Chapter 10: File Systems

Operating Systems

The file concept and attributes | File operations | Open-file tables and fds

Access methods | Directory structures and path resolution | Hard and symbolic links

Mounting and protection (rwx) | MBR, GPT, and the superblock

Contiguous, linked, FAT, i-node | Free-space management | Caching and performance

fsck, journaling, CoW, LFS | NTFS, ext4, FAT32, exFAT | VFS, NFS, and SSDs

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/os/slides/Chapter_10_File_Systems.html

Learning Outcomes

The file abstraction

Define a file by its name, type, attributes, and operations; trace open-file tables and file descriptors.

Directories

Compare directory structures, resolve a path step by step, and explain hard/symbolic links and mounting.

Protection

Read rwx permission bits and the special bits; tell what r, w, and x mean for a file and for a directory.

Allocation

Apply contiguous, linked, FAT, and i-node allocation to the same file; compute the maximum file size.

Free space and performance

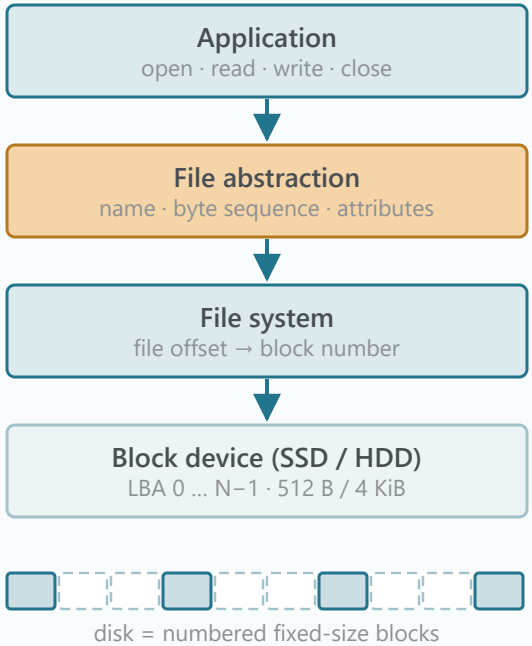
Compare bitmap, list, grouping, and counting methods; explain the buffer/page cache and read-ahead.

Reliability and real systems

Compare fsck, journaling, CoW, and LFS; recognize NTFS, ext4, FAT32/exFAT, VFS, NFS, and SSD TRIM.

Why a File System?

- Three requirements: **very large** data (it won't fit in an address space), **persistence** (it outlives the process), and **concurrent access** (many processes use the same data).
- A disk offers only numbered blocks: "*read / write block k*". The file system builds the **file** and **directory** abstractions on top.
- A file is a **logical** unit, much like an address space in memory; processes create files, and the OS manages naming, structure, access, protection, and placement.
- Two viewpoints:
  - **User**: name, type, directory, permissions, operations: what is visible?
  - **Implementer**: how are blocks allocated, free space tracked, and consistency kept after a crash?



File Naming

File system	Name length	Case	Not
FAT (MS-DOS)	8.3 (e.g., `REPORT~1.TXT`)	insensitive	VFAT long names in extra entries: 255 UTF-16
NTFS	255 UTF-16 units	preserving, insensitive	`\ / : * ? " < > |` forbidden
ext4, XFS	255 bytes	sensitive	only `/` and NUL forbidden
APFS	255 UTF-8 characters	insensitive by default on macOS	Unicode normalization

Extension (suffix)

The part of the name after the dot: `thesis.pdf`. On UNIX it is just a *convention*; the kernel gives it no meaning, but programs may (`gcc` tells `.c` from `.o`).

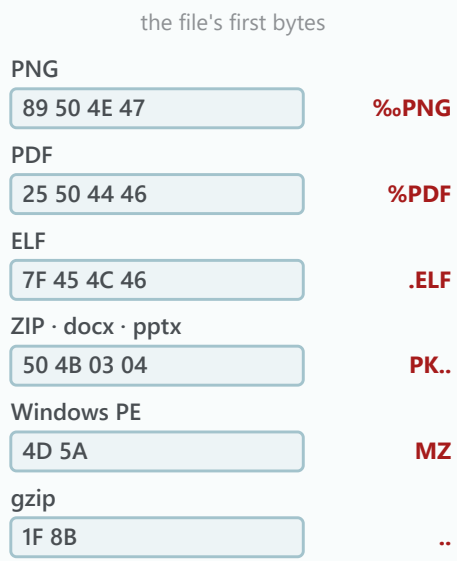
Windows

The extension is **associated** with a program in the registry; double-clicking opens that program. The hidden-extension trap: `invoice.pdf.exe`.

Older Windows versions used FAT16/FAT32, current Windows uses NTFS; FAT32 and exFAT live on today on USB sticks and SD cards. Names are stored as Unicode; non-ASCII characters are no problem.

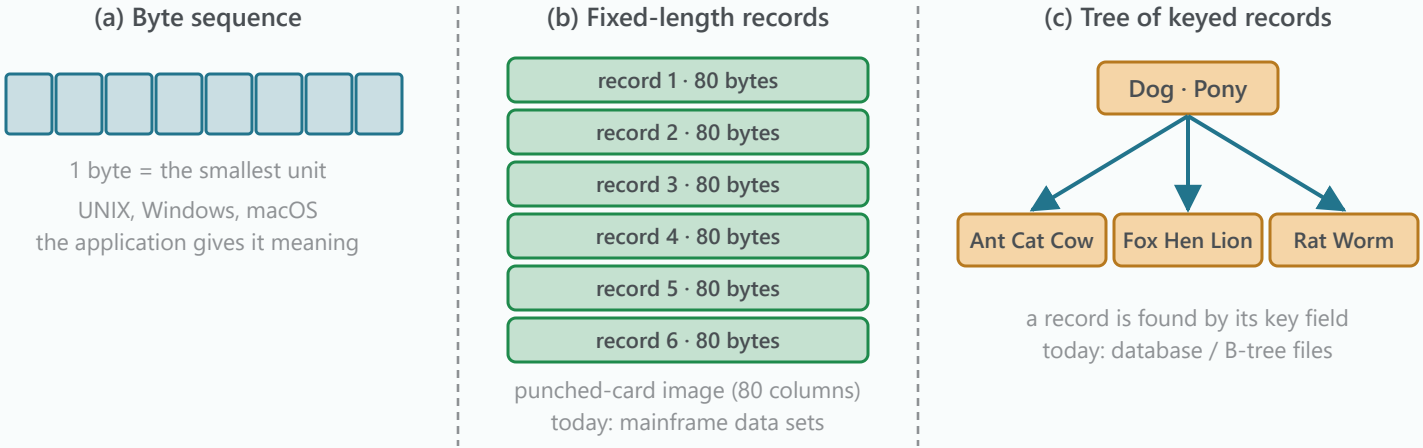
Extensions and Magic Numbers

Group	Extensions and what they stand for
Document	`docx` Word (Office Open XML), `doc` legacy binary Word, `pdf` Portable Document Format, `odt` OpenDocument, `rtf`, `txt`
Image	`jpg` JPEG, `png` Portable Network Graphics, `gif`, `bmp` bitmap, `tiff`; new: `webp`, `avif`, `heic`
Audio / video	`mp3` MPEG-1 Audio Layer III, `wav`, `m4a`, `flac`; `mp4` MPEG-4 Part 14, `mkv`, `webm`, `mov`; `flv` is obsolete
Spreadsheet / slides	`xlsx`, `csv` Comma-Separated Values, `pptx`, `odp`
Web	`html` HyperText Markup Language, `css` Cascading Style Sheets, `js` JavaScript
Executable	`exe` Windows PE, `msi` Windows Installer; on Linux ELF has no extension, libraries are `.so`
Archive	`zip`, `7z`, `rar`, `tar` Tape Archive, `gz` gzip, `zst` Zstandard



A file's type can also be recognized from its **magic number**, regardless of extension: the `file report` command does this. `docx/xlsx/pptx` are really ZIP archives.

File Structure



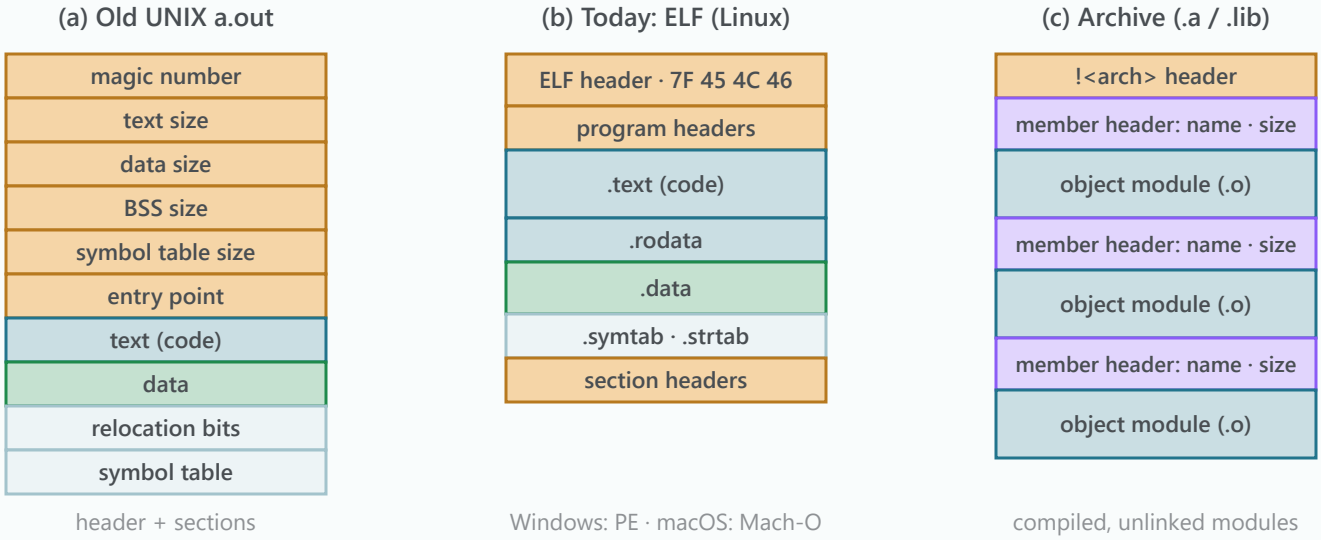
- **Maximum flexibility**: the OS sees a file only as a sequence of bytes; anything can go in it, and the application decides the structure.
- With records, the unit of reading/writing is a record; with a tree, requests like "*get the record whose key is Pony*" are supported directly.
- Today structured access is implemented in libraries and databases (SQLite, B+ trees), not in the OS.

File Types

- **Regular files** hold user data: *text* (lines of printable characters, ending in LF or CRLF) or *binary* (the program knows the structure).
- **Directories** are system files that hold the structure of the file system.
- **Special files** model I/O devices: *character* devices (printer, terminal) and *block* devices (disk).
- "Everything is a file": the same `open/read/write` interface works for devices, pipes, and sockets. The letter in the table is the first character of `ls -l` output.

Letter	Type	Example
`-`	regular file (text or binary)	`notes.txt`, `/bin/ls`
`d`	directory	`/home`
`l`	symbolic link	`/bin → usr/bin`
`c`	character device (serial, byte by byte)	`/dev/tty`, `/dev/null`
`b`	block device (block by block, random)	`/dev/nvme0n1`, `/dev/sda`
`p`	named pipe (FIFO)	`mkfifo chan`
`s`	UNIX socket	`/run/systemd/notify`

Binary Files: Executables and Archives



Each OS recognizes its own executable format; the kernel runs a file only if it starts with the right magic number. The UNIX file in the figure is the old `a.out` format (Tanenbaum); the archive extension is `.a`, and the modules inside are `.o`.

File Attributes

Attribute	Meaning	Where / how
Name, type, location	the name is in the directory entry; the type and block addresses are in the i-node/MFT record	all systems
Size	logical size (`st_size`) and allocated space (`st_blocks`)	they differ in a sparse file
Owner, group	UID / GID; SID on Windows	`chown` , `icacls`
Protection	rwx bits, ACLs	`chmod` , `setfacl`
Time stamps	access (atime), content change (mtime), i-node change (ctime), creation (btime)	`stat` ; `relatime` on Linux
Link count	number of names that refer to the file	blocks freed when it reaches 0
Read-only, Hidden, System	read-only, hidden, system file	Windows `attrib` ; on UNIX, hidden = the name starts with `.`
Archive	changed since the last backup, to be backed up	Windows; the backup tool clears it
Compressed, Encrypted	compressed / encrypted	NTFS, Btrfs, APFS; fscrypt, BitLocker
Sparse, Temporary	zero blocks absent on disk; temporary (a cache hint)	NTFS flags
Reparse point	special handling: symbolic link, junction, cloud placeholder	NTFS
Extended attributes	name = value pairs	xattr (SELinux label), NTFS ADS

Symlink, Device, Junction are file types, not attributes; *Temporary* doesn't delete the file by itself (it's only a hint to delay writes); *Creator* and *Max size* belong to old systems.

The stat Structure and ls -l

```
struct stat {
  dev_t     st_dev;      /* device (file system)   */
  ino_t     st_ino;      /* i-node number          */
  mode_t    st_mode;     /* type + permission bits */
  nlink_t   st_nlink;    /* hard link count        */
  uid_t     st_uid;      /* owner user             */
  gid_t     st_gid;      /* owner group            */
  dev_t     st_rdev;     /* no. if a device file   */
  off_t     st_size;     /* size (bytes)           */
  blksize_t st_blksize;  /* preferred I/O size     */
  blkcnt_t  st_blocks;   /* 512 B units allocated  */
  struct timespec st_atim; /* last access  */
  struct timespec st_mtim; /* last content */
  struct timespec st_ctim; /* last i-node  */
};
```

$ ls -l b.txt



1
→ hard link count (st_nlink)

root root
→ owner and group

81
→ size in bytes (st_size)

May 26 19:20
→ last modification (mtime)

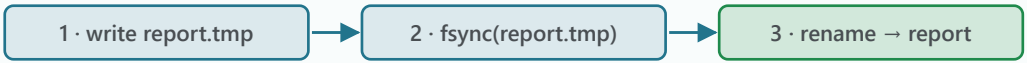
b.txt
→ name: comes from the directory entry

the name is in the directory, not the i-node!

Since POSIX.1-2008 the times are `struct timespec` fields with nanoseconds (`st_atime` remains as a macro). On Linux the creation time is read with `statx()`.

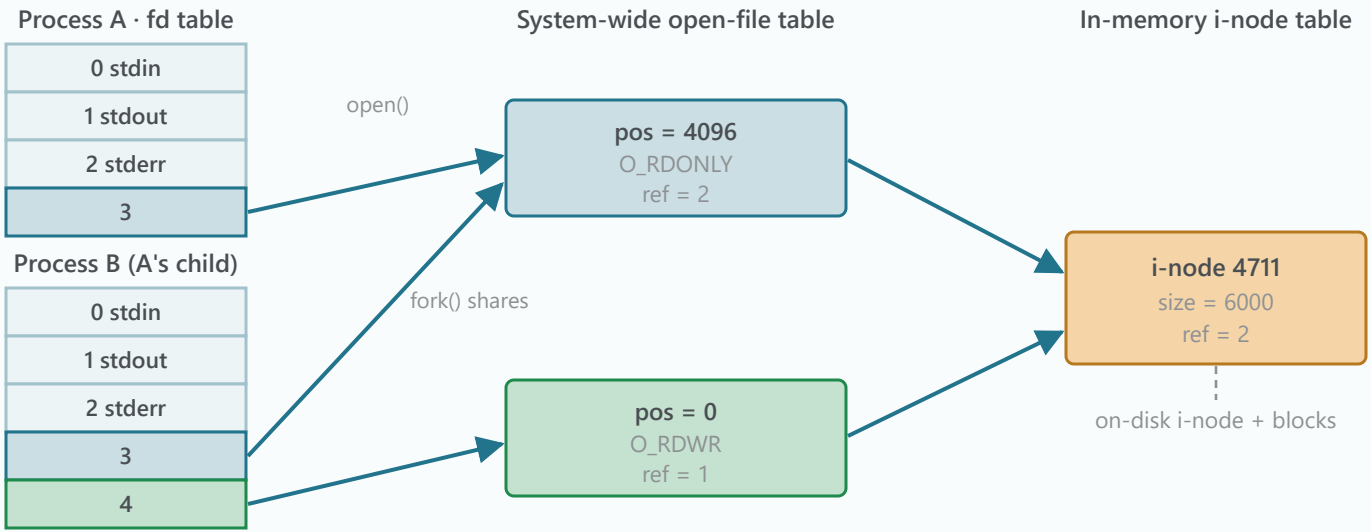
File Operations and System Calls

Operation	POSIX call	What it does
Create	`open(O_CREAT)`, `creat`	creates an empty file and directory entry, sets the attributes
Delete	`unlink`	removes the name; the space is freed when the link count is 0 and the last descriptor closes
Open	`open`	resolves the path, brings the i-node into memory, returns a descriptor (fd)
Close	`close`	frees the fd and, if needed, the table entries
Read / Write	`read`, `write`	reads / writes from the file position and advances it
Append	`open(O_APPEND)`	each write goes atomically to the end of the file
Seek	`lseek`	changes the position (random access); does no I/O
Get / Set attributes	`stat`, `chmod`, `chown`, `utimensat`	reads / changes attributes
Rename	`rename`	changes the name; atomic within one file system
Truncate, Map	`ftruncate`, `mmap`	shortens/extends the file; maps it into the address space



The safe-save pattern: write the new contents to a temporary file → `fsync` → `rename(tmp, real)`. After a crash either the old or the new file is visible, never a half-written one.

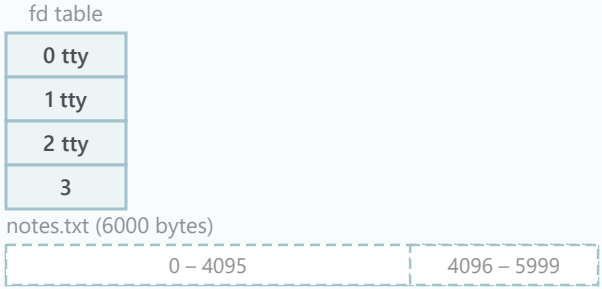
Open-File Tables



- **fd**: an index into the kernel's per-process table; 0, 1, 2 are standard input/output/error, and `open` returns the lowest free number.
- An **open-file entry** holds the position (offset) and access mode: `fork` and `dup` share the entry (a common position); a second `open` creates a new entry (a separate position).
- The **i-node** is in memory once per file; further opens don't reread it from disk. The Windows equivalent is the handle table.

open, read, close, Step by Step

```
1  #include <fcntl.h>
2  #include <unistd.h>
3
4  int main(void) {
▶    char buf[4096];
6    int fd = open("notes.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 }
```



Initially only 0, 1, 2 are open (inherited from the shell: the terminal).

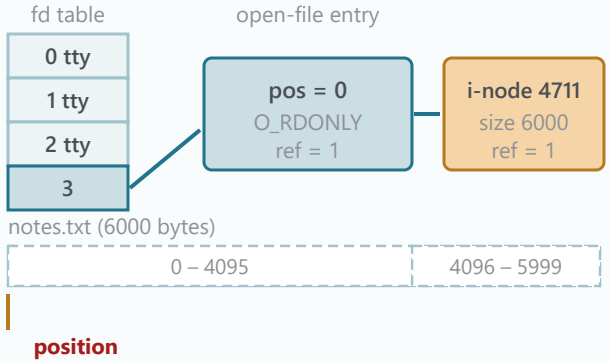
Step 1 / 6

VARIABLES

`fd = –` `n = –` `pos = –`

open, read, close, Step by Step

```
1  #include <fcntl.h>
2  #include <unistd.h>
3
4  int main(void) {
5    char buf[4096];
▶    int fd = open("notes.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` : the path is resolved, i-node 4711 is brought into memory, and an open-file entry with position 0 is set up; the lowest free number, **3**, is returned.

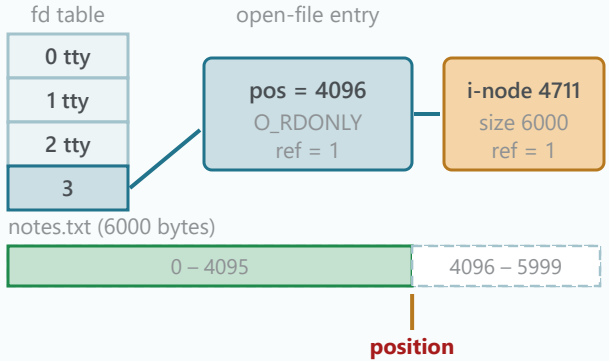
Step 2 / 6

VARIABLES

`fd = 3` `n = –` `pos = 0`

open, read, close, Step by Step

```
1  #include <fcntl.h>
2  #include <unistd.h>
3
4  int main(void) {
5    char buf[4096];
6    int fd = open("notes.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 }
```



First `read` : the block(s) are read/cached, 4096 bytes are copied to `buf` , and the position becomes **4096**.

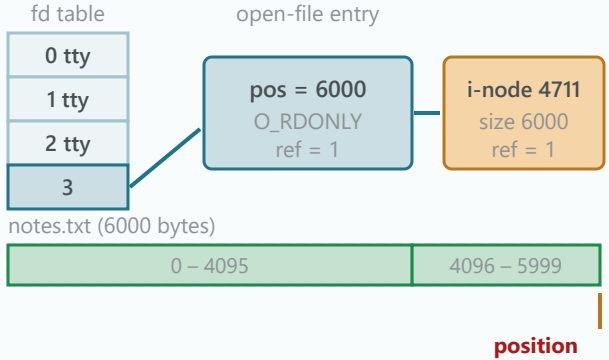
Step 3 / 6

VARIABLES

`fd = 3` `n = 4096` `pos = 4096`

open, read, close, Step by Step

```
1  #include <fcntl.h>
2  #include <unistd.h>
3
4  int main(void) {
5    char buf[4096];
6    int fd = open("notes.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 }
```



Second `read`: only 1904 bytes are left in the file; a **short read** is not an error. Position 6000.

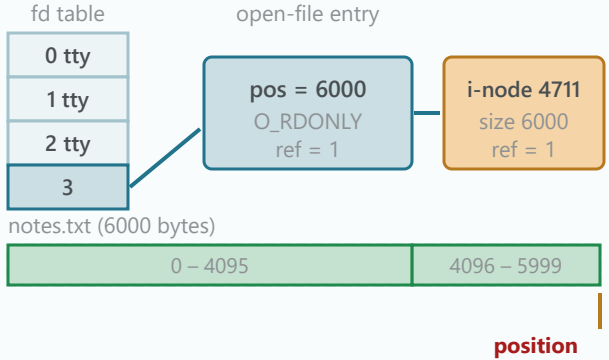
Step 4 / 6

VARIABLES

`fd = 3` `n = 1904` `pos = 6000`

open, read, close, Step by Step

```
1  #include <fcntl.h>
2  #include <unistd.h>
3
4  int main(void) {
5    char buf[4096];
6    int fd = open("notes.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 }
```



The third `read` returns **0**: end of file (EOF). On an error it would return −1 and set `errno`.

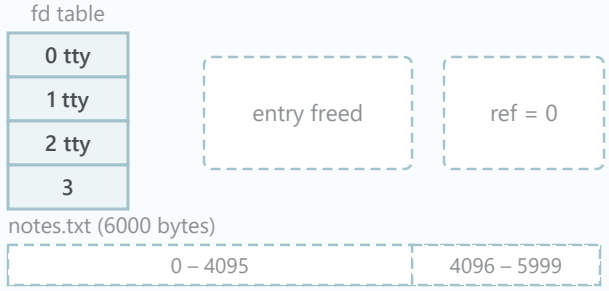
Step 5 / 6

VARIABLES

`fd = 3` `n = 0` `pos = 6000`

open, read, close, Step by Step

```
1  #include <fcntl.h>
2  #include <unistd.h>
3
4  int main(void) {
5    char buf[4096];
6    int fd = open("notes.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 is freed; the entry's reference count drops to 0 → it is deleted, and the i-node's to 0 → it can be evicted from memory.

Step 6 / 6

VARIABLES

`fd = closed` `n = 0` `pos = –`

Copying a File: copy abc xyz

```
1  #define BUF 4096
2
3  int copy(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` : if xyz exists its contents are erased; otherwise `O_CREAT` creates it with mode 0644.

Step 1 / 6

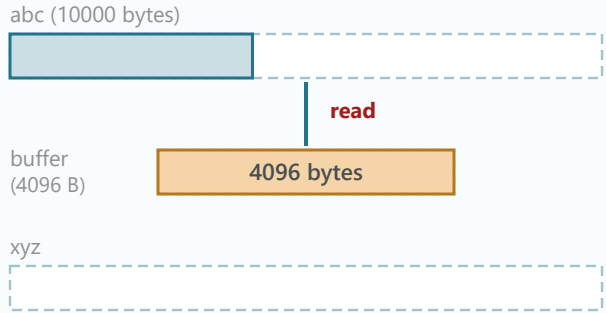
VARIABLES

`in = 3` `out = 4` `n = –` `copied = 0`

Real code must check every call for a −1 return and complete a short `write` in a loop; with `O_CREAT` the third argument (the mode) must be given.

Copying a File: copy abc xyz

```
1  #define BUF 4096
2
3  int copy(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 }
```



First `read`: 4096 bytes into the buffer.

Step 2 / 6

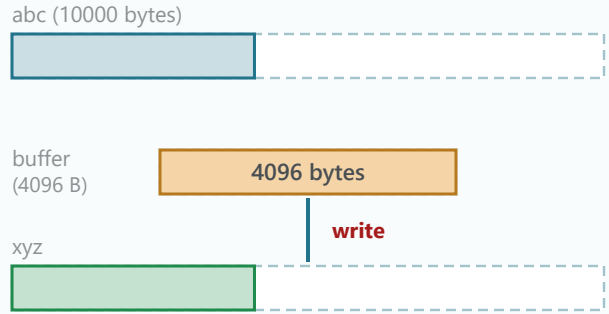
VARIABLES

`in = 3` `out = 4` `n = 4096` `copied = 0`

Real code must check every call for a −1 return and complete a short `write` in a loop; with `O_CREAT` the third argument (the mode) must be given.

Copying a File: copy abc xyz

```
1  #define BUF 4096
2
3  int copy(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` writes the same 4096 bytes to xyz (to the page cache first; it reaches the disk later).

Step 3 / 6

VARIABLES

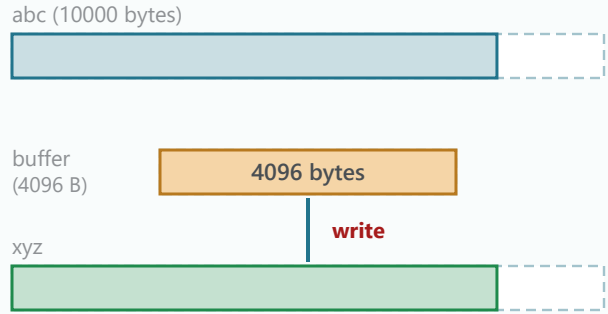
`in = 3` `out = 4` `n = 4096`

`copied = 4096`

Real code must check every call for a −1 return and complete a short `write` in a loop; with `O_CREAT` the third argument (the mode) must be given.

Copying a File: copy abc xyz

```
1  #define BUF 4096
2
3  int copy(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 }
```



Second round: 8192 bytes copied.

Step 4 / 6

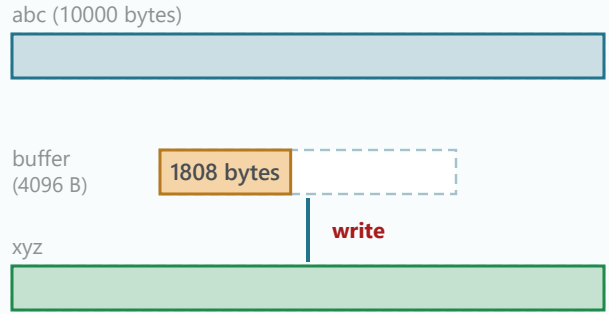
VARIABLES

`in = 3` `out = 4` `n = 4096` `copied = 8192`

Real code must check every call for a −1 return and complete a short `write` in a loop; with `O_CREAT` the third argument (the mode) must be given.

Copying a File: copy abc xyz

```
1  #define BUF 4096
2
3  int copy(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 }
```



The third `read` returns only **1808** bytes; `write` writes exactly `n` bytes (not BUF!).

Step 5 / 6

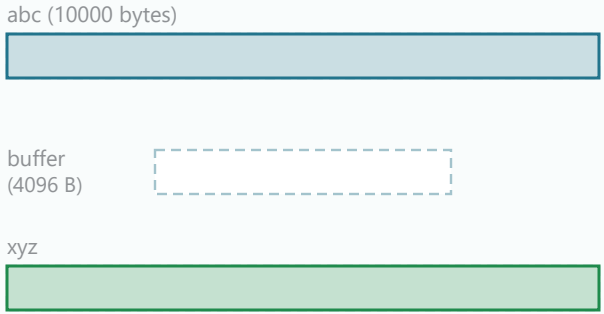
VARIABLES

`in = 3` `out = 4` `n = 1808` `copied = 10000`

Real code must check every call for a −1 return and complete a short `write` in a loop; with `O_CREAT` the third argument (the mode) must be given.

Copying a File: copy abc xyz

```
1  #define BUF 4096
2
3  int copy(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` returns 0 → the loop ends; both descriptors are closed. In total 4 `read` + 3 `write` calls.

Step 6 / 6

VARIABLES

`in = closed` `out = closed` `n = 0` `copied = 10000`

Real code must check every call for a −1 return and complete a short `write` in a loop; with `O_CREAT` the third argument (the mode) must be given.

Access Methods

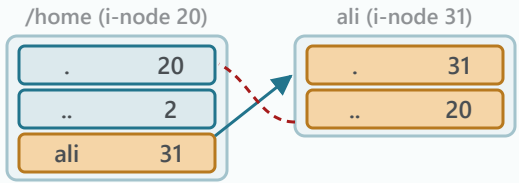


- **Sequential**: from start to end in order; rewinding instead of jumping. Tapes and cassettes worked this way; it is still the most common pattern for logs and streams.
- **Direct**: the file is a sequence of numbered blocks/records; you start where you like (`lseek` , `pread`). Disks and SSDs support this.
- **Indexed**: a small index (and if needed an index of the index) finds the right block by key; it is built on top of direct access.
- With **memory-mapped** access (`mmap`) the file's pages are mapped into the address space; reads and writes are ordinary memory accesses.

Directory Operations

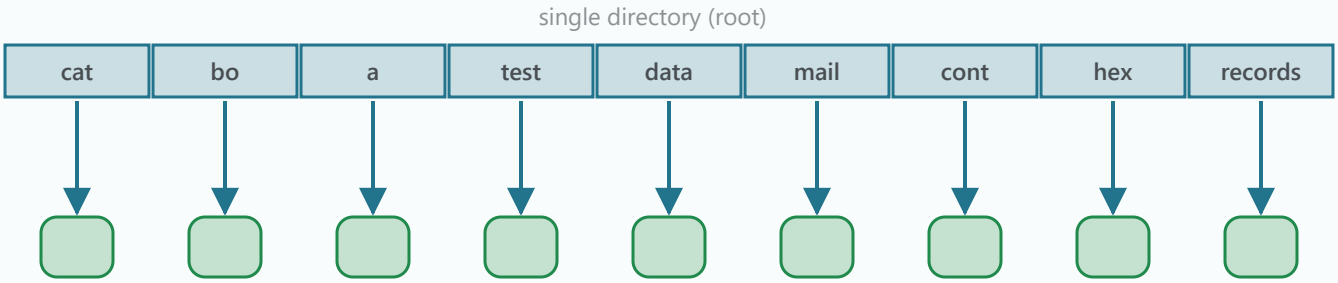
Call	Function
`mkdir`	creates an empty directory (containing only `.` and `..`)
`rmdir`	deletes a directory; it must be **empty**
`opendir` / `closedir`	opens / closes a directory stream
`readdir`	returns the next entry (name, i-node no., type) in the open directory
`rename`	renames or moves a file or directory (atomic within one file system)
`link`	adds a new name (a hard link) for an existing *file*; the i-node's link count goes up
`symlink`	creates a symbolic link containing the target path
`unlink`	removes a name; the file is deleted when the link count reaches 0

- A directory is a file too: its contents are *(name → file)* mappings; but users can change it only through these calls, not with `write`.
- `link` doesn't attach a directory to another directory; it gives a file a second name; hard links to directories are forbidden to users (to prevent cycles).



link counts: 20 → 3 (new ..), 31 → 2 (ali and .)

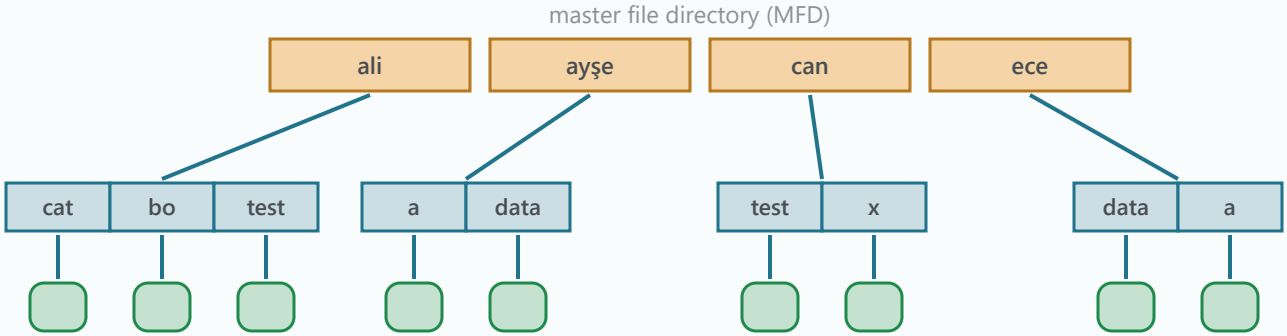
Directory Structures



All users' files are in one place: two users can't both have a file named "test",
and files can't be grouped. Early systems (CDC 6600) and some embedded devices worked this way.

Single-level: simple, but has **naming** and **grouping** problems.

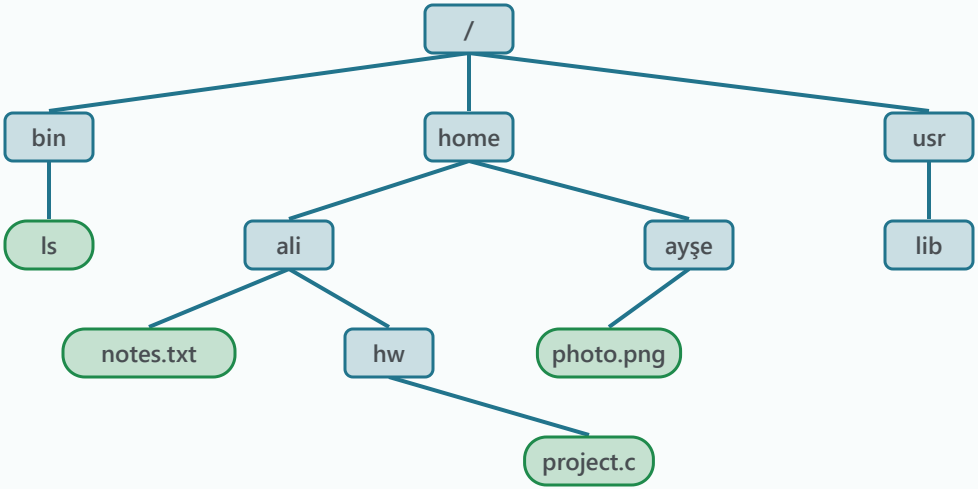
Directory Structures



A user file directory (UFD) per user: no name clashes (ali/test ≠ can/test), but sharing and grouping are hard.

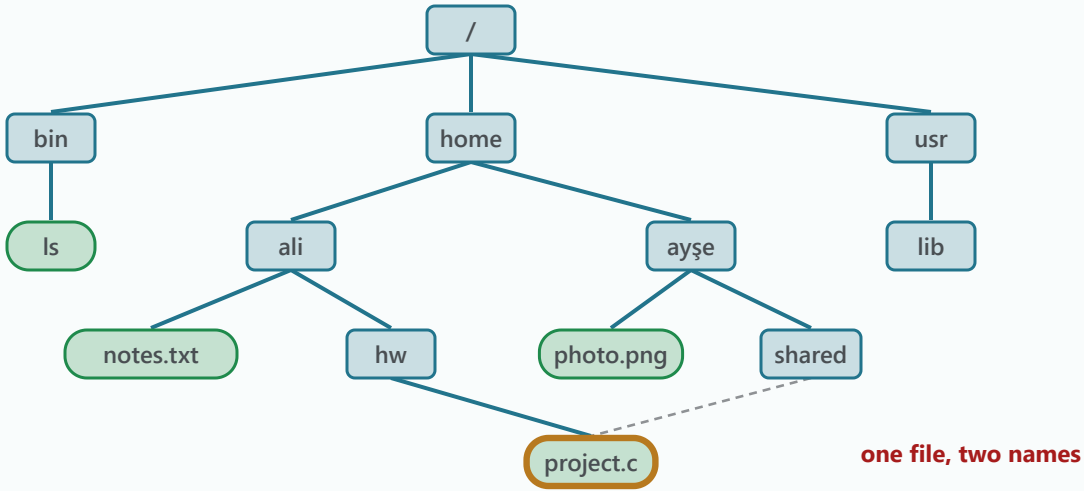
Two-level: path = `user/file`. System programs need a separate "user" directory; the idea of a *search path* comes from here.

Directory Structures



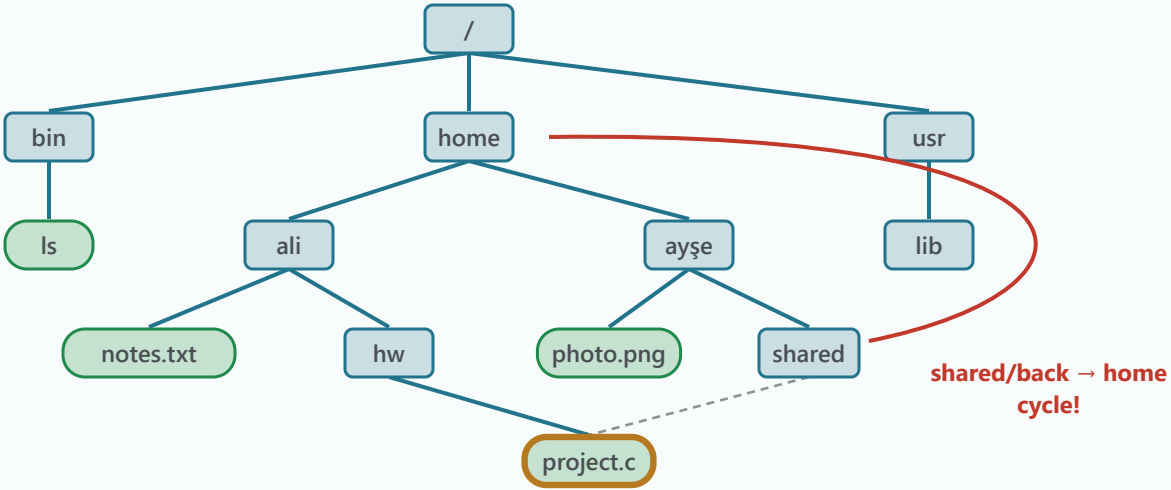
Tree (hierarchical): users create as many subdirectories as they like; every file has a single absolute path. The basis of all general-purpose systems today.

Directory Structures



Acyclic graph (DAG): `project.c` appears under both `ali/hw` and `ayşe/shared` (a link). Problems: a traversal may count it twice, and deletion becomes ambiguous.

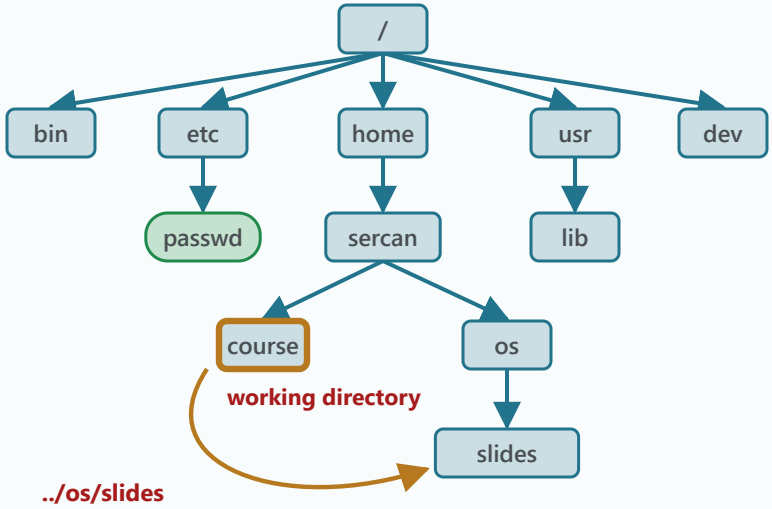
Directory Structures



General graph: linking a directory creates a **cycle**; searches may never end, and unreachable cycles need garbage collection. That's why UNIX forbids hard links to directories.

Path Names

- An **absolute path** starts at the root: `/home/sercan/os/slides`. Windows: `C:\Users\sercan\os`; a network share `\\server\share`.
- A **relative path** starts at the working directory (`pwd`, `cd`): from `/home/sercan/course`, `../os/slides`.
- `.` is the current directory, `..` the parent; at the root, `..` leads back to the root.
- The separator is `/` on UNIX and `\` on Windows (the API accepts `/` too).



In historical UNIX, user directories were under `/usr`; today they are under `/home` (`/Users` on macOS), and `/usr` is for read-only system software.

Path Resolution, Step by Step

root directory

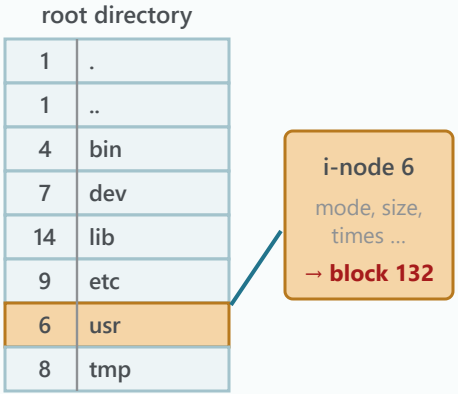
1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Absolute path: the root directory's i-node is known (1 in this example; 2 in V7 UNIX and ext4). **usr** is looked up in the root directory → **i-node 6**.

Step 1 / 5

The example is Tanenbaum's figure: `/usr/ast/mbox` is looked up. For a relative path the search starts at the working directory's i-node instead of the root.

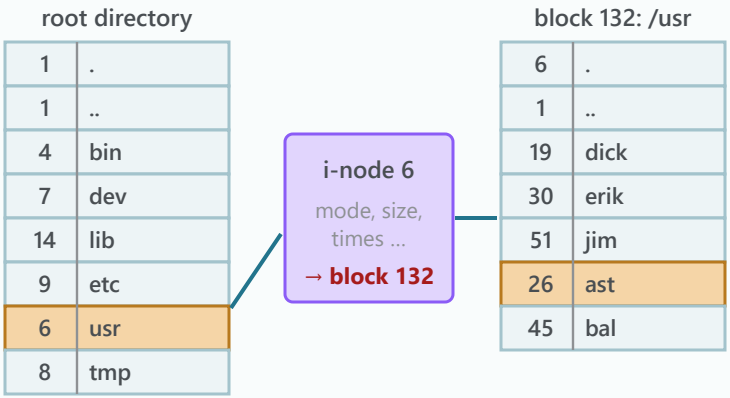
Path Resolution, Step by Step



i-node 6 is read: `/usr` is a directory whose contents are in **block 132**. (Permissions are checked here too: `x` on the directory is required.)

The example is Tanenbaum's figure: `/usr/ast/mbox` is looked up. For a relative path the search starts at the working directory's i-node instead of the root.

Path Resolution, Step by Step

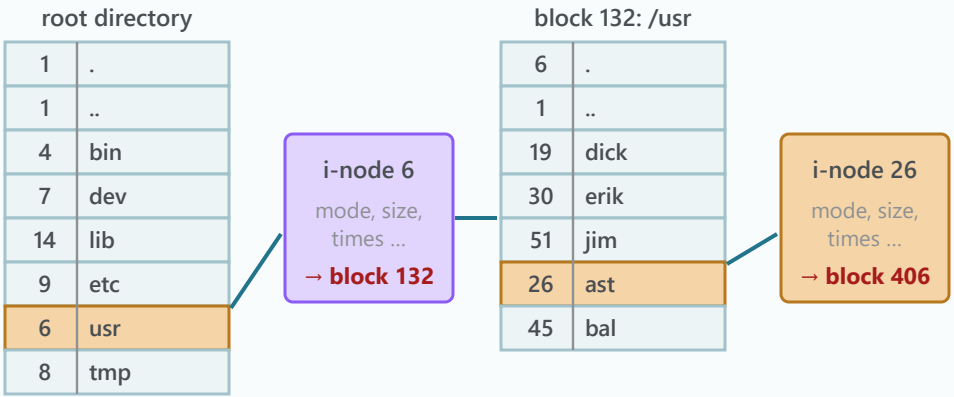


Block 132 is read and **ast** is looked up → **i-node 26**.

Step 3 / 5

The example is Tanenbaum's figure: `/usr/ast/mbox` is looked up. For a relative path the search starts at the working directory's i-node instead of the root.

Path Resolution, Step by Step

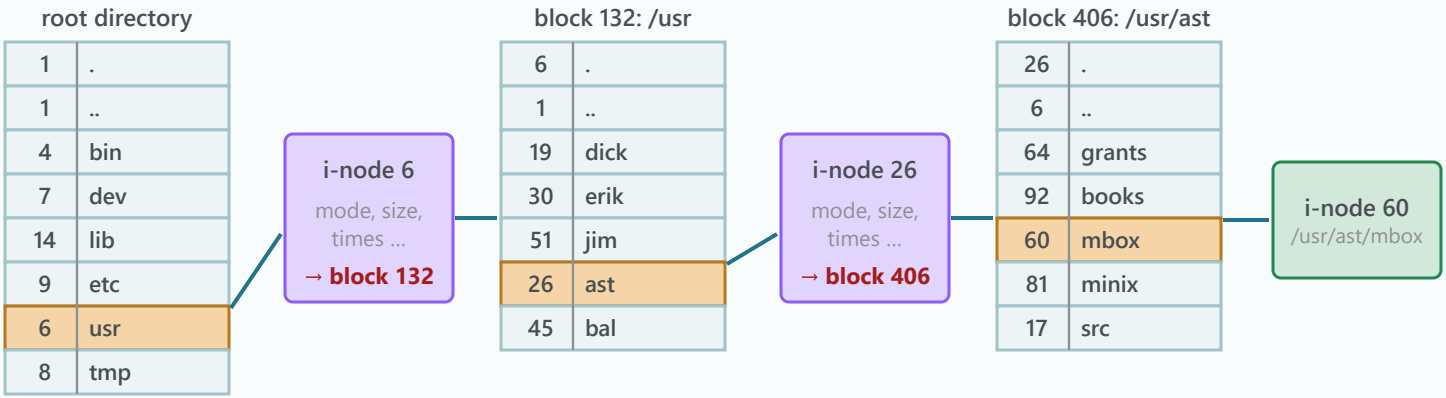


i-node 26: the contents of the `/usr/ast` directory are in **block 406**.

Step 4 / 5

The example is Tanenbaum's figure: `/usr/ast/mbox` is looked up. For a relative path the search starts at the working directory's i-node instead of the root.

Path Resolution, Step by Step

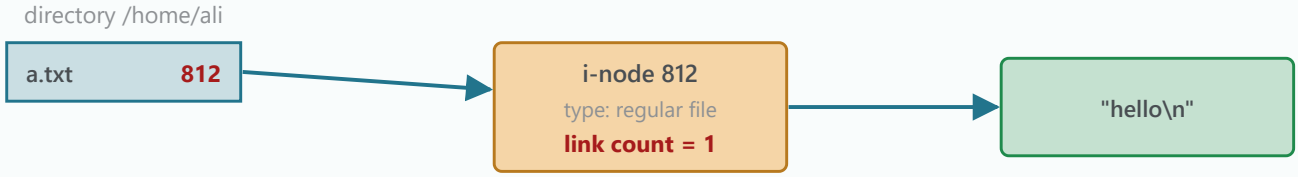


In block 406, **mbox → i-node 60**. Without caching that's 5 disk reads; that's why Linux keeps name → i-node mappings in the **dentry cache**.

Step 5 / 5

The example is Tanenbaum's figure: `/usr/ast/mbox` is looked up. For a relative path the search starts at the working directory's i-node instead of the root.

Hard and Symbolic Links

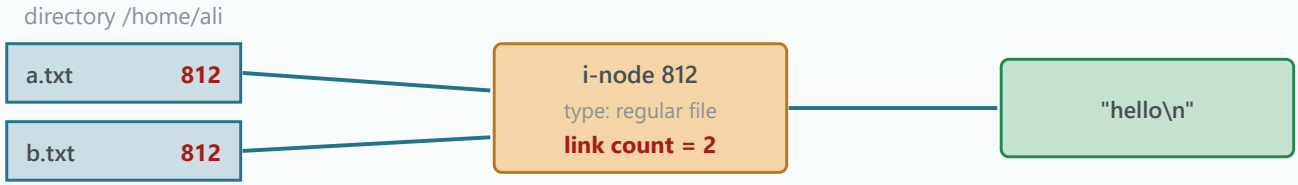


`echo hello > a.txt` : directory entry **a.txt** → **i-node 812**, link count 1.

Step 1 / 4

Hard links: within one file system, not to directories, no extra cost. Symbolic links: can point to directories, files on other file systems, even network paths; each access needs extra path resolution, and the kernel limits loops (`ELOOP`). Windows: `mklink /H` , `mklink` , junctions.

Hard and Symbolic Links

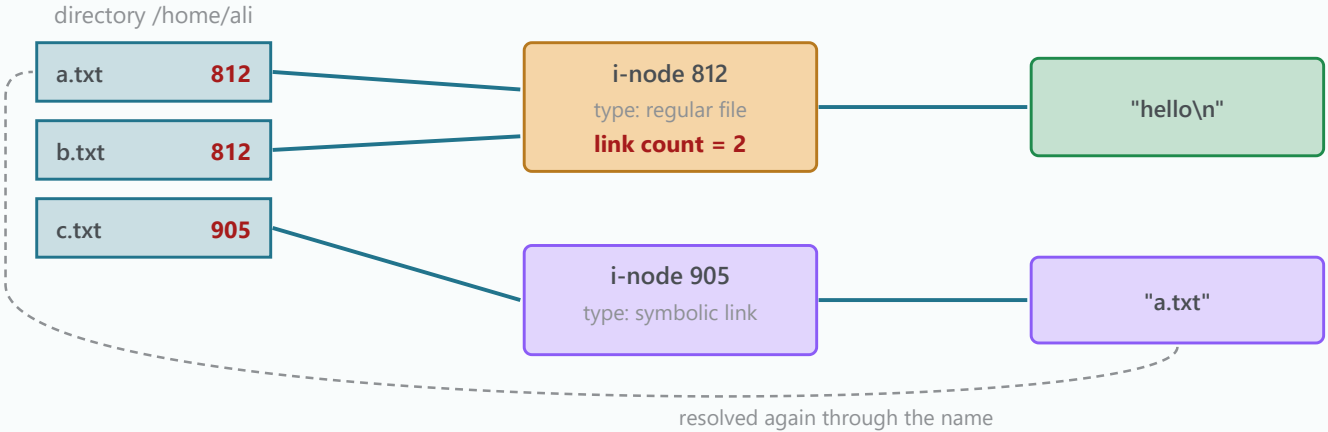


`ln a.txt b.txt` (a **hard link**): the new entry points to the same i-node; link count **2**. The two names are equal; neither is the "original".

Step 2 / 4

Hard links: within one file system, not to directories, no extra cost. Symbolic links: can point to directories, files on other file systems, even network paths; each access needs extra path resolution, and the kernel limits loops (`ELOOP`). Windows: `mklink /H` , `mklink` , junctions.

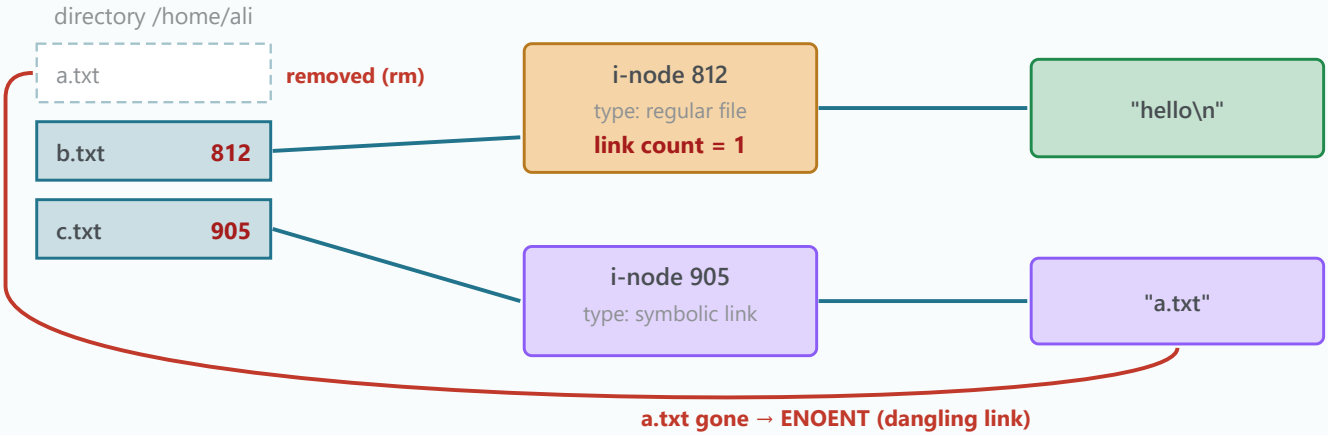
Hard and Symbolic Links



`ln -s a.txt c.txt` (a **symbolic link**): a small file with its own i-node (905), whose contents are the target's **path**.

Hard links: within one file system, not to directories, no extra cost. Symbolic links: can point to directories, files on other file systems, even network paths; each access needs extra path resolution, and the kernel limits loops (`ELOOP`). Windows: `mklink /H` , `mklink` , junctions.

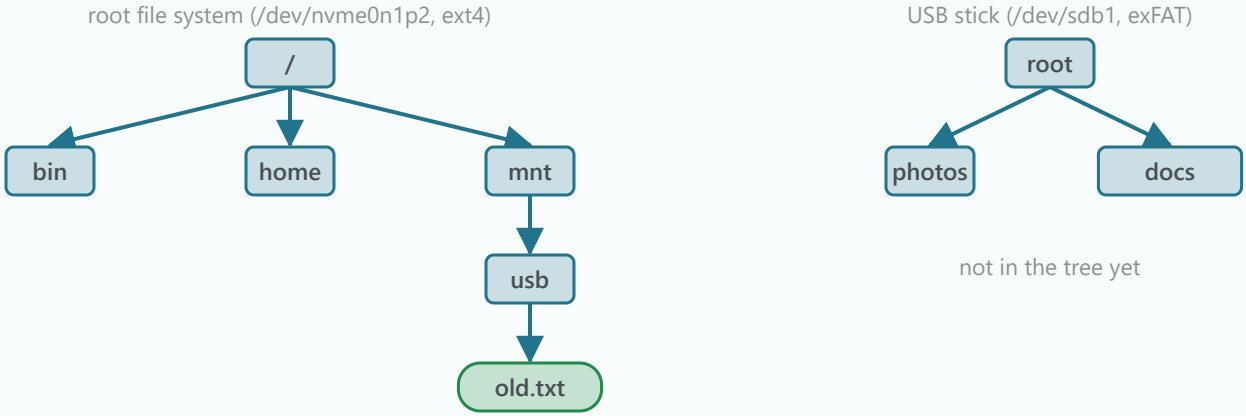
Hard and Symbolic Links



`rm a.txt` : link count 1 → the data stays, and **b.txt** works. **c.txt** dangles. (Tanenbaum: if owner C deletes the file, B's link remains but C still owns the i-node, so it counts against C's quota.)

Hard links: within one file system, not to directories, no extra cost. Symbolic links: can point to directories, files on other file systems, even network paths; each access needs extra path resolution, and the kernel limits loops (`ELOOP`). Windows: `mklink /H` , `mklink` , junctions.

Mounting

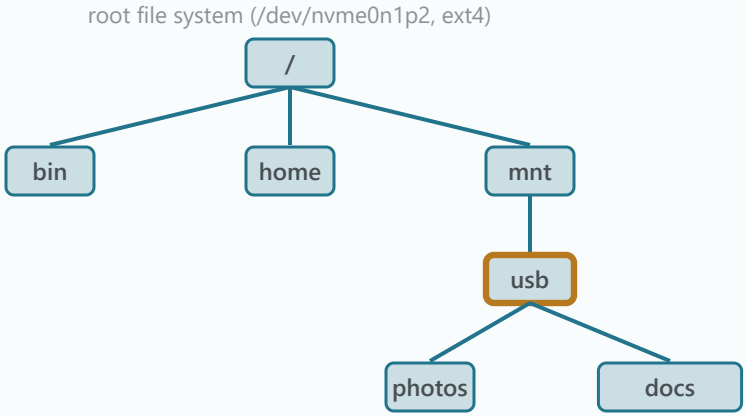


Two separate file systems. `/mnt/usb` is an ordinary directory containing `old.txt`.

Step 1 / 3

At boot the root file system is mounted, the others via `/etc/fstab` or systemd units; `findmnt` shows the tree. Windows gives each volume a drive letter (`C:` , `D:`), but NTFS can also mount on an empty folder. Crossing a mount point applies permissions and options such as `nosuid` , `noexec` , `ro` .

Mounting



the USB root took /mnt/usb's place

/mnt/usb/photos → photos on the USB

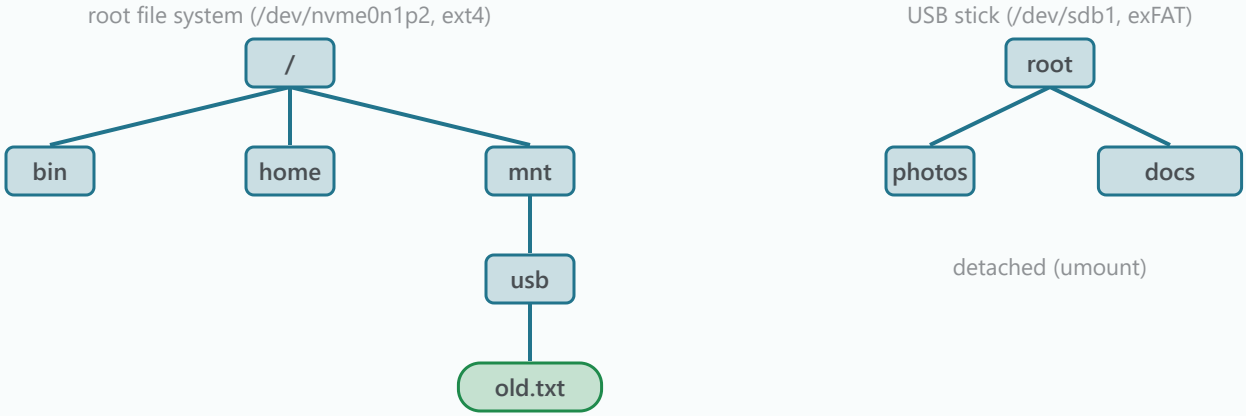
old.txt temporarily hidden

`mount /dev/sdb1 /mnt/usb` : the kernel adds an entry to the mount table; when path resolution reaches `/mnt/usb` it jumps to the USB's root i-node. The old contents are **hidden**, not deleted.

Step 2 / 3

At boot the root file system is mounted, the others via `/etc/fstab` or systemd units; `findmnt` shows the tree. Windows gives each volume a drive letter (`C:` , `D:`), but NTFS can also mount on an empty folder. Crossing a mount point applies permissions and options such as `nosuid` , `noexec` , `ro` .

Mounting



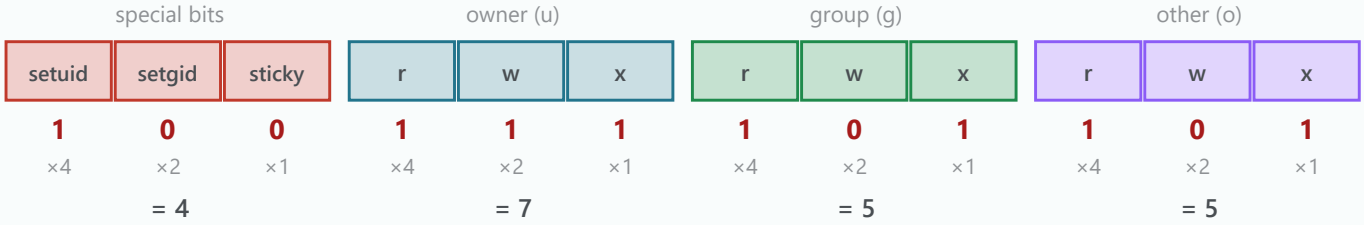
`umount /mnt/usb`: detached if no files are open (otherwise `EBUSY`); `old.txt` is visible again.

Step 3 / 3

At boot the root file system is mounted, the others via `/etc/fstab` or systemd units; `findmnt` shows the tree. Windows gives each volume a drive letter (`C:`, `D:`), but NTFS can also mount on an empty folder. Crossing a mount point applies permissions and options such as `nosuid`, `noexec`, `ro`.

PROTECTION

Access Rights: rwx



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

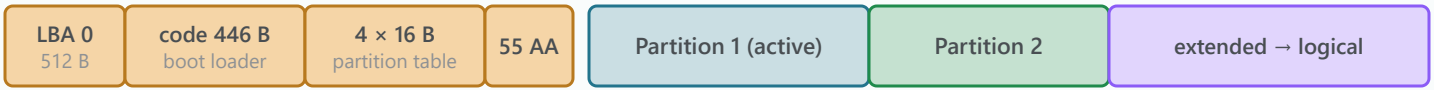
Right	On a file	On a directory
`r`	read the contents	list the entries (`ls`)
`w`	change the contents	add/delete/rename entries (the file's own permission doesn't matter!)
`x`	run as a program	pass through (path resolution, `cd`)

- Order of checks: *owner?* → owner rights; if not, *in the group?* → group rights; otherwise, other. The first match is used.
- **setuid**: the program runs with the identity of the file's owner (`passwd`). **sticky** on a directory: anyone can write, but only the owner can delete (`/tmp` : `drwxrwxrwt`). Example `drwxrwx--T` : sticky on, no `x` for others → a capital `T` .
- Owner/group/other is coarse-grained; for named users and groups an access-control list (ACL) is used, shown as `+` in `ls` output.

DISK LAYOUT

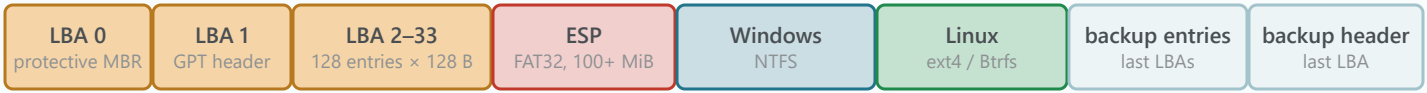
Disk Partitioning: MBR and GPT

MBR (BIOS, 1983)



at most 4 primary partitions; 32-bit LBA × 512 B ⇒ a 2 TiB limit

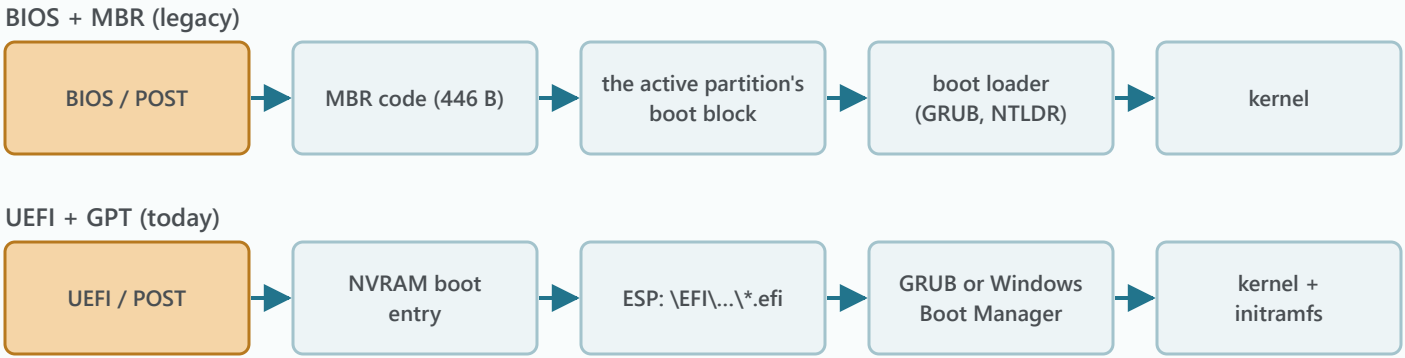
GPT (UEFI, 2000s)



64-bit LBA, 128 partitions, CRC32 checksums, a backup at the end of the disk; each partition has a GUID and a type GUID

- A disk is divided into one or more **partitions**; each can hold a separate file system (or swap space, LVM, an encrypted container).
- The partition table is at the end of the MBR: each entry holds a start and a length; the BIOS boots only from the partition marked *active*.
- **GPT** is part of the UEFI standard; it is required for disks over 2 TiB and for Windows 11. The protective MBR stops old tools from treating the disk as "empty".

Booting: BIOS/MBR and UEFI/GPT

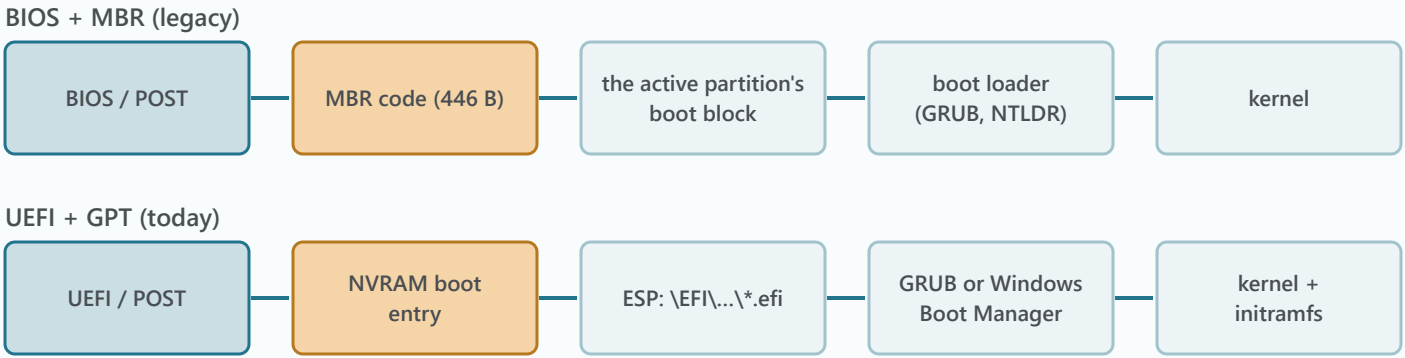


On power-up the hardware tests itself (POST). The BIOS runs in 16-bit real mode; UEFI in 32/64 bits, with its own drivers and file-system support.

Step 1 / 4

PCs have shipped with UEFI since around 2012; after 2020 the BIOS compatibility mode (CSM) was dropped from most new systems.

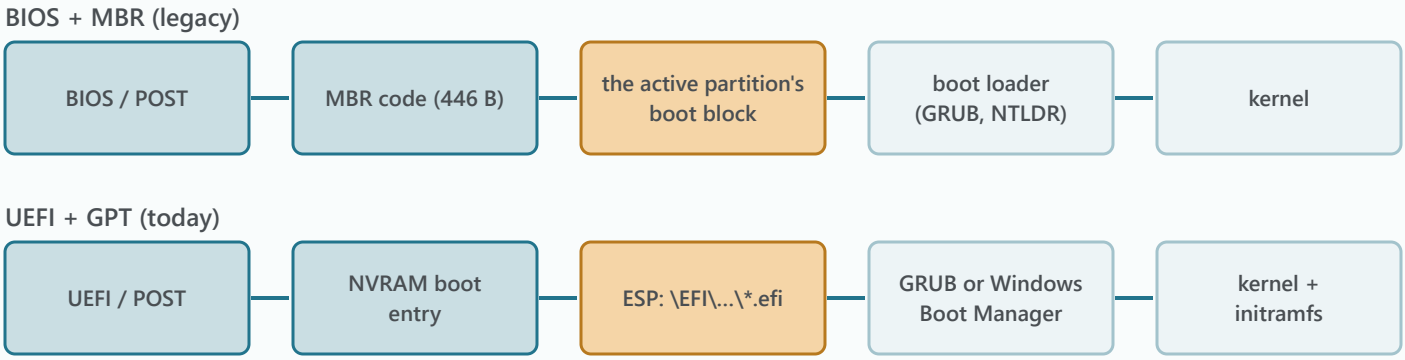
Booting: BIOS/MBR and UEFI/GPT



The BIOS loads the disk's **first sector** (the MBR) into memory and runs it. UEFI instead reads the boot entries in NVRAM (`efibootmgr`); it looks for a **file**, not a sector.

PCs have shipped with UEFI since around 2012; after 2020 the BIOS compatibility mode (CSM) was dropped from most new systems.

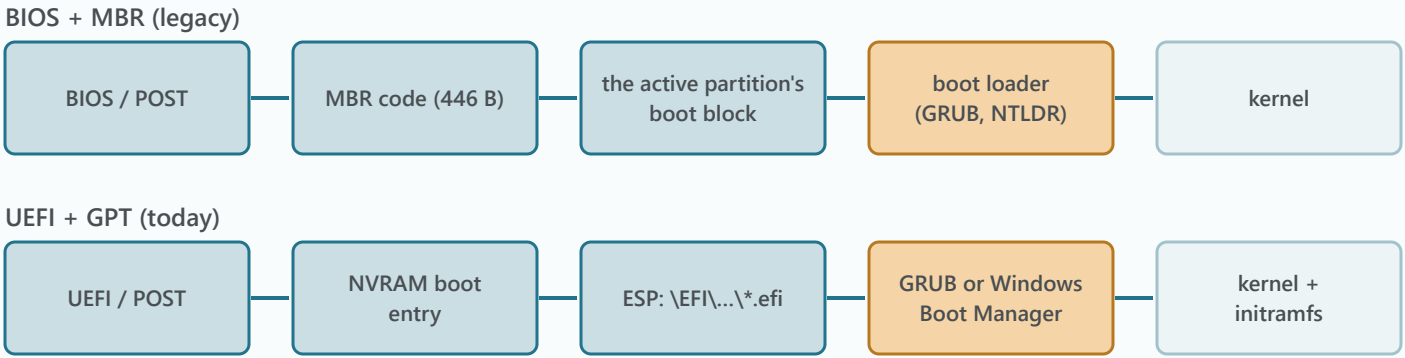
Booting: BIOS/MBR and UEFI/GPT



The MBR code finds the **active** partition in the partition table and runs its first block (the boot block). UEFI loads the `.efi` program from the ESP (FAT32); the active flag isn't used.

PCs have shipped with UEFI since around 2012; after 2020 the BIOS compatibility mode (CSM) was dropped from most new systems.

Booting: BIOS/MBR and UEFI/GPT

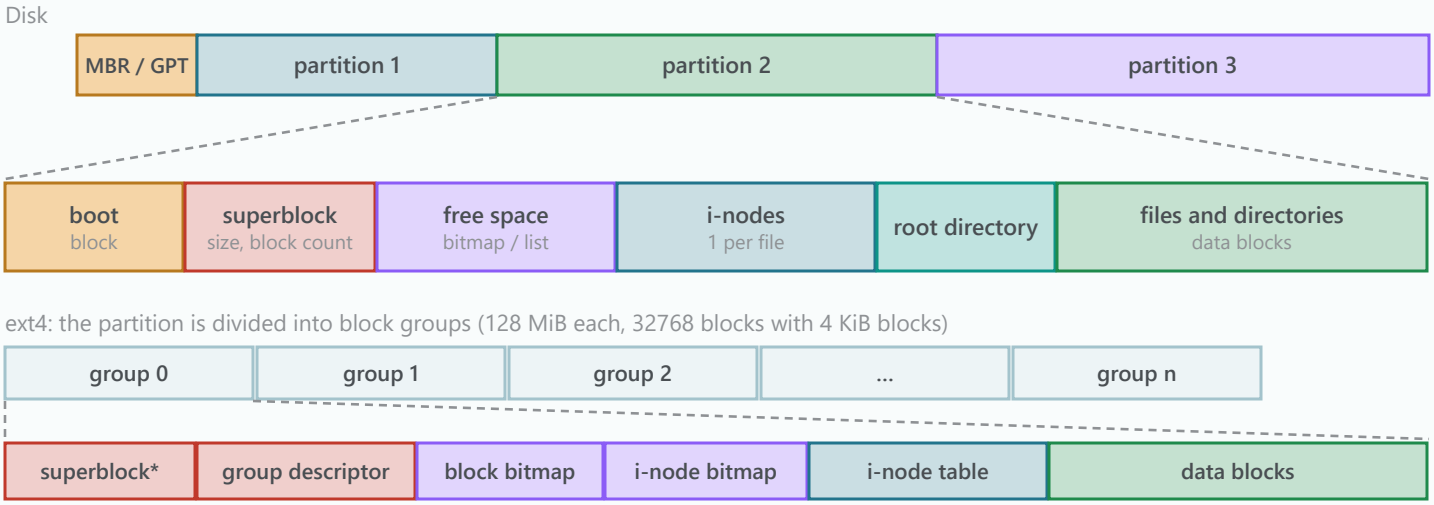


In both layouts the boot loader then loads and runs the kernel; the only difference is how the boot loader is reached.

PCs have shipped with UEFI since around 2012; after 2020 the BIOS compatibility mode (CSM) was dropped from most new systems.

DISK LAYOUT

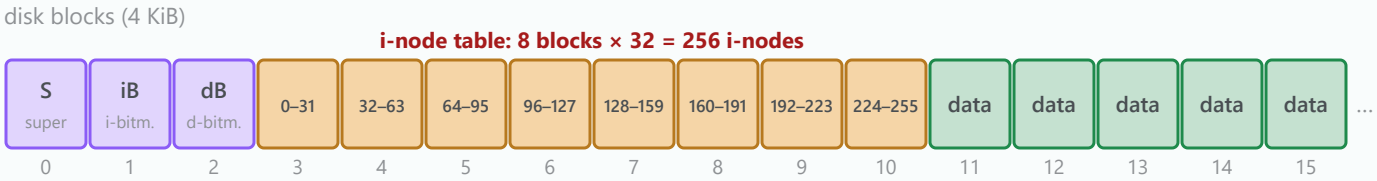
File-System Layout



* backups of the superblock and descriptors are kept in some groups

- **Superblock**: the file system's identity (magic number), block size, block and i-node counts, free counters, mount state. It is the first structure read when mounting.
- The boot block is the first block of the *partition* (the disk's first block is the MBR); space is reserved for it in every partition even if it is empty.
- ext4 places related i-nodes and their data in the same group to reduce head movement (the cylinder-group idea of BSD FFS).

From i-node Number to Disk Address

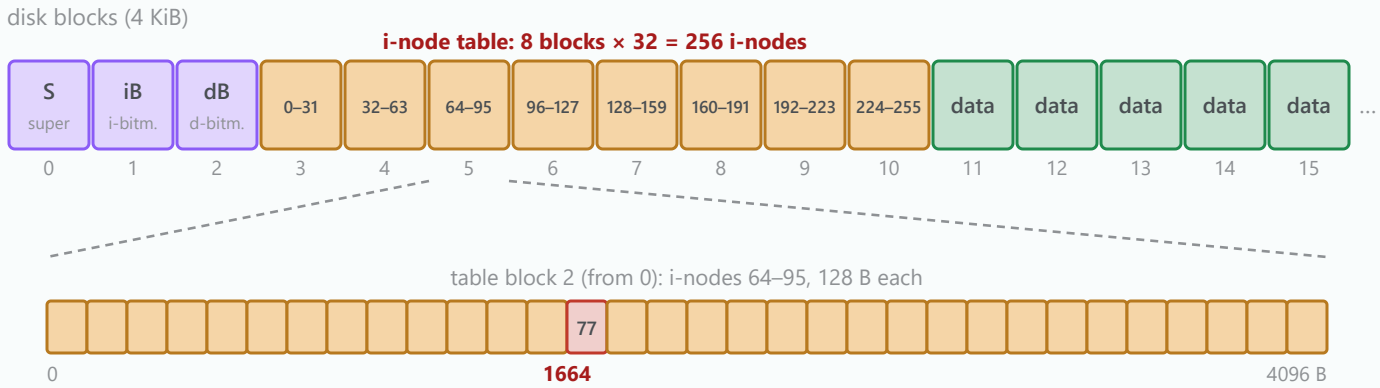


Example layout: 4 KiB blocks, 128 B i-nodes, i.e., **32 i-nodes** per block. Block 0 is the superblock, 1–2 the bitmaps, 3–10 the i-node table. Question: where on disk is **i-node 77** (numbered from 0)?

Step 1 / 4

In ext4 numbers start at 1 and the table is split across block groups: group = (n − 1) div g, index = (n − 1) mod g. With g = 8192 i-nodes/group and 256 B i-nodes, i-node 20,000 → group 2, index 3615 → block 225 of the group's table, offset 3840.

From i-node Number to Disk Address

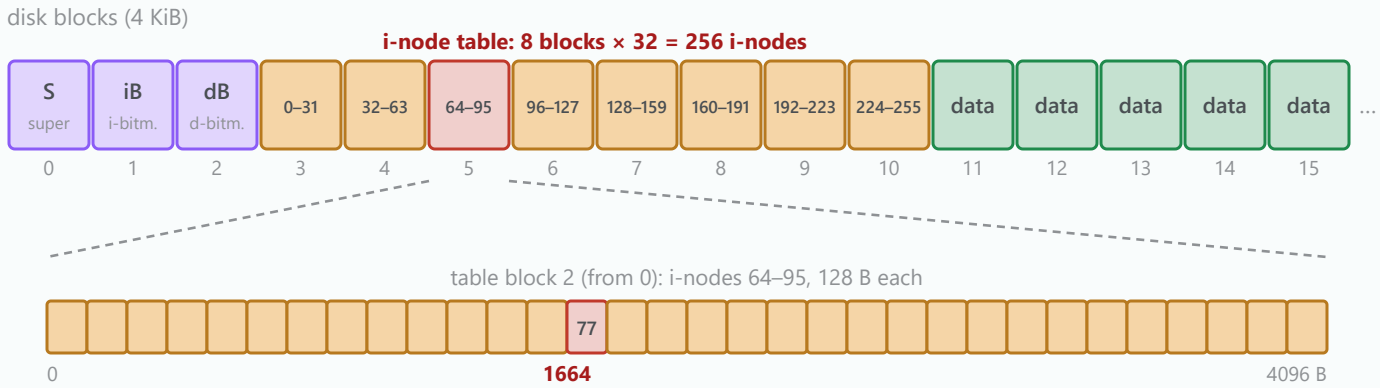


Byte offset from the start of the table: 77 × 128 = **9856** = 2 × 4096 + 1664. So **block 2 of the table**, byte **1664** within it (that block's 13th i-node).

Step 2 / 4

In ext4 numbers start at 1 and the table is split across block groups: group = (n − 1) div g, index = (n − 1) mod g. With g = 8192 i-nodes/group and 256 B i-nodes, i-node 20,000 → group 2, index 3615 → block 225 of the group's table, offset 3840.

From i-node Number to Disk Address

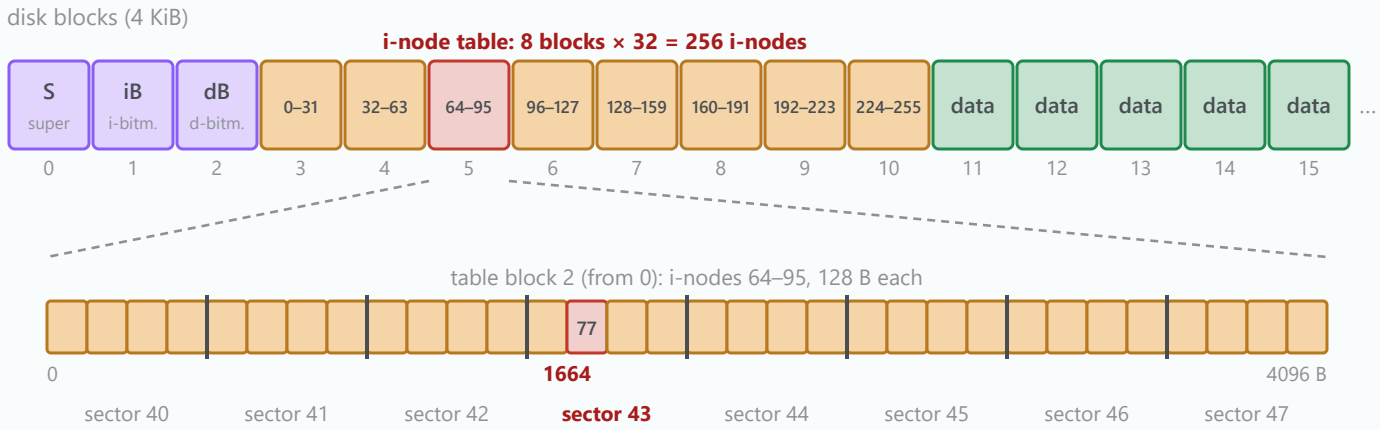


Disk block = table start + 2 = 3 + 2 = **block 5**. That's why the superblock's "where is the i-node table" field is needed.

Step 3 / 4

In ext4 numbers start at 1 and the table is split across block groups: group = (n − 1) div g, index = (n − 1) mod g. With g = 8192 i-nodes/group and 256 B i-nodes, i-node 20,000 → group 2, index 3615 → block 225 of the group's table, offset 3840.

From i-node Number to Disk Address

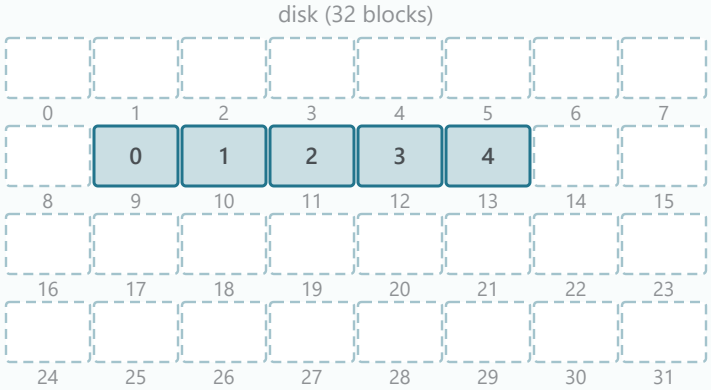


The disk reads sectors: byte address 5 × 4096 + 1664 = 22,144 → 22,144 / 512 = **sector 43**, byte 128 within it. A sector holds 4 i-nodes.

Step 4 / 4

In ext4 numbers start at 1 and the table is split across block groups: group = (n − 1) div g, index = (n − 1) mod g. With g = 8192 i-nodes/group and 256 B i-nodes, i-node 20,000 → group 2, index 3615 → block 225 of the group's table, offset 3840.

Allocation Methods: One File, Four Ways



directory entry

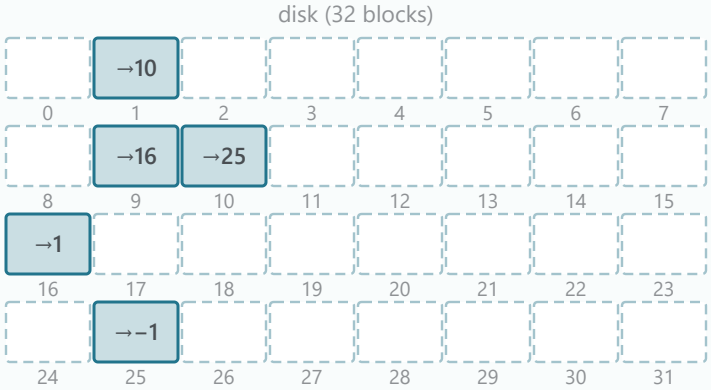
name	start	length
jeep	9	5

block i = start + i

one seek, sequential read

Contiguous: 5 blocks side by side (9–13). The directory entry holds only the start and length.

Allocation Methods: One File, Four Ways



directory entry

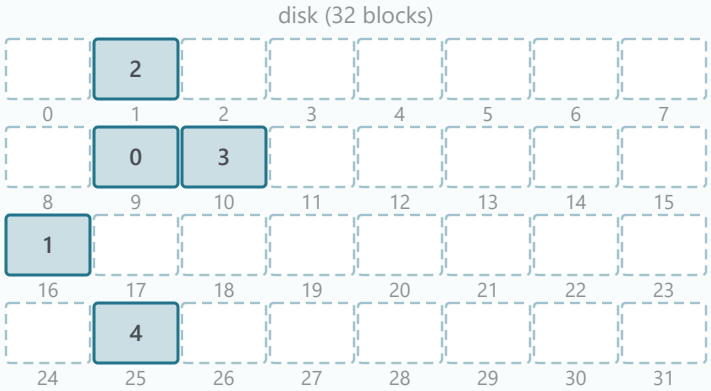
name	first	last
jeep	9	25

each block carries the number of the next

9 → 16 → 1 → 10 → 25 → −1

Linked list: the blocks are scattered over the disk (9, 16, 1, 10, 25); each points to the next.

Allocation Methods: One File, Four Ways



directory entry

name	first block	size
jeep	9	5 blocks

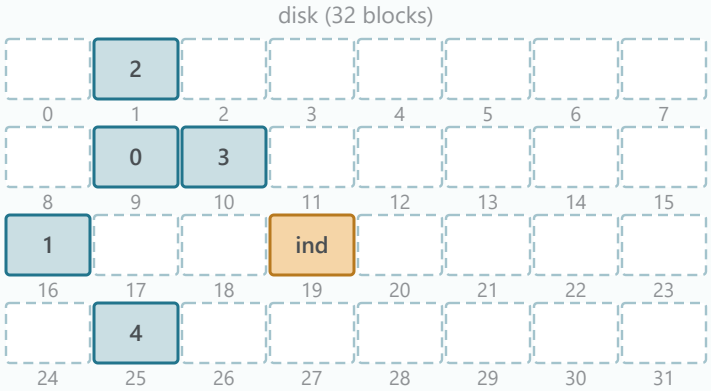
1	10
9	16
10	25
16	1
25	EOF

FAT (in memory)

chain in the table

FAT: the same chain, but the pointers live in a **table** in memory, not in the blocks; the blocks are all data.

Allocation Methods: One File, Four Ways



directory entry

name	index block	size
jeep	19	5 blocks

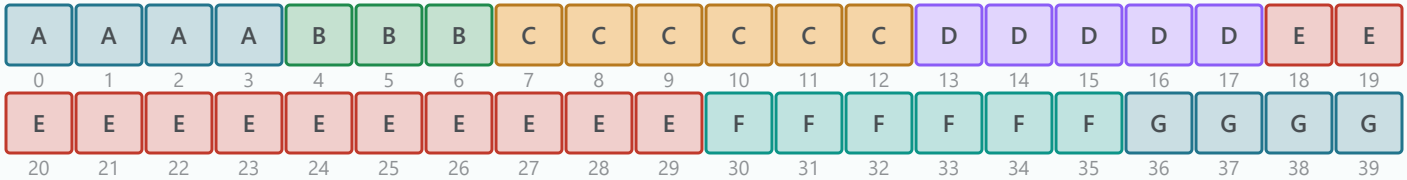
index block 19

9	16	1	10
25	−1	−1	

entry i = address of block i

Indexed: all addresses are in a single **index block** (19); block i is accessed directly.

Contiguous Allocation



7 files: A4 · B3 · C6 · D5 · E12 · F6 · G4 = 40 blocks, disk full

Each file was placed in consecutive blocks; directory entry = (start, length).

Step 1 / 4

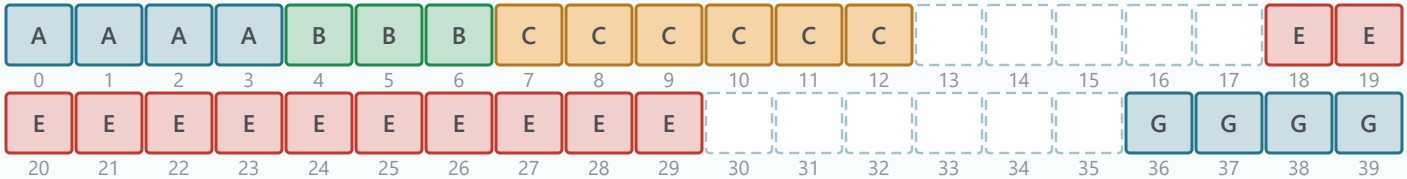
Pros

Simple; block i = start + i, so random access is easy; the whole file is read in one seek, back to back (the best HDD performance).

Cons

External fragmentation; files can't grow or must be moved. Suited to write-once media: CD-ROM/DVD (ISO 9660, UDF).

Contiguous Allocation



D and F deleted: two holes of 5 and 6 blocks (11 free blocks in total)

Deleted files leave **holes**; free space is broken into pieces.

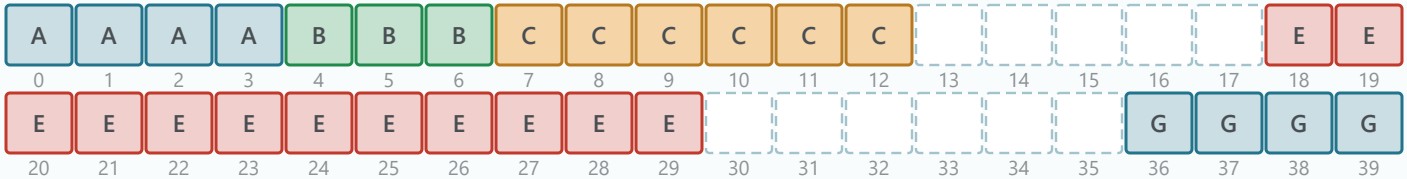
Pros

Simple; block i = start + i, so random access is easy; the whole file is read in one seek, back to back (the best HDD performance).

Cons

External fragmentation; files can't grow or must be moved. Suited to write-once media: CD-ROM/DVD (ISO 9660, UDF).

Contiguous Allocation



H wants 8 blocks: 11 free in total, but the largest hole is 6 → it doesn't fit!

external fragmentation

The new file fits no hole: the same external fragmentation as in memory management. Worse, the file's final size must be known up front.

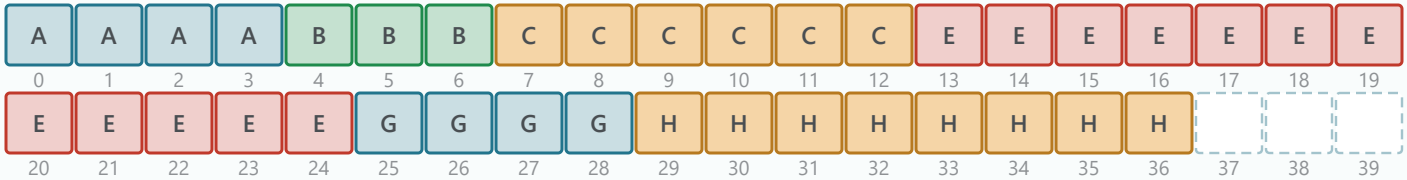
Pros

Simple; block i = start + i, so random access is easy; the whole file is read in one seek, back to back (the best HDD performance).

Cons

External fragmentation; files can't grow or must be moved. Suited to write-once media: CD-ROM/DVD (ISO 9660, UDF).

Contiguous Allocation



compaction: G and E moved, one 11-block hole → H placed

the cost: rewriting 16 blocks on disk

Remedies: defragmentation/compaction (expensive), or splitting the file into several contiguous pieces, **extents**.

Pros

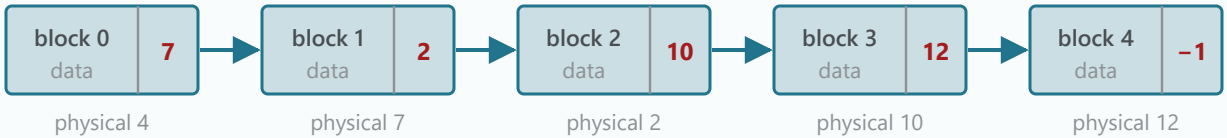
Simple; block i = start + i, so random access is easy; the whole file is read in one seek, back to back (the best HDD performance).

Cons

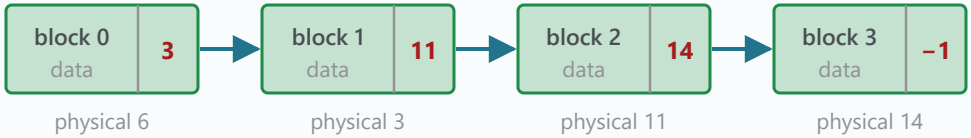
External fragmentation; files can't grow or must be moved. Suited to write-once media: CD-ROM/DVD (ISO 9660, UDF).

Linked Allocation

File A



File B

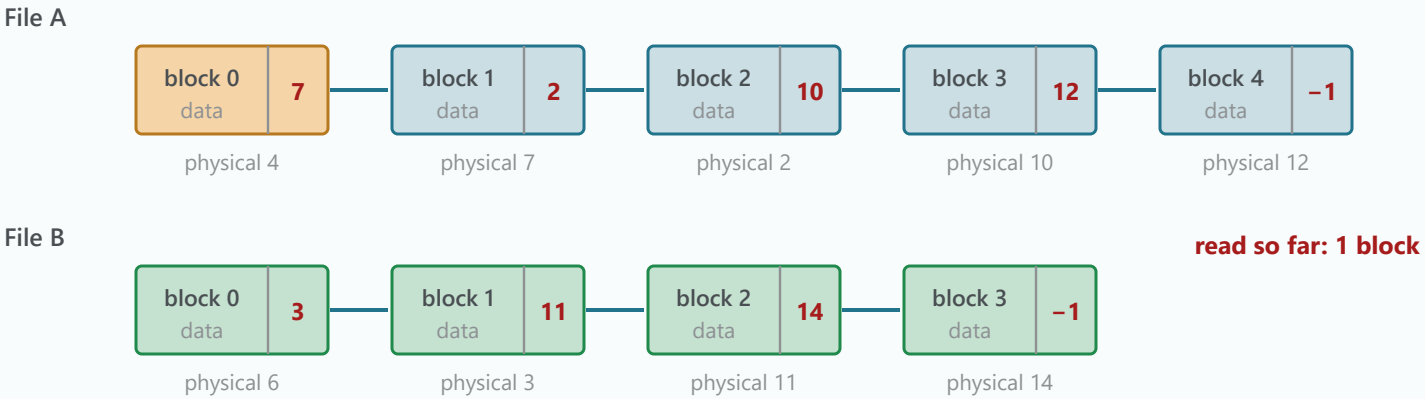


The directory entry knows the first block (4 for A) (and the last one too in Silberschatz's layout). The first/last few bytes of each block hold the next block's number.

Step 1 / 5

- **No** external fragmentation, and files grow easily; internal fragmentation (in the last block) exists with every method.
- The data per block isn't a power of two (4096 − 4 bytes); if one pointer is corrupted, the rest of the chain is lost.

Linked Allocation



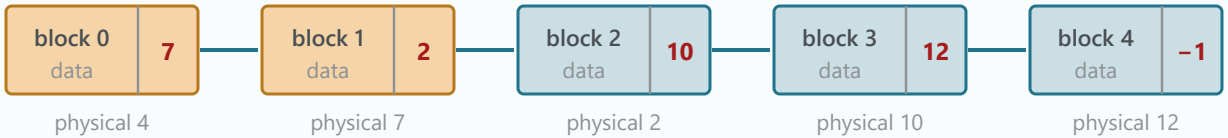
We want logical **block 3**: first physical 4 is read → next is 7.

Step 2 / 5

- **No** external fragmentation, and files grow easily; internal fragmentation (in the last block) exists with every method.
- The data per block isn't a power of two (4096 − 4 bytes); if one pointer is corrupted, the rest of the chain is lost.

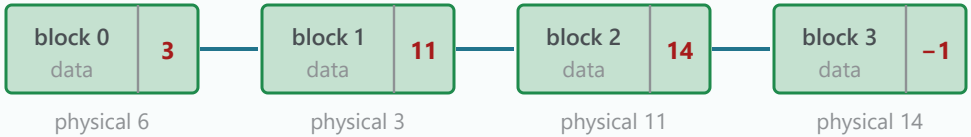
Linked Allocation

File A



File B

read so far: 2 blocks

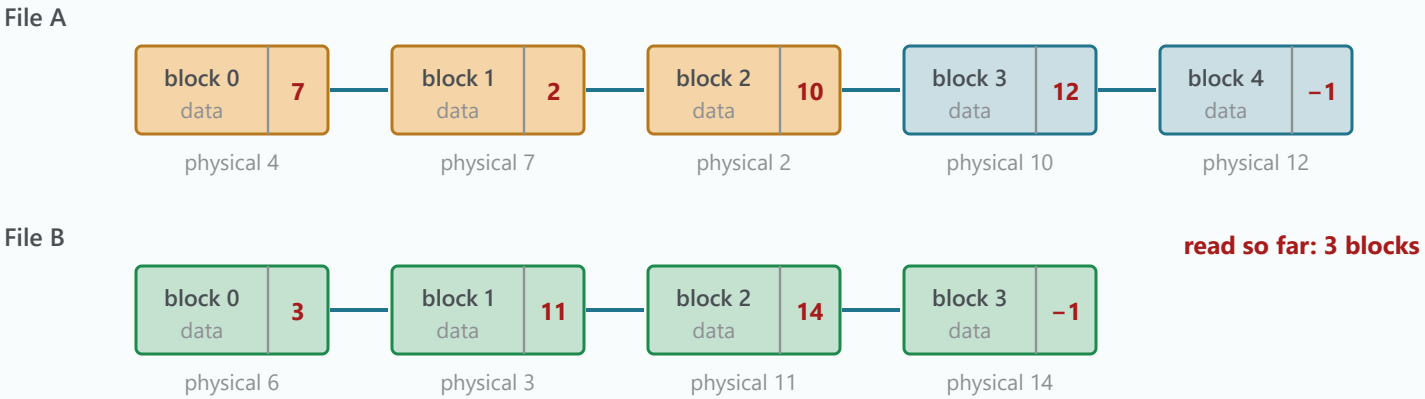


Physical 7 is read → next 2.

Step 3 / 5

- **No** external fragmentation, and files grow easily; internal fragmentation (in the last block) exists with every method.
- The data per block isn't a power of two (4096 − 4 bytes); if one pointer is corrupted, the rest of the chain is lost.

Linked Allocation

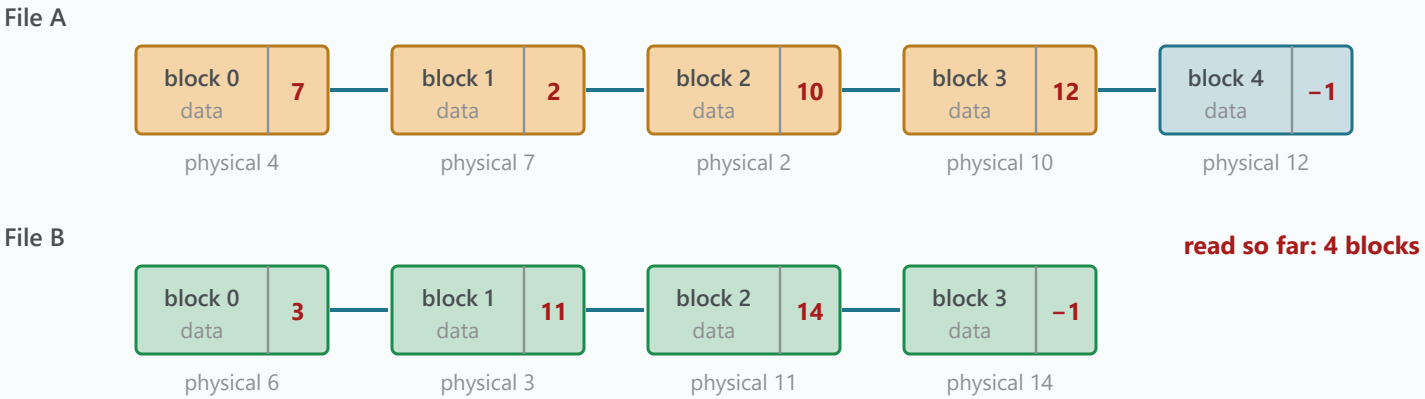


Physical 2 is read → next 10.

Step 4 / 5

- **No** external fragmentation, and files grow easily; internal fragmentation (in the last block) exists with every method.
- The data per block isn't a power of two (4096 − 4 bytes); if one pointer is corrupted, the rest of the chain is lost.

Linked Allocation

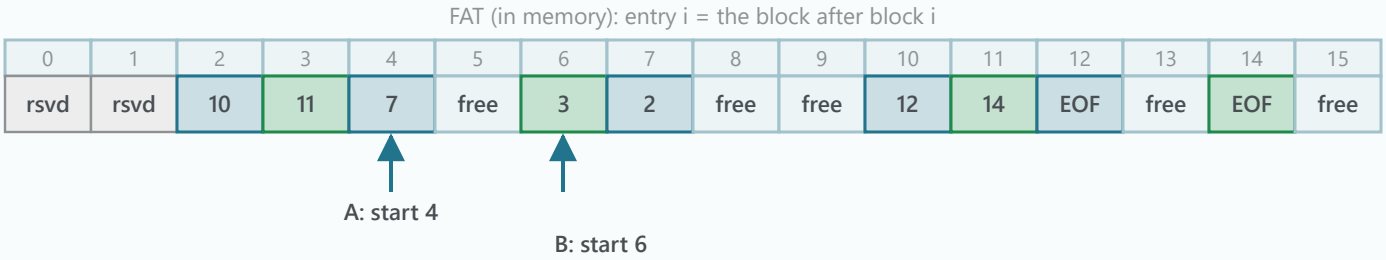


Physical 10 = logical 3: **4 disk reads**. Random access is O(n); on an HDD each may be a separate seek.

Step 5 / 5

- **No** external fragmentation, and files grow easily; internal fragmentation (in the last block) exists with every method.
- The data per block isn't a power of two (4096 − 4 bytes); if one pointer is corrupted, the rest of the chain is lost.

FAT: Allocation with a Table



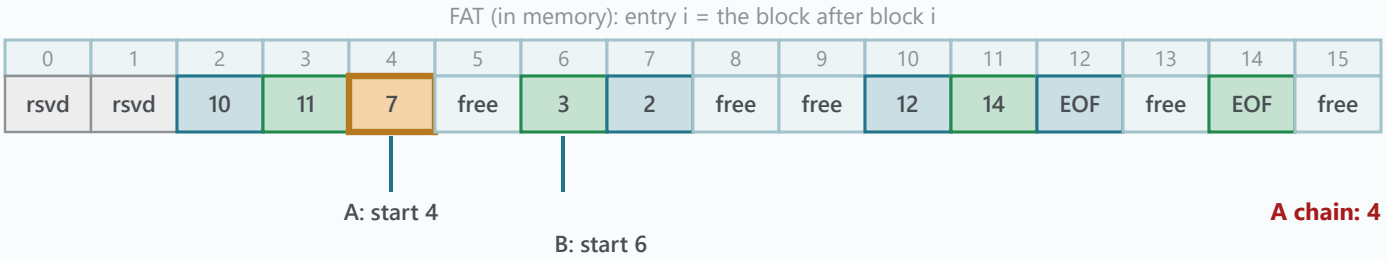
A: 4 → 7 → 2 → 10 → 12; B: 6 → 3 → 11 → 14.

Step 1 / 6

- The pointers of the linked list are taken out of the blocks and moved into one **file-allocation table** (FAT); the blocks are left entirely for data.
- Random access still follows the chain, but in memory: fast. Two copies of the table are kept on disk.
- **The problem:** the table grows with the disk, and all of it must be in memory.

Example: a 200 GiB disk with 1 KiB blocks → 200×2^{20} = **209,715,200** entries; at 4 B per entry → **800 MiB** (600 MiB at 3 B). In FAT32 a 2 TiB volume with 32 KiB clusters → 67,108,864 entries = 256 MiB.

FAT: Allocation with a Table



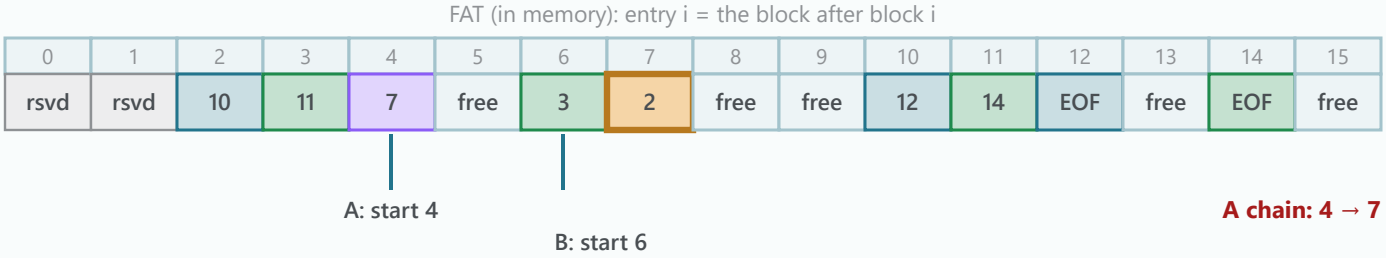
Entry 4 = 7.

Step 2 / 6

- The pointers of the linked list are taken out of the blocks and moved into one **file-allocation table** (FAT); the blocks are left entirely for data.
- Random access still follows the chain, but in memory: fast. Two copies of the table are kept on disk.
- **The problem:** the table grows with the disk, and all of it must be in memory.

Example: a 200 GiB disk with 1 KiB blocks → 200×2^{20} = **209,715,200** entries; at 4 B per entry → **800 MiB** (600 MiB at 3 B). In FAT32 a 2 TiB volume with 32 KiB clusters → 67,108,864 entries = 256 MiB.

FAT: Allocation with a Table



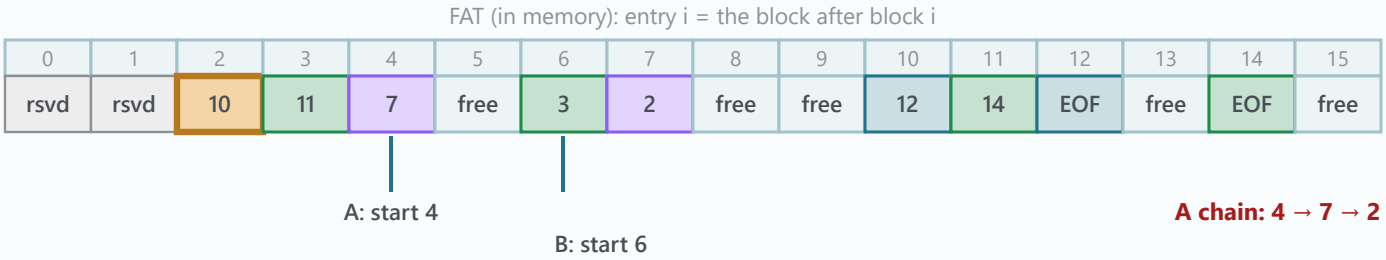
Entry 7 = 2.

Step 3 / 6

- The pointers of the linked list are taken out of the blocks and moved into one **file-allocation table** (FAT); the blocks are left entirely for data.
- Random access still follows the chain, but in memory: fast. Two copies of the table are kept on disk.
- **The problem:** the table grows with the disk, and all of it must be in memory.

Example: a 200 GiB disk with 1 KiB blocks → 200×2^{20} = **209,715,200** entries; at 4 B per entry → **800 MiB** (600 MiB at 3 B). In FAT32 a 2 TiB volume with 32 KiB clusters → 67,108,864 entries = 256 MiB.

FAT: Allocation with a Table



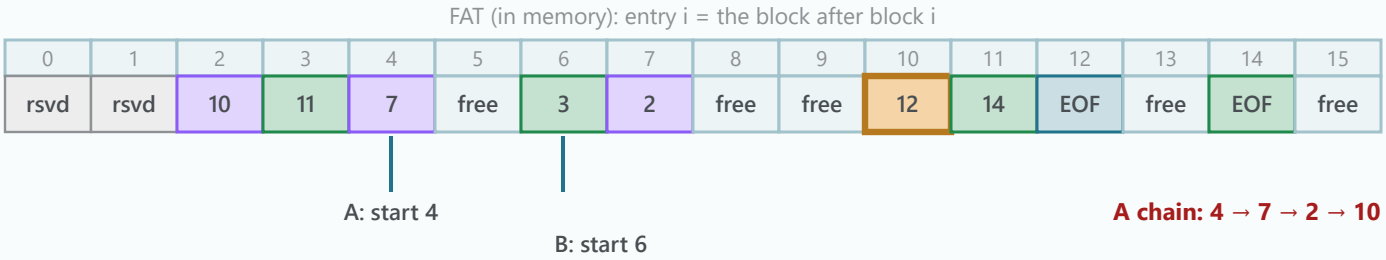
Entry 2 = 10.

Step 4 / 6

- The pointers of the linked list are taken out of the blocks and moved into one **file-allocation table** (FAT); the blocks are left entirely for data.
- Random access still follows the chain, but in memory: fast. Two copies of the table are kept on disk.
- **The problem:** the table grows with the disk, and all of it must be in memory.

Example: a 200 GiB disk with 1 KiB blocks → 200×2^{20} = **209,715,200** entries; at 4 B per entry → **800 MiB** (600 MiB at 3 B). In FAT32 a 2 TiB volume with 32 KiB clusters → 67,108,864 entries = 256 MiB.

FAT: Allocation with a Table



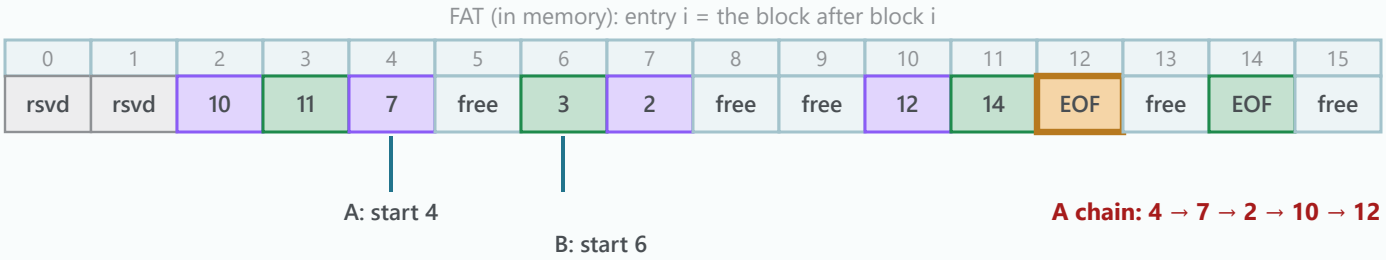
Entry 10 = 12.

Step 5 / 6

- The pointers of the linked list are taken out of the blocks and moved into one **file-allocation table** (FAT); the blocks are left entirely for data.
- Random access still follows the chain, but in memory: fast. Two copies of the table are kept on disk.
- **The problem:** the table grows with the disk, and all of it must be in memory.

Example: a 200 GiB disk with 1 KiB blocks → 200×2^{20} = **209,715,200** entries; at 4 B per entry → **800 MiB** (600 MiB at 3 B). In FAT32 a 2 TiB volume with 32 KiB clusters → 67,108,864 entries = 256 MiB.

FAT: Allocation with a Table



Entry 12 = EOF. The chain was followed **in memory**: no disk reads.

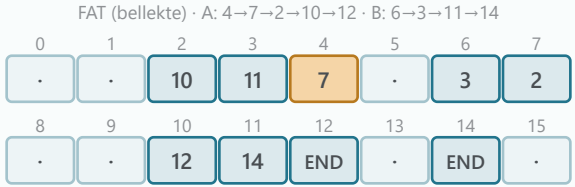
Step 6 / 6

- The pointers of the linked list are taken out of the blocks and moved into one **file-allocation table** (FAT); the blocks are left entirely for data.
- Random access still follows the chain, but in memory: fast. Two copies of the table are kept on disk.
- **The problem:** the table grows with the disk, and all of it must be in memory.

Example: a 200 GiB disk with 1 KiB blocks → 200×2^{20} = **209,715,200** entries; at 4 B per entry → **800 MiB** (600 MiB at 3 B). In FAT32 a 2 TiB volume with 32 KiB clusters → 67,108,864 entries = 256 MiB.

Following a FAT Chain: Code Trace

```
1  int find_block(int first, long off) {
▶    int b = first;
▶    long n = off / BLOCK;  /* block index */
4    while (n > 0 && b != END) {
5      b = fat[b];          /* follow chain */
6      n--;
7    }
8    return b;            /* END: past EOF */
9  }
```



`find_block(4, 3500)` : file A starts at block 4. 3500 / 1024 = **3**: we want the file's block 3 (counting from 0), so the chain must advance 3 steps.

Step 1 / 11

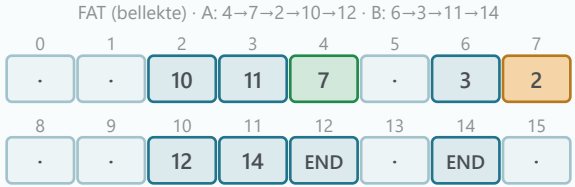
VARIABLES

`first = 4` `off = 3500` `n = 3` `b = 4`

OUTPUT

Following a FAT Chain: Code Trace

```
1 int find_block(int first, long off) {
2   int b = first;
3   long n = off / BLOCK;  /* block index */
▶   while (n > 0 && b != END) {
▶     b = fat[b];          /* follow chain */
▶     n--;
7   }
8   return b;            /* END: past EOF */
9 }
```



fat[4] = 7: next block 7; n = 2.

Step 2 / 11

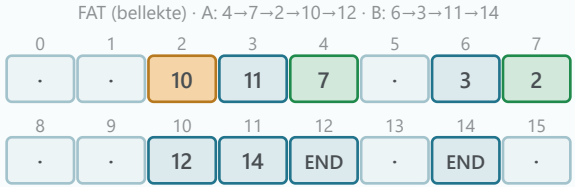
VARIABLES

`first = 4` `off = 3500` `n = 2` `b = 7`

OUTPUT

Following a FAT Chain: Code Trace

```
1 int find_block(int first, long off) {
2   int b = first;
3   long n = off / BLOCK;  /* block index */
▶   while (n > 0 && b != END) {
▶     b = fat[b];          /* follow chain */
▶     n--;
7   }
8   return b;            /* END: past EOF */
9 }
```



fat[7] = 2: next block 2; n = 1.

Step 3 / 11

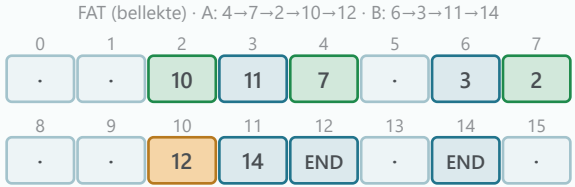
VARIABLES

`first = 4` `off = 3500` `n = 1` `b = 2`

OUTPUT

Following a FAT Chain: Code Trace

```
1 int find_block(int first, long off) {
2   int b = first;
3   long n = off / BLOCK;  /* block index */
▶   while (n > 0 && b != END) {
▶     b = fat[b];          /* follow chain */
▶     n--;
7   }
8   return b;            /* END: past EOF */
9 }
```



followed: 4 → 7 → 2 → 10

blue: file blocks · green: followed · orange: b

fat[2] = 10: next block 10; n = 0.

Step 4 / 11

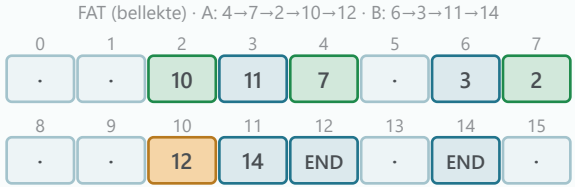
VARIABLES

`first = 4` `off = 3500` `n = 0` `b = 10`

OUTPUT

Following a FAT Chain: Code Trace

```
1  int find_block(int first, long off) {
2    int b = first;
3    long n = off / BLOCK;  /* block index */
4    while (n > 0 && b != END) {
5      b = fat[b];          /* follow chain */
6      n--;
7    }
▶    return b;            /* END: past EOF */
9  }
```



followed: 4 → 7 → 2 → 10

blue: file blocks · green: followed · orange: b

n = 0: the loop ends at disk block **10**; byte 3500 is byte 3500 − 3·1024 = 428 of this block. All three steps were in memory: no disk reads.

Step 5 / 11

VARIABLES

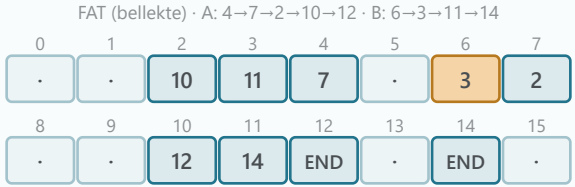
`first = 4` `off = 3500` `n = 0` `b = 10`

OUTPUT

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

Following a FAT Chain: Code Trace

```
1 int find_block(int first, long off) {
▶   int b = first;
▶   long n = off / BLOCK;  /* block index */
4   while (n > 0 && b != END) {
5     b = fat[b];          /* follow chain */
6     n--;
7   }
8   return b;            /* END: past EOF */
9 }
```



`find_block(6, 5000)` : file B has 4 blocks (at most 4096 bytes), but 5000 / 1024 = block **4** is requested.

Step 6 / 11

VARIABLES

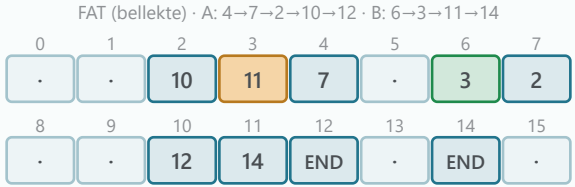
`first = 6` `off = 5000` `n = 4` `b = 6`

OUTPUT

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

Following a FAT Chain: Code Trace

```
1 int find_block(int first, long off) {
2   int b = first;
3   long n = off / BLOCK;  /* block index */
▶   while (n > 0 && b != END) {
▶     b = fat[b];          /* follow chain */
▶     n--;
7   }
8   return b;            /* END: past EOF */
9 }
```



followed: 6 → 3

blue: file blocks · green: followed · orange: b

fat[6] = 3: next block 3; n = 3.

Step 7 / 11

VARIABLES

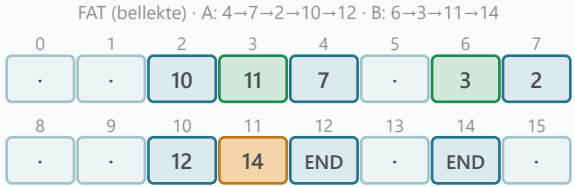
`first = 6` `off = 5000` `n = 3` `b = 3`

OUTPUT

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

Following a FAT Chain: Code Trace

```
1 int find_block(int first, long off) {
2   int b = first;
3   long n = off / BLOCK;  /* block index */
▶   while (n > 0 && b != END) {
▶     b = fat[b];          /* follow chain */
▶     n--;
7   }
8   return b;            /* END: past EOF */
9 }
```



followed: 6 → 3 → 11

blue: file blocks · green: followed · orange: b

fat[3] = 11: next block 11; n = 2.

Step 8 / 11

VARIABLES

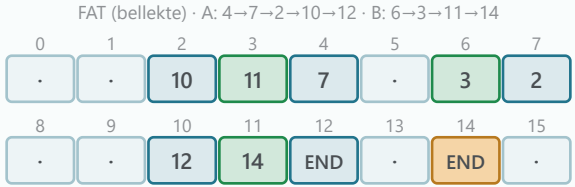
`first = 6` `off = 5000` `n = 2` `b = 11`

OUTPUT

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

Following a FAT Chain: Code Trace

```
1 int find_block(int first, long off) {
2   int b = first;
3   long n = off / BLOCK;  /* block index */
▶   while (n > 0 && b != END) {
▶     b = fat[b];          /* follow chain */
▶     n--;
7   }
8   return b;            /* END: past EOF */
9 }
```



followed: 6 → 3 → 11 → 14

blue: file blocks · green: followed · orange: b

fat[11] = 14: next block 14; n = 1.

Step 9 / 11

VARIABLES

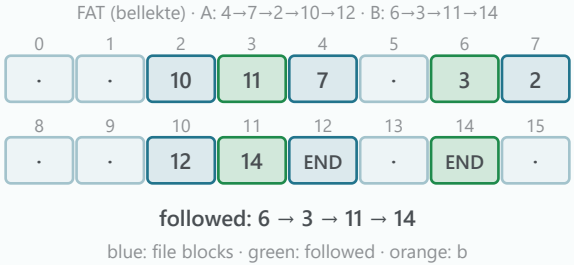
`first = 6` `off = 5000` `n = 1` `b = 14`

OUTPUT

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

Following a FAT Chain: Code Trace

```
1 int find_block(int first, long off) {
2   int b = first;
3   long n = off / BLOCK;  /* block index */
▶   while (n > 0 && b != END) {
▶     b = fat[b];          /* follow chain */
▶     n--;
7   }
8   return b;            /* END: past EOF */
9 }
```



fat[14] = END: the chain is over and the end of the file reached; n = 0.

Step 10 / 11

VARIABLES

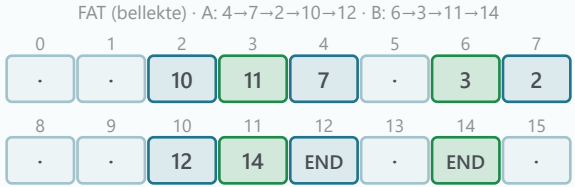
`first = 6` `off = 5000` `n = 0` `b = END`

OUTPUT

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

Following a FAT Chain: Code Trace

```
1  int find_block(int first, long off) {
2    int b = first;
3    long n = off / BLOCK;  /* block index */
4    while (n > 0 && b != END) {
5      b = fat[b];          /* follow chain */
6      n--;
7    }
▶    return b;            /* END: past EOF */
9  }
```



followed: 6 → 3 → 11 → 14

blue: file blocks · green: followed · orange: b

At the fourth step b = END: the file has no block 4, and the offset lies past the file. `read` returns 0 bytes (end of file); `write` instead allocates a new block and links it into entry 14.

Step 11 / 11

VARIABLES

`first = 6` `off = 5000` `n = 0` `b = END`

OUTPUT

```
find_block(4, 3500) = 10
find_block(6, 5000) = END
```

The FAT Family: FAT12/16/32 and exFAT

MS-DOS directory entry (32 bytes)



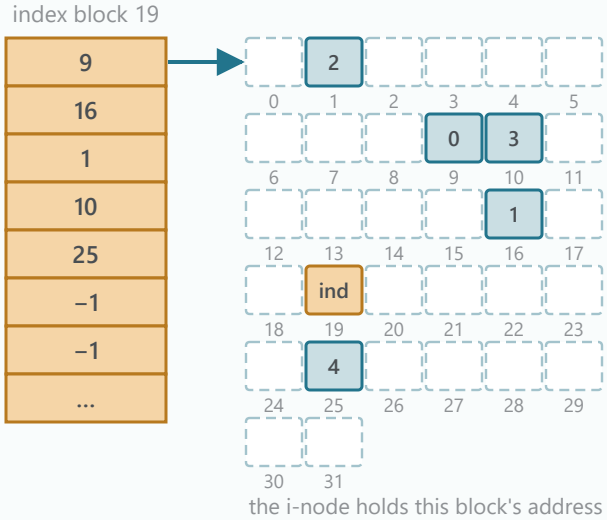
Attr: attributes. FAT32 put creation/access times and the upper 16 bits of the first cluster into the "reserved" area; long names live in extra entries.

	FAT12	FAT16	FAT32	exFAT
FAT entry	12 bit	16 bit	28 bits used	32 bit
Largest volume	≈ 16 MiB	2 GiB (32 KiB clusters)	2 TiB (512 B sectors)	128 PiB
Largest file	size of the volume	2 GiB	**4 GiB − 1 bytes**	16 EiB (theoretical)
Free space	0 in the FAT	0 in the FAT	0 in the FAT + FSInfo hint	separate **bitmap**
Used for	floppies	old disks	USB, UEFI ESP	SDXC/SDUC, large USB drives

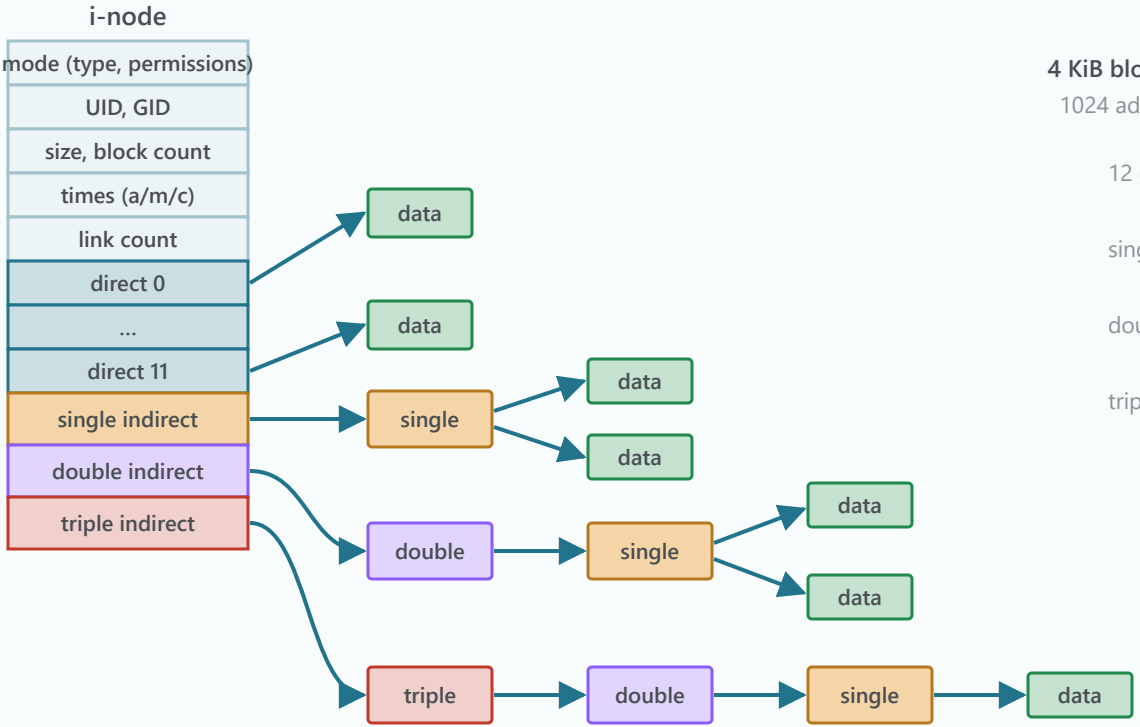
The size field is 32 bits, so FAT32 has a 4 GiB file-size limit; exFAT removes it and may skip the FAT chain entirely for contiguous files. Windows' graphical format tool has historically limited FAT32 to 32 GB; that's a limit of the tool, not the file system.

Indexed Allocation and the i-node

- Each file's block addresses are gathered in an **index block** (on UNIX, the **i-node**): entry i is the address of block i. Fast random access, no external fragmentation.
- An i-node is in memory only while the file is **open**: K open files × i-node size. Proportional to the number of open files, not to the disk size as with FAT.
- An i-node has a **fixed size**; for very large files the last entries point to further index blocks (indirect blocks).
- The cost: even a small file uses index space; a large file needs indirect-block reads (caching reduces this).



i-node Structure: Indirect Blocks



4 KiB blocks, 4 B pointers:
1024 addresses per block

12 direct	**48 KiB**
single indirect	**+ 4 MiB**
double indirect	**+ 4 GiB**
triple indirect	**+ 4 TiB**

each level ×1024

The ext2/ext3 and classic UNIX (V7: 10 direct) structure. Small files use only the direct pointers and never allocate an indirect block; as a file grows, the tree deepens.

Maximum File Size

Assumptions: **4 KiB** blocks = 4096 B, **4 B** pointers → an indirect block holds 4096 / 4 = **1024** addresses; the i-node has **12 direct**, 1 single, 1 double, 1 triple indirect pointer.

Direct: 12 × 4 KiB = **48 KiB** (49,152 B) · Single indirect: 1024 × 4 KiB = **4 MiB**

The first 12 blocks come straight from the i-node; the next 1024 from one index block.

Tanenbaum's example (1 KiB blocks, 256 addresses, 10 direct): 16,843,018 blocks ≈ 16 GiB. ext2/ext3 with 4 KiB blocks actually stop at **2 TiB** (32-bit counter × 512 B); ext4 extents reach 16 TiB.

Maximum File Size

Assumptions: **4 KiB** blocks = 4096 B, **4 B** pointers → an indirect block holds 4096 / 4 = **1024** addresses; the i-node has **12 direct**, 1 single, 1 double, 1 triple indirect pointer.

Direct: 12 × 4 KiB = **48 KiB** (49,152 B) · Single indirect: 1024 × 4 KiB = **4 MiB**

The first 12 blocks come straight from the i-node; the next 1024 from one index block.

Double indirect: 1024^2 × 4 KiB = 1,048,576 blocks = **4 GiB**

Each of 1024 index blocks points to 1024 data blocks.

Tanenbaum's example (1 KiB blocks, 256 addresses, 10 direct): 16,843,018 blocks ≈ 16 GiB. ext2/ext3 with 4 KiB blocks actually stop at **2 TiB** (32-bit counter × 512 B); ext4 extents reach 16 TiB.

Maximum File Size

Assumptions: **4 KiB** blocks = 4096 B, **4 B** pointers → an indirect block holds 4096 / 4 = **1024** addresses; the i-node has **12 direct**, 1 single, 1 double, 1 triple indirect pointer.

Direct: 12 × 4 KiB = **48 KiB** (49,152 B) · Single indirect: 1024 × 4 KiB = **4 MiB**

The first 12 blocks come straight from the i-node; the next 1024 from one index block.

Double indirect: 1024^2 × 4 KiB = 1,048,576 blocks = **4 GiB**

Each of 1024 index blocks points to 1024 data blocks.

Triple indirect: 1024^3 × 4 KiB = 1,073,741,824 blocks = **4 TiB**

One more level: ×1024.

Tanenbaum's example (1 KiB blocks, 256 addresses, 10 direct): 16,843,018 blocks ≈ 16 GiB. ext2/ext3 with 4 KiB blocks actually stop at **2 TiB** (32-bit counter × 512 B); ext4 extents reach 16 TiB.

Maximum File Size

Assumptions: **4 KiB** blocks = 4096 B, **4 B** pointers → an indirect block holds 4096 / 4 = **1024** addresses; the i-node has **12 direct**, 1 single, 1 double, 1 triple indirect pointer.

Direct: 12 × 4 KiB = **48 KiB** (49,152 B) · Single indirect: 1024 × 4 KiB = **4 MiB**

The first 12 blocks come straight from the i-node; the next 1024 from one index block.

Double indirect: 1024^2 × 4 KiB = 1,048,576 blocks = **4 GiB**

Each of 1024 index blocks points to 1024 data blocks.

Triple indirect: 1024^3 × 4 KiB = 1,073,741,824 blocks = **4 TiB**

One more level: ×1024.

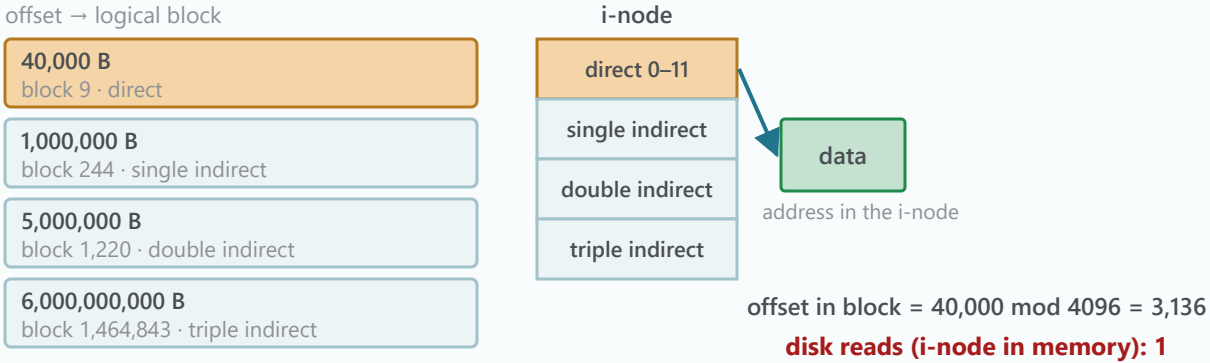
Total: 12 + 1024 + 1024^2 + 1024^3 = **1,074,791,436** blocks × 4096 = **4,402,345,721,856 B ≈ 4.004 TiB**

Nearly all of it comes from the triple indirect block.

Step 4 / 4

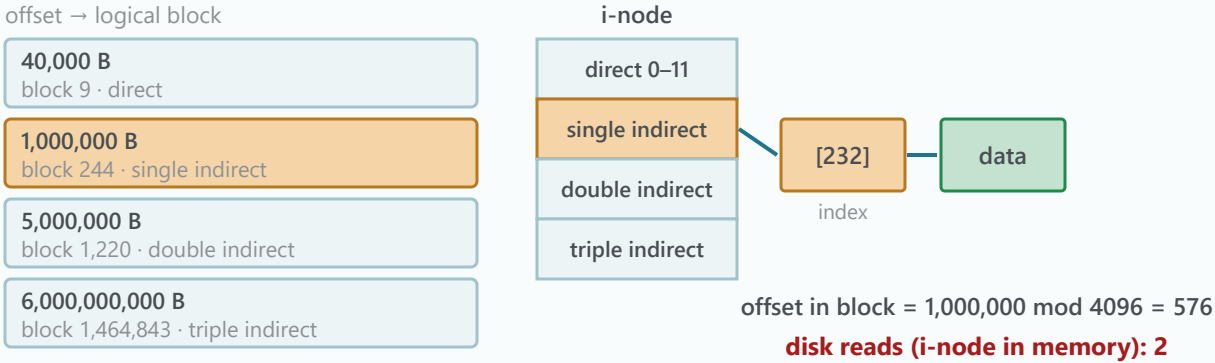
Tanenbaum's example (1 KiB blocks, 256 addresses, 10 direct): 16,843,018 blocks ≈ 16 GiB. ext2/ext3 with 4 KiB blocks actually stop at **2 TiB** (32-bit counter × 512 B); ext4 extents reach 16 TiB.

From Offset to Disk Block



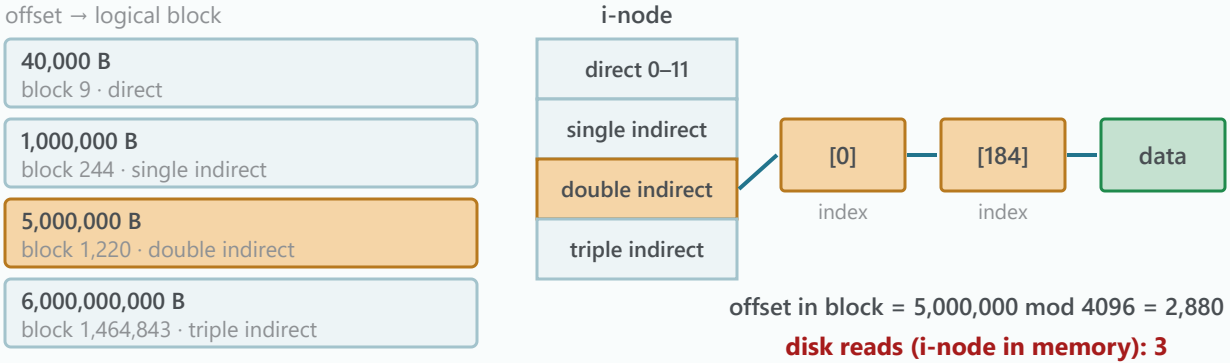
40,000 / 4096 = **9** → below 12: direct pointer 9. One read.

From Offset to Disk Block



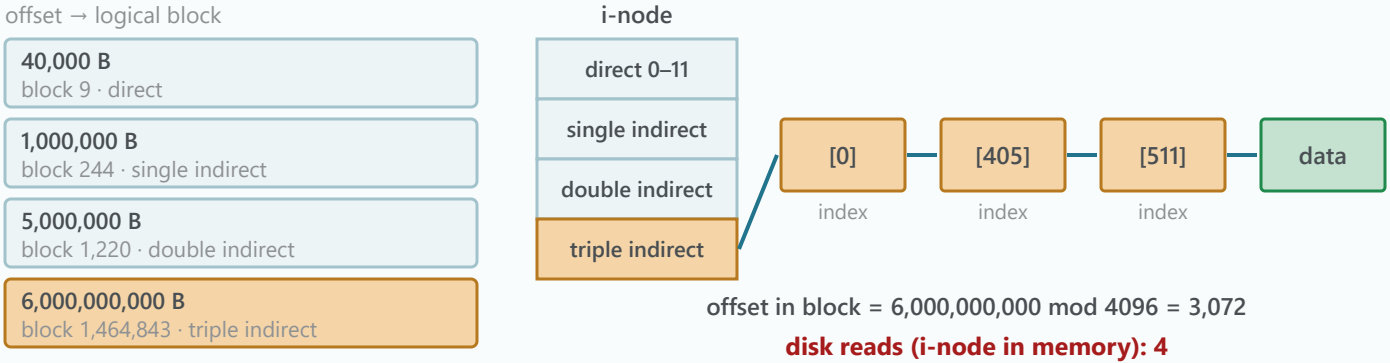
1,000,000 / 4096 = 244; 244 − 12 = **232** < 1024 → entry 232 of the single indirect block. 2 reads.

From Offset to Disk Block



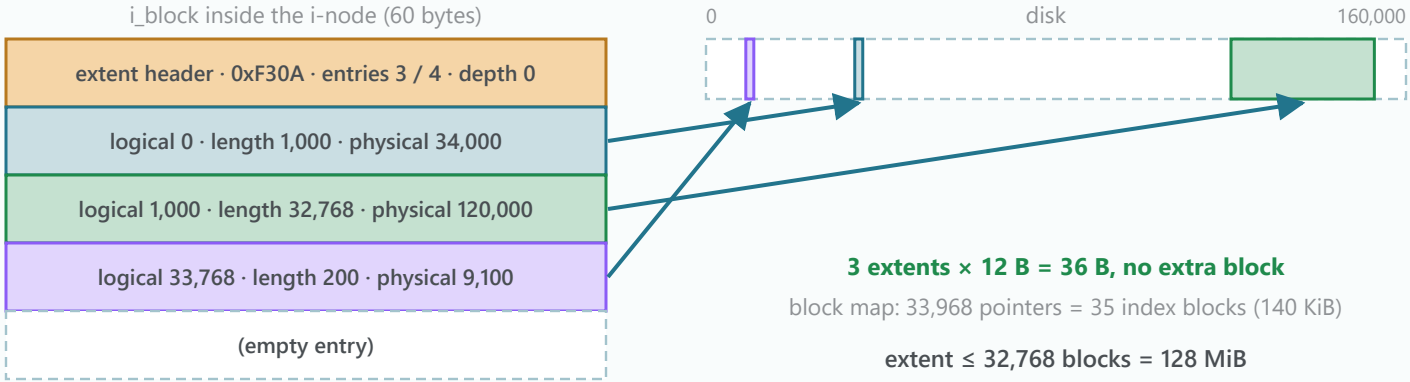
5,000,000 / 4096 = 1220; 1220 − 12 − 1024 = 184 < 1024^2 → double indirect: outer index 184 div 1024 = **0**, inner index **184**. 3 reads.

From Offset to Disk Block



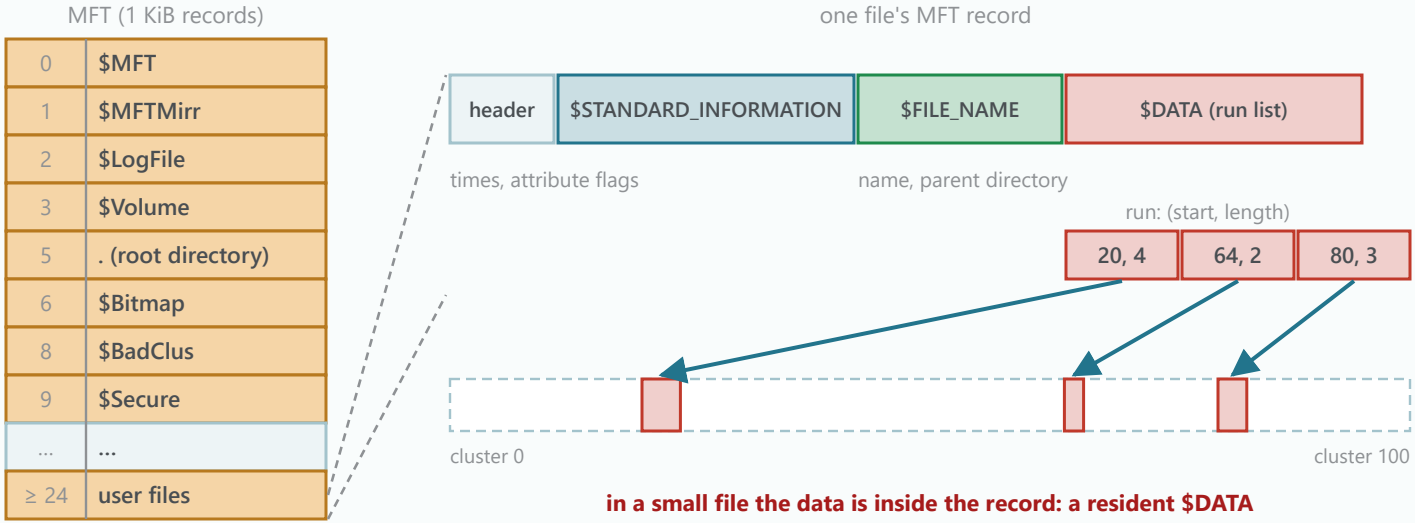
6 GB / 4096 = 1,464,843; minus 12 + 1024 + 1024^2 leaves 415,231 → triple indirect: **0, 405, 511**. 4 reads; the cache holds the first levels.

ext4: Extents



- An **extent** = (logical start, length, physical start): a run of contiguous blocks described by one record. 48-bit physical addresses → 1 EiB volumes; files up to **16 TiB**.
- Four extents fit in the i-node; if more are needed, the entries point to index nodes of an **extent tree** (depth > 0).
- Delayed allocation and the multiblock allocator choose large contiguous areas when writing data to disk. XFS, Btrfs, NTFS (run lists), and APFS use extents too.

NTFS: The Master File Table (MFT)



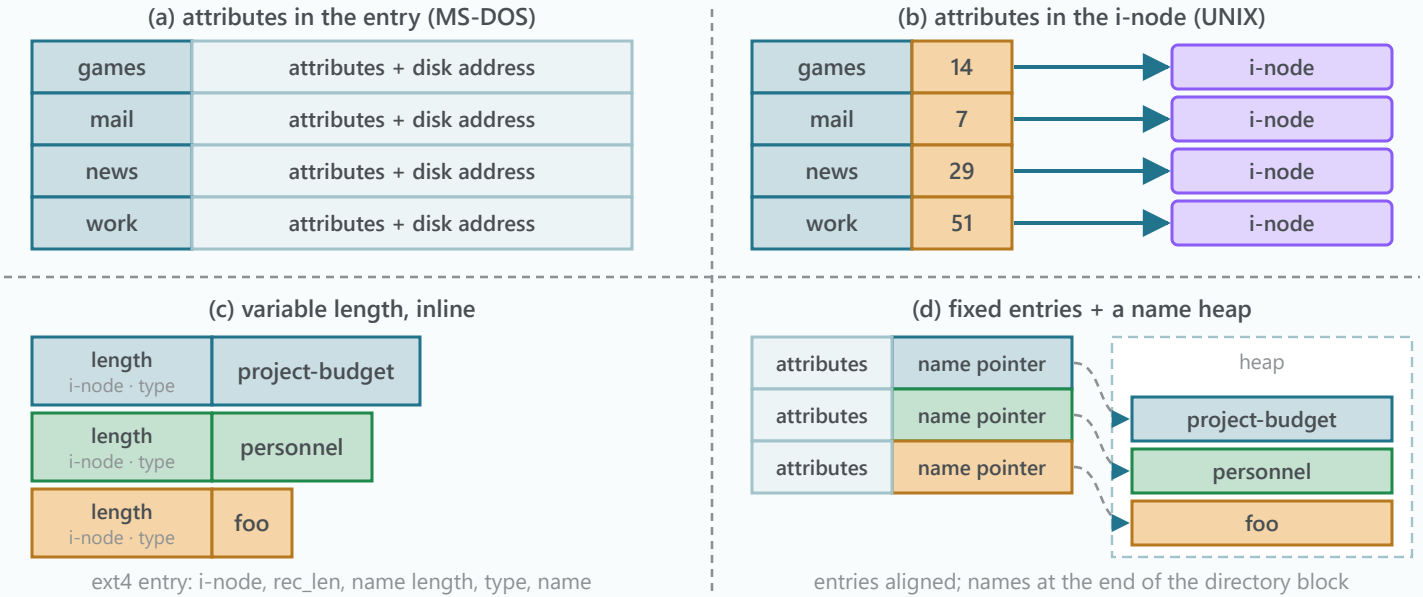
- In NTFS **everything is a file**, including the MFT itself (record 0). Every file is at least one 1 KiB record whose contents are a list of *attributes*.
- Large data is kept as a *run list* (extents); directories are B+ trees (`$INDEX_ROOT` , `$INDEX_ALLOCATION`). `$LogFile` is the metadata journal.
- Extra features: hard links, reparse points, alternate data streams (ADS), compression, EFS encryption, quotas. Servers also have ReFS (CoW, checksums).

Comparing File Systems

File system	Largest file	Largest volume	Crash consistency	Typical use
FAT32	4 GiB − 1 B	2 TiB	none (chkdsk/fsck)	USB, UEFI ESP
exFAT	16 EiB*	128 PiB	none	SD cards, external disks
NTFS	16 EiB* (Windows: 256 TiB–8 PiB)	256 TiB–8 PiB	metadata journal	Windows
ext4	16 TiB (4 KiB blocks)	1 EiB	journal (jbd2)	Linux, Android
XFS	8 EiB	8 EiB	metadata journal	RHEL, large servers
Btrfs	16 EiB	16 EiB	CoW + checksums	Fedora, openSUSE, NAS
ZFS	16 EiB	2^{128} B (theoretical)	CoW + checksums	FreeBSD, TrueNAS, servers
APFS	8 EiB	8 EiB	CoW, snapshots	macOS, iOS

* theoretical limit; the OS and cluster size impose lower ones. Since Windows 10 (1709), NTFS reaches 8 PiB with 2 MiB clusters. HFS+ gave way to APFS in 2017; ext3 to ext4.

Directory Entries and Long Names

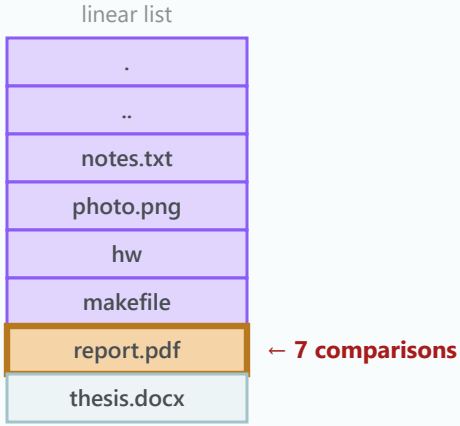


ext4 entry: i-node, rec_len, name length, type, name

entries aligned; names at the end of the directory block

To locate a directory it suffices to know the first block's address (contiguous), the first block's number (linked list/FAT), or the i-node number (UNIX). With long names, fixed entries like (a)/(b) waste space; (c) leaves holes on deletion and a name may cross a page boundary; (d) requires managing the heap.

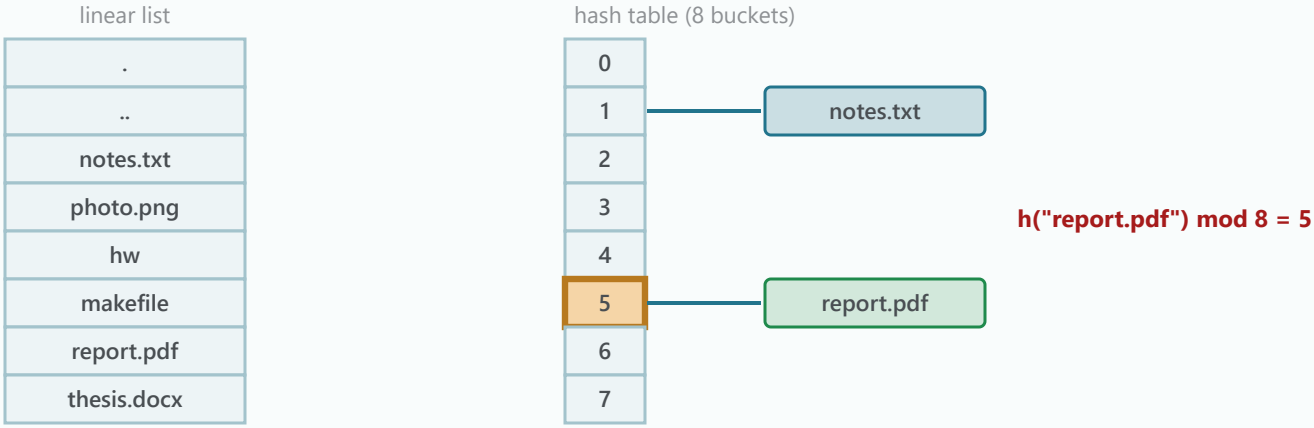
Directory Lookup: List, Hash, Tree



Linear list: simple, but lookup is O(n). Inserting first checks that the name doesn't exist; deleting marks the entry free or moves the last entry into it.

For large directories ext4 uses an *htree* ordered by name hash; NTFS, XFS, and Btrfs use **B+ trees**: sorted listing and O(log n) lookup together. Linux also keeps recently used name → i-node mappings in the **dentry cache**.

Directory Lookup: List, Hash, Tree

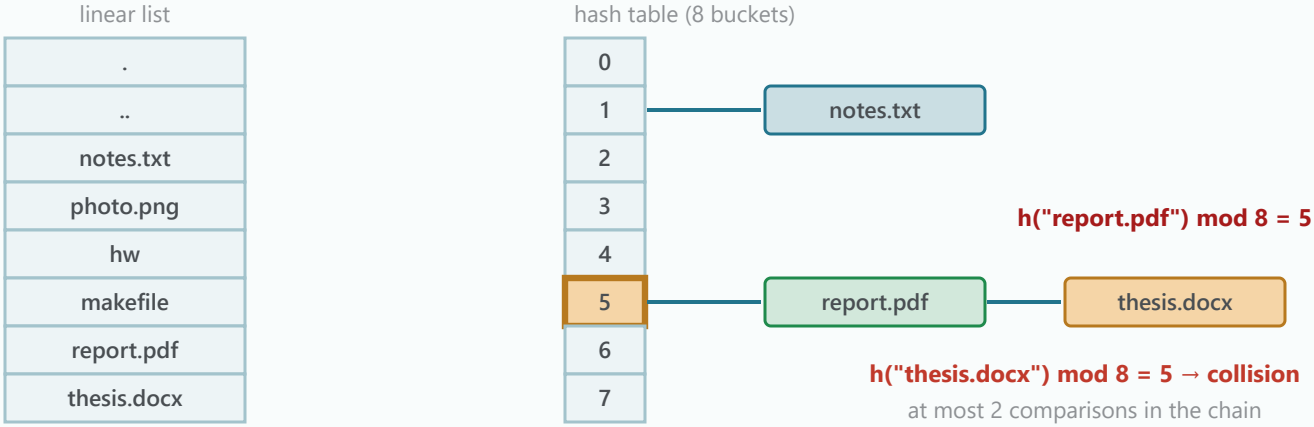


Hash table: the name's hash selects the bucket; lookup is O(1) on average.

Step 2 / 3

For large directories ext4 uses an *htree* ordered by name hash; NTFS, XFS, and Btrfs use **B+ trees**: sorted listing and O(log n) lookup together. Linux also keeps recently used name → i-node mappings in the **dentry cache**.

Directory Lookup: List, Hash, Tree

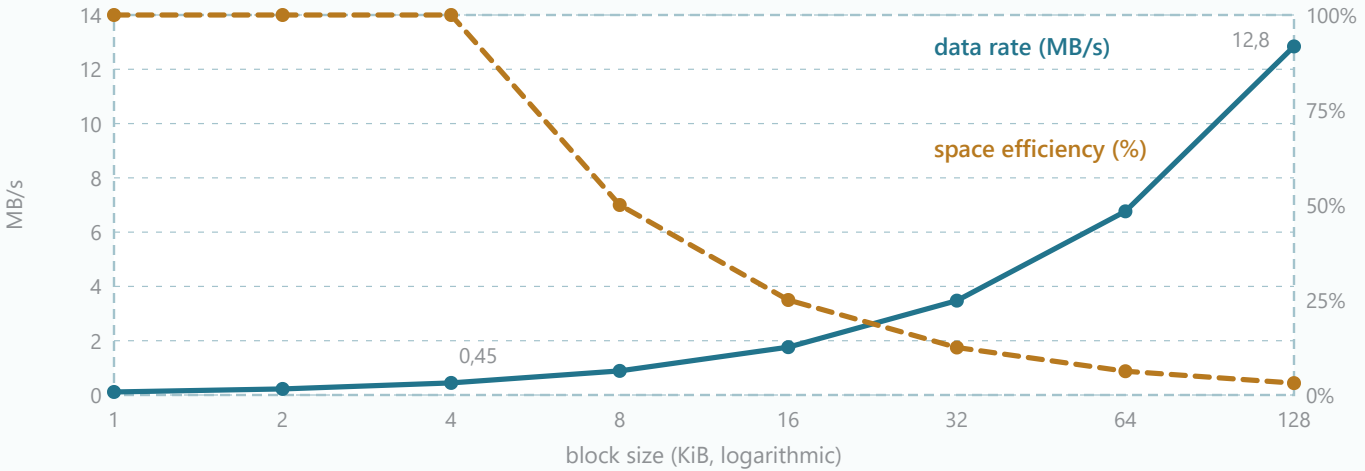


Collisions are resolved by chaining. A fixed number of buckets is a problem for a growing directory; that's why modern systems use **trees**.

Step 3 / 3

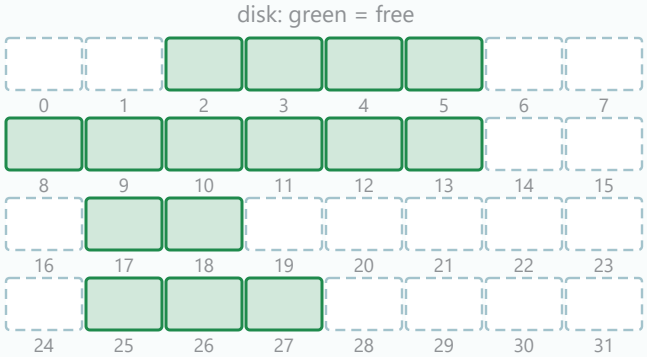
For large directories ext4 uses an *htree* ordered by name hash; NTFS, XFS, and Btrfs use **B+ trees**: sorted listing and O(log n) lookup together. Linux also keeps recently used name → i-node mappings in the **dentry cache**.

Choosing the Block Size



- Model (Tanenbaum): all files are 4 KiB; average seek 5 ms, rotation 8.33 ms, a 1 MB track. Time = 5 + 4,165 + (b / 1 MB) × 8.33 ms. Computed with Python.
- **Larger blocks → better data rate, worse space utilization**; smaller blocks the opposite.
- Measurements show that most files are small while most space is taken by a few large files. Today's default is **4 KiB** (the page size); large files are served by extents and cluster options.

Free-Space Management: Four Methods



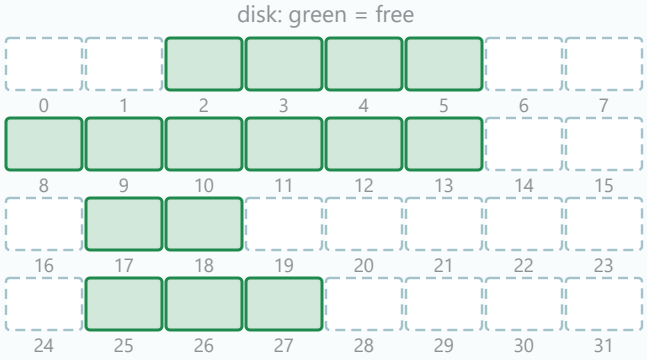
free blocks:

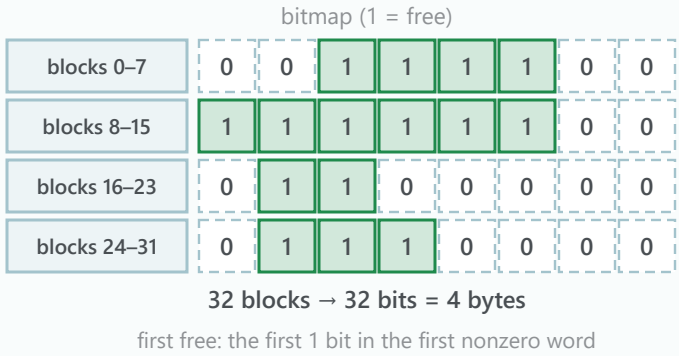
2–5, 8–13, 17–18, 25–27

15 free blocks / 32

Example disk: 32 blocks, 15 free (the Silberschatz example).

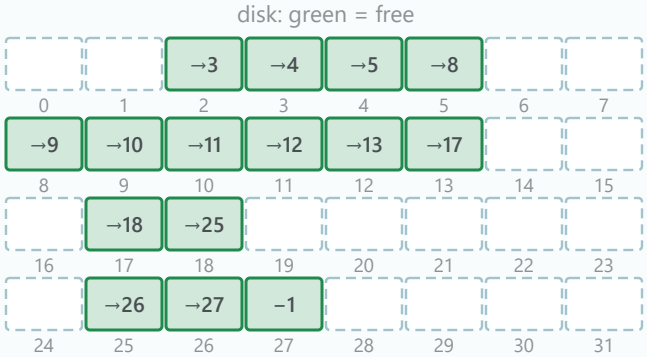
Free-Space Management: Four Methods





Bitmap: 1 bit per block. Fixed and small; finding contiguous free blocks is easy. It must be kept in memory.

Free-Space Management: Four Methods



head pointer → 2

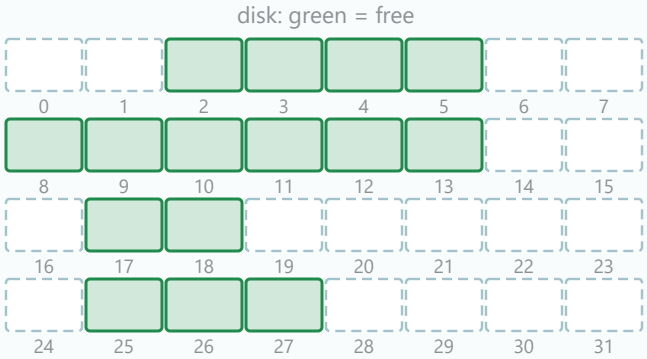
each free block points to the next

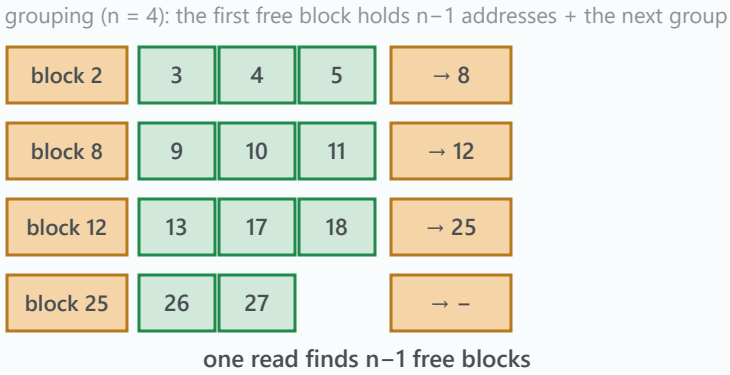
no extra space needed (the information is inside the free blocks)

but walking the list = I/O for every block

Linked list: free blocks are linked together. Taking one free block is cheap; finding many contiguous blocks is expensive.

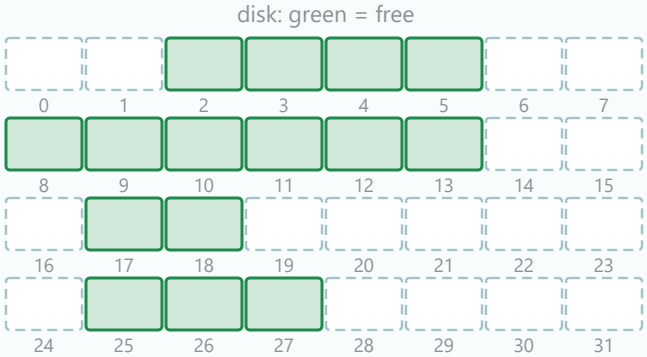
Free-Space Management: Four Methods





Grouping: the first free block holds the addresses of the next n−1 free blocks and of the next group block (UNIX's free-list blocks).

Free-Space Management: Four Methods



counting: (first block, run length)

(2, 4)	blocks 2–5
(8, 6)	blocks 8–13
(17, 2)	blocks 17–18
(25, 3)	blocks 25–27

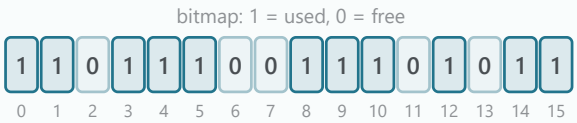
15 free blocks → 4 pairs

very short if free space comes in contiguous runs

Counting: (start, length) pairs; the extent idea applied to free space. XFS keeps these in two B+ trees.

Allocating a Block with a Bitmap: Code Trace

```
▶ int alloc_block(int hint) {
2   for (int k = 0; k < N; k++) {
3     int i = (hint + k) % N;
4     if (bitmap[i] == 0) {  /* free? */
5       bitmap[i] = 1;
6       return i;
7     }
8   }
9   return -1;               /* disk full */
10 }
```



the file's blocks: —

green: allocated to this file

The file ends at block 3 and needs 3 more blocks. For locality the search starts right after the last block: `alloc_block(4)`.

Step 1 / 9

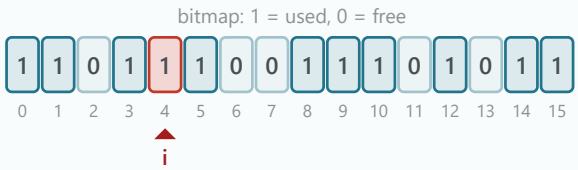
VARIABLES

`hint = 4` `k = –` `i = –`

OUTPUT

Allocating a Block with a Bitmap: Code Trace

```
1  int alloc_block(int hint) {
2    for (int k = 0; k < N; k++) {
▶      int i = (hint + k) % N;
▶      if (bitmap[i] == 0) {  /* free? */
5        bitmap[i] = 1;
6        return i;
7      }
8    }
9    return -1;                /* disk full */
10 }
```



the file's blocks: —

green: allocated to this file

k = 0: i = (4 + 0) mod 16 = 4; bit 1, block in use. Move on.

Step 2 / 9

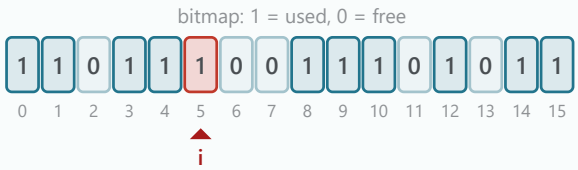
VARIABLES

`hint = 4` `k = 0` `i = 4`

OUTPUT

Allocating a Block with a Bitmap: Code Trace

```
1  int alloc_block(int hint) {
2    for (int k = 0; k < N; k++) {
▶      int i = (hint + k) % N;
▶      if (bitmap[i] == 0) {  /* free? */
5        bitmap[i] = 1;
6        return i;
7      }
8    }
9    return -1;                /* disk full */
10 }
```



the file's blocks: —

green: allocated to this file

k = 1: i = (4 + 1) mod 16 = 5; bit 1, block in use. Move on.

Step 3 / 9

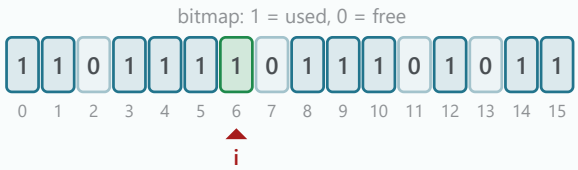
VARIABLES

`hint = 4` `k = 1` `i = 5`

OUTPUT

Allocating a Block with a Bitmap: Code Trace

```
1  int alloc_block(int hint) {
2    for (int k = 0; k < N; k++) {
▶      int i = (hint + k) % N;
▶      if (bitmap[i] == 0) {  /* free? */
▶        bitmap[i] = 1;
▶        return i;
7      }
8    }
9    return -1;                /* disk full */
10 }
```



the file's blocks: 6

green: allocated to this file

bitmap[6] = 0: **free**. The bit is set and block 6 is returned.

Step 4 / 9

VARIABLES

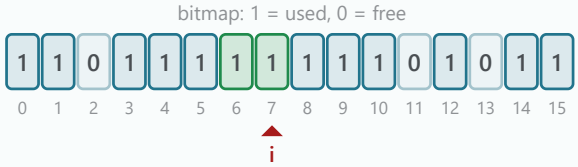
`hint = 4` `k = 2` `i = 6`

OUTPUT

```
6
```

Allocating a Block with a Bitmap: Code Trace

```
1  int alloc_block(int hint) {
2    for (int k = 0; k < N; k++) {
▶      int i = (hint + k) % N;
▶      if (bitmap[i] == 0) {  /* free? */
▶        bitmap[i] = 1;
▶        return i;
7      }
8    }
9    return -1;                /* disk full */
10 }
```



the file's blocks: 6, 7

green: allocated to this file

alloc_block(7): i = 7, bitmap[7] = 0: **free**. The bit is set and block 7 is returned. Consecutive blocks reduce head movement on sequential reads.

Step 5 / 9

VARIABLES

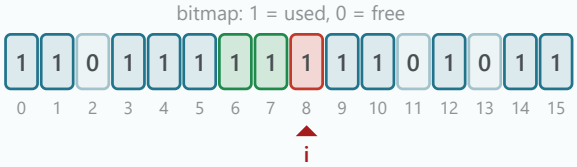
`hint = 7` `k = 0` `i = 7`

OUTPUT

```
6 7
```

Allocating a Block with a Bitmap: Code Trace

```
1  int alloc_block(int hint) {
2    for (int k = 0; k < N; k++) {
▶      int i = (hint + k) % N;
▶      if (bitmap[i] == 0) {  /* free? */
5        bitmap[i] = 1;
6        return i;
7      }
8    }
9    return -1;                /* disk full */
10 }
```



the file's blocks: 6, 7

green: allocated to this file

`alloc_block(8)` : k = 0, i = 8; bit 1, in use.

Step 6 / 9

VARIABLES

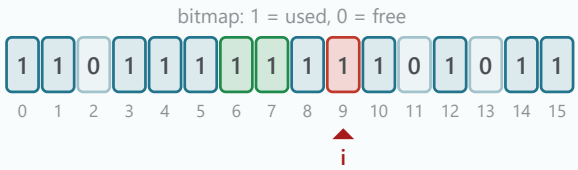
`hint = 8` `k = 0` `i = 8`

OUTPUT

```
6 7
```

Allocating a Block with a Bitmap: Code Trace

```
1  int alloc_block(int hint) {
2    for (int k = 0; k < N; k++) {
▶      int i = (hint + k) % N;
▶      if (bitmap[i] == 0) {  /* free? */
5        bitmap[i] = 1;
6        return i;
7      }
8    }
9    return -1;                /* disk full */
10 }
```



the file's blocks: 6, 7

green: allocated to this file

k = 1: i = (8 + 1) mod 16 = 9; bit 1, block in use. Move on.

Step 7 / 9

VARIABLES

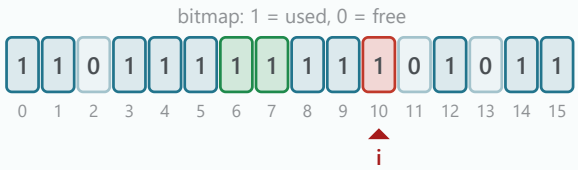
`hint = 8` `k = 1` `i = 9`

OUTPUT

```
6 7
```

Allocating a Block with a Bitmap: Code Trace

```
1  int alloc_block(int hint) {
2    for (int k = 0; k < N; k++) {
▶      int i = (hint + k) % N;
▶      if (bitmap[i] == 0) {  /* free? */
5        bitmap[i] = 1;
6        return i;
7      }
8    }
9    return -1;                /* disk full */
10 }
```



the file's blocks: 6, 7

green: allocated to this file

k = 2: i = (8 + 2) mod 16 = 10; bit 1, block in use. Move on.

Step 8 / 9

VARIABLES

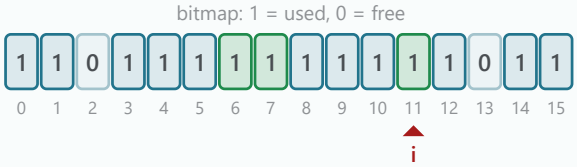
`hint = 8` `k = 2` `i = 10`

OUTPUT

```
6 7
```

Allocating a Block with a Bitmap: Code Trace

```
1  int alloc_block(int hint) {
2    for (int k = 0; k < N; k++) {
▶      int i = (hint + k) % N;
▶      if (bitmap[i] == 0) {  /* free? */
▶        bitmap[i] = 1;
▶        return i;
7      }
8    }
9    return -1;                /* disk full */
10 }
```



the file's blocks: 6, 7, 11

green: allocated to this file

bitmap[11] = 0: **free**. The bit is set and block 11 is returned. The file got blocks 6, 7, 11: not contiguous, but close together.

Step 9 / 9

VARIABLES

`hint = 8` `k = 3` `i = 11`

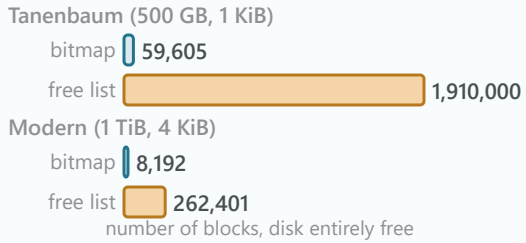
OUTPUT

```
6 7 11
```

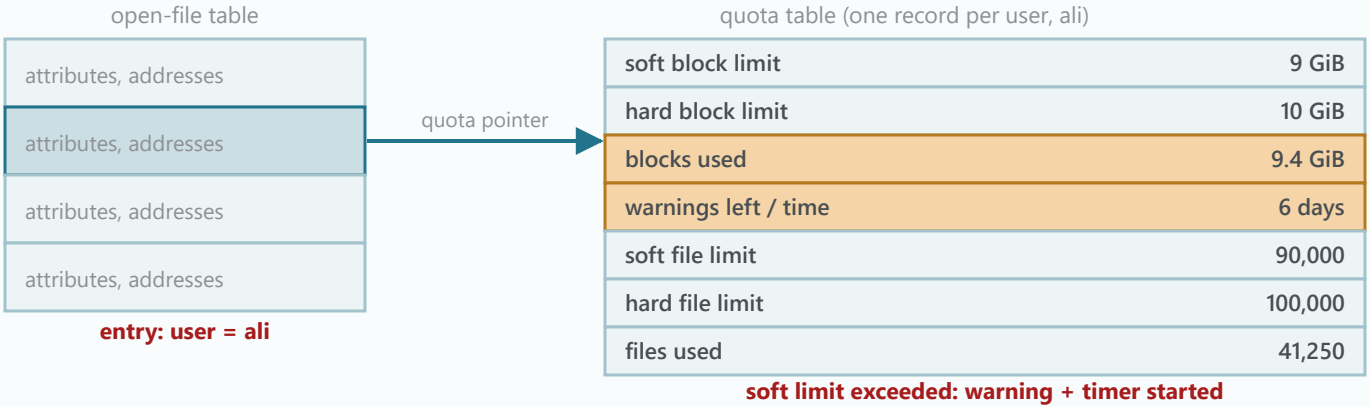
Free Space: A Numerical Comparison

	Tanenbaum's example	A modern example
Disk / block	500 GB, 1 KiB blocks	1 TiB, 4 KiB blocks
Number of blocks	$500 \times 10^9 / 1024 \approx$ **488.3 million**	$2^{40} / 2^{12} =$ **268,435,456**
Free-list block (32-bit numbers)	1 KiB / 4 B = 256 → 255 addresses + 1 link	4096 / 4 = 1024 → 1023 addresses + 1 link
List, with the disk entirely free	488.3 M / 255 ≈ **1.91 million** blocks	268.4 M / 1023 ≈ **262,401** blocks (≈ 1 GiB)
Bitmap	488.3 M bits / 8192 ≈ **59,605** blocks (≈ 60,000)	2^{28} bits = 32 MiB = **8192** blocks

- The bitmap has a fixed size; the list is smaller only when the disk is *nearly full* (since it lists only free blocks).
- Keeping one pointer block in memory is enough: write it to disk when full, read the next when empty. Tanenbaum's refinement: keep a half-full block in memory so temporary files don't cause constant disk I/O.
- Real systems: ext4 has a bitmap per block group; XFS two B+ trees by position and by size (counting); ZFS an allocate/free log (space maps); Btrfs an extent tree.

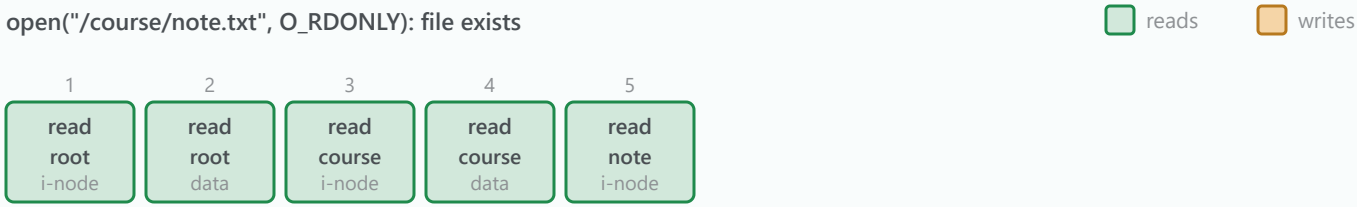


Disk Quotas



- Every open-file entry points to its owner's quota record; on every new block or i-node allocation the counters go up and are compared with the limits.
- The **soft limit** may be exceeded, but a warning is given and a grace period (e.g., 7 days) starts; when it expires, or the **hard limit** is reached, writes fail with `EDQUOT`.
- Today: Linux `quota` tools (ext4, XFS project quotas), NTFS volume quotas, ZFS dataset and Btrfs subvolume (qgroup) quotas.

How Many I/Os Does Creating a File Take?



5 I/Os = 5 reads + 0 writes

The cache is empty; the root's i-node location is known from the superblock. The path is resolved component by component: for each directory, its i-node + data block, and finally the file's i-node. **5 reads**; more for longer paths.

Step 1 / 5

Assumption: a simple file system with i-nodes and bitmaps, single-block directories, no cache. In real systems the dentry and i-node caches remove most reads; journaling and delayed writes also change the number and order of writes.

How Many I/Os Does Creating a File Take?

read(fd, buf, 4096): first block

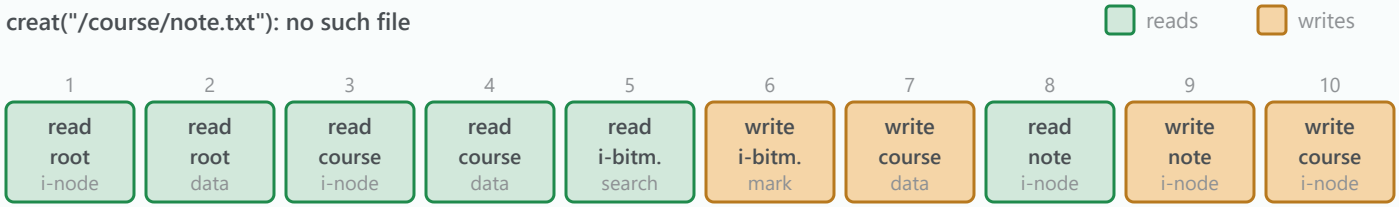


3 I/Os = 2 reads + 1 writes

Each `read` finds the block address from the i-node, reads the block, and writes the **access time**. The `noatime` / `relatime` mount options usually eliminate that write.

Assumption: a simple file system with i-nodes and bitmaps, single-block directories, no cache. In real systems the dentry and i-node caches remove most reads; journaling and delayed writes also change the number and order of writes.

How Many I/Os Does Creating a File Take?

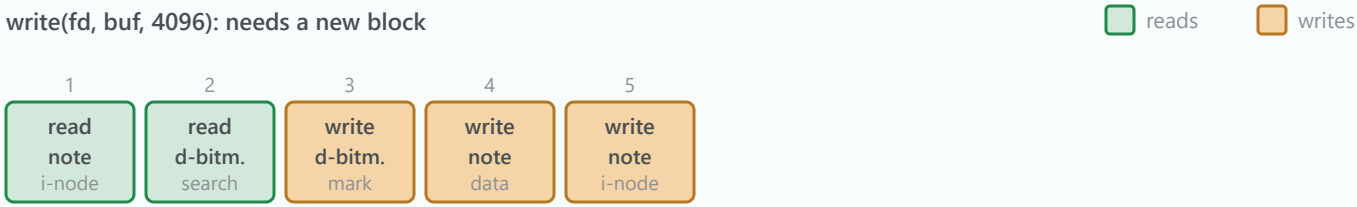


10 I/Os = 6 reads + 4 writes

Creating is costlier: a free i-node is chosen from the bitmap, the new i-node initialized, an entry added to the directory, and the directory's i-node (size, times) updated. **10 I/Os**.

Assumption: a simple file system with i-nodes and bitmaps, single-block directories, no cache. In real systems the dentry and i-node caches remove most reads; journaling and delayed writes also change the number and order of writes.

How Many I/Os Does Creating a File Take?



5 I/Os = 2 reads + 3 writes

Every `write` that needs a new block: allocate a block from the data bitmap (read + write), write the data, write the address and size into the i-node. **5 I/Os**, only 1 of them actual data.

Assumption: a simple file system with i-nodes and bitmaps, single-block directories, no cache. In real systems the dentry and i-node caches remove most reads; journaling and delayed writes also change the number and order of writes.

How Many I/Os Does Creating a File Take?

creat + 3 × write(4096): 10 + 3 × 5 = 25 I/Os



3 thick-bordered boxes are user data; the other 22 I/Os are metadata (bitmaps, i-nodes, directory)

cache: reads come from memory · delayed writes: each block is written once

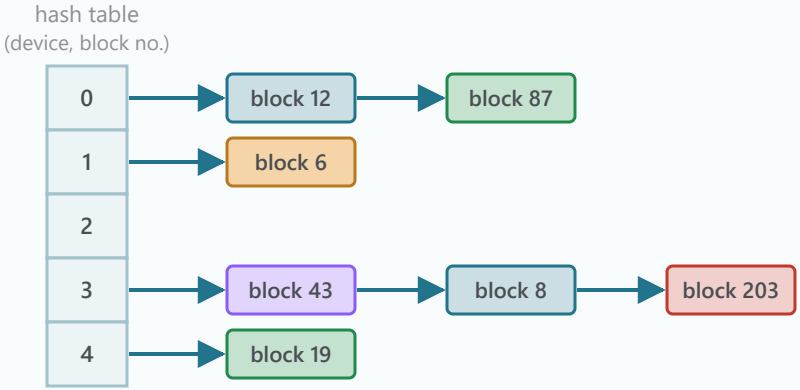
→ 8 writes remain: i-bitmap, d-bitmap, course data, course i-node, note i-node, 3 data blocks

Creating a file and writing 12 KiB without a cache costs **25 I/Os**. That's why the buffer cache and delayed writes are indispensable; the order of writes also determines crash consistency.

Assumption: a simple file system with i-nodes and bitmaps, single-block directories, no cache. In real systems the dentry and i-node caches remove most reads; journaling and delayed writes also change the number and order of writes.

PERFORMANCE

The Buffer Cache



lookup: O(1) via hash

replacement: the head of the LRU queue

critical blocks (i-nodes, directories)
written at once if dirty

LRU list (oldest → newest)

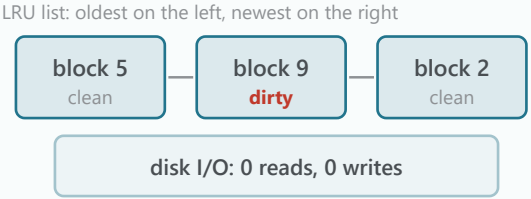
6 → 43 → 12 → 19 → 8 → 87 → 203

Access	Latency (2026, typical)
64 bytes from DRAM	≈ 80–120 ns
NVMe SSD 4 KiB random read	≈ 50–100 µs
HDD seek + rotation	≈ 5–10 ms (100–280 MB/s sequential)

- **Write policy**: *write-through* sends every write to disk at once (old MS-DOS, removable media); *write-back* accumulates dirty blocks. Linux writes back dirty pages older than 30 s in the background; applications force it with `fsync`.

The Buffer Cache: Code Trace

```
▶ Buffer *bread(int blk) {
2   Buffer *t = hash_find(blk);
3   if (t != NULL) {          /* hit */
4     lru_to_tail(t);
5     return t;
6   }
7   t = lru_head();           /* oldest */
8   if (t->dirty)
9     disk_write(t);        /* write back */
10  t->blk = blk; t->dirty = 0;
11  disk_read(t);             /* miss: I/O */
12  lru_to_tail(t);
13  return t;
14 }
```



The cache holds 3 buffers: blocks 5, 9, 2 (9 is dirty: modified but not written to disk). Requests in order: 9, 7, 2, 4.

Step 1 / 10

VARIABLES

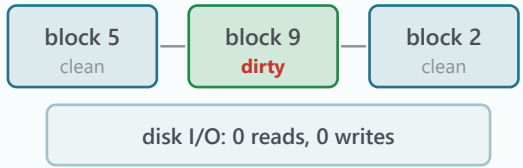
`blk = –` `t = –` `reads = 0` `writes = 0`

OUTPUT

The Buffer Cache: Code Trace

```
1  Buffer *bread(int blk) {
▶    Buffer *t = hash_find(blk);
▶    if (t != NULL) {          /* hit */
4      lru_to_tail(t);
5      return t;
6    }
7    t = lru_head();           /* oldest */
8    if (t->dirty)
9      disk_write(t);        /* write back */
10   t->blk = blk; t->dirty = 0;
11   disk_read(t);             /* miss: I/O */
12   lru_to_tail(t);
13   return t;
14 }
```

LRU list: oldest on the left, newest on the right



`bread(9)` : found in the hash table, a **hit**. No disk I/O.

Step 2 / 10

VARIABLES

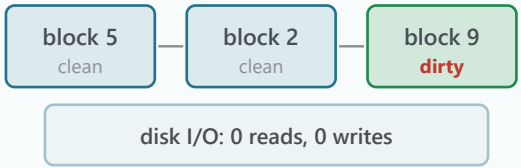
`blk = 9` `t = block 9` `reads = 0` `writes = 0`

OUTPUT

The Buffer Cache: Code Trace

```
1  Buffer *bread(int blk) {
2    Buffer *t = hash_find(blk);
3    if (t != NULL) {          /* hit */
▶      lru_to_tail(t);
▶      return t;
6    }
7    t = lru_head();           /* oldest */
8    if (t->dirty)
9      disk_write(t);        /* write back */
10   t->blk = blk; t->dirty = 0;
11   disk_read(t);             /* miss: I/O */
12   lru_to_tail(t);
13   return t;
14 }
```

LRU list: oldest on the left, newest on the right



Block 9 moves to the tail of the list (newest) and is returned. Now block 5 is the oldest.

Step 3 / 10

VARIABLES

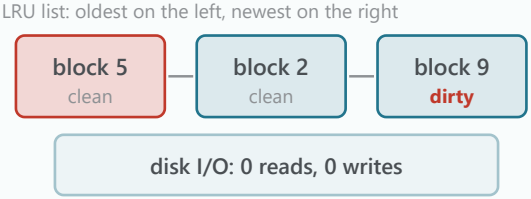
`blk = 9` `t = block 9` `reads = 0` `writes = 0`

OUTPUT

```
9: hit
```

The Buffer Cache: Code Trace

```
1  Buffer *bread(int blk) {
▶    Buffer *t = hash_find(blk);
3    if (t != NULL) {          /* hit */
4      lru_to_tail(t);
5      return t;
6    }
▶    t = lru_head();           /* oldest */
8    if (t->dirty)
9      disk_write(t);        /* write back */
10   t->blk = blk; t->dirty = 0;
11   disk_read(t);             /* miss: I/O */
12   lru_to_tail(t);
13   return t;
14 }
```



`bread(7)` : not in the cache, a **miss**. To make room, the buffer at the head of the LRU list (block 5) is chosen.

Step 4 / 10

VARIABLES

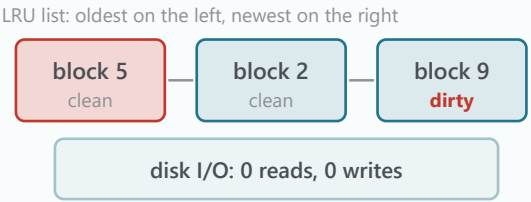
`blk = 7` `t = block 5` `reads = 0` `writes = 0`

OUTPUT

```
9: hit
```

The Buffer Cache: Code Trace

```
1  Buffer *bread(int blk) {
2    Buffer *t = hash_find(blk);
3    if (t != NULL) {          /* hit */
4      lru_to_tail(t);
5      return t;
6    }
7    t = lru_head();           /* oldest */
▶    if (t->dirty)
9      disk_write(t);        /* write back */
10   t->blk = blk; t->dirty = 0;
11   disk_read(t);             /* miss: I/O */
12   lru_to_tail(t);
13   return t;
14 }
```



Block 5 is clean: its copy on disk is current, so no write-back is needed.

Step 5 / 10

VARIABLES

`blk = 7` `t = block 5` `reads = 0` `writes = 0`

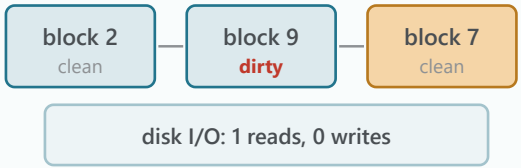
OUTPUT

```
9: hit
```

The Buffer Cache: Code Trace

```
1  Buffer *bread(int blk) {
2    Buffer *t = hash_find(blk);
3    if (t != NULL) {          /* hit */
4      lru_to_tail(t);
5      return t;
6    }
7    t = lru_head();           /* oldest */
8    if (t->dirty)
9      disk_write(t);        /* write back */
▶    t->blk = blk; t->dirty = 0;
▶    disk_read(t);             /* miss: I/O */
▶    lru_to_tail(t);
13   return t;
14 }
```

LRU list: oldest on the left, newest on the right



The buffer is reused for block 7; the contents are read from disk (1 read) and it is added to the tail of the list.

Step 6 / 10

VARIABLES

`blk = 7` `t = block 7` `reads = 1` `writes = 0`

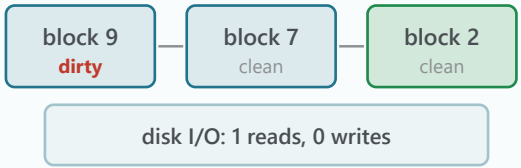
OUTPUT

```
9: hit
7: miss
```

The Buffer Cache: Code Trace

```
1  Buffer *bread(int blk) {
▶    Buffer *t = hash_find(blk);
▶    if (t != NULL) {          /* hit */
▶      lru_to_tail(t);
▶      return t;
6    }
7    t = lru_head();           /* oldest */
8    if (t->dirty)
9      disk_write(t);        /* write back */
10   t->blk = blk; t->dirty = 0;
11   disk_read(t);             /* miss: I/O */
12   lru_to_tail(t);
13   return t;
14 }
```

LRU list: oldest on the left, newest on the right



`bread(2)` : a hit; block 2 moves to newest. Dirty block 9 is now at the head of the list.

Step 7 / 10

VARIABLES

`blk = 2` `t = block 2` `reads = 1` `writes = 0`

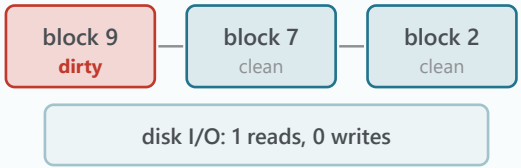
OUTPUT

```
9: hit
7: miss
2: hit
```

The Buffer Cache: Code Trace

```
1  Buffer *bread(int blk) {
▶    Buffer *t = hash_find(blk);
3    if (t != NULL) {          /* hit */
4      lru_to_tail(t);
5      return t;
6    }
▶    t = lru_head();           /* oldest */
8    if (t->dirty)
9      disk_write(t);        /* write back */
10   t->blk = blk; t->dirty = 0;
11   disk_read(t);             /* miss: I/O */
12   lru_to_tail(t);
13   return t;
14 }
```

LRU list: oldest on the left, newest on the right



`bread(4)` : a miss. The oldest buffer holds block 9.

Step 8 / 10

VARIABLES

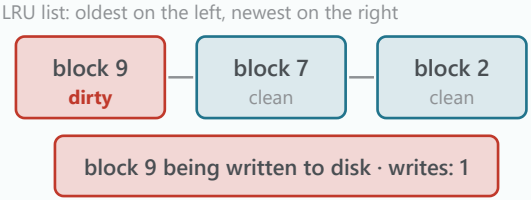
`blk = 4` `t = block 9` `reads = 1` `writes = 0`

OUTPUT

```
9: hit
7: miss
2: hit
```

The Buffer Cache: Code Trace

```
1  Buffer *bread(int blk) {
2    Buffer *t = hash_find(blk);
3    if (t != NULL) {          /* hit */
4      lru_to_tail(t);
5      return t;
6    }
7    t = lru_head();           /* oldest */
▶    if (t->dirty)
▶      disk_write(t);        /* write back */
10   t->blk = blk; t->dirty = 0;
11   disk_read(t);             /* miss: I/O */
12   lru_to_tail(t);
13   return t;
14 }
```



Block 9 is **dirty**: so the change isn't lost, the buffer is written to disk before reuse (1 write). This doubles the cost of the miss.

Step 9 / 10

VARIABLES

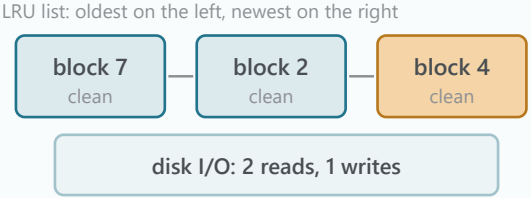
`blk = 4` `t = block 9` `reads = 1` `writes = 1`

OUTPUT

```
9: hit
7: miss
2: hit
```

The Buffer Cache: Code Trace

```
1  Buffer *bread(int blk) {
2    Buffer *t = hash_find(blk);
3    if (t != NULL) {          /* hit */
4      lru_to_tail(t);
5      return t;
6    }
7    t = lru_head();           /* oldest */
8    if (t->dirty)
9      disk_write(t);        /* write back */
▶    t->blk = blk; t->dirty = 0;
▶    disk_read(t);             /* miss: I/O */
▶    lru_to_tail(t);
▶    return t;
14 }
```



Block 4 is read. 2 hits out of 4 requests; 2 reads and 1 write in total. Without a cache it would take 4 reads plus a separate write for block 9.

Step 10 / 10

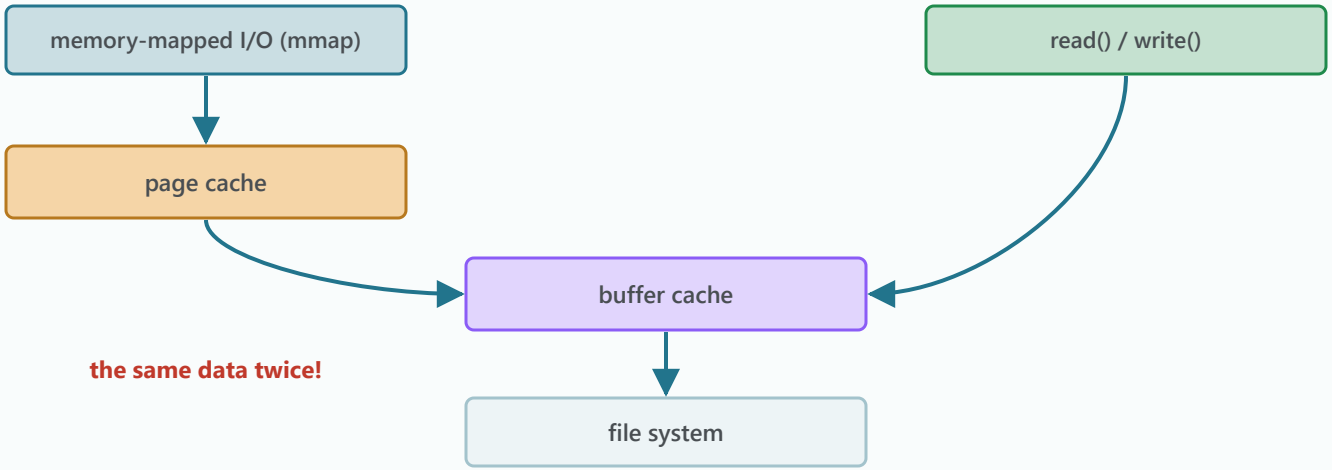
VARIABLES

`blk = 4` `t = block 4` `reads = 2` `writes = 1`

OUTPUT

```
9: hit
7: miss
2: hit
4: miss (9 written back)
```

The Page Cache and the Unified Buffer Cache

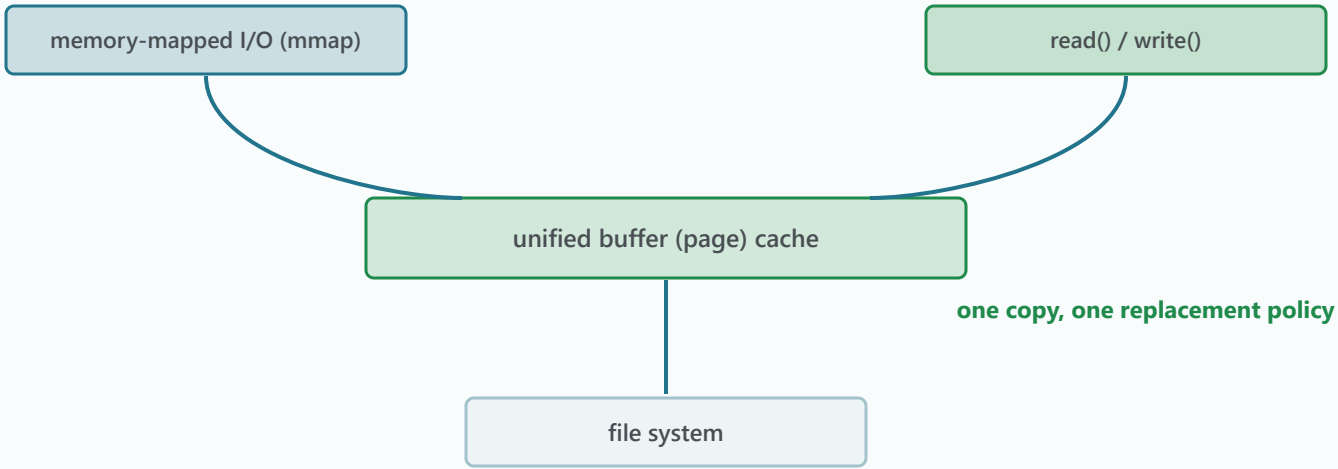


Not unified: `mmap` data is in the page cache, `read/write` data in the buffer cache, and it's copied between them (**double caching**): wasted memory and consistency problems.

Step 1 / 2

The page cache shares the page-replacement algorithms of virtual memory. The exception: ZFS keeps its own ARC cache. `O_DIRECT` lets databases bypass the cache.

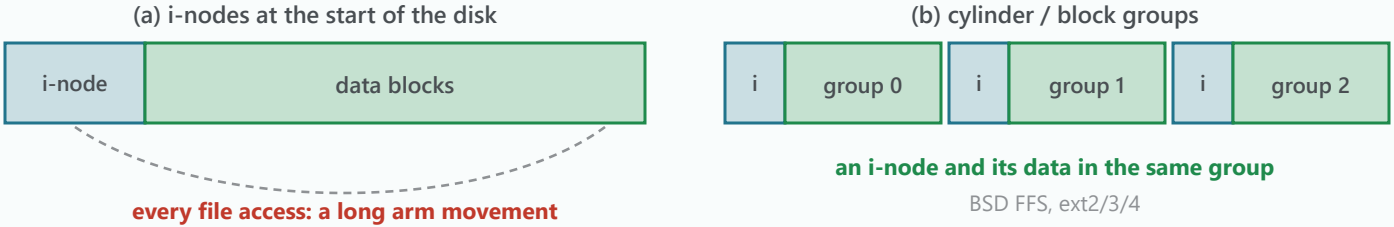
The Page Cache and the Unified Buffer Cache



Unified buffer cache: file pages live in one place, and both paths see the same pages. The page cache since Linux 2.4, the Windows cache manager, and Solaris all work this way.

The page cache shares the page-replacement algorithms of virtual memory. The exception: ZFS keeps its own ARC cache. `O_DIRECT` lets databases bypass the cache.

Read-Ahead, Placement, and Defragmentation



- **Read-ahead**: when block k is requested, block k+1 is read too (if not cached). It helps only with sequential access; a "sequential?" hint is tracked per file. Linux adapts the window (default 128 KiB); applications give hints with `posix_fadvise`.
- **Reducing arm motion**: place blocks that will be read consecutively close together; a bitmap makes it easy to find contiguous blocks. Allocate blocks two at a time (an old suggestion), or, as today, use extents, delayed allocation, and a multiblock allocator.
- **Defragmentation**: over time files get fragmented. Windows has "Defragment and Optimize Drives"; Linux has it too: `e4defrag`, `xfs_fsr`, `btrfs filesystem defragment`. ext4's allocation policy reduces the need.
- An SSD has no seek time: defragmentation just means unnecessary writes (wear); Windows sends TRIM instead of defragmenting an SSD.

PERFORMANCE

SSD Caveats: FTL and TRIM

- A flash page can't be updated in place; the SSD's **FTL** redirects every write to a new page, so internally an SSD is log-structured.
- When the file system deletes a file it only marks the blocks free in its own bitmap. Unless it tells the SSD with **TRIM** (NVMe: Deallocate), the SSD thinks those pages are valid and copies them needlessly during garbage collection.
- On Linux the notification is sent either in batches by the weekly `fstrim.timer` or on every delete with the `discard` mount option.

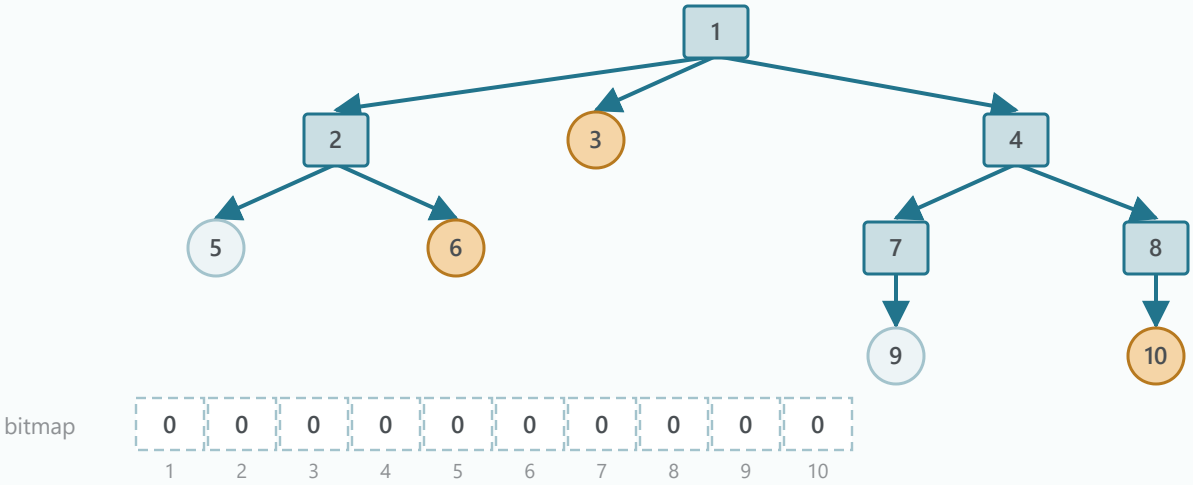
Backup Strategies

	Physical dump	Logical dump
What is copied?	all (used) blocks from block 0 to the end	selected directories and files, with their paths
Speed, simplicity	very fast, simple	slower, complex
Skipping directories, incremental backups	can't	can (changed files)
Restoring a single file	hard	easy; on another machine too
Bad blocks	the controller must remap them, or the OS must know the list	no problem



- Two goals: **recovery from disaster** (disk failure, fire, ransomware) and **recovery from carelessness** (an accidentally deleted file).
- **Full + incremental**: e.g., a full dump monthly, and daily only what changed since the last dump. Restore: the latest full dump onto an empty file system (directories first, then files), then the incrementals in order.
- Temporary and special files (`/dev` , `/proc` , `/tmp`) aren't backed up. For a consistent backup while the system is in use, take a **snapshot** (LVM, ZFS/Btrfs, Windows VSS).
- **The 3-2-1 rule**: 3 copies, on 2 different media, with 1 copy elsewhere (offline or immutable). Tape is still used for archiving: an LTO-10 cartridge holds up to 40 TB uncompressed.

The Logical Dump Algorithm

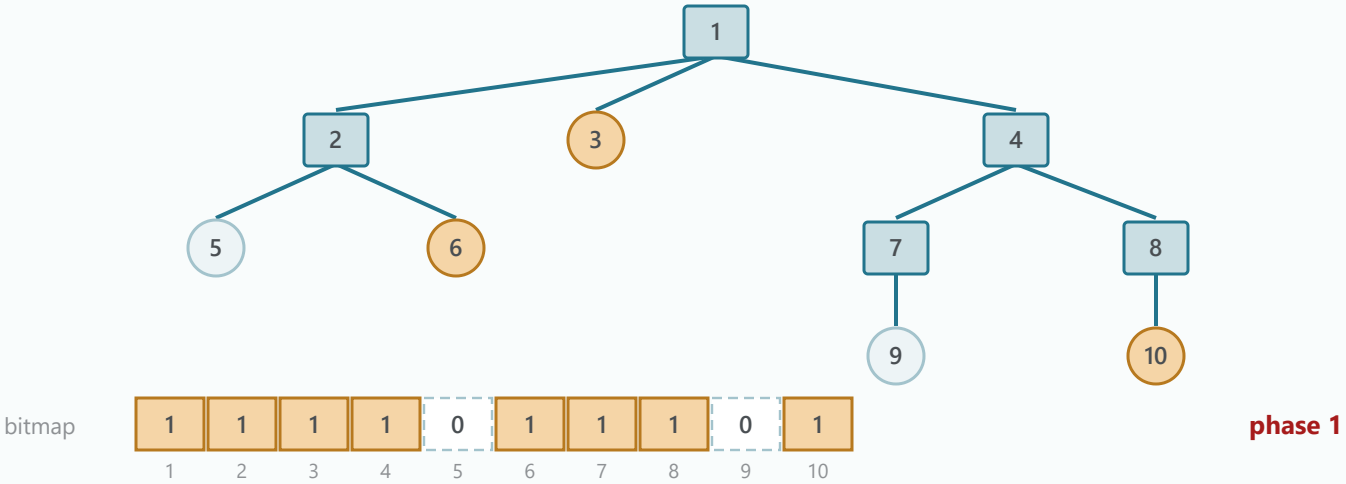


Squares are directories, circles files; the numbers are i-node numbers. Orange: **modified** since the last dump (3, 6, 10).

Step 1 / 5

More details: the free list isn't dumped but rebuilt on restore; a file with several hard links must be dumped only once; holes in sparse files and FIFO/device files need special handling.

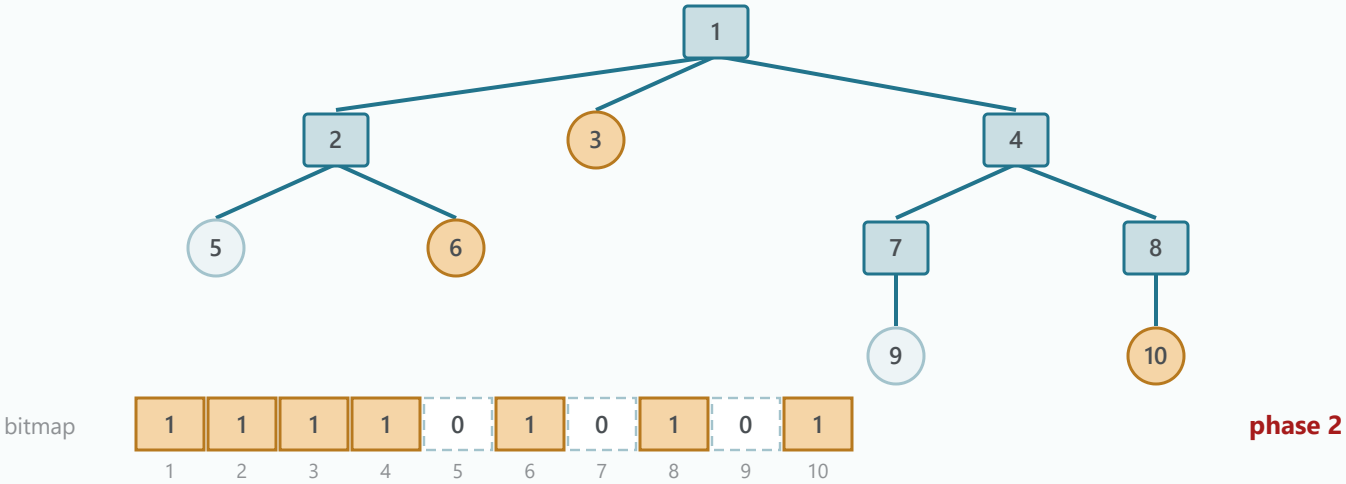
The Logical Dump Algorithm



Phase 1: walk the tree from the root, marking every modified file and **all directories**.

More details: the free list isn't dumped but rebuilt on restore; a file with several hard links must be dumped only once; holes in sparse files and FIFO/device files need special handling.

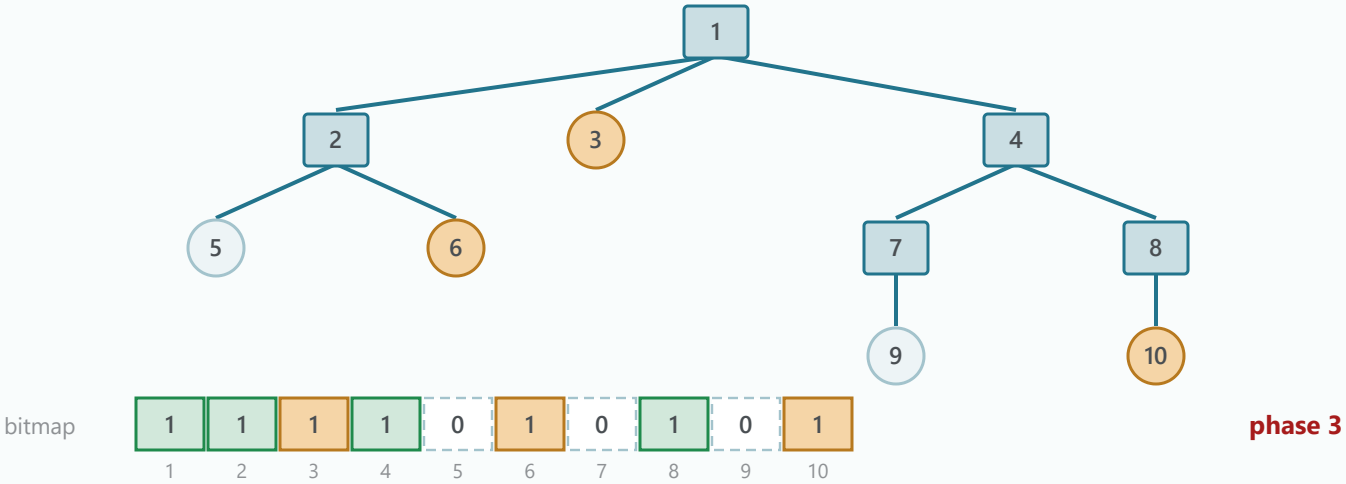
The Logical Dump Algorithm



Phase 2: unmark directories with nothing modified beneath them: **7** (it contains only the unmodified 9).

More details: the free list isn't dumped but rebuilt on restore; a file with several hard links must be dumped only once; holes in sparse files and FIFO/device files need special handling.

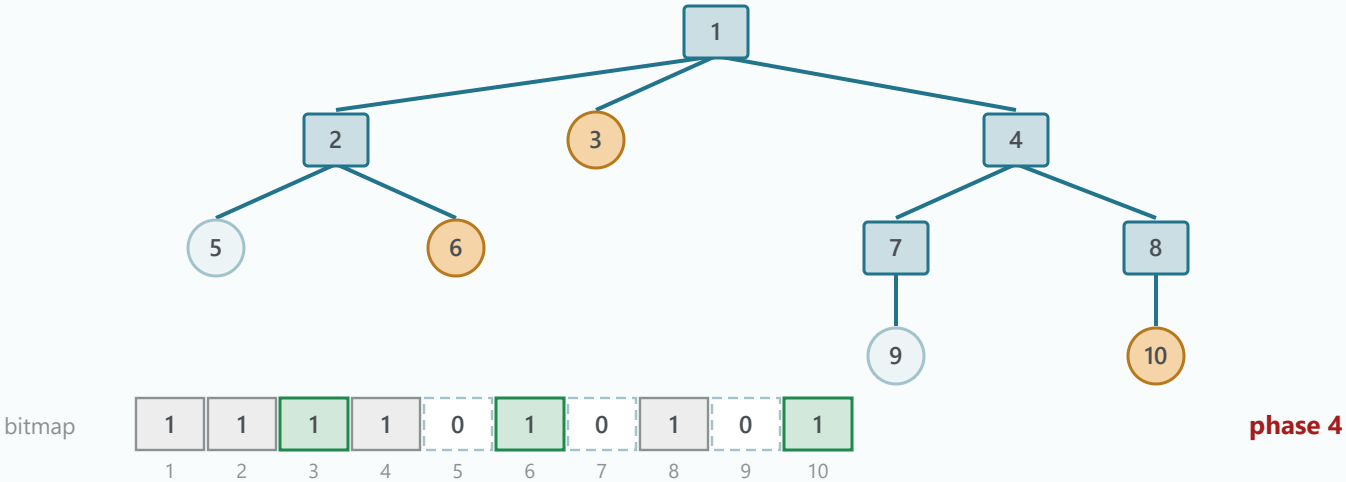
The Logical Dump Algorithm



Phase 3: dump the marked **directories** in i-node order (1, 2, 4, 8), even if unmodified, so paths can be rebuilt.

More details: the free list isn't dumped but rebuilt on restore; a file with several hard links must be dumped only once; holes in sparse files and FIFO/device files need special handling.

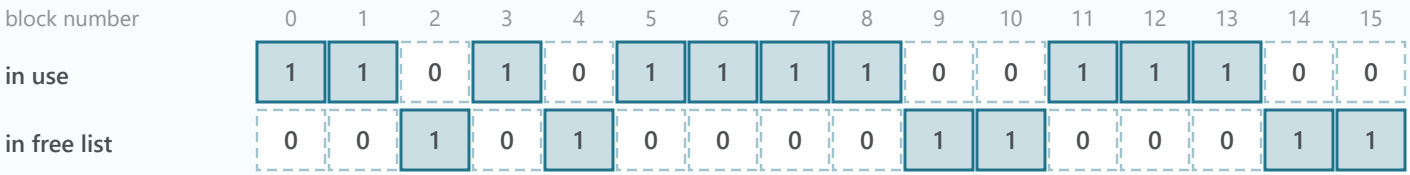
The Logical Dump Algorithm



Phase 4: dump the marked **files** (3, 6, 10). On restore, directories are written first, then files.

More details: the free list isn't dumped but rebuilt on restore; a file with several hard links must be dumped only once; holes in sparse files and FIFO/device files need special handling.

Consistency Checking: fsck



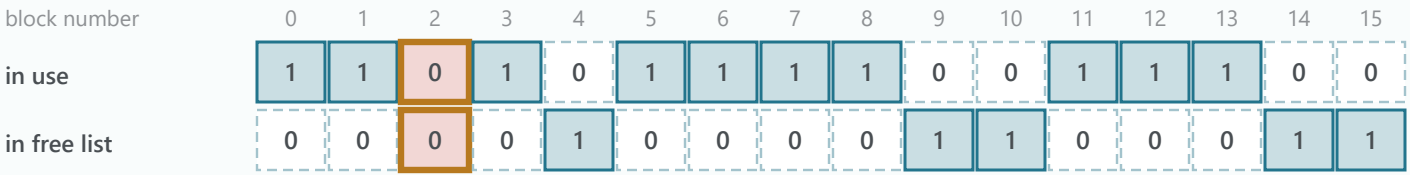
Each block is 1 in one counter and 0 in the other: consistent.

(a) Consistent: fsck reads all i-nodes and the free list/bitmap, keeping two counters per block.

Step 1 / 4

- **File consistency**: walk all directories from the root, counting how often each i-node is seen, and compare with the i-node's link count. A link count that's too high just wastes space; one that's *too low* is dangerous (the file would be deleted early) → it is fixed. Orphaned i-nodes go into `lost+found`.
- The Windows equivalent is `chkdsk` (called *scandisk* in Windows 9x). A full scan of a terabyte disk takes hours; that's why journaling and CoW were born.

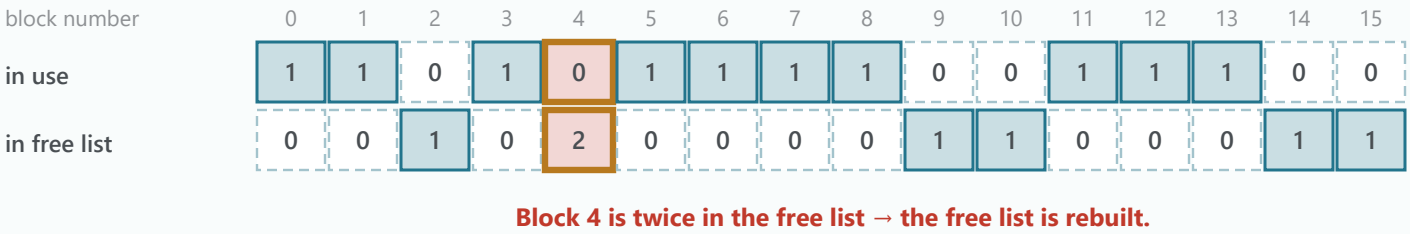
Consistency Checking: fsck



(b) Missing block: harmless but wastes space; it goes on the free list.

- **File consistency**: walk all directories from the root, counting how often each i-node is seen, and compare with the i-node's link count. A link count that's too high just wastes space; one that's *too low* is dangerous (the file would be deleted early) → it is fixed. Orphaned i-nodes go into `lost+found`.
- The Windows equivalent is `chkdsk` (called *scandisk* in Windows 9x). A full scan of a terabyte disk takes hours; that's why journaling and CoW were born.

Consistency Checking: fsck



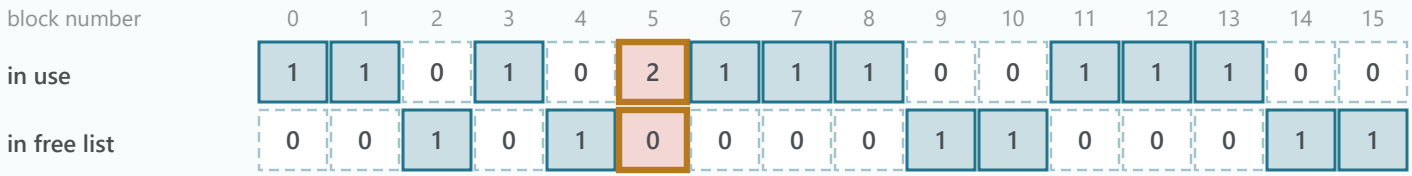
Block 4 is twice in the free list → the free list is rebuilt.

(c) Duplicate block in the free list: the same block could be handed out twice; the list is rebuilt. (This can't happen with a bitmap.)

Step 3 / 4

- **File consistency**: walk all directories from the root, counting how often each i-node is seen, and compare with the i-node's link count. A link count that's too high just wastes space; one that's *too low* is dangerous (the file would be deleted early) → it is fixed. Orphaned i-nodes go into `lost+found`.
- The Windows equivalent is `chkdsk` (called *scandisk* in Windows 9x). A full scan of a terabyte disk takes hours; that's why journaling and CoW were born.

Consistency Checking: fsck



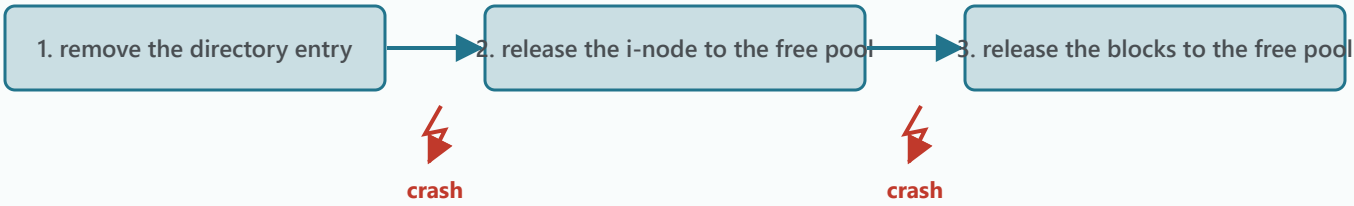
Block 5 in two files → allocate a new block, copy, notify the user.

(d) Duplicate data block: the worst case; the block is copied so each file gets its own, and the user is informed.

Step 4 / 4

- **File consistency**: walk all directories from the root, counting how often each i-node is seen, and compare with the i-node's link count. A link count that's too high just wastes space; one that's *too low* is dangerous (the file would be deleted early) → it is fixed. Orphaned i-nodes go into `lost+found`.
- The Windows equivalent is `chkdsk` (called *scandisk* in Windows 9x). A full scan of a terabyte disk takes hours; that's why journaling and CoW were born.

Crashes and Atomicity: Why a Journal?



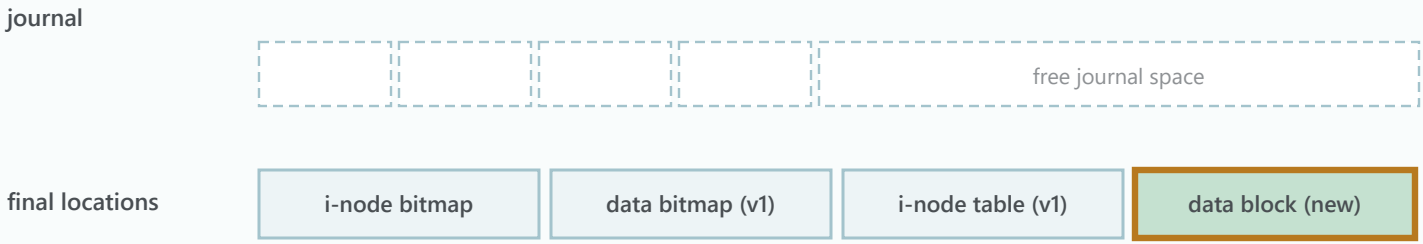
crash after 1: the i-node and blocks are orphaned (a leak; fsck finds it)

crash after 2: the blocks leak

in the reverse order: the directory could point to an i-node reused by another file → data gets mixed up!

- One file operation changes several blocks; disks make only single-sector writes atomic. A crash in between leaves the structure **inconsistent**.
- The family of solutions: (1) repair afterward: **fsck**; (2) write the intent first: **journaling**; (3) never overwrite: **CoW** and **LFS**; (4) order the writes: BSD *soft updates*.
- **Idempotence**: the operations in the journal must give the same result however many times they are replayed. "Mark block n free" is idempotent; "append n to the free list" is not (it could be added twice).

Journaling: Write to the Log First

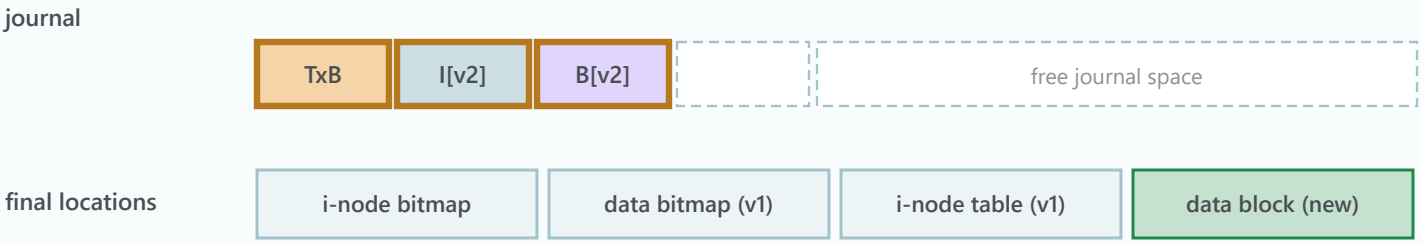


1. Data: the block Db added to the file is written to its final location first (*ordered* mode, ext4's default).

Step 1 / 6

Modes: *journal* (data journaled too: safest, slowest), *ordered*, *writeback* (no data ordering). Journaling file systems: ext3/ext4 (jbd2), XFS, NTFS (`$LogFile`), JFS. At Apple, HFS+ gave way to the CoW-based APFS.

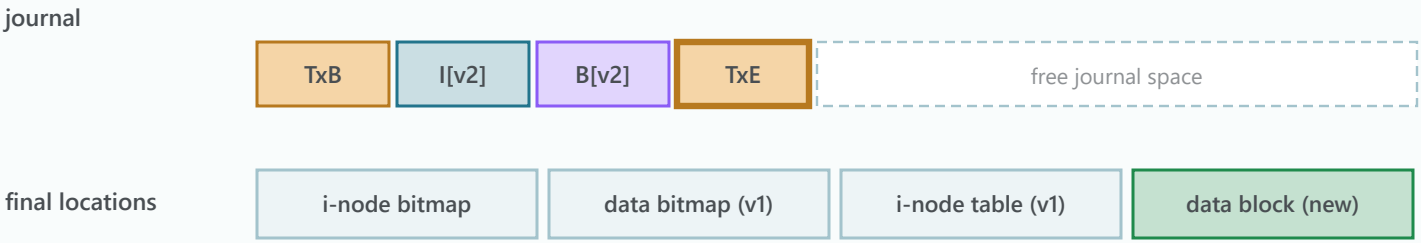
Journaling: Write to the Log First



2. Journal write: the transaction begin TxB and the metadata blocks to change (i-node I, bitmap B) are written to the journal **sequentially**.

Modes: *journal* (data journaled too: safest, slowest), *ordered*, *writeback* (no data ordering). Journaling file systems: ext3/ext4 (jbd2), XFS, NTFS (`$LogFile`), JFS. At Apple, HFS+ gave way to the CoW-based APFS.

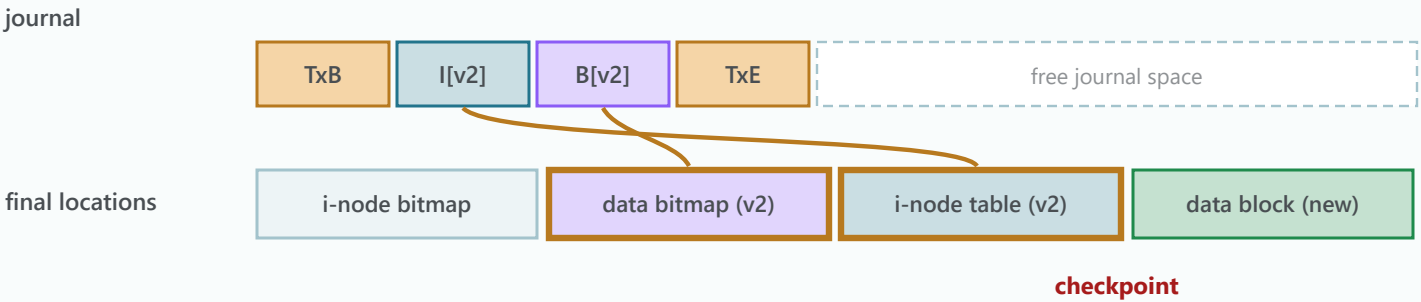
Journaling: Write to the Log First



3. Commit: once the earlier writes reach the disk (barrier/FLUSH), TxE is written. Once TxE is on disk, the transaction is **committed**.

Modes: *journal* (data journaled too: safest, slowest), *ordered*, *writeback* (no data ordering). Journaling file systems: ext3/ext4 (jbd2), XFS, NTFS (`$LogFile`), JFS. At Apple, HFS+ gave way to the CoW-based APFS.

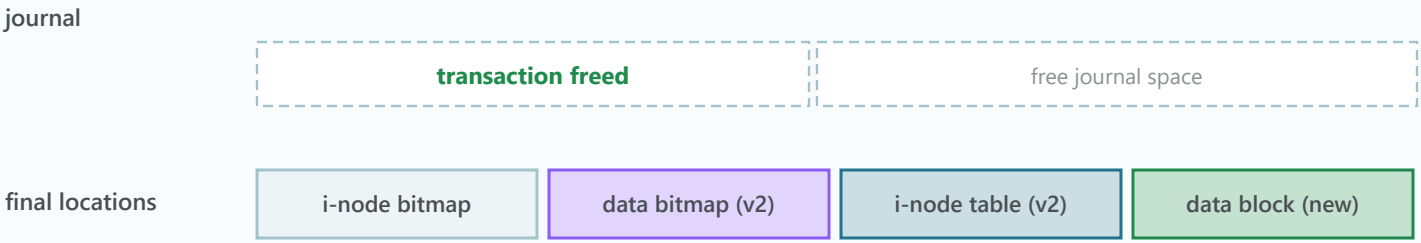
Journaling: Write to the Log First



4. Checkpoint: I and B are written to their final locations; if these writes are interrupted, they can be repeated from the journal.

Modes: *journal* (data journaled too: safest, slowest), *ordered*, *writeback* (no data ordering). Journaling file systems: ext3/ext4 (jbd2), XFS, NTFS (`$LogFile`), JFS. At Apple, HFS+ gave way to the CoW-based APFS.

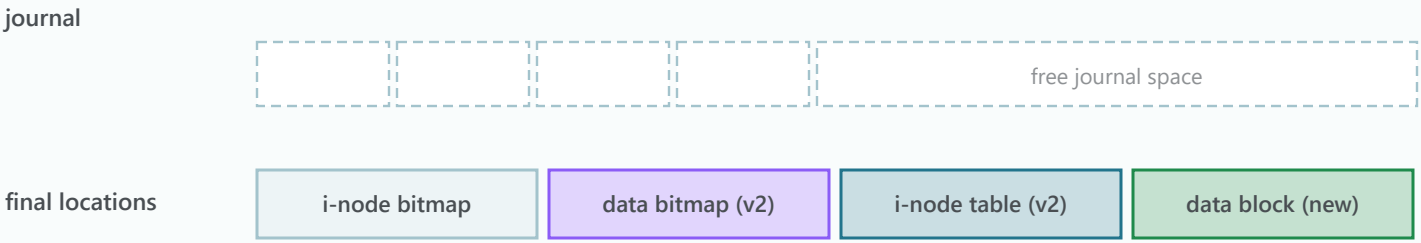
Journaling: Write to the Log First



5. Free: the transaction is marked complete in the journal superblock; the journal is reused like a circular buffer.

Modes: *journal* (data journaled too: safest, slowest), *ordered*, *writeback* (no data ordering). Journaling file systems: ext3/ext4 (jbd2), XFS, NTFS (`$LogFile`), JFS. At Apple, HFS+ gave way to the CoW-based APFS.

Journaling: Write to the Log First



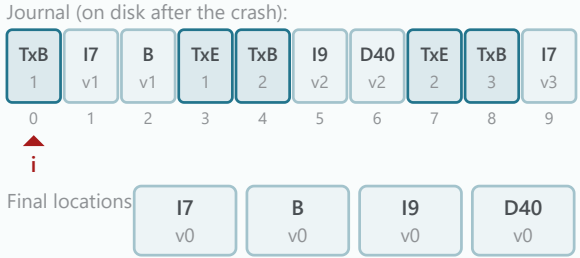
crash before TxE → the transaction is ignored; after → the journal is replayed

Recovery: at boot the journal is scanned; transactions with a TxE are replayed from the start (redo, idempotent), the others discarded. No fsck needed; it takes seconds.

Modes: *journal* (data journaled too: safest, slowest), *ordered*, *writeback* (no data ordering). Journaling file systems: ext3/ext4 (jbd2), XFS, NTFS (`$LogFile`), JFS. At Apple, HFS+ gave way to the CoW-based APFS.

Recovering from the Journal: Code Trace

```
▶ void recover(Rec g[], int n) {
▶   int i = 0;
3   while (i < n && g[i].type == TXB) {
4     int j = i + 1;
5     while (j < n && g[j].type == BLOCK)
6       j++;
7     if (j == n || g[j].type != TXE)
8       break;              /* uncommitted: drop
  */
9     for (int k = i + 1; k < j; k++)
10      write(g[k].dest, g[k].data);
11    i = j + 1;          /* next */
12  }
13 }
```



The system crashed while transaction 3 was being written. The journal holds three transaction headers; the blocks in their final locations are still old (v0). At boot, `recover` runs.

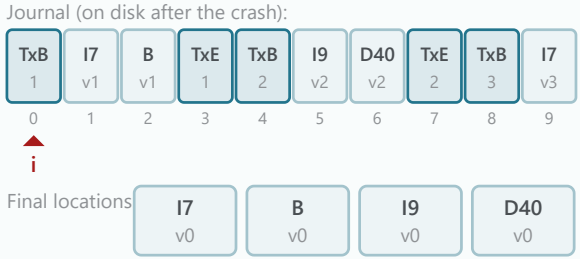
Step 1 / 12

VARIABLES

`i = 0` `j = –` `k = –`

Recovering from the Journal: Code Trace

```
1  void recover(Rec g[], int n) {
2    int i = 0;
▶    while (i < n && g[i].type == TXB) {
4      int j = i + 1;
5      while (j < n && g[j].type == BLOCK)
6        j++;
7      if (j == n || g[j].type != TXE)
8        break;              /* uncommitted: drop
   */
9      for (int k = i + 1; k < j; k++)
10       write(g[k].dest, g[k].data);
11     i = j + 1;          /* next */
12   }
13 }
```



g[0] is a TxB: transaction 1 begins.

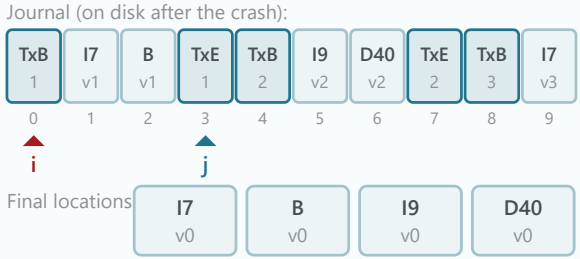
Step 2 / 12

VARIABLES

`i = 0` `j = –` `k = –`

Recovering from the Journal: Code Trace

```
1  void recover(Rec g[], int n) {
2    int i = 0;
3    while (i < n && g[i].type == TXB) {
4      int j = i + 1;
▶      while (j < n && g[j].type == BLOCK)
▶        j++;
▶      if (j == n || g[j].type != TXE)
8        break;              /* uncommitted: drop
   */
9      for (int k = i + 1; k < j; k++)
10       write(g[k].dest, g[k].data);
11     i = j + 1;           /* next */
12   }
13 }
```



At j = 3, **TxE 1** is found: the transaction is committed and its blocks can be replayed.

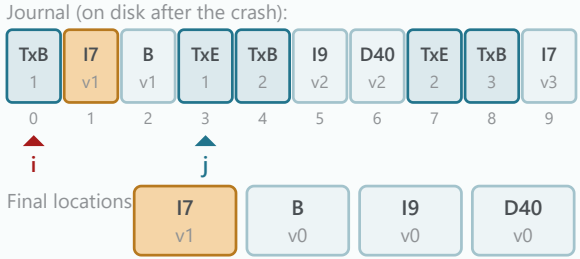
Step 3 / 12

VARIABLES

`i = 0` `j = 3` `k = –`

Recovering from the Journal: Code Trace

```
1  void recover(Rec g[], int n) {
2    int i = 0;
3    while (i < n && g[i].type == TXB) {
4      int j = i + 1;
5      while (j < n && g[j].type == BLOCK)
6        j++;
7      if (j == n || g[j].type != TXE)
8        break;              /* uncommitted: drop
   */
▶      for (int k = i + 1; k < j; k++)
▶        write(g[k].dest, g[k].data);
11     i = j + 1;          /* next */
12   }
13 }
```



I7's v1 copy is written to its final location. Even if it was written before the crash, that's fine: rewriting is idempotent.

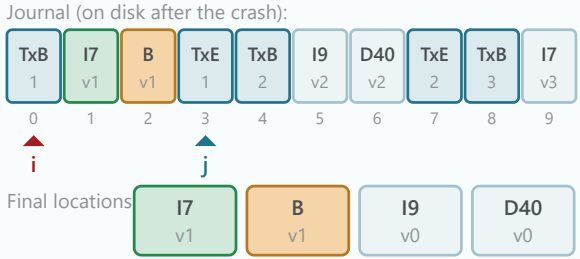
Step 4 / 12

VARIABLES

`i = 0` `j = 3` `k = 1`

Recovering from the Journal: Code Trace

```
1  void recover(Rec g[], int n) {
2    int i = 0;
3    while (i < n && g[i].type == TXB) {
4      int j = i + 1;
5      while (j < n && g[j].type == BLOCK)
6        j++;
7      if (j == n || g[j].type != TXE)
8        break;              /* uncommitted: drop
   */
▶      for (int k = i + 1; k < j; k++)
▶        write(g[k].dest, g[k].data);
11     i = j + 1;          /* next */
12   }
13 }
```



B's v1 copy is written to its final location.

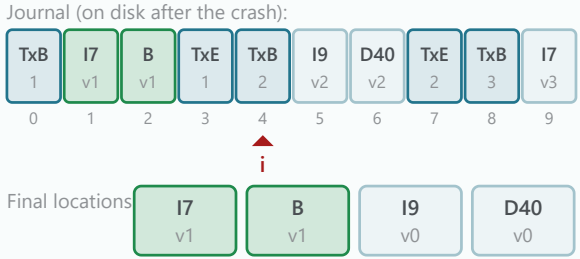
Step 5 / 12

VARIABLES

`i = 0` `j = 3` `k = 2`

Recovering from the Journal: Code Trace

```
1  void recover(Rec g[], int n) {
2    int i = 0;
▶    while (i < n && g[i].type == TXB) {
4      int j = i + 1;
5      while (j < n && g[j].type == BLOCK)
6        j++;
7      if (j == n || g[j].type != TXE)
8        break;              /* uncommitted: drop
   */
9      for (int k = i + 1; k < j; k++)
10       write(g[k].dest, g[k].data);
11     i = j + 1;          /* next */
12   }
13 }
```



g[4] is a TxB: transaction 2 begins.

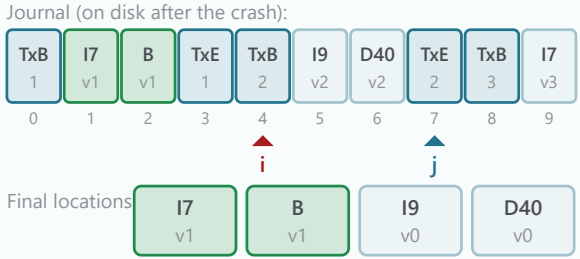
Step 6 / 12

VARIABLES

`i = 4` `j = –` `k = –`

Recovering from the Journal: Code Trace

```
1  void recover(Rec g[], int n) {
2    int i = 0;
3    while (i < n && g[i].type == TXB) {
4      int j = i + 1;
▶      while (j < n && g[j].type == BLOCK)
▶        j++;
▶      if (j == n || g[j].type != TXE)
8        break;              /* uncommitted: drop
   */
9      for (int k = i + 1; k < j; k++)
10       write(g[k].dest, g[k].data);
11     i = j + 1;           /* next */
12   }
13 }
```



At j = 7, **TxE 2** is found: the transaction is committed and its blocks can be replayed.

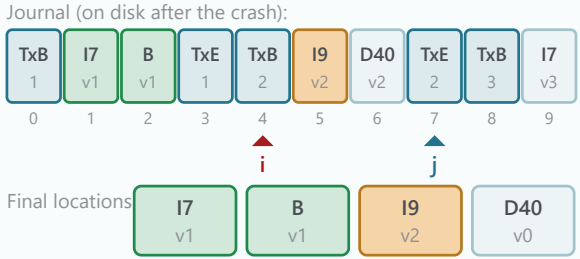
Step 7 / 12

VARIABLES

`i = 4` `j = 7` `k = –`

Recovering from the Journal: Code Trace

```
1  void recover(Rec g[], int n) {
2    int i = 0;
3    while (i < n && g[i].type == TXB) {
4      int j = i + 1;
5      while (j < n && g[j].type == BLOCK)
6        j++;
7      if (j == n || g[j].type != TXE)
8        break;              /* uncommitted: drop
   */
▶      for (int k = i + 1; k < j; k++)
▶        write(g[k].dest, g[k].data);
11     i = j + 1;          /* next */
12   }
13 }
```



I9's v2 copy is written to its final location.

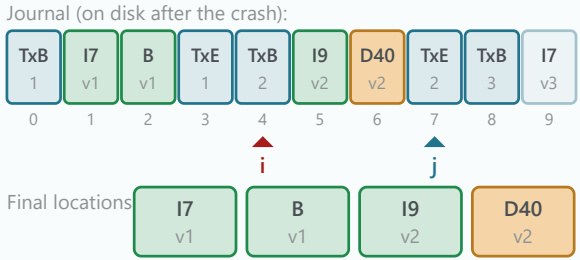
Step 8 / 12

VARIABLES

`i = 4` `j = 7` `k = 5`

Recovering from the Journal: Code Trace

```
1  void recover(Rec g[], int n) {
2    int i = 0;
3    while (i < n && g[i].type == TXB) {
4      int j = i + 1;
5      while (j < n && g[j].type == BLOCK)
6        j++;
7      if (j == n || g[j].type != TXE)
8        break;              /* uncommitted: drop
   */
▶      for (int k = i + 1; k < j; k++)
▶        write(g[k].dest, g[k].data);
11     i = j + 1;          /* next */
12   }
13 }
```



D40's v2 copy is written to its final location.

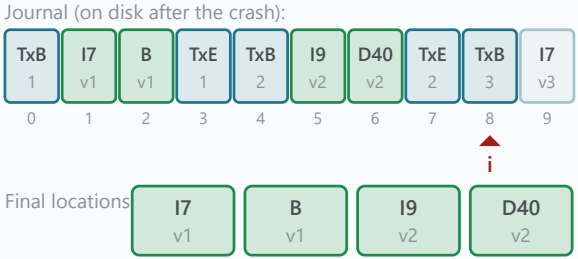
Step 9 / 12

VARIABLES

`i = 4` `j = 7` `k = 6`

Recovering from the Journal: Code Trace

```
1  void recover(Rec g[], int n) {
2    int i = 0;
▶    while (i < n && g[i].type == TXB) {
4      int j = i + 1;
5      while (j < n && g[j].type == BLOCK)
6        j++;
7      if (j == n || g[j].type != TXE)
8        break;              /* uncommitted: drop
   */
9      for (int k = i + 1; k < j; k++)
10       write(g[k].dest, g[k].data);
11     i = j + 1;           /* next */
12   }
13 }
```



g[8] is a TxB: transaction 3 begins.

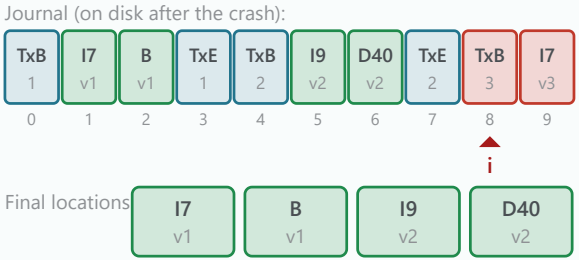
Step 10 / 12

VARIABLES

`i = 8` `j = –` `k = –`

Recovering from the Journal: Code Trace

```
1  void recover(Rec g[], int n) {
2    int i = 0;
3    while (i < n && g[i].type == TXB) {
4      int j = i + 1;
▶      while (j < n && g[j].type == BLOCK)
▶        j++;
▶      if (j == n || g[j].type != TXE)
▶        break;              /* uncommitted: drop
   */
9      for (int k = i + 1; k < j; k++)
10       write(g[k].dest, g[k].data);
11     i = j + 1;          /* next */
12   }
13 }
```



Skipping the block records, j = 10 = n: the journal ended but there's **no TxE**. Transaction 3 wasn't committed: it is **discarded**, and I7's v3 is never written.

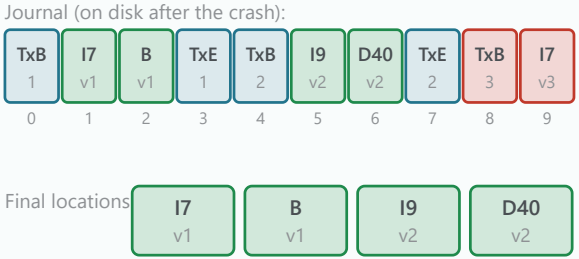
Step 11 / 12

VARIABLES

`i = 8` `j = 10` `k = –`

Recovering from the Journal: Code Trace

```
1  void recover(Rec g[], int n) {
2    int i = 0;
3    while (i < n && g[i].type == TXB) {
4      int j = i + 1;
5      while (j < n && g[j].type == BLOCK)
6        j++;
7      if (j == n || g[j].type != TXE)
8        break;              /* uncommitted: drop
   */
9      for (int k = i + 1; k < j; k++)
10       write(g[k].dest, g[k].data);
11     i = j + 1;          /* next */
12   }
▶ }
```



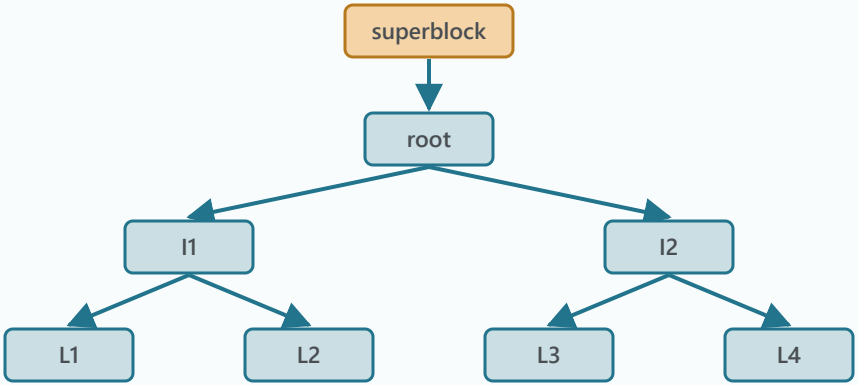
Result: transactions 1 and 2 were applied in full, transaction 3 as if it never happened. The file system is consistent; no whole-disk fsck is needed.

Step 12 / 12

VARIABLES

`i = 8` `j = 10` `k = –`

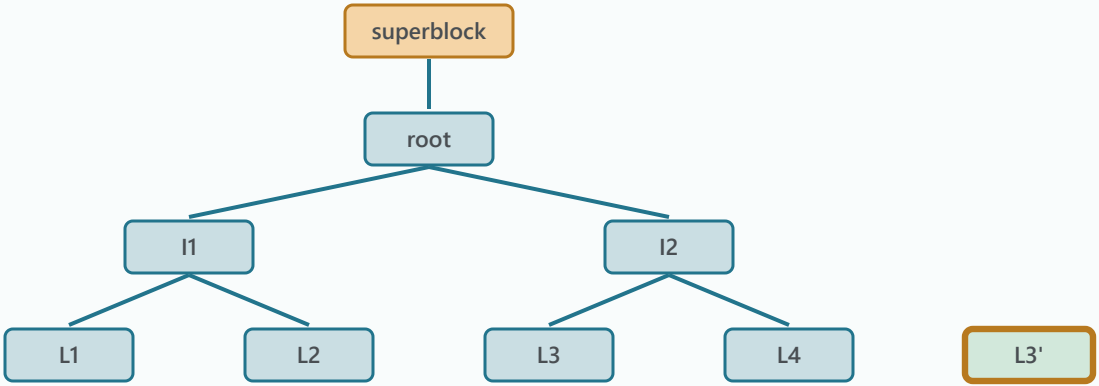
Copy-on-Write (CoW): ZFS, Btrfs, APFS



The file system is a tree of blocks: superblock (the *uberblock* in ZFS) → root → internal nodes → data.

ZFS (Sun 2005, now OpenZFS), Btrfs (Linux), APFS (Apple, 2017), ReFS (Windows Server). The cost: fragmentation and write amplification on random overwrites; CoW can be turned off for database files.

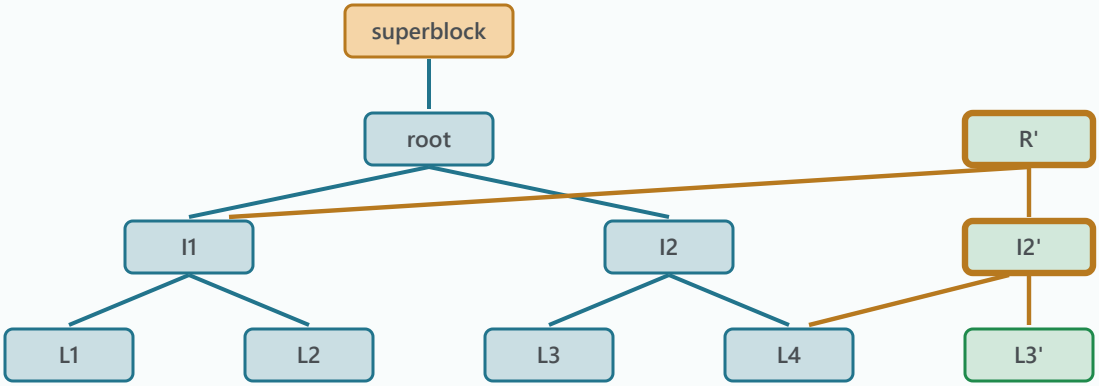
Copy-on-Write (CoW): ZFS, Btrfs, APFS



L3 is to change: it is **not written in place**; a new copy, L3', goes to a free location. The old L3 is still intact.

ZFS (Sun 2005, now OpenZFS), Btrfs (Linux), APFS (Apple, 2017), ReFS (Windows Server). The cost: fragmentation and write amplification on random overwrites; CoW can be turned off for database files.

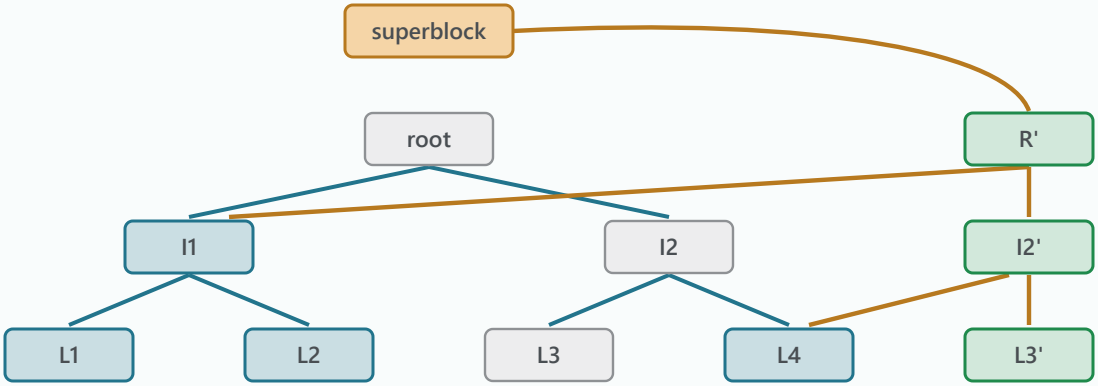
Copy-on-Write (CoW): ZFS, Btrfs, APFS



The parents pointing to L3' are copied too: I2', root R'. Unchanged subtrees (I1, L4) are shared.

ZFS (Sun 2005, now OpenZFS), Btrfs (Linux), APFS (Apple, 2017), ReFS (Windows Server). The cost: fragmentation and write amplification on random overwrites; CoW can be turned off for database files.

Copy-on-Write (CoW): ZFS, Btrfs, APFS

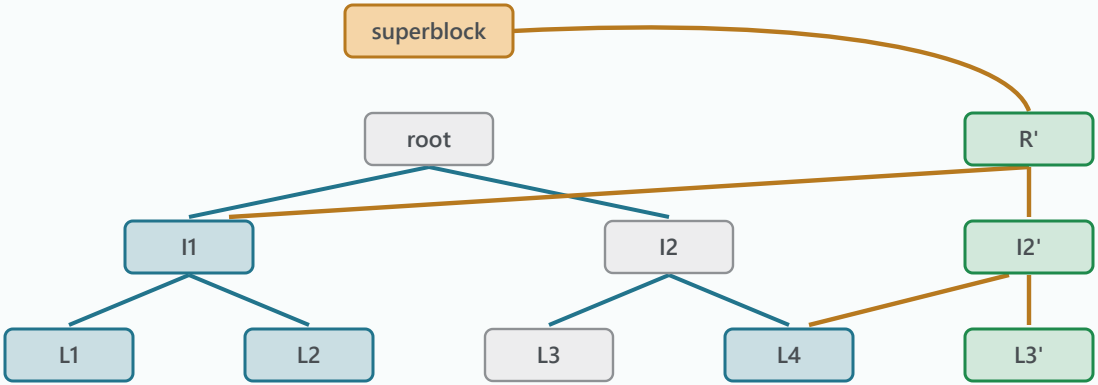


gray: the old version = a snapshot (or free)

One atomic write: the superblock points to R'. After a crash either the old or the new tree is visible, never an intermediate state; no journal is needed.

ZFS (Sun 2005, now OpenZFS), Btrfs (Linux), APFS (Apple, 2017), ReFS (Windows Server). The cost: fragmentation and write amplification on random overwrites; CoW can be turned off for database files.

Copy-on-Write (CoW): ZFS, Btrfs, APFS



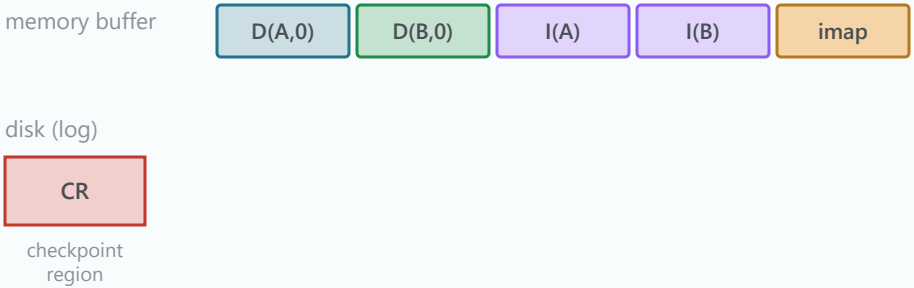
gray: the old version = a snapshot (or free)

every pointer carries the child's checksum

If the old root is kept, a **snapshot** is free. ZFS/Btrfs keep a checksum in every pointer: silent corruption is caught on read and repaired from a mirror (*scrub*).

ZFS (Sun 2005, now OpenZFS), Btrfs (Linux), APFS (Apple, 2017), ReFS (Windows Server). The cost: fragmentation and write amplification on random overwrites; CoW can be turned off for database files.

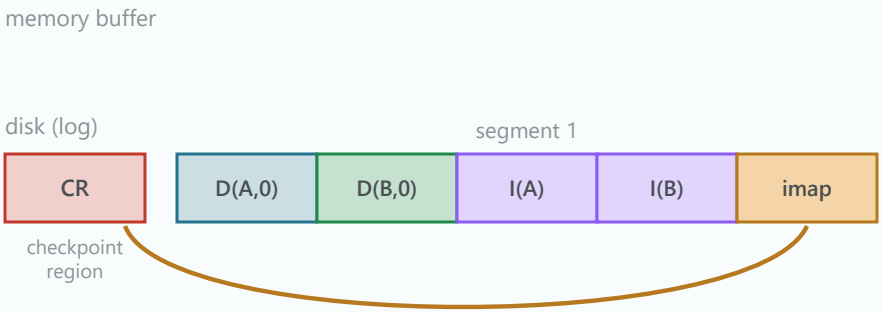
The Log-Structured File System (LFS)



Writes are buffered in memory: data blocks, modified i-nodes, and pieces of the i-node map (imap).

Rosenblum and Ousterhout, Sprite LFS (1991). Designed on the assumption that reads are served from a large cache; it brings write bandwidth close to the disk's sequential speed. The idea lives on in F2FS (Android), NILFS2, NetApp WAFL, ZFS/Btrfs CoW, and SSD FTLs.

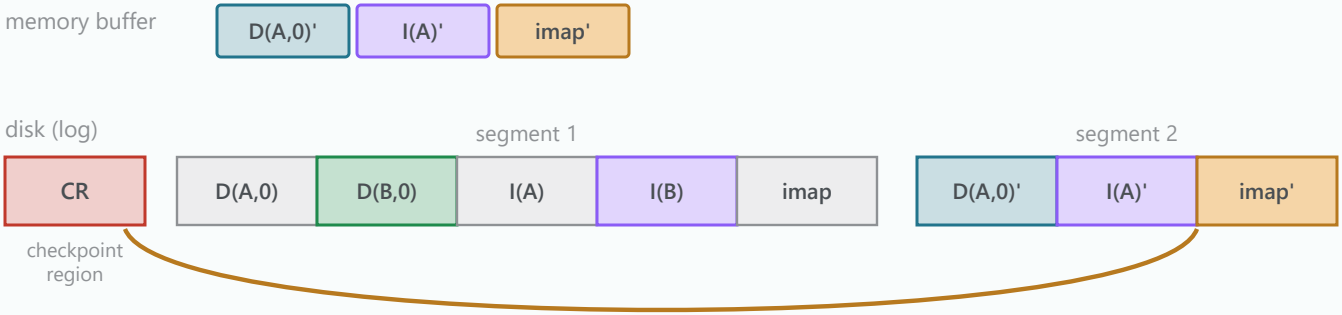
The Log-Structured File System (LFS)



When the buffer fills, everything is written **sequentially** to the end of the log as one **segment** (e.g., 1–2 MiB). i-nodes have no fixed location; the **imap** finds them, and the checkpoint region (CR) points to the latest imap.

Rosenblum and Ousterhout, Sprite LFS (1991). Designed on the assumption that reads are served from a large cache; it brings write bandwidth close to the disk's sequential speed. The idea lives on in F2FS (Android), NILFS2, NetApp WAFL, ZFS/Btrfs CoW, and SSD FTLs.

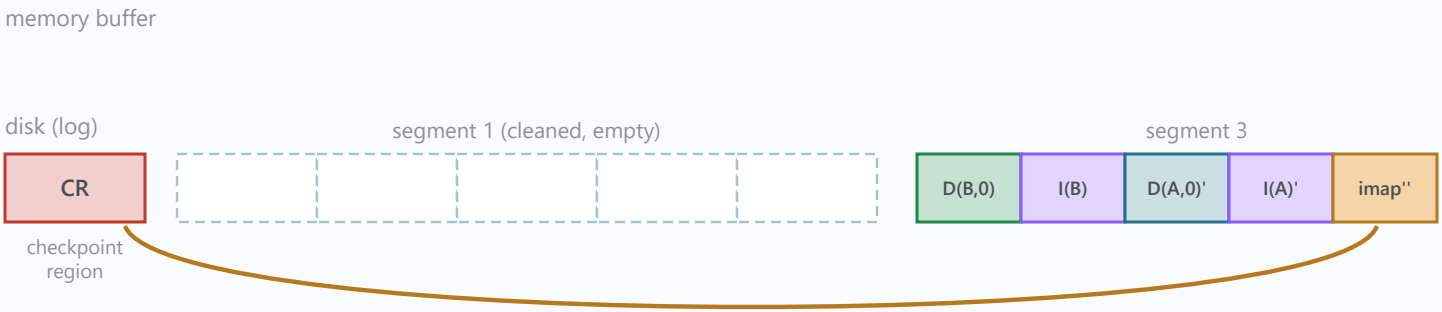
The Log-Structured File System (LFS)



When A's block changes, the new version goes to a new segment; the old D(A,0), I(A), and imap in segment 1 become **dead**.

Rosenblum and Ousterhout, Sprite LFS (1991). Designed on the assumption that reads are served from a large cache; it brings write bandwidth close to the disk's sequential speed. The idea lives on in F2FS (Android), NILFS2, NetApp WAFL, ZFS/Btrfs CoW, and SSD FTLs.

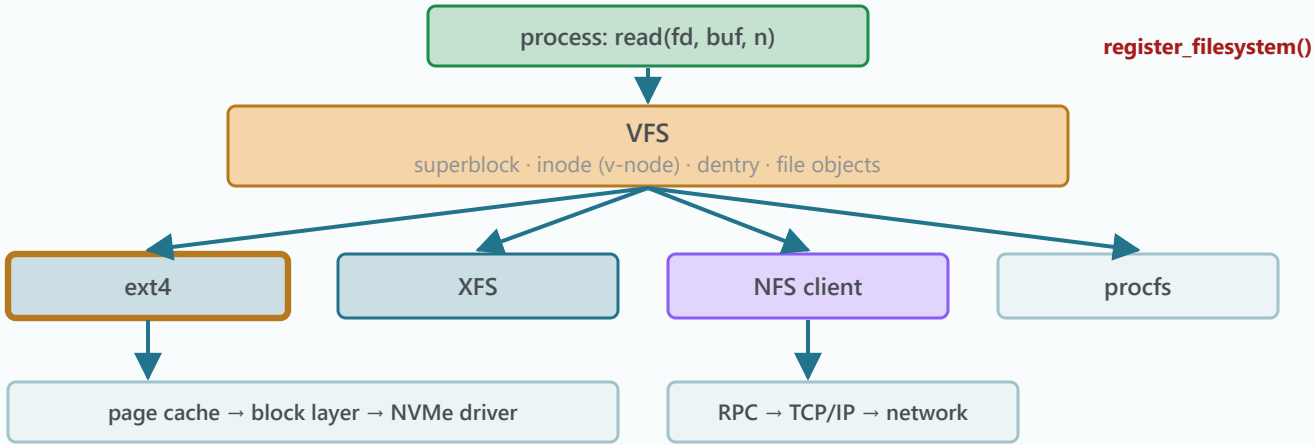
The Log-Structured File System (LFS)



The cleaner: it reads segments with few live blocks, compacts the live ones into a new segment, and frees the old. Cleaning cost under load is LFS's weak spot.

Rosenblum and Ousterhout, Sprite LFS (1991). Designed on the assumption that reads are served from a large cache; it brings write bandwidth close to the disk's sequential speed. The idea lives on in F2FS (Android), NILFS2, NetApp WAFL, ZFS/Btrfs CoW, and SSD FTLs.

The Virtual File System (VFS)

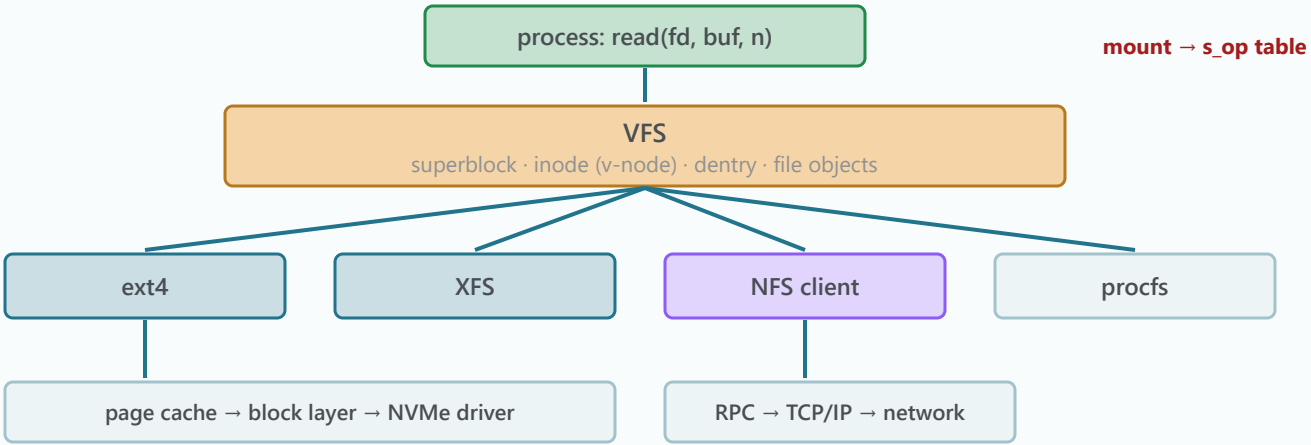


Registration: a file system registers with the VFS at boot or when its module loads, handing over a table of the functions the VFS will call.

Step 1 / 4

The VFS's purpose is to offer one interface to different file systems *at the same time* (several systems on one disk are already possible via partitioning). Its overhead is one indirect call; the performance loss is small. The Windows counterpart is the I/O manager and file-system drivers.

The Virtual File System (VFS)

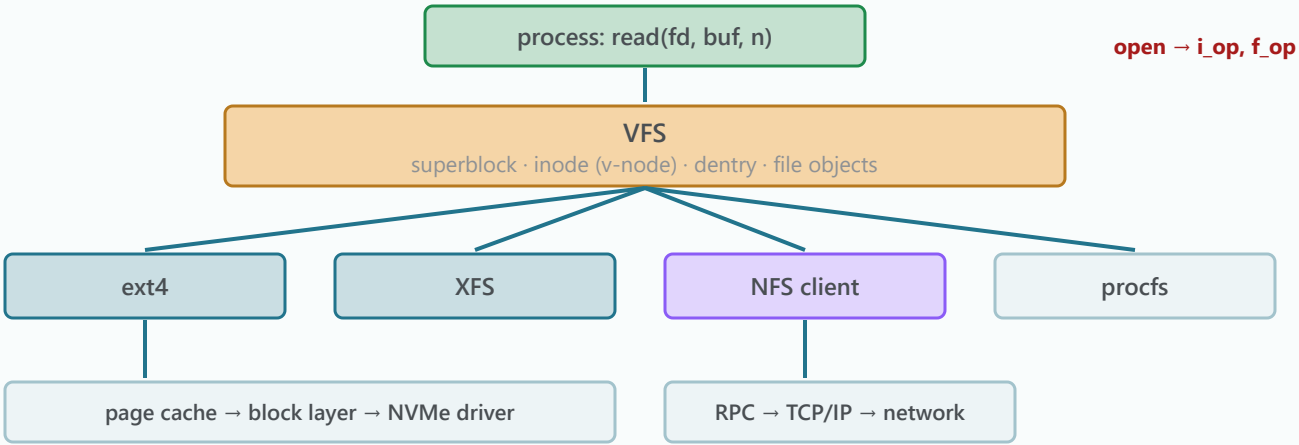


Mount: the VFS asks that file system to build a superblock object, which carries the operations table (`s_op`).

Step 2 / 4

The VFS's purpose is to offer one interface to different file systems *at the same time* (several systems on one disk are already possible via partitioning). Its overhead is one indirect call; the performance loss is small. The Windows counterpart is the I/O manager and file-system drivers.

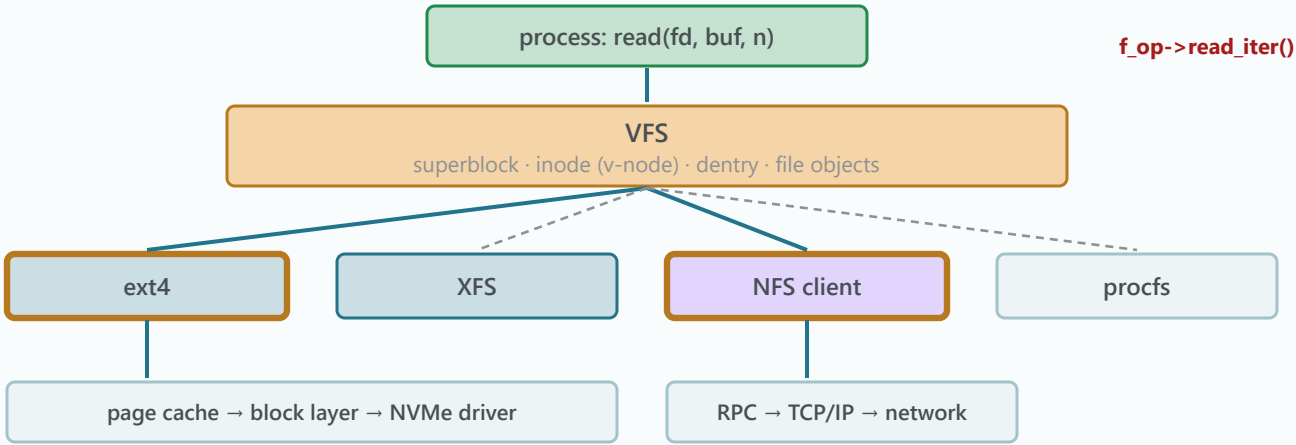
The Virtual File System (VFS)



open: the path is resolved through dentries; the concrete system fills a VFS i-node (v-node) with its i-node data; a `file` object is created and added to the fd table.

The VFS's purpose is to offer one interface to different file systems *at the same time* (several systems on one disk are already possible via partitioning). Its overhead is one indirect call; the performance loss is small. The Windows counterpart is the I/O manager and file-system drivers.

The Virtual File System (VFS)



read: the VFS just calls the `file->f_op->read_iter` function pointer; for an ext4 file ext4's function runs, for an NFS file an RPC goes over the network. The application sees no difference.

The VFS's purpose is to offer one interface to different file systems *at the same time* (several systems on one disk are already possible via partitioning). Its overhead is one indirect call; the performance loss is small. The Windows counterpart is the I/O manager and file-system drivers.

The Network File System (NFS)



- A remote directory is **mounted** into the local tree, and applications use it like local files. Sun designed it for SunOS in 1984, and it became an open standard.
- Machine and architecture independence come from **RPC** and **XDR**. NFSv3 is stateless (UDP or TCP); **NFSv4** is stateful, TCP only, on a single port (2049), with Kerberos security, ACLs, locking, and delegation; v4.1 adds pNFS.
- The Windows-world counterpart is SMB 3 (encryption, multichannel). NFS originally ran over UDP/IP; today TCP is preferred; the only requirement is an IP network.

Summary

Files and directories

File: a named, persistent byte sequence + attributes. Directory: a name → i-node mapping; a tree + links, one tree via mounting.

Open files

fd → system-wide entry (position) → in-memory i-node. Path resolution proceeds component by component.

Allocation

Contiguous (fast, external fragmentation), linked/FAT (flexible, poor random access), i-node/extents (today's solution).

Free space

Bitmap, list, grouping, counting; block size is a trade-off between speed and space.

Performance

A unified page cache, read-ahead, nearby placement; TRIM on SSDs.

Consistency

fsck → journaling → CoW/LFS. Backups are still a must: 3-2-1.

SUMMARY

Review Questions

1. In a loop reading a 6000-byte file with a 4096-byte buffer, what does each `read` return?
2. After `ln a b` and `ln -s a c`, what happens to b and c if you run `rm a`?
3. How many disk reads does resolving `/usr/ast/mbox` take without caching (root i-node in memory)?
4. 200 GiB disk, 1 KiB blocks, 4 B entries: how much memory does the FAT need?
5. 4 KiB blocks, 4 B pointers, 12 direct + single/double/triple indirect: the largest file?
6. In the same structure, at what level is byte 5,000,000, and how many reads does it take?
7. Free blocks 2–5, 8–13, 17–18, 25–27: show them with the counting method.
8. In journaling, what is the difference between a crash before and after TxE is written?
9. Why is TRIM needed when a file is deleted on an SSD?
10. The i-node table starts at block 3, blocks are 4 KiB, i-nodes 256 B (numbered from 0): which block and sector hold i-node 40?

▶ **Answers**

References

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 (Chapters 13 File-System Interface, 14 File-System Implementation, 15 File-System Internals).
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson (Chapter 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 et al., "A Fast File System for UNIX", ACM TOCS, 1984.
- Linux kernel documentation: "Overview of the Linux Virtual File System", "ext4 Data Structures and Algorithms"; Microsoft Learn: NTFS and the exFAT specification; OpenZFS and Btrfs documentation; the UEFI specification (GPT).

The End

Chapter 10: File Systems

Previous: Chapter 9: Virtual Memory · Next: Chapter 11: Input/Output