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) | 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

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_10_Dosya_Sistemleri.html

GİRİŞ

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; r, w ve x'in dosyada ve dizinde ne anlama geldiğini ayırt etmek.

Tahsis

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

Boş alan ve performans

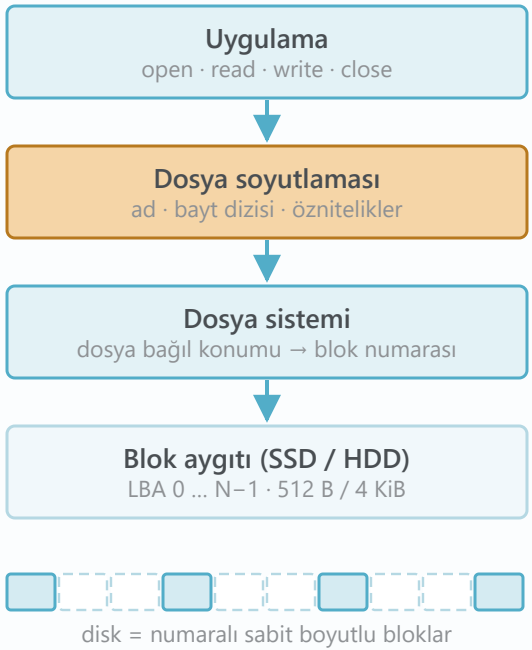
Biteşlem, liste, gruplama ve sayma yöntemlerini hesaplamak; tampon/sayfa önbelleğini ve önden okumayı açıklamak.

Güvenilirlik ve gerçek sistemler

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

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, dizin, izin, işlem — ne görünüyor?
  - **Gerçekleştirici**: bloklar nasıl tahsis edilir, boş alan nasıl izlenir, çökme sonrası tutarlılık nasıl sağlanır?



DOSYA KAVRAMI

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 *gelenek*tir; çekirdek anlam yüklemez, 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.

DOSYA KAVRAMI

Uzantılar ve Sihirli Sayılar

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

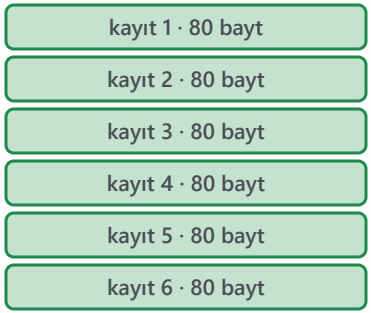
(a) Bayt dizisi



1 bayt = en küçük birim

UNIX, Windows, macOS anlamı uygulama verir

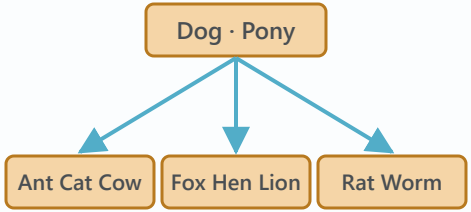
(b) Sabit uzunluklu kayıtlar



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.

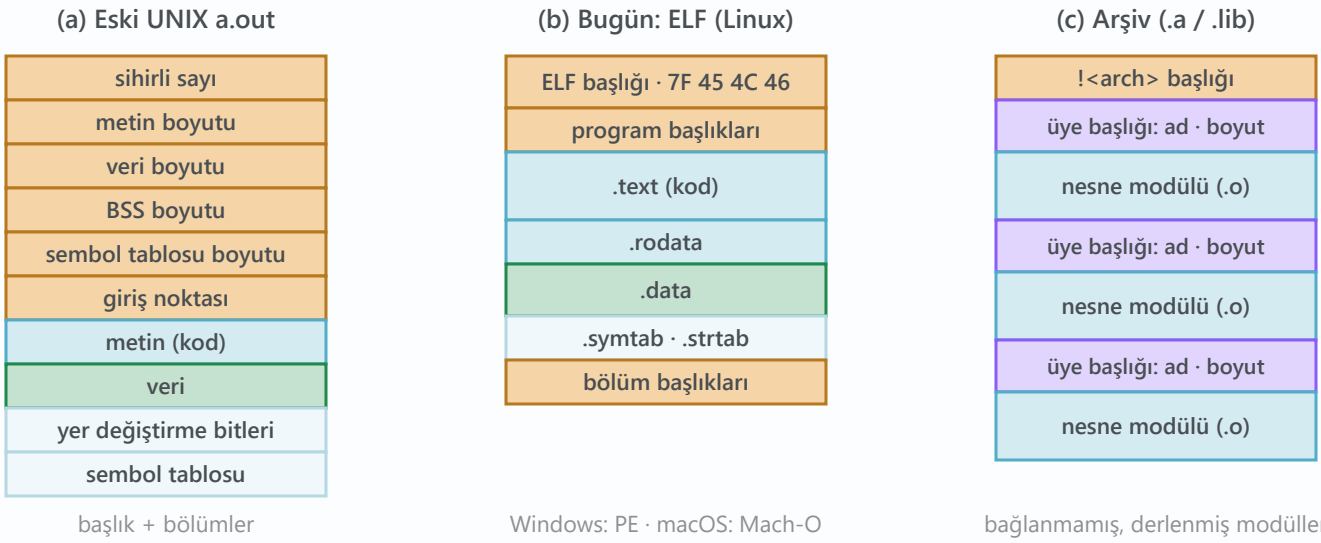
Dosya Türleri

- **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)	`notlar.txt`, `/bin/ls`
`d`	dizin	`/home`
`l`	sembolik bağ	`/bin → usr/bin`
`c`	karakter aygıtı (seri, bayt bayt)	`/dev/tty`, `/dev/null`
`b`	blok aygıtı (blok blok, rastgele)	`/dev/nvme0n1`, `/dev/sda`
`p`	adlandırılmış boru (FIFO)	`mkfifo kanal`
`s`	UNIX soketi	`/run/systemd/notify`

DOSYA KAVRAMI

İkili Dosyalar: Çalıştırılabilir ve Arşiv



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 (`st_size`) ve ayrılmış alan (`st_blocks`)	seyrek dosyada ikisi farklıdır
Sahip, grup	UID / GID; Windows'ta SID	`chown` , `icacls`
Koruma	rwx bitleri, ACL	`chmod` , `setfacl`
Zaman damgaları	erişim (atime), içerik değişimi (mtime), i-node değişimi (ctime), oluşturma (btime)	`stat` ; Linux'ta `relatime`
Bağ sayısı	dosyaya işaret eden ad sayısı	0 olunca bloklar serbest
Read-only, Hidden, System	salt okunur, gizli, sistem dosyası	Windows `attrib` ; UNIX'te gizli = ad `.` 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.

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

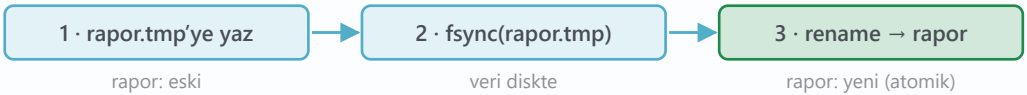
ad i-node'da değil, dizindedir!

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

DOSYA KAVRAMI

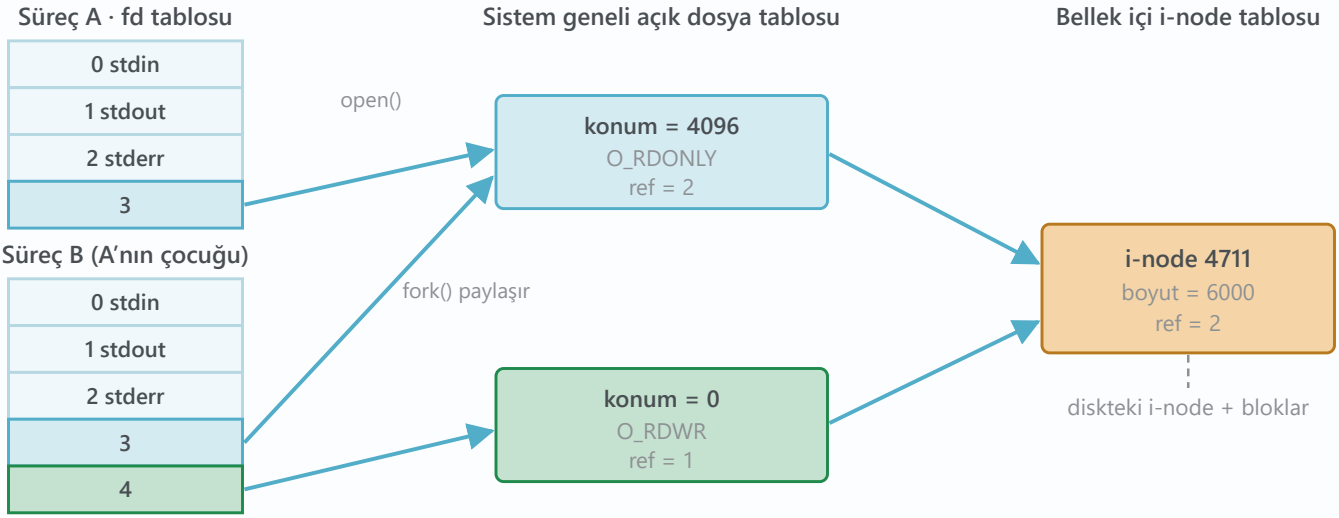
Dosya İşlemleri ve Sistem Çağrıları

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

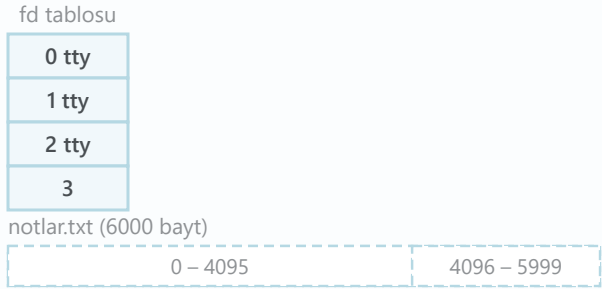
Açık Dosya Tabloları



- **fd**: çekirdeğin süreç başına tablosunda bir indis; 0, 1, 2 standart girdi/çıktı/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) {
▶    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 }
```



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

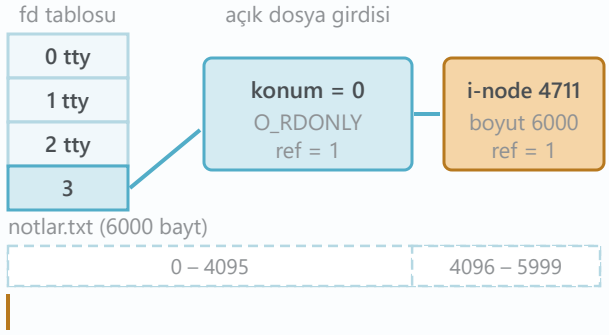
Adım 1 / 6

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 }
```



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

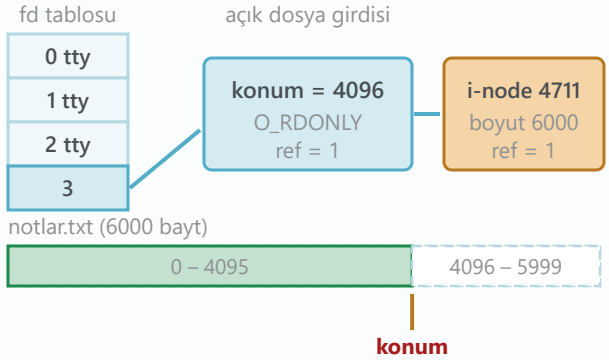
Adım 2 / 6

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);
▶    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) önbelleğe/okunur, 4096 bayt `buf` 'a kopyalanır, konum **4096** olur.

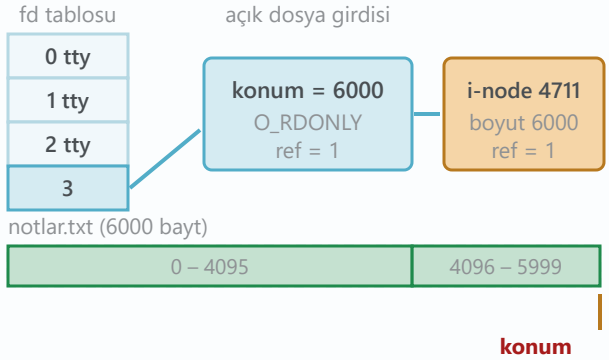
Adım 3 / 6

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);
▶    n = read(fd, buf, sizeof buf);
9    n = read(fd, buf, sizeof buf);
10   close(fd);
11   return 0;
12 }
```



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

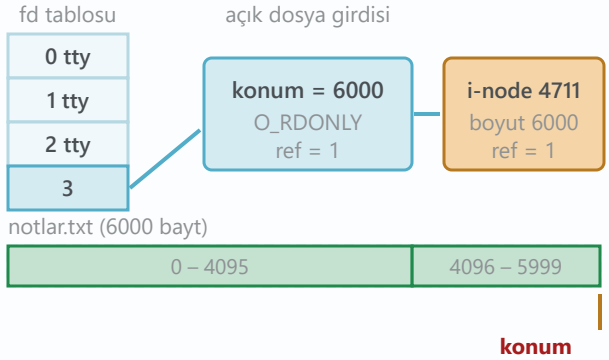
Adım 4 / 6

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);
▶    n = read(fd, buf, sizeof buf);
10   close(fd);
11   return 0;
12 }
```



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

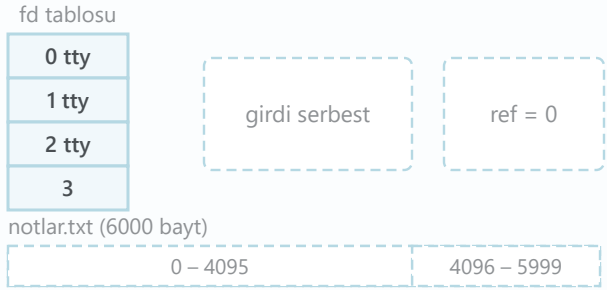
Adım 5 / 6

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);
▶    close(fd);
11   return 0;
12 }
```



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

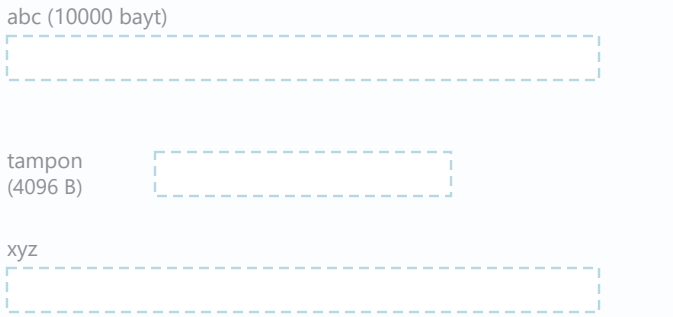
Adım 6 / 6

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    char buf[BUF];
▶    int in  = open(src, O_RDONLY);
▶    int out = open(dst, O_WRONLY | O_CREAT
▶                        | O_TRUNC, 0644);
8    ssize_t n;
9    while ((n = read(in, buf, BUF)) > 0)
10     write(out, buf, n);
11   close(in);
12   close(out);
13   return 0;
14 }
```



`in = 3` ; `out = 4` . `O_TRUNC` : xyz varsa içeriği silinir, yoksa `O_CREAT` ile 0644 izinle oluşturulur.

Adım 1 / 6

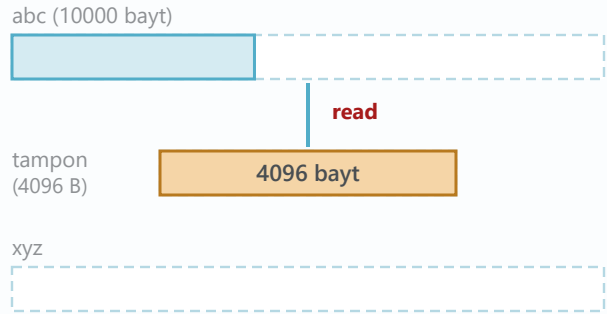
DEĞİŞKENLER

`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    char buf[BUF];
5    int in  = open(src, O_RDONLY);
6    int out = open(dst, O_WRONLY | O_CREAT
7                        | O_TRUNC, 0644);
8    ssize_t n;
▶    while ((n = read(in, buf, BUF)) > 0)
10     write(out, buf, n);
11   close(in);
12   close(out);
13   return 0;
14 }
```



İlk `read` : 4096 bayt tampona.

Adım 2 / 6

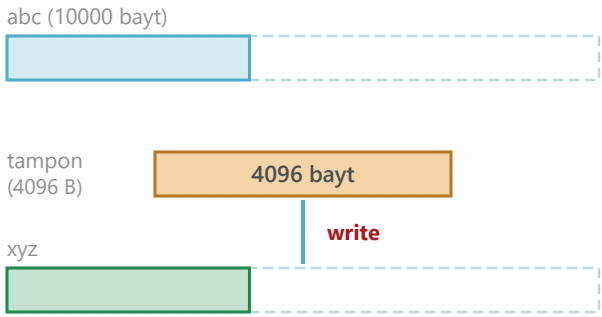
DEĞİŞKENLER

`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    char buf[BUF];
5    int in  = open(src, O_RDONLY);
6    int out = open(dst, O_WRONLY | O_CREAT
7                        | O_TRUNC, 0644);
8    ssize_t n;
9    while ((n = read(in, buf, BUF)) > 0)
▶      write(out, buf, n);
11   close(in);
12   close(out);
13   return 0;
14 }
```



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

Adım 3 / 6

DEĞİŞKENLER

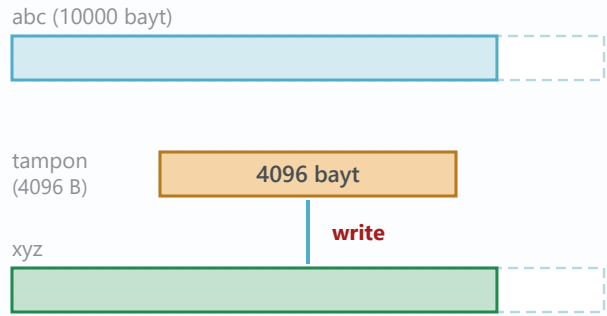
`in = 3` `out = 4` `n = 4096`

`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    char buf[BUF];
5    int in  = open(src, O_RDONLY);
6    int out = open(dst, O_WRONLY | O_CREAT
7                        | O_TRUNC, 0644);
8    ssize_t n;
▶    while ((n = read(in, buf, BUF)) > 0)
▶      write(out, buf, n);
11   close(in);
12   close(out);
13   return 0;
14 }
```



İkinci tur: 8192 bayt kopyalandı.

Adım 4 / 6

DEĞİŞKENLER

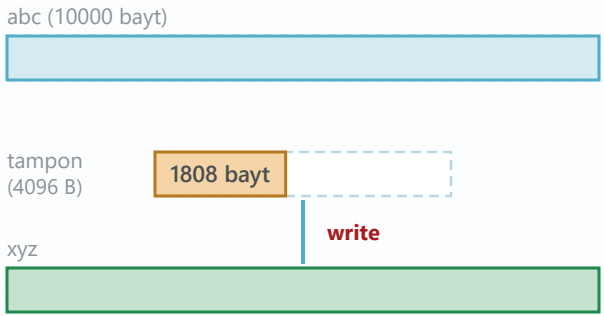
`in = 3` `out = 4` `n = 4096`

`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    char buf[BUF];
5    int in  = open(src, O_RDONLY);
6    int out = open(dst, O_WRONLY | O_CREAT
7                        | O_TRUNC, 0644);
8    ssize_t n;
▶    while ((n = read(in, buf, BUF)) > 0)
▶      write(out, buf, n);
11   close(in);
12   close(out);
13   return 0;
14 }
```



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

Adım 5 / 6

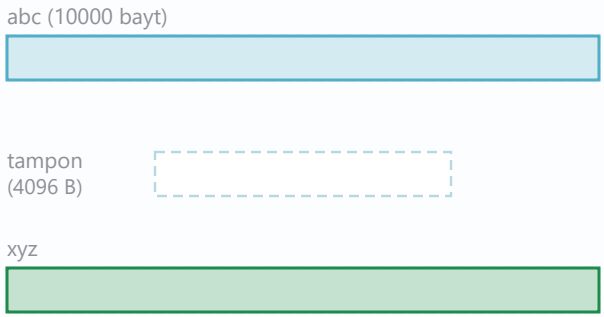
DEĞİŞKENLER

`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    char buf[BUF];
5    int in  = open(src, O_RDONLY);
6    int out = open(dst, O_WRONLY | O_CREAT
7                        | O_TRUNC, 0644);
8    ssize_t n;
9    while ((n = read(in, buf, BUF)) > 0)
10     write(out, buf, n);
▶    close(in);
▶    close(out);
13   return 0;
14 }
```



`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

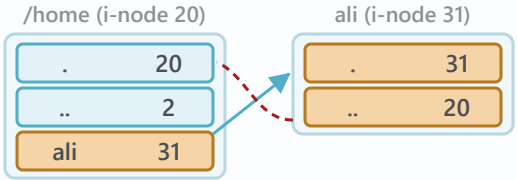


- **Sıralı**: baştan sona sırayla; atlama yerine sarma. Teyp ve kaset böyleydi; bugün de günlük dosyaları ve akışlar için en yaygın kalıptır.
- **Doğrudan**: dosya numaralı bloklar/kayıtlar dizisidir; istenen yerden başlanır (`lseek` , `pread`). Diskler ve SSD'ler bunu destekler.
- **İndisli**: küçük 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.

Dizin İşlemleri

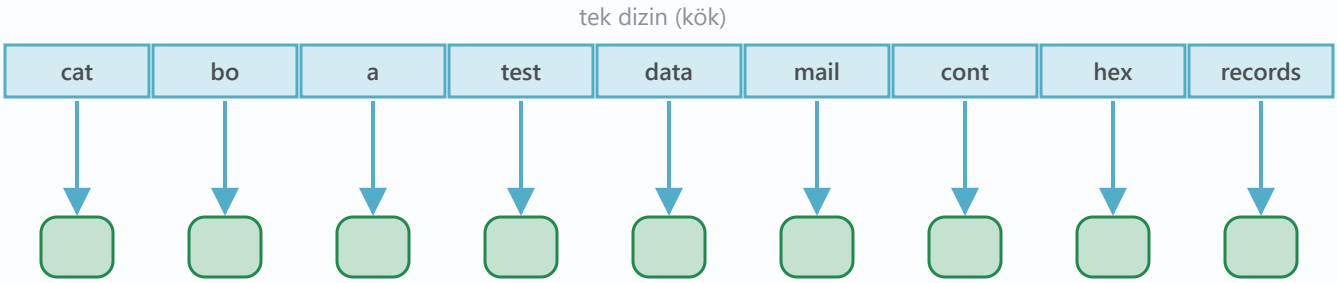
Çağrı	İşlev
`mkdir`	boş dizin oluşturur (içinde yalnız `.` ve `..`)
`rmdir`	dizini siler; dizin **boş** olmalıdır
`opendir` / `closedir`	dizin akışını açar / kapatır
`readdir`	açık dizindeki bir sonraki girdiyi (ad, i-node no., tür) döndürür
`rename`	dosya ya da dizini yeniden adlandırır veya taşır (aynı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

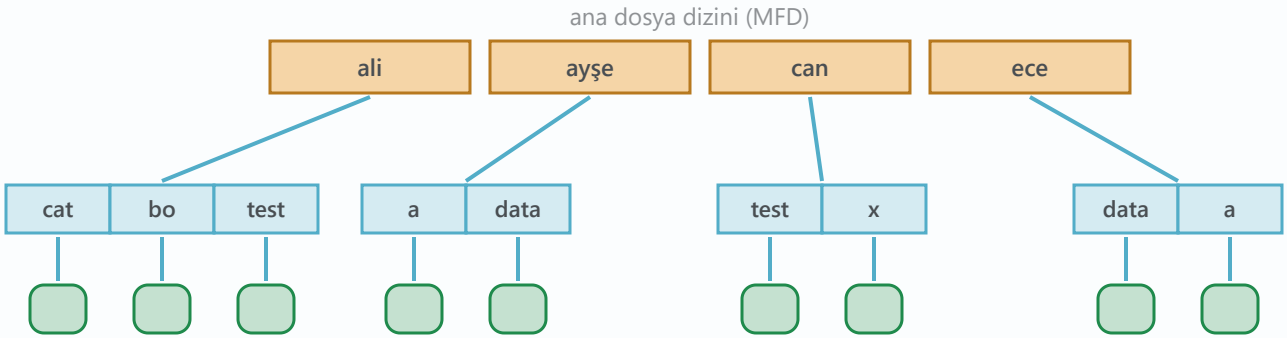
Dizin Yapıları



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.

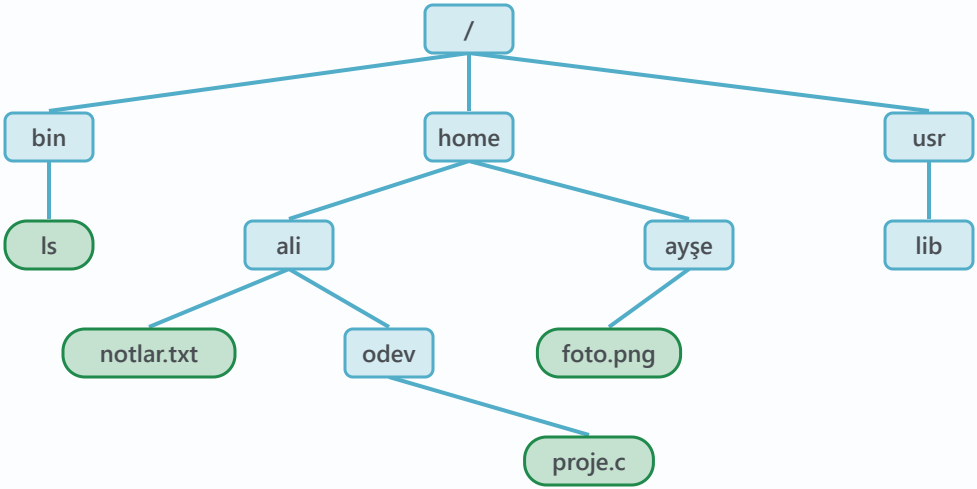
Dizin Yapıları



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

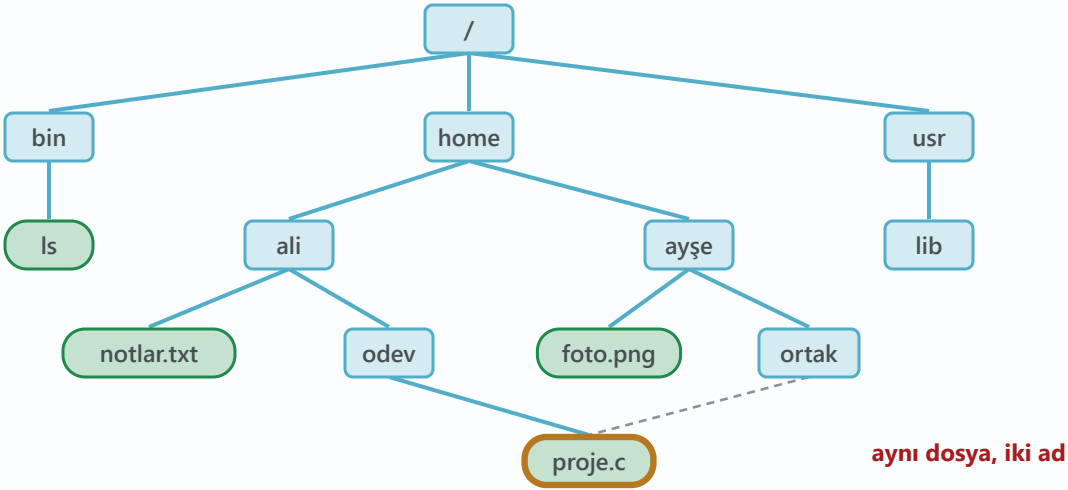
İki düzeyli: yol = `kullanıcı/dosya`. Sistem programları için ayrı bir "kullanıcı" dizini gerekir; aranacak dizin listesi (*search path*) fikri buradan gelir.

Dizin Yapıları



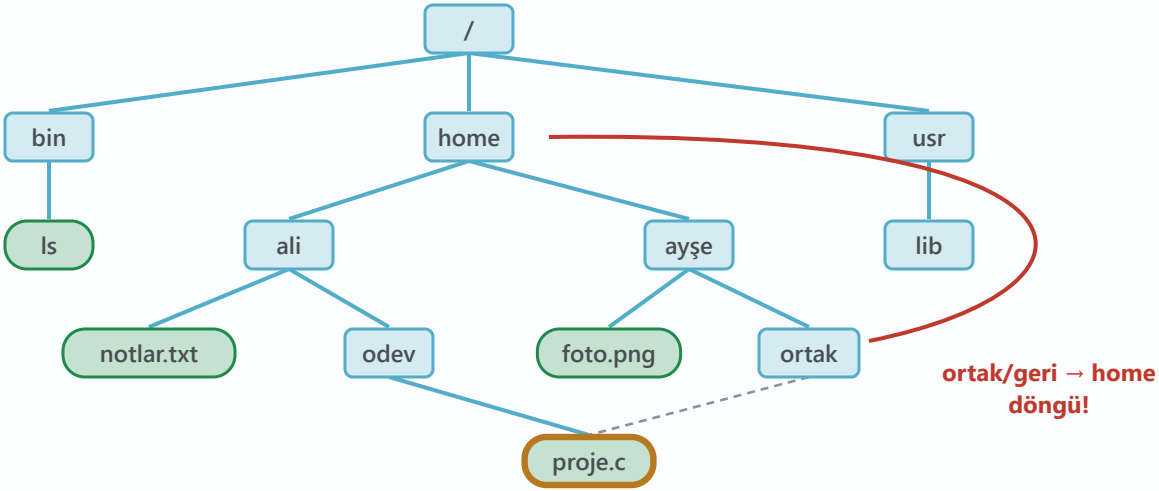
Ağaç (hiyerarşik): kullanıcı istediği kadar alt dizin açar; her dosyanın tek bir mutlak yolu vardır. Bugünkü tüm genel amaçlı sistemlerin temeli.

Dizin Yapıları



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

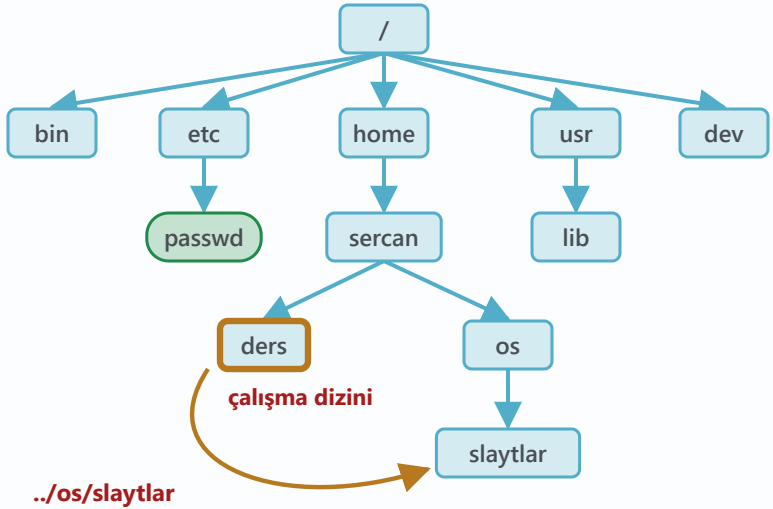
Dizin Yapıları



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

Yol Adları

- **Mutlak yol** kökten başlar: `/home/sercan/os/slaytlar`. Windows: `C:\Users\sercan\os`; ağ paylaşımı `\\sunucu\pay`.
- **Göreli 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

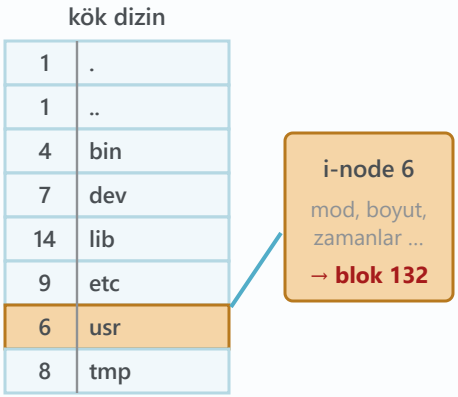
1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Mutlak yol: kök dizinin i-node'u (bu örnekte 1; V7 UNIX ve 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

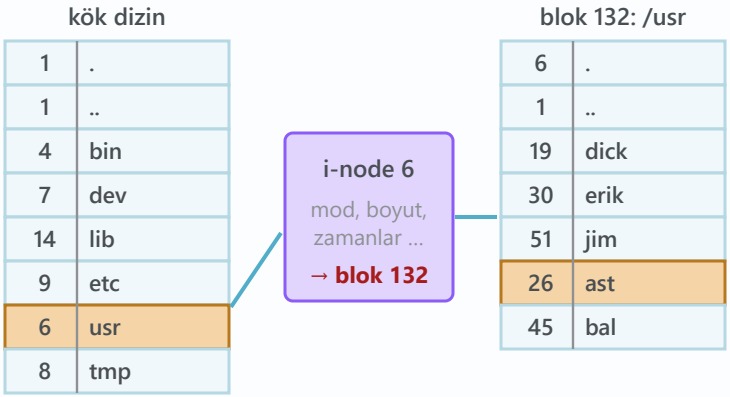


i-node 6 okunur: `/usr` bir dizin, içeriği **blok 132**'de. (Burada izin denetimi de yapılır: dizinde `x` hakkı gerekir.)

Adım 2 / 5

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

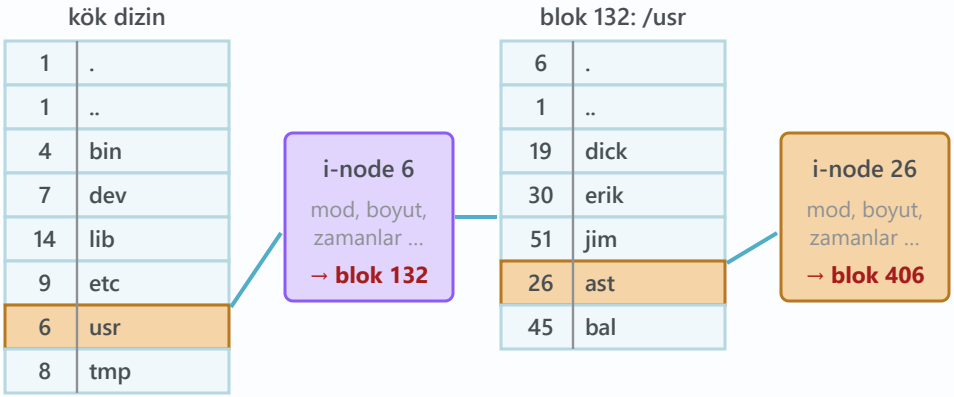
Yol Çözümleme Adım Adım



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

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

Yol Çözümleme Adım Adım

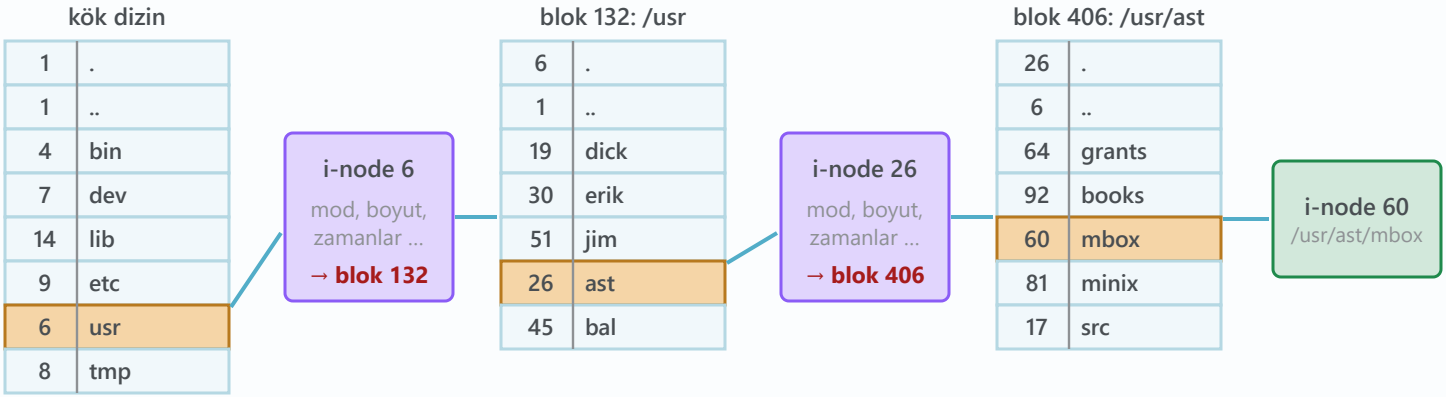


i-node 26: `/usr/ast` dizininin içeriği **blok 406**'da.

Adım 4 / 5

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

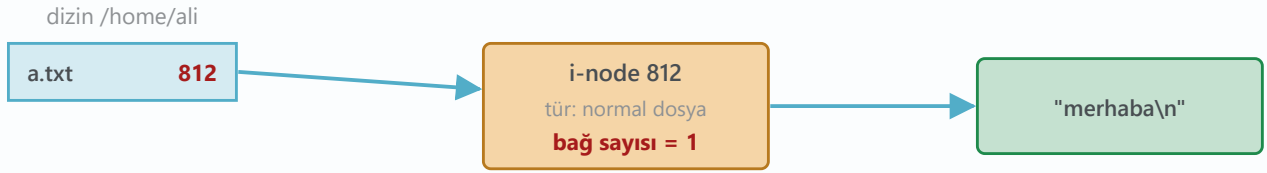
Yol Çözümleme Adım Adım



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

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

Katı ve Sembolik Bağlar

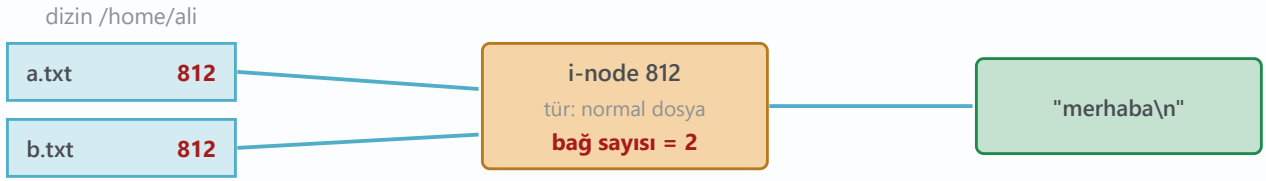


`echo merhaba > a.txt` : dizin girdisi **a.txt** → **i-node 812**, bağ sayısı 1.

Adım 1 / 4

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

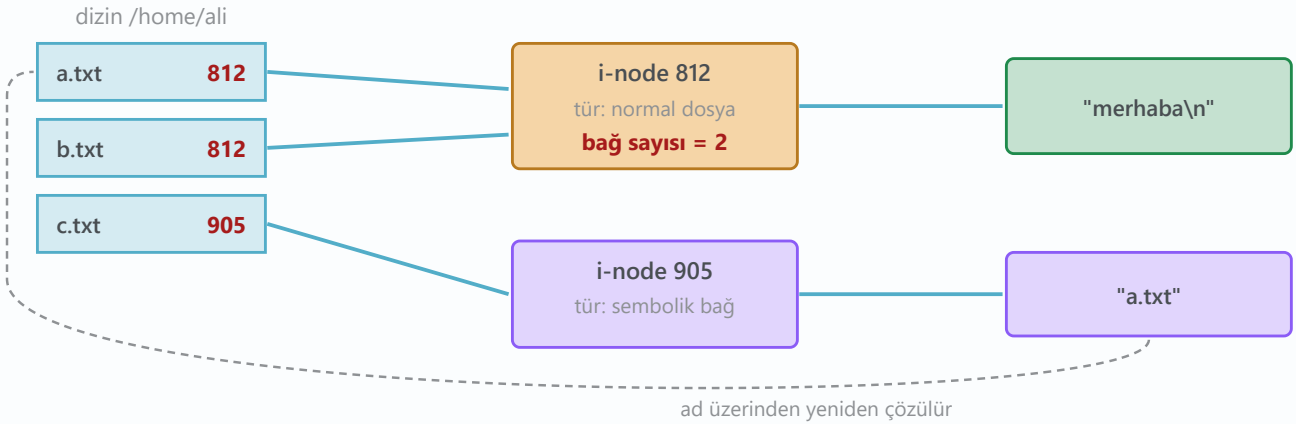
Katı ve Sembolik Bağlar



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

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

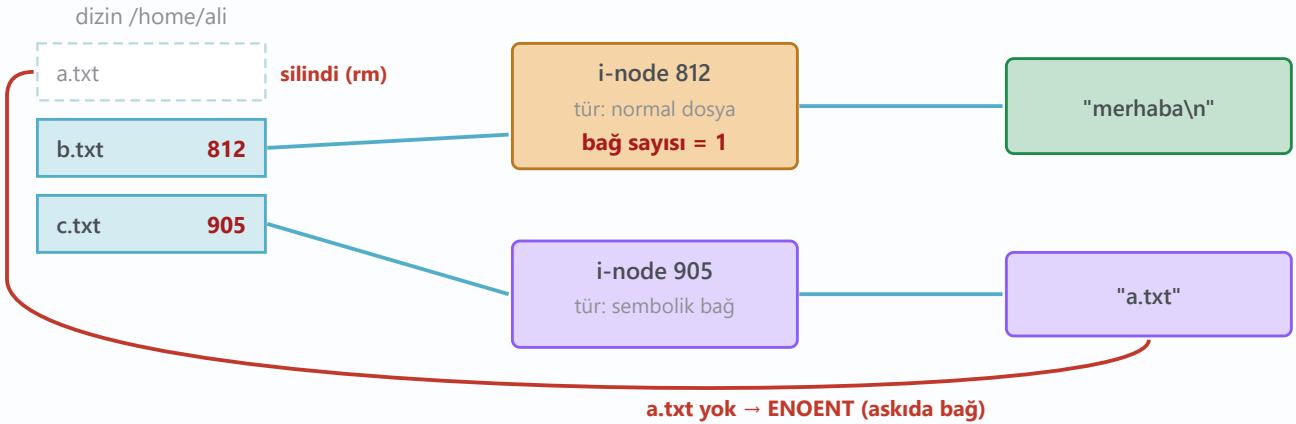
Katı ve Sembolik Bağlar



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

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

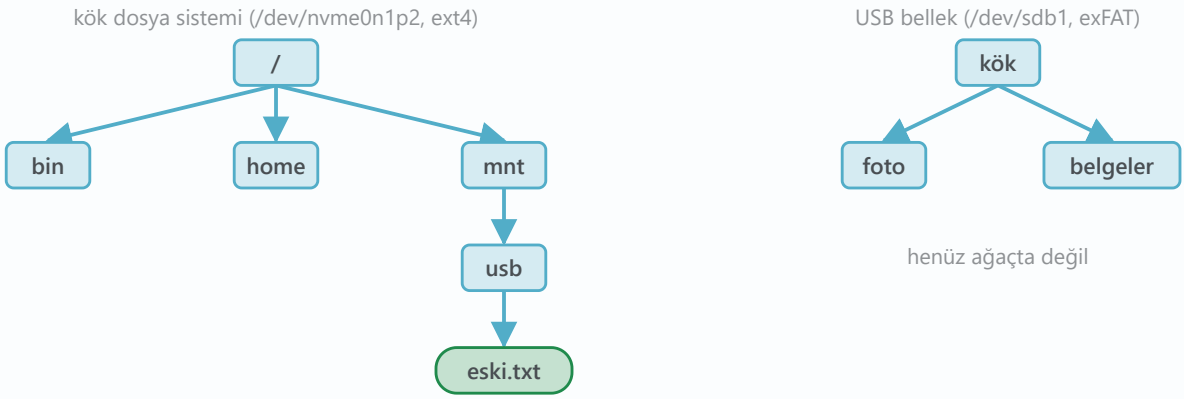
Katı ve Sembolik Bağlar



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

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

Bağlama (Mounting)

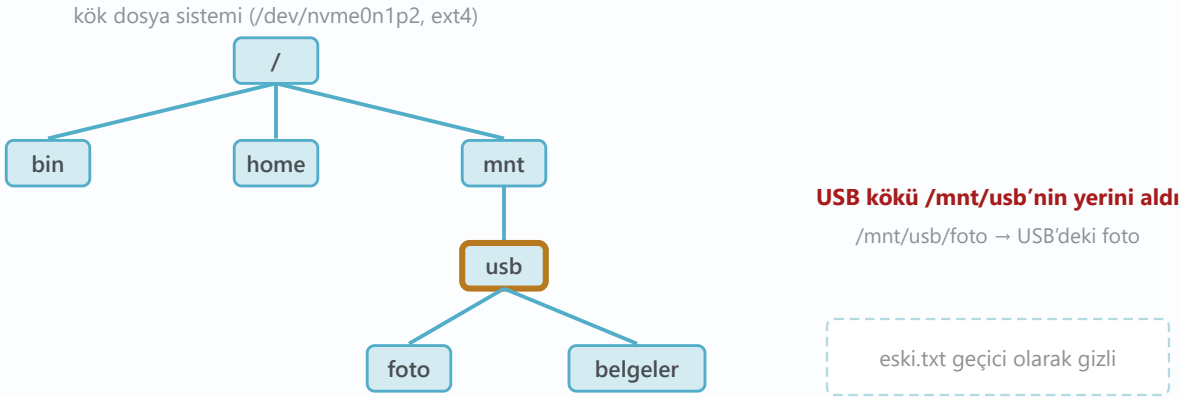


İki ayrı dosya sistemi. `/mnt/usb` sıradan bir dizin ve içinde `eski.txt` var.

Adım 1 / 3

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)

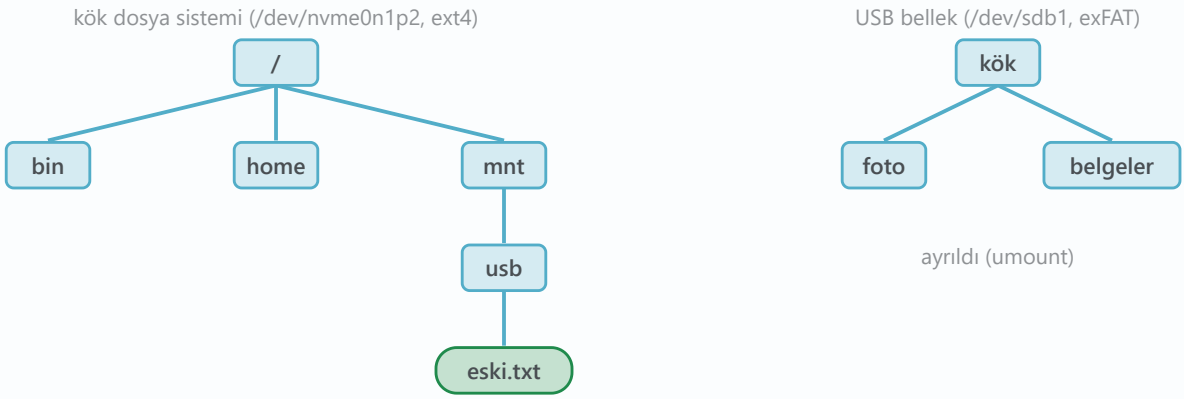


`mount /dev/sdb1 /mnt/usb` : çekirdek bağlama tablosuna kayıt ekler; yol çözümü `/mnt/usb` 'ye gelince USB'nin kök i-node'una atlar. Eski içerik **gizlenir**, silinmez.

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)



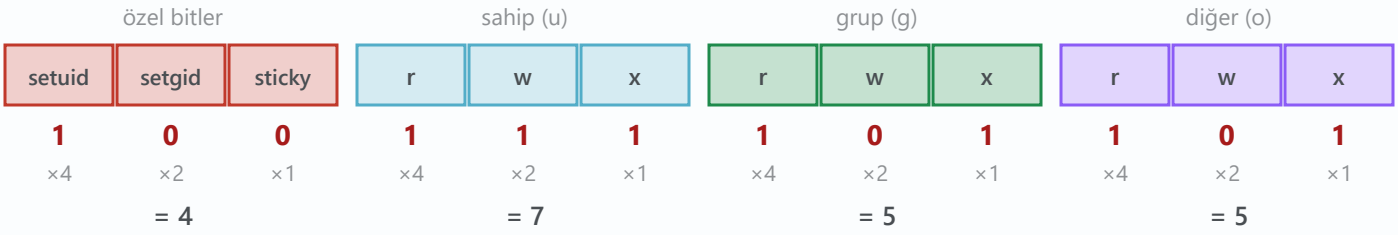
`umount /mnt/usb` : açık dosya yoksa ayrılır (yoksa `EBUSY`); `eski.txt` yeniden görünür.

Adım 3 / 3

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.

KORUMA

Erişim Hakları: rwx



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

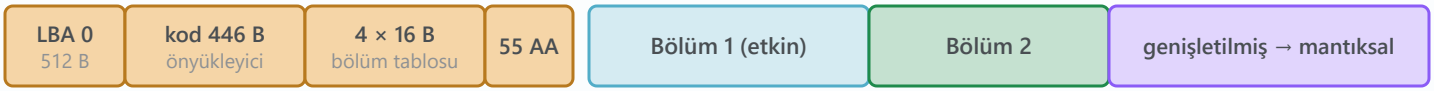
Hak	Dosyada	Dizinde
`r`	içeriği oku	girdileri listele (`ls`)
`w`	içeriği değiştir	girdi ekle/sil/yeniden adlandır (dosyanın kendi izni önemsiz!)
`x`	program olarak çalıştır	içinden geç (yol çözümü, `cd`)

- 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 `drwxrwx--T` : sticky açık, diğerlerinde `x` yok → büyük `T` .
- Sahip/grup/diğer ayrımı kaba tanelidir; adlı kullanıcı ve grup girdileri için erişim denetim listesi (ACL) kullanılır, `ls` çıktısında `+` görünür.

DİSK DÜZENİ

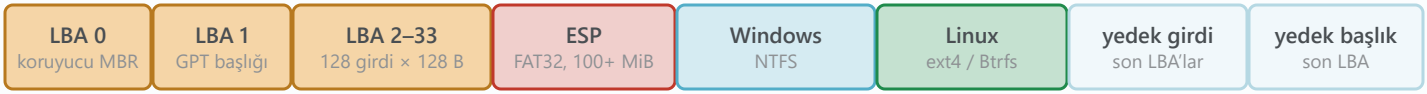
Disk Bölümleme: MBR ve GPT

MBR (BIOS, 1983)



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

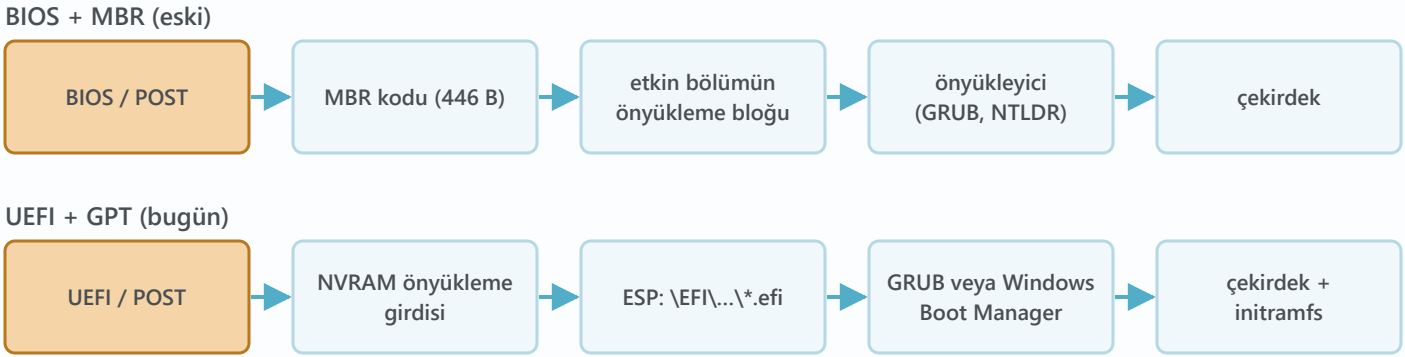
GPT (UEFI, 2000'ler)



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

- Disk bir veya daha çok **bölüme** (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.

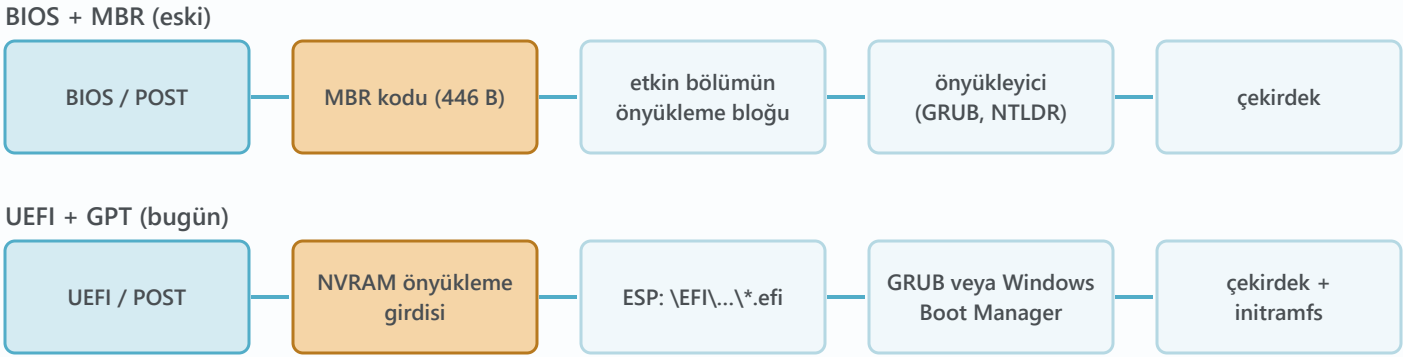
Açılış: BIOS/MBR ve UEFI/GPT



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

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

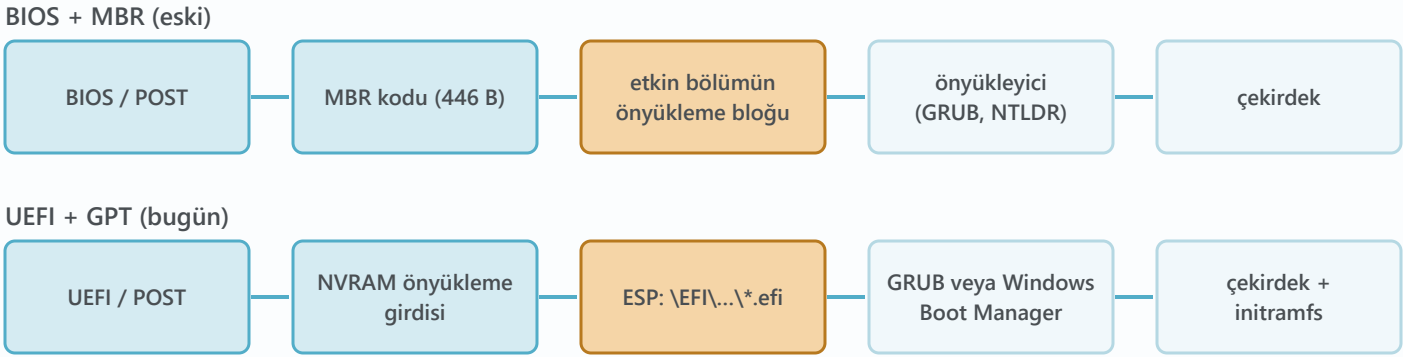
Açılış: BIOS/MBR ve UEFI/GPT



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

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

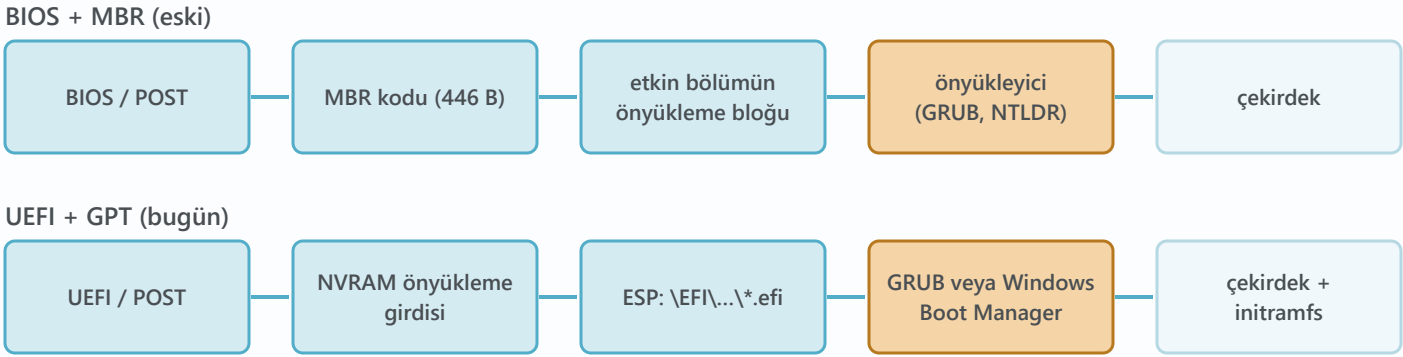
Açılış: BIOS/MBR ve UEFI/GPT



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

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

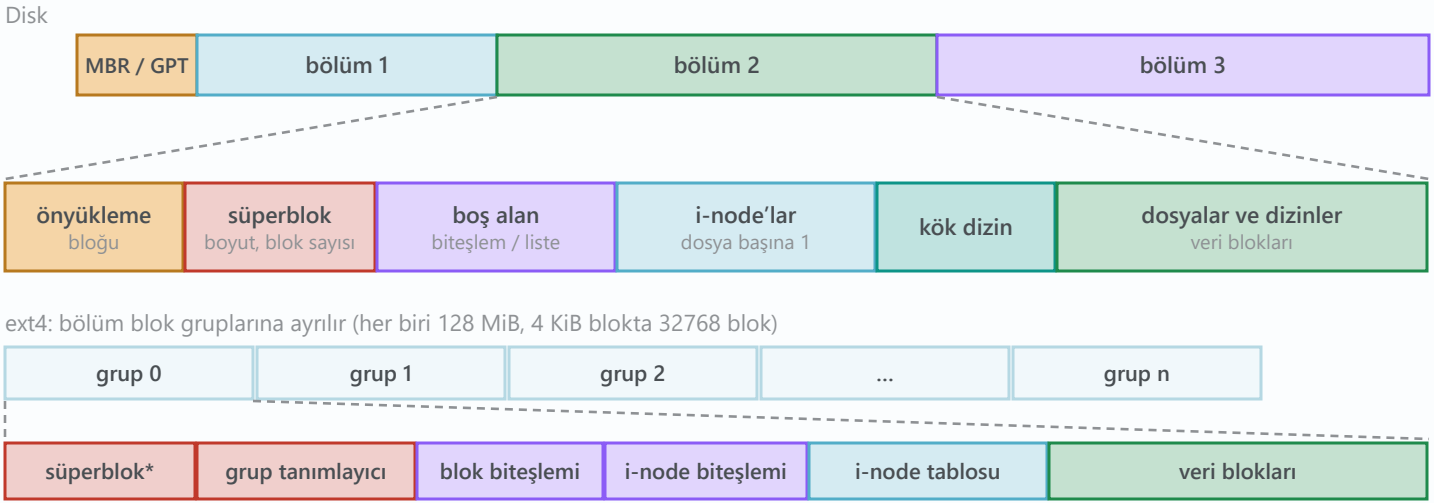
Açılış: BIOS/MBR ve UEFI/GPT



Her iki düzende de bundan sonra önyükleyici çekirdeği yükler ve çalıştırır; fark yalnızca önyükleyiciye nasıl ulaşıldığındadır.

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

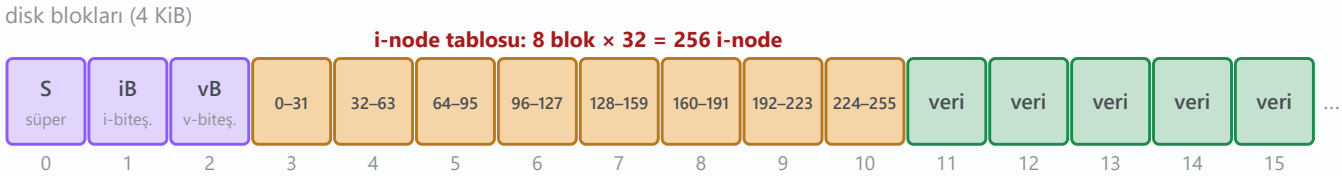
Dosya Sistemi Düzeni



* 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

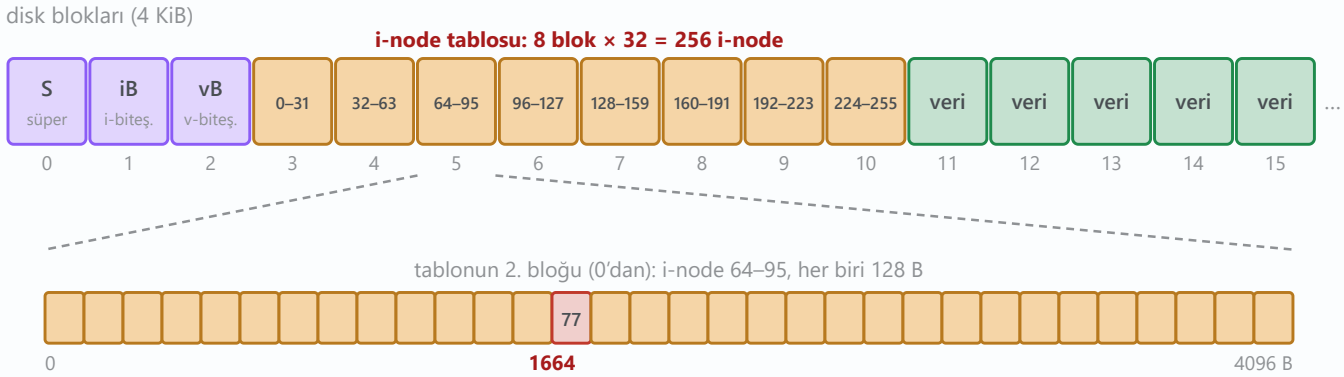


Örnek düzen: 4 KiB blok, 128 B i-node, yani blok başına **32 i-node**. Blok 0 süperblok, 1–2 biteşlemler, 3–10 i-node tablosu. Soru: **i-node 77** (numaralar 0'dan) diskte nerede?

Adım 1 / 4

ext4'te numaralar 1'den başlar ve tablo blok gruplarına bölünür: grup = (n − 1) div g, indis = (n − 1) mod g. g = 8192 i-node/grup, 256 B i-node ile i-node 20.000 → grup 2, indis 3615 → grubun tablosunda blok 225, bağıl konum 3840.

i-node Numarasından Disk Adresine

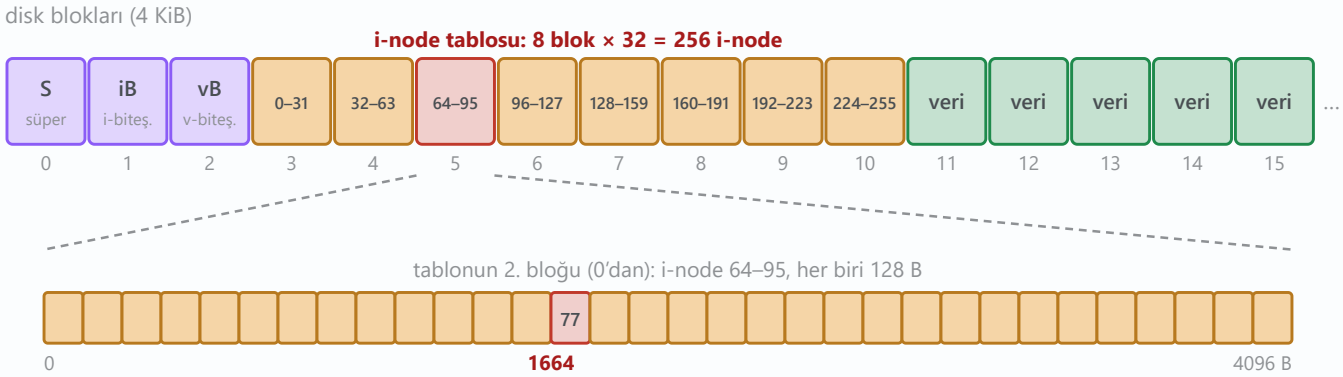


Tablo başından bayt bağıl konumu: 77 × 128 = **9856** = 2 × 4096 + 1664. Yani tablonun **2. bloğu**, blok içinde **1664**. bayt (o bloğun 13. i-node'u).

Adım 2 / 4

ext4'te numaralar 1'den başlar ve tablo blok gruplarına bölünür: grup = (n − 1) div g, indis = (n − 1) mod g. g = 8192 i-node/grup, 256 B i-node ile i-node 20.000 → grup 2, indis 3615 → grubun tablosunda blok 225, bağıl konum 3840.

i-node Numarasından Disk Adresine

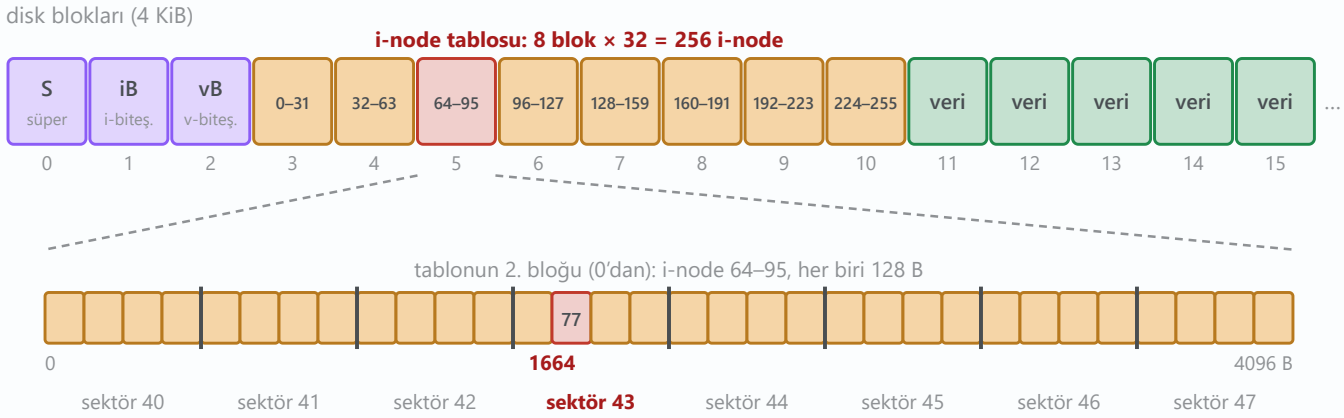


Disk bloğu = tablonun başlangıcı + 2 = 3 + 2 = **blok 5**. Süperbloktaki "i-node tablosu nerede" bilgisi bu yüzden gereklidir.

Adım 3 / 4

ext4'te numaralar 1'den başlar ve tablo blok gruplarına bölünür: grup = (n − 1) div g, indis = (n − 1) mod g. g = 8192 i-node/grup, 256 B i-node ile i-node 20.000 → grup 2, indis 3615 → grubun tablosunda blok 225, bağıl konum 3840.

i-node Numarasından Disk Adresine

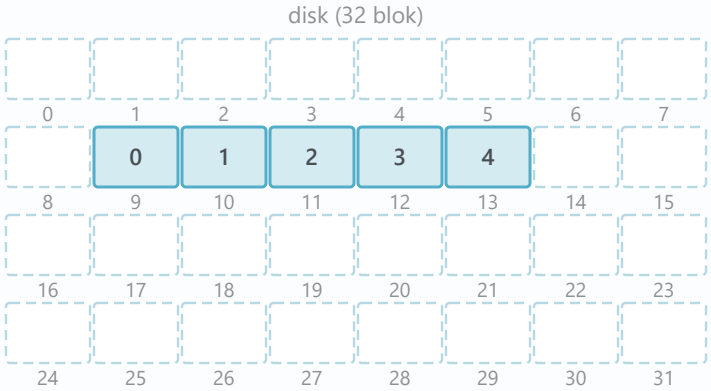


Disk sektör okur: bayt adresi 5 × 4096 + 1664 = 22.144 → 22.144 / 512 = **sektör 43**, içinde 128. bayt. Bir sektörde 4 i-node vardır.

Adım 4 / 4

ext4'te numaralar 1'den başlar ve tablo blok gruplarına bölünür: grup = (n − 1) div g, indis = (n − 1) mod g. g = 8192 i-node/grup, 256 B i-node ile i-node 20.000 → grup 2, indis 3615 → grubun tablosunda blok 225, bağıl konum 3840.

Tahsis Yöntemleri: Aynı Dosya, Dört Yol



dizin girdisi

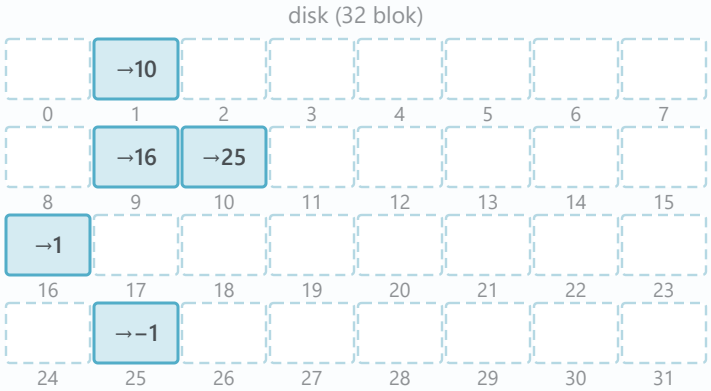
ad	başlangıç	uzunluk
jeep	9	5

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

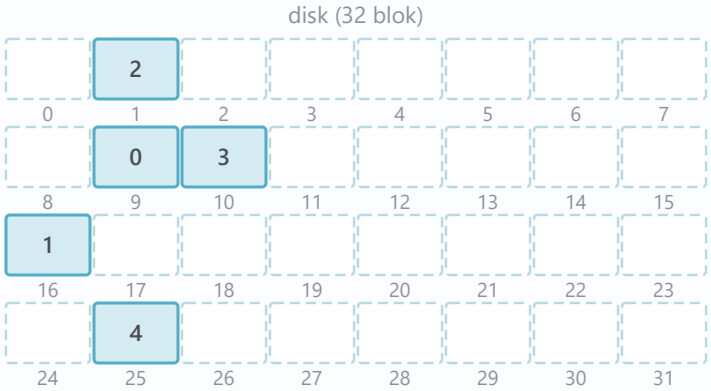
ad	ilk	son
jeep	9	25

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

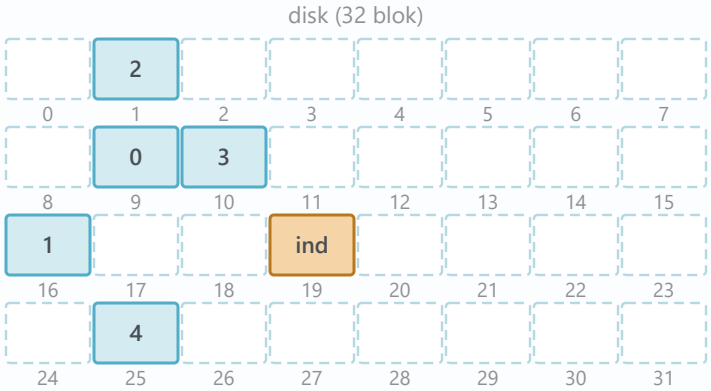
FAT (bellekte)

1	10
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

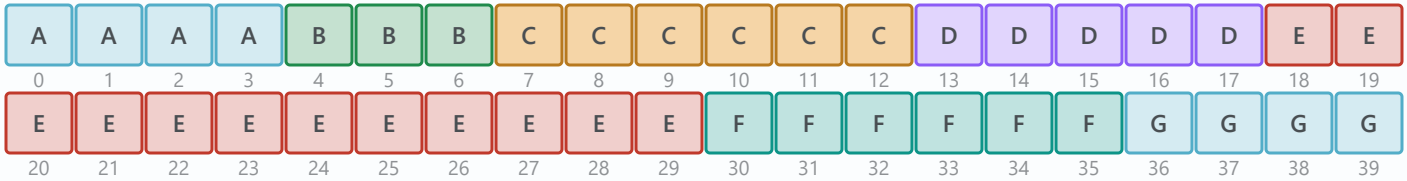
indis bloğu 19

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 · B3 · C6 · D5 · E12 · F6 · G4 = 40 blok, disk dolu

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

Adım 1 / 4

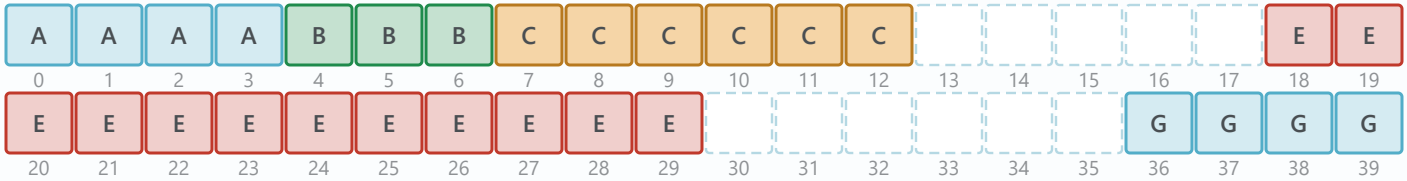
Artılar

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.

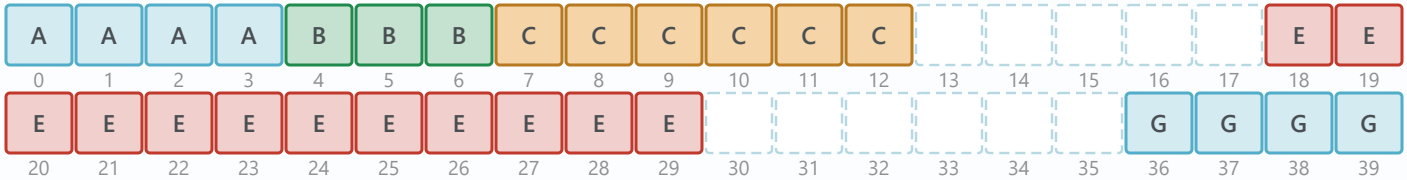
Artılar

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.

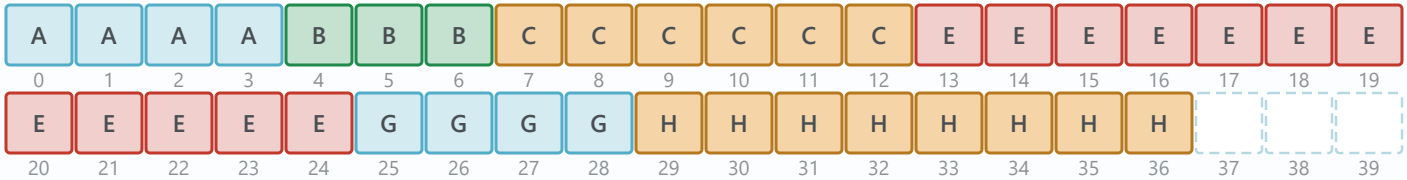
Artılar

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.

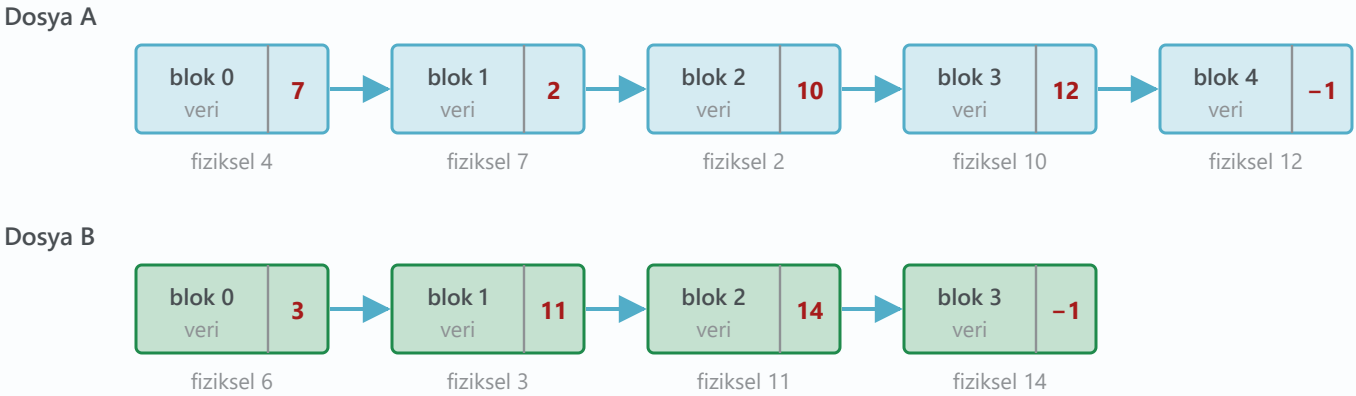
Artılar

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



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

Adım 1 / 5

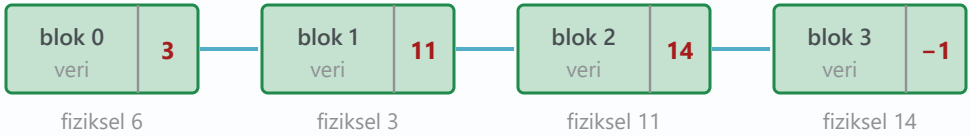
- Dış parçalanma **yoktur**, dosya kolayca büyür; iç parçalanma (son blok) her yöntemde vardır.
- Bloktaki veri miktarı ikinin kuvveti değildir (4096 − 4 bayt); tek bir işaretçi bozulursa zincirin geri kalanı kaybolur.

Bağlı Liste ile Tahsis

Dosya A



Dosya B



okunan: 1 blok

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

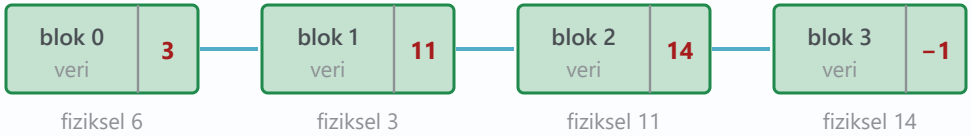
- Dış parçalanma **yoktur**, dosya kolayca büyür; iç parçalanma (son blok) her yöntemde vardır.
- Bloktaki veri miktarı ikinin kuvveti değildir (4096 − 4 bayt); tek bir işaretçi bozulursa zincirin geri kalanı kaybolur.

Bağlı Liste ile Tahsis

Dosya A



Dosya B

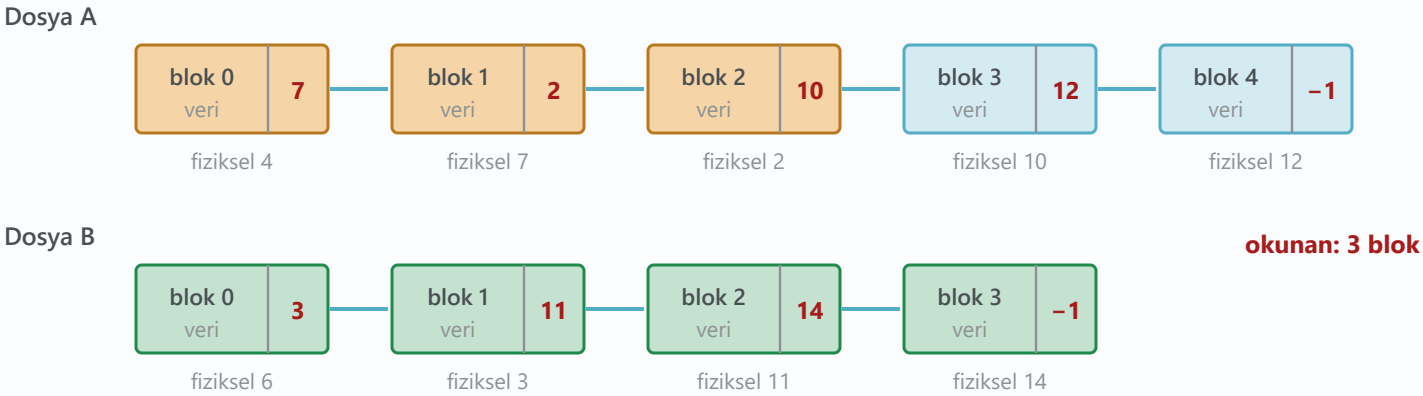


okunan: 2 blok

Fiziksel 7 okunur → sonraki 2.

- 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.

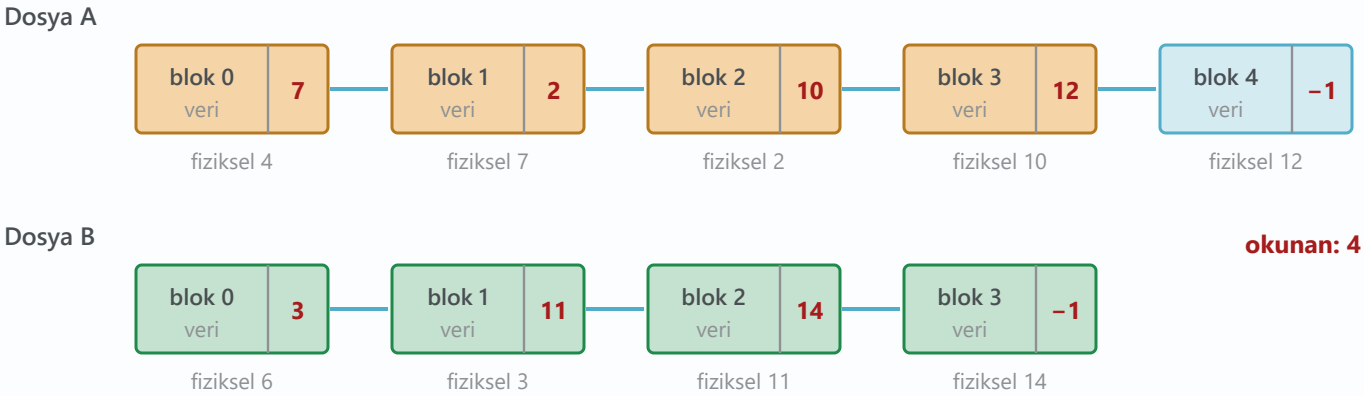
Bağlı Liste ile Tahsis



Fiziksel 2 okunur → sonraki 10.

- 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.

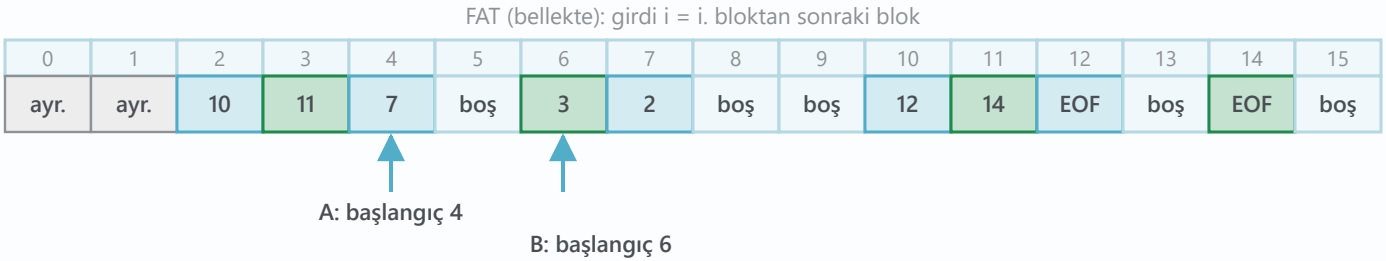
Bağlı Liste ile Tahsis



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

- 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



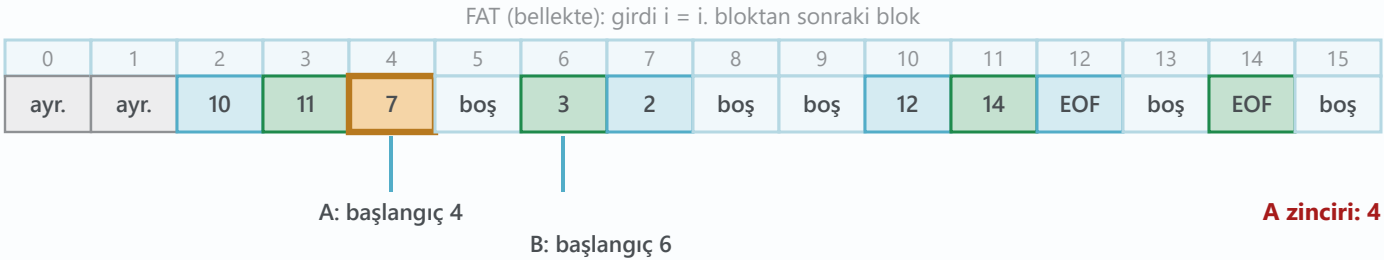
A: 4 → 7 → 2 → 10 → 12; B: 6 → 3 → 11 → 14.

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 × 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

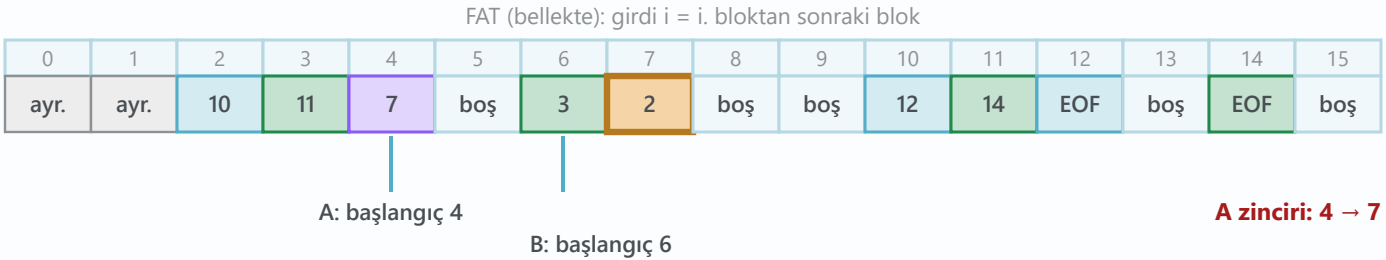


Girdi 4 = 7.

- 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 × 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

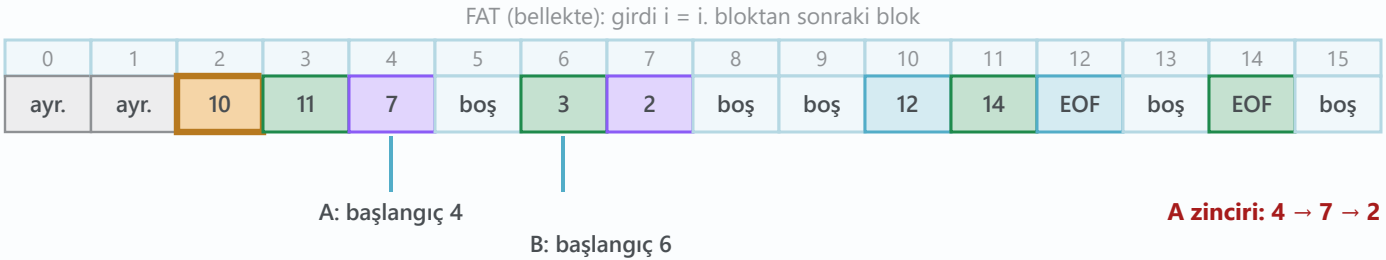


Girdi 7 = 2.

- 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 × 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



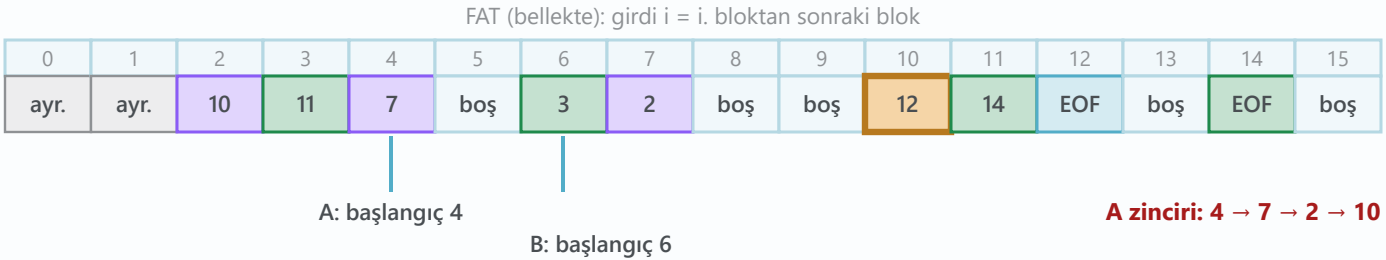
Girdi 2 = 10.

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 × 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



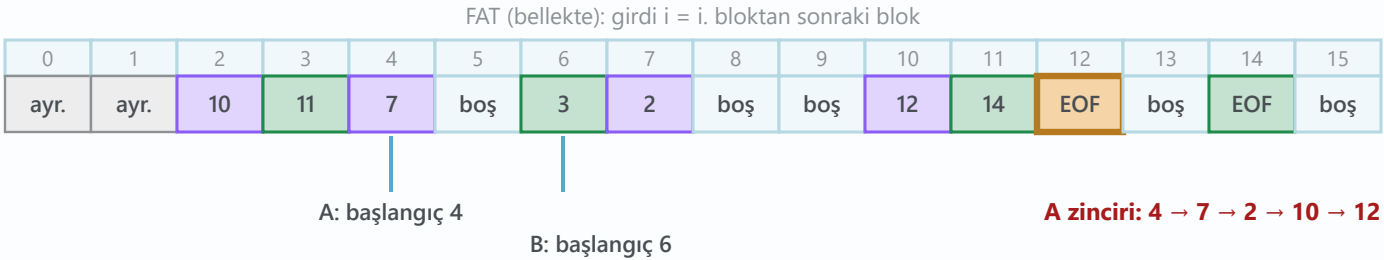
Girdi 10 = 12.

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 × 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



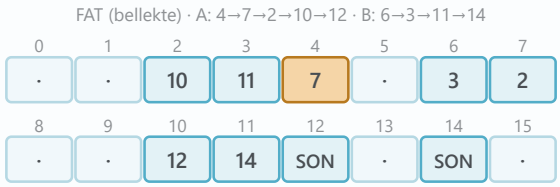
Girdi 12 = EOF. Zincir **bellekte** izlendi: disk okuması yok.

- 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 × 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 }
```



`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

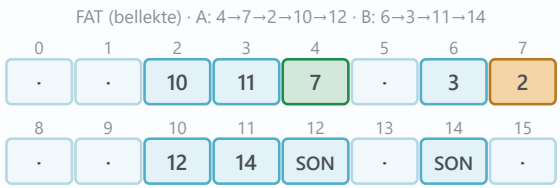
DEĞİŞKENLER

`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

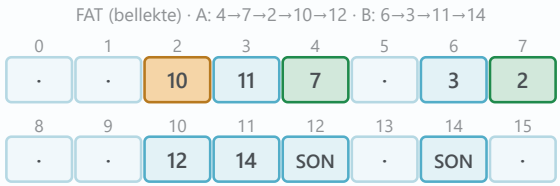
DEĞİŞKENLER

`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

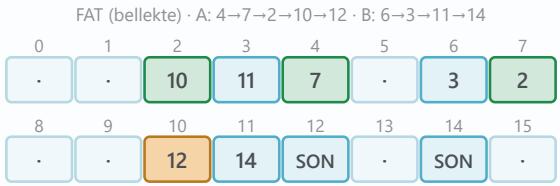
DEĞİŞKENLER

`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 */
▶   while (n > 0 && b != SON) {
▶     b = fat[b];          /* zinciri izle */
▶     n--;
7   }
8   return b;          /* SON: dosya dışı */
9 }
```



izlenen: 4 → 7 → 2 → 10

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

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

Adım 4 / 11

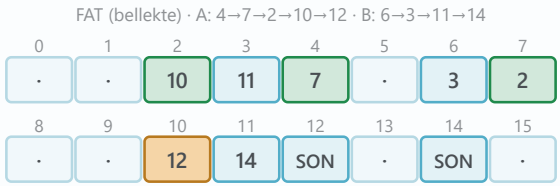
DEĞİŞKENLER

`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   }
▶   return b;            /* SON: dosya dışı */
9 }
```



izlenen: 4 → 7 → 2 → 10

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

n = 0: döngü biter, disk bloğu **10**; bayt 3500 bu bloğun 3500 − 3·1024 = 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 }
```



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

Adım 6 / 11

DEĞİŞKENLER

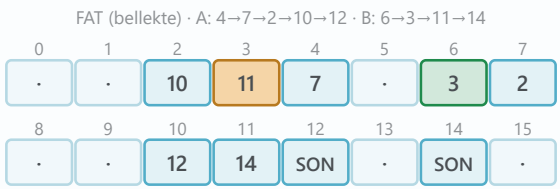
`ilk = 6` `ofset = 5000` `n = 4` `b = 6`

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 */
▶   while (n > 0 && b != SON) {
▶     b = fat[b];          /* zinciri izle */
▶     n--;
7   }
8   return b;          /* SON: dosya dışı */
9 }
```



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

Adım 7 / 11

DEĞİŞKENLER

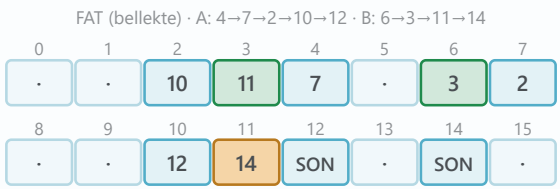
`ilk = 6` `ofset = 5000` `n = 3` `b = 3`

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

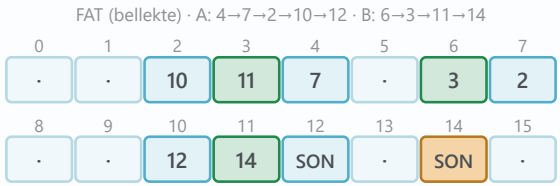
`ilk = 6` `ofset = 5000` `n = 2` `b = 11`

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 }
```



izlenen: 6 → 3 → 11 → 14

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

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

Adım 9 / 11

DEĞİŞKENLER

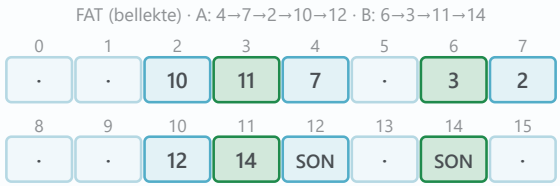
`ilk = 6` `ofset = 5000` `n = 1` `b = 14`

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

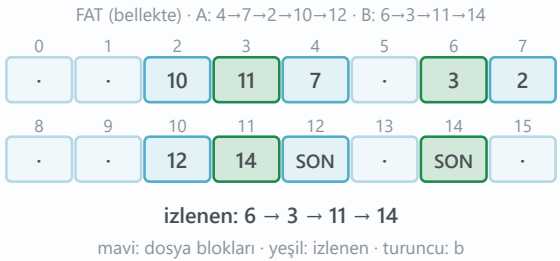
`ilk = 6` `ofset = 5000` `n = 0` `b = SON`

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   }
▶   return b;            /* SON: dosya dışı */
9 }
```



izlenen: 6 → 3 → 11 → 14

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

Dördüncü adımda b = SON: dosyanın 4. bloğu yok, bağıl konum 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`

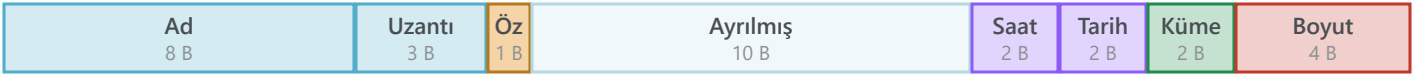
ÇIKTI

```
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)



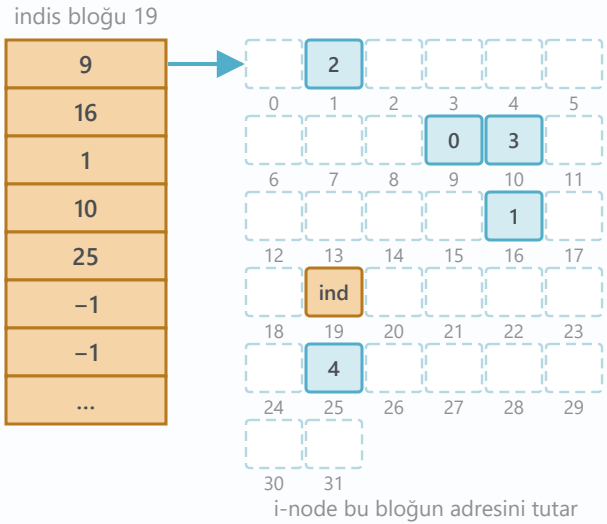
Öz: öznitelik. FAT32: "ayrılmış" alana oluşturma/erişim zamanı ve ilk kümenin üst 16 biti yerleşti; uzun adlar ek girdilerde.

	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	**4 GiB − 1 bayt**	16 EiB (kuramsal)
Boş alan	FAT'ta 0	FAT'ta 0	FAT'ta 0 + FSInfo ipucu	ayrı **biteşlem**
Kullanım	disket	eski diskler	USB, UEFI ESP	SDXC/SDUC, büyük USB

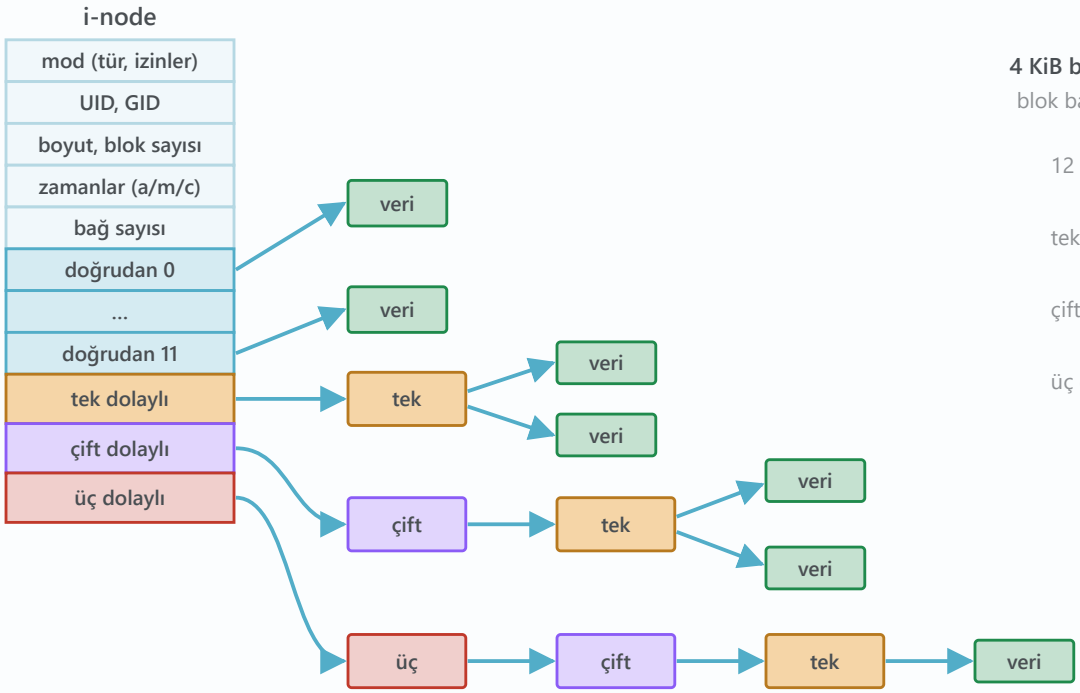
Boyut alanı 32 bit olduğundan FAT32'de 4 GiB'lık dosya sınırı vardır; exFAT bunu kaldırır ve bitişik dosyalarda FAT zincirini hiç kullanmayabilir. Windows'un grafik biçimlendirme aracı FAT32'yi tarihsel olarak 32 GB ile sınırlar; bu dosya sisteminin değil, aracın sınırıdır.

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 × 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).



i-node Yapısı: Dolaylı Bloklar



4 KiB blok, 4 B işaretçi:
blok başına 1024 adres

12 doğrudan	**48 KiB**
tek dolaylı	**+ 4 MiB**
çift dolaylı	**+ 4 GiB**
üç dolaylı	**+ 4 TiB**

her düzey ×1024

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 = **1024** adres tutar; i-node'da **12 doğrudan**, 1 tek, 1 çift, 1 üç dolaylı işaretçi.

Doğrudan: 12 × 4 KiB = **48 KiB** (49.152 B) · Tek dolaylı: 1024 × 4 KiB = **4 MiB**

İlk 12 blok i-node'dan doğrudan; sonraki 1024 blok tek bir indis bloğundan.

Tanenbaum örneği (1 KiB blok, 256 adres, 10 doğrudan): 16.843.018 blok ≈ 16 GiB. Gerçekte ext2/ext3'te 4 KiB blokla sınır **2 TiB**'dir (blok sayacı 32 bit × 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 = **1024** adres tutar; i-node'da **12 doğrudan**, 1 tek, 1 çift, 1 üç dolaylı işaretçi.

Doğrudan: 12 × 4 KiB = **48 KiB** (49.152 B) · Tek dolaylı: 1024 × 4 KiB = **4 MiB**

İlk 12 blok i-node'dan doğrudan; sonraki 1024 blok tek bir indis bloğundan.

Çift dolaylı: 1024^2 × 4 KiB = 1.048.576 blok = **4 GiB**

1024 indis bloğunun her biri 1024 veri bloğu gösterir.

Tanenbaum örneği (1 KiB blok, 256 adres, 10 doğrudan): 16.843.018 blok ≈ 16 GiB. Gerçekte ext2/ext3'te 4 KiB blokla sınır **2 TiB**'dir (blok sayacı 32 bit × 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 = **1024** adres tutar; i-node'da **12 doğrudan**, 1 tek, 1 çift, 1 üç dolaylı işaretçi.

Doğrudan: 12 × 4 KiB = **48 KiB** (49.152 B) · Tek dolaylı: 1024 × 4 KiB = **4 MiB**

İlk 12 blok i-node'dan doğrudan; sonraki 1024 blok tek bir indis bloğundan.

Çift dolaylı: 1024^2 × 4 KiB = 1.048.576 blok = **4 GiB**

1024 indis bloğunun her biri 1024 veri bloğu gösterir.

Üç dolaylı: 1024^3 × 4 KiB = 1.073.741.824 blok = **4 TiB**

Bir düzey daha: ×1024.

Adım 3 / 4

Tanenbaum örneği (1 KiB blok, 256 adres, 10 doğrudan): 16.843.018 blok ≈ 16 GiB. Gerçekte ext2/ext3'te 4 KiB blokla sınır **2 TiB**'dir (blok sayacı 32 bit × 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 = **1024** adres tutar; i-node'da **12 doğrudan**, 1 tek, 1 çift, 1 üç dolaylı işaretçi.

Doğrudan: 12 × 4 KiB = **48 KiB** (49.152 B) · Tek dolaylı: 1024 × 4 KiB = **4 MiB**

İlk 12 blok i-node'dan doğrudan; sonraki 1024 blok tek bir indis bloğundan.

Çift dolaylı: 1024^2 × 4 KiB = 1.048.576 blok = **4 GiB**

1024 indis bloğunun her biri 1024 veri bloğu gösterir.

Üç dolaylı: 1024^3 × 4 KiB = 1.073.741.824 blok = **4 TiB**

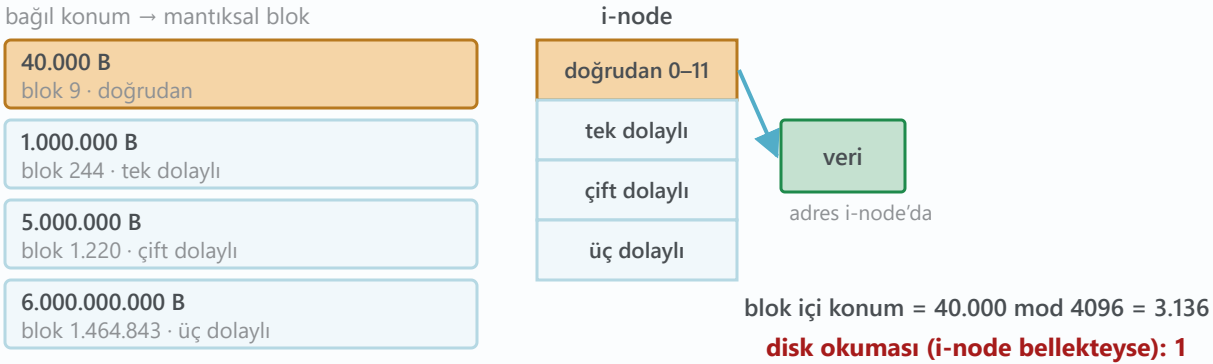
Bir düzey daha: ×1024.

Toplam: 12 + 1024 + 1024^2 + 1024^3 = **1.074.791.436** blok × 4096 = **4.402.345.721.856 B ≈ 4,004 TiB**

Neredeyse tamamı üç dolaylı bloktan gelir.

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

Bağıl Konumdan Disk Bloğuna

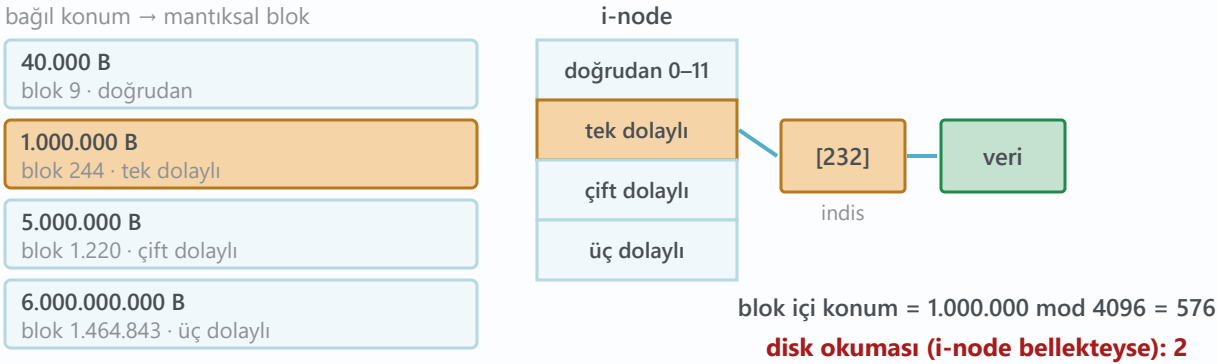


blok içi konum = 40.000 mod 4096 = 3.136

disk okuması (i-node bellekteyse): 1

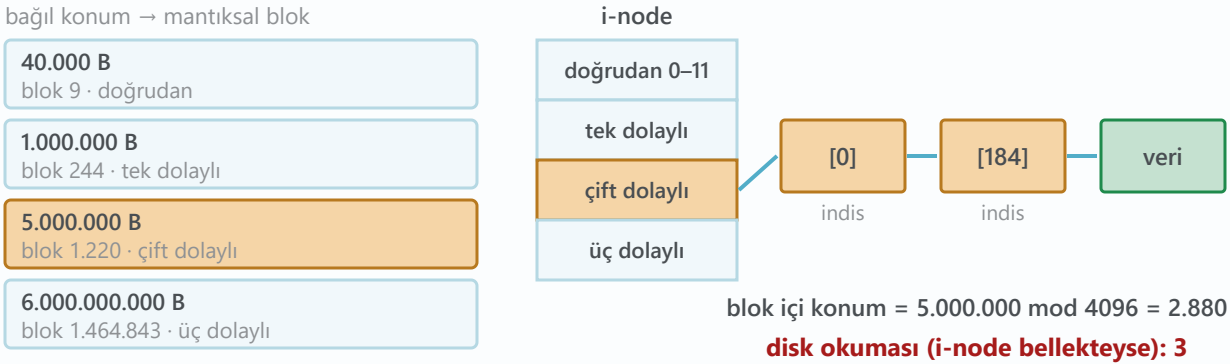
40.000 / 4096 = **9** → 12'den küçük: doğrudan işaretçi 9. Tek okuma.

Bağıl Konumdan Disk Bloğuna



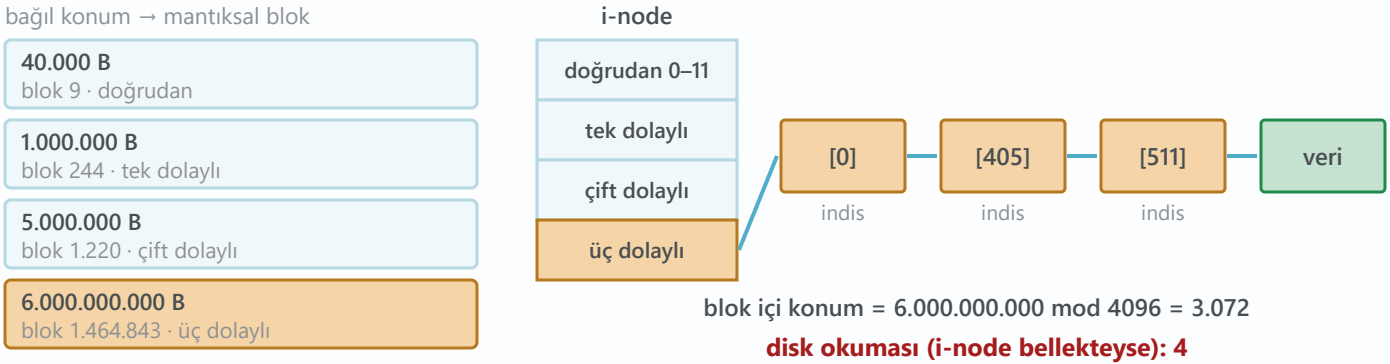
1.000.000 / 4096 = 244; 244 − 12 = **232** < 1024 → tek dolaylı bloğun 232. girdisi. 2 okuma.

Bağıl Konumdan Disk Bloğuna



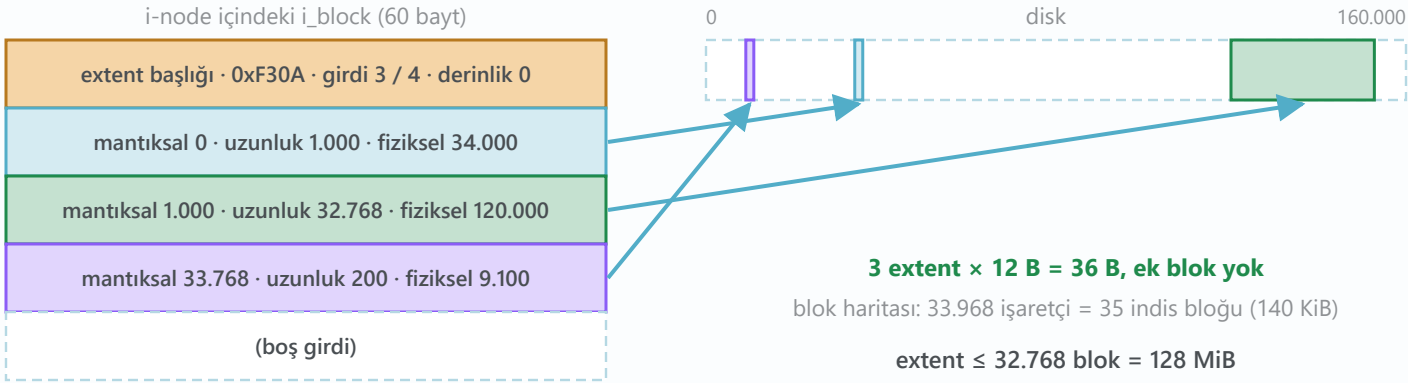
5.000.000 / 4096 = 1220; 1220 − 12 − 1024 = 184 < 1024^2 → çift dolaylı: dış indis 184 div 1024 = **0**, iç indis **184**. 3 okuma.

Bağıl Konumdan Disk Bloğuna



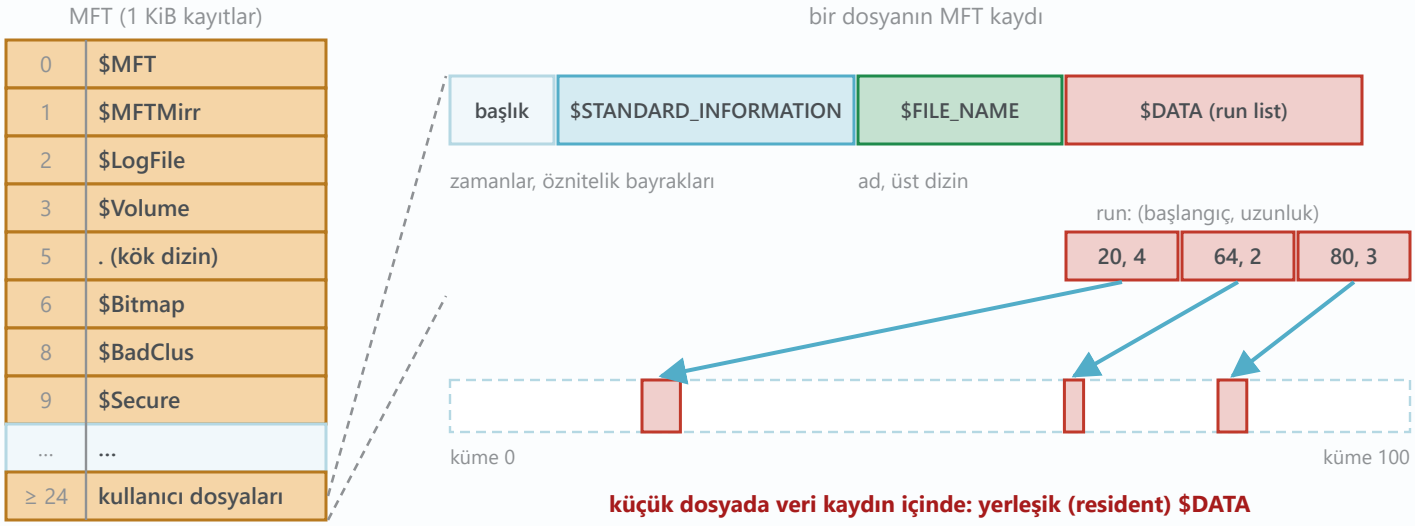
6 GB / 4096 = 1.464.843; 12 + 1024 + 1024^2 çıkarılınca 415.231 → üç dolaylı: **0, 405, 511**. 4 okuma; önbellek ilk düzeyleri tutar.

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.

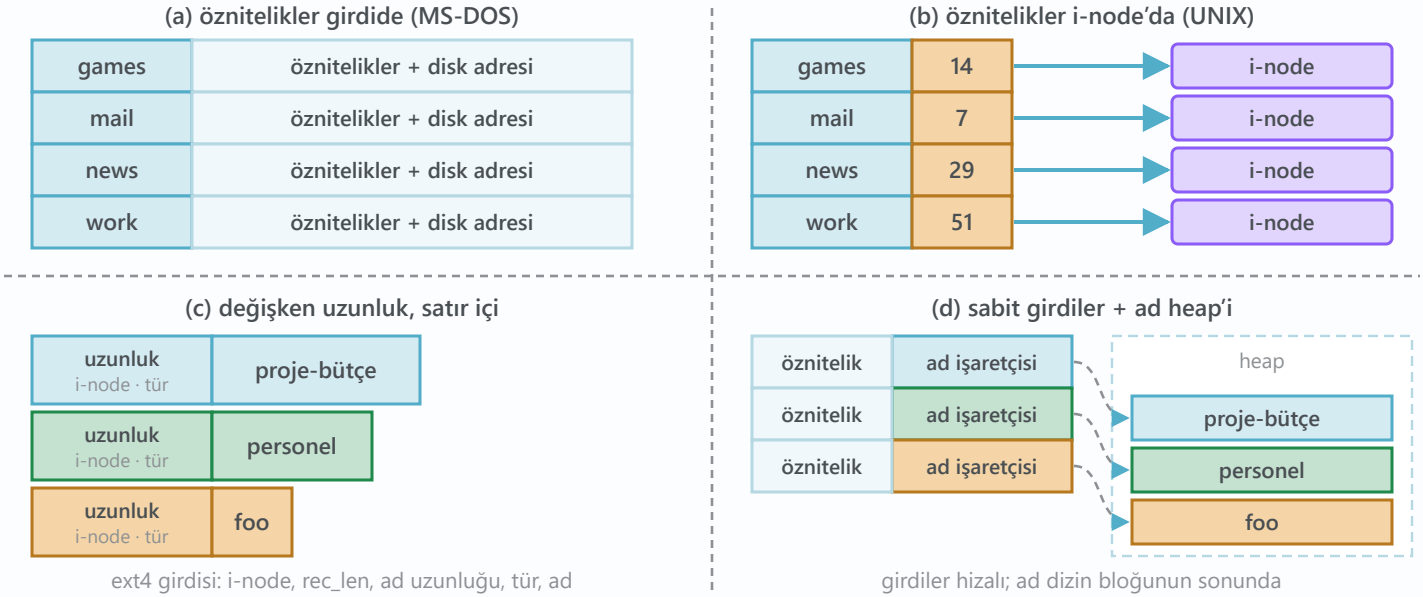
TAHSİS YÖNTEMLERİ

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

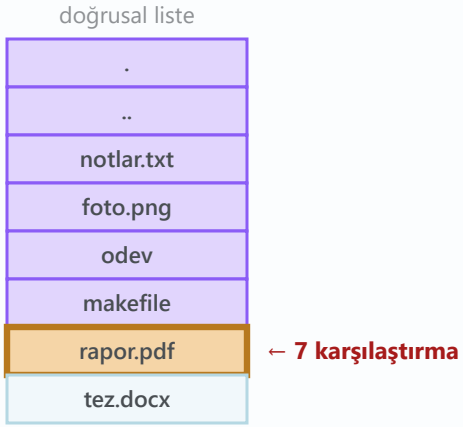


ext4 girdisi: i-node, rec_len, ad uzunluğu, tür, ad

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) heap'i yönetmeyi gerektirir.

Dizin Arama: Liste, Hash, Ağaç

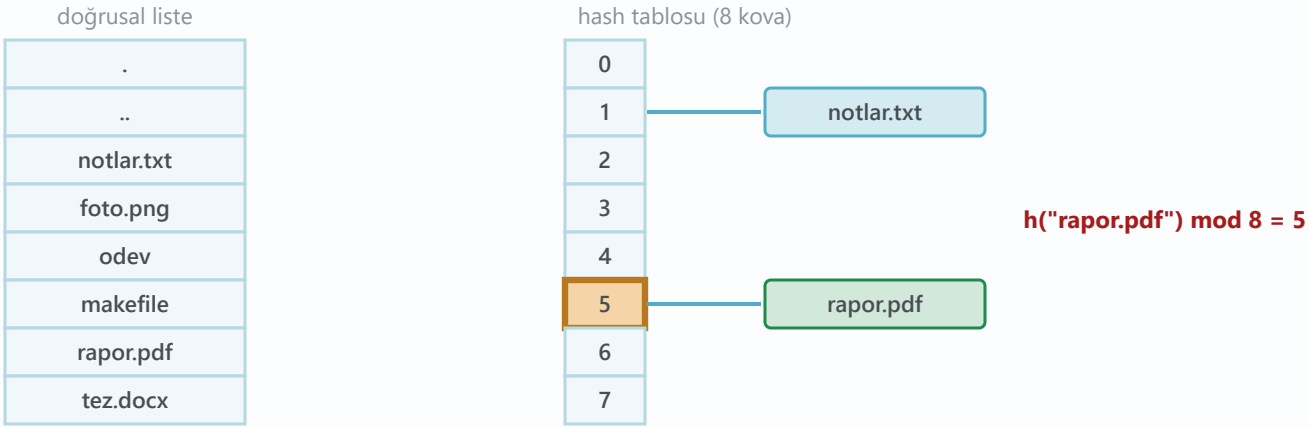


Doğrusal liste: basit, ama arama O(n). Ekleme için önce adın olmadığı denetlenir; silmede girdi boş işaretlenir ya da son girdi taşınır.

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 önbelleğinde** tutar.

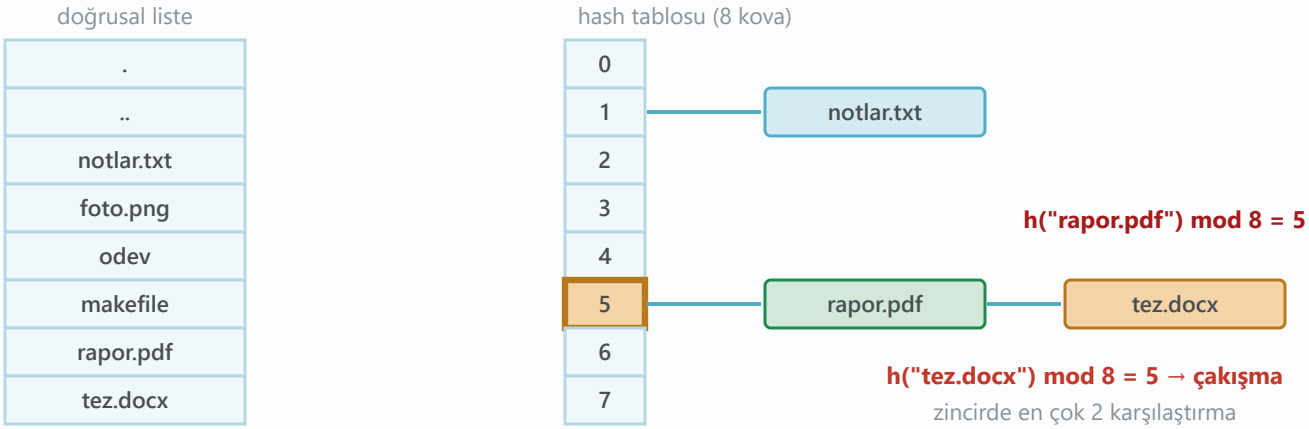
Dizin Arama: Liste, Hash, Ağaç



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

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 önbelleğinde** tutar.

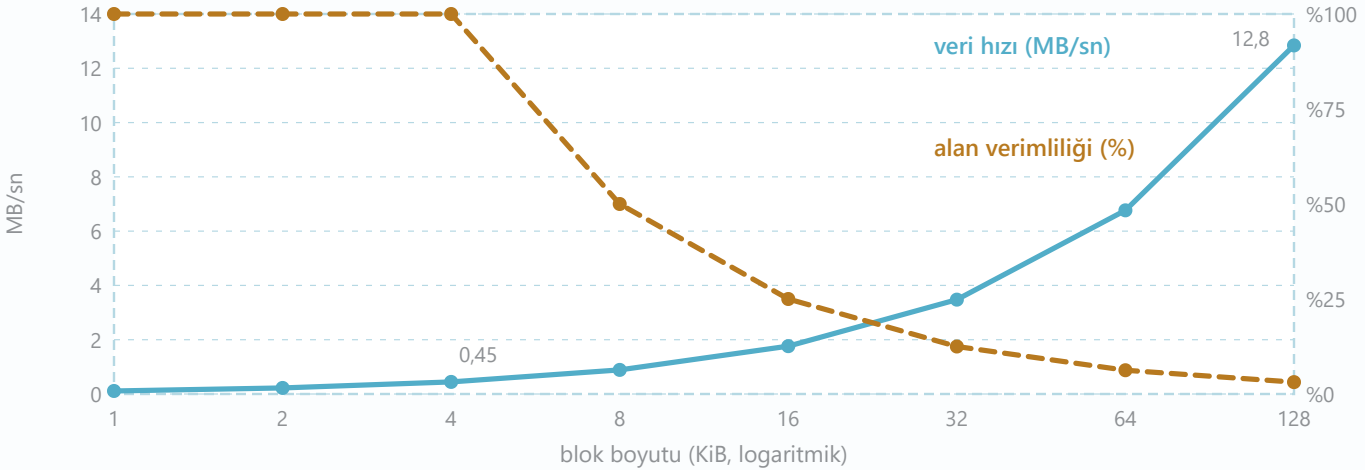
Dizin Arama: Liste, Hash, Ağaç



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

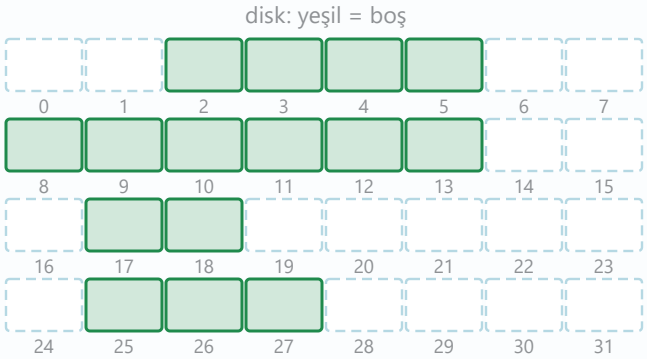
ext4 büyük dizinlerde ad özetine göre sıralı *htree*, NTFS, XFS ve Btrfs **B+ ağacı** kullanır: sıralı listeleme ve O(log n) arama birlikte. Linux en son kullanılan ad → i-node eşlemelerini ayrıca **dentry önbelleğinde** tutar.

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 MB) × 8,33 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



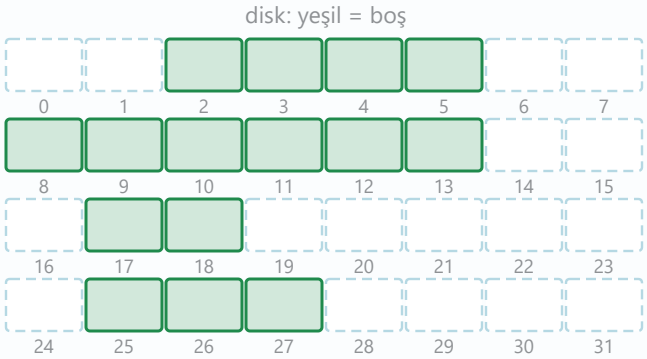
boş bloklar:

2–5, 8–13, 17–18, 25–27

15 boş blok / 32

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

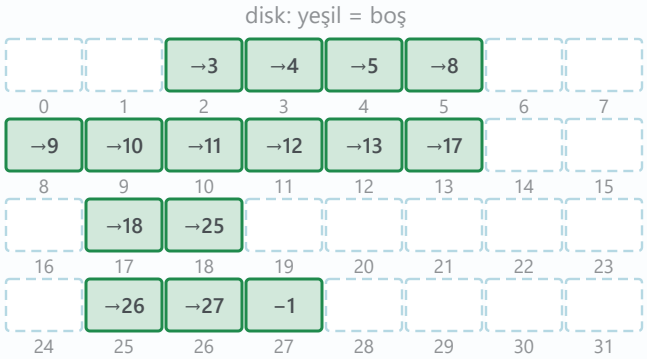
Boş Alan Yönetimi: Dört Yöntem





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

Boş Alan Yönetimi: Dört Yöntem



baş işaretçisi → 2

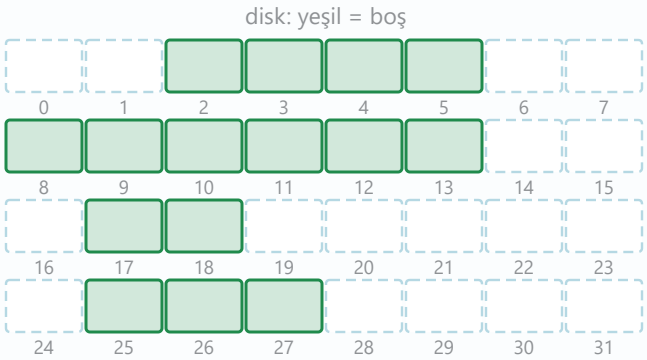
her boş blok bir sonrakini gösterir

ek alan gerekmez (bilgi boş blokların içinde)

ama listede gezinmek = her blok için G/Ç

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

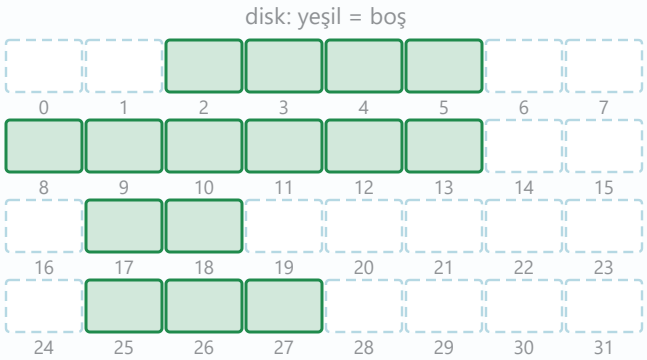
Boş Alan Yönetimi: Dört Yöntem

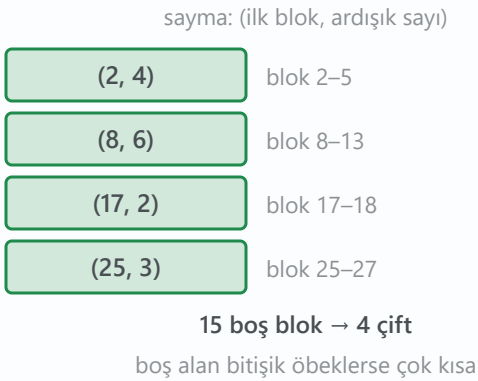




Gruplama: ilk boş blok sonraki n−1 boş bloğun adresini ve bir sonraki grup bloğunu tutar (UNIX'in boş liste blokları).

Boş Alan Yönetimi: Dört Yöntem





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

Biteşlemle 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 }
```

biteşlem: 1 = dolu, 0 = boş

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

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

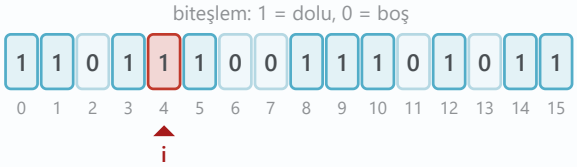
DEĞİŞKENLER

`ipucu = 4` `k = –` `i = –`

ÇIKTI

Biteşlemle Blok Ayırma: Kod İzleme

```
1  int blok_ayir(int ipucu) {
2    for (int k = 0; k < N; k++) {
▶      int i = (ipucu + k) % N;
▶      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

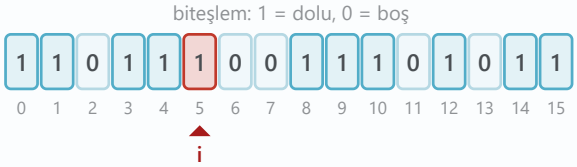
DEĞİŞKENLER

`ipucu = 4` `k = 0` `i = 4`

ÇIKTI

Biteşlemle Blok Ayırma: Kod İzleme

```
1  int blok_ayir(int ipucu) {
2    for (int k = 0; k < N; k++) {
▶      int i = (ipucu + k) % N;
▶      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) mod 16 = 5; bit 1, blok dolu. Sonraki konuma geçilir.

Adım 3 / 9

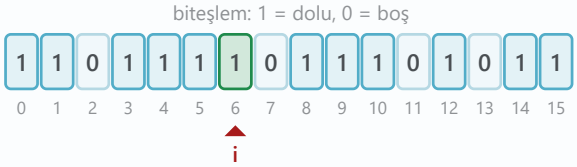
DEĞİŞKENLER

`ipucu = 4` `k = 1` `i = 5`

ÇIKTI

Biteşlemle Blok Ayırma: Kod İzleme

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



dosyanın blokları: 6

yeşil: bu dosyaya ayrılan

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

Adım 4 / 9

DEĞİŞKENLER

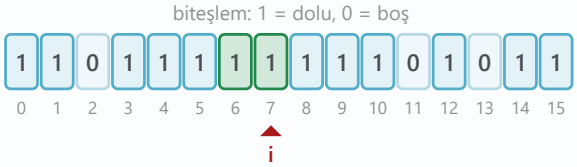
`ipucu = 4` `k = 2` `i = 6`

ÇIKTI

```
6
```

Biteşlemle Blok Ayırma: Kod İzleme

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



dosyanın blokları: 6, 7

yeşil: bu dosyaya ayrılan

blok_ayir(7): i = 7, bitesl[7] = 0: **boş**. Bit 1 yapılır, blok 7 döner. Ardışık bloklar sıralı okumada kafa hareketini azaltır.

Adım 5 / 9

DEĞİŞKENLER

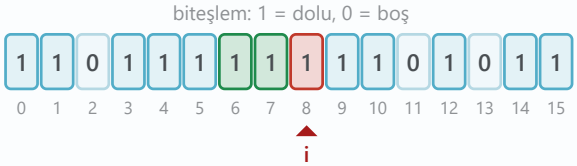
`ipucu = 7` `k = 0` `i = 7`

ÇIKTI

```
6 7
```

Biteşlemle Blok Ayırma: Kod İzleme

```
1  int blok_ayir(int ipucu) {
2    for (int k = 0; k < N; k++) {
▶      int i = (ipucu + k) % N;
▶      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

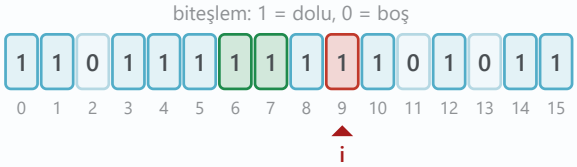
`ipucu = 8` `k = 0` `i = 8`

ÇIKTI

```
6 7
```

Biteşlemle Blok Ayırma: Kod İzleme

```
1  int blok_ayir(int ipucu) {
2    for (int k = 0; k < N; k++) {
▶      int i = (ipucu + k) % N;
▶      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
```

Biteşlemle Blok Ayırma: Kod İzleme

```
1  int blok_ayir(int ipucu) {
2    for (int k = 0; k < N; k++) {
▶      int i = (ipucu + k) % N;
▶      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) mod 16 = 10; bit 1, blok dolu. Sonraki konuma geçilir.

Adım 8 / 9

DEĞİŞKENLER

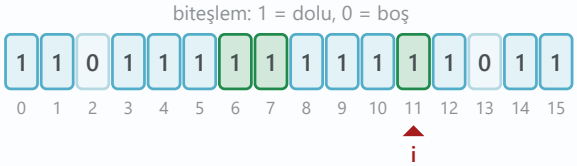
`ipucu = 8` `k = 2` `i = 10`

ÇIKTI

```
6 7
```

Biteşlemle Blok Ayırma: Kod İzleme

```
1  int blok_ayir(int ipucu) {
2    for (int k = 0; k < N; k++) {
▶      int i = (ipucu + k) % N;
▶      if (bitesl[i] == 0) {  /* boş mu? */
▶        bitesl[i] = 1;
▶        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`

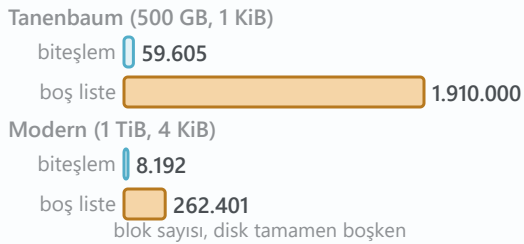
ÇIKTI

```
6 7 11
```

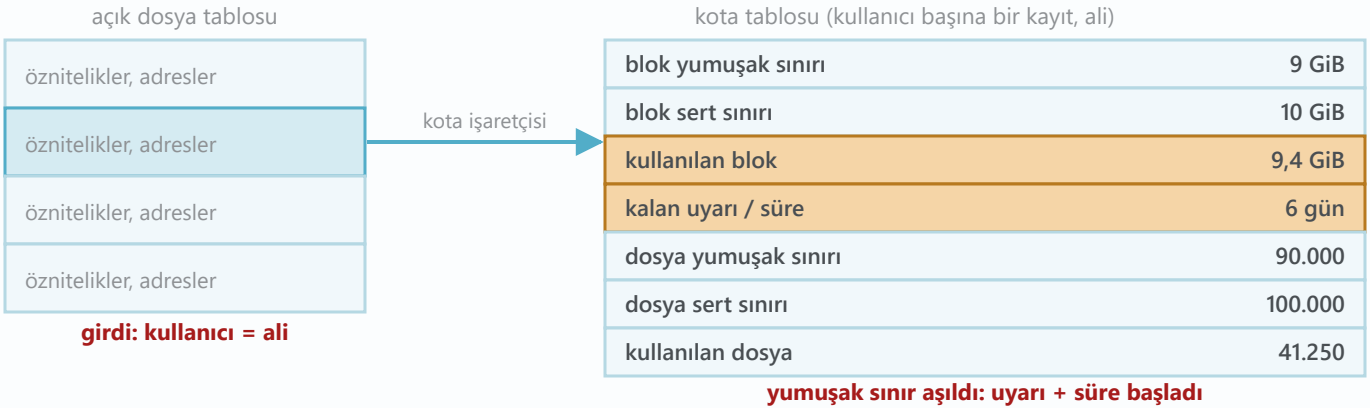
Boş Alan: Sayısal Karşılaştırma

	Tanenbaum örneği	Modern örnek
Disk / blok	500 GB, 1 KiB blok	1 TiB, 4 KiB blok
Blok sayısı	$500 \times 10^9 / 1024 \approx$ **488,3 milyon**	$2^{40} / 2^{12} =$ **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 ≈ **1,91 milyon** blok	268,4 M / 1023 ≈ **262.401** blok (≈ 1 GiB)
Biteşlem	488,3 M bit / 8192 ≈ **59.605** blok (≈ 60.000)	2^{28} bit = 32 MiB = **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.



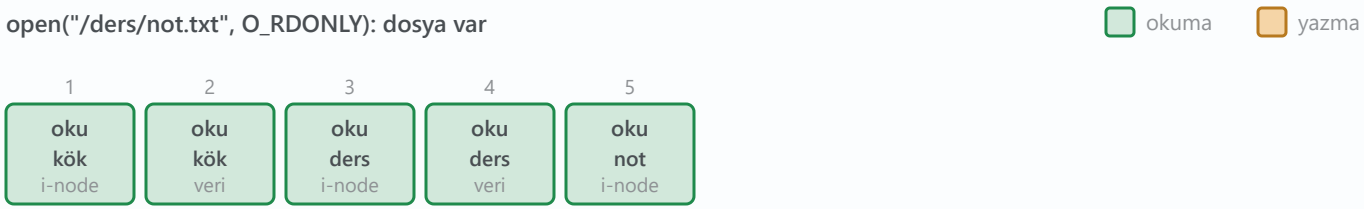
Disk Kotaları



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

PERFORMANS · ADIM ADIM

Bir Dosyayı Oluşturmak Kaç G/Ç'dir?



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?

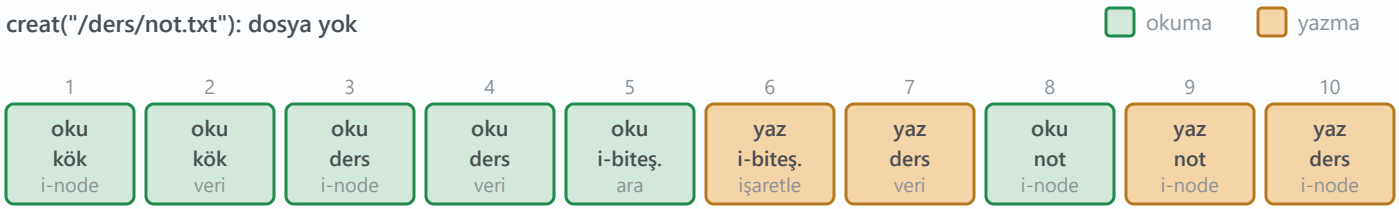


3 G/Ç = 2 okuma + 1 yazma

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

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?

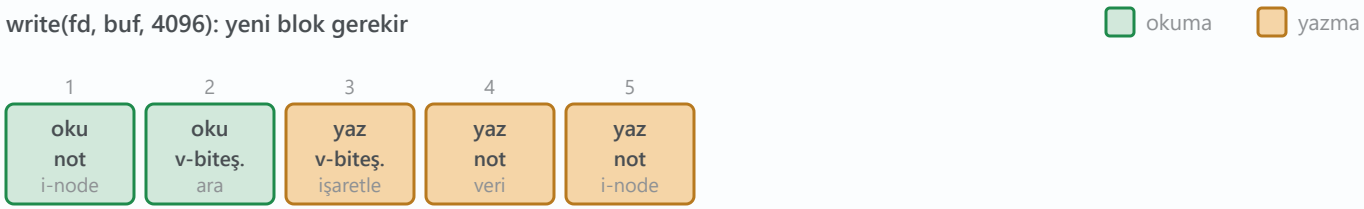


10 G/Ç = 6 okuma + 4 yazma

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**.

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?



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.

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 · ADIM ADIM

Bir Dosyayı Oluşturmak Kaç G/Ç'dir?



kalın 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

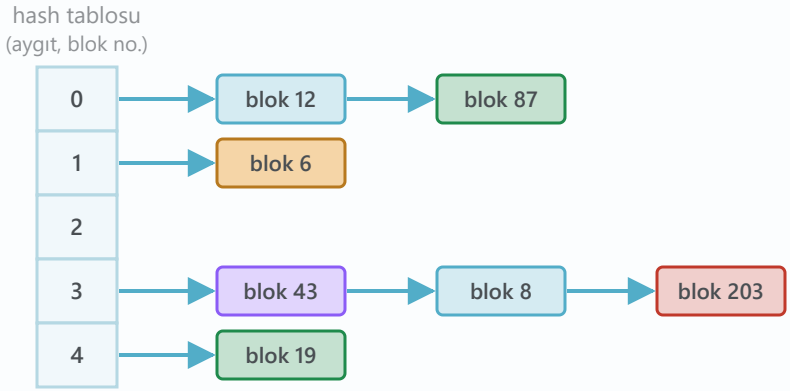
→ 8 yazma kalır: i-biteşlem, v-biteşlem, ders veri, ders i-node, not i-node, 3 veri bloğu

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.

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)



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

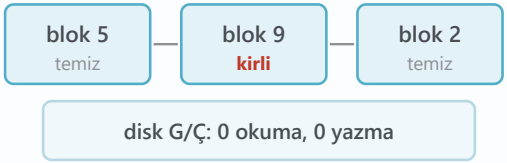
Erişim	Gecikme (2026, tipik)
DRAM'den 64 bayt	≈ 80–120 ns
NVMe SSD 4 KiB rastgele okuma	≈ 50–100 µs
HDD arama + dönüş	≈ 5–10 ms (100–280 MB/sn sıralı)

- **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

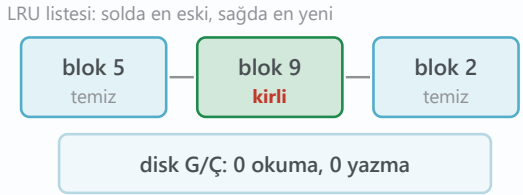
DEĞİŞKENLER

`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 */
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 }
```



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

Adım 2 / 10

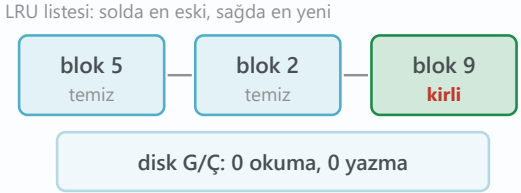
DEĞİŞKENLER

`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 */
▶      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 }
```



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`

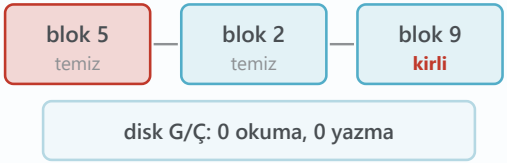
ÇIKTI

```
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   diskten_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

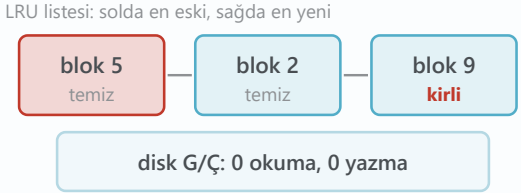
`blok = 7` `t = blok 5` `okuma = 0` `yazma = 0`

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 */
▶    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 }
```



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

Adım 5 / 10

DEĞİŞKENLER

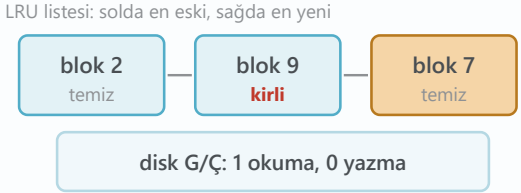
`blok = 7` `t = blok 5` `okuma = 0` `yazma = 0`

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 */
▶    t->blok = blok; t->kirli = 0;
▶    diskten_oku(t);           /* ıska: G/Ç */
▶    lru_sona(t);
13   return t;
14 }
```



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`

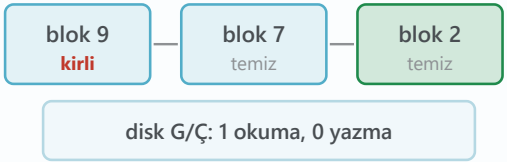
ÇIKTI

```
9: isabet
7: ıska
```

Tampon Önbelleği: 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`

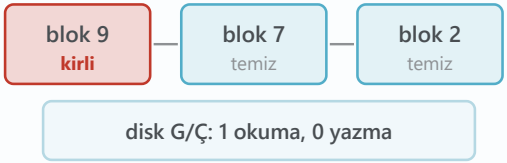
ÇIKTI

```
9: isabet
7: ıska
2: isabet
```

Tampon 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   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

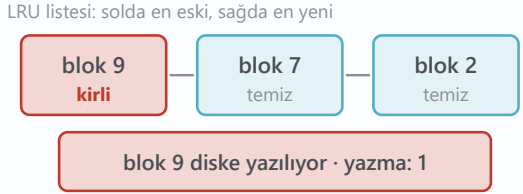
`blok = 4` `t = blok 9` `okuma = 1` `yazma = 0`

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 */
▶    if (t->kirli)
▶      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 }
```



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`

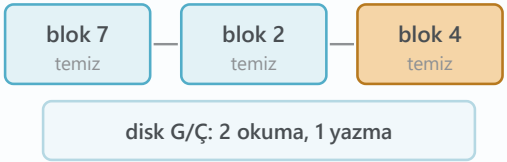
ÇIKTI

```
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

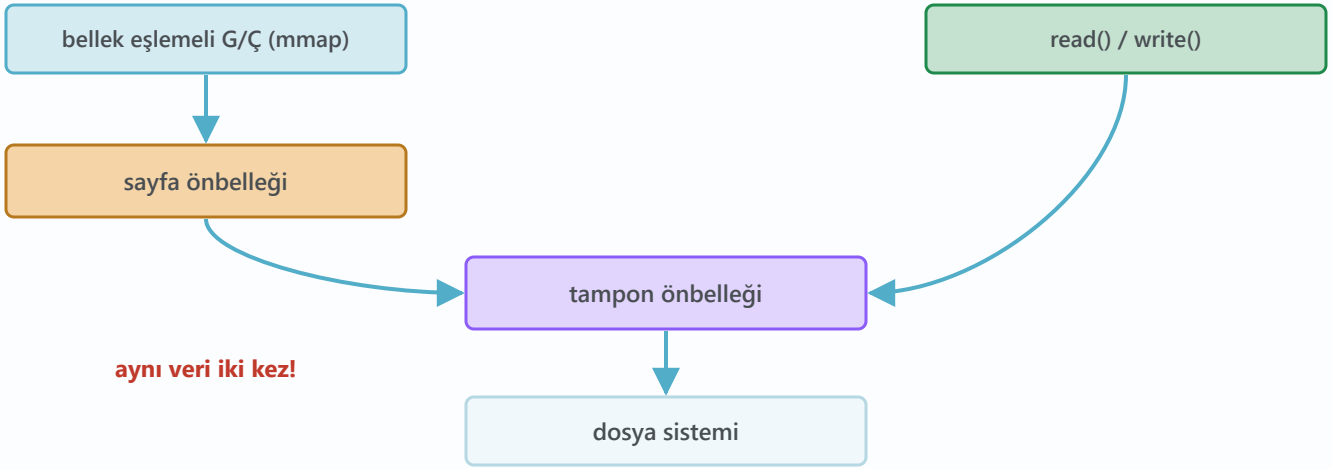
`blok = 4` `t = blok 4` `okuma = 2` `yazma = 1`

ÇIKTI

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

PERFORMANS · ADIM ADIM

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 yer değiştirme algoritmalarını paylaşır. İstisna: ZFS kendi ARC önbelleğini tutar. `O_DIRECT` veritabanlarının önbelleği atlamasını sağlar.

Sayfa Önbelleği ve Birleşik Tampon Önbelleği

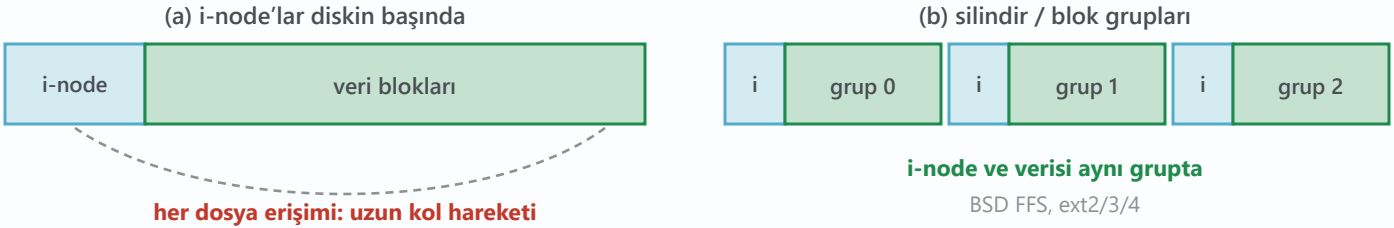


Birleşik tampon önbelleği: dosya sayfaları tek yerde; iki yol da aynı sayfaları görür. Linux 2.4'ten beri sayfa önbelleği, Windows önbellek yöneticisi, Solaris böyledir.

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

PERFORMANS

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 koymak; biteşlemle bitişik blok bulmak kolaydır. Bloklar ikişer ayrılır (eski öneri) ya da bugünkü gibi extent, gecikmeli tahsis ve çoklu blok ayırıcı kullanılır.
- **Birleştirme (defrag)**: zamanla dosyalar parçalanır. Windows "Sürücüleri 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.

PERFORMANS

SSD Dikkat Noktaları: FTL ve TRIM

- Flaş sayfası yerinde güncellenemez; SSD'deki **FTL** her yazmayı yeni bir sayfaya yönlendirir, yani SSD'nin içi günlük yapılıdır.
- Dosya sistemi bir dosyayı silince blokları yalnızca kendi biteşleminde boş işaretler. Bunu SSD'ye **TRIM** (NVMe: Deallocate) ile bildirmezse SSD o sayfaları geçerli sanar ve çöp toplamada boşuna kopyalar.
- Linux'ta bildirim ya haftalık `fstrim.timer` ile toplu ya da `discard` bağlama seçeneğiyle her silmede yapılır.

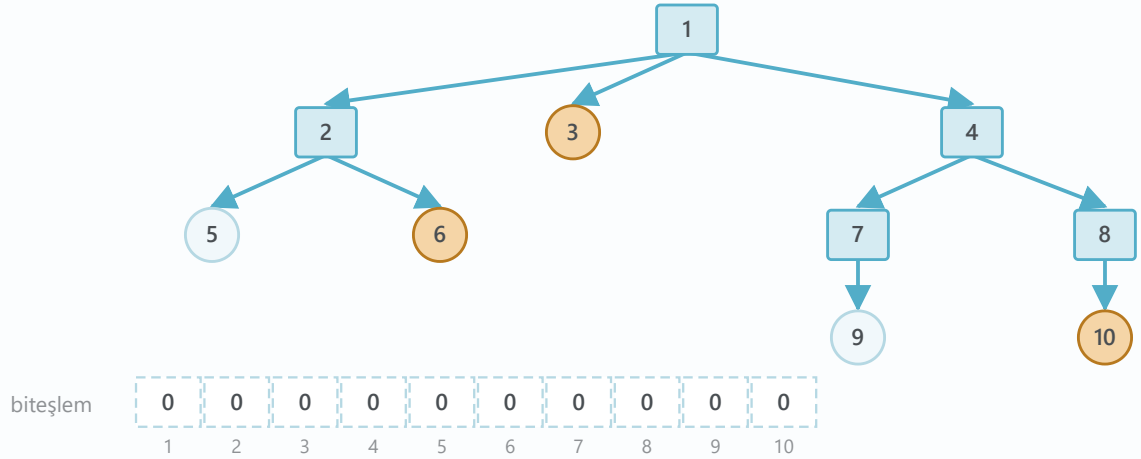
Yedekleme Stratejileri

	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



- 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 (çevrim dışı ya da değiştirilemez). Teyp bugün de arşivde: LTO-10 kartuşu sıkıştırmasız 40 TB'a kadar.

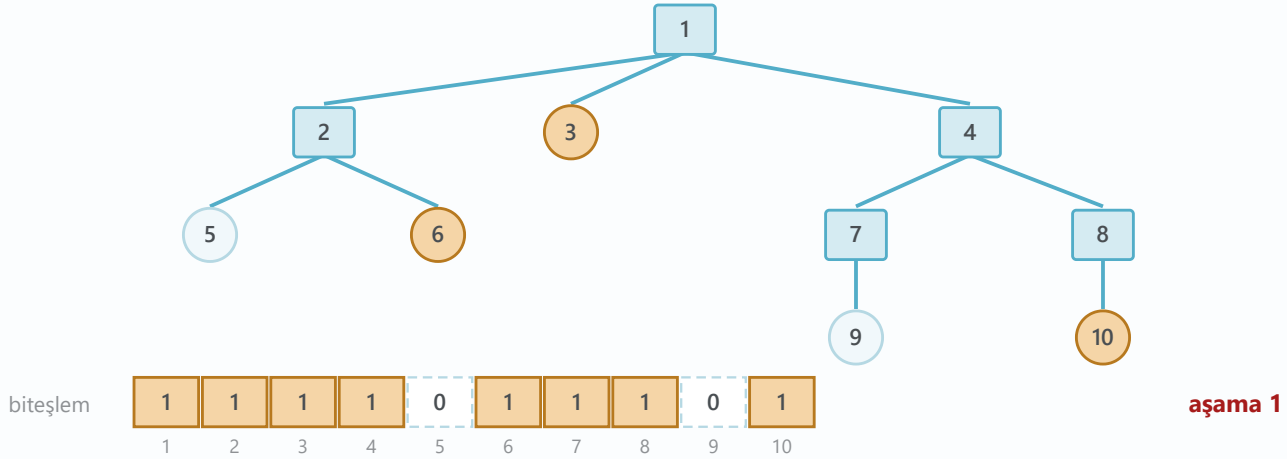
Mantıksal Döküm Algoritması



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

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.

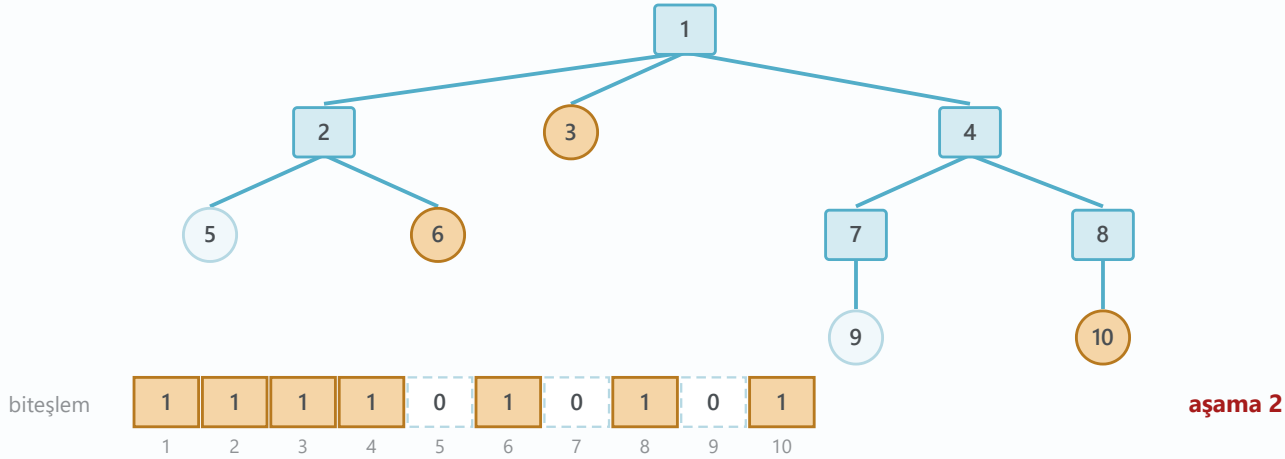
Mantıksal Döküm Algoritması



Aşama 1: ağaçta kökten başlayarak gezinilir; değişen her dosya ve **tüm dizinler** işaretlenir.

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.

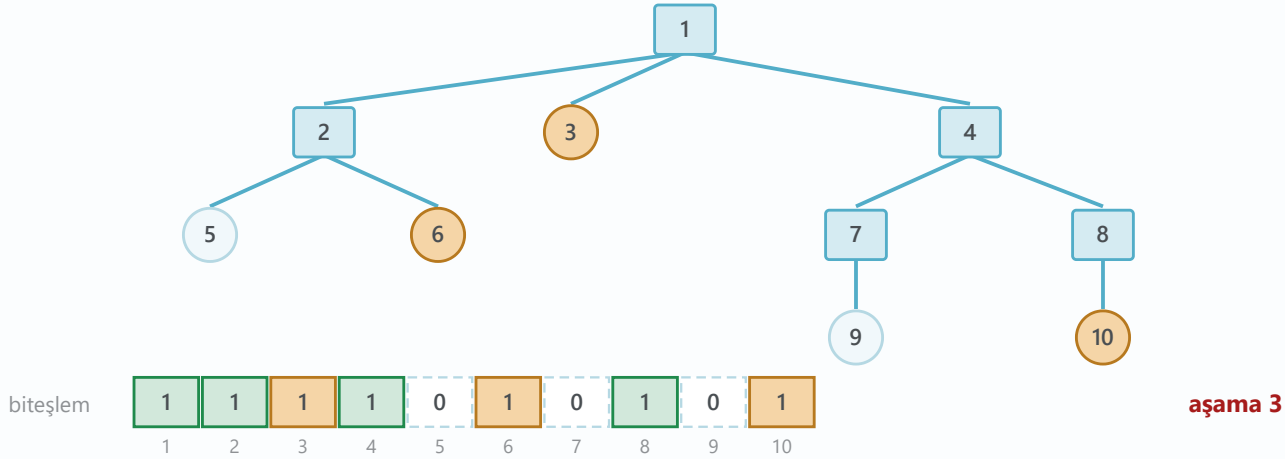
Mantıksal Döküm Algoritması



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

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.

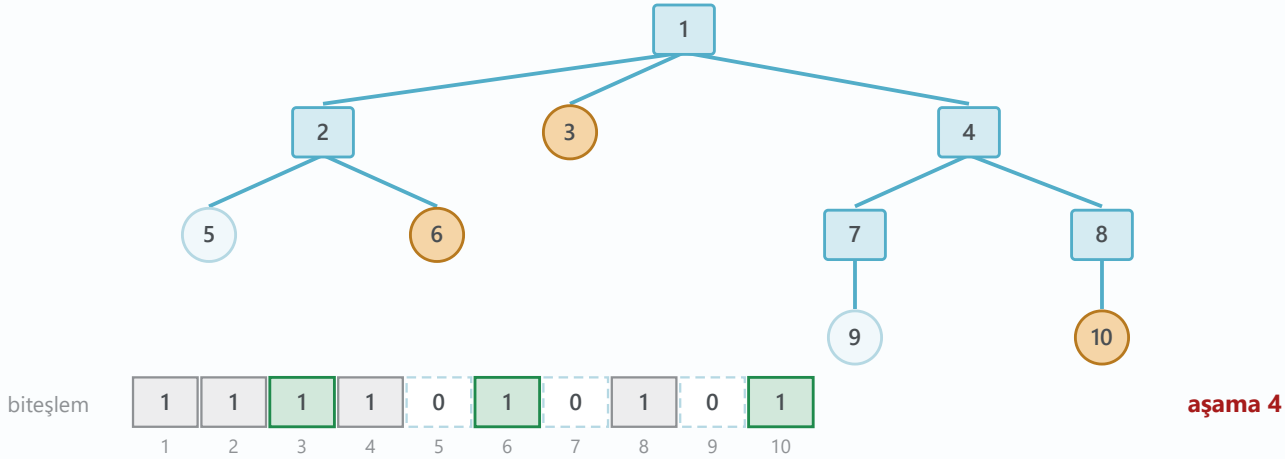
Mantıksal Döküm Algoritması



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

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.

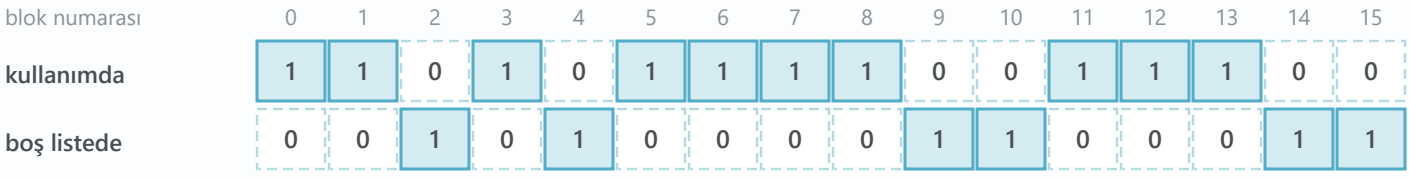
Mantıksal Döküm Algoritması



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

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



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 dizinlerde gezinilir, her i-node için görülme sayısı tutulur ve i-node'daki bağ sayısıyla 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



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

- **Dosya tutarlılığı**: kökten tüm dizinlerde gezinilir, her i-node için görülme sayısı tutulur ve i-node'daki bağ sayısıyla 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.

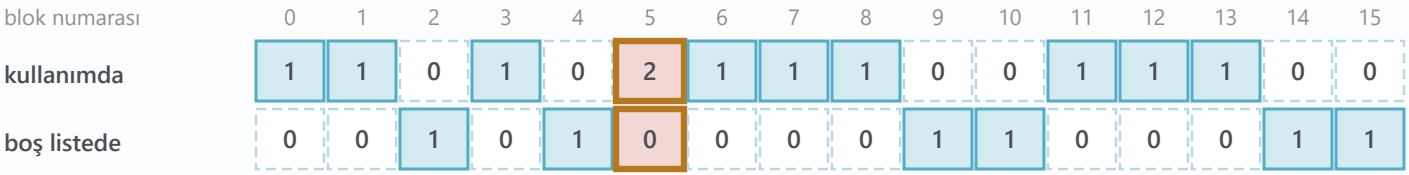
Tutarlılık Denetimi: fsck



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

- **Dosya tutarlılığı**: kökten tüm dizinlerde gezinilir, her i-node için görülme sayısı tutulur ve i-node'daki bağ sayısıyla 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

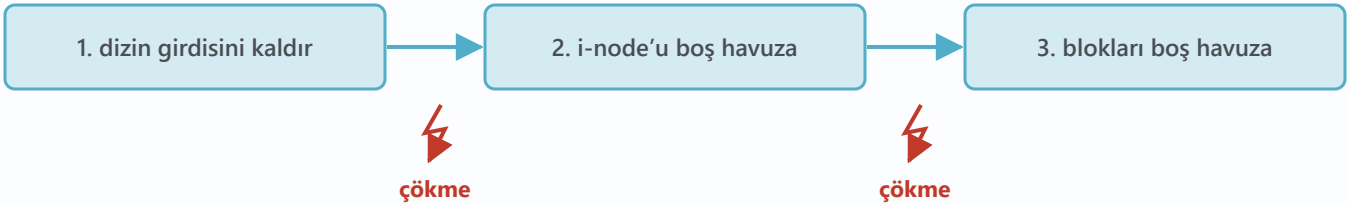


Blok 5 iki dosyada → yeni blok ayır, kopyala, kullanıcıya bildir.

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

- **Dosya tutarlılığı**: kökten tüm dizinlerde gezinilir, her i-node için görülme sayısı tutulur ve i-node'daki bağ sayısıyla 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?



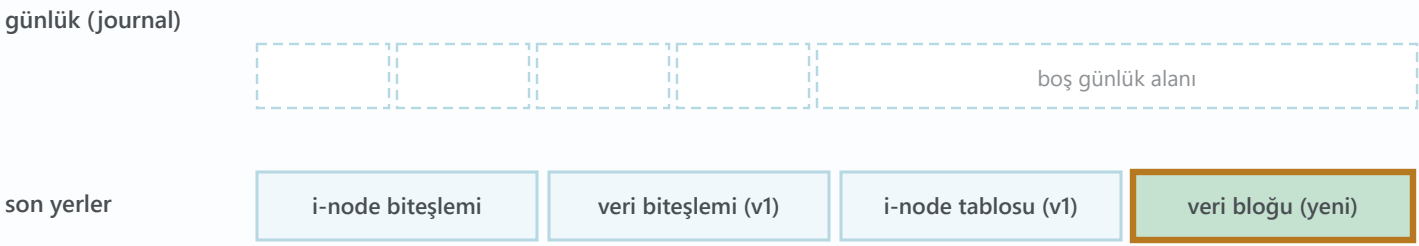
1'den sonra çökme: i-node ve bloklar sahipsiz (sızıntı, fsck bulur)

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

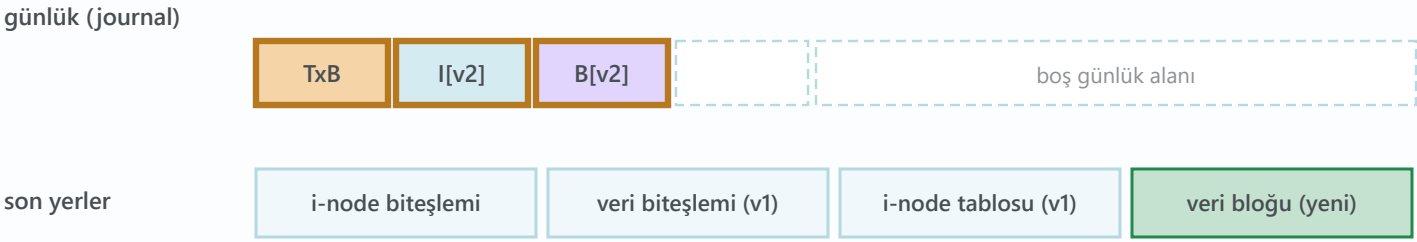


1. Veri: dosyaya eklenen blok Db önce son yerine yazılır (*ordered* mod, ext4 varsayılanı).

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.

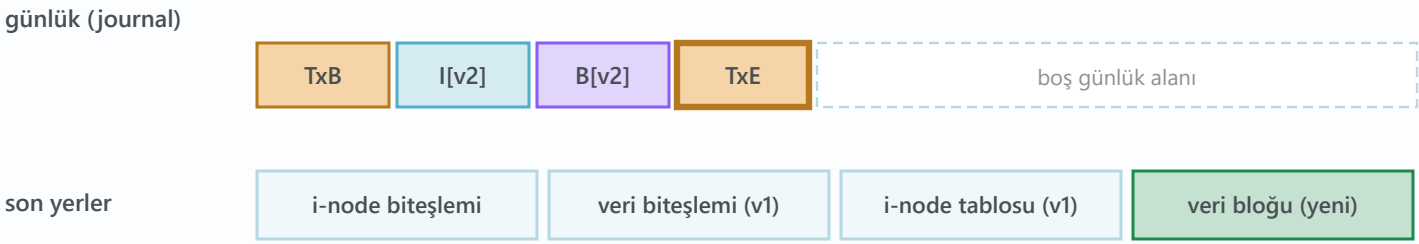
Günlükleme: Önce Günlüğe Yaz



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

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.

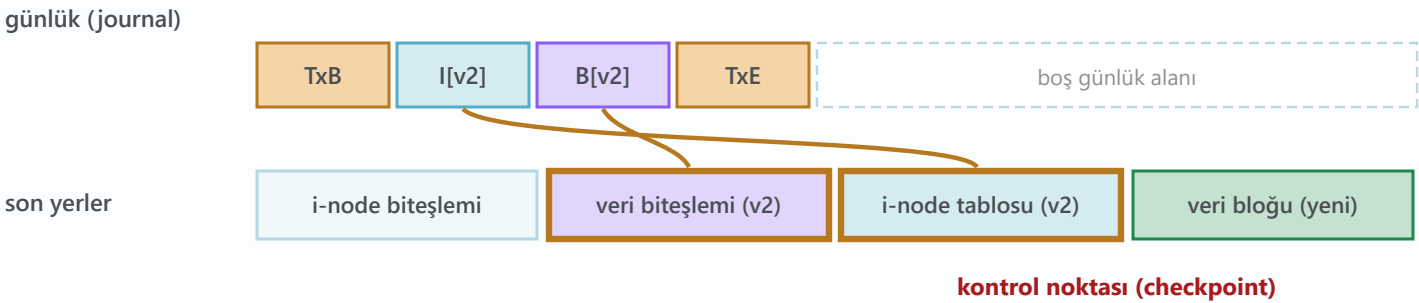
Günlükleme: Önce Günlüğe Yaz



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

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.

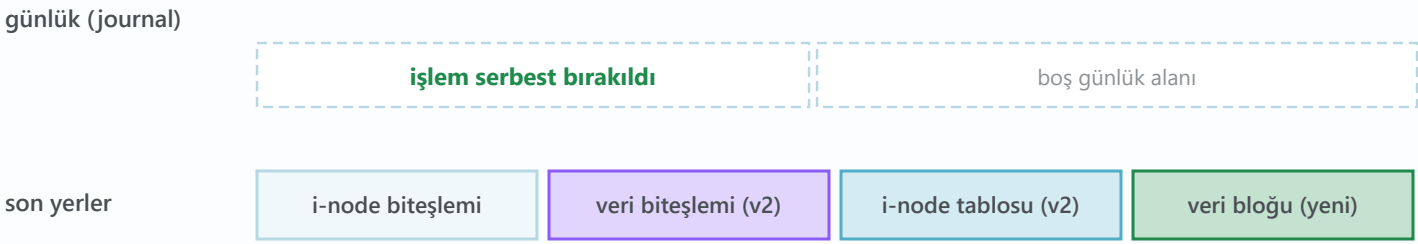
Günlükleme: Önce Günlüğe Yaz



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

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.

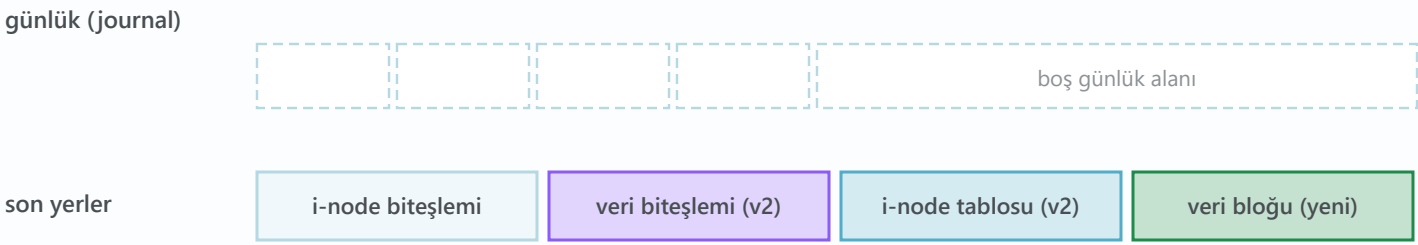
Günlükleme: Önce Günlüğe Yaz



5. Serbest bırak: günlüğün süperbloğunda işlem tamamlandı olarak işaretlenir; günlük dairesel bir tampon gibi yeniden kullanılır.

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



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.

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 }
```



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

Adım 1 / 12

DEĞİŞKENLER

`i = 0` `j = –` `k = –`

Günlükten Kurtarma: Kod İzleme

```
1  void kurtar(Kayit g[], int n) {
2    int i = 0;
▶   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 }
```



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;
▶      while (j < n && g[j].tur == BLOK)
▶        j++;
▶      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 }
```



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(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 */
▶      for (int k = i + 1; k < j; k++)
▶        yaz(g[k].hedef, g[k].veri);
11     i = j + 1;            /* sonraki */
12   }
13 }
```



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

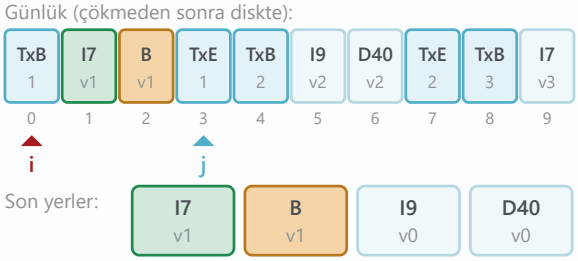
Adım 4 / 12

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 */
▶      for (int k = i + 1; k < j; k++)
▶        yaz(g[k].hedef, g[k].veri);
11     i = j + 1;          /* sonraki */
12   }
13 }
```



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;
▶   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 }
```



g[4] bir TxB: işlem 2 başlıyor.

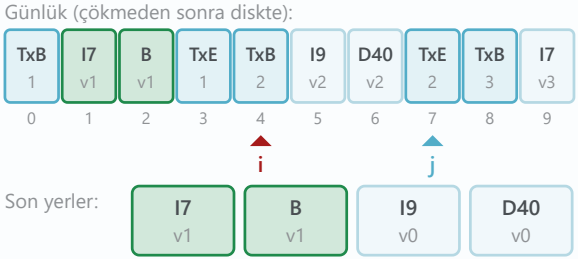
Adım 6 / 12

DEĞİŞKENLER

`i = 4` `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;
▶      while (j < n && g[j].tur == BLOK)
▶        j++;
▶      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 }
```



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

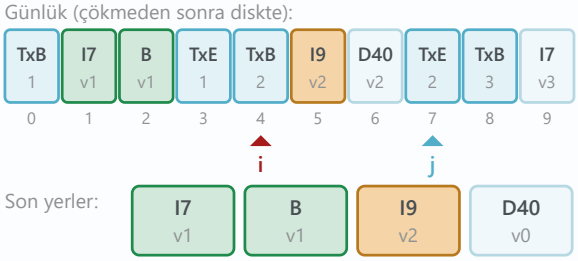
Adım 7 / 12

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 */
▶      for (int k = i + 1; k < j; k++)
▶        yaz(g[k].hedef, g[k].veri);
11     i = j + 1;            /* sonraki */
12   }
13 }
```



I9'un v2 kopyası son yerine yazılır.

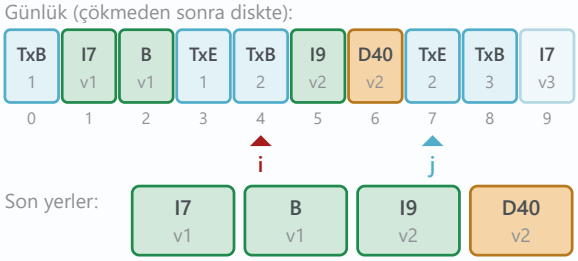
Adım 8 / 12

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 */
▶      for (int k = i + 1; k < j; k++)
▶        yaz(g[k].hedef, g[k].veri);
11     i = j + 1;            /* sonraki */
12   }
13 }
```



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;
▶   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 }
```



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;
▶      while (j < n && g[j].tur == BLOK)
▶        j++;
▶      if (j == n || g[j].tur != TXE)
▶        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 }
```



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   }
▶ }
```

Günlük (çökmeden sonra diskte):



Son yerler:



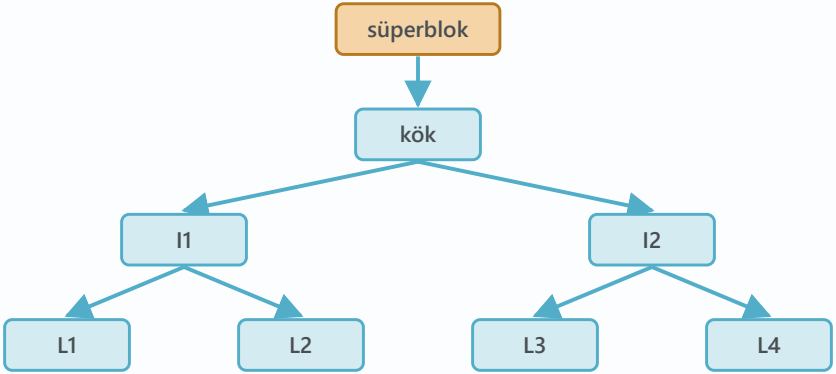
Sonuç: işlem 1 ve 2 tam olarak uygulandı, işlem 3 hiç olmamış gibi. Dosya sistemi tutarlı; tüm diski tarayan fsck gerekmez.

Adım 12 / 12

DEĞİŞKENLER

`i = 8` `j = 10` `k = –`

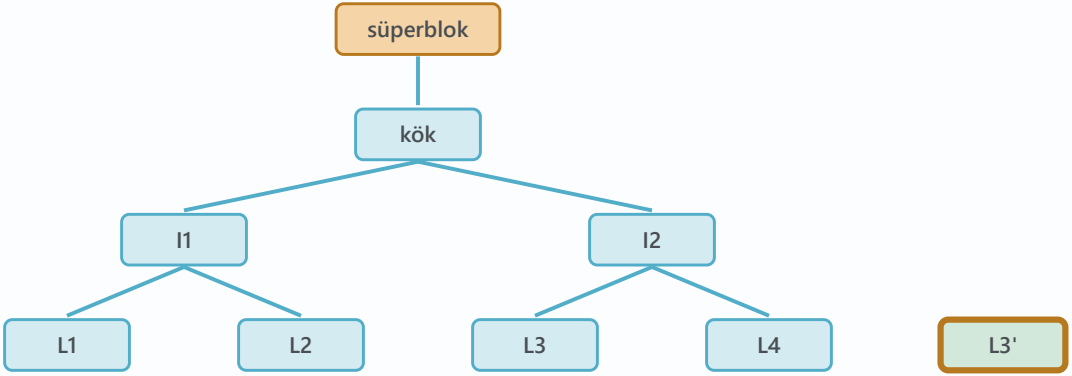
Yazınca Kopyala (CoW): ZFS, Btrfs, APFS



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

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.

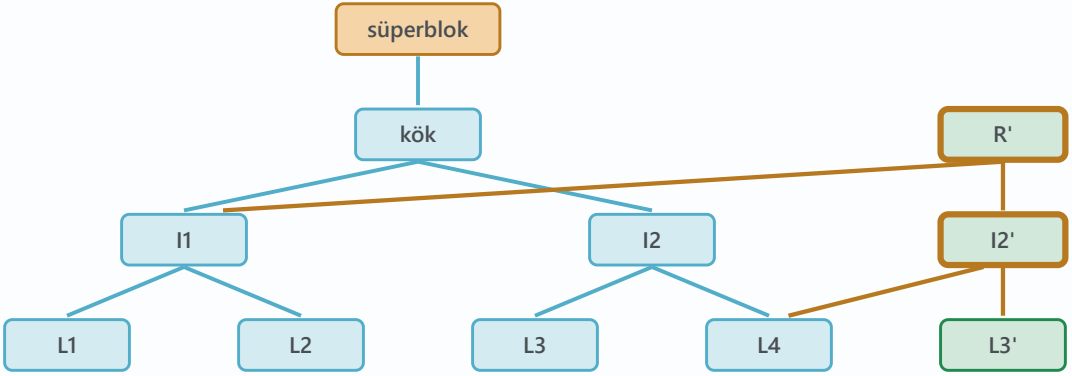
Yazınca Kopyala (CoW): ZFS, Btrfs, APFS



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

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.

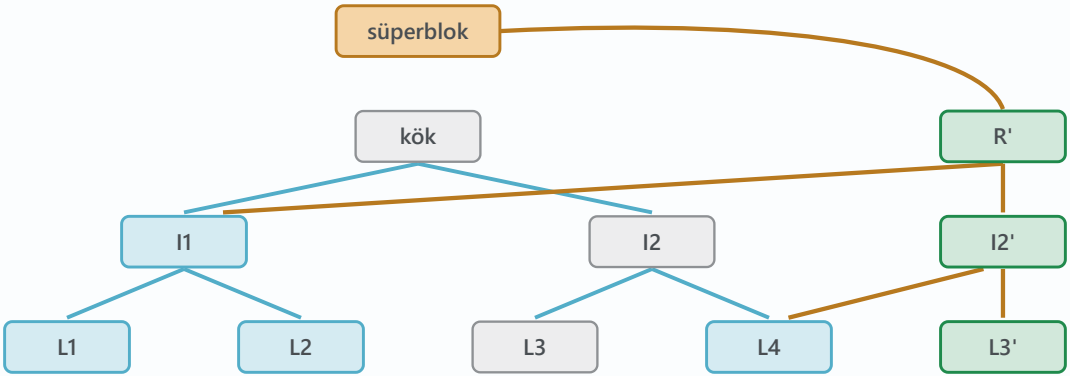
Yazınca Kopyala (CoW): ZFS, Btrfs, APFS



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

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

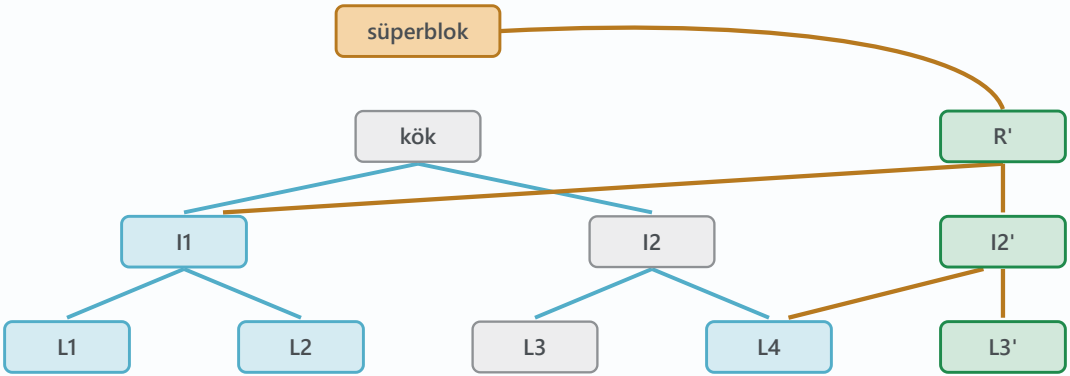


gri: eski sürüm = anlık görüntü (ya da serbest)

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

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



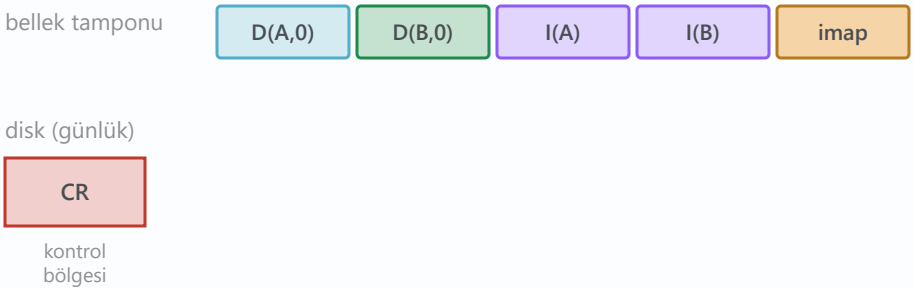
gri: eski sürüm = anlık görüntü (ya da serbest)

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*).

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)

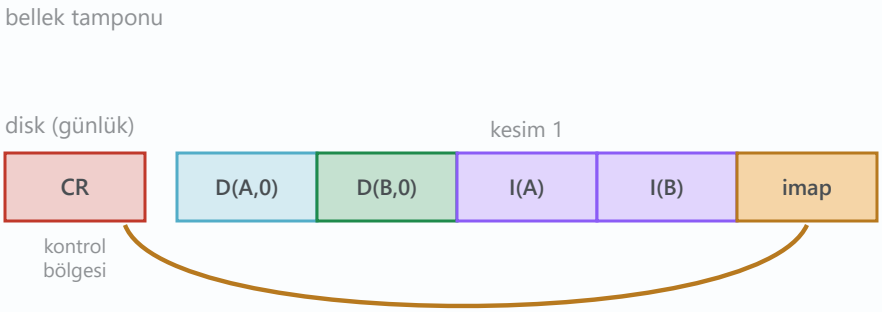


Yazmalar bellekte biriktirilir: veri blokları, değişen i-node'lar ve i-node haritası (imap) parçaları.

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 F2FS (Android), NILFS2, NetApp WAFL, ZFS/Btrfs CoW ve SSD'lerin FTL'lerinde yaşıyor.

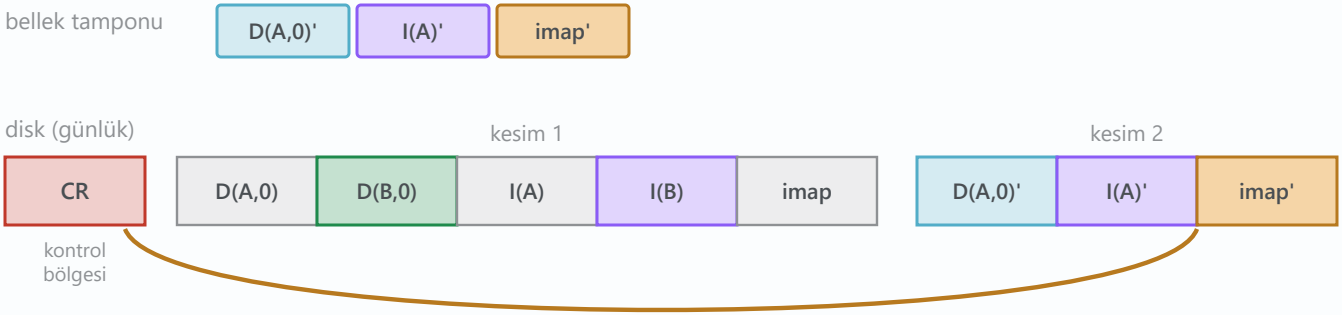
Günlük Yapılı Dosya Sistemi (LFS)



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'i gösterir.

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

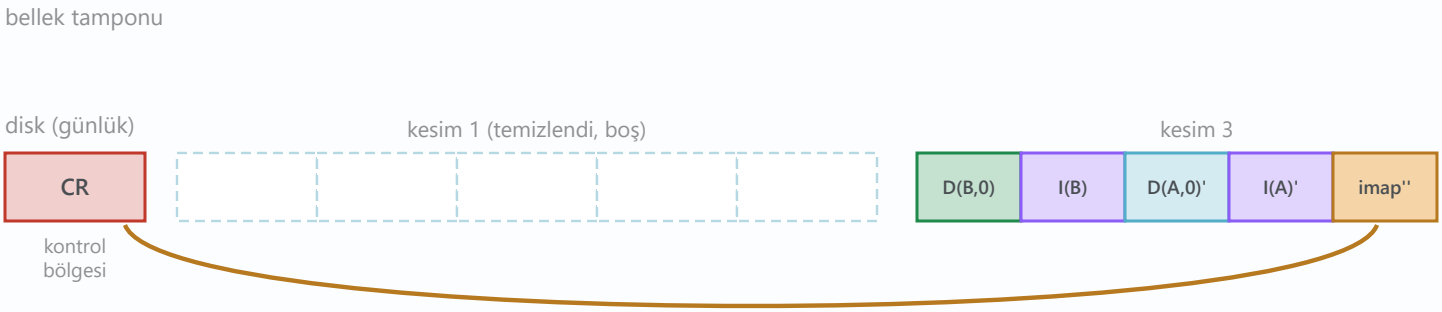
Günlük Yapılı Dosya Sistemi (LFS)



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

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

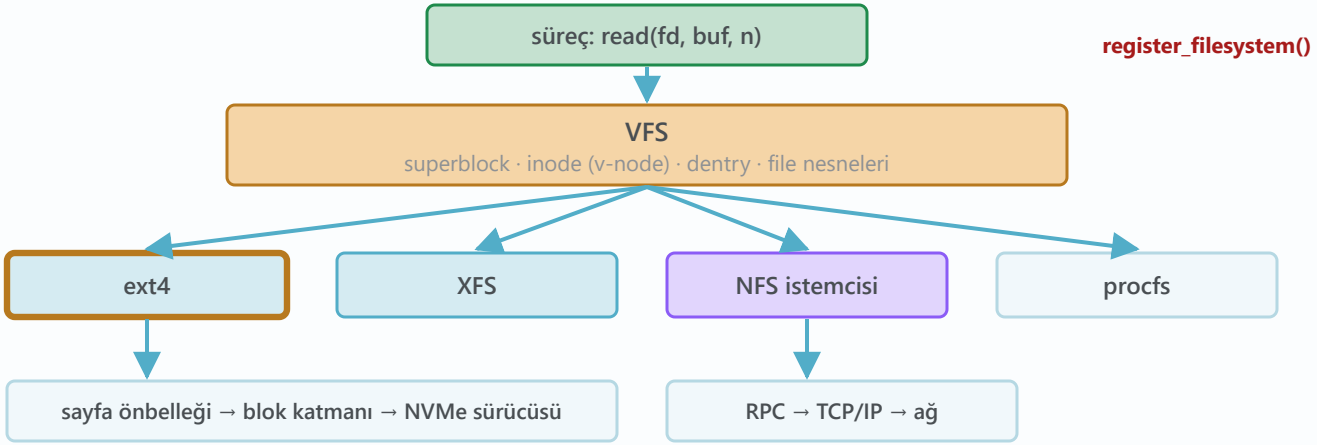
Günlük Yapılı Dosya Sistemi (LFS)



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

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

Sanal Dosya Sistemi (VFS)

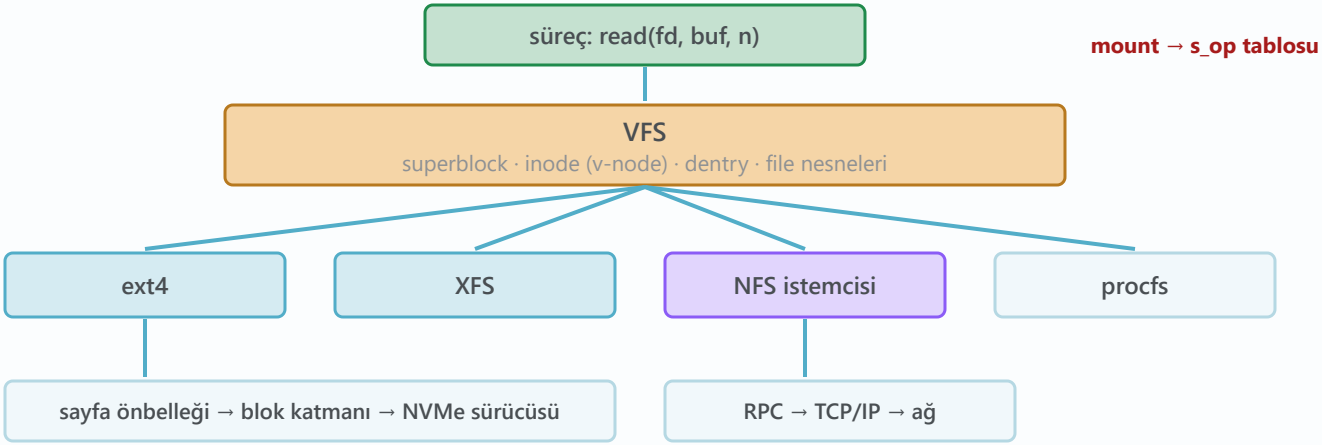


Kayıt: dosya sistemi açılışta ya da modül yüklenince VFS'ye kaydolur ve VFS'nin çağıracağı işlevlerin adres tablosunu verir.

Adım 1 / 4

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

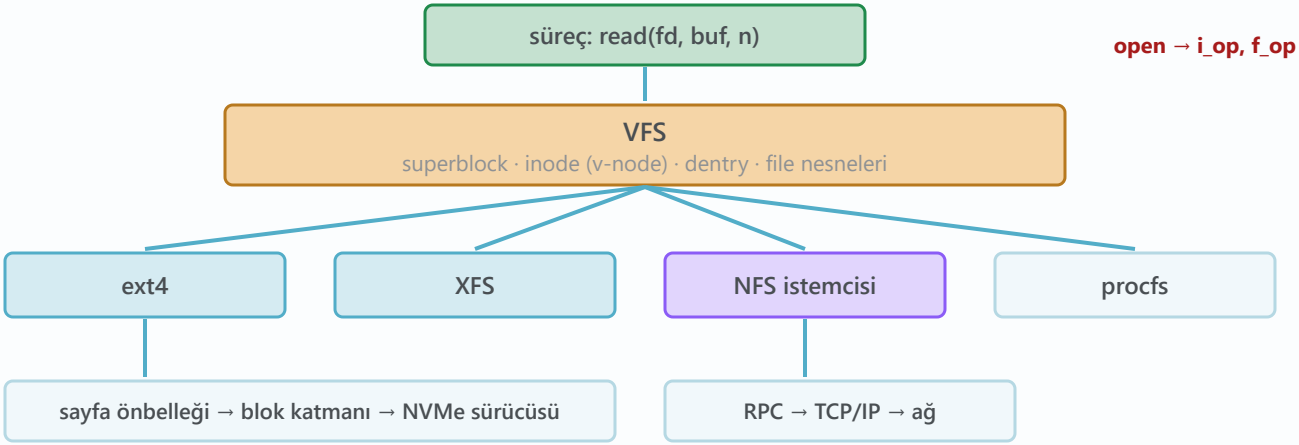
Sanal Dosya Sistemi (VFS)



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

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

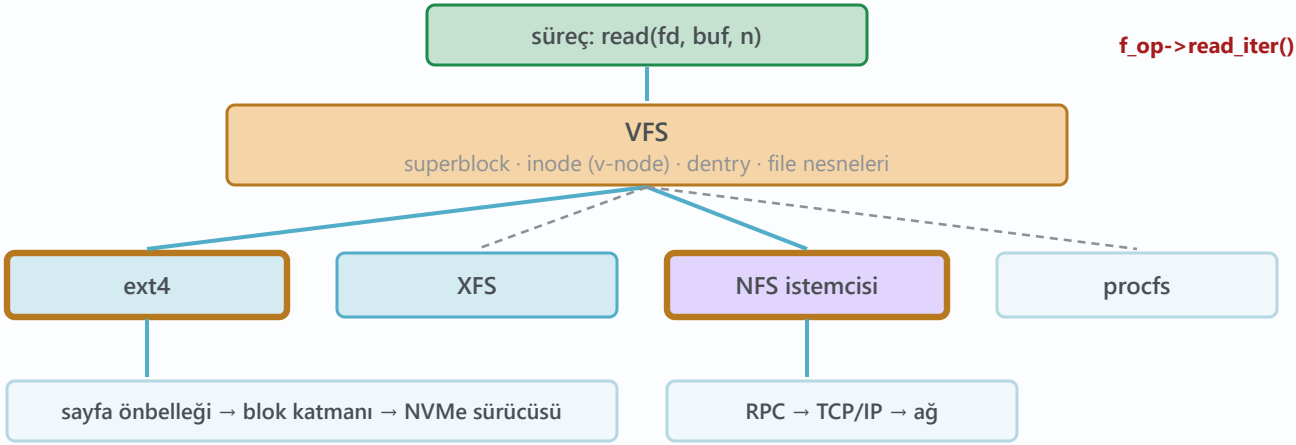
Sanal Dosya Sistemi (VFS)



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

VFS'nin amacı farklı dosya sistemlerine *aynı anda* tek arayüz sunmaktır (aynı diskte birden çok sistem bölümlemeyle 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` fonksiyon işaretçisini çağırır; ext4 dosyasında ext4'ün işlevi, NFS dosyasında ağ üzerinden RPC çalışır. Uygulama farkı görmez.

VFS'nin amacı farklı dosya sistemlerine *aynı anda* tek arayüz sunmaktır (aynı diskte birden çok sistem bölümlemeyle 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 (2049), Kerberos güvenliği, ACL, 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

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, gruplama, 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