullanır; FAT32 ve exFAT bugün USB bellek ve SD kartlarda yaşar. Adlar Unicode saklanır; Türkçe karakter sorun değildir.

# Uzantılar ve Sihirli Sayılar

| Grup             | Uzantılar ve açılımları                                                                                                                                                                                                   |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Belge            | <code>docx</code> Word (Office Open XML), <code>doc</code> eski ikili Word, <code>pdf</code> Portable Document Format, <code>odt</code> OpenDocument, <code>rtf</code> , <code>txt</code>                                 |
| Görsel           | <code>jpg</code> JPEG, <code>png</code> Portable Network Graphics, <code>gif</code> , <code>bmp</code> bitmap, <code>tiff</code> ; yeni: <code>webp</code> , <code>avif</code> , <code>heic</code>                        |
| Ses / video      | <code>mp3</code> MPEG-1 Audio Layer III, <code>wav</code> , <code>m4a</code> , <code>flac</code> ; <code>mp4</code> MPEG-4 Part 14, <code>mkv</code> , <code>webm</code> , <code>mov</code> ; <code>flv</code> artık eski |
| Tablo / sunum    | <code>xlsx</code> , <code>csv</code> Comma-Separated Values, <code>pptx</code> , <code>odp</code>                                                                                                                         |
| Web              | <code>html</code> HyperText Markup Language, <code>css</code> Cascading Style Sheets, <code>js</code> JavaScript                                                                                                          |
| Çalıştırılabilir | <code>exe</code> Windows PE, <code>msi</code> Windows Installer; Linux'ta ELF uzantısızdır, kütüphane <code>.so</code>                                                                                                    |
| Arşiv            | <code>zip</code> , <code>7z</code> , <code>rar</code> , <code>tar</code> Tape Archive, <code>gz</code> gzip, <code>zst</code> Zstandard                                                                                   |

dosyanın ilk baytları

PNG

89 50 4E 47

%PNG

PDF

25 50 44 46

%PDF

ELF

7F 45 4C 46

.ELF

ZIP · docx · pptx

50 4B 03 04

PK..

Windows PE

4D 5A

MZ

gzip

1F 8B

..

Tür, uzantıdan bağımsız olarak **sihirli sayı** ile de anlaşılır: `file rapor`

komutu bunu yapar. `docx/xlsx/pptx` aslında ZIP arşividir.

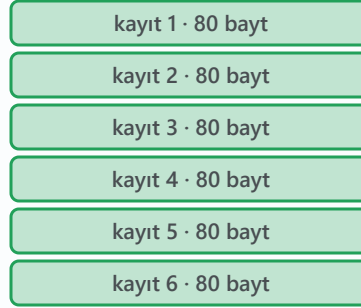
# Dosya Yapısı

(a) Bayt dizisi



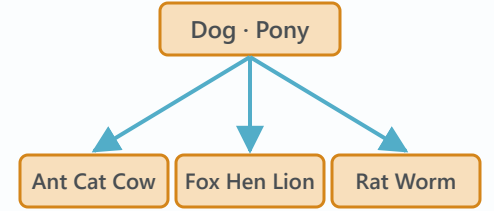
1 bayt = en küçük birim  
UNIX, Windows, macOS  
anlamı uygulama verir

(b) Sabit uzunluklu kayıtlar



delikli kart imajı (80 sütun)  
bugün: anabilgisayar veri kümeleri

(c) Anahtarlı kayıt ağacı



kayıt, anahtar alanıyla bulunur  
bugün: veritabanı / B-ağacı dosyaları

- **Maksimum esneklik:** işletim sistemi dosyayı yalnızca bayt dizisi olarak görür; içine her şey konabilir, yapıyı uygulama belirler.
- Kayıt yapısında okuma/yazma birimi kayıttır; ağaç yapısında *"anahtarı Pony olan kaydı getir"* gibi istekler doğrudan desteklenir.
- Bugün yapılandırılmış erişim işletim sisteminde değil, kütüphanelerde ve veritabanlarında (SQLite, B+ ağaçları) gerçekleştirilir.

# Dosya Türleri

- **Normal dosyalar** kullanıcı verisini tutar: *metin* (satırlar, yazdırılabilir karakterler, satır sonu LF ya da CRLF) veya *ikili* (yapısını program bilir).
- **Dizinler**, dosya sisteminin yapısını tutan sistem dosyalarıdır.
- **Özel dosyalar** G/Ç aygıtlarını modeller: *karakter* aygıtı (yazıcı, terminal) ve *blok* aygıtı (disk).
- "Her şey bir dosyadır": aynı `open/read/write` arayüzü aygıtlar, borular ve soketler için de çalışır. Tablodaki harf `ls -l` çıktısının ilk karakteridir.

| Harf | Tür                               | Örnek                                             |
|------|-----------------------------------|---------------------------------------------------|
| -    | normal dosya (metin ya da ikili)  | <code>notlar.txt</code> , <code>/bin/ls</code>    |
| d    | dizin                             | <code>/home</code>                                |
| l    | sembolik bağ                      | <code>/bin → usr/bin</code>                       |
| c    | karakter aygıtı (seri, bayt bayt) | <code>/dev/tty</code> , <code>/dev/null</code>    |
| b    | blok aygıtı (blok blok, rastgele) | <code>/dev/nvme0n1</code> , <code>/dev/sda</code> |
| p    | adlandırılmış boru (FIFO)         | <code>mkfifo kanal</code>                         |
| s    | UNIX soketi                       | <code>/run/systemd/notify</code>                  |

# İkili Dosyalar: Çalıştırılabilir ve Arşiv

(a) Eski UNIX a.out

|                        |
|------------------------|
| sihirli sayı           |
| metin boyutu           |
| veri boyutu            |
| BSS boyutu             |
| sembol tablosu boyutu  |
| giriş noktası          |
| metin (kod)            |
| veri                   |
| yer değiştirme bitleri |
| sembol tablosu         |

başlık + bölümler

(b) Bugün: ELF (Linux)

|                           |
|---------------------------|
| ELF başlığı · 7F 45 4C 46 |
| program başlıkları        |
| .text (kod)               |
| .rodata                   |
| .data                     |
| .symtab · .strtab         |
| bölüm başlıkları          |

Windows: PE · macOS: Mach-O

(c) Arşiv (.a / .lib)

|                         |
|-------------------------|
| !<arch> başlığı         |
| üye başlığı: ad · boyut |
| nesne modülü (.o)       |
| üye başlığı: ad · boyut |
| nesne modülü (.o)       |
| üye başlığı: ad · boyut |
| nesne modülü (.o)       |

bağlanmamış, derlenmiş modüller

Her işletim sistemi kendi çalıştırılabilir biçimini tanır; çekirdek dosyayı yalnızca doğru sihirli sayıyla başlıyorsa çalıştırır. Şekildeki UNIX dosyası eski **a.out** biçimidir (Tanenbaum); arşiv uzantısı **.a**, içindeki modüller **.o** 'dur.

# Dosya Öznitelikleri

| Öznitelik                 | Anlam                                                                                | Nerede / nasıl                                                             |
|---------------------------|--------------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| Ad, tür, konum            | dizin girdisindeki ad; i-node/MFT kaydındaki tür ve blok adresleri                   | tüm sistemler                                                              |
| Boyut                     | mantıksal boyut ( <code>st_size</code> ) ve ayrılmış alan ( <code>st_blocks</code> ) | seyrek dosyada ikisi farklıdır                                             |
| Sahip, grup               | UID / GID; Windows'ta SID                                                            | <code>chown</code> , <code>icacIs</code>                                   |
| Koruma                    | rwx bitleri, ACL                                                                     | <code>chmod</code> , <code>setfacl</code>                                  |
| Zaman damgaları           | erişim (atime), içerik değişimi (mtime), i-node değişimi (ctime), oluşturma (btime)  | <code>stat</code> ; Linux'ta <code>relatime</code>                         |
| Bağ sayısı                | dosyaya işaret eden ad sayısı                                                        | 0 olunca bloklar serbest                                                   |
| Read-only, Hidden, System | salt okunur, gizli, sistem dosyası                                                   | Windows <code>attrib</code> ; UNIX'te gizli = ad <code>.</code> ile başlar |
| Archive                   | son yedekten sonra değişti, yedeklenecek                                             | Windows; yedekleme aracı sıfırlar                                          |
| Compressed, Encrypted     | sıkıştırılmış / şifrelenmiş                                                          | NTFS, Btrfs, APFS; fscrypt, BitLocker                                      |
| Sparse, Temporary         | sıfır blokları diskte yok; geçici (önbellek ipucu)                                   | NTFS bayrakları                                                            |
| Reparse point             | özel işleme: sembolik bağ, junction, bulut yer tutucu                                | NTFS                                                                       |
| Genişletilmiş öznitelik   | ad = değer çiftleri                                                                  | xattr (SELinux etiketi), NTFS ADS                                          |

*Symlink, Device, Junction* öznitelik değil dosya türüdür; *Temporary* dosyayı kendiliğinden silmez (yalnızca yazmayı geciktirme ipucu); *Creator* ve *Max size* eski sistemlere aittir.

## DOSYA KAVRAMI

# stat Yapısı ve ls -l

```
struct stat {  
    dev_t      st_dev;        /* aygıt (dosya sistemi) */  
    ino_t      st_ino;        /* i-node numarası */  
    mode_t     st_mode;       /* tür + izin bitleri */  
    nlink_t    st_nlink;      /* katı bağ sayısı */  
    uid_t      st_uid;        /* sahip kullanıcı */  
    gid_t      st_gid;        /* sahip grup */  
    dev_t      st_rdev;       /* aygıt dosyası ise no. */  
    off_t      st_size;       /* boyut (bayt) */  
    blksize_t  st_blksize;    /* tercih edilen G/Ç boyu */  
    blkcnt_t   st_blocks;     /* ayrılmış 512 B birim */  
    struct timespec st_atim; /* son erişim */  
    struct timespec st_mtim; /* son içerik */  
    struct timespec st_ctim; /* son i-node */  
};
```

\$ ls -l b.txt



**1**

→ katı bağ sayısı (st\_nlink)

**root root**

→ sahip ve grup

**81**

→ boyut, bayt (st\_size)

**May 26 19:20**

→ son değişiklik (mtime)

**b.txt**

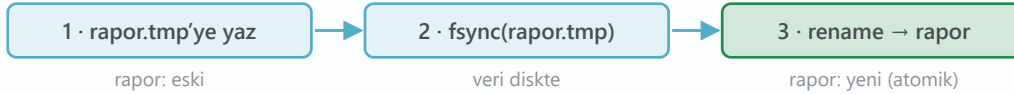
→ ad: dizin girdisinden gelir

ad i-node'da değil, dizindedir!

POSIX.1-2008'den beri zamanlar nanosaniyeli `struct timespec` alanlarıdır ( `st_atime` makro olarak kalır). Linux'ta oluşturma zamanı `statx()` ile okunur.

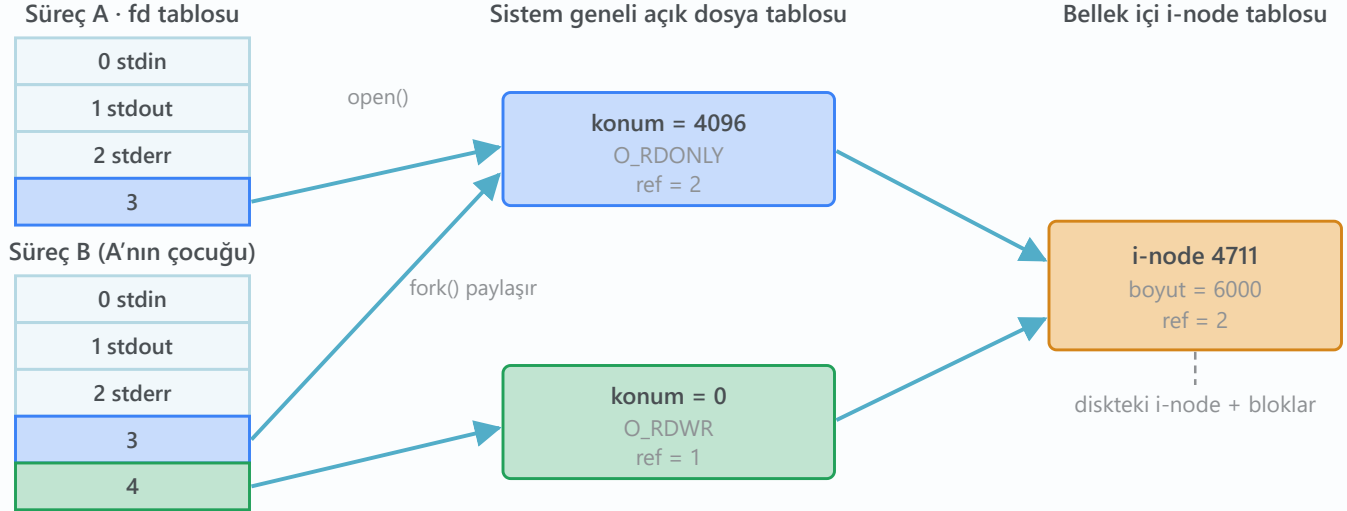
# Dosya İşlemleri ve Sistem Çağrıları

| İşlem                | POSIX çağrısı                                                                        | Ne yapar?                                                              |
|----------------------|--------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| Create               | <code>open(O_CREAT)</code> , <code>creat</code>                                      | boş dosya ve dizin girdisi oluşturur, öznitelikleri atar               |
| Delete               | <code>unlink</code>                                                                  | adı siler; bağ sayısı 0 olup son tanımlayıcı da kapanınca alan boşalır |
| Open                 | <code>open</code>                                                                    | yolu çözer, i-node'u belleğe alır, tanımlayıcı (fd) döndürür           |
| Close                | <code>close</code>                                                                   | fd'yi ve gerekirse tablo girdilerini serbest bırakır                   |
| Read / Write         | <code>read</code> , <code>write</code>                                               | dosya konumundan itibaren okur / yazar, konumu ilerletir               |
| Append               | <code>open(O_APPEND)</code>                                                          | her yazma atomik olarak dosya sonuna gider                             |
| Seek                 | <code>lseek</code>                                                                   | konumu değiştirir (rastgele erişim); G/Ç yapmaz                        |
| Get / Set attributes | <code>stat</code> , <code>chmod</code> , <code>chown</code> , <code>utimensat</code> | öznitelikleri okur / değiştirir                                        |
| Rename               | <code>rename</code>                                                                  | adı değiştirir; aynı dosya sisteminde atomiktir                        |
| Truncate, Map        | <code>ftruncate</code> , <code>mmap</code>                                           | boyutu kısaltır/uzatır; dosyayı adres alanına eşler                    |



Güvenli kaydetme kalıbı: yeni içeriği geçici dosyaya yaz → `fsync` → `rename(gecici, asil)` .  
 Çökme olursa ya eski ya yeni dosya görünür, yarım dosya görünmez.

# Açık Dosya Tabloları



- fd**: çekirdeğin süreç başına tablosunda bir indis; 0, 1, 2 standart girdi/çıkıtı/hata, `open` en küçük boş numarayı verir.
- Açık dosya girdisi** konumu (offset) ve erişim modunu tutar: `fork` ve `dup` girdiyi paylaşır (ortak konum); ikinci bir `open` yeni girdi açar (ayrı konum).
- i-node** dosya başına bir kez bellektedir; diskte birden çok açılış onu tekrar okumaz. Windows'ta karşılığı tanıtıcı (handle) tablosudur.

## open, read, close Adım Adım

```
1 #include <fcntl.h>
2 #include <unistd.h>
3
4 int main(void) {
5     ▶ char buf[4096];
6     int fd = open("notlar.txt", O_RDONLY);
7     ssize_t n = read(fd, buf, sizeof buf);
8     n = read(fd, buf, sizeof buf);
9     n = read(fd, buf, sizeof buf);
10    close(fd);
11    return 0;
12 }
```

fd tablosu

|   |     |
|---|-----|
| 0 | tty |
| 1 | tty |
| 2 | tty |
| 3 |     |

notlar.txt (6000 bayt)

|          |             |
|----------|-------------|
| 0 – 4095 | 4096 – 5999 |
|----------|-------------|

Başlangıç: yalnız 0, 1, 2 açık (kabuktan miras: terminal).

Adım 1 / 6

### DEĞİŞKENLER

fd = -

n = -

konum = -

# open, read, close Adım Adım

```
1 #include <fcntl.h>
2 #include <unistd.h>
3
4 int main(void) {
5     char buf[4096];
6     ► int fd = open("notlar.txt", O_RDONLY);
7     ssize_t n = read(fd, buf, sizeof buf);
8     n = read(fd, buf, sizeof buf);
9     n = read(fd, buf, sizeof buf);
10    close(fd);
11    return 0;
12 }
```

fd tablosu

|       |
|-------|
| 0 tty |
| 1 tty |
| 2 tty |
| 3     |

açık dosya girdisi

konum = 0  
O\_RDONLY  
ref = 1

i-node 4711  
boyut 6000  
ref = 1

notlar.txt (6000 bayt)

|          |             |
|----------|-------------|
| 0 – 4095 | 4096 – 5999 |
|----------|-------------|

**konum**

**open** : yol çözülür, i-node 4711 belleğe alınır, konumu 0 olan bir açık dosya girdisi kurulur; en küçük boş numara **3** döner.

Adım 2 / 6

**DEĞİŞKENLER**

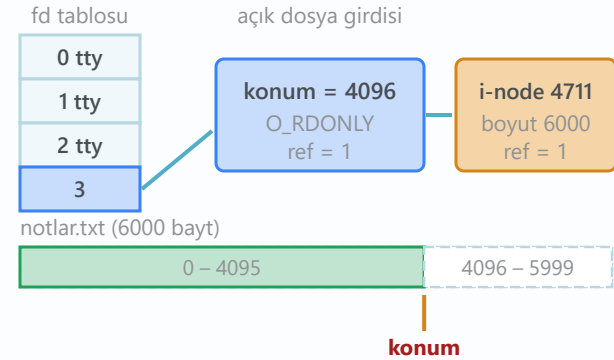
fd = 3

n = -

konum = 0

## open, read, close Adım Adım

```
1 #include <fcntl.h>
2 #include <unistd.h>
3
4 int main(void) {
5     char buf[4096];
6     int fd = open("notlar.txt", O_RDONLY);
7     ► ssize_t n = read(fd, buf, sizeof buf);
8     n = read(fd, buf, sizeof buf);
9     n = read(fd, buf, sizeof buf);
10    close(fd);
11    return 0;
12 }
```



İlk **read** : blok(lar) önbellege/okunur, 4096 bayt **buf** 'a kopyalanır, konum **4096** olur.

Adım 3 / 6

### DEĞİŞKENLER

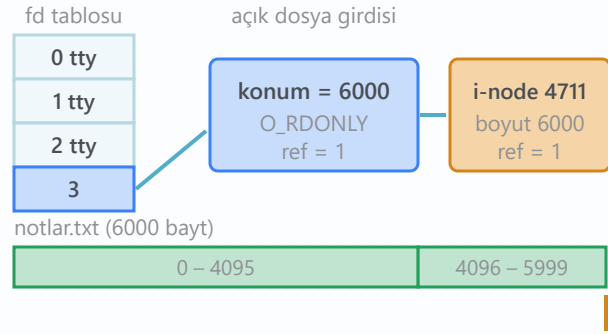
fd = 3

n = 4096

konum = 4096

# open, read, close Adım Adım

```
1 #include <fcntl.h>
2 #include <unistd.h>
3
4 int main(void) {
5     char buf[4096];
6     int fd = open("notlar.txt", O_RDONLY);
7     ssize_t n = read(fd, buf, sizeof buf);
8     ► n = read(fd, buf, sizeof buf);
9     n = read(fd, buf, sizeof buf);
10    close(fd);
11    return 0;
12 }
```



ikinci **read** : dosyada yalnız 1904 bayt kaldı; **kısa okuma** hata değildir. Konum 6000.

Adım 4 / 6

## DEĞİŞKENLER

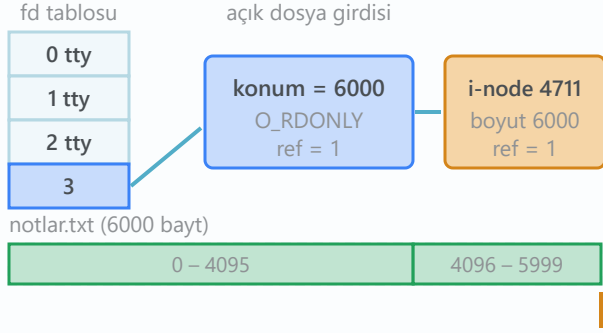
fd = 3

n = 1904

konum = 6000

## open, read, close Adım Adım

```
1 #include <fcntl.h>
2 #include <unistd.h>
3
4 int main(void) {
5     char buf[4096];
6     int fd = open("notlar.txt", O_RDONLY);
7     ssize_t n = read(fd, buf, sizeof buf);
8     n = read(fd, buf, sizeof buf);
9     n = read(fd, buf, sizeof buf);
10    close(fd);
11    return 0;
12 }
```



**konum**

Üçüncü **read** **0** döndürür: dosya sonu (EOF). Hata olsaydı -1 ve **errno**.

Adım 5 / 6

### DEĞİŞKENLER

fd = 3

n = 0

konum = 6000

# open, read, close Adım Adım

```
1 #include <fcntl.h>
2 #include <unistd.h>
3
4 int main(void) {
5     char buf[4096];
6     int fd = open("notlar.txt", O_RDONLY);
7     ssize_t n = read(fd, buf, sizeof buf);
8     n = read(fd, buf, sizeof buf);
9     n = read(fd, buf, sizeof buf);
10    ► close(fd);
11    return 0;
12 }
```

fd tablosu

|       |
|-------|
| 0 tty |
| 1 tty |
| 2 tty |
| 3     |

girdi serbest

ref = 0

notlar.txt (6000 bayt)

|          |             |
|----------|-------------|
| 0 – 4095 | 4096 – 5999 |
|----------|-------------|

**close** : fd 3 boşalır; girdinin referans sayısı 0 → silinir, i-node'un da 0 → bellekten atılabilir.

Adım 6 / 6

**DEĞİŞKENLER**fd = **kapalı**n = **0**konum = **-**

# Dosya Kopyalama: copy abc xyz

```
1 #define BUF 4096
2
3 int kopyala(const char *src, const char *dst)
4 {
5     char buf[BUF];
6     int in = open(src, O_RDONLY);
7     int out = open(dst, O_WRONLY | O_CREAT
8                 | O_TRUNC, 0644);
9     ssize_t n;
10    while ((n = read(in, buf, BUF)) > 0)
11        write(out, buf, n);
12    close(in);
13    close(out);
14    return 0;
15 }
```

abc (10000 bayt)

tampon  
(4096 B)

xyz

in = 3 ; out = 4 . O\_TRUNC : xyz varsa içeriği silinir, yoksa O\_CREAT ile 0644 izinle oluşturulur.

Adım 1 / 6

## DEĞİŞKENLER

in = 3

out = 4

n = -

kopyalanan = 0

Gerçek kodda her çağrının -1 dönüşü denetlenmeli ve **write** 'ın kısa yazması döngüyle tamamlanmalıdır; **O\_CREAT** kullanılıyorsa üçüncü argüman (izinler) de verilmelidir.

# Dosya Kopyalama: copy abc xyz

```
1 #define BUF 4096
2
3 int kopyala(const char *src, const char *dst)
4 {
5     char buf[BUF];
6     int in = open(src, O_RDONLY);
7     int out = open(dst, O_WRONLY | O_CREAT
8                 | O_TRUNC, 0644);
9     ssize_t n;
10    while ((n = read(in, buf, BUF)) > 0)
11        write(out, buf, n);
12    close(in);
13    close(out);
14    return 0;
15 }
```

abc (10000 bayt)

tampon  
(4096 B)

xyz

İlk **read** : 4096 bayt tampona.

Adım 2 / 6

## DEĞİŞKENLER

in = 3

out = 4

n = 4096

kopyalanan = 0

Gerçek kodda her çağrının `-1` dönüşü denetlenmeli ve **write** 'ın kısa yazması döngüyle tamamlanmalıdır; **O\_CREAT** kullanılıyorsa üçüncü argüman (izinler) de verilmelidir.

# Dosya Kopyalama: copy abc xyz

```
1 #define BUF 4096
2
3 int kopyala(const char *src, const char *dst)
4 {
5     char buf[BUF];
6     int in = open(src, O_RDONLY);
7     int out = open(dst, O_WRONLY | O_CREAT
8                 | O_TRUNC, 0644);
9     ssize_t n;
10    while ((n = read(in, buf, BUF)) > 0)
11        write(out, buf, n);
12    close(in);
13    close(out);
14    return 0;
15 }
```

abc (10000 bayt)

tampon  
(4096 B)

write

xyz



**write** aynı 4096 baytı xyz'ye yazar (önce sayfa önbelleğine; diske sonra gider).

Adım 3 / 6

## DEĞİŞKENLER

in = 3

out = 4

n = 4096

kopyalanan = 4096

Gerçek kodda her çağrının `-1` dönüşü denetlenmeli ve **write** 'ın kısa yazması döngüyle tamamlanmalıdır; **O\_CREAT** kullanılıyorsa üçüncü argüman (izinler) de verilmelidir.

# Dosya Kopyalama: copy abc xyz

```
1 #define BUF 4096
2
3 int kopyala(const char *src, const char *dst)
4 {
5     char buf[BUF];
6     int in = open(src, O_RDONLY);
7     int out = open(dst, O_WRONLY | O_CREAT
8                 | O_TRUNC, 0644);
9     ssize_t n;
10    while ((n = read(in, buf, BUF)) > 0)
11        write(out, buf, n);
12    close(in);
13    close(out);
14    return 0;
15 }
```

abc (10000 bayt)



tampon  
(4096 B)



write

xyz



İkinci tur: 8192 bayt kopyalandı.

Adım 4 / 6

## DEĞİŞKENLER

in = 3

out = 4

n = 4096

kopyalanan = 8192

Gerçek kodda her çağrının `-1` dönüşü denetlenmeli ve `write` 'ın kısa yazması döngüyle tamamlanmalıdır; `O_CREAT` kullanılıyorsa üçüncü argüman (izinler) de verilmelidir.

# Dosya Kopyalama: copy abc xyz

```
1 #define BUF 4096
2
3 int kopyala(const char *src, const char *dst)
4 {
5     char buf[BUF];
6     int in = open(src, O_RDONLY);
7     int out = open(dst, O_WRONLY | O_CREAT
8                 | O_TRUNC, 0644);
9     ssize_t n;
10    while ((n = read(in, buf, BUF)) > 0)
11        write(out, buf, n);
12    close(in);
13    close(out);
14    return 0;
15 }
```

abc (10000 bayt)

tampon  
(4096 B)

1808 bayt



write

xyz



Üçüncü **read** yalnız **1808** bayt döndürür;  
**write** tam **n** bayt yazar (BUF değil!).

Adım 5 / 6

## DEĞİŞKENLER

in = 3

out = 4

n = 1808

kopyalanan = 10000

Gerçek kodda her çağrının `-1` dönüşü denetlenmeli ve **write** 'ın kısa yazması döngüyle tamamlanmalıdır; **O\_CREAT** kullanılıyorsa üçüncü argüman (izinler) de verilmelidir.

# Dosya Kopyalama: copy abc xyz

```
1 #define BUF 4096
2
3 int kopyala(const char *src, const char *dst)
4 {
5     char buf[BUF];
6     int in = open(src, O_RDONLY);
7     int out = open(dst, O_WRONLY | O_CREAT
8                 | O_TRUNC, 0644);
9     ssize_t n;
10    while ((n = read(in, buf, BUF)) > 0)
11        write(out, buf, n);
12    close(in);
13    close(out);
14    return 0;
15 }
```

abc (10000 bayt)

tampon  
(4096 B)

xyz



**read** 0 → döngü biter; iki tanımlayıcı kapatılır.  
Toplam 4 **read** + 3 **write** çağrısı.

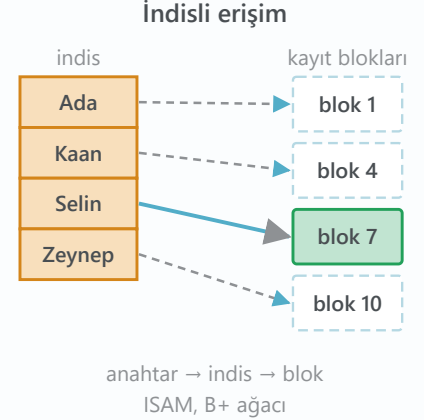
Adım 6 / 6

## DEĞİŞKENLER

in = **kapalı**out = **kapalı**n = **0**kopyalanan = **10000**

Gerçek kodda her çağrının -1 dönüşü denetlenmeli ve **write** 'ın kısa yazması döngüyle tamamlanmalıdır; **O\_CREAT** kullanılıyorsa üçüncü argüman (izinler) de verilmelidir.

# Erişim Yöntemleri



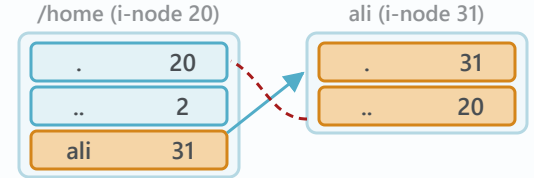
- **Sıralı:** baştan sona sırayla; atlama yerine sarma. Teyp ve kaset böyleydi; bugün de günlük dosyaları ve akışlar için en yaygın kalıptır.
- **Doğrudan:** dosya numaralı bloklar/kayıtlar dizisidir; istenen yerden başlanır ( `lseek` , `pread` ). Diskler ve SSD'ler bunu destekler.
- **İndisli:** küçük bir indis (gerekirse indisin indisi) anahtarla doğru bloğu bulur; doğrudan erişimin üzerine kurulur.
- **Bellek eşlemeli** erişimde ( `mmap` ) dosya sayfaları adres alanına eşlenir; okuma/yazma sıradan bellek erişimidir.

## DİZİNLER

# Dizin İşlemleri

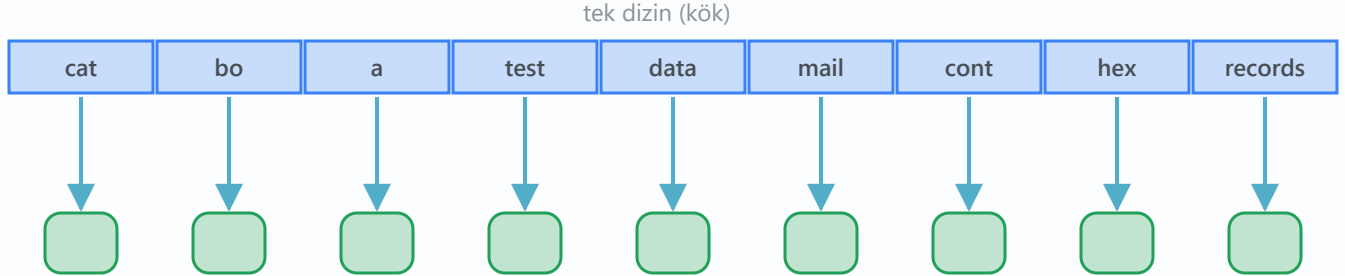
| Çağrı                                        | İşlev                                                                                |
|----------------------------------------------|--------------------------------------------------------------------------------------|
| <code>mkdir</code>                           | boş dizin oluşturur (içinde yalnız <code>.</code> ve <code>..</code> )               |
| <code>rmdir</code>                           | dizini siler; dizin <b>boş</b> olmalıdır                                             |
| <code>opendir</code> / <code>closedir</code> | dizin akışını açar / kapatır                                                         |
| <code>readdir</code>                         | açık dizindeki bir sonraki girdiyi (ad, i-node no., tür) döndürür                    |
| <code>rename</code>                          | dosya ya da dizini yeniden adlandırır veya taşır (aynı dosya sisteminde atomik)      |
| <code>link</code>                            | var olan bir <i>dosyaya</i> yeni bir ad (katı bağ) ekler; i-node'un bağ sayısı artar |
| <code>symlink</code>                         | hedef yolunu içeren sembolik bağ oluşturur                                           |
| <code>unlink</code>                          | bir adı kaldırır; bağ sayısı 0 olursa dosya silinir                                  |

- Dizin de bir dosyadır: içeriği (*ad* → *dosya*) eşlemeleridir; ama kullanıcı onu `write` ile değil, yalnızca bu çağrılarla değiştirebilir.
- `link` dizini başka bir dizine bağlamaz, dosyaya ikinci bir ad verir; dizinlere katı bağ kullanıcıya yasaktır (döngü önlenir).



bağ sayısı: 20 → 3 (yeni ..), 31 → 2 (ali ve ..)

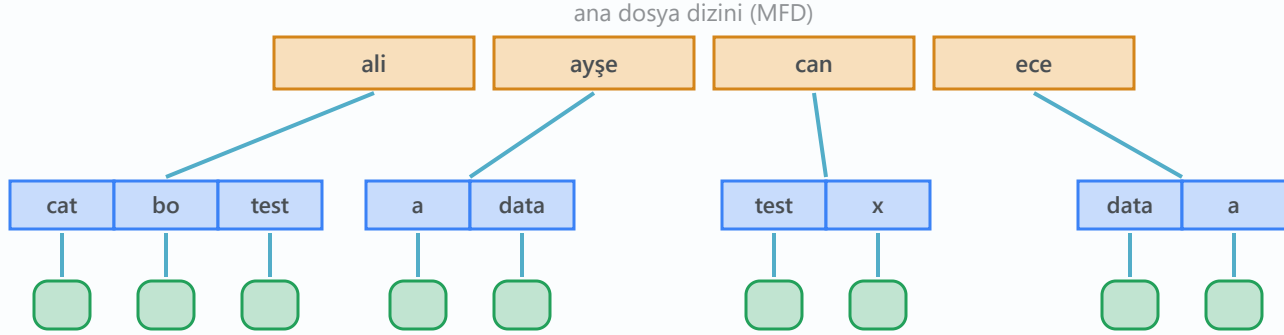
# Dizin Yapıları



Tüm kullanıcıların dosyaları aynı yerde: iki kullanıcı da “test” adını isteyemez, dosyalar gruplanamaz. İlk sistemler (CDC 6600) ve bazı gömülü aygıtlar böyleydi.

**Tek düzeyli:** basit, ama **adlandırma** ve **gruplama** sorunları var.

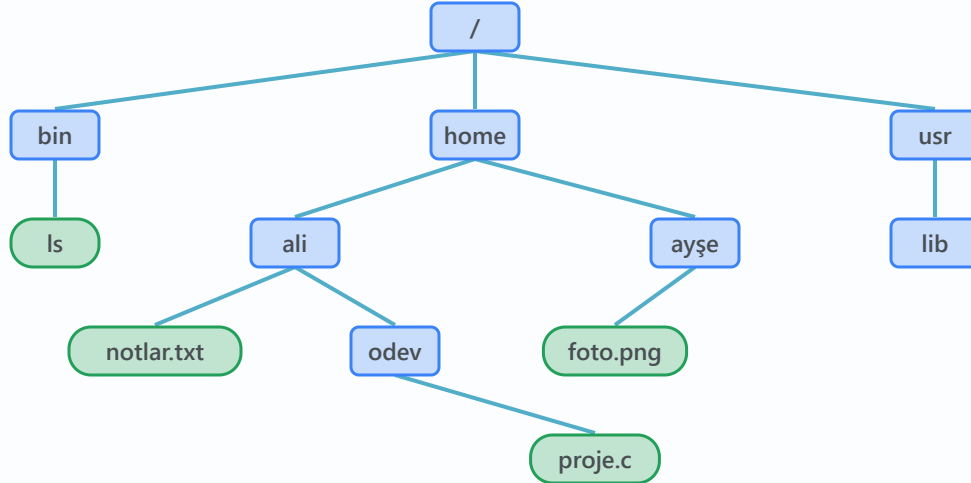
## Dizin Yapıları



Kullanıcı başına dosya dizini (UFD): ad çakışması yok (ali/test ≠ can/test), ama paylaşım ve gruplama zor.

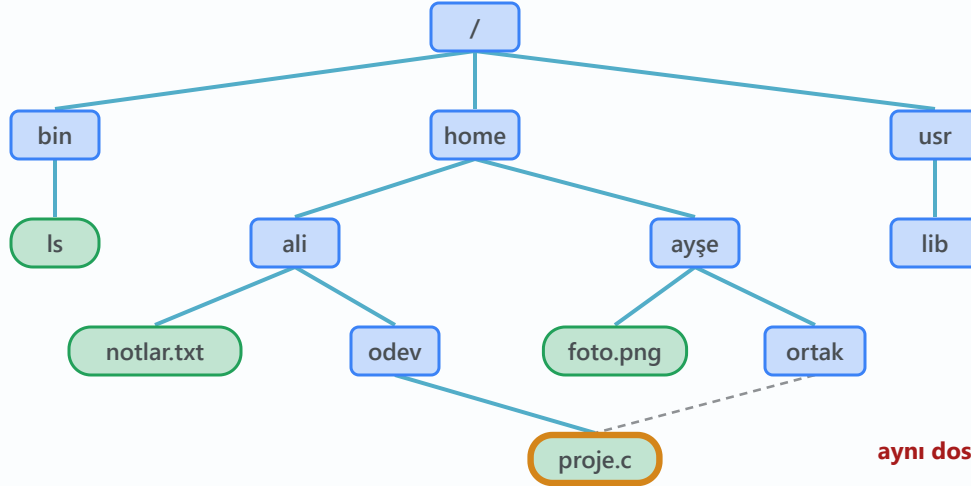
**İki düzeyli:** yol = `kullanıcı/dosya` . Sistem programları için ayrı bir “kullanıcı” dizini gerekir; aranacak dizin listesi (*search path*) fikri buradan gelir.

# Dizin Yapıları



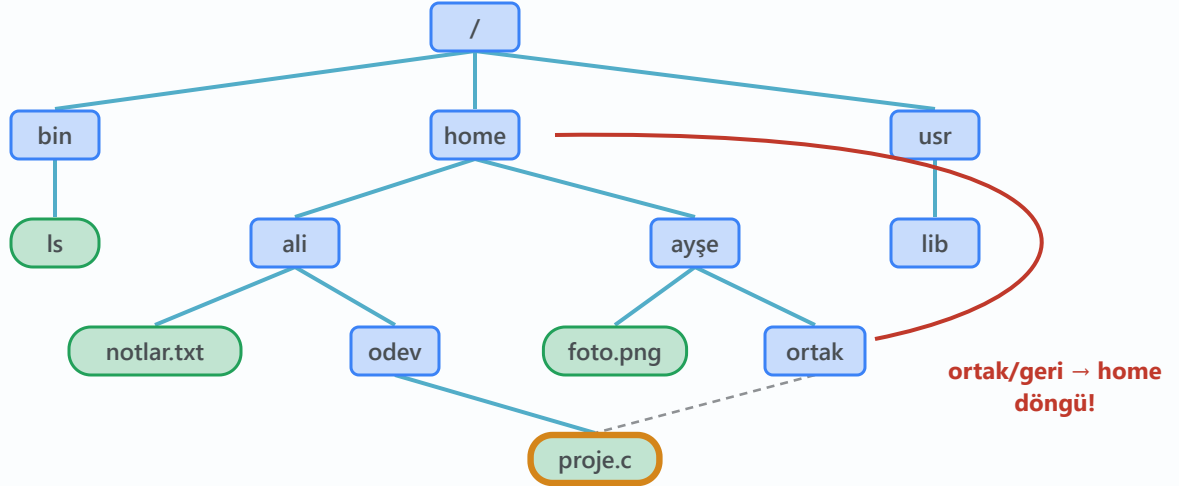
**Ağaç** (hiyerarşik): kullanıcı istediği kadar alt dizin açar; her dosyanın tek bir mutlak yolu vardır. Bugünkü tüm genel amaçlı sistemlerin temeli.

# Dizin Yapıları



**Döngüsüz çizge** (DAG): `proje.c` hem `ali/odev` hem `ayşe/ortak` altında görünür (bağ). Sorun: dolaşırken iki kez sayılabilir, silerken kimin silindiği belirsizleşir.

# Dizin Yapıları



**Genel çizge:** dizine bağ verilirse **döngü** oluşur; arama sonsuza gidebilir, erişilemeyen döngüler için çöp toplama gerekir. UNIX bu yüzden dizinlere katı bağı yasaklar.

## DİZİNLER

# Yol Adları

- **Mutlak yol** kökten başlar:

`/home/sercan/os/slaytlar` . Windows:

`C:\Users\sercan\os` ; ağ paylaşımı

`\\sunucu\pay` .

- **Görelî yol** çalışma dizininden ( `pwd` , `cd` )

başlar: `/home/sercan/ders` içindeyken

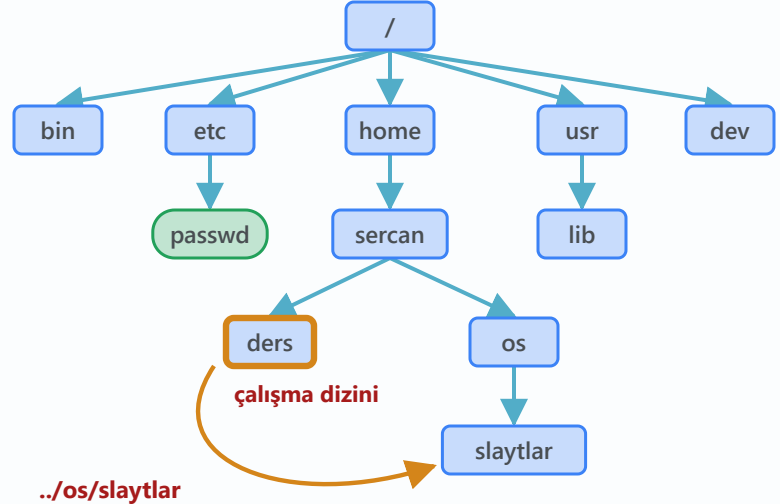
`../os/slaytlar` .

- `.` bulunulan dizin, `..` üst dizin; kökte

`..` yine köke çıkar.

- Ayırıcı UNIX'te `/` , Windows'ta `\` (API

`/` 'yi de kabul eder).



Tarihsel UNIX'te kullanıcı dizinleri `/usr` altındaydı; bugün `/home` (macOS'ta `/Users` ) altındadır, `/usr` salt okunur sistem yazılımı içindir.

## Yol Çözümleme Adım Adım

kök dizin

|    |     |
|----|-----|
| 1  | .   |
| 1  | ..  |
| 4  | bin |
| 7  | dev |
| 14 | lib |
| 9  | etc |
| 6  | usr |
| 8  | tmp |

Mutlak yol: kök dizinin i-node'u (UNIX'te 1; ext4'te 2) bilinir. Kök dizinde **usr** aranır → **i-node 6**.

Adım 1 / 5

Örnek Tanenbaum'un şeklidir: `/usr/ast/mbox` aranır. Göreli yolda arama kök yerine çalışma dizininin i-node'undan başlar.

## Yol Çözümleme Adım Adım

kök dizin

|    |     |
|----|-----|
| 1  | .   |
| 1  | ..  |
| 4  | bin |
| 7  | dev |
| 14 | lib |
| 9  | etc |
| 6  | usr |
| 8  | tmp |

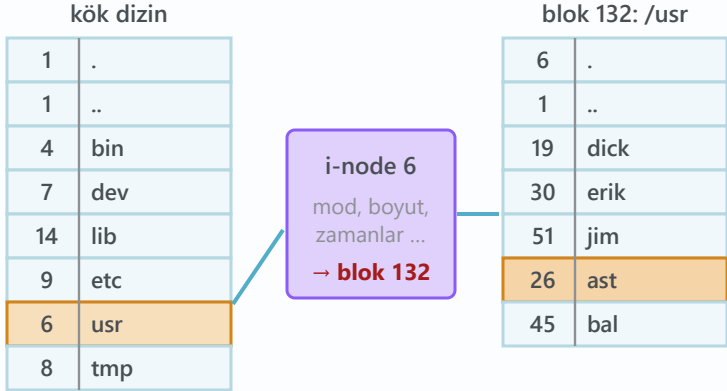
i-node 6  
mod, boyut,  
zamanlar ...  
→ **blok 132**

i-node 6 okunur: `/usr` bir dizin, içeriği **blok 132**'de. (Burada izin denetimi de yapılır: dizinde `x` hakkı gerekir.)

Adım 2 / 5

Örnek Tanenbaum'un şeklidir: `/usr/ast/mbox` aranır. Göreli yolda arama kök yerine çalışma dizininin i-node'undan başlar.

## Yol Çözümleme Adım Adım

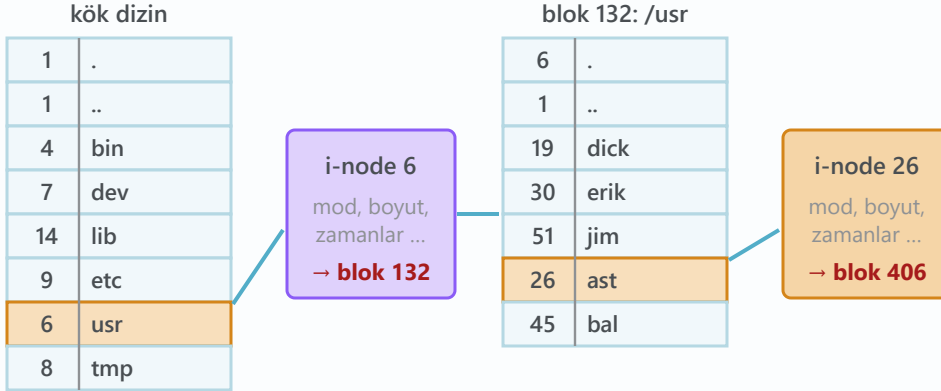


Blok 132 okunur, **ast** aranır → **i-node 26**.

Adım 3 / 5

Örnek Tanenbaum'un şeklidir: `/usr/ast/mbox` aranır. Görelî yolda arama kök yerine çalışma dizininin i-node'undan başlar.

# Yol Çözümleme Adım Adım

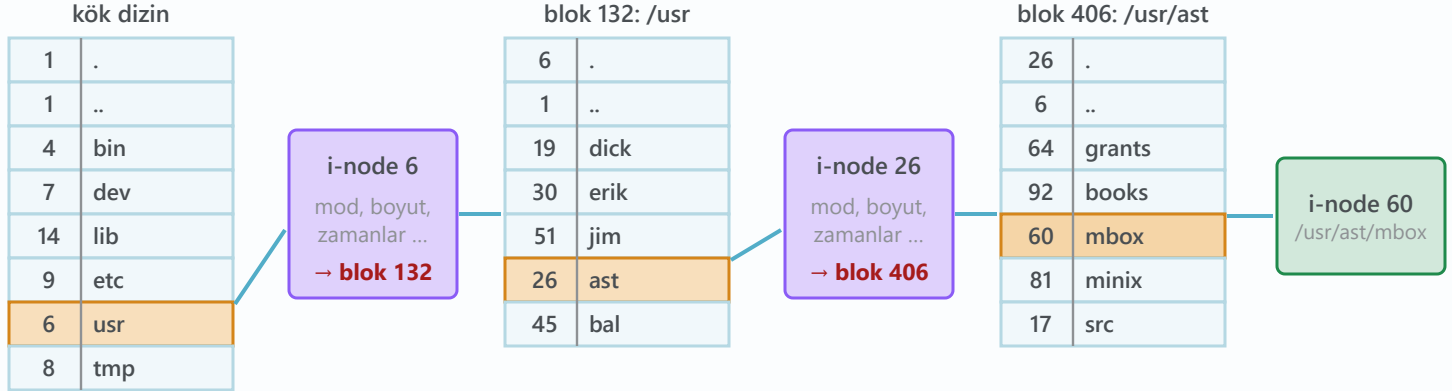


i-node 26: `/usr/ast` dizininin içeriği **blok 406**'da.

Adım 4 / 5

Örnek Tanenbaum'un şeklidir: `/usr/ast/mbox` aranır. Görelî yolda arama kök yerine çalışma dizininin i-node'undan başlar.

# Yol Çözümleme Adım Adım

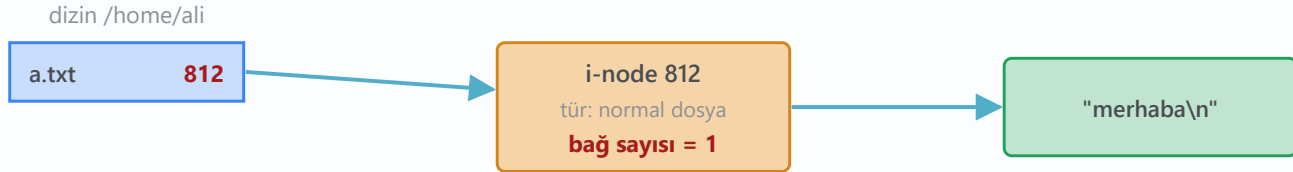


Blok 406'da **mbox** → **i-node 60**. Önbelleksiz 5 disk okuması; Linux bu yüzden ad → i-node eşlemelerini **dentry önbelleğinde** tutar.

Adım 5 / 5

Örnek Tanenbaum'un şeklidir: `/usr/ast/mbox` aranır. Göreli yolda arama kök yerine çalışma dizininin i-node'undan başlar.

## Katı ve Sembolik Bağlar

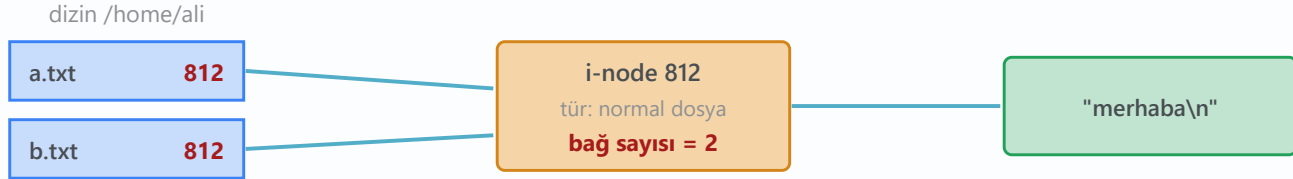


`echo merhaba > a.txt` : dizin girdisi **a.txt** → **i-node 812**, bağ sayısı 1.

Adım 1 / 4

Katı bağ: aynı dosya sistemi içinde, dizinlere değil, ek maliyet yok. Sembolik bağ: dizinlere, başka dosya sistemlerine, hatta ağ yollarına gösterebilir; her erişimde ek yol çözümü gerekir, döngüleri çekirdek sınırlar ( `ELOOP` ). Windows: `mklink /H` , `mklink` , junction.

## Katı ve Sembolik Bağlar

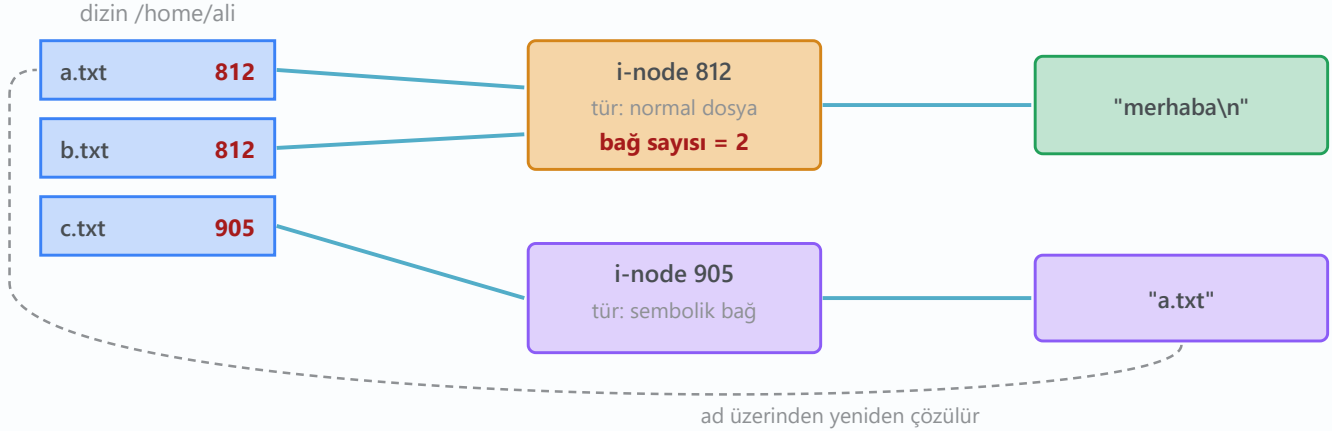


`ln a.txt b.txt` (**katı bağ**): yeni girdi aynı i-node'u gösterir; bağ sayısı **2**. İki ad eşittir, "asıl" yoktur.

Adım 2 / 4

Katı bağ: aynı dosya sistemi içinde, dizinlere değil, ek maliyet yok. Sembolik bağ: dizinlere, başka dosya sistemlerine, hatta ağ yollarına gösterebilir; her erişimde ek yol çözümü gerekir, döngüleri çekirdek sınırlar ( `ELOOP` ). Windows: `mklink /H` , `mklink` , junction.

## Katı ve Sembolik Bağlar

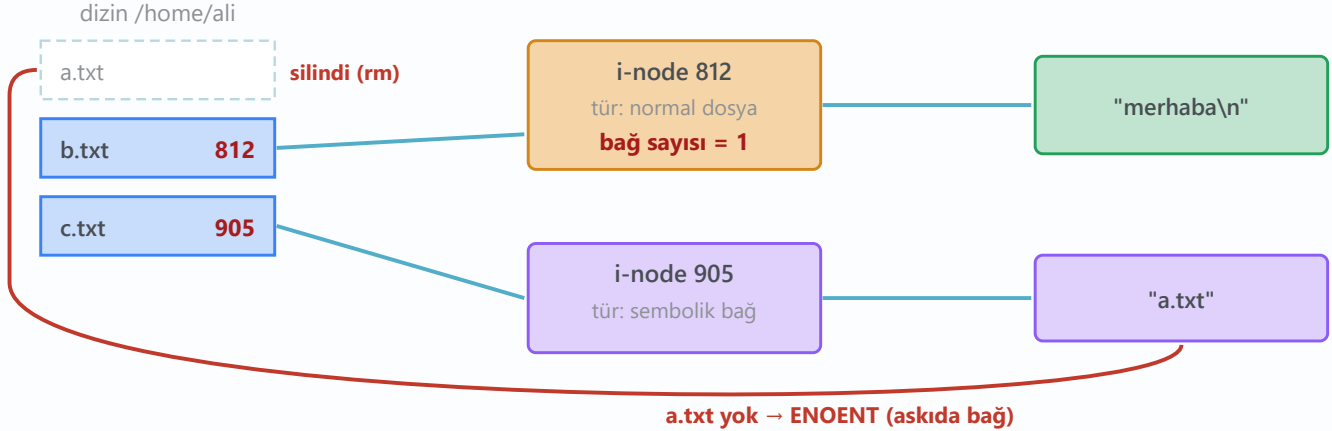


`ln -s a.txt c.txt` (**sembolik bağ**): kendi i-node'u (905) olan küçük bir dosya; içeriği hedefin **yoludur**.

Adım 3 / 4

Katı bağ: aynı dosya sistemi içinde, dizinlere değil, ek maliyet yok. Sembolik bağ: dizinlere, başka dosya sistemlerine, hatta ağ yollarına gösterebilir; her erişimde ek yol çözümü gerekir, döngüleri çekirdek sınırlar (`ELOOP`). Windows: `mklink /H`, `mklink`, junction.

## Katı ve Sembolik Bağlar

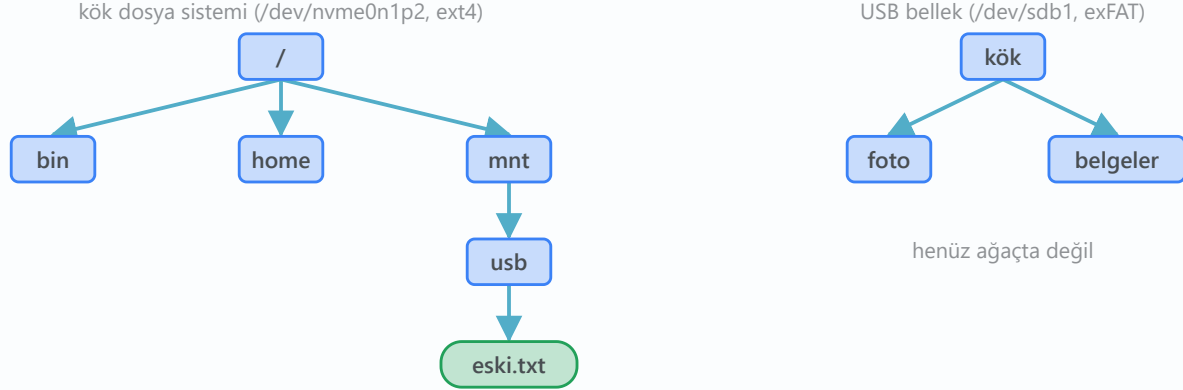


`rm a.txt` : bağ sayısı 1 → veri durur, **b.txt** çalışır. **c.txt** askıda kalır. (Tanenbaum: sahip C silince B'nin bağ kalır ama i-node sahibi hâlâ C'dir — kota C'ye yazılır.)

Adım 4 / 4

Katı bağ: aynı dosya sistemi içinde, dizinlere değil, ek maliyet yok. Sembolik bağ: dizinlere, başka dosya sistemlerine, hatta ağ yollarına gösterebilir; her erişimde ek yol çözümü gerekir, döngüleri çekirdek sınırlar ( `ELOOP` ). Windows: `mklink /H` , `mklink` , junction.

# Bağlama (Mounting)

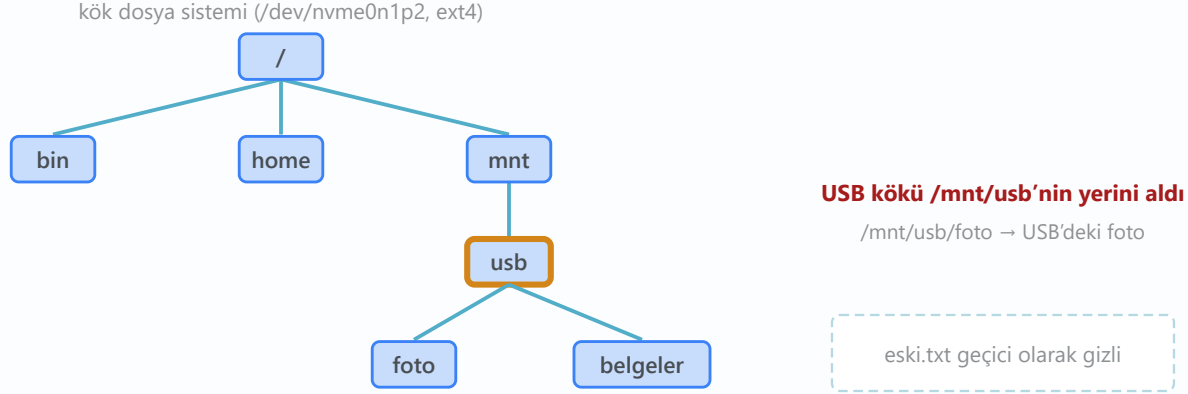


İki ayrı dosya sistemi. `/mnt/usb` sıradan bir dizin ve içinde `eski.txt` var.

Adım 1 / 3

Açılıшта kök dosya sistemi bağlanır, diğerleri `/etc/fstab` veya `systemd` birimleriyle; `findmnt` ağacı gösterir. Windows her birime sürücü harfi ( `C:` , `D:` ) verir, ama NTFS boş bir klasöre de bağlayabilir. Bağlama noktasından geçerken izinler ve `nosuid` , `noexec` , `ro` gibi seçenekler uygulanır.

# Bağlama (Mounting)

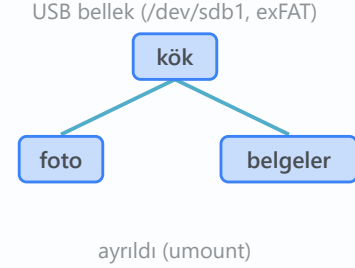
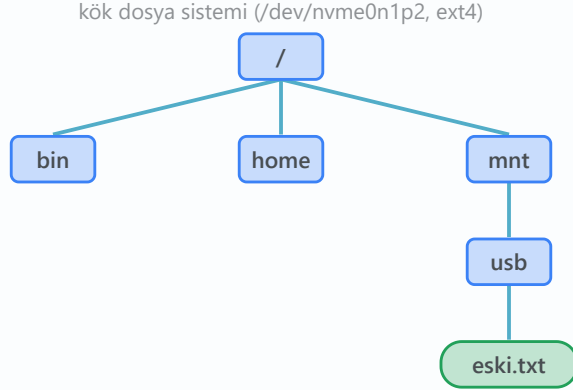


`mount /dev/sdb1 /mnt/usb` : çekirdek bağlama tablosuna kayıt ekler; yol çözümü `/mnt/usb` 'ye gelince USB'nin kök i-node'una atlar. Eski içerik **gizlenir**, silinmez.

Adım 2 / 3

Açılışta kök dosya sistemi bağlanır, diğerleri `/etc/fstab` veya `systemd` birimleriyle; `findmnt` ağacı gösterir. Windows her birime sürücü harfi ( `C:` , `D:` ) verir, ama NTFS boş bir klasöre de bağlayabilir. Bağlama noktasından geçerken izinler ve `nosuid` , `noexec` , `ro` gibi seçenekler uygulanır.

## Bağlama (Mounting)



`umount /mnt/usb` : açık dosya yoksa ayrılır (yoksa `EBUSY` ); `eski.txt` yeniden görünür.

Adım 3 / 3

Açılıştaki kök dosya sistemi bağlanır, diğerleri `/etc/fstab` veya `systemd` birimleriyle; `findmnt` ağacı gösterir. Windows her birime sürücü harfi ( `C:` , `D:` ) verir, ama NTFS boş bir klasöre de bağlayabilir. Bağlama noktasından geçerken izinler ve `nosuid` , `noexec` , `ro` gibi seçenekler uygulanır.

## KORUMA

# Erişim Hakları: rwx

| özel bitler |        |        | sahip (u) |    |    | grup (g) |    |    | diğer (o) |    |    |
|-------------|--------|--------|-----------|----|----|----------|----|----|-----------|----|----|
| setuid      | setgid | sticky | r         | w  | x  | r        | w  | x  | r         | w  | x  |
| 1           | 0      | 0      | 1         | 1  | 1  | 1        | 0  | 1  | 1         | 0  | 1  |
| ×4          | ×2     | ×1     | ×4        | ×2 | ×1 | ×4       | ×2 | ×1 | ×4        | ×2 | ×1 |
| = 4         |        |        | = 7       |    |    | = 5      |    |    | = 5       |    |    |

örnek: **chmod 4755 → -rwsr-xr-x (setuid + rwx r-x r-x)**

| Hak      | Dosyada                 | Dizinde                                                        |
|----------|-------------------------|----------------------------------------------------------------|
| <b>r</b> | içeriği oku             | girdileri listele ( <b>ls</b> )                                |
| <b>w</b> | içeriği değiştir        | girdi ekle/sil/yeniden adlandır (dosyanın kendi izni önemsiz!) |
| <b>x</b> | program olarak çalıştır | içinden geç (yol çözümü, <b>cd</b> )                           |

- Denetim sırası: *sahip mi?* → sahip hakları; değilse *grupta mı?* → grup hakları; değilse diğer. İlk eşleşen kullanılır.
- setuid**: program dosya sahibinin kimliğiyle çalışır ( **passwd** ). **sticky** dizinde: herkes yazabilir ama yalnız sahip siler ( **/tmp** : **drwxrwxrwt** ). Örnek **-rwxrwx--T** : sticky açık, diğerlerinde **x** yok → büyük **T** .

## Erişim Denetim Listeleri (ACL)

```
$ setfacl -m u:ayse:rw- rapor.txt
$ getfacl rapor.txt
# file: rapor.txt
# owner: ali   # group: ogr
user::rw-
user:ayse:rw-    <- adlı kullanıcı
group::r--
mask::rw-        <- üst sınır
other::---
```

### POSIX ACL (Linux)

rwX üçlünün üzerine adlı kullanıcı/grup girdileri;

**mask** grup sınıfının en fazla alabileceği hakkı sınırlar;

**ls** çıktısında **+** görünür.

### NTFS ACL (Windows)

Güvenlik tanımlayıcısında DACL: sıralı *izin ver / reddet* ACE'leri (SID + hak maskesi), açık *reddet* önce gelir;

klasörden **kalıtılır**. Araç: **icacls**.

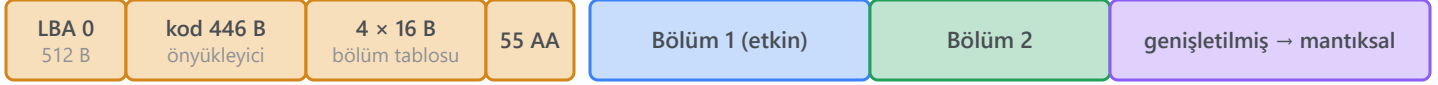
| Yaklaşım               | Artı                           | Eksi                    |
|------------------------|--------------------------------|-------------------------|
| ACL (nesne başına)     | "bu dosyaya kim erişir?" kolay | liste uzar, yönetim zor |
| rwX (sahip/grup/diğer) | 9 bit, hızlı                   | kaba taneli             |
| Yetenek (capability)   | "bu süreç ne yapabilir?" kolay | geri alma zor           |

Paylaşım anlambilimi: UNIX'te yazma diğer süreçlere hemen görünür; AFS gibi dağıtık sistemlerde değişiklik dosya kapatılınca (oturum anlambilimi) yayılır.

# DISK DÜZENİ

## Disk Bölümlleme: MBR ve GPT

### MBR (BIOS, 1983)



en fazla 4 birincil bölüm; 32 bit LBA × 512 B ⇒ 2 TiB sınırı

### GPT (UEFI, 2000'ler)

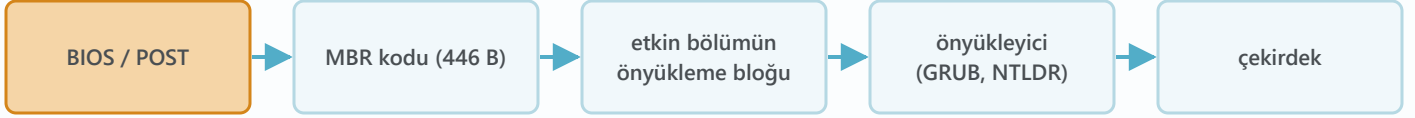


64 bit LBA, 128 bölüm, CRC32 sağlama, disk sonunda yedek; her bölümün GUID'i ve tür GUID'i var

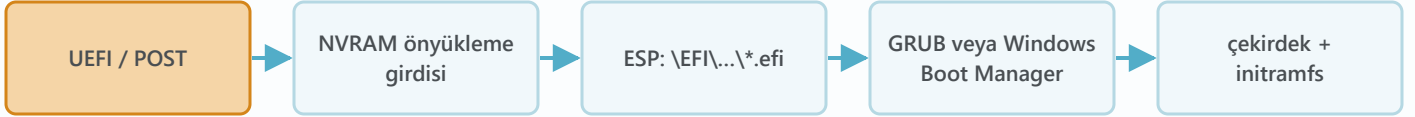
- Disk bir veya daha çok **bölüm** (partition) ayrılır; her bölümde ayrı bir dosya sistemi olabilir (ya da takas alanı, LVM, şifreli kap).
- MBR'nin sonunda bölüm tablosu: her girdi başlangıç ve uzunluk tutar; BIOS yalnızca *etkin* işaretli bölümden açılır.
- **GPT** UEFI standardının parçasıdır; 2 TiB üzeri diskler ve Windows 11 için gereklidir. Koruyucu MBR, eski araçların diski "boş" sanmasını önler.

# Açılış: BIOS/MBR ve UEFI/GPT

BIOS + MBR (eski)



UEFI + GPT (bugün)



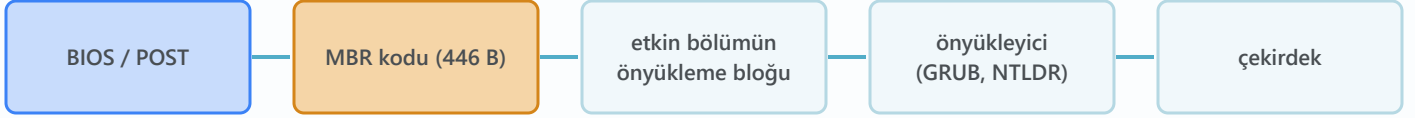
Güç verilince donanım sınaması (POST). BIOS 16 bit gerçek modda çalışır; UEFI 32/64 bit, kendi sürücüleri ve dosya sistemi desteğiyle.

Adım 1 / 5

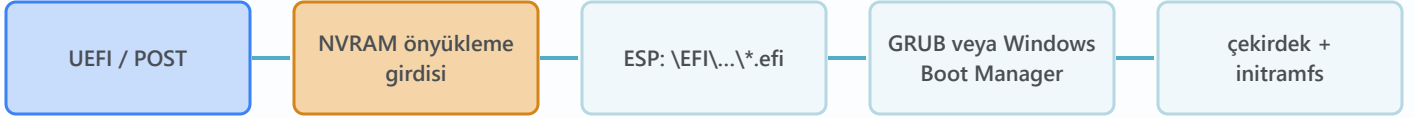
2012 civarından beri PC'ler UEFI ile gelir; 2020'den sonra BIOS uyumluluk modu (CSM) çoğu yeni sistemden kaldırıldı. Ayrıntı: Bölüm 2 (açılış süreci).

# Açılış: BIOS/MBR ve UEFI/GPT

BIOS + MBR (eski)



UEFI + GPT (bugün)



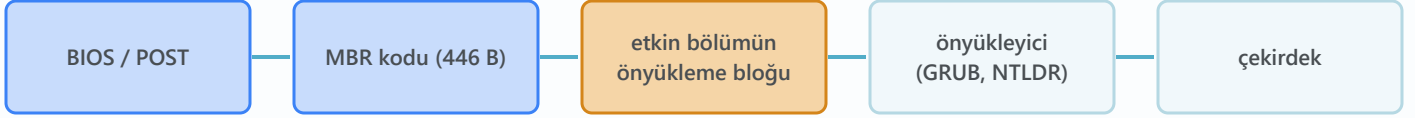
BIOS diskin **ilk sektörünü** (MBR) belleğe alıp çalıştırır. UEFI ise NVRAM'deki önyükleme girdilerini ( `efibootmgr` ) okur; sektör değil **dosya** arar.

Adım 2 / 5

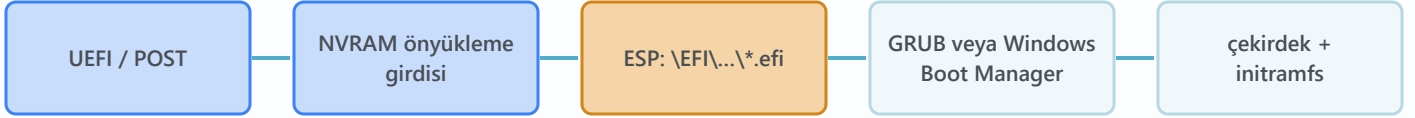
2012 civarından beri PC'ler UEFI ile gelir; 2020'den sonra BIOS uyumluluk modu (CSM) çoğu yeni sistemden kaldırıldı. Ayrıntı: Bölüm 2 (açılış süreci).

# Açılış: BIOS/MBR ve UEFI/GPT

### BIOS + MBR (eski)



### UEFI + GPT (bugün)



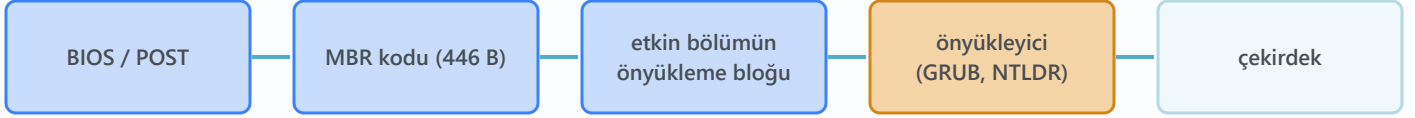
MBR kodu bölüm tablosunda **etkin** bölümü bulur ve onun ilk bloğunu (önyükleme bloğu) çalıştırır. UEFI ESP'deki (FAT32) **.efi** programını yükler; etkin bayrağı kullanılmaz.

Adım 3 / 5

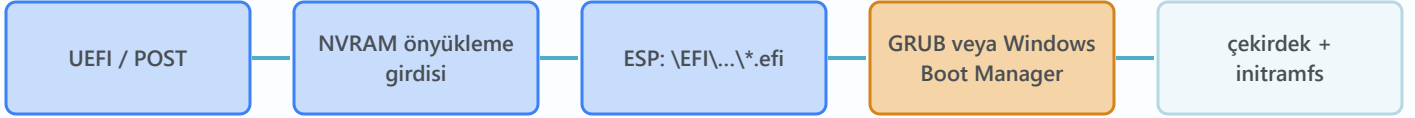
2012 civarından beri PC'ler UEFI ile gelir; 2020'den sonra BIOS uyumluluk modu (CSM) çoğu yeni sistemden kaldırıldı. Ayrıntı: Bölüm 2 (açılış süreci).

# Açılış: BIOS/MBR ve UEFI/GPT

### BIOS + MBR (eski)



### UEFI + GPT (bugün)



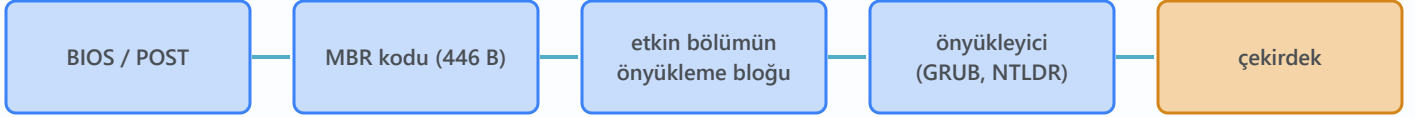
Önyükleyici çekirdeği bulur, belleğe yükler; gerekirse menü sunar. Güvenli Önyükleme (Secure Boot) imzaları bu aşamada doğrular.

Adım 4 / 5

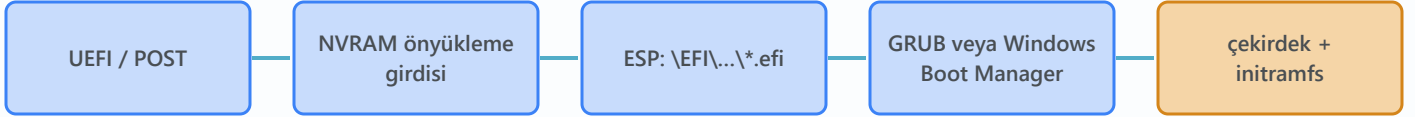
2012 civarından beri PC'ler UEFI ile gelir; 2020'den sonra BIOS uyumluluk modu (CSM) çoğu yeni sistemden kaldırıldı. Ayrıntı: Bölüm 2 (açılış süreci).

# Açılış: BIOS/MBR ve UEFI/GPT

BIOS + MBR (eski)



UEFI + GPT (bugün)



Çekirdek başlar, kök dosya sistemini bağlar (initramfs ile) ve ilk kullanıcı sürecini ( `systemd` ) çalıştırır.

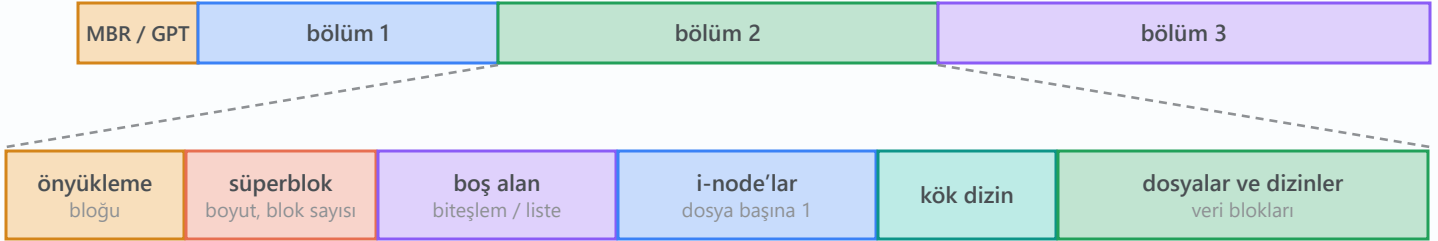
Adım 5 / 5

2012 civarından beri PC'ler UEFI ile gelir; 2020'den sonra BIOS uyumluluk modu (CSM) çoğu yeni sistemden kaldırıldı. Ayrıntı: Bölüm 2 (açılış süreci).

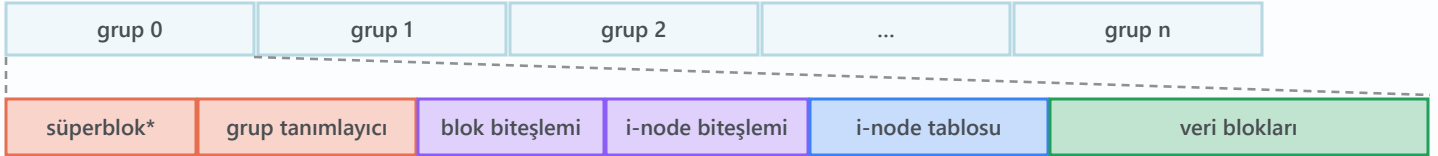
# DISK DÜZENİ

## Dosya Sistemi Düzeni

Disk



ext4: bölüm blok gruplarına ayrılır (her biri 128 MiB, 4 KiB blokta 32768 blok)

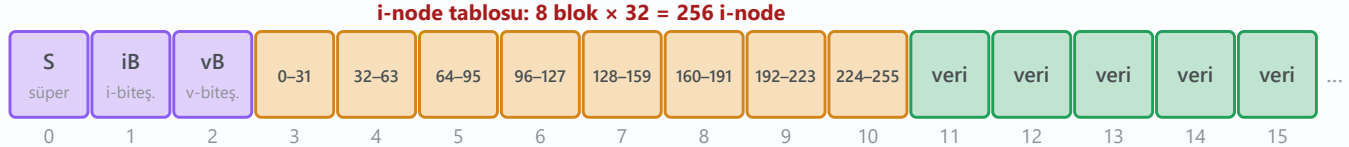


\* süperblok ve tanımlayıcıların yedekleri bazı gruplarda tutulur

- **Süperblok:** dosya sisteminin kimliği (sihirli sayı), blok boyutu, blok ve i-node sayıları, boş sayaçlar, bağlama durumu. Bağlarken ilk okunan yapıdır.
- Önyükleme bloğu *bölümün* ilk bloğudur (diskin ilk bloğu ise MBR'dir); boş olsa da her bölümde yer ayrılır.
- ext4 ilgili i-node'u ve verisini aynı gruba koyarak baş hareketini azaltır (BSD FFS silindir grupları fikri).

# i-node Numarasından Disk Adresine

disk blokları (4 KiB)



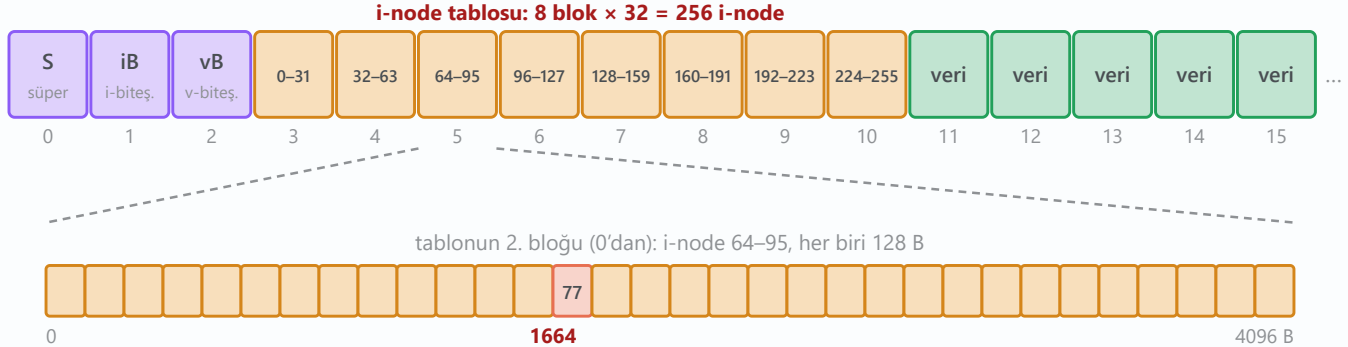
Örnek düzen: 4 KiB blok, 128 B i-node, yani blok başına **32 i-node**. Blok 0 süperblok, 1-2 biteşlemler, 3-10 i-node tablosu. Soru: **i-node 77** (numaralar 0'dan) diskte nerede?

Adım 1 / 4

ext4'te numaralar 1'den başlar ve tablo blok gruplarına bölünür:  $\text{grup} = (n - 1) \div g$ ,  $\text{indis} = (n - 1) \bmod g$ .  $g = 8192$  i-node/grup, 256 B i-node ile i-node 20 000 → grup 2, indis 3615 → grubun tablosunda blok 225, ofset 3840.

# i-node Numarasından Disk Adresine

disk blokları (4 KiB)



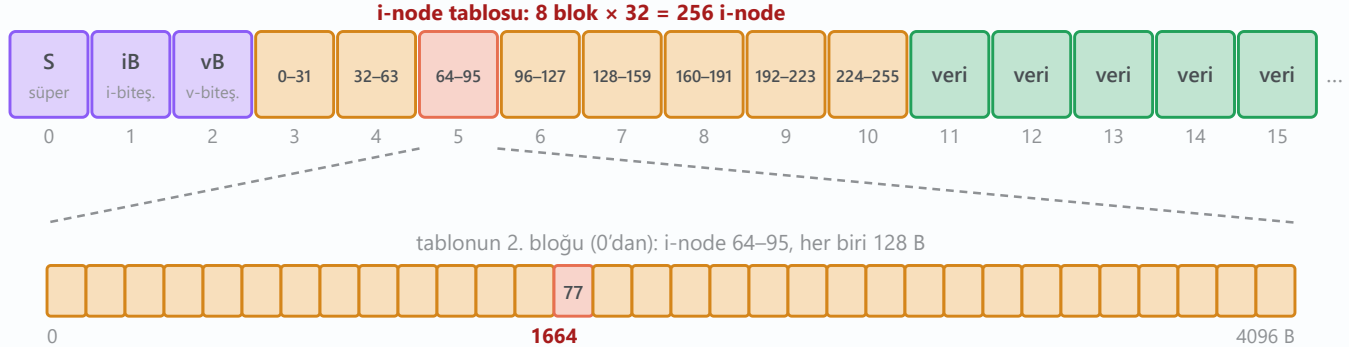
Tablo başından bayt offseti:  $77 \times 128 = \mathbf{9856} = 2 \times 4096 + 1664$ . Yani tablonun **2. bloğu**, blok içinde **1664.** bayt (o bloğun 13. i-node'u).

Adım 2 / 4

ext4'te numaralar 1'den başlar ve tablo blok gruplarına bölünür:  $\text{grup} = (n - 1) \text{ div } g$ ,  $\text{indis} = (n - 1) \text{ mod } g$ .  $g = 8192$  i-node/grup, 256 B i-node ile i-node 20 000 → grup 2, indis 3615 → grubun tablosunda blok 225, offset 3840.

# i-node Numarasından Disk Adresine

disk blokları (4 KiB)



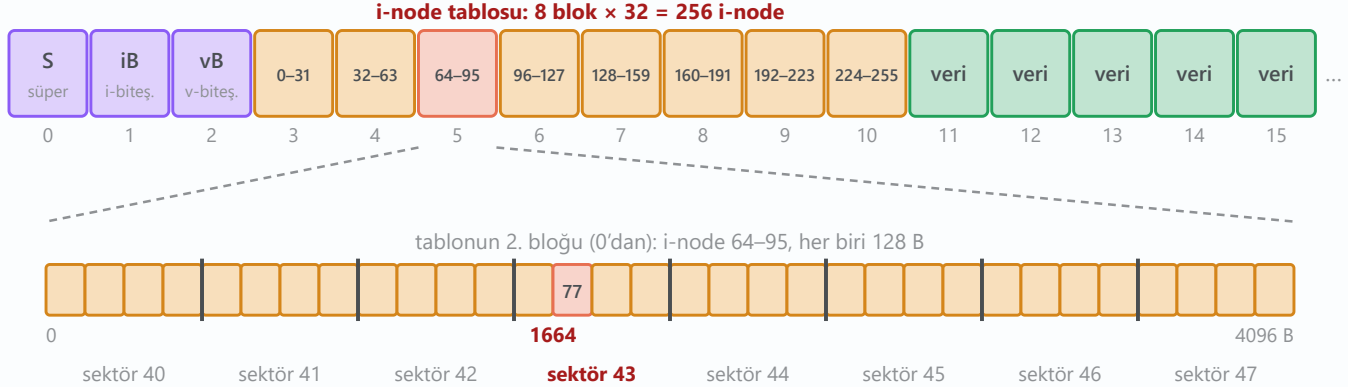
Disk bloğu = tablonun başlangıcı + 2 = 3 + 2 = **blok 5**. Süperbloktaki “i-node tablosu nerede” bilgisi bu yüzden gereklidir.

Adım 3 / 4

ext4'te numaralar 1'den başlar ve tablo blok gruplarına bölünür:  $\text{grup} = (n - 1) \div g$ ,  $\text{indis} = (n - 1) \bmod g$ .  $g = 8192$  i-node/grup, 256 B i-node ile i-node 20 000 → grup 2, indis 3615 → grubun tablosunda blok 225, ofset 3840.

## i-node Numarasından Disk Adresine

disk blokları (4 KiB)

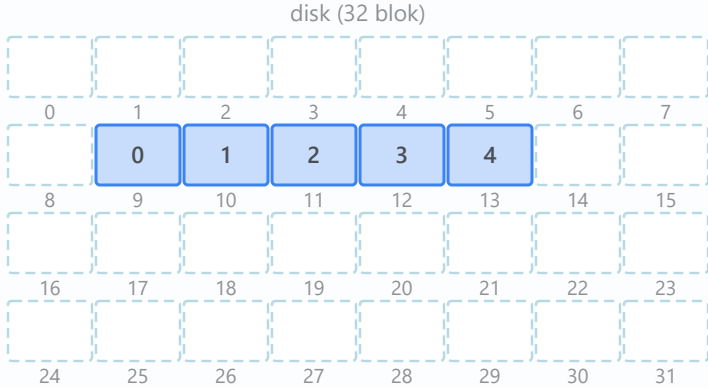


Disk sektör okur: bayt adresi  $5 \times 4096 + 1664 = 22\,144 \rightarrow 22\,144 / 512 = \textbf{sektör 43}$ , içinde 128. bayt. Bir sektörde 4 i-node vardır.

Adım 4 / 4

ext4'te numaralar 1'den başlar ve tablo blok gruplarına bölünür:  $\text{grup} = (n - 1) \text{ div } g$ ,  $\text{indis} = (n - 1) \text{ mod } g$ .  $g = 8192$  i-node/grup, 256 B i-node ile i-node 20 000  $\rightarrow$  grup 2, indis 3615  $\rightarrow$  grubun tablosunda blok 225, ofset 3840.

## Tahsis Yöntemleri: Aynı Dosya, Dört Yol



dizin girdisi

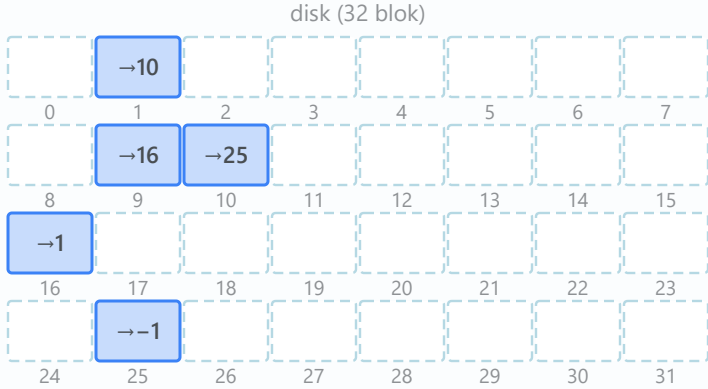
| ad   | başlangıç | uzunluk |
|------|-----------|---------|
| jeep | 9         | 5       |

**blok i = başlangıç + i**

tek arama, ardışık okuma

**Bitişik:** 5 blok yan yana (9–13). Dizin girdisinde yalnız başlangıç ve uzunluk.

## Tahsis Yöntemleri: Aynı Dosya, Dört Yol



dizin girdisi

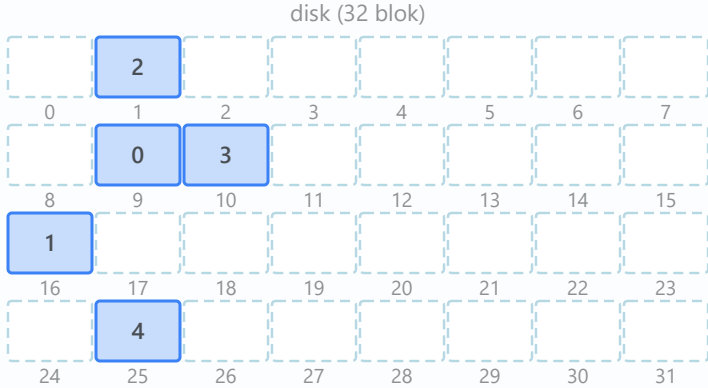
| ad   | ilk | son |
|------|-----|-----|
| jeep | 9   | 25  |

her blok sonrakinin numarasını taşır

**9 → 16 → 1 → 10 → 25 → -1**

**Bağlı liste:** bloklar diske dağılmış (9, 16, 1, 10, 25); her blok sonrakini gösterir.

# Tahsis Yöntemleri: Aynı Dosya, Dört Yol



dizin girdisi

| ad   | ilk blok | boyut  |
|------|----------|--------|
| jeep | 9        | 5 blok |

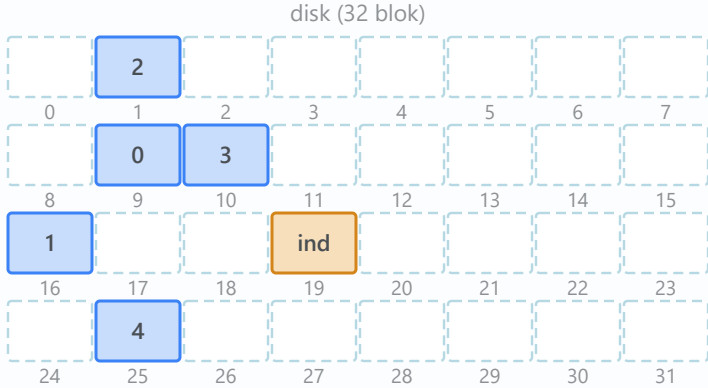
  

|    |     |                |
|----|-----|----------------|
| 1  | 10  | FAT (bellekte) |
| 9  | 16  |                |
| 10 | 25  |                |
| 16 | 1   |                |
| 25 | EOF |                |

zincir tabloda

**FAT:** aynı zincir, ama işaretçiler bloklarda değil bellekteki **tabloda**; bloklar tamamen veri.

## Tahsis Yöntemleri: Aynı Dosya, Dört Yol



dizin girdisi

| ad   | indis bloğu | boyut  |
|------|-------------|--------|
| jeep | 19          | 5 blok |

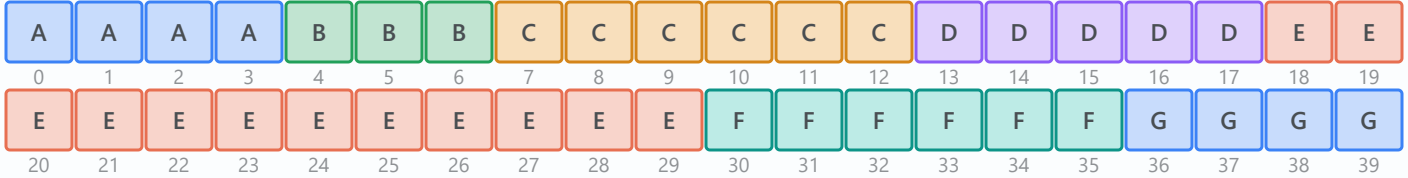
indis bloğu 19

|    |    |    |    |
|----|----|----|----|
| 9  | 16 | 1  | 10 |
| 25 | -1 | -1 |    |

**i. girdi = i. bloğun adresi**

**İndisli:** tüm adresler tek bir **indis bloğunda** (19); i. bloğa doğrudan erişilir.

## Bitişik Tahsis



7 dosya:  $A4 \cdot B3 \cdot C5 \cdot D5 \cdot E2 \cdot F6 \cdot G4 = 40$  blok, disk dolu

Her dosya ardışık bloklara yerleştirildi; dizin girdisi = (başlangıç, uzunluk).

Adım 1 / 4

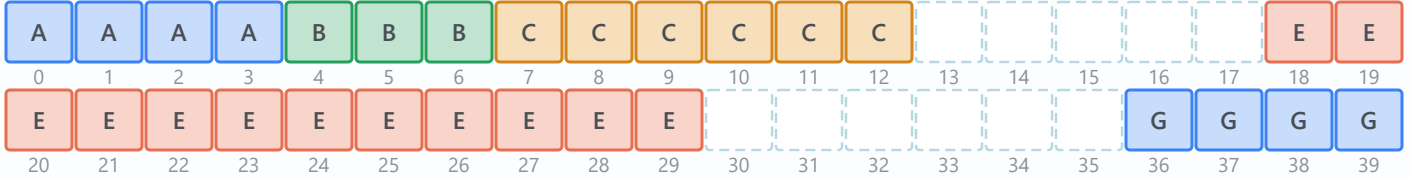
### Artılar

Basit;  $i$ . blok = başlangıç +  $i$ , rastgele erişim kolay; tüm dosya tek aramayla, art arda okunur (HDD'de en iyi performans).

### Eksiler

**Dış** parçalanma; dosya büyüyemez ya da taşınmalı. Sabit içerikli ortamlara uygun: CD-ROM/DVD (ISO 9660, UDF).

## Bitişik Tahsis



D ve F silindi: 5 ve 6 blokluk iki delik (toplam 11 boş blok)

Silinen dosyalar **delik** bırakır; boş alan parçalara bölünür.

Adım 2 / 4

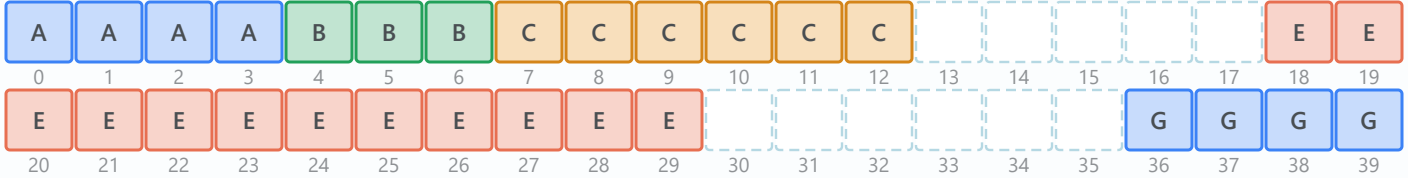
### Artılar

Basit;  $i$ . blok = başlangıç +  $i$ , rastgele erişim kolay; tüm dosya tek aramayla, art arda okunur (HDD'de en iyi performans).

### Eksiler

**Dış** parçalanma; dosya büyüyemez ya da taşınmalı. Sabit içerikli ortamlara uygun: CD-ROM/DVD (ISO 9660, UDF).

## Bitişik Tahsis



**H 8 blok istiyor: toplam 11 boş, ama en büyük delik 6 → sığmaz!**

**dış parçalanma (external fragmentation)**

Yeni dosya hiçbir deliğe sığmaz: bellek yönetimindeki dış parçalanmanın aynısı. Üstelik dosyanın son boyutu baştan bilinmelidir.

Adım 3 / 4

### Artılar

Basit;  $i$ . blok = başlangıç +  $i$ , rastgele erişim kolay; tüm dosya tek aramayla, art arda okunur (HDD'de en iyi performans).

### Eksiler

**Dış** parçalanma; dosya büyüyemez ya da taşınmalı. Sabit içerikli ortamlara uygun: CD-ROM/DVD (ISO 9660, UDF).

## Bitişik Tahsis



**sıkıştırma: G ve E kaydırıldı, tek bir 11 blokluk delik → H yerleşti**

bedeli: diskte 16 bloğun yeniden yazılması

Çözüm: birleştirme/sıkıştırma (pahalı) ya da dosyayı birden çok bitişik parçaya, **extent**'lere bölmek.

Adım 4 / 4

### Artılar

Basit;  $i$ . blok = başlangıç +  $i$ , rastgele erişim kolay; tüm dosya tek aramayla, art arda okunur (HDD'de en iyi performans).

### Eksiler

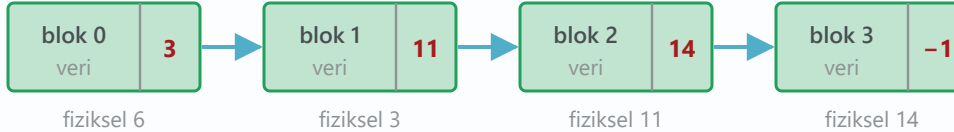
**Dış** parçalanma; dosya büyüyemez ya da taşınmalı. Sabit içerikli ortamlara uygun: CD-ROM/DVD (ISO 9660, UDF).

## Bağlı Liste ile Tahsis

Dosya A



Dosya B



Dizin girdisi yalnız ilk bloğu (A için 4) bilir. Her bloğun ilk/son birkaç baytı sonraki bloğun numarasıdır.

Adım 1 / 5

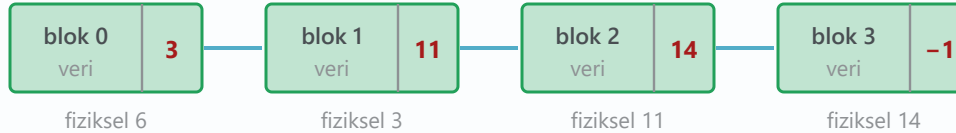
- Dış parçalanma **yoktur**, dosya kolayca büyür; iç parçalanma (son blok) her yöntemde vardır.
- Bloktaki veri miktarı ikinin kuvveti değildir (4096 – 4 bayt); tek bir işaretçi bozulursa zincirin geri kalanı kaybolur.

## Bağlı Liste ile Tahsis

Dosya A



Dosya B



okunan: 1 blok

Mantıksal **3. bloğu** okumak istiyoruz: önce fiziksel 4 okunur → sonraki 7.

Adım 2 / 5

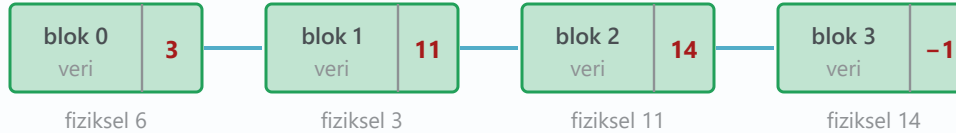
- Dış parçalanma **yoktur**, dosya kolayca büyür; iç parçalanma (son blok) her yöntemde vardır.
- Bloktaki veri miktarı ikinin kuvveti değildir ( $4096 - 4$  bayt); tek bir işaretçi bozulursa zincirin geri kalanı kaybolur.

## Bağlı Liste ile Tahsis

Dosya A



Dosya B



okunan: 2 blok

Fiziksel 7 okunur → sonraki 2.

Adım 3 / 5

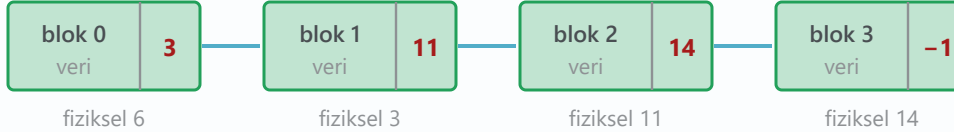
- Dış parçalanma **yoktur**, dosya kolayca büyür; iç parçalanma (son blok) her yöntemde vardır.
- Bloktaki veri miktarı ikinin kuvveti değildir ( $4096 - 4$  bayt); tek bir işaretçi bozulursa zincirin geri kalanı kaybolur.

## Bağlı Liste ile Tahsis

Dosya A



Dosya B



okunan: 3 blok

Fiziksel 2 okunur → sonraki 10.

Adım 4 / 5

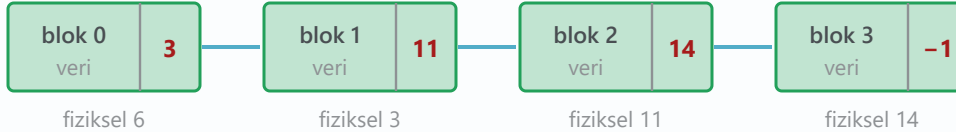
- Dış parçalanma **yoktur**, dosya kolayca büyür; iç parçalanma (son blok) her yöntemde vardır.
- Bloktaki veri miktarı ikinin kuvveti değildir ( $4096 - 4$  bayt); tek bir işaretçi bozulursa zincirin geri kalanı kaybolur.

## Bağlı Liste ile Tahsis

Dosya A



Dosya B



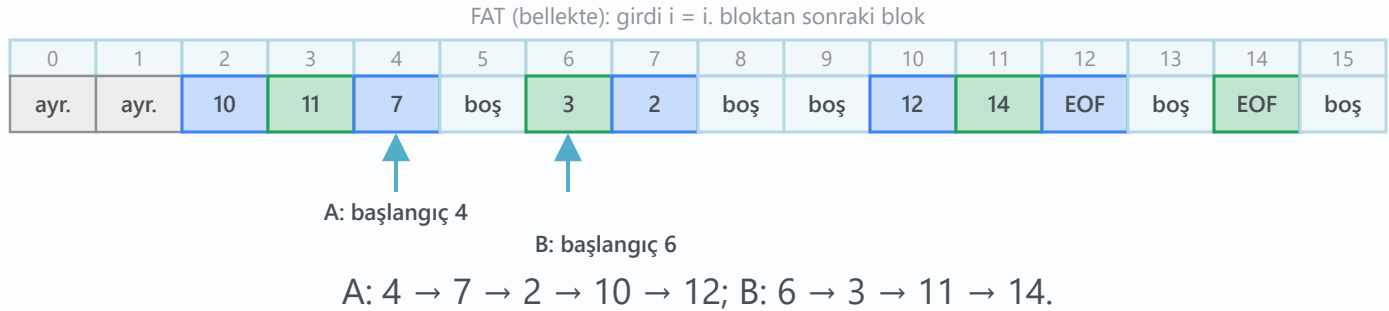
**okunan: 4 blok**

Fiziksel 10 = mantıksal 3: **4 disk okuması**. Rastgele erişim  $O(n)$ ; HDD'de her biri ayrı arama olabilir.

Adım 5 / 5

- Dış parçalanma **yoktur**, dosya kolayca büyür; iç parçalanma (son blok) her yöntemde vardır.
- Bloktaki veri miktarı ikinin kuvveti değildir ( $4096 - 4$  bayt); tek bir işaretçi bozulursa zincirin geri kalanı kaybolur.

## FAT: Tablo ile Tahsis

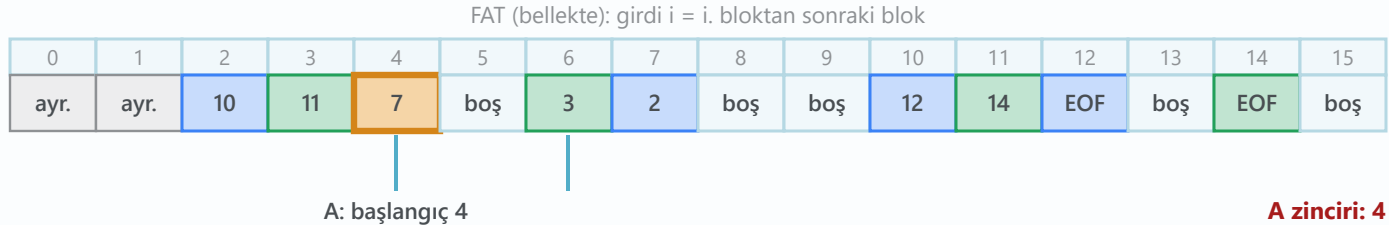


Adım 1 / 6

- Bağlı listedeki işaretçiler bloklardan alınıp tek bir **dosya tahsis tablosuna** (FAT) taşınır; blokların tamamı veriye kalır.
- Rastgele erişim hâlâ zincir izler, ama bellekte: hızlı. Tablo diskte iki kopya tutulur.
- Sorun:** tablo diskle orantılı büyür ve tamamı bellekte olmalıdır.

Örnek: 200 GiB disk, 1 KiB blok →  $200 \times 2^{20}$   
= **209.715.200** girdi; 4 B/girdi → **800 MiB**  
(3 B ile 600 MiB). FAT32'de 2 TiB birim, 32  
KiB küme → 67.108.864 girdi = 256 MiB.

## FAT: Tablo ile Tahsis



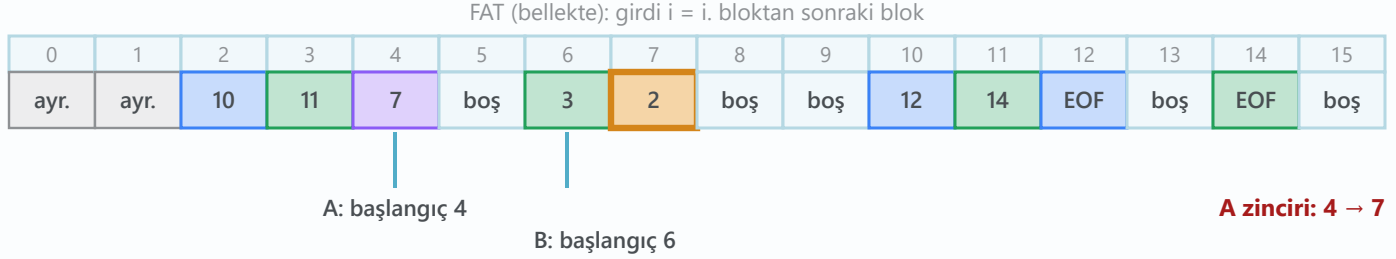
Girdi 4 = 7.

Adım 2 / 6

- Bağlı listedeki işaretçiler bloklardan alınıp tek bir **dosya tahsis tablosuna** (FAT) taşınır; blokların tamamı veriye kalır.
- Rastgele erişim hâlâ zincir izler, ama bellekte: hızlı. Tablo diskte iki kopya tutulur.
- Sorun:** tablo diskle orantılı büyür ve tamamı bellekte olmalıdır.

Örnek: 200 GiB disk, 1 KiB blok  $\rightarrow 200 \times 2^{20}$   
= **209.715.200** girdi; 4 B/girdi  $\rightarrow$  **800 MiB**  
(3 B ile 600 MiB). FAT32'de 2 TiB birim, 32  
KiB küme  $\rightarrow 67.108.864$  girdi = 256 MiB.

## FAT: Tablo ile Tahsis



Girdi 7 = 2.

Adım 3 / 6

- Bağlı listedeki işaretçiler bloklardan alınıp tek bir **dosya tahsis tablosuna** (FAT) taşınır; blokların tamamı veriye kalır.
- Rastgele erişim hâlâ zincir izler, ama bellekte: hızlı. Tablo diskte iki kopya tutulur.
- Sorun:** tablo diskle orantılı büyür ve tamamı bellekte olmalıdır.

Örnek: 200 GiB disk, 1 KiB blok →  $200 \times 2^{20}$   
= **209.715.200** girdi; 4 B/girdi → **800 MiB**  
(3 B ile 600 MiB). FAT32'de 2 TiB birim, 32  
KiB küme → 67.108.864 girdi = 256 MiB.

## FAT: Tablo ile Tahsis



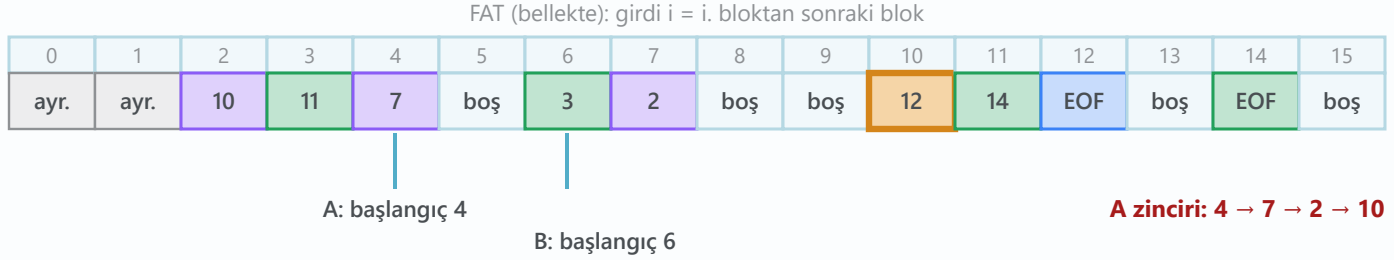
Girdi 2 = 10.

Adım 4 / 6

- Bağlı listedeki işaretçiler bloklardan alınıp tek bir **dosya tahsis tablosuna** (FAT) taşınır; blokların tamamı veriye kalır.
- Rastgele erişim hâlâ zincir izler, ama bellekte: hızlı. Tablo diskte iki kopya tutulur.
- Sorun:** tablo diskle orantılı büyür ve tamamı bellekte olmalıdır.

Örnek: 200 GiB disk, 1 KiB blok →  $200 \times 2^{20}$   
= **209.715.200** girdi; 4 B/girdi → **800 MiB**  
(3 B ile 600 MiB). FAT32'de 2 TiB birim, 32  
KiB küme → 67.108.864 girdi = 256 MiB.

## FAT: Tablo ile Tahsis



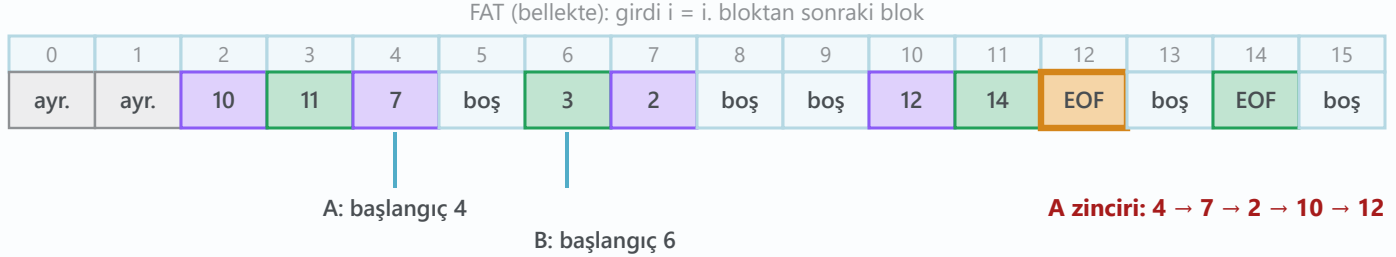
Girdi 10 = 12.

Adım 5 / 6

- Bağlı listedeki işaretçiler bloklardan alınıp tek bir **dosya tahsis tablosuna** (FAT) taşınır; blokların tamamı veriye kalır.
- Rastgele erişim hâlâ zincir izler, ama bellekte: hızlı. Tablo diskte iki kopya tutulur.
- Sorun:** tablo diskle orantılı büyür ve tamamı bellekte olmalıdır.

Örnek: 200 GiB disk, 1 KiB blok →  $200 \times 2^{20}$   
= **209.715.200** girdi; 4 B/girdi → **800 MiB**  
(3 B ile 600 MiB). FAT32'de 2 TiB birim, 32  
KiB küme → 67.108.864 girdi = 256 MiB.

## FAT: Tablo ile Tahsis



Girdi 12 = EOF. Zincir **bellekte** izlendi: disk okuması yok.

Adım 6 / 6

- Bağlı listedeki işaretçiler bloklardan alınıp tek bir **dosya tahsis tablosuna** (FAT) taşınır; blokların tamamı veriye kalır.
- Rastgele erişim hâlâ zincir izler, ama bellekte: hızlı. Tablo diskte iki kopya tutulur.
- Sorun:** tablo diskle orantılı büyür ve tamamı bellekte olmalıdır.

Örnek: 200 GiB disk, 1 KiB blok →  $200 \times 2^{20}$   
= **209.715.200** girdi; 4 B/girdi → **800 MiB**  
(3 B ile 600 MiB). FAT32'de 2 TiB birim, 32  
KiB küme → 67.108.864 girdi = 256 MiB.

# FAT Zinciri: Kod İzleme

```
1 int blok_bul(int ilk, long ofset) {  
  ► int b = ilk;  
  ► long n = ofset / BLOK; /* blok sırası */  
  4 while (n > 0 && b != SON) {  
    5   b = fat[b];          /* zinciri izle */  
    6   n--;  
    7 }  
  8 return b;               /* SON: dosya dışı */  
  9 }
```

FAT (bellekte) · A: 4→7→2→10→12 · B: 6→3→11→14

|   |   |    |    |     |    |     |    |
|---|---|----|----|-----|----|-----|----|
| 0 | 1 | 2  | 3  | 4   | 5  | 6   | 7  |
| . | . | 10 | 11 | 7   | .  | 3   | 2  |
| 8 | 9 | 10 | 11 | 12  | 13 | 14  | 15 |
| . | . | 12 | 14 | SON | .  | SON | .  |

izlenen: 4

mavi: dosya blokları · yeşil: izlenen · turuncu: b

**blok\_bul(4, 3500)** : A dosyası blok 4'ten başlıyor.  
 $3500 / 1024 = 3$ : aranan, dosyanın 3. bloğu (0'dan  
sayınca); zincirde 3 adım ilerlenmeli.

Adım 1 / 11

## DEĞİŞKENLER

ilk = 4

ofset = 3500

n = 3

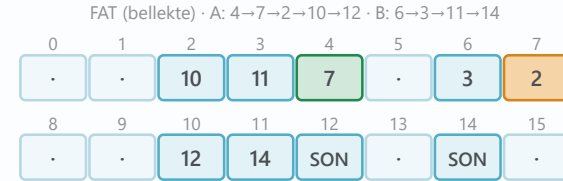
b = 4

## ÇIKTI



# FAT Zinciri: Kod İzleme

```
1 int blok_bul(int ilk, long ofset) {  
2     int b = ilk;  
3     long n = ofset / BLOK; /* blok sırası */  
▶ while (n > 0 && b != SON) {  
▶     b = fat[b];           /* zinciri izle */  
▶     n--;  
7 }  
8 return b;                /* SON: dosya dışı */  
9 }
```



izlenen: 4 → 7

mavi: dosya blokları · yeşil: izlenen · turuncu: b

fat[4] = 7: sonraki blok 7; n = 2.

Adım 2 / 11

## DEĞİŞKENLER

ilk = 4

ofset = 3500

n = 2

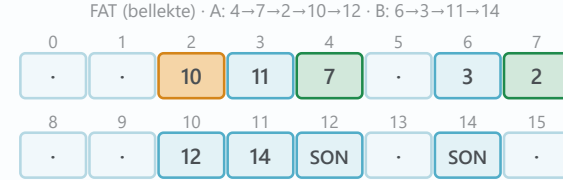
b = 7

## ÇIKTI



# FAT Zinciri: Kod İzleme

```
1 int blok_bul(int ilk, long ofset) {  
2   int b = ilk;  
3   long n = ofset / BLOK; /* blok sırası */  
▶ while (n > 0 && b != SON) {  
▶   b = fat[b];           /* zinciri izle */  
▶   n--;  
7 }  
8 return b;               /* SON: dosya dışı */  
9 }
```



izlenen: 4 → 7 → 2

mavi: dosya blokları · yeşil: izlenen · turuncu: b

fat[7] = 2: sonraki blok 2; n = 1.

Adım 3 / 11

## DEĞİŞKENLER

ilk = 4

ofset = 3500

n = 1

b = 2

## ÇIKTI



# FAT Zinciri: Kod İzleme

```
1 int blok_bul(int ilk, long ofset) {  
2     int b = ilk;  
3     long n = ofset / BLOK; /* blok sırası */  
4     while (n > 0 && b != SON) {  
5         b = fat[b];          /* zinciri izle */  
6         n--;  
7     }  
8     return b;                /* SON: dosya dışı */  
9 }
```

FAT (bellekte) · A: 4→7→2→10→12 · B: 6→3→11→14



izlenen: 4 → 7 → 2 → 10

mavi: dosya blokları · yeşil: izlenen · turuncu: b

fat[2] = 10: sonraki blok 10; n = 0.

Adım 4 / 11

## DEĞİŞKENLER

ilk = 4

ofset = 3500

n = 0

b = 10

## ÇIKTI



# FAT Zinciri: Kod İzleme

```

1 int blok_bul(int ilk, long ofset) {
2   int b = ilk;
3   long n = ofset / BLOK; /* blok sırası */
4   while (n > 0 && b != SON) {
5     b = fat[b];           /* zinciri izle */
6     n--;
7   }
8   return b;              /* SON: dosya dışı */
9 }

```

FAT (bellekte) · A: 4→7→2→10→12 · B: 6→3→11→14



izlenen: 4 → 7 → 2 → 10

mavi: dosya blokları · yeşil: izlenen · turuncu: b

n = 0: döngü biter, disk bloğu **10**; bayt 3500 bu bloğun  
 $3500 - 3 \cdot 1024 = 428$ . baytıdır. Üç adım da bellekte: disk  
 okuması yok.

Adım 5 / 11

## DEĞİŞKENLER

ilk = 4

ofset = 3500

n = 0

b = 10

## ÇIKTI

`blok_bul(4, 3500) = 10`

# FAT Zinciri: Kod İzleme

```
1 int blok_bul(int ilk, long ofset) {  
  ▶ int b = ilk;  
  ▶ long n = ofset / BLOK; /* blok sırası */  
  4 while (n > 0 && b != SON) {  
    5   b = fat[b];          /* zinciri izle */  
    6   n--;  
    7 }  
  8 return b;              /* SON: dosya dışı */  
  9 }
```

FAT (bellekte) · A: 4→7→2→10→12 · B: 6→3→11→14

|   |   |    |    |     |    |     |    |
|---|---|----|----|-----|----|-----|----|
| 0 | 1 | 2  | 3  | 4   | 5  | 6   | 7  |
| . | . | 10 | 11 | 7   | .  | 3   | 2  |
| 8 | 9 | 10 | 11 | 12  | 13 | 14  | 15 |
| . | . | 12 | 14 | SON | .  | SON | .  |

izlenen: 6

mavi: dosya blokları · yeşil: izlenen · turuncu: b

`blok_bul(6, 5000)` : B dosyası 4 blok (en çok 4096 bayt) ama  $5000 / 1024 = 4$ . blok isteniyor.

Adım 6 / 11

## DEĞİŞKENLER

ilk = 6

ofset = 5000

n = 4

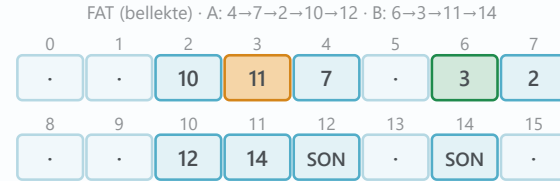
b = 6

## ÇIKTI

`blok_bul(4, 3500) = 10`

# FAT Zinciri: Kod İzleme

```
1 int blok_bul(int ilk, long ofset) {  
2   int b = ilk;  
3   long n = ofset / BLOK; /* blok sırası */  
4   while (n > 0 && b != SON) {  
5     b = fat[b];           /* zinciri izle */  
6     n--;  
7   }  
8   return b;               /* SON: dosya dışı */  
9 }
```



izlenen: 6 → 3

mavi: dosya blokları · yeşil: izlenen · turuncu: b

fat[6] = 3: sonraki blok 3; n = 3.

Adım 7 / 11

## DEĞİŞKENLER

ilk = 6

ofset = 5000

n = 3

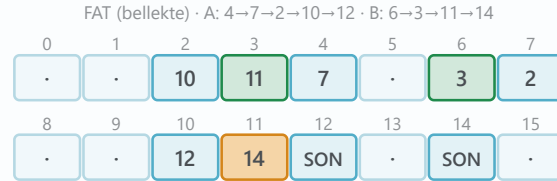
b = 3

## ÇIKTI

```
blok_bul(4, 3500) = 10
```

# FAT Zinciri: Kod İzleme

```
1 int blok_bul(int ilk, long ofset) {  
2   int b = ilk;  
3   long n = ofset / BLOK; /* blok sırası */  
▶ while (n > 0 && b != SON) {  
▶   b = fat[b];           /* zinciri izle */  
▶   n--;  
7 }  
8 return b;               /* SON: dosya dışı */  
9 }
```



izlenen: 6 → 3 → 11

mavi: dosya blokları · yeşil: izlenen · turuncu: b

fat[3] = 11: sonraki blok 11; n = 2.

Adım 8 / 11

## DEĞİŞKENLER

ilk = 6

ofset = 5000

n = 2

b = 11

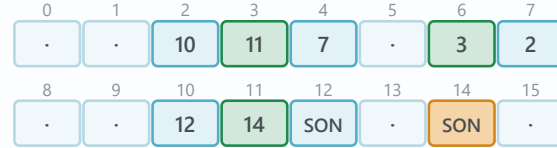
## ÇIKTI

```
blok_bul(4, 3500) = 10
```

# FAT Zinciri: Kod İzleme

```
1 int blok_bul(int ilk, long ofset) {  
2   int b = ilk;  
3   long n = ofset / BLOK; /* blok sırası */  
▶ while (n > 0 && b != SON) {  
▶   b = fat[b];           /* zinciri izle */  
▶   n--;  
7 }  
8 return b;               /* SON: dosya dışı */  
9 }
```

FAT (bellekte) · A: 4→7→2→10→12 · B: 6→3→11→14



izlenen: 6 → 3 → 11 → 14

mavi: dosya blokları · yeşil: izlenen · turuncu: b

fat[11] = 14; sonraki blok 14; n = 1.

Adım 9 / 11

## DEĞİŞKENLER

ilk = 6

ofset = 5000

n = 1

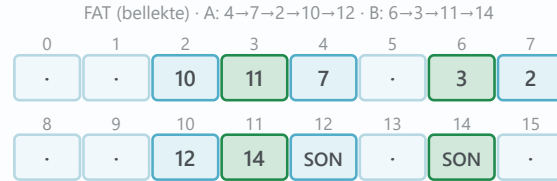
b = 14

## ÇIKTI

blok\_bul(4, 3500) = 10

# FAT Zinciri: Kod İzleme

```
1 int blok_bul(int ilk, long ofset) {  
2   int b = ilk;  
3   long n = ofset / BLOK; /* blok sırası */  
▶ while (n > 0 && b != SON) {  
▶   b = fat[b];           /* zinciri izle */  
▶   n--;  
7 }  
8 return b;               /* SON: dosya dışı */  
9 }
```



izlenen: 6 → 3 → 11 → 14

mavi: dosya blokları · yeşil: izlenen · turuncu: b

fat[14] = SON: zincir bitti, dosyanın sonuna gelindi; n = 0.

Adım 10 / 11

## DEĞİŞKENLER

ilk = 6

ofset = 5000

n = 0

b = SON

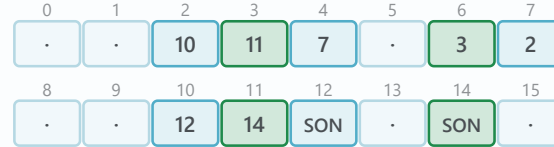
## ÇIKTI

```
blok_bul(4, 3500) = 10
```

# FAT Zinciri: Kod İzleme

```
1 int blok_bul(int ilk, long ofset) {  
2   int b = ilk;  
3   long n = ofset / BLOK; /* blok sırası */  
4   while (n > 0 && b != SON) {  
5     b = fat[b];           /* zinciri izle */  
6     n--;  
7   }  
8   return b;               /* SON: dosya dışı */  
9 }
```

FAT (bellekte) · A: 4→7→2→10→12 · B: 6→3→11→14



izlenen: 6 → 3 → 11 → 14

mavi: dosya blokları · yeşil: izlenen · turuncu: b

Dördüncü adımda b = SON: dosyanın 4. bloğu yok, ofset dosyanın dışında. `read` 0 bayt döndürür (dosya sonu); `write` ise yeni bir blok ayırıp 14'ün girdisine bağlar.

Adım 11 / 11

## DEĞİŞKENLER

ilk = 6

ofset = 5000

n = 0

b = SON

## ÇIKTI

blok\_bul(4, 3500) = 10

blok\_bul(6, 5000) = SON

## TAHSİS YÖNTEMLERİ

# FAT Ailesi: FAT12/16/32 ve exFAT

MS-DOS dizin girdisi (32 bayt)

| Ad<br>8 B | Uzantı<br>3 B | Öz<br>1 B | Ayrılmış<br>10 B | Saat<br>2 B | Tarih<br>2 B | Küme<br>2 B | Boyut<br>4 B |
|-----------|---------------|-----------|------------------|-------------|--------------|-------------|--------------|
|-----------|---------------|-----------|------------------|-------------|--------------|-------------|--------------|

Öz: öznitelik. FAT32: "ayrılmış" alana oluşturma/erişim zamanı ve ilk kümenin üst 16 biti yerleşti; uzun adlar ek girdilerde.

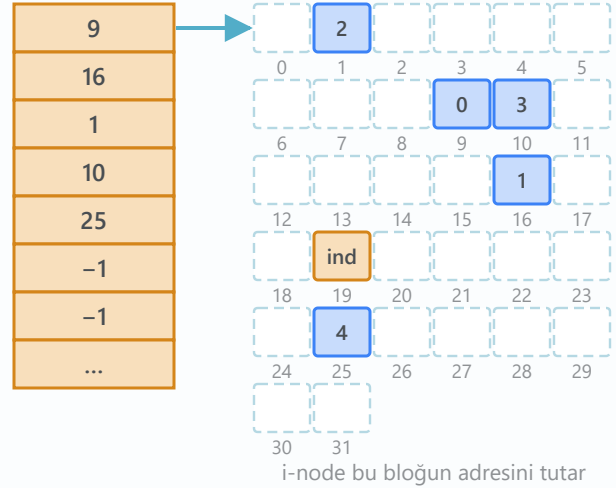
|                | FAT12       | FAT16               | FAT32                   | exFAT                |
|----------------|-------------|---------------------|-------------------------|----------------------|
| FAT girdisi    | 12 bit      | 16 bit              | 28 bit kullanılır       | 32 bit               |
| En büyük birim | ≈ 16 MiB    | 2 GiB (32 KiB küme) | 2 TiB (512 B sektör)    | 128 PiB              |
| En büyük dosya | birim kadar | 2 GiB               | <b>4 GiB – 1 bayt</b>   | 16 EiB (kuramsal)    |
| Boş alan       | FAT'ta 0    | FAT'ta 0            | FAT'ta 0 + FSInfo ipucu | ayrı <b>biteşlem</b> |
| Kullanım       | disket      | eski diskler        | USB, UEFI ESP           | SDXC/SDUC, büyük USB |

Boyut alanı 32 bit olduğundan FAT32'de 4 GiB'lık dosya sınırı vardır; exFAT bunu kaldırır ve bitişik dosyalarda FAT zincirini hiç kullanmayabilir. Windows'un grafik biçimlendirme aracı FAT32'yi tarihsel olarak 32 GB ile sınırlar; bu dosya sisteminin değil, aracın sınırıdır.

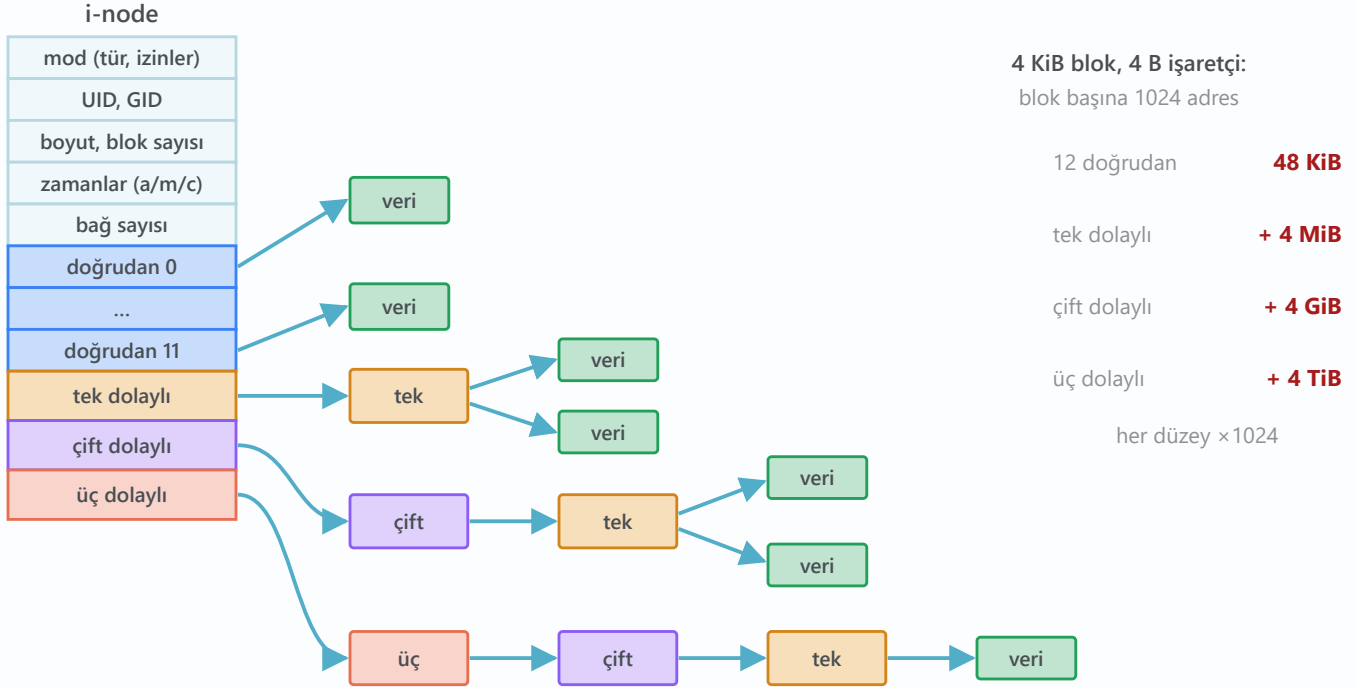
# İndisli Tahsis ve i-node

- Her dosyanın blok adresleri bir **indis bloğunda** (UNIX'te **i-node**) toplanır: i. girdi i. bloğun adresidir. Rastgele erişim hızlı, dış parçalanma yok.
- i-node yalnızca dosya **açıkken** bellektedir: K açık dosya  $\times$  i-node boyutu. FAT gibi disk boyutuyla değil, açık dosya sayısıyla orantılı.
- i-node **sabit boyutludur**; çok büyük dosyalar için son girdiler başka indis bloklarını gösterir (dolaylı bloklar).
- Bedel: küçük dosyada bile indis yeri harcanır; büyük dosyada dolaylı blok okumaları gerekir (önbellek bunu azaltır).

indis bloğu 19



## i-node Yapısı: Dolaylı Bloklar



ext2/ext3 ve klasik UNIX (V7: 10 doğrudan) yapısı. Küçük dosyalar yalnız doğrudan işaretçileri kullanır, dolaylı blok hiç ayrılmaz; dosya büyüdükçe ağaç derinleşir.

## En Büyük Dosya Boyutu

Varsayımlar: blok **4 KiB** = 4096 B, işaretçi **4 B** → bir dolaylı blok  $4096 / 4 = \mathbf{1024}$  adres tutar; i-node'da **12 doğrudan**, 1 tek, 1 çift, 1 üç dolaylı işaretçi.

Doğrudan:  $12 \times 4 \text{ KiB} = \mathbf{48 \text{ KiB}}$  (49.152 B) · Tek dolaylı:  $1024 \times 4 \text{ KiB} = \mathbf{4 \text{ MiB}}$   
İlk 12 blok i-node'dan doğrudan; sonraki 1024 blok tek bir indis bloğundan.

Adım 1 / 4

Tanenbaum örneği (1 KiB blok, 256 adres, 10 doğrudan): 16.843.018 blok  $\approx$  16 GiB. Gerçekte ext2/ext3'te 4 KiB blokla sınır **2 TiB**'dir (blok sayacı 32 bit  $\times$  512 B); ext4 extent'lerle 16 TiB.

## En Büyük Dosya Boyutu

Varsayımlar: blok **4 KiB** = 4096 B, işaretçi **4 B** → bir dolaylı blok  $4096 / 4 = \mathbf{1024}$  adres tutar; i-node'da **12 doğrudan**, 1 tek, 1 çift, 1 üç dolaylı işaretçi.

Doğrudan:  $12 \times 4 \text{ KiB} = \mathbf{48 \text{ KiB}}$  (49.152 B) · Tek dolaylı:  $1024 \times 4 \text{ KiB} = \mathbf{4 \text{ MiB}}$   
İlk 12 blok i-node'dan doğrudan; sonraki 1024 blok tek bir indis bloğundan.

Çift dolaylı:  $1024^2 \times 4 \text{ KiB} = 1.048.576 \text{ blok} = \mathbf{4 \text{ GiB}}$   
1024 indis bloğunun her biri 1024 veri bloğu gösterir.

Adım 2 / 4

Tanenbaum örneği (1 KiB blok, 256 adres, 10 doğrudan): 16.843.018 blok  $\approx 16 \text{ GiB}$ . Gerçekte ext2/ext3'te 4 KiB blokla sınır **2 TiB**'dir (blok sayacı 32 bit  $\times 512 \text{ B}$ ); ext4 extent'lerle 16 TiB.

## En Büyük Dosya Boyutu

Varsayımlar: blok **4 KiB** = 4096 B, işaretçi **4 B** → bir dolaylı blok  $4096 / 4 = \mathbf{1024}$  adres tutar; i-node'da **12 doğrudan**, 1 tek, 1 çift, 1 üç dolaylı işaretçi.

Doğrudan:  $12 \times 4 \text{ KiB} = \mathbf{48 \text{ KiB}}$  (49.152 B) · Tek dolaylı:  $1024 \times 4 \text{ KiB} = \mathbf{4 \text{ MiB}}$   
İlk 12 blok i-node'dan doğrudan; sonraki 1024 blok tek bir indis bloğundan.

Çift dolaylı:  $1024^2 \times 4 \text{ KiB} = 1.048.576 \text{ blok} = \mathbf{4 \text{ GiB}}$   
1024 indis bloğunun her biri 1024 veri bloğu gösterir.

Üç dolaylı:  $1024^3 \times 4 \text{ KiB} = 1.073.741.824 \text{ blok} = \mathbf{4 \text{ TiB}}$   
Bir düzey daha:  $\times 1024$ .

Adım 3 / 4

Tanenbaum örneği (1 KiB blok, 256 adres, 10 doğrudan): 16.843.018 blok  $\approx 16 \text{ GiB}$ . Gerçekte ext2/ext3'te 4 KiB blokla sınır **2 TiB**'dir (blok sayacı 32 bit  $\times 512 \text{ B}$ ); ext4 extent'lerle 16 TiB.

## En Büyük Dosya Boyutu

Varsayımlar: blok **4 KiB** = 4096 B, işaretçi **4 B** → bir dolaylı blok  $4096 / 4 = \mathbf{1024}$  adres tutar; i-node'da **12 doğrudan**, 1 tek, 1 çift, 1 üç dolaylı işaretçi.

Doğrudan:  $12 \times 4 \text{ KiB} = \mathbf{48 \text{ KiB}}$  (49.152 B) · Tek dolaylı:  $1024 \times 4 \text{ KiB} = \mathbf{4 \text{ MiB}}$   
İlk 12 blok i-node'dan doğrudan; sonraki 1024 blok tek bir indis bloğundan.

Çift dolaylı:  $1024^2 \times 4 \text{ KiB} = 1.048.576 \text{ blok} = \mathbf{4 \text{ GiB}}$   
1024 indis bloğunun her biri 1024 veri bloğu gösterir.

Üç dolaylı:  $1024^3 \times 4 \text{ KiB} = 1.073.741.824 \text{ blok} = \mathbf{4 \text{ TiB}}$   
Bir düzey daha:  $\times 1024$ .

Toplam:  $12 + 1024 + 1024^2 + 1024^3 = \mathbf{1.074.791.436 \text{ blok} \times 4096 = 4.402.345.721.856 \text{ B} \approx 4,004 \text{ TiB}}$

Neredeyse tamamı üç dolaylı bloktan gelir.

Adım 4 / 4

Tanenbaum örneği (1 KiB blok, 256 adres, 10 doğrudan): 16.843.018 blok  $\approx 16 \text{ GiB}$ . Gerçekte ext2/ext3'te 4 KiB blokla sınır **2 TiB**'dir (blok sayacı 32 bit  $\times 512 \text{ B}$ ); ext4 extent'lerle 16 TiB.

## Ofsetten Disk Bloğuna

ofset → mantıksal blok

40.000 B

blok 9 · doğrudan

1.000.000 B

blok 244 · tek dolaylı

5.000.000 B

blok 1.220 · çift dolaylı

6.000.000.000 B

blok 1.464.843 · üç dolaylı

i-node

doğrudan 0–11

tek dolaylı

çift dolaylı

üç dolaylı

veri

adres i-node'da

$\text{blok içi konum} = 40.000 \bmod 4096 = 3.136$

**disk okuması (i-node bellekteyse): 1**

$40.000 / 4096 = 9 \rightarrow 12\text{'den küçük: doğrudan işaretçi 9. Tek okuma.}$

# Ofsetten Disk Bloğuna

ofset → mantıksal blok

40.000 B

blok 9 · doğrudan

1.000.000 B

blok 244 · tek dolaylı

5.000.000 B

blok 1.220 · çift dolaylı

6.000.000.000 B

blok 1.464.843 · üç dolaylı

i-node

doğrudan 0–11

tek dolaylı

çift dolaylı

üç dolaylı

[232]

indis

veri

blok içi konum =  $1.000.000 \bmod 4096 = 576$

**disk okuması (i-node bellekteyse): 2**

$1.000.000 / 4096 = 244; 244 - 12 = \mathbf{232} < 1024 \rightarrow$  tek dolaylı bloğun 232. girdisi. 2 okuma.

Adım 2 / 4

## Ofsetten Disk Bloğuna

ofset → mantıksal blok

40.000 B

blok 9 · doğrudan

1.000.000 B

blok 244 · tek dolaylı

5.000.000 B

blok 1.220 · çift dolaylı

6.000.000.000 B

blok 1.464.843 · üç dolaylı

i-node

doğrudan 0–11

tek dolaylı

çift dolaylı

üç dolaylı

[0]

indis

[184]

indis

veri

blok içi konum =  $5.000.000 \bmod 4096 = 2.880$

**disk okuması (i-node bellekteyse): 3**

$5.000.000 / 4096 = 1220$ ;  $1220 - 12 - 1024 = 184 < 1024^2 \rightarrow$  çift dolaylı: dış indis 184 div 1024 = **0**, iç indis **184**. 3 okuma.

Adım 3 / 4

## Ofsetten Disk Bloğuna

ofset → mantıksal blok

40.000 B

blok 9 · doğrudan

1.000.000 B

blok 244 · tek dolaylı

5.000.000 B

blok 1.220 · çift dolaylı

6.000.000.000 B

blok 1.464.843 · üç dolaylı

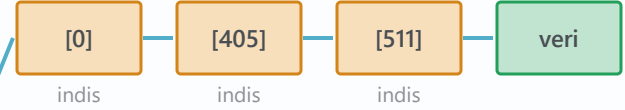
i-node

doğrudan 0–11

tek dolaylı

çift dolaylı

üç dolaylı



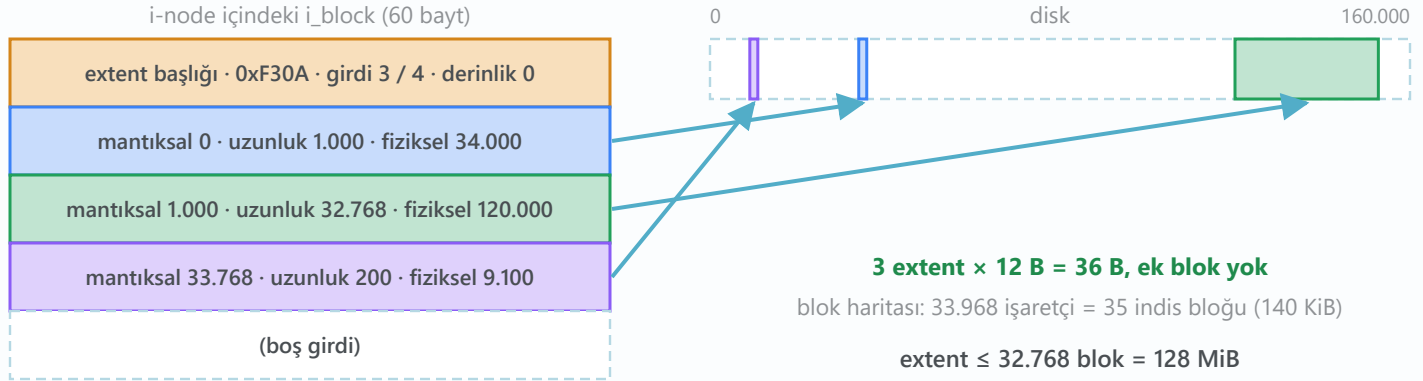
blok içi konum =  $6.000.000.000 \bmod 4096 = 3.072$

**disk okuması (i-node bellekteyse): 4**

$6 \text{ GB} / 4096 = 1.464.843$ ;  $12 + 1024 + 1024^2$  çıkarılınca  $415.231 \rightarrow$  üç dolaylı: **0, 405, 511**. 4 okuma; önbellek ilk düzeyleri tutar.

# TAHSİS YÖNTEMLERİ

## ext4: Extent'ler



- Bir **extent** = (mantıksal başlangıç, uzunluk, fiziksel başlangıç): bitişik blok dizisi tek kayıtla anlatılır. 48 bit fiziksel adres → 1 EiB birim; dosya en fazla **16 TiB**.
- i-node'a 4 extent sığar; daha fazlası gerekirse girdiler bir **extent ağacının** (derinlik > 0) indis düğümlerini gösterir.
- Gecikmeli tahsis (delayed allocation) ve çoklu blok ayırıcı, veriyi diske yazarken büyük bitişik alanlar seçer. XFS, Btrfs, NTFS (run list) ve APFS de extent kullanır.

## NTFS: Ana Dosya Tablosu (MFT)



- NTFS'te **her şey bir dosyadır**; MFT'nin kendisi de (kayıt 0). Her dosya en az bir 1 KiB kayıttır; içeriği *öznitelikler* listesidir.
- Büyük veri *run list* (extent) olarak tutulur; dizinler B+ ağacıdır ( `$INDEX_ROOT` , `$INDEX_ALLOCATION` ). `$LogFile` meta veri günlüğüdür.
- Ek özellikler: katı bağ, reparse point, alternatif veri akışı (ADS), sıkıştırma, EFS şifreleme, kotalar. Sunucularda ReFS (CoW, sağlama toplamı) da var.

## Dosya Sistemlerinin Karşılaştırması

| Dosya sistemi | En büyük dosya                   | En büyük birim         | Çökme tutarlılığı     | Tipik kullanım           |
|---------------|----------------------------------|------------------------|-----------------------|--------------------------|
| FAT32         | 4 GiB – 1 B                      | 2 TiB                  | yok (chkdsk/fsck)     | USB, UEFI ESP            |
| exFAT         | 16 EiB*                          | 128 PiB                | yok                   | SD kart, harici disk     |
| NTFS          | 16 EiB* (Windows: 256 TiB–8 PiB) | 256 TiB–8 PiB          | meta veri günlüğü     | Windows                  |
| ext4          | 16 TiB (4 KiB blok)              | 1 EiB                  | günlük (jbd2)         | Linux, Android           |
| XFS           | 8 EiB                            | 8 EiB                  | meta veri günlüğü     | RHEL, büyük sunucular    |
| Btrfs         | 16 EiB                           | 16 EiB                 | CoW + sağlama toplamı | Fedora, openSUSE, NAS    |
| ZFS           | 16 EiB                           | $2^{128}$ B (kuramsal) | CoW + sağlama toplamı | FreeBSD, TrueNAS, sunucu |
| APFS          | 8 EiB                            | 8 EiB                  | CoW, anlık görüntü    | macOS, iOS               |

\* kuramsal sınır; işletim sistemi ve küme boyutu daha düşük sınır koyar. Windows 10 (1709) sonrası NTFS 2 MiB kümeyle 8 PiB'a ulaşır. HFS+ yerini 2017'de APFS'e bıraktı; ext3 yerini ext4'e.

## Dizin Girdileri ve Uzun Adlar

(a) öznitelikler girdide (MS-DOS)

|       |                            |
|-------|----------------------------|
| games | öznitelikler + disk adresi |
| mail  | öznitelikler + disk adresi |
| news  | öznitelikler + disk adresi |
| work  | öznitelikler + disk adresi |

(b) öznitelikler i-node'da (UNIX)

|       |    |   |        |
|-------|----|---|--------|
| games | 14 | → | i-node |
| mail  | 7  | → | i-node |
| news  | 29 | → | i-node |
| work  | 51 | → | i-node |

(c) değişken uzunluk, satır içi

|                         |             |
|-------------------------|-------------|
| uzunluk<br>i-node - tür | proje-bütçe |
| uzunluk<br>i-node - tür | personel    |
| uzunluk<br>i-node - tür | foo         |

ext4 girdisi: i-node, rec\_len, ad uzunluğu, tür, ad

(d) sabit girdiler + ad öbeği

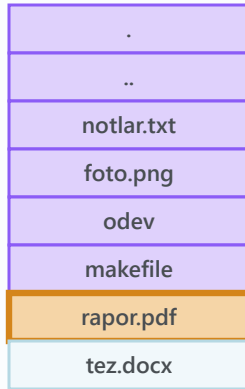
|           |               |   |             |
|-----------|---------------|---|-------------|
| öznitelik | ad işaretçisi | → | öbek (heap) |
| öznitelik | ad işaretçisi |   |             |
| öznitelik | ad işaretçisi |   |             |
|           |               |   | proje-bütçe |
|           |               |   | personel    |
|           |               |   | foo         |

girdiler hizalı; ad dizin bloğunun sonunda

Dizinin yerini bulmak için ilk bloğun adresi (bitişik), ilk bloğun numarası (bağlı liste/FAT) ya da i-node numarası (UNIX) yeterlidir. Uzun adlarda (a)/(b) gibi sabit girdi alan israf eder; (c) silinince delik bırakır ve ad sayfa sınırını aşabilir, (d) öbeği yönetmeyi gerektirir.

# Dizin Arama: Liste, Hash, Ağaç

doğrusal liste



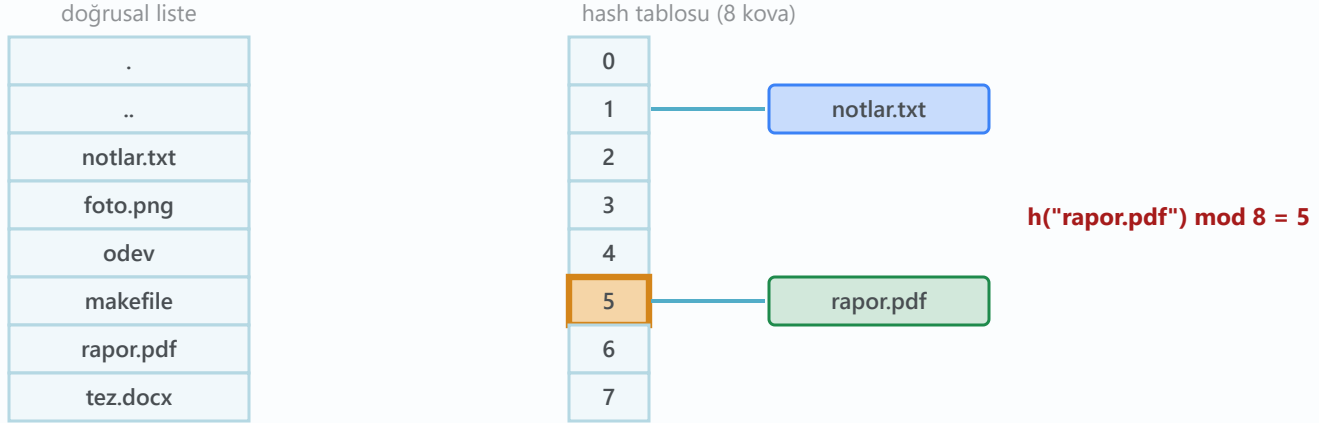
← 7 karşılaştırma

**Doğrusal liste:** basit, ama arama  $O(n)$ . Ekleme için önce adın olmadığı denetlenir; silmede girdi boş işaretlenir ya da son girdi taşınır.

Adım 1 / 3

ext4 büyük dizinlerde ad özetine göre sıralı *htree*, NTFS, XFS ve Btrfs **B+ ağacı** kullanır: sıralı listeleme ve  $O(\log n)$  arama birlikte. Linux en son kullanılan ad → i-node eşlemelerini ayrıca **dentry ön belleğinde** tutar.

# Dizin Arama: Liste, Hash, Ağaç

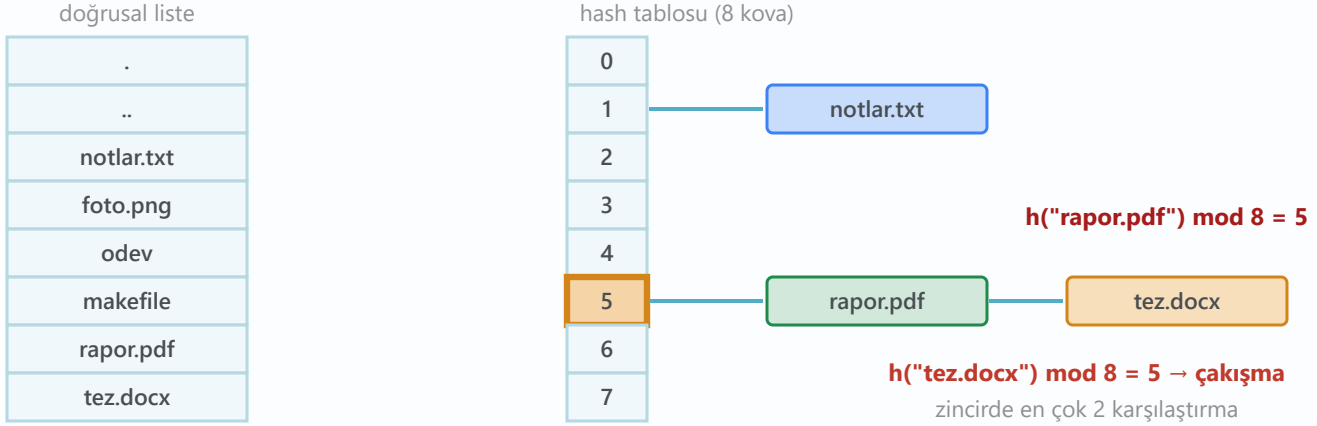


**Hash tablosu:** adın özeti kovayı seçer; arama ortalama  $O(1)$ .

Adım 2 / 3

ext4 büyük dizinlerde ad özetine göre sıralı *htree*, NTFS, XFS ve Btrfs **B+ ağacı** kullanır: sıralı listeleme ve  $O(\log n)$  arama birlikte. Linux en son kullanılan ad → i-node eşlemelerini ayrıca **dentry ön belleğinde** tutar.

# Dizin Arama: Liste, Hash, Ağaç

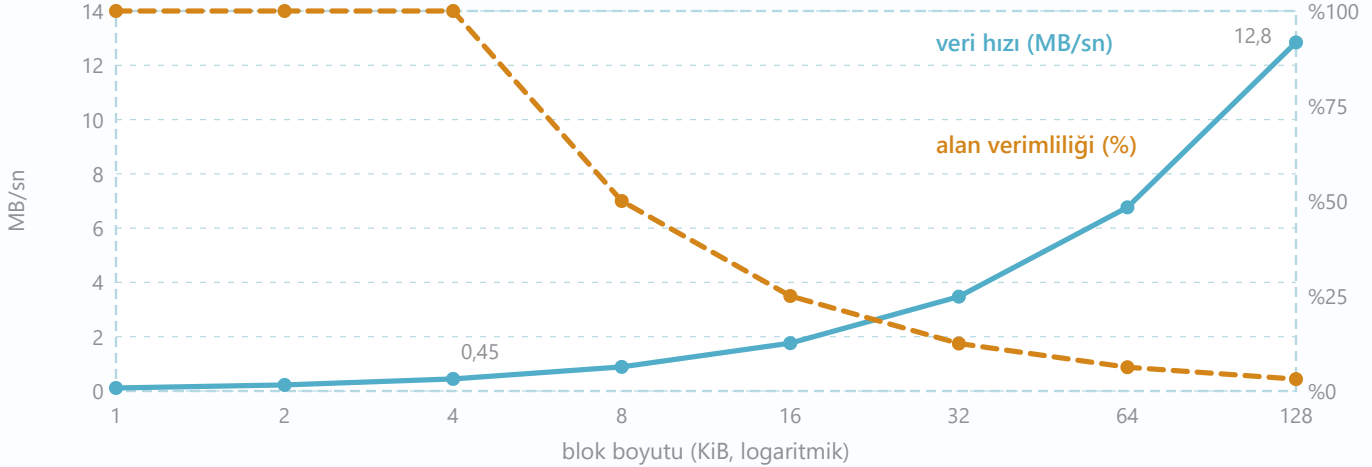


Çakışmalar zincirle çözülür. Sabit kova sayısı büyüyen dizinde sorun olur; bu yüzden modern sistemler **ağaç** kullanır.

Adım 3 / 3

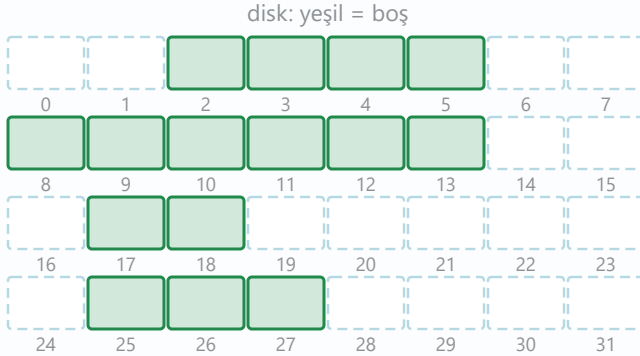
ext4 büyük dizinlerde ad özetine göre sıralı *htree*, NTFS, XFS ve Btrfs **B+ ağacı** kullanır: sıralı listeleme ve  $O(\log n)$  arama birlikte. Linux en son kullanılan ad  $\rightarrow$  i-node eşlemelerini ayrıca **dentry ön belleğinde** tutar.

# Blok Boyutu Seçimi



- Model (Tanenbaum): tüm dosyalar 4 KiB; ortalama arama 5 ms, dönüş 8,33 ms, iz 1 MB. Süre =  $5 + 4,165 + (b / 1 \text{ MB}) \times 8,33 \text{ ms}$ . Hesap Python ile.
- **Büyük blok** → **daha iyi veri hızı, daha kötü alan kullanımı**; küçük blok tersi.
- Ölçümler dosyaların çoğunun küçük, alanın çoğunu ise az sayıda büyük dosyanın kapladığını gösterir. Bugün varsayılan **4 KiB** (sayfa boyutu); büyük dosyalar extent ve küme seçenekleriyle desteklenir.

## Boş Alan Yönetimi: Dört Yöntem



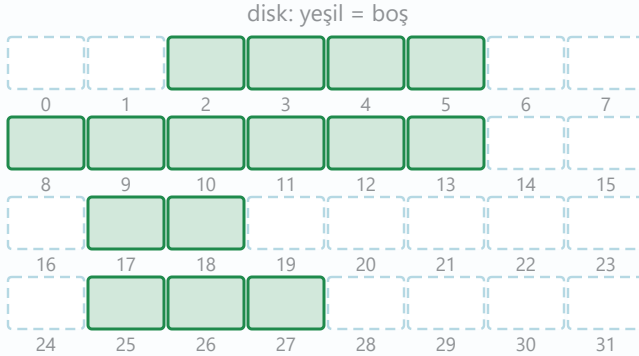
boş bloklar:

**2-5, 8-13, 17-18, 25-27**

15 boş blok / 32

Örnek disk: 32 blok, 15'i boş (Silberschatz örneği).

# Boş Alan Yönetimi: Dört Yöntem



biteşlem (1 = boş)

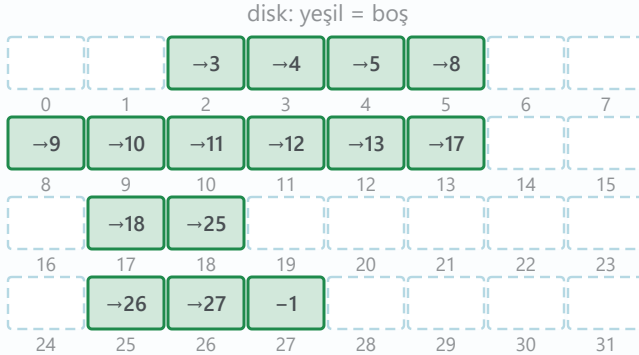
|            |   |   |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|---|---|
| blok 0-7   | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| blok 8-15  | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| blok 16-23 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| blok 24-31 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

32 blok → 32 bit = 4 bayt

ilk boş: ilk sıfır olmayan sözcükte ilk 1 biti

**Biteşlem:** blok başına 1 bit. Boyutu sabit ve küçük; bitişik boş blok aramak kolay. Bellekte tutulması gerekir.

# Boş Alan Yönetimi: Dört Yöntem



baş işaretçisi → 2

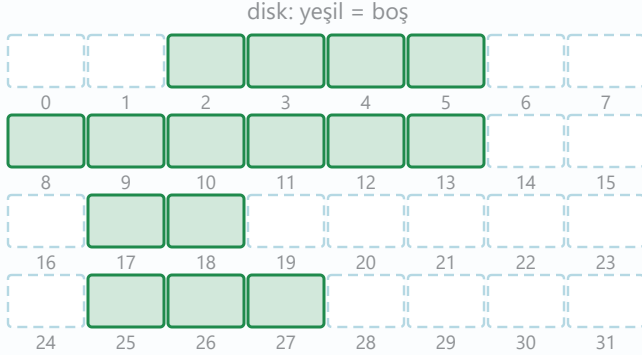
her boş blok bir sonrakini gösterir

ek alan gerekmez (bilgi boş blokların içinde)

**ama listeyi dolaşmak = her blok için G/Ç**

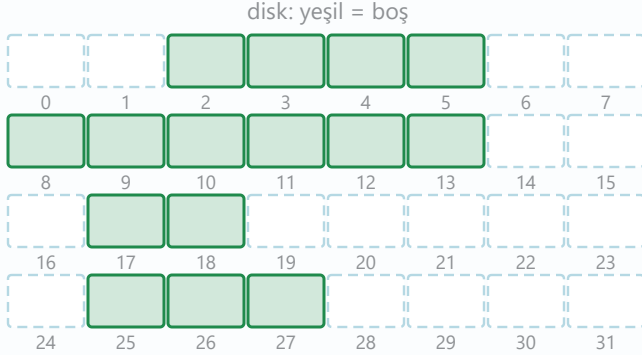
**Bağlı liste:** boş bloklar birbirine bağlanır. Tek boş blok almak ucuz, çok sayıda bitişik blok bulmak pahalı.

# Boş Alan Yönetimi: Dört Yöntem



**Gruplama:** ilk boş blok sonraki  $n-1$  boş bloğun adresini ve bir sonraki grup bloğunu tutar (UNIX'in boş liste blokları).

# Boş Alan Yönetimi: Dört Yöntem



sayma: (ilk blok, ardışık sayı)

|         |            |
|---------|------------|
| (2, 4)  | blok 2–5   |
| (8, 6)  | blok 8–13  |
| (17, 2) | blok 17–18 |
| (25, 3) | blok 25–27 |

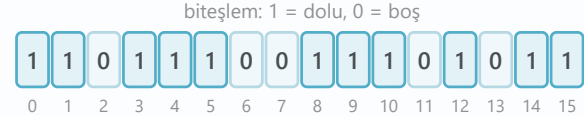
15 boş blok → 4 çift

boş alan bitişik öbeklerse çok kısa

**Sayma:** (başlangıç, uzunluk) çiftleri; extent fikrinin boş alana uygulanmış. XFS bunu iki B+ ağacında tutar.

# Biteşleme Blok Ayırma: Kod İzleme

```
► int blok_ayir(int ipucu) {  
  2   for (int k = 0; k < N; k++) {  
  3       int i = (ipucu + k) % N;  
  4       if (bitesl[i] == 0) { /* boş mu? */  
  5           bitesl[i] = 1;  
  6           return i;  
  7       }  
  8   }  
  9   return -1; /* disk dolu */  
10 }
```



dosyanın blokları: —

yeşil: bu dosyaya ayrılan

Dosya blok 3'te bitiyor ve 3 blok daha istiyor. Yerellik için arama son bloğun hemen arkasından başlar:

`blok_ayir(4)`.

Adım 1 / 9

## DEĞİŞKENLER

`ipucu = 4`

`k = -`

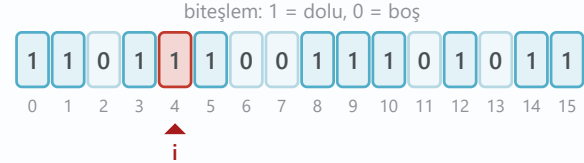
`i = -`

## ÇIKTI



# Biteşleme Blok Ayırma: Kod İzleme

```
1 int blok_ayir(int ipucu) {  
2   for (int k = 0; k < N; k++) {  
3     ▶ int i = (ipucu + k) % N;  
4     ▶ if (bitesl[i] == 0) { /* boş mu? */  
5       bitesl[i] = 1;  
6       return i;  
7     }  
8   }  
9   return -1; /* disk dolu */  
10 }
```



dosyanın blokları: —

yeşil: bu dosyaya ayrılan

k = 0: i = (4 + 0) mod 16 = 4; bit 1, blok dolu.  
Sonraki konuma geçilir.

Adım 2 / 9

## DEĞİŞKENLER

ipucu = 4

k = 0

i = 4

## ÇIKTI



## Biteşleme Blok Ayırma: Kod İzleme

```
1 int blok_ayir(int ipucu) {  
2   for (int k = 0; k < N; k++) {  
3     ▶ int i = (ipucu + k) % N;  
4     ▶ if (bitesl[i] == 0) { /* boş mu? */  
5       bitesl[i] = 1;  
6       return i;  
7     }  
8   }  
9   return -1; /* disk dolu */  
10 }
```



dosyanın blokları: —

yeşil: bu dosyaya ayrılan

k = 1:  $i = (4 + 1) \bmod 16 = 5$ ; bit 1, blok dolu.  
Sonraki konuma geçilir.

Adım 3 / 9

### DEĞİŞKENLER

ipucu = 4

k = 1

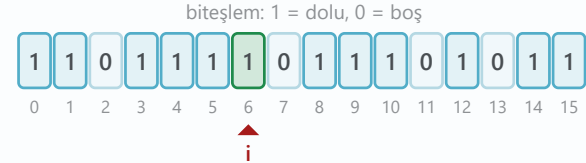
i = 5

### ÇIKTI



## Biteşleme Blok Ayırma: Kod İzleme

```
1 int blok_ayir(int ipucu) {  
2   for (int k = 0; k < N; k++) {  
3     ▶ int i = (ipucu + k) % N;  
4     ▶ if (bitesl[i] == 0) { /* boş mu? */  
5       ▶ bitesl[i] = 1;  
6       ▶ return i;  
7   }  
8 }  
9 return -1; /* disk dolu */  
10 }
```



dosyanın blokları: 6

yeşil: bu dosyaya ayrılan

bitesl[6] = 0: **boş**. Bit 1 yapılıır, blok 6 döner.

Adım 4 / 9

### DEĞİŞKENLER

ipucu = 4

k = 2

i = 6

### ÇIKTI

6

# Biteşleme Blok Ayırma: Kod İzleme

```
1 int blok_ayir(int ipucu) {  
2   for (int k = 0; k < N; k++) {  
3     ▶ int i = (ipucu + k) % N;  
4     ▶ if (bitesl[i] == 0) { /* boş mu? */  
5       ▶ bitesl[i] = 1;  
6       ▶ return i;  
7   }  
8 }  
9 return -1; /* disk dolu */  
10 }
```



dosyanın blokları: 6, 7

yeşil: bu dosyaya ayrılan

bitesl[7] = 0: **boş**. Bit 1 yapılıır, blok 7 döner. İpucu tuttu: ardışık bloklar sıralı okumada kafa hareketini azaltır.

Adım 5 / 9

## DEĞİŞKENLER

ipucu = 7

k = 0

i = 7

## ÇIKTI

6 7

## Biteşleme Blok Ayırma: Kod İzleme

```
1 int blok_ayir(int ipucu) {  
2   for (int k = 0; k < N; k++) {  
3     ▶ int i = (ipucu + k) % N;  
4     ▶ if (bitesl[i] == 0) { /* boş mu? */  
5       bitesl[i] = 1;  
6       return i;  
7     }  
8   }  
9   return -1; /* disk dolu */  
10 }
```



dosyanın blokları: 6, 7

yeşil: bu dosyaya ayrılan

`blok_ayir(8)` : k = 0, i = 8; bit 1, dolu.

Adım 6 / 9

### DEĞİŞKENLER

ipucu = 8

k = 0

i = 8

### ÇIKTI

6 7

## Biteşleme Blok Ayırma: Kod İzleme

```
1 int blok_ayir(int ipucu) {  
2   for (int k = 0; k < N; k++) {  
3     ▶ int i = (ipucu + k) % N;  
4     ▶ if (bitesl[i] == 0) { /* boş mu? */  
5       bitesl[i] = 1;  
6       return i;  
7     }  
8   }  
9   return -1; /* disk dolu */  
10 }
```



**dosyanın blokları: 6, 7**

yeşil: bu dosyaya ayrılan

k = 1: i = (8 + 1) mod 16 = 9; bit 1, blok dolu.  
Sonraki konuma geçilir.

Adım 7 / 9

### DEĞİŞKENLER

ipucu = 8

k = 1

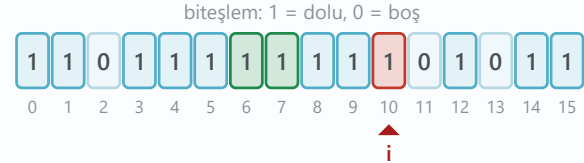
i = 9

### ÇIKTI

6 7

# Biteşleme Blok Ayırma: Kod İzleme

```
1 int blok_ayir(int ipucu) {  
2   for (int k = 0; k < N; k++) {  
3     ▶ int i = (ipucu + k) % N;  
4     ▶ if (bitesl[i] == 0) { /* boş mu? */  
5       bitesl[i] = 1;  
6       return i;  
7     }  
8   }  
9   return -1; /* disk dolu */  
10 }
```



**dosyanın blokları: 6, 7**

yeşil: bu dosyaya ayrılan

$k = 2: i = (8 + 2) \bmod 16 = 10$ ; bit 1, blok dolu.  
Sonraki konuma geçilir.

Adım 8 / 9

## DEĞİŞKENLER

ipucu = 8

k = 2

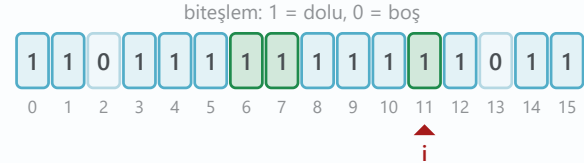
i = 10

## ÇIKTI

6 7

# Biteşleme Blok Ayırma: Kod İzleme

```
1 int blok_ayir(int ipucu) {  
2   for (int k = 0; k < N; k++) {  
3     ▶ int i = (ipucu + k) % N;  
4     ▶ if (bitesl[i] == 0) { /* boş mu? */  
5       ▶ bitesl[i] = 1;  
6       ▶ return i;  
7   }  
8 }  
9 return -1; /* disk dolu */  
10 }
```



dosyanın blokları: 6, 7, 11

yeşil: bu dosyaya ayrılan

bitesl[11] = 0: **boş**. Bit 1 yapılır, blok 11 döner. Dosya 6, 7, 11 bloklarını aldı: bitişik değil, ama birbirine yakın.

Adım 9 / 9

## DEĞİŞKENLER

ipucu = 8

k = 3

i = 11

## ÇIKTI

6 7 11

## Boş Alan: Sayısal Karşılaştırma

|                              | Tanenbaum örneği                                                       | Modern örnek                                                               |
|------------------------------|------------------------------------------------------------------------|----------------------------------------------------------------------------|
| Disk / blok                  | 500 GB, 1 KiB blok                                                     | 1 TiB, 4 KiB blok                                                          |
| Blok sayısı                  | $500 \times 10^9 / 1024 \approx \mathbf{488,3 \text{ milyon}}$         | $2^{40} / 2^{12} = \mathbf{268.435.456}$                                   |
| Boş liste bloğu (32 bit no.) | 1 KiB / 4 B = 256 → 255 adres + 1 bağ                                  | 4096 / 4 = 1024 → 1023 adres + 1 bağ                                       |
| Liste, disk tamamen boşken   | 488,3 M / 255 $\approx \mathbf{1,91 \text{ milyon}}$ blok              | 268,4 M / 1023 $\approx \mathbf{262.401}$ blok ( $\approx 1 \text{ GiB}$ ) |
| Biteşlem                     | 488,3 M bit / 8192 $\approx \mathbf{59.605}$ blok ( $\approx 60.000$ ) | $2^{28} \text{ bit} = 32 \text{ MiB} = \mathbf{8192}$ blok                 |

- Biteşlem sabit boyutludur; liste yalnızca disk *neredeyse doluyken* daha küçüktür (çünkü yalnız boş blokları listeler).
- Bellekte tek bir işaretçi bloğu tutmak yeterlidir: dolunca diske yazılır, boşalınca yenisi okunur. Tanenbaum'un iyileştirmesi: bellekte yarı dolu blok tutarak geçici dosyaların sürekli disk G/Ç'sine yol açmasını önlemek.
- Gerçek sistemler: ext4 blok grubu başına biteşlem; XFS konuma ve boyuta göre iki B+ ağacı (sayma); ZFS ayırma/serbest bırakma günlüğü (space map); Btrfs extent ağacı.

Tanenbaum (500 GB, 1 KiB)

biteşlem  **59.605**

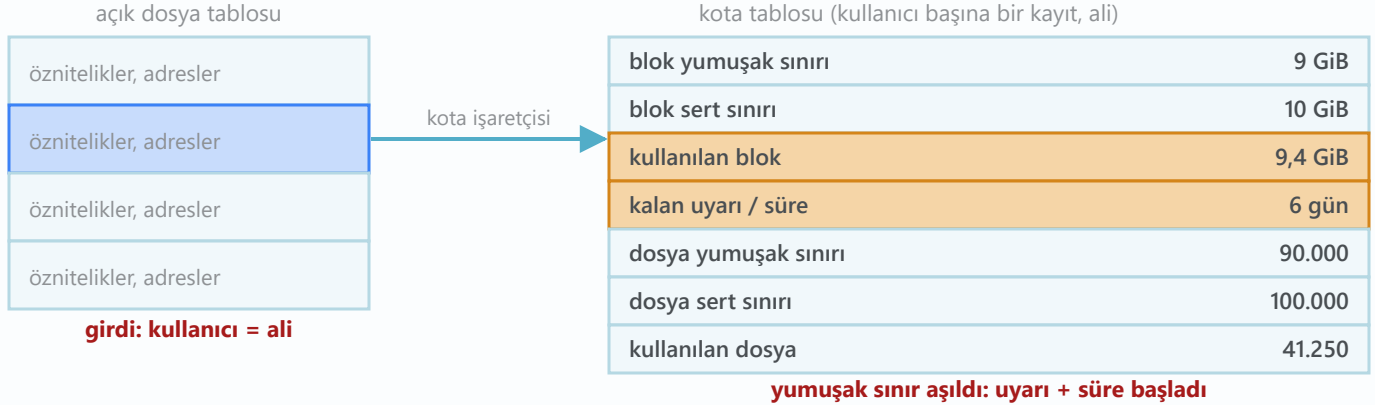
boş liste  **1.910.000**

Modern (1 TiB, 4 KiB)

biteşlem  **8.192**

boş liste  **262.401**  
blok sayısı, disk tamamen boşken

## Disk Kotaları



- Her açık dosya girdisi sahibinin kota kaydını gösterir; her yeni blok ya da i-node tahsisinde sayaçlar artar ve sınırlarla karşılaştırılır.
- Yumuşak sınır** aşılabılır ama uyarı verilir ve bir süre (ör. 7 gün) başlar; süre biterse ya da **sert sınıra** ulaşırsa yazma `EDQUOT` ile reddedilir.
- Bugün: Linux `quota` araçları (ext4, XFS proje kotaları), NTFS birim kotaları, ZFS veri kümesi ve Btrfs alt birim (qgroup) kotaları.

# Bir Dosyayı Oluşturmak Kaç G/Ç'dir?

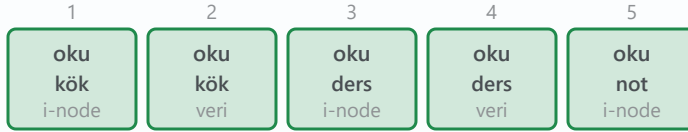
`open("/ders/not.txt", O_RDONLY): dosya var`



okuma



yazma



**5 G/Ç = 5 okuma + 0 yazma**

Önbellek boş, kökün i-node yeri süperbloktan biliniyor. Yol bileşen bileşen çözülür: her dizin için i-node + veri bloğu, sonda dosyanın i-node'u. **5 okuma**; yol uzadıkça artar.

Adım 1 / 5

Varsayım: i-node ve biteşlem kullanan basit bir dosya sistemi, dizinler tek blok, önbellek yok. Gerçek sistemlerde dentry ve i-node önbelleği okumaların çoğunu kaldırır; günlük ve gecikmeli yazma da yazmaların sayısını ve sırasını değiştirir.

# Bir Dosyayı Oluşturmak Kaç G/Ç'dir?

read(fd, buf, 4096): ilk blok

 okuma  yazma



**3 G/Ç = 2 okuma + 1 yazma**

Her `read` i-node'dan blok adresini bulur, bloğu okur ve **erişim zamanını** yazar.  
`noatime` / `relatime` bağlama seçenekleri bu yazmayı çoğu kez kaldırır.

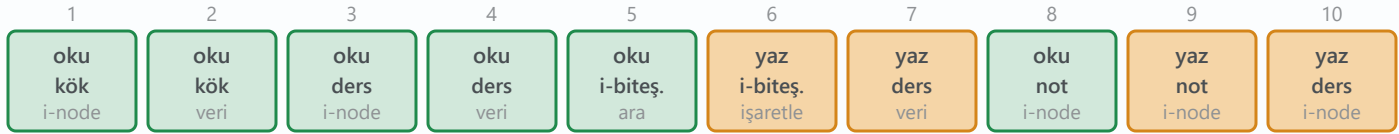
Adım 2 / 5

Varsayım: i-node ve biteşlem kullanan basit bir dosya sistemi, izinler tek blok, önbellek yok. Gerçek sistemlerde dentry ve i-node önbelleği okumaların çoğunu kaldırır; günlük ve gecikmeli yazma da yazmaların sayısını ve sırasını değiştirir.

## Bir Dosyayı Oluşturmak Kaç G/Ç'dir?

creat("/ders/not.txt"): dosya yok

■ okuma ■ yazma



**10 G/Ç = 6 okuma + 4 yazma**

Oluşturmak daha pahalı: boş i-node biteşlemden seçilir, yeni i-node ilklendirilir, dizine girdi eklenir, dizinin i-node'u (boyut, zaman) güncellenir. **10 G/Ç.**

Adım 3 / 5

Varsayım: i-node ve biteşlem kullanan basit bir dosya sistemi, dizinler tek blok, önbellek yok. Gerçek sistemlerde dentry ve i-node önbelleği okumaların çoğunu kaldırır; günlük ve gecikmeli yazma da yazmaların sayısını ve sırasını değiştirir.

## Bir Dosyayı Oluşturmak Kaç G/Ç'dir?

write(fd, buf, 4096): yeni blok gerekir

 okuma  yazma



**5 G/Ç = 2 okuma + 3 yazma**

Yeni blok isteyen her `write` : veri biteşleminden blok ayır (oku + yaz), veriyi yaz, i-node'a adresi ve boyutu yaz. **5 G/Ç**, yalnız 1'i asıl veri.

Adım 4 / 5

Varsayım: i-node ve biteşlem kullanan basit bir dosya sistemi, izinler tek blok, önbellek yok. Gerçek sistemlerde dentry ve i-node önbelleği okumaların çoğunu kaldırır; günlük ve gecikmeli yazma da yazmaların sayısını ve sırasını değiştirir.

## Bir Dosyayı Oluşturmak Kaç G/Ç'dir?

creat + 3 × write(4096):  $10 + 3 \times 5 = 25$  G/Ç

  okuma   yazma



kalin çerçeveli 3 kutu kullanıcı verisi; kalan 22 G/Ç üst veri (biteşlem, i-node, dizin)

**önbellek: okumalar bellekten gelir · gecikmeli yazma: aynı blok bir kez yazılır**

→ 8 yazma kalır: i-biteşlem, v-biteşlem, ders veri, ders i-node, not i-node, 3 veri bloğu

Önbelleksiz bir dosyayı oluşturup 12 KiB yazmak **25 G/Ç** eder. Tampon önbelleği ve gecikmeli yazma bu yüzden vazgeçilmezdir; yazma sırası da çökme tutarlılığını belirler.

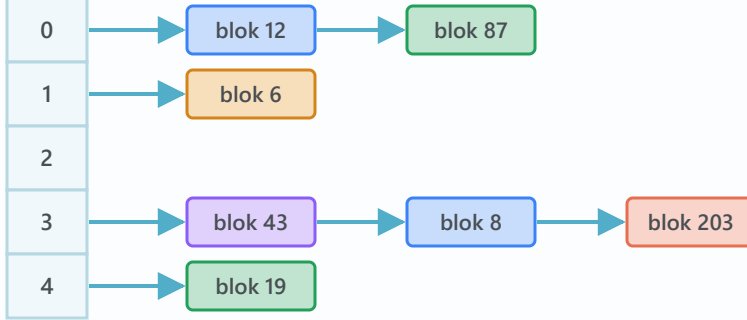
Adım 5 / 5

Varsayım: i-node ve biteşlem kullanan basit bir dosya sistemi, dizinler tek blok, önbellek yok. Gerçek sistemlerde dentry ve i-node önbelleği okumaların çoğunu kaldırır; günlük ve gecikmeli yazma da yazmaların sayısını ve sırasını değiştirir.

## PERFORMANS

# Tampon Önbelleği (Buffer Cache)

hash tablosu  
(aygıt, blok no.)



arama: hash ile  $O(1)$

değiştirme: LRU kuyruğunun başı

önemli bloklar (i-node, dizin)  
kirliyse hemen yazılır

**LRU listesi (en eski → en yeni)**

6 → 43 → 12 → 19 → 8 → 87 → 203

| Erişim                        | Gecikme (2026, tipik)            |
|-------------------------------|----------------------------------|
| DRAM'den 64 bayt              | ≈ 80–120 ns                      |
| NVMe SSD 4 KiB rastgele okuma | ≈ 50–100 μs                      |
| HDD arama + dönüş             | ≈ 5–10 ms (100–280 MB/sn sıralı) |

- Yazma politikası:** *write-through* her yazmayı hemen diske gönderir (eski MS-DOS, çıkarılabilir ortam); *write-back* kirli blokları biriktirir. Linux 30 sn'den eski kirli sayfaları arka planda yazar; uygulama `fsync` ile zorlar.

# Tampon Önbelleği: Kod İzleme

```
► Tampon *bread(int blok) {  
2   Tampon *t = hash_ara(blok);  
3   if (t != NULL) {           /* isabet */  
4       lru_sona(t);  
5       return t;  
6   }  
7   t = lru_bas();              /* en eski */  
8   if (t->kirli)  
9       diske_yaz(t);          /* geri yaz */  
10  t->blok = blok; t->kirli = 0;  
11  diskten_oku(t);            /* ıska: G/Ç */  
12  lru_sona(t);  
13  return t;  
14 }
```

LRU listesi: solda en eski, sağda en yeni



Önbellekte 3 tampon var: bloklar 5, 9, 2 (9 kirli: değiştirildi ama diske yazılmadı). İstekler sırayla: 9, 7, 2, 4.

Adım 1 / 10

## DEĞİŞKENLER

blok = -

t = -

okuma = 0

yazma = 0

## ÇIKTI



# Tampon Önbelleği: Kod İzleme

```
1 Tampon *bread(int blok) {  
    ▶ Tampon *t = hash_ara(blok);  
    ▶ if (t != NULL) {          /* isabet */  
6      lru_sona(t);  
7      return t;  
8    }  
9    t = lru_bas();             /* en eski */  
10   if (t->kirli)  
11     diske_yaz(t);            /* geri yaz */  
12   t->blok = blok; t->kirli = 0;  
13   diskten_oku(t);            /* ıska: G/Ç */  
14   lru_sona(t);  
15   return t;  
16 }
```

LRU listesi: solda en eski, sağda en yeni



**bread(9)** : hash tablosunda bulundu, **isabet**. Disk G/Ç'si yok.

Adım 2 / 10

## DEĞİŞKENLER

blok = 9   t = blok 9   okuma = 0  
yazma = 0

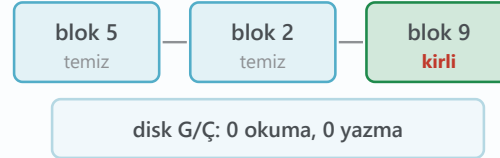
## ÇIKTI



# Tampon Önbelleği: Kod İzleme

```
1 Tampon *bread(int blok) {  
2   Tampon *t = hash_ara(blok);  
3   if (t != NULL) {           /* isabet */  
4       ▶   lru_sona(t);  
5       ▶   return t;  
6   }  
7   t = lru_bas();              /* en eski */  
8   if (t->kirli)  
9       diske_yaz(t);          /* geri yaz */  
10  t->blok = blok; t->kirli = 0;  
11  diskten_oku(t);            /* ıska: G/Ç */  
12  lru_sona(t);  
13  return t;  
14 }
```

LRU listesi: solda en eski, sağda en yeni



Blok 9 listenin sonuna (en yeni) taşınır ve döndürülür. Artık en eski blok 5.

Adım 3 / 10

## DEĞİŞKENLER

blok = 9

t = blok 9

okuma = 0

yazma = 0

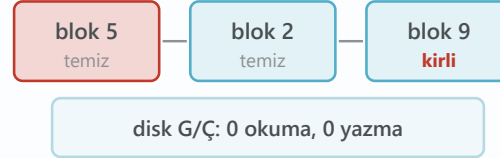
## ÇIKTI

9: isabet

# Tampon Önbelleği: Kod İzleme

```
1 Tampon *bread(int blok) {  
  ► Tampon *t = hash_ara(blok);  
3  if (t != NULL) {          /* isabet */  
4    lru_sona(t);  
5    return t;  
6  }  
  ► t = lru_bas();           /* en eski */  
8  if (t->kirli)  
9    diske_yaz(t);           /* geri yaz */  
10 t->blok = blok; t->kirli = 0;  
11 disken_oku(t);            /* ıska: G/Ç */  
12 lru_sona(t);  
13 return t;  
14 }
```

LRU listesi: solda en eski, sağda en yeni



**bread(7)** : önbellekte yok, **ıska**. Yer açmak için LRU'nun başındaki tampon (blok 5) seçilir.

Adım 4 / 10

## DEĞİŞKENLER

blok = 7

t = blok 5

okuma = 0

yazma = 0

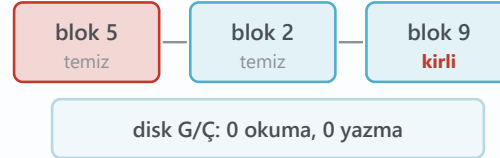
## ÇIKTI

9: isabet

# Tampon Önbelleği: Kod İzleme

```
1 Tampon *bread(int blok) {
2   Tampon *t = hash_ara(blok);
3   if (t != NULL) {           /* isabet */
4       lru_sona(t);
5       return t;
6   }
7   t = lru_bas();              /* en eski */
8   if (t->kirli)
9       diske_yaz(t);           /* geri yaz */
10  t->blok = blok; t->kirli = 0;
11  diskten_oku(t);             /* ıska: G/Ç */
12  lru_sona(t);
13  return t;
14 }
```

LRU listesi: solda en eski, sağda en yeni



Blok 5 temiz: diskteki kopyası güncel, geri yazmaya gerek yok.

Adım 5 / 10

## DEĞİŞKENLER

blok = 7

t = blok 5

okuma = 0

yazma = 0

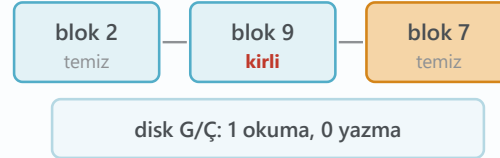
## ÇIKTI

9: isabet

# Tampon Önbelleği: Kod İzleme

```
1 Tampon *bread(int blok) {
2   Tampon *t = hash_ara(blok);
3   if (t != NULL) {           /* isabet */
4       lru_sona(t);
5       return t;
6   }
7   t = lru_bas();              /* en eski */
8   if (t->kirli)
9       diske_yaz(t);          /* geri yaz */
10  ▶ t->blok = blok; t->kirli = 0;
11  ▶ disken_oku(t);            /* 1ska: G/Ç */
12  ▶ lru_sona(t);
13  return t;
14 }
```

LRU listesi: solda en eski, sağda en yeni



Tampon blok 7 için yeniden kullanılır, içerik diskten okunur (1 okuma) ve listenin sonuna eklenir.

Adım 6 / 10

## DEĞİŞKENLER

blok = 7

t = blok 7

okuma = 1

yazma = 0

## ÇIKTI

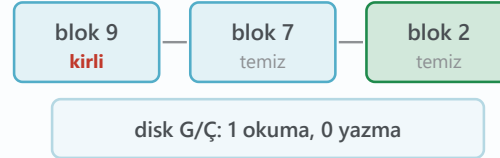
9: isabet

7: 1ska

# Tampon Ön Belleği: Kod İzleme

```
1 Tampon *bread(int blok) {  
    ▶ Tampon *t = hash_ara(blok);  
    ▶ if (t != NULL) {           /* isabet */  
    ▶     lru_sona(t);  
    ▶     return t;  
6 }  
7 t = lru_bas();                /* en eski */  
8 if (t->kirli)  
9     diske_yaz(t);             /* geri yaz */  
10 t->blok = blok; t->kirli = 0;  
11 diskten_oku(t);              /* ıska: G/Ç */  
12 lru_sona(t);  
13 return t;  
14 }
```

LRU listesi: solda en eski, sağda en yeni



**bread(2)** : isabet; blok 2 en yeniye taşınır. Kirli blok 9 şimdi listenin başında.

Adım 7 / 10

## DEĞİŞKENLER

blok = 2    t = blok 2    okuma = 1  
yazma = 0

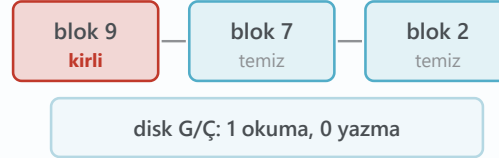
## ÇIKTI

```
9: isabet  
7: ıska  
2: isabet
```

# Tampon Ön Belleği: Kod İzleme

```
1 Tampon *bread(int blok) {  
    ▶ Tampon *t = hash_ara(blok);  
3   if (t != NULL) {           /* isabet */  
4       lru_sona(t);  
5       return t;  
6   }  
    ▶ t = lru_bas();           /* en eski */  
8   if (t->kirli)  
9       diske_yaz(t);          /* geri yaz */  
10  t->blok = blok; t->kirli = 0;  
11  diskten_oku(t);            /* ıska: G/Ç */  
12  lru_sona(t);  
13  return t;  
14 }
```

LRU listesi: solda en eski, sağda en yeni



**bread(4)** : ıska. En eski tampon blok 9'u tutuyor.

Adım 8 / 10

## DEĞİŞKENLER

**blok = 4**

**t = blok 9**

**okuma = 1**

**yazma = 0**

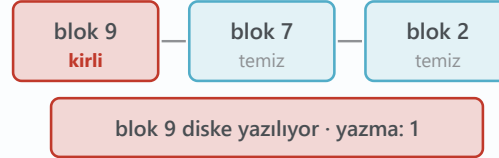
## ÇIKTI

```
9: isabet  
7: ıska  
2: isabet
```

# Tampon Önbelleği: Kod İzleme

```
1 Tampon *bread(int blok) {
2   Tampon *t = hash_ara(blok);
3   if (t != NULL) {           /* isabet */
4     lru_sona(t);
5     return t;
6   }
7   t = lru_bas();              /* en eski */
8   if (t->kirli)
9     diske_yaz(t);             /* geri yaz */
10  t->blok = blok; t->kirli = 0;
11  diskten_oku(t);              /* ıska: G/Ç */
12  lru_sona(t);
13  return t;
14 }
```

LRU listesi: solda en eski, sağda en yeni



Blok 9 **kirli**: değişiklik kaybolmasın diye tampon yeniden kullanılmadan önce diske yazılır (1 yazma). Bu, ıskanın maliyetini ikiye katlar.

Adım 9 / 10

## DEĞİŞKENLER

blok = 4

t = blok 9

okuma = 1

yazma = 1

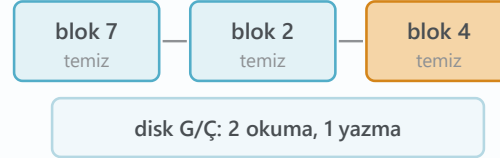
## ÇIKTI

```
9: isabet
7: ıska
2: isabet
```

# Tampon Önbelleği: Kod İzleme

```
1 Tampon *bread(int blok) {  
2   Tampon *t = hash_ara(blok);  
3   if (t != NULL) {           /* isabet */  
4     lru_sona(t);  
5     return t;  
6   }  
7   t = lru_bas();              /* en eski */  
8   if (t->kirli)  
9     diske_yaz(t);            /* geri yaz */  
▶   t->blok = blok; t->kirli = 0;  
▶   diskten_oku(t);            /* ıska: G/Ç */  
▶   lru_sona(t);  
▶   return t;  
14 }
```

LRU listesi: solda en eski, sağda en yeni



Blok 4 okunur. 4 istekte 2 isabet; toplam 2 okuma ve 1 yazma. Önbellek olmasaydı 4 okuma ve blok 9 için ayrı bir yazma gerekirdi.

Adım 10 / 10

## DEĞİŞKENLER

blok = 4

t = blok 4

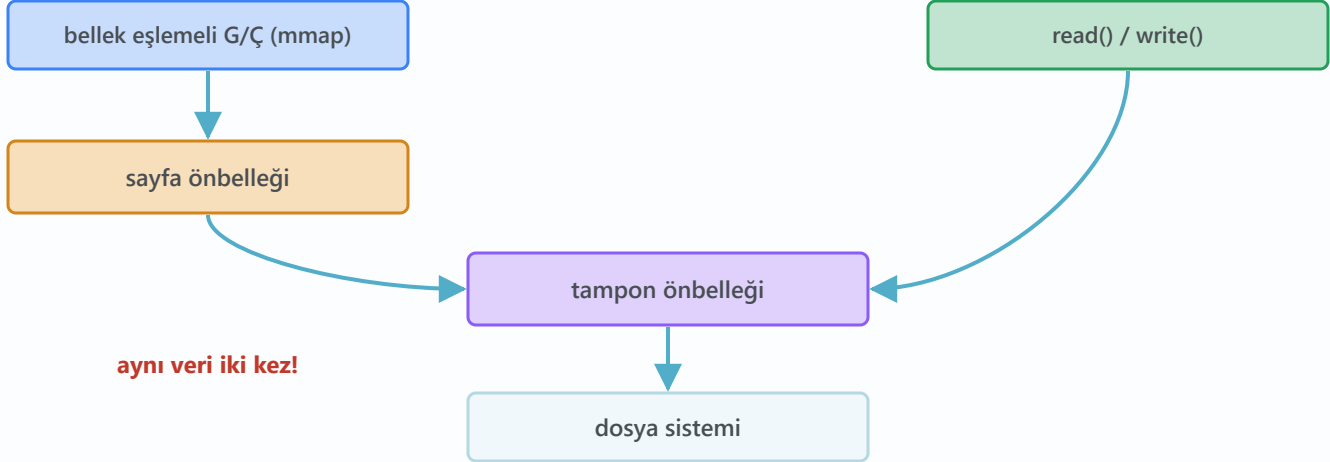
okuma = 2

yazma = 1

## ÇIKTI

```
9: isabet  
7: ıska  
2: isabet  
4: ıska (9 geri yazıldı)
```

# Sayfa Önbelleği ve Birleşik Tampon Önbelleği



**Birleşik değil:** `mmap` verisi sayfa önbelleğinde, `read/write` verisi tampon önbelleğinde; ikisi arasında kopyalanır (**çift önbellekleme**), bellek ve tutarlılık sorunu.

Adım 1 / 2

Sayfa önbelleği sanal bellekle aynı sayfa değiştirme algoritmalarını paylaşır. İstisna: ZFS kendi ARC önbelleğini tutar. `O_DIRECT` veritabanlarının önbelleği atlamasını sağlar.

# Sayfa Önbelleği ve Birleşik Tampon Önbelleği

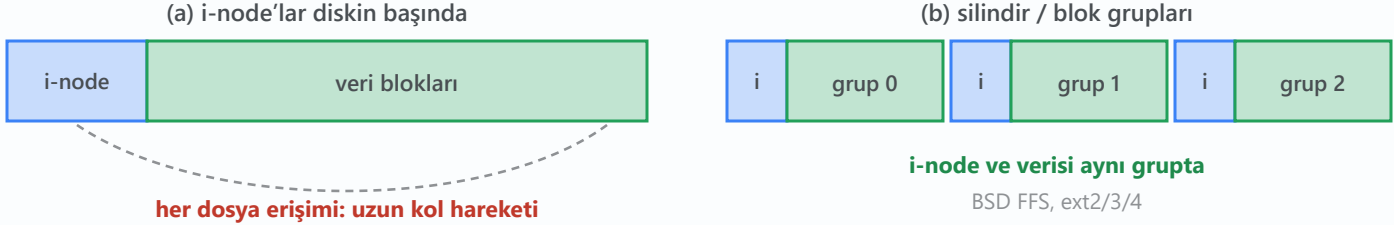


**Birleşik tampon önbelleği:** dosya sayfaları tek yerde; iki yol da aynı sayfaları görür. Linux 2.4'ten beri sayfa önbelleği, Windows önbellek yöneticisi, Solaris böyledir.

Adım 2 / 2

Sayfa önbelleği sanal bellekle aynı sayfa değiştirme algoritmalarını paylaşır. İstisna: ZFS kendi ARC önbelleğini tutar. `O_DIRECT` veritabanlarının önbelleği atlamasını sağlar.

# Önden Okuma, Yerleşim ve Birleştirme



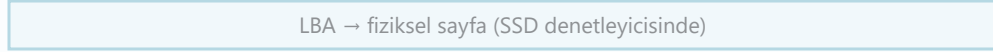
- **Önden okuma:** k. blok istenince k+1 de (önbellekte yoksa) okunur. Yalnız sıralı erişimde işe yarar; dosya başına bir "sıralı mı?" izi tutulur. Linux pencereyi uyarlar (varsayılan 128 KiB), uygulama `posix_fadvise` ile ipucu verir.
- **Kol hareketini azaltmak:** ardışık okunacak blokları yakına koy; biteşlemlerle bitişik blok bulmak kolaydır. Bloklar ikiye ayrılır (eski öneri) ya da bugünkü gibi extent, gecikmeli tahsis ve çoklu blok ayırıcı kullanılır.
- **Birleştirme (defrag):** zamanla dosyalar parçalanır. Windows "Sürücülerini birleştir ve iyileştir"; Linux'ta da vardır: `e4defrag`, `xfs_fsr`, `btrfs filesystem defragment`. ext4'ün tahsis politikası ihtiyacı azaltır.
- SSD'de arama süresi yoktur; birleştirme gereksiz yazma (aşınma) demektir; Windows SSD'de birleştirme yerine TRIM gönderir.

## SSD Dikkat Noktaları: FTL ve TRIM

dosya sistemi (LBA)



FTL eşleme tablosu



flaş silme bloğu X



silme bloğu Y



A dosyası LBA 0–3'e yazılır; FTL bunları X bloğundaki sayfalara eşler.

Adım 1 / 4

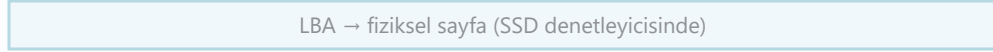
Linux: haftalık `fstrim.timer` ya da `discard` bağlama seçeneği; Windows "İyileştir" TRIM gönderir. Ayrıca: bölümleri 1 MiB sınırına hizala, SSD'yi birleştirme; flaş için tasarlanan F2FS (Android) günlük yapısıdır.

## SSD Dikkat Noktaları: FTL ve TRIM

dosya sistemi (LBA)



FTL eşleme tablosu



flaş silme bloğu X



silme bloğu Y



**yerinde yazılamaz: yeni sayfaya yaz, eskisini geçersiz işaretle**

Flaş sayfası yerinde güncellenemez (önce tüm blok silinmeli): A1 yeni sayfaya yazılır, eski sayfa geçersiz. SSD'nin içi aslında **günlük yapılıdır**.

Adım 2 / 4

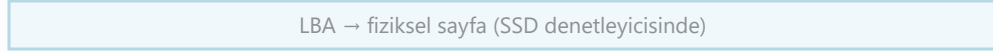
Linux: haftalık `fstrim.timer` ya da `discard` bağlama seçeneği; Windows "İyileştir" TRIM gönderir. Ayrıca: bölümleri 1 MiB sınırına hizala, SSD'yi birleştirme; flaş için tasarlanan F2FS (Android) günlük yapılıdır.

## SSD Dikkat Noktaları: FTL ve TRIM

dosya sistemi (LBA)



FTL eşleme tablosu



flaş silme bloğu X



silme bloğu Y



**SSD bunları hâlâ geçerli sanıyor → çöp toplamada boşuna kopyalanır**

**rm A** : dosya sistemi bloklarını kendi biteşleminde boş işaretler, ama SSD bunu bilmez.

Adım 3 / 4

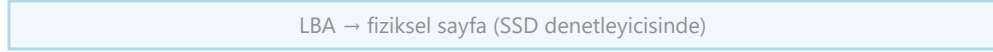
Linux: haftalık `fstrim.timer` ya da `discard` bağlama seçeneği; Windows "İyileştir" TRIM gönderir. Ayrıca: bölümleri 1 MiB sınırına hizala, SSD'yi birleştirme; flaş için tasarlanan F2FS (Android) günlük yapısıdır.

## SSD Dikkat Noktaları: FTL ve TRIM

dosya sistemi (LBA)



FTL eşleme tablosu



flaş silme bloğu X



silme bloğu Y



**TRIM: LBA 0–3 kullanılmıyor → X bloğu kopyasız silinebilir**

**TRIM** (ATA) / **Deallocate** (NVMe): dosya sistemi boşalan LBA'ları bildirir; çöp toplama daha az kopyalar, yazma büyütmesi ve aşınma azalır.

Adım 4 / 4

Linux: haftalık `fstrim.timer` ya da `discard` bağlama seçeneği; Windows “İyileştir” TRIM gönderir. Ayrıca: bölümleri 1 MiB sınırına hizala, SSD’yi birleştirme; flaş için tasarlanan F2FS (Android) günlük yapılıdır.

## Yedekleme Stratejileri

|                             | Fiziksel döküm                                        | Mantıksal döküm                          |
|-----------------------------|-------------------------------------------------------|------------------------------------------|
| Ne kopyalanır?              | blok 0'dan sona tüm (kullanılan) bloklar              | seçilen dizinler ve dosyalar, yollarıyla |
| Hız, basitlik               | çok hızlı, basit                                      | daha yavaş, karmaşık                     |
| Dizin atlama, artımlı yedek | yapamaz                                               | yapabilir (değişenler)                   |
| Tek dosyayı geri yükleme    | zor                                                   | kolay; başka makinede de                 |
| Bozuk bloklar               | denetleyici yeniden eşlemeli ya da İS listeyi bilmeli | sorun değil                              |

tam döküm  
+ artımlılar



gün 1



gün 2



gün 3



gün 4



gün 5



gün 6

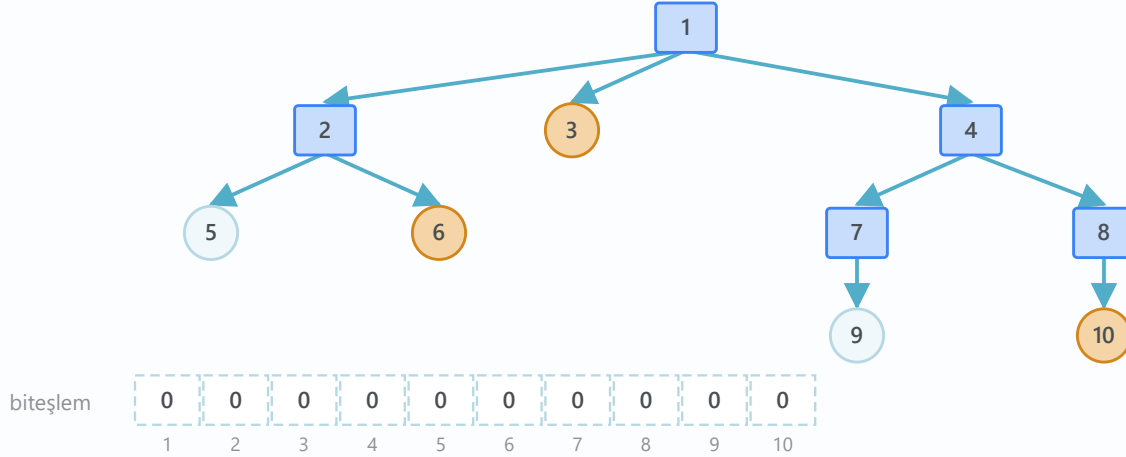


gün 7

geri yükleme: gün 1 tam,  
sonra 2 ... 7 sırayla

- İki amaç: **felaketten kurtarma** (disk, yangın, fidye yazılımı) ve **dikkatsizlikten kurtarma** (yanlışlıkla silinen dosya).
- Tam + artımlı**: ör. ayda bir tam döküm, her gün yalnız son dökümden beri değişenler. Geri yükleme: boş dosya sistemine son tam döküm (önce dizinler, sonra dosyalar), ardından artımlılar sırayla.
- Geçici dosyalar ve özel dosyalar ( `/dev` , `/proc` , `/tmp` ) yedeklenmez. Sistem kullanılırken tutarlı yedek için **anlık görüntü** (LVM, ZFS/Btrfs, Windows VSS) alınır.
- 3-2-1 kuralı**: 3 kopya, 2 farklı ortam, 1 kopya başka yerde (çevrimdışı ya da değiştirilemez). Teyp bugün de arşivde: LTO-9 kartuş 18 TB yerel.

## Mantıksal Döküm Algoritması

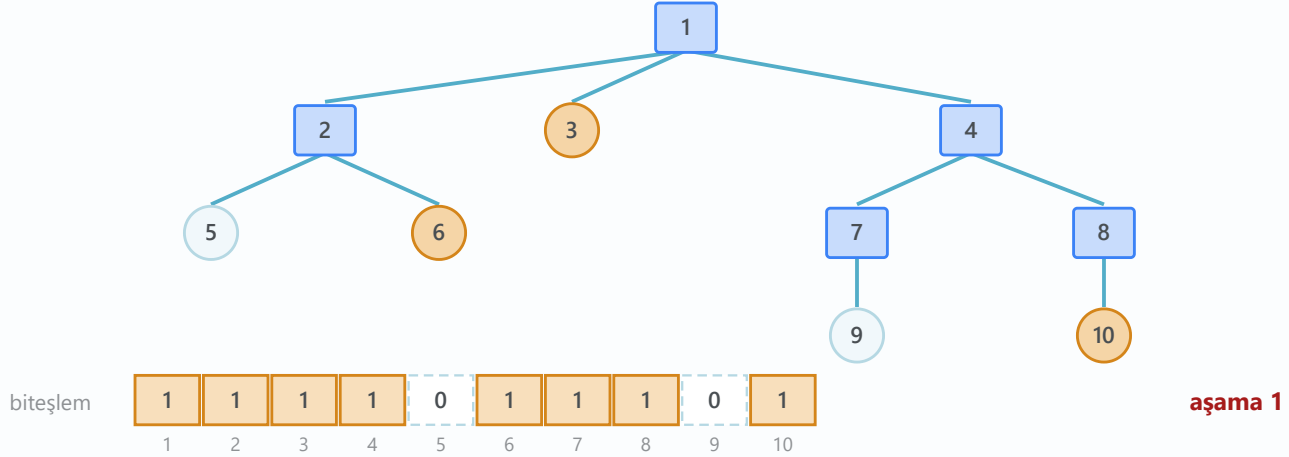


Kareler dizin, daireler dosya; sayılar i-node numarası. Turuncu: son dökümden beri **değişenler** (3, 6, 10).

Adım 1 / 5

Ek ayrıntılar: boş liste dökülmez, geri yüklemede yeniden kurulur; bir dosyanın birden çok katı bağı yalnız bir kez dökülmeli; seyrek dosyaların delikleri ve FIFO/aygıt dosyaları özel işlenir.

## Mantıksal Döküm Algoritması

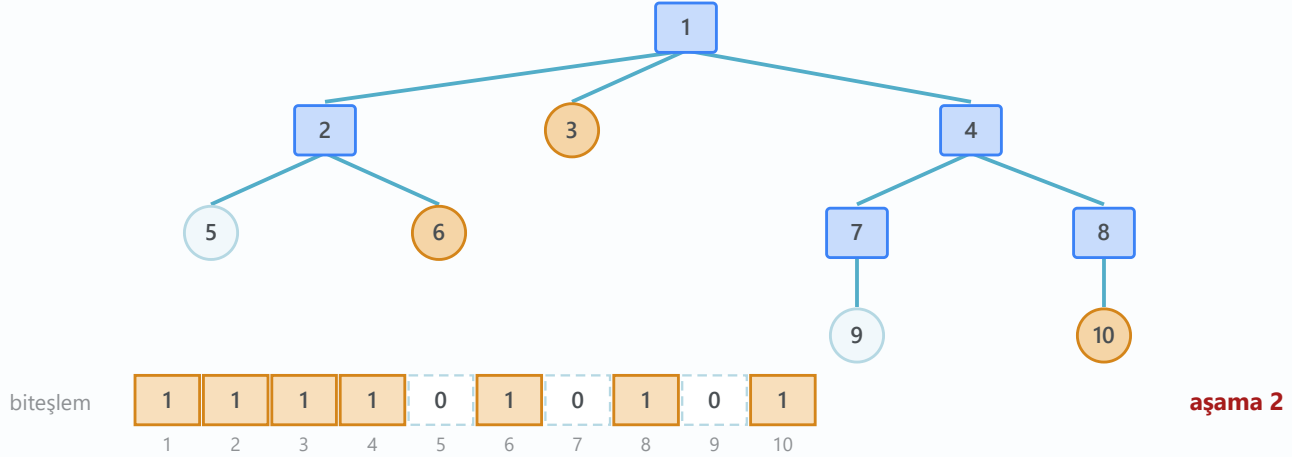


**Aşama 1:** kökten başlayıp ağacı dolaş; değişen her dosyayı ve **tüm dizinleri** işaretle.

Adım 2 / 5

Ek ayrıntılar: boş liste dökülmez, geri yüklemede yeniden kurulur; bir dosyanın birden çok katı bağı yalnız bir kez dökülmeli; seyrek dosyaların delikleri ve FIFO/aygıt dosyaları özel işlenir.

## Mantıksal Döküm Algoritması

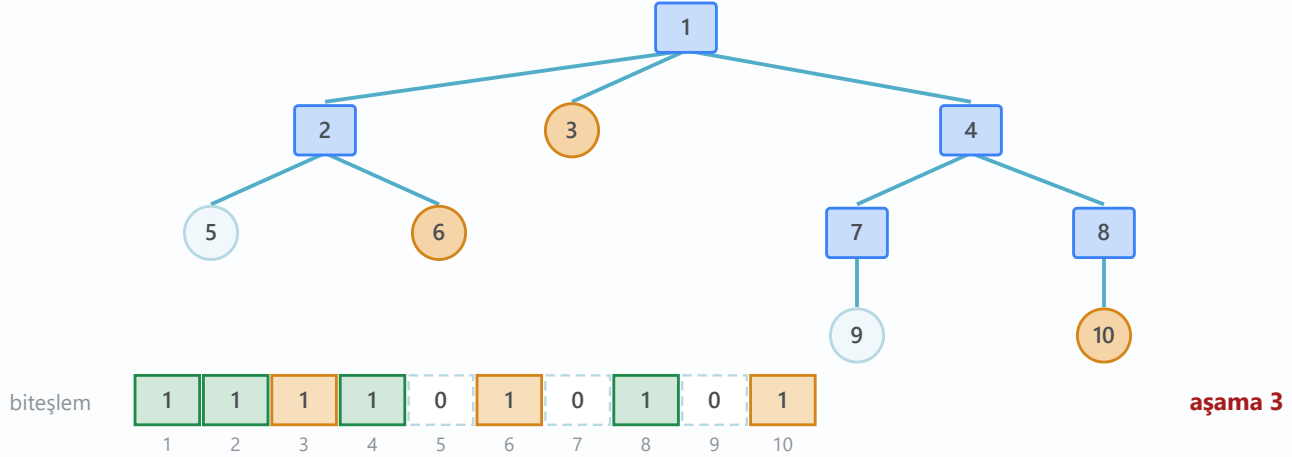


**Aşama 2:** altında değişen hiçbir şey olmayan dizinlerin işaretini kaldır: **7** (yalnız değişmemiş 9'u içeriyor).

Adım 3 / 5

Ek ayrıntılar: boş liste dökülmez, geri yüklemede yeniden kurulur; bir dosyanın birden çok katı bağı yalnız bir kez dökülmeli; seyrek dosyaların delikleri ve FIFO/aygıt dosyaları özel işlenir.

## Mantıksal Döküm Algoritması

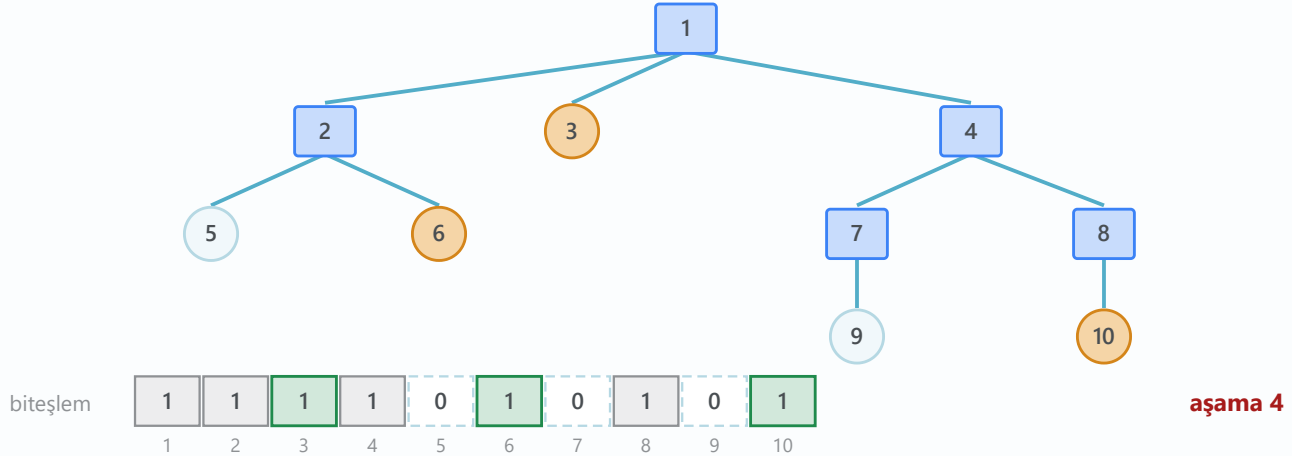


**Aşama 3:** i-node sırasıyla işaretli **dizinleri** dök (1, 2, 4, 8) — değişmemiş olsalar da, yolları geri kurabilmek için.

Adım 4 / 5

Ek ayrıntılar: boş liste dökülmez, geri yüklemede yeniden kurulur; bir dosyanın birden çok katı bağı yalnız bir kez dökülmeli; seyrek dosyaların delikleri ve FIFO/aygıt dosyaları özel işlenir.

## Mantıksal Döküm Algoritması



**Aşama 4:** işaretli **dosyaları** dök (3, 6, 10). Geri yüklemede önce dizinler, sonra dosyalar yazılır.

Adım 5 / 5

Ek ayrıntılar: boş liste dökülmez, geri yüklemede yeniden kurulur; bir dosyanın birden çok katı bağı yalnız bir kez dökülmeli; seyrek dosyaların delikleri ve FIFO/aygıt dosyaları özel işlenir.

## Tutarlılık Denetimi: fsck

| blok numarası | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| kullanımda    | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0  | 1  | 1  | 1  | 0  | 0  |
| boş listede   | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1  | 0  | 0  | 0  | 1  | 1  |

Her blok bir sayaçta 1, diğerinde 0: tutarlı.

**(a) Tutarlı:** fsck tüm i-node'ları ve boş listeyi/biteşlemi okur, blok başına iki sayaç tutar.

Adım 1 / 4

- **Dosya tutarlılığı:** kökten tüm dizinler dolaşılır, her i-node için görülme sayısı tutulur ve i-node'daki bağ sayısı ile karşılaştırılır. Bağ sayısı büyükse yalnız alan israfı; *küçükse* tehlikeli (dosya erken silinir) → düzeltilir. Sahipsiz i-node'lar **lost+found** 'a konur.
- Windows'ta karşılığı **chkdsk** (Windows 9x'teki adı *scandisk*). Tam tarama terabaytlık diskte saatler sürer; günlükleme ve CoW bu yüzden doğdu.

# Tutarlılık Denetimi: fsck

| blok numarası | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| kullanımda    | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0  | 1  | 1  | 1  | 0  | 0  |
| boş listede   | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1  | 0  | 0  | 0  | 1  | 1  |

**Blok 2 hiçbir yerde yok: eksik blok → boş listeye eklenir.**

**(b) Eksik blok:** zararsız ama alan kaybı; boş listeye konur.

Adım 2 / 4

- **Dosya tutarlılığı:** kökten tüm dizinler dolaşılır, her i-node için görülme sayısı tutulur ve i-node'daki bağ sayısı ile karşılaştırılır. Bağ sayısı büyükse yalnız alan israfı; *küçükse* tehlikeli (dosya erken silinir) → düzeltilir. Sahipsiz i-node'lar **lost+found** 'a konur.
- Windows'ta karşılığı **chkdsk** (Windows 9x'teki adı *scandisk*). Tam tarama terabaytlık diskte saatler sürer; günlükleme ve CoW bu yüzden doğru.

## Tutarlılık Denetimi: fsck

| blok numarası | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| kullanımda    | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0  | 1  | 1  | 1  | 0  | 0  |
| boş listede   | 0 | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 1  | 0  | 0  | 0  | 1  | 1  |

**Blok 4 boş listede iki kez → boş liste yeniden kurulur.**

**(c) Boş listede yinelenen blok:** aynı blok iki kez verilebilirdi; liste yeniden oluşturulur.  
(Biteşlemde bu durum oluşamaz.)

Adım 3 / 4

- **Dosya tutarlılığı:** kökten tüm dizinler dolaşılır, her i-node için görülme sayısı tutulur ve i-node'daki bağ sayısı ile karşılaştırılır. Bağ sayısı büyükse yalnız alan israfı; *küçükse* tehlikeli (dosya erken silinir) → düzeltilir. Sahipsiz i-node'lar **lost+found** 'a konur.
- Windows'ta karşılığı **chkdsk** (Windows 9x'teki adı *scandisk*). Tam tarama terabaytlık diskte saatler sürer; günlükleme ve CoW bu yüzden doğdu.

# Tutarlılık Denetimi: fsck

|               |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|---------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| blok numarası | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| kullanımda    | 1 | 1 | 0 | 1 | 0 | 2 | 1 | 1 | 1 | 0 | 0  | 1  | 1  | 1  | 0  | 0  |
| boş listede   | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1  | 0  | 0  | 0  | 1  | 1  |

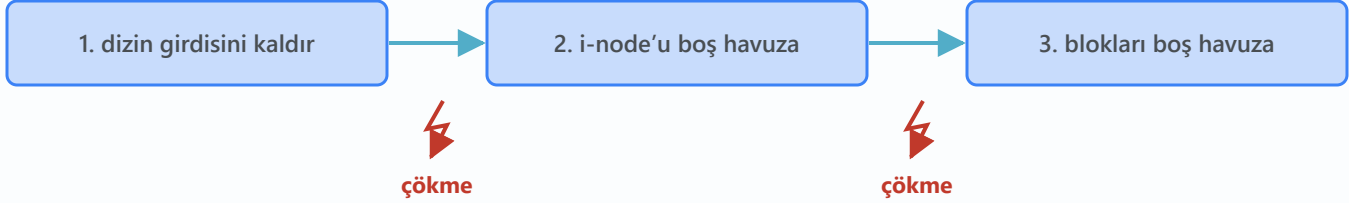
**Blok 5 iki dosyada → yeni blok ayır, kopyala, kullanıcıya bildir.**

**(d) Yinelenen veri bloğu:** en kötüsü; blok kopyalanıp iki dosyaya ayrı verilir, kullanıcı bilgilendirilir.

Adım 4 / 4

- **Dosya tutarlılığı:** kökten tüm dizinler dolaşılır, her i-node için görülme sayısı tutulur ve i-node'daki bağ sayısı ile karşılaştırılır. Bağ sayısı büyükse yalnız alan israfı; *küçükse* tehlikeli (dosya erken silinir) → düzeltilir. Sahipsiz i-node'lar **lost+found** 'a konur.
- Windows'ta karşılığı **chkdsk** (Windows 9x'teki adı *scandisk*). Tam tarama terabaytlık diskte saatler sürer; günlükleme ve CoW bu yüzden doğdu.

# Çökme ve Atomiklik: Neden Günlük?



1'den sonra çökme: i-node ve bloklar sahipsiz (sızıntı, fsck bulur)

2'den sonra çökme: bloklar sızar

**sıra ters olsaydı: dizin, başka dosyaya verilmiş bir i-node'u gösterebilirdi → veri karışır!**

- Bir dosya işlemi birden çok bloğu değiştirir; diskler yalnızca tek sektörlük yazmayı atomik yapar. Arada çökme olursa yapı **tutarsız** kalır.
- Çözüm ailesi: (1) sonra onar: **fsck**; (2) önce niyeti yaz: **günlükleme**; (3) hiç üzerine yazma: **CoW** ve **LFS**; (4) yazma sırasını ayarla: BSD *soft updates*.
- **Eşgüçlülük (idempotence)**: günlükteki işlemler kaç kez yeniden oynatılırsa oynatılsın aynı sonucu vermeli. "Blok n'yi boş işaretle" eşgüçlüdür; "boş listenin sonuna n'yi ekle" değildir (iki kez eklenebilir).

# Günlükleme: Önce Günlüğe Yaz

günlük (journal)



son yerler



**1. Veri:** dosyaya eklenen blok Db önce son yerine yazılır (*ordered* mod, ext4 varsayılanı).

Adım 1 / 6

Modlar: *journal* (veri de günlüğe, en güvenli, en yavaş), *ordered*, *writeback* (veri sırası yok).  
Günlük kullananlar: ext3/ext4 (jbd2), XFS, NTFS ( **\$LogFile** ), JFS. Apple'da HFS+ yerini CoW tabanlı APFS'e bıraktı.

# Günlükleme: Önce Günlüğe Yaz

günlük (journal)



son yerler



**2. Günlüğe yaz:** işlem başı TxB ve değişecek meta veri blokları (i-node I, biteşlem B) günlüğe **sıralı** yazılır.

Adım 2 / 6

Modlar: *journal* (veri de günlüğe, en güvenli, en yavaş), *ordered*, *writeback* (veri sırası yok).  
Günlük kullananlar: ext3/ext4 (jbd2), XFS, NTFS ( **\$LogFile** ), JFS. Apple'da HFS+ yerini CoW tabanlı APFS'e bıraktı.

## Günlükleme: Önce Günlüğe Yaz

günlük (journal)



son yerler



**3. İşlemi onayla:** önceki yazmalar diske ulaşınca (bariyer/FLUSH) TxE yazılır. TxE diskteyse işlem **kesinleşmiştir**.

Adım 3 / 6

Modlar: *journal* (veri de günlüğe, en güvenli, en yavaş), *ordered*, *writeback* (veri sırası yok).  
Günlük kullananlar: ext3/ext4 (jbd2), XFS, NTFS ( `$LogFile` ), JFS. Apple'da HFS+ yerini CoW tabanlı APFS'e bıraktı.

## Günlükleme: Önce Günlüğe Yaz

günlük (journal)



son yerler



**kontrol noktası (checkpoint)**

**4. Kontrol noktası:** I ve B son yerlerine yazılır; bu yazmalar arada kesilse de günlükten tekrarlanabilir.

Adım 4 / 6

Modlar: *journal* (veri de günlüğe, en güvenli, en yavaş), *ordered*, *writeback* (veri sırası yok).  
Günlük kullananlar: ext3/ext4 (jbd2), XFS, NTFS ( **\$LogFile** ), JFS. Apple'da HFS+ yerini CoW tabanlı APFS'e bıraktı.

## Günlükleme: Önce Günlüğe Yaz

günlük (journal)



son yerler



**5. Serbest bırak:** günlük süperbloğu işlemi tamamlandı işaretler; günlük dairesel bir tampon gibi yeniden kullanılır.

Adım 5 / 6

Modlar: *journal* (veri de günlüğe, en güvenli, en yavaş), *ordered*, *writeback* (veri sırası yok).  
Günlük kullananlar: ext3/ext4 (jbd2), XFS, NTFS ( `$LogFile` ), JFS. Apple'da HFS+ yerini CoW tabanlı APFS'e bıraktı.

# Günlükleme: Önce Günlüğe Yaz

günlük (journal)



son yerler



çökme TxE'den önce → işlem yok sayılır; sonra → günlük yeniden oynatılır

**Kurtarma:** açılışta günlük taranır; TxE'si olan işlemler baştan oynatılır (redo, eşgüçlü), olmayanlar atılır. fsck gerekmez, saniyeler sürer.

Adım 6 / 6

Modlar: *journal* (veri de günlüğe, en güvenli, en yavaş), *ordered*, *writeback* (veri sırası yok).  
Günlük kullananlar: ext3/ext4 (jbd2), XFS, NTFS ( **\$LogFile** ), JFS. Apple'da HFS+ yerini CoW tabanlı APFS'e bıraktı.

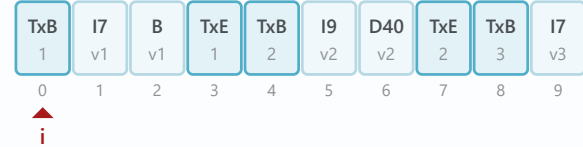
# Günlükten Kurtarma: Kod İzleme

```

▶ void kurtar(Kayit g[], int n) {
▶   int i = 0;
3   while (i < n && g[i].tur == TXB) {
4     int j = i + 1;
5     while (j < n && g[j].tur == BLOK)
6       j++;
7     if (j == n || g[j].tur != TXE)
8       break;          /* onaysız: at */
9     for (int k = i + 1; k < j; k++)
10      yaz(g[k].hedef, g[k].veri);
11    i = j + 1;          /* sonraki */
12  }
13 }

```

Günlük (çökmeden sonra diskte):



Son yerler:



Sistem işlem 3 yazılırken çöktü. Günlükte üç işlem başlığı var; son yerlerdeki bloklar hâlâ eski (v0).

Açılıştaki **kurtar** çalışır.

Adım 1 / 12

## DEĞİŞKENLER

i = 0

j = -

k = -

# Günlükten Kurtarma: Kod İzleme

```
1 void kurtar(Kayıt g[], int n) {  
2   int i = 0;  
3   ► while (i < n && g[i].tur == TXB) {  
4     int j = i + 1;  
5     while (j < n && g[j].tur == BLOK)  
6       j++;  
7     if (j == n || g[j].tur != TXE)  
8       break;          /* onaysız: at */  
9     for (int k = i + 1; k < j; k++)  
10      yaz(g[k].hedef, g[k].veri);  
11     i = j + 1;        /* sonraki */  
12   }  
13 }
```

Günlük (çökmeden sonra diskte):

|          |          |         |          |          |          |           |          |          |          |
|----------|----------|---------|----------|----------|----------|-----------|----------|----------|----------|
| TxB<br>1 | I7<br>v1 | B<br>v1 | TxE<br>1 | TxB<br>2 | I9<br>v2 | D40<br>v2 | TxE<br>2 | TxB<br>3 | I7<br>v3 |
| 0        | 1        | 2       | 3        | 4        | 5        | 6         | 7        | 8        | 9        |
| ↑<br>i   |          |         |          |          |          |           |          |          |          |

Son yerler:

|          |         |          |           |
|----------|---------|----------|-----------|
| I7<br>v0 | B<br>v0 | I9<br>v0 | D40<br>v0 |
|----------|---------|----------|-----------|

g[0] bir TxB: işlem 1 başlıyor.

Adım 2 / 12

## DEĞİŞKENLER

i = 0

j = -

k = -

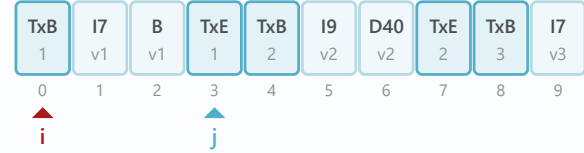
# Günlükten Kurtarma: Kod İzleme

```

1 void kurtar(Kayit g[], int n) {
2   int i = 0;
3   while (i < n && g[i].tur == TXB) {
4     int j = i + 1;
5     while (j < n && g[j].tur == BLOK)
6       j++;
7     if (j == n || g[j].tur != TXE)
8       break;          /* onaysız: at */
9     for (int k = i + 1; k < j; k++)
10       yaz(g[k].hedef, g[k].veri);
11     i = j + 1;        /* sonraki */
12   }
13 }

```

Günlük (çökmeden sonra diskte):



Son yerler:



j = 3'te **TxE 1** bulundu: işlem kesinleşmiş, blokları yeniden oynatılabilir.

Adım 3 / 12

## DEĞİŞKENLER

i = 0

j = 3

k = -

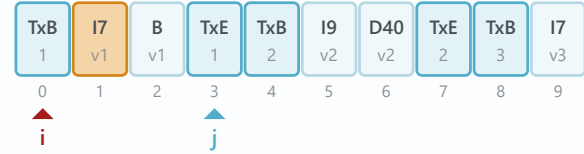
# Günlükten Kurtarma: Kod İzleme

```

1 void kurtar(Kayıt g[], int n) {
2   int i = 0;
3   while (i < n && g[i].tur == TXB) {
4     int j = i + 1;
5     while (j < n && g[j].tur == BLOK)
6       j++;
7     if (j == n || g[j].tur != TXE)
8       break;          /* onaysız: at */
9     for (int k = i + 1; k < j; k++)
10      yaz(g[k].hedef, g[k].veri);
11    i = j + 1;          /* sonraki */
12  }
13 }

```

Günlük (çökmeden sonra diskte):



Son yerler:



I7'nin v1 kopyası son yerine yazılır. Çökmeden önce yazılmış olsa da sorun yok: yeniden yazmak eşgüçlüdür.

Adım 4 / 12

## DEĞİŞKENLER

i = 0

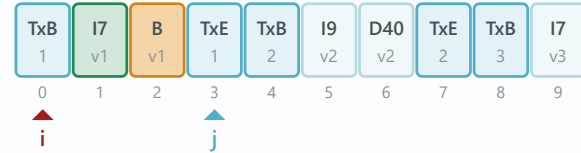
j = 3

k = 1

# Günlükten Kurtarma: Kod İzleme

```
1 void kurtar(Kayit g[], int n) {  
2   int i = 0;  
3   while (i < n && g[i].tur == TXB) {  
4     int j = i + 1;  
5     while (j < n && g[j].tur == BLOK)  
6       j++;  
7     if (j == n || g[j].tur != TXE)  
8       break;          /* onaysız: at */  
9     for (int k = i + 1; k < j; k++)  
10      yaz(g[k].hedef, g[k].veri);  
11    i = j + 1;          /* sonraki */  
12  }  
13 }
```

Günlük (çökmeden sonra diskte):



Son yerler:



B'nin v1 kopyası son yerine yazılır.

Adım 5 / 12

## DEĞİŞKENLER

i = 0

j = 3

k = 2

# Günlükten Kurtarma: Kod İzleme

```
1 void kurtar(Kayit g[], int n) {  
2   int i = 0;  
3   ► while (i < n && g[i].tur == TXB) {  
4     int j = i + 1;  
5     while (j < n && g[j].tur == BLOK)  
6       j++;  
7     if (j == n || g[j].tur != TXE)  
8       break;          /* onaysız: at */  
9     for (int k = i + 1; k < j; k++)  
10      yaz(g[k].hedef, g[k].veri);  
11     i = j + 1;        /* sonraki */  
12   }  
13 }
```

Günlük (çökmeden sonra diskte):



Son yerler:



g[4] bir TxB: işlem 2 başlıyor.

Adım 6 / 12

## DEĞİŞKENLER

i = 4

j = -

k = -

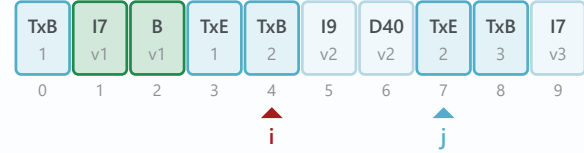
# Günlükten Kurtarma: Kod İzleme

```

1 void kurtar(Kayıt g[], int n) {
2   int i = 0;
3   while (i < n && g[i].tur == TXB) {
4     int j = i + 1;
5     while (j < n && g[j].tur == BLOK)
6       j++;
7     if (j == n || g[j].tur != TXE)
8       break;          /* onaysız: at */
9     for (int k = i + 1; k < j; k++)
10       yaz(g[k].hedef, g[k].veri);
11     i = j + 1;        /* sonraki */
12   }
13 }

```

Günlük (çökmeden sonra diskte):



Son yerler:



$j = 7$ 'de **TxE 2** bulundu: işlem kesinleşmiş, blokları yeniden oynatılabilir.

Adım 7 / 12

## DEĞİŞKENLER

$i = 4$

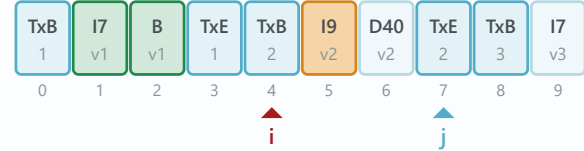
$j = 7$

$k = -$

# Günlükten Kurtarma: Kod İzleme

```
1 void kurtar(Kayit g[], int n) {  
2   int i = 0;  
3   while (i < n && g[i].tur == TXB) {  
4     int j = i + 1;  
5     while (j < n && g[j].tur == BLOK)  
6       j++;  
7     if (j == n || g[j].tur != TXE)  
8       break; /* onaysız: at */  
9     for (int k = i + 1; k < j; k++)  
10      yaz(g[k].hedef, g[k].veri);  
11    i = j + 1; /* sonraki */  
12  }  
13 }
```

Günlük (çökmeden sonra diskte):



Son yerler:



I9'un v2 kopyası son yerine yazılır.

Adım 8 / 12

## DEĞİŞKENLER

i = 4

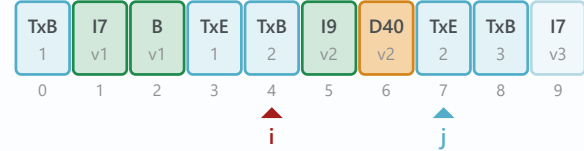
j = 7

k = 5

# Günlükten Kurtarma: Kod İzleme

```
1 void kurtar(Kayit g[], int n) {  
2   int i = 0;  
3   while (i < n && g[i].tur == TXB) {  
4     int j = i + 1;  
5     while (j < n && g[j].tur == BLOK)  
6       j++;  
7     if (j == n || g[j].tur != TXE)  
8       break;          /* onaysız: at */  
9     for (int k = i + 1; k < j; k++)  
10      yaz(g[k].hedef, g[k].veri);  
11    i = j + 1;          /* sonraki */  
12  }  
13 }
```

Günlük (çökmeden sonra diskte):



Son yerler:



D40'ın v2 kopyası son yerine yazılır.

Adım 9 / 12

## DEĞİŞKENLER

i = 4

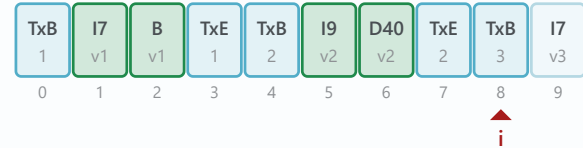
j = 7

k = 6

# Günlükten Kurtarma: Kod İzleme

```
1 void kurtar(Kayit g[], int n) {  
2   int i = 0;  
3   ► while (i < n && g[i].tur == TXB) {  
4     int j = i + 1;  
5     while (j < n && g[j].tur == BLOK)  
6       j++;  
7     if (j == n || g[j].tur != TXE)  
8       break;          /* onaysız: at */  
9     for (int k = i + 1; k < j; k++)  
10      yaz(g[k].hedef, g[k].veri);  
11     i = j + 1;        /* sonraki */  
12   }  
13 }
```

Günlük (çökmeden sonra diskte):



Son yerler:



g[8] bir TxB: işlem 3 başlıyor.

Adım 10 / 12

## DEĞİŞKENLER

i = 8

j = -

k = -

# Günlükten Kurtarma: Kod İzleme

```

1 void kurtar(Kayit g[], int n) {
2   int i = 0;
3   while (i < n && g[i].tur == TXB) {
4     int j = i + 1;
5     while (j < n && g[j].tur == BLOK)
6       j++;
7     if (j == n || g[j].tur != TXE)
8       break; /* onaysız: at */
9     for (int k = i + 1; k < j; k++)
10      yaz(g[k].hedef, g[k].veri);
11    i = j + 1; /* sonraki */
12  }
13 }

```

Günlük (çökmeden sonra diskte):



Son yerler:



Blok kayıtları atlanınca  $j = 10 = n$ : günlük bitti ama **TxE yok**. İşlem 3 onaylanmamış: **atılır**, I7'nin v3 sürümü hiç yazılmaz.

Adım 11 / 12

## DEĞİŞKENLER

i = 8

j = 10

k = -

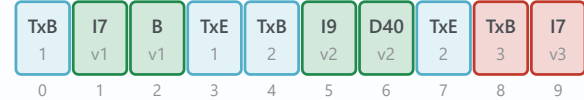
# Günlükten Kurtarma: Kod İzleme

```

1 void kurtar(Kayit g[], int n) {
2   int i = 0;
3   while (i < n && g[i].tur == TXB) {
4     int j = i + 1;
5     while (j < n && g[j].tur == BLOK)
6       j++;
7     if (j == n || g[j].tur != TXE)
8       break;          /* onaysız: at */
9     for (int k = i + 1; k < j; k++)
10      yaz(g[k].hedef, g[k].veri);
11    i = j + 1;          /* sonraki */
12  }
13 }

```

Günlük (çökmeden sonra diskte):



Son yerler:



Sonuç: işlem 1 ve 2 tam olarak uygulandı, işlem 3 hiç olmamış gibi. Dosya sistemi tutarlı; tüm disk tarayan fsck gerekmez.

Adım 12 / 12

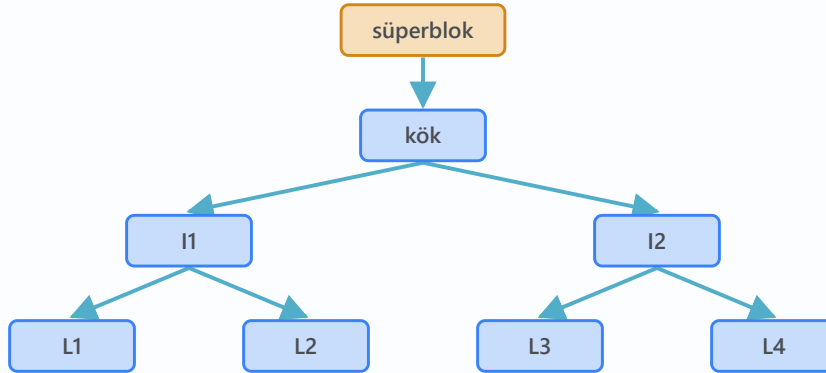
## DEĞİŞKENLER

i = 8

j = 10

k = -

## Yazınca Kopyala (CoW): ZFS, Btrfs, APFS



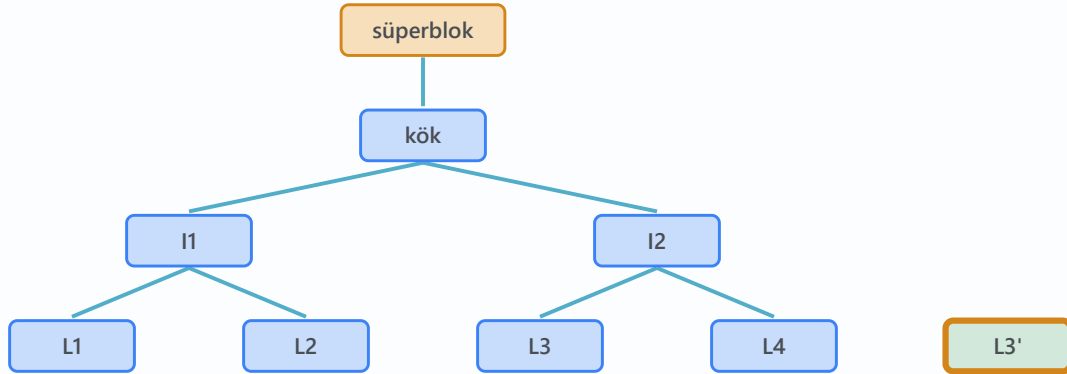
Dosya sistemi bir blok ağacıdır: süperblok (ZFS'te *uberblock*) → kök → iç düğümler → veri.

Adım 1 / 5

ZFS (Sun 2005, bugün OpenZFS), Btrfs (Linux), APFS (Apple, 2017), ReFS (Windows Server).

Bedeli: rastgele üzerine yazmalarda parçalanma ve yazma büyütmesi; veritabanı dosyalarında CoW kapatılabilir.

## Yazınca Kopyala (CoW): ZFS, Btrfs, APFS

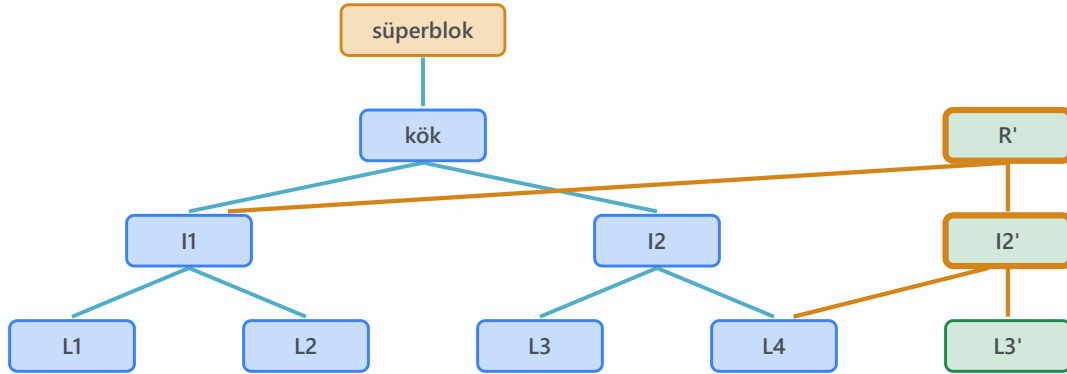


L3 değişecek: **yerinde yazılmaz**; yeni kopya L3' boş bir yere yazılır. Eski L3 hâlâ sağlam.

Adım 2 / 5

ZFS (Sun 2005, bugün OpenZFS), Btrfs (Linux), APFS (Apple, 2017), ReFS (Windows Server).  
Bedeli: rastgele üzerine yazmalarda parçalanma ve yazma büyütmesi; veritabanı dosyalarında CoW kapatılabilir.

## Yazınca Kopyala (CoW): ZFS, Btrfs, APFS

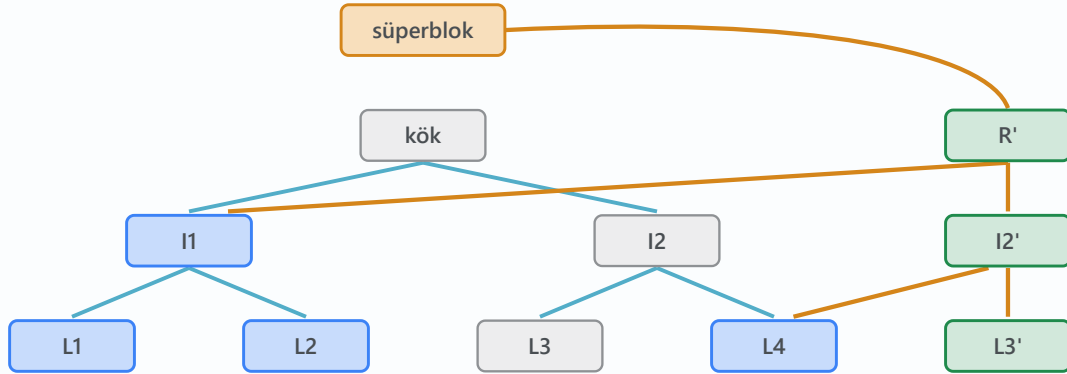


L3''ü gösteren üst düğümler de kopyalanır: I2', kök R'. Değişmeyen alt ağaçlar (I1, L4) paylaşılır.

Adım 3 / 5

ZFS (Sun 2005, bugün OpenZFS), Btrfs (Linux), APFS (Apple, 2017), ReFS (Windows Server).  
Bedeli: rastgele üzerine yazmalarda parçalanma ve yazma büyütmesi; veritabanı dosyalarında CoW kapatılabilir.

## Yazınca Kopyala (CoW): ZFS, Btrfs, APFS



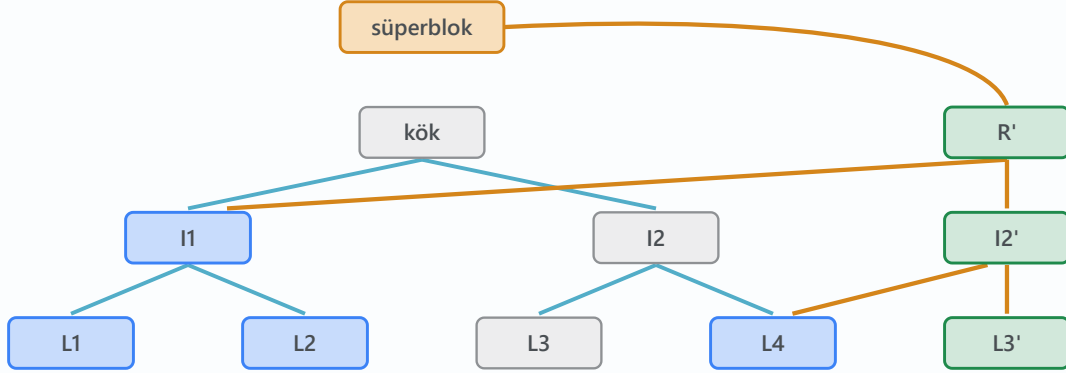
gri: eski sürüm = anlık görüntü (ya da serbest)

Tek atomik yazma: süperblok R''yi gösterir. Çökme öncesinde ya eski ya yeni ağaç görünür, ara durum yoktur; günlük gerekmez.

Adım 4 / 5

ZFS (Sun 2005, bugün OpenZFS), Btrfs (Linux), APFS (Apple, 2017), ReFS (Windows Server).  
Bedeli: rastgele üzerine yazmalarda parçalanma ve yazma büyütmesi; veritabanı dosyalarında CoW kapatılabilir.

## Yazınca Kopyala (CoW): ZFS, Btrfs, APFS



gri: eski sürüm = anlık görüntü (ya da serbest)

**her işaretçi çocuğun sağlama toplamını taşır**

Eski kök saklanırsa **anlık görüntü** bedavadır. ZFS/Btrfs her işaretçide sağlama toplamı tutar: sessiz bozulma okunurken yakalanır, aynadan onarılır (*scrub*).

Adım 5 / 5

ZFS (Sun 2005, bugün OpenZFS), Btrfs (Linux), APFS (Apple, 2017), ReFS (Windows Server).  
Bedeli: rastgele üzerine yazmalarda parçalanma ve yazma büyütmesi; veritabanı dosyalarında CoW kapatılabilir.

## Günlük Yapılı Dosya Sistemi (LFS)

bellek tamponu



disk (günlük)



kontrol  
bölgesi

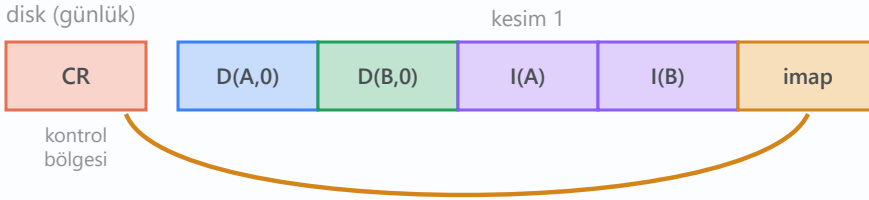
Yazmalar bellekte biriktirilir: veri blokları, değişen i-node'lar ve i-node haritası (imap) parçaları.

Adım 1 / 4

Rosenblum ve Ousterhout, Sprite LFS (1991). Okumalar büyük önbellekten karşılanır varsayımıyla tasarlandı; yazma bant genişliğini neredeyse diskin sıralı hızına çıkarır. Fikir bugün de yaşıyor: F2FS (Android), NILFS2, NetApp WAFL, ZFS/Btrfs CoW ve SSD'lerin FTL'lerinde yaşıyor.

# Günlük Yapılı Dosya Sistemi (LFS)

bellek tamponu



Tampon dolunca tümü tek bir **kesim** (segment, ör. 1–2 MiB) olarak günlüğün sonuna **sıralı** yazılır. i-node'lar sabit yerde değildir; **imap** onları bulur, kontrol bölgesi (CR) son imap'ı gösterir.

Adım 2 / 4

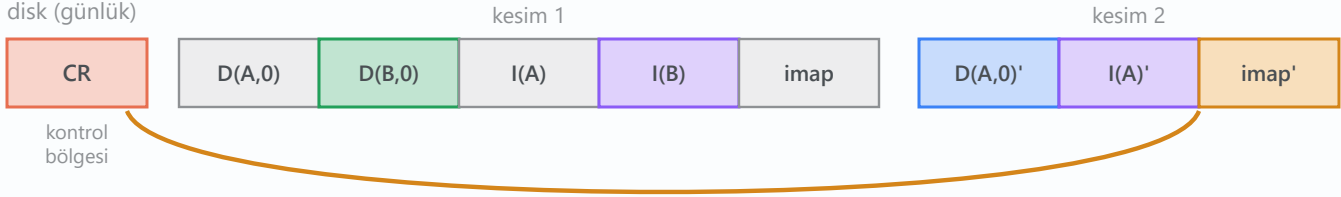
Rosenblum ve Ousterhout, Sprite LFS (1991). Okumalar büyük önbellekten karşılanır varsayımıyla tasarlandı; yazma bant genişliğini neredeyse diskin sıralı hızına çıkarır. Fikir bugün de yaşıyor: F2FS (Android), NILFS2, NetApp WAFL, ZFS/Btrfs CoW ve SSD'lerin FTL'lerinde yaşıyor.

# Günlük Yapılı Dosya Sistemi (LFS)

bellek tamponu



disk (günlük)



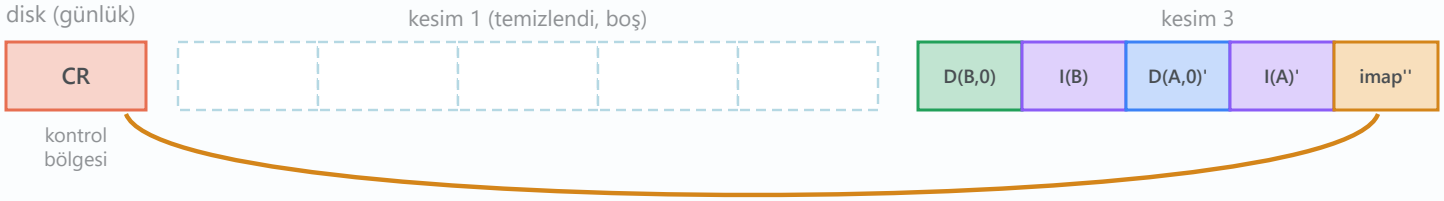
A'nın bloğu değişince yeni sürüm yeni kesime yazılır; kesim 1'deki eski D(A,0), I(A) ve imap **ölü** olur.

Adım 3 / 4

Rosenblum ve Ousterhout, Sprite LFS (1991). Okumalar büyük önbellekten karşılanır varsayımıyla tasarlandı; yazma bant genişliğini neredeyse diskin sıralı hızına çıkarır. Fikir bugün de yaşıyor: F2FS (Android), NILFS2, NetApp WAFL, ZFS/Btrfs CoW ve SSD'lerin FTL'lerinde yaşıyor.

# Günlük Yapılı Dosya Sistemi (LFS)

bellek tamponu

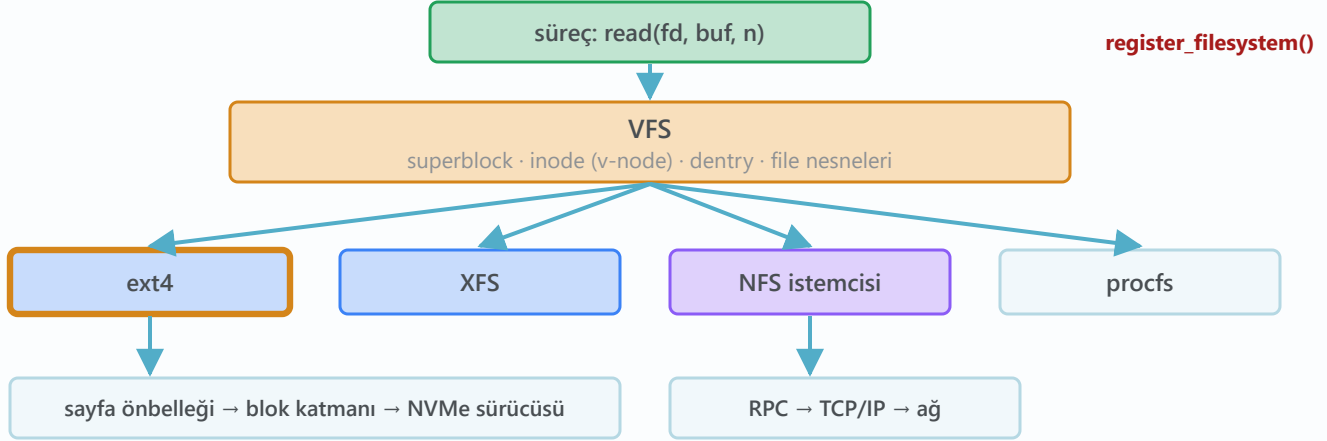


**Temizleyici:** az canlı bloğu olan kesimleri okur, canlıları yeni kesime sıkıştırır, eskiyi boşaltır. Yük altında temizleme maliyeti LFS'in zayıf noktasıdır.

Adım 4 / 4

Rosenblum ve Ousterhout, Sprite LFS (1991). Okumalar büyük önbellekten karşılanır varsayımıyla tasarlandı; yazma bant genişliğini neredeyse diskin sıralı hızına çıkarır. Fikir bugün de yaşıyor: F2FS (Android), NILFS2, NetApp WAFL, ZFS/Btrfs CoW ve SSD'lerin FTL'lerinde yaşıyor.

# Sanal Dosya Sistemi (VFS)

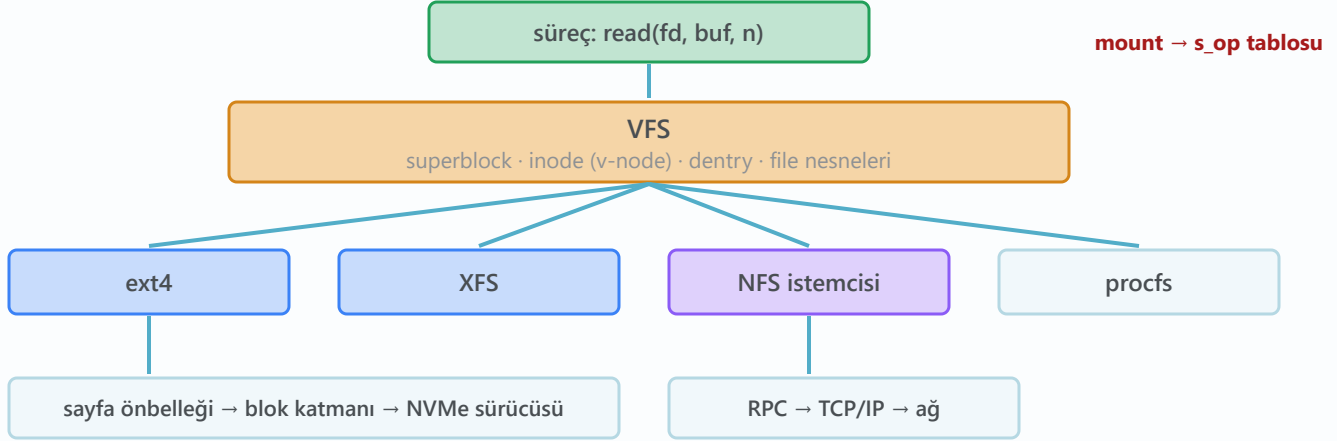


**Kayıt:** dosya sistemi açılıştan ya da modül yüklenince VFS'ye kaydolur ve VFS'nin çağıracağı işlevlerin adres tablosunu verir.

Adım 1 / 4

VFS'nin amacı farklı dosya sistemlerine *aynı anda* tek arayüz sunmaktır (aynı diskte birden çok sistem bölümlenmesi zaten mümkündür). Ek maliyeti bir dolaylı çağrıdır; performans kaybı küçüktür. Windows'ta karşılığı G/Ç yöneticisi ve dosya sistemi sürücüleridir.

# Sanal Dosya Sistemi (VFS)

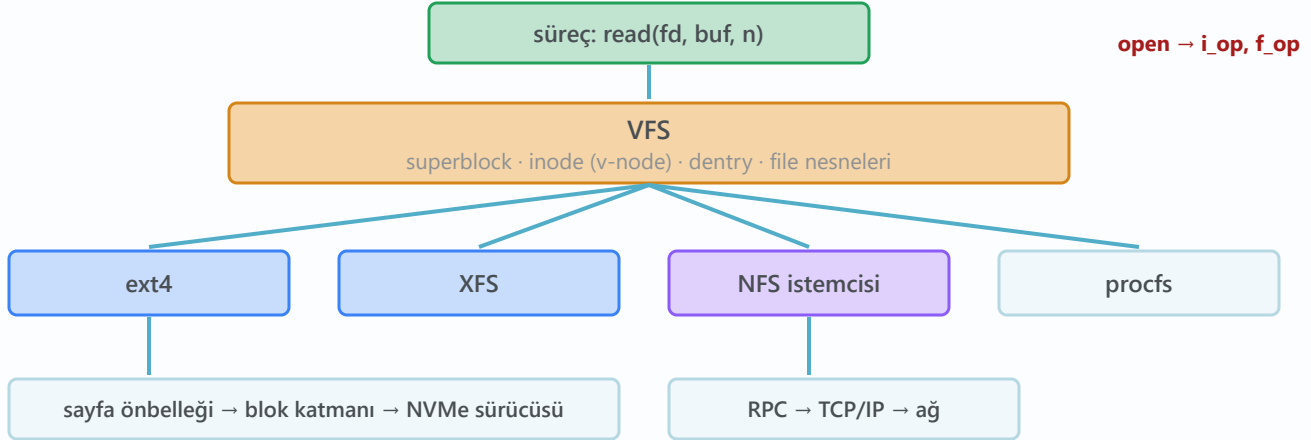


**Bağlama:** VFS o dosya sisteminden süperblok nesnesini kurmasını ister; nesne işlem tablosunu ( `s_op` ) taşır.

Adım 2 / 4

VFS'nin amacı farklı dosya sistemlerine *aynı anda* tek arayüz sunmaktır (aynı diskte birden çok sistem bölümlenmeyle zaten mümkündür). Ek maliyeti bir dolaylı çağrıdır; performans kaybı küçüktür. Windows'ta karşılığı G/Ç yöneticisi ve dosya sistemi sürücüleridir.

# Sanal Dosya Sistemi (VFS)

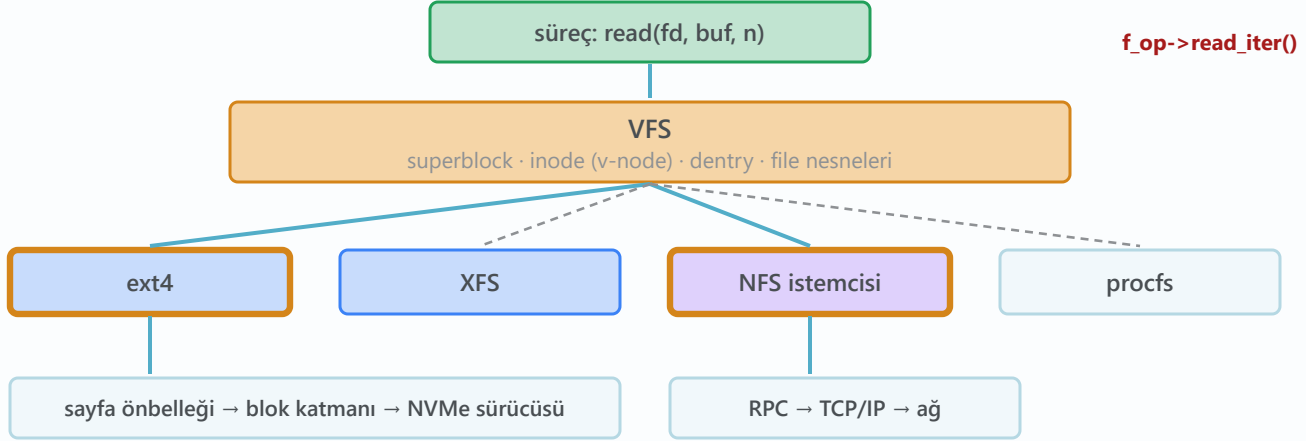


**open:** yol dentry'lerle çözülür; somut sistem i-node bilgisini bir VFS i-node'una (v-node) yerleştirir; `file` nesnesi oluşur, fd tablosuna eklenir.

Adım 3 / 4

VFS'nin amacı farklı dosya sistemlerine *aynı anda* tek arayüz sunmaktır (aynı diskte birden çok sistem bölümlenmeyle zaten mümkündür). Ek maliyeti bir dolaylı çağrıdır; performans kaybı küçüktür. Windows'ta karşılığı G/Ç yöneticisi ve dosya sistemi sürücüleridir.

# Sanal Dosya Sistemi (VFS)



**read:** VFS yalnız `file→f_op→read_iter` işlev işaretçisini çağırır; ext4 dosyasında ext4'ün işlevi, NFS dosyasında ağ üzerinden RPC çalışır. Uygulama farkı görmez.

Adım 4 / 4

VFS'nin amacı farklı dosya sistemlerine *aynı anda* tek arayüz sunmaktır (aynı diskte birden çok sistem bölümlenmeyle zaten mümkündür). Ek maliyeti bir dolaylı çağrıdır; performans kaybı küçüktür. Windows'ta karşılığı G/Ç yöneticisi ve dosya sistemi sürücüleridir.

# Ağ Dosya Sistemi (NFS)



- Uzaktaki bir dizin yerel ağaca **bağlanır**; uygulama yerel dosya gibi kullanır. Sun 1984'te SunOS için tasarladı, açık standart oldu.
- Makine ve mimari bağımsızlığı **RPC** ve **XDR** ile sağlanır. NFSv3 durumsuzdur (UDP ya da TCP); **NFSv4** durumludur, yalnız TCP, tek port, Kerberos güvenliği, kilitleme ve devretme (delegation) içerir; v4.1 pNFS.
- Windows dünyasının karşılığı SMB 3 (şifreleme, çok kanallı). NFS başlangıçta UDP/IP kullanıyordu, bugün TCP tercih edilir; tek şart IP ağıdır.

## ÖZET

# Özet

### Dosya ve dizin

Dosya: adlı, kalıcı bayt dizisi + öznitelikler.  
Dizin: ad → i-node eşlemesi; ağaç + bağlar, bağlama ile tek ağaç.

### Açık dosya

fd → sistem geneli girdi (konum) → bellek içi i-node. Yol çözümü bileşen bileşen ilerler.

### Tahsis

Bitişik (hızlı, dış parçalanma), bağlı/FAT (esnek, rastgele erişim zayıf), i-node/extent (bugünkü çözüm).

### Boş alan

Biteşlem, liste, gruptama, sayma; blok boyutu hız ile alan arasında ödünleşimdir.

### Performans

Birleşik sayfa önbelleği, önden okuma, yakın yerleşim; SSD'de TRIM.

### Tutarlılık

fsck → günlükleme → CoW/LFS. Yedek yine de şart: 3-2-1.

## ÖZET

# Kontrol Soruları

1. 6000 baytlık dosyayı 4096 baytlık tamponla okuyan döngüde `read` sırasıyla ne döndürür?
2. `ln a b` ve `ln -s a c` sonrası `rm a` yapılırsa b ve c ne olur?
3. `/usr/ast/mbox` önbelleksiz kaç disk okumasıyla çözülür (kök i-node bellekte)?
4. 200 GiB disk, 1 KiB blok, 4 B girdi: FAT ne kadar bellek ister?
5. 4 KiB blok, 4 B işaretçi, 12 doğrudan + tek/çift/üç dolaylı: en büyük dosya?
6. Aynı yapıda 5.000.000. bayt hangi düzeydedir, kaç okuma gerekir?
7. Boş bloklar 2–5, 8–13, 17–18, 25–27: sayma yöntemiyle gösterin.
8. Günlüklemede TxE yazılmadan önce ve sonra çökmenin farkı nedir?
9. SSD’de dosya silindiğinde neden TRIM gerekir?
10. i-node tablosu blok 3’te, blok 4 KiB, i-node 256 B (0’dan numaralı): i-node 40 hangi blok ve sektörde?

### ► Cevaplar

# Kaynaklar

---

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 13 (File-System Interface), 14 (File-System Implementation), 15 (File-System Internals).
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 4: File Systems.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Files and Directories", "File System Implementation", "Fast File System", "Crash Consistency: FSCK and Journaling", "Log-structured File Systems", "Flash-based SSDs", "Data Integrity", "Sun's NFS".
- M. Rosenblum, J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", ACM TOCS, 1992; M. K. McKusick ve diğ., "A Fast File System for UNIX", ACM TOCS, 1984.
- Linux çekirdek belgeleri: "Overview of the Linux Virtual File System", "ext4 Data Structures and Algorithms"; Microsoft Learn: NTFS ve exFAT belirtimi; OpenZFS ve Btrfs belgeleri; UEFI belirtimi (GPT).

# Son

## Bölüm 10: Dosya Sistemleri

Sonraki bölüm: Giriş/Çıkış · [Terim sözlüğü](#)