文件系统

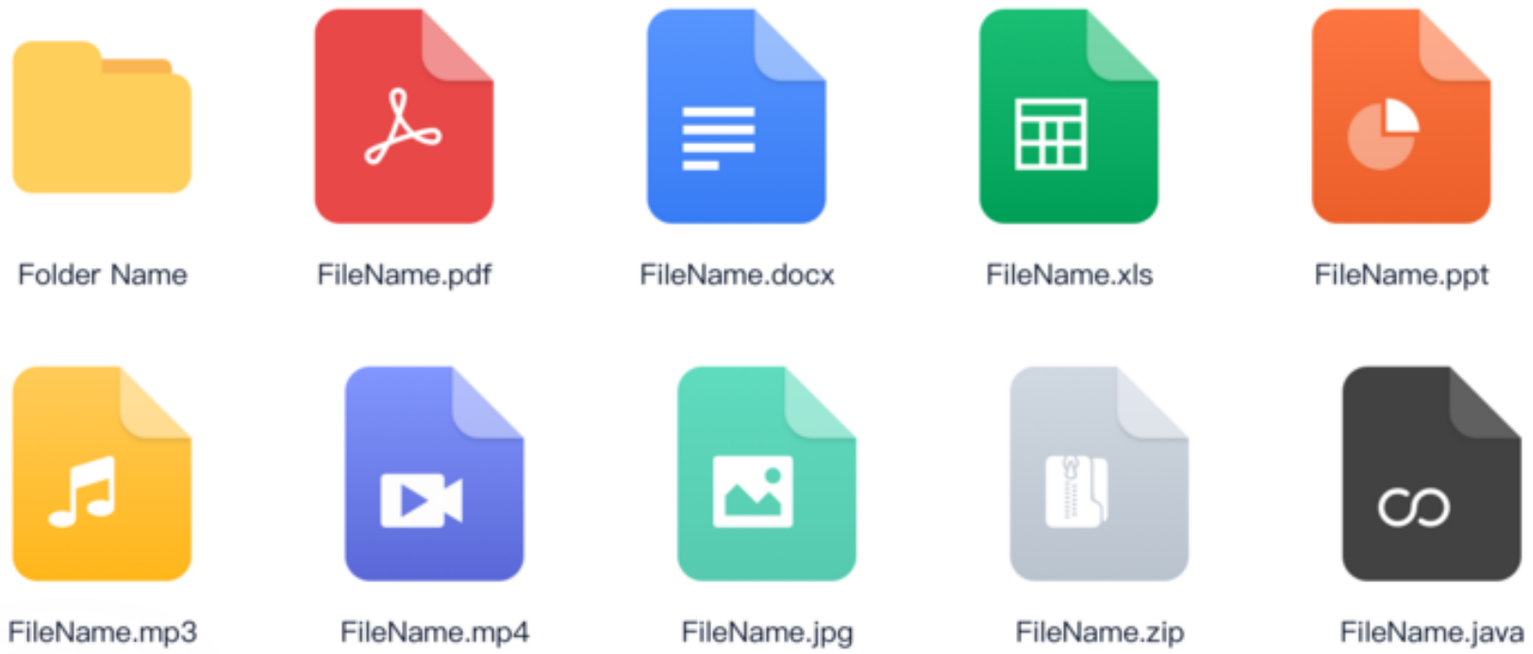
Section 5: Part I

2024

文件

数据持久化存储是现代操作系统的重要功能之一

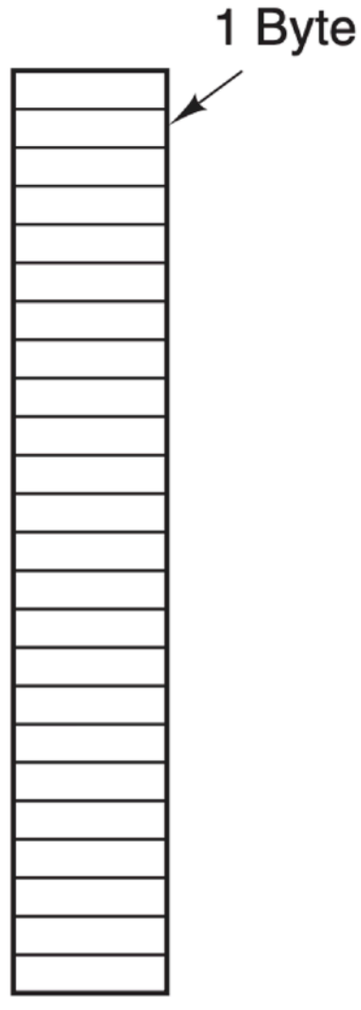
- 应用程序数据 (可执行文件、动态链接库 ...)
- 用户数据 (文档、图片、视频 ...)
- 系统数据 (配置文件 ...)



文件

文件 (File) 是操作系统为存储信息而创建的一个抽象概念 (提供了一种将信息存储在磁盘上并在随后进行访问的机制)

- 一个文件就是一个按名存取的字符序列 (a linear array of bytes)
- 包含两个部分
  - 文件数据 (File Data)：字符序列
    - 用户可以对其进行创建、读取、写入和删除等操作
    - 具体内容由应用程序负责解释 (for flexibility)



文件

文件 (File) 是操作系统为存储信息而创建的一个抽象概念 (提供了一种将信息存储在磁盘上并在随后进行访问的机制)

- 一个文件就是一个按名存取的字符序列 (a linear array of bytes)
- 包含两个部分
  - 文件数据 (File Data)：字符序列
  - 文件属性 (File Metadata)：用于支撑文件功能的其他信息
    - ***Size***: 文件大小 (字符序列的长度)
    - ***Owner & Protection***: 文件的所有者、不同用户对文件的访问权限
    - ***Time***: 文件的创建时间、最近访问时间、最近修改时间
    - …

文件

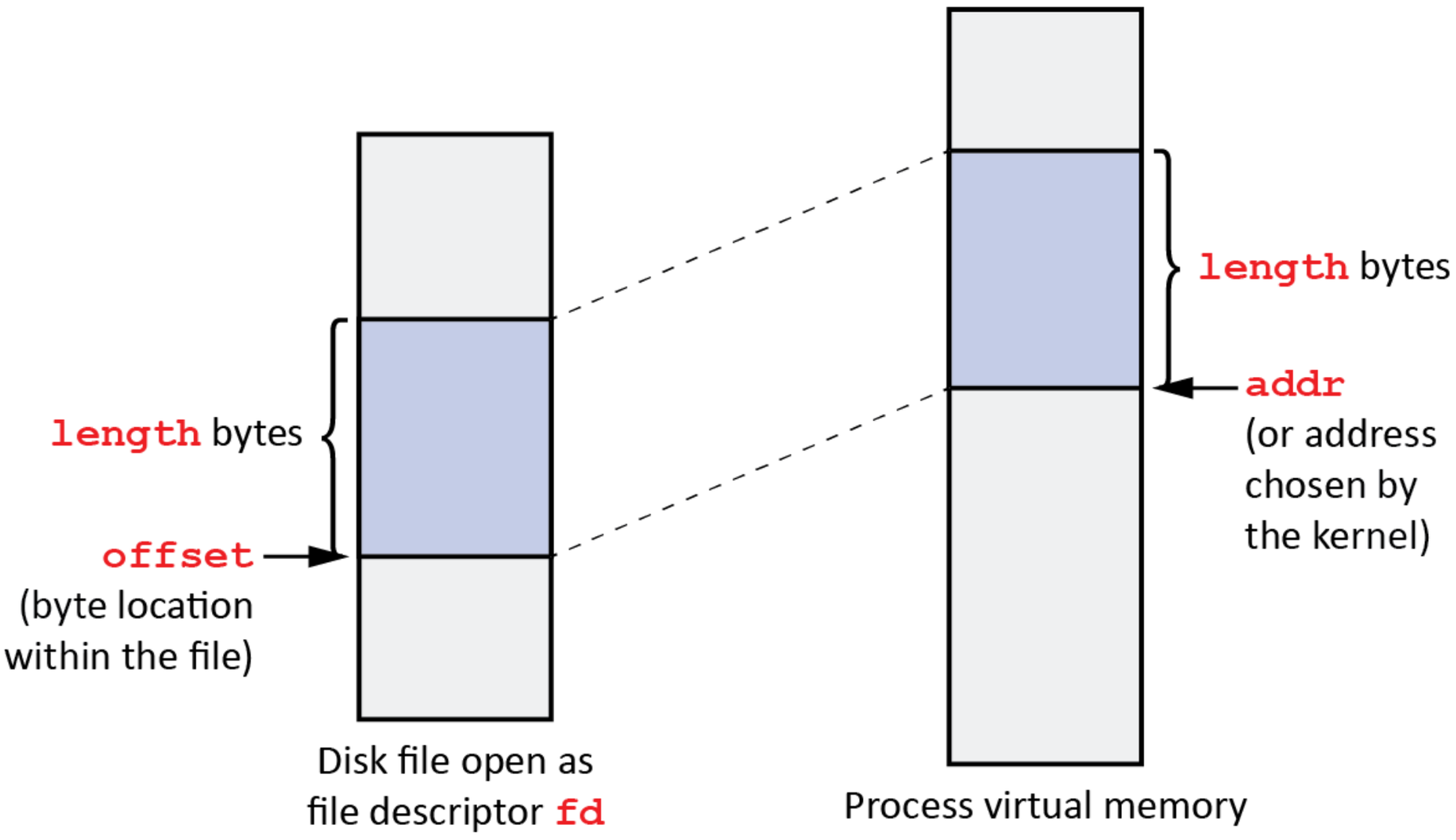
用户程序使用一组特定的文件操作对文件进行访问

- `open()`:打开一个文件
- `read()`:顺序读取一个打开文件中的若干字节
- `write()`:顺序向一个打开文件中写入若干字节
- `lseek()`:移动当前的偏移量
- `fsync()`:将当前对文件的修改立即写回磁盘
- `close()`:关闭一个已打开的文件
- `stat()`:获取文件的属性信息
- ...

文件

内存映射文件 (Memory-Mapped File): 将文件字节序列映射到进程虚拟地址空间，随后进程可以使用内存访问指令来直接访问文件内容

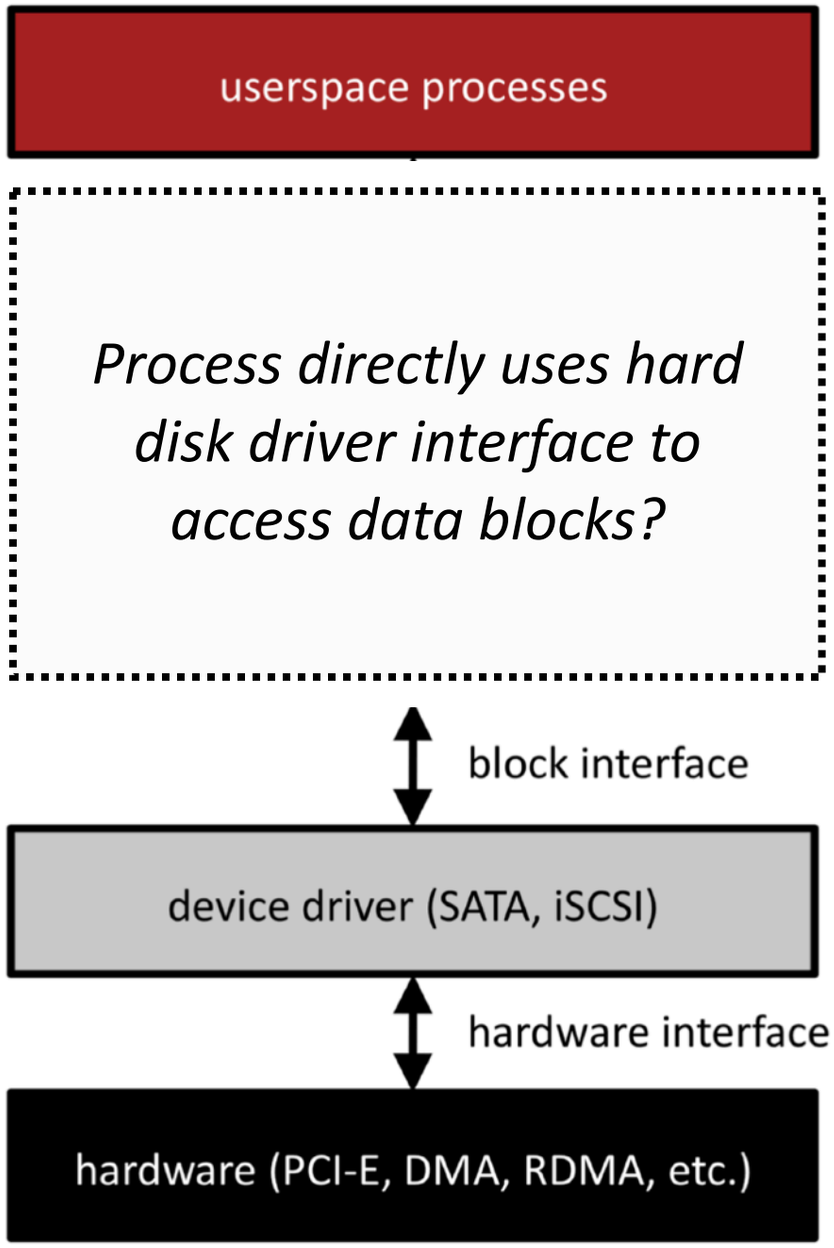
```
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset)
```



文件系统

实现文件接口并负责管理文件数据和属性的系统即为文件系统

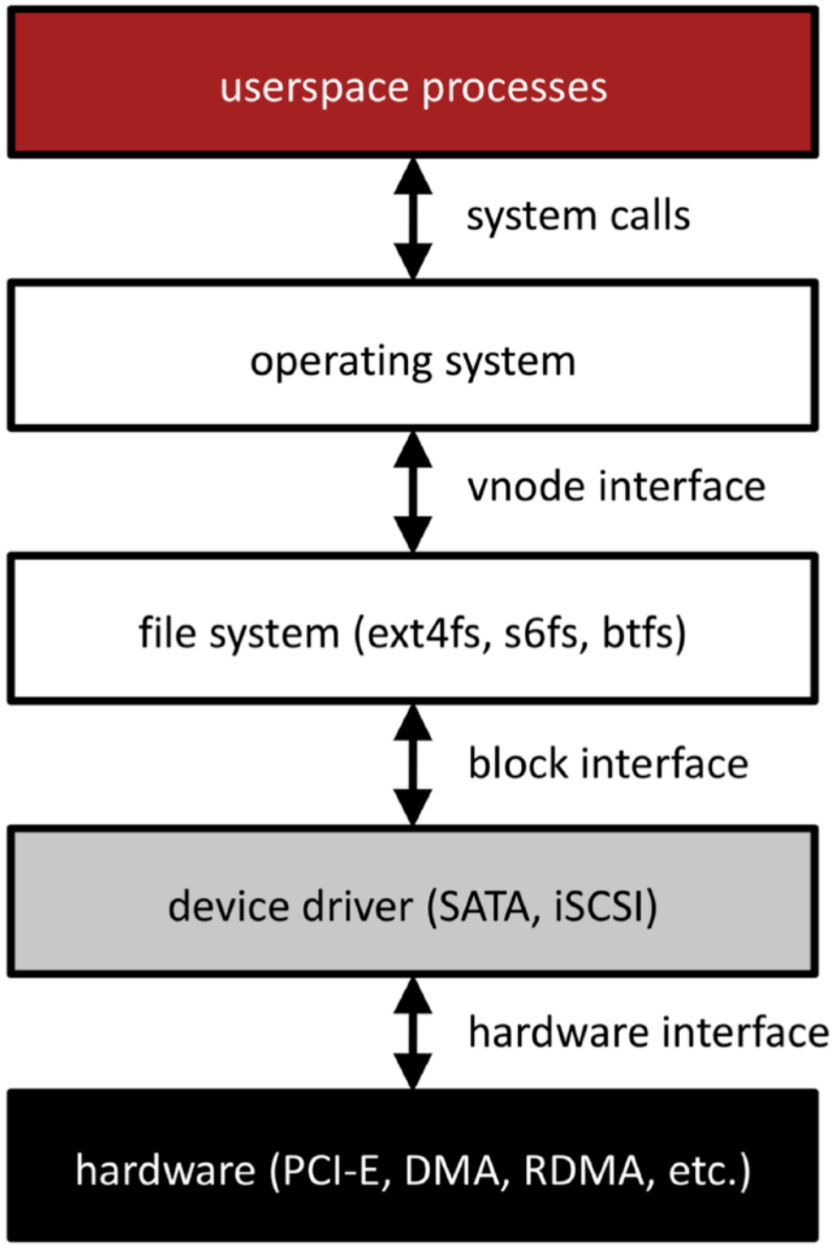
- 数据需保存在存储设备中以实现持久化
- 磁盘驱动提供 Block Interface 来从磁盘中读取和写入数据
  - `read(block k)`
  - `write(block k)`



文件系统

实现文件接口并负责管理文件数据和属性的系统即为文件系统

- 构建对数据持久化存储的分层抽象
  - 文件 (File) 和目录 (Directory)
- 向用户程序提供统一简洁的 API 以方便对文件的操作 (user's views)
- 以特定格式在存储设备上维护每个文件的数据和属性信息 (hard disk driver's view)
  - 高效利用磁盘空间 (allowing data to be stored, located, and retrieved easily)
  - 尽可能可靠 (do not loss data)



文件命名

命名 (Naming) 是文件系统抽象的一个重要特征

- 用一个具有意义的单一名称来引用系统中的特定数据
- 同一个文件在不同场景下具有不同的名字
  - ***Path Name:*** 易于人类理解的名称 (human friendly name)
  - ***File Descriptor:*** 进程运行过程中的一个已打开文件
  - ***Low-Level Name:*** 文件系统内部实现中使用的名称

文件控制块

File Control Block: 为了实现对文件数据和属性的存储和管理而维护的特定数据结构 (该结构同样需持久化存储在磁盘中)

- 在 Unix 文件系统中，该结构被称为 inode (index node)

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Simplified ext2 inode structure

对文件操作的典型流程

```
// a typical file operation process (ignore error handling)
int in_fd, out_fd, rd_count, wt_count;
char buffer[BUFFER_SIZE];

// open the files
in_fd = open(input_filename, O_RDONLY);
out_fd = creat(output_filename, OUTPUT_MODE);

// copy data from input file to output file
while (1) {
  rd_count = read(in_fd, buffer, BUFFER_SIZE);
  if (rd_count <= 0) // end of the file
    break;
  wt_count = write(out_fd, buffer, rd_count);
}

// close the files
close(in_fd);
close(out_fd);
```

文件描述符

File Descriptor (Handler): 文件系统为进程每打开的一个文件分配的一个整型编号 (private per process)

- 执行文件特定操作的一种能力 (like a pointer)
- 在打开文件时，需要将文件路径名转换为文件系统中的内部名称 (e.g., human friendly name → inode number)
  - 避免了每次读写文件操作都需要进行名称转换和权限检查
  - 同一个文件还可以以不同的方式打开 (e.g., read only / read write)

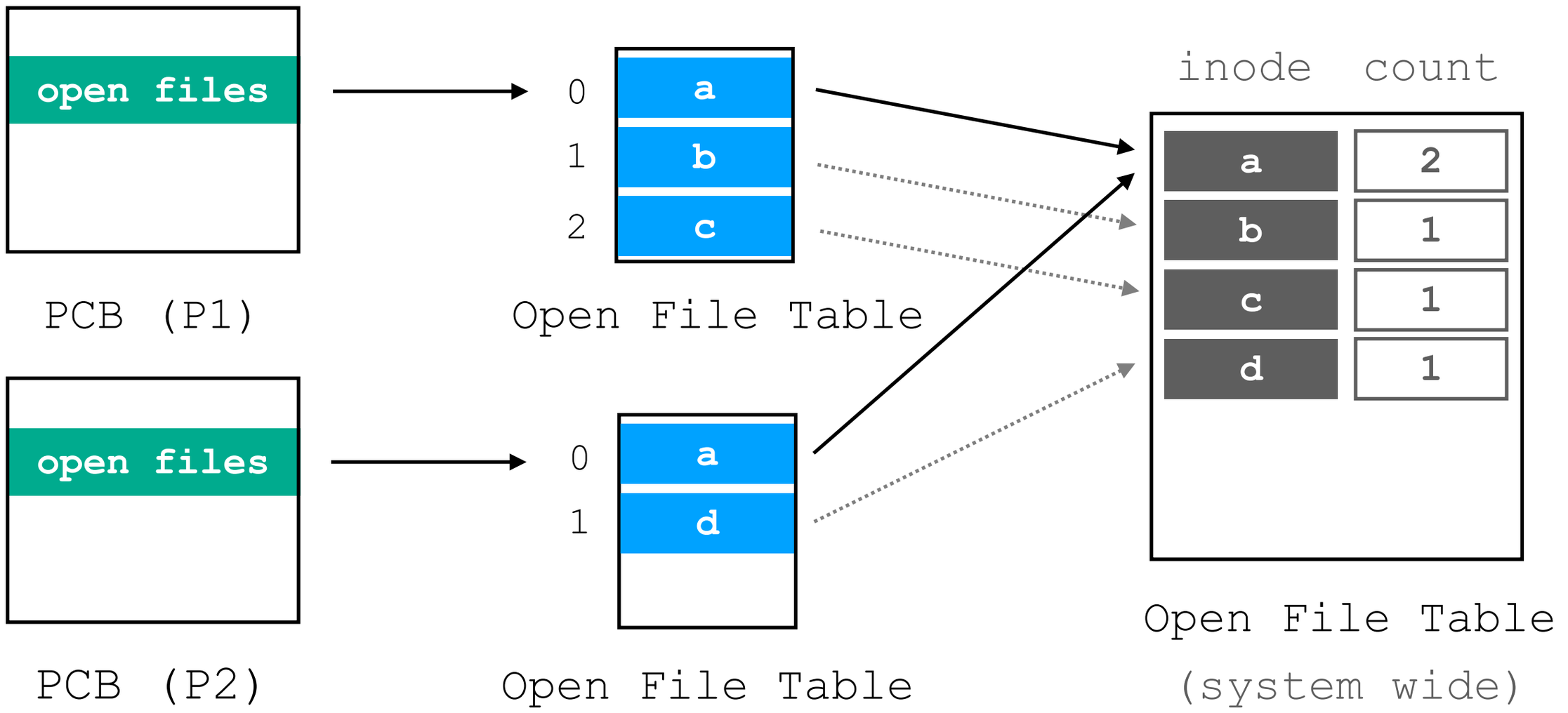
打开文件表

利用打开文件表 (open file table) 建立 file descriptor 和 inode 之间的关联

- 每个进程维护一个进程打开文件表 (per-process)
  - 一个由 file descriptor 索引的数组
  - 表中的每一项包含一个指向系统打开文件表的指针
- 整个系统维护一个系统打开文件表 (system-wide)
  - 同一个文件可以被多个进程打开
  - 在打开文件时，先在系统打开文件表中进行搜索
    - 如果已打开，则可直接让进程打开文件表的表项指向对应系统打开文件表的表项，并更新 open count
    - 只有当所有进程都关闭该文件时才能删除该表项

打开文件表

利用打开文件表 (open file table) 建立 file descriptor 和 inode 之间的关联



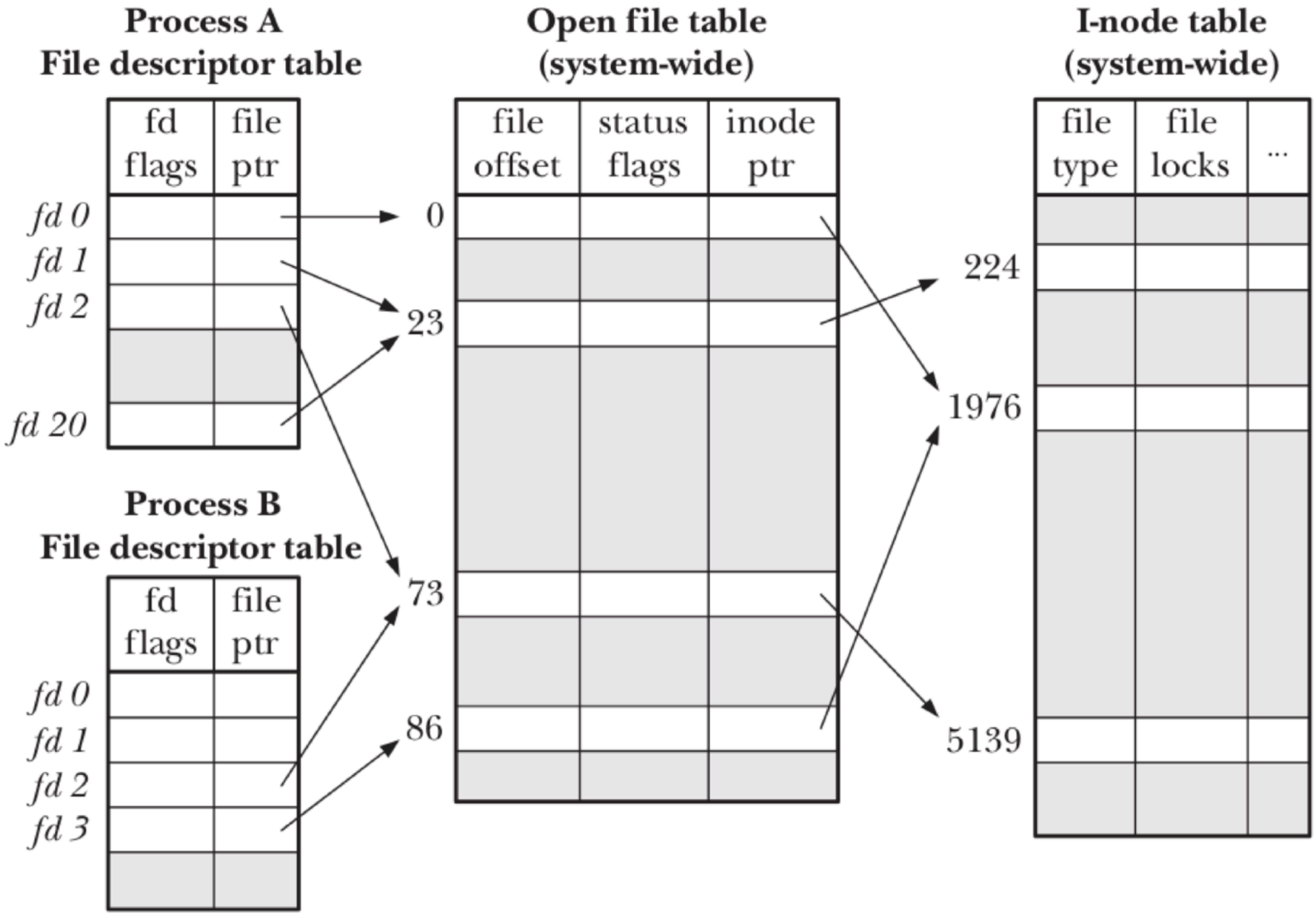
应该在哪个打开文件表中记录文件偏移量 (offset) 信息?

打开文件表

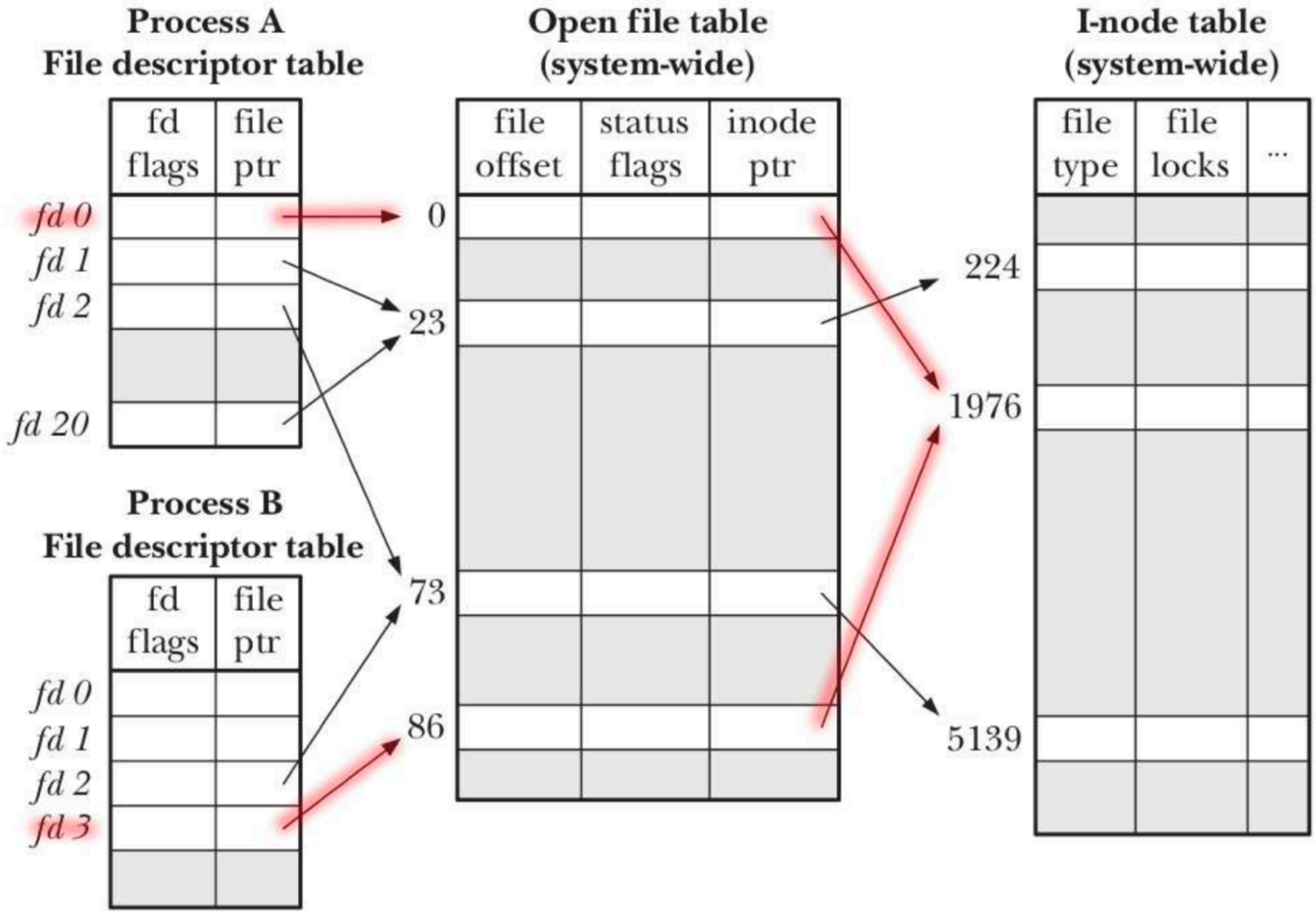
Unix 文件系统通过维护三类打开文件表来描述不同 file descriptors 和不同 open files 之间的关系

- File Descriptor Table (per-process): 每项对应进程打开的一个文件
- Open File Table (system-wide): 每项对应系统当前已打开的一个文件
  - 文件偏移量 (offset)、打开方式 (access mode) 以及相关的 flags
- inode table (system-wide): 每项对应一个 inode
  - 文件的静态属性信息 (type, size, permissions, timestamps, …)

打开文件表


Process A
File descriptor table
fd flags
file ptr
fd 0
fd 1
fd 2
fd 20
Process B
File descriptor table
fd flags
file ptr
fd 0
fd 1
fd 2
fd 3
Open file table (system-wide)
file offset
status flags
inode ptr
0
23
73
86
I-node table (system-wide)
file type
file locks
...
224
1976
5139

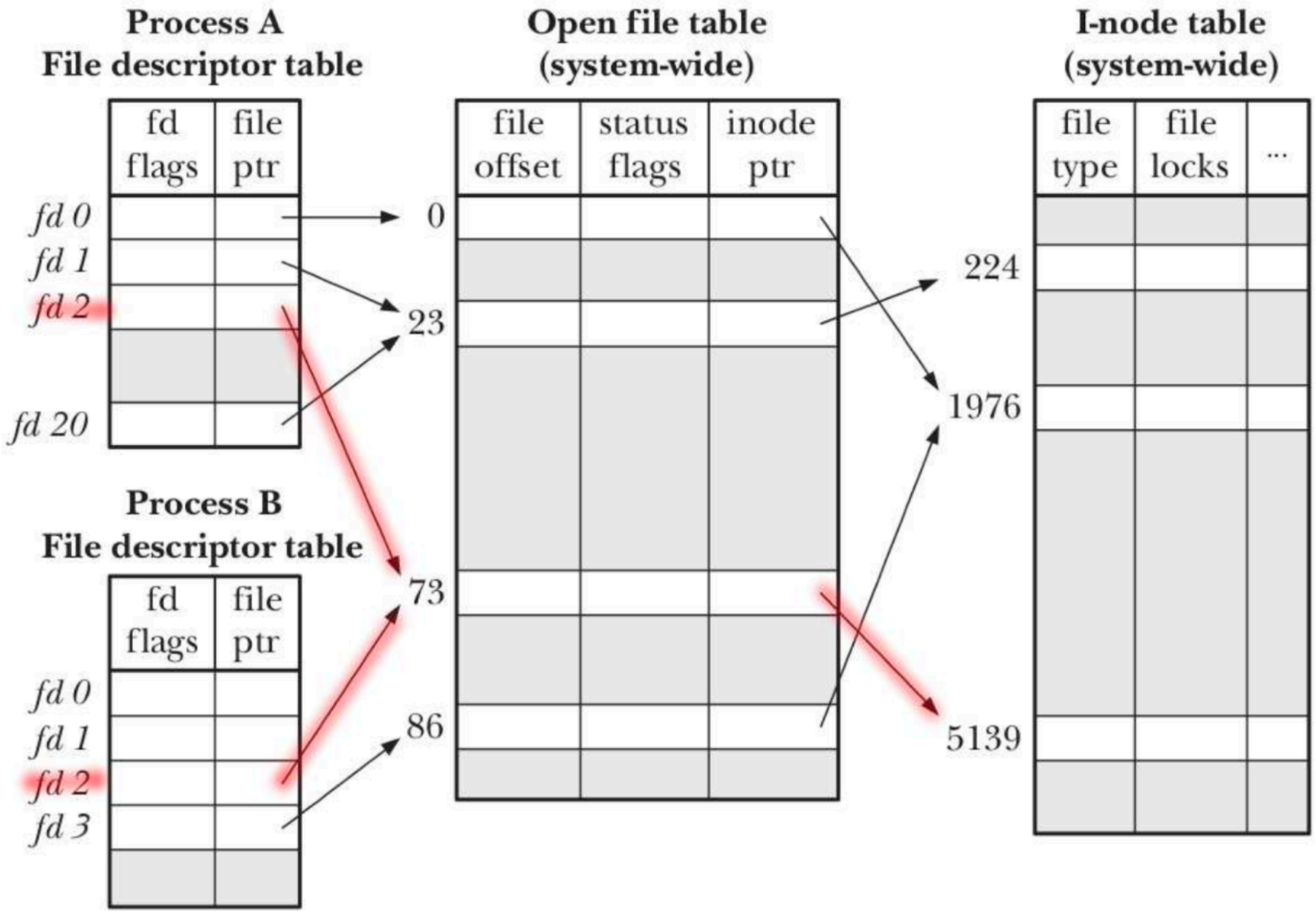
打开文件表



open ( ) the same file

不同进程的 file descriptors 指向不同的 open file table 表项

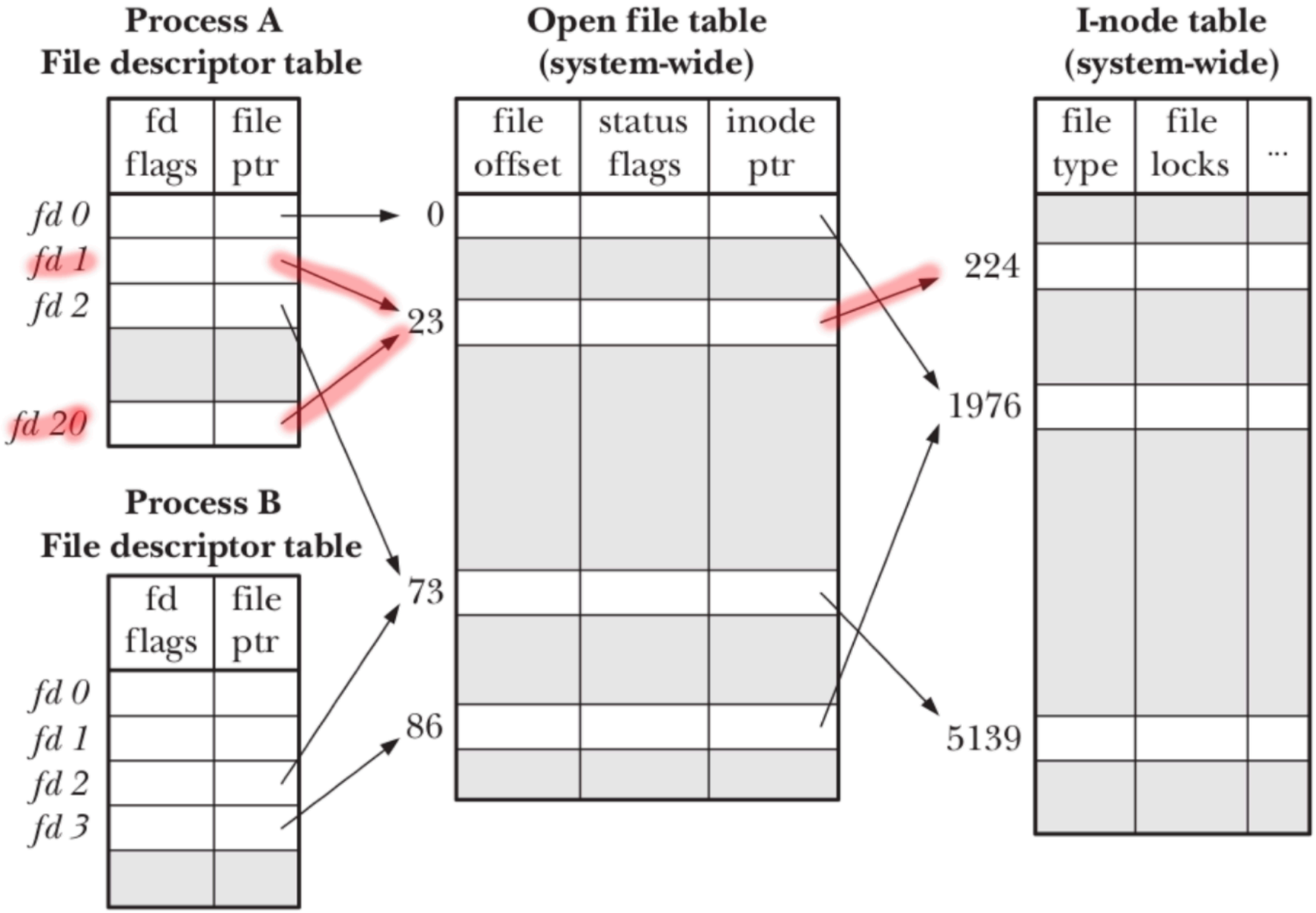
打开文件表



a result of `fork()`

不同进程的 file descriptors 指向同一个 open file table 表项

打开文件表



a result of dup ( ) / dup2 ( )

同一个进程的两个 file descriptors 指向同一个 open file table 表项

Everything is a File

An object / a file descriptor / a stream of bytes

使用同一组 APIs (with a file descriptor) 来操作不同的资源 (use common tools to operate on different things)

- `-`:普通文件 、`d`:目录文件、`b`:块设备文件、`c`:字符设备文件、`p`:命名管道文件、`l`:符号链接文件 ...
- `stdin(fd = 0)`、`stdout(fd = 1)`、`stderr(fd = 2)`
- `/proc`:内核内部数据结构的接口

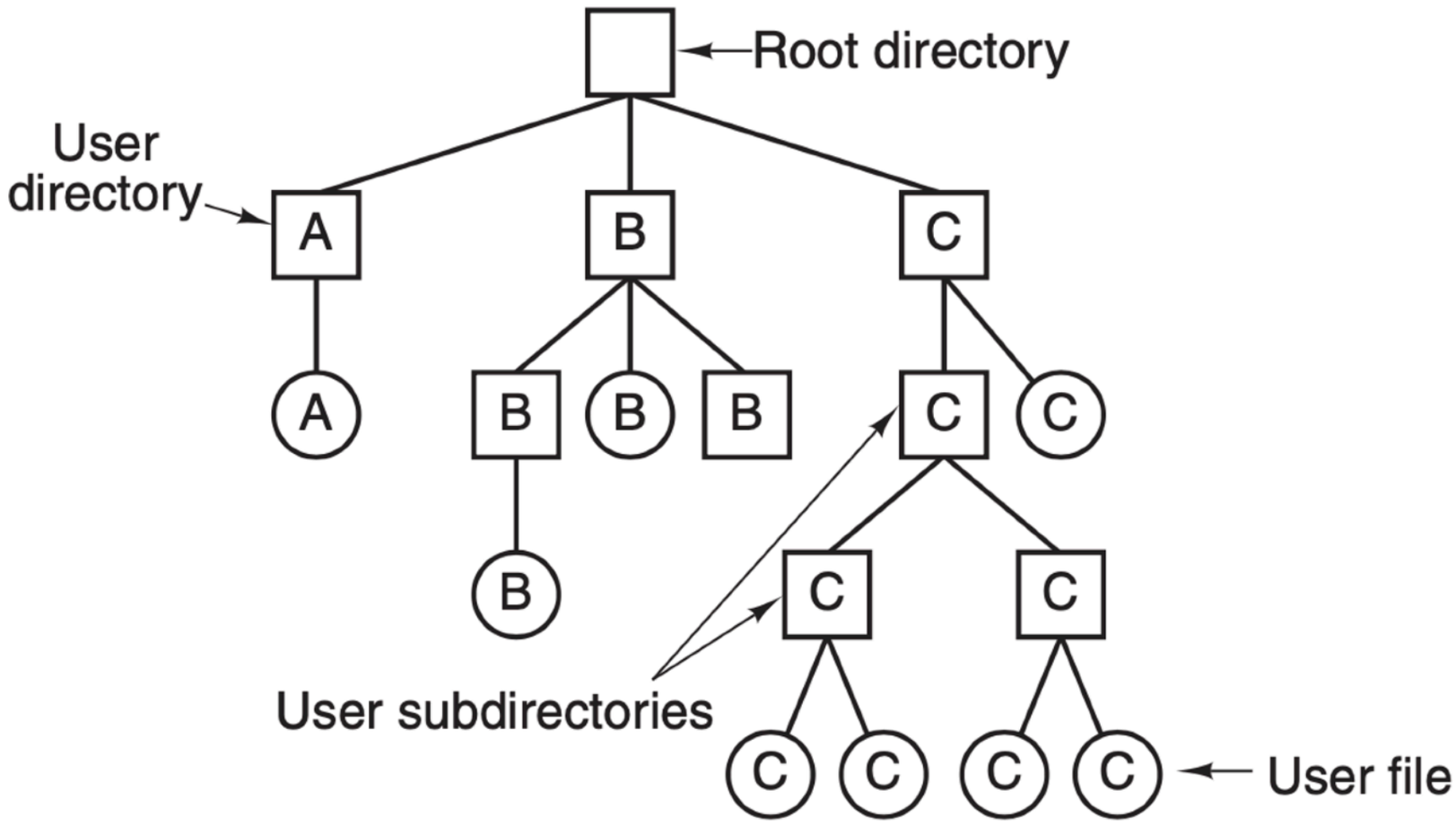
  e.g., `/proc/{PID}/maps`、`/proc/{PID}/mem`
- `/dev`:设备驱动的接口

  e.g., `/dev/random`、`/dev/null`

目录

文件系统通常使用目录 (directory) 来管理文件

- 从用户视角，可以将逻辑相关的数据放在同一个目录
- 通过在目录中创建子目录，形成一种目录树 (direcotry tree) 层次结构



目录

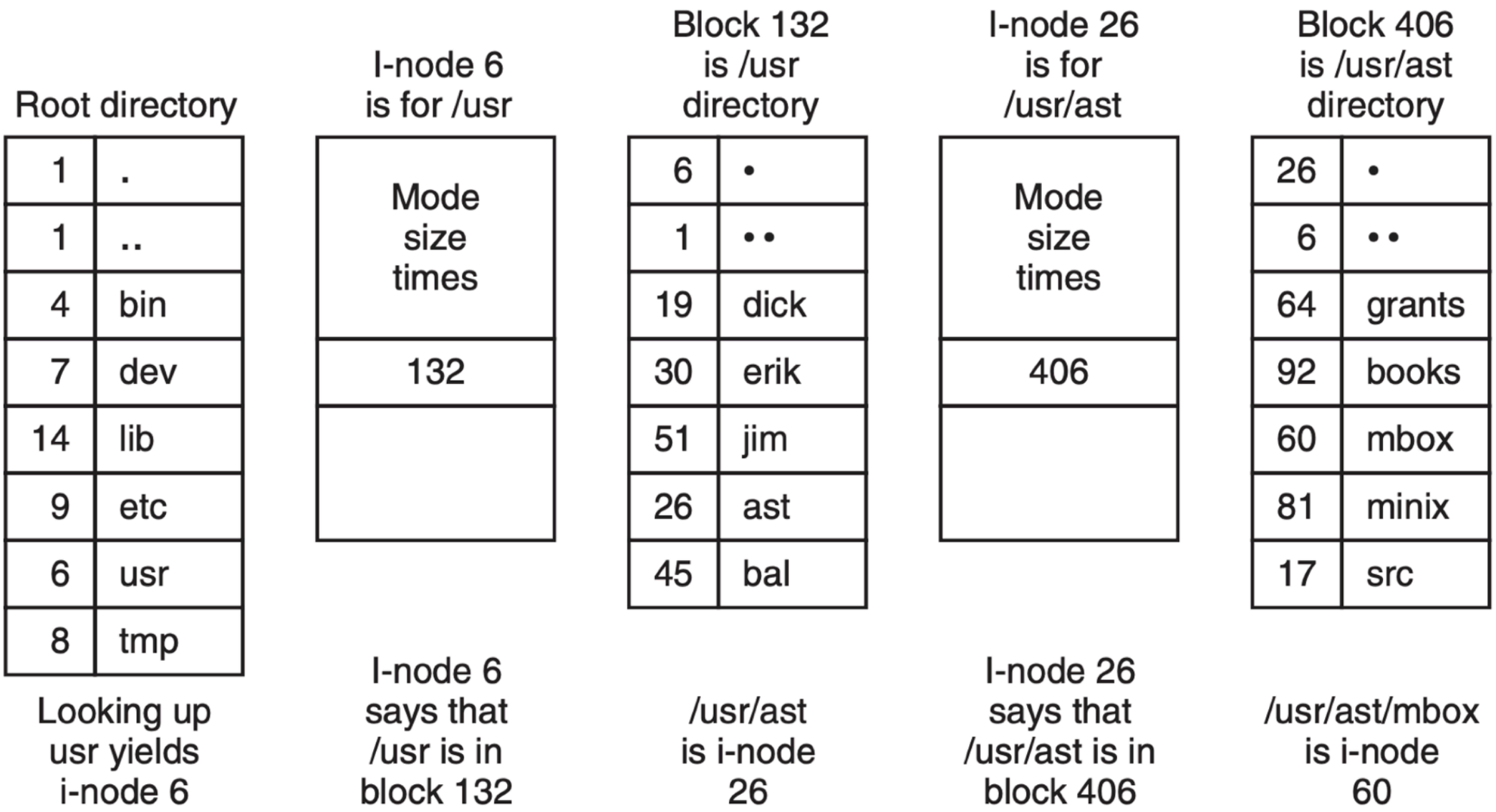
文件系统通常使用目录 (directory) 来管理文件

- 目录是一种用于存储文件名 (human friendly name) 和文件底层结构 (low-level file control block structure) 间映射关系的一种特殊文件
- 在 Unix 文件系统中，每个目录项 (directory entry) 就是一个 **⟨file name, inode number⟩ pair**
- 目录文件也是一个文件，其也有 inode
- 为了找到某文件路径名对应的 inode (resolve file name)，从根目录开始在目录文件中逐级查找
- 为根目录分配一个特殊的 inode

.	1952
..	6253
tmp	224
test.txt	1976
main.c	4594

file name　　inode number

目录



Steps in looking up `/usr/ast/mbox`

对目录的操作

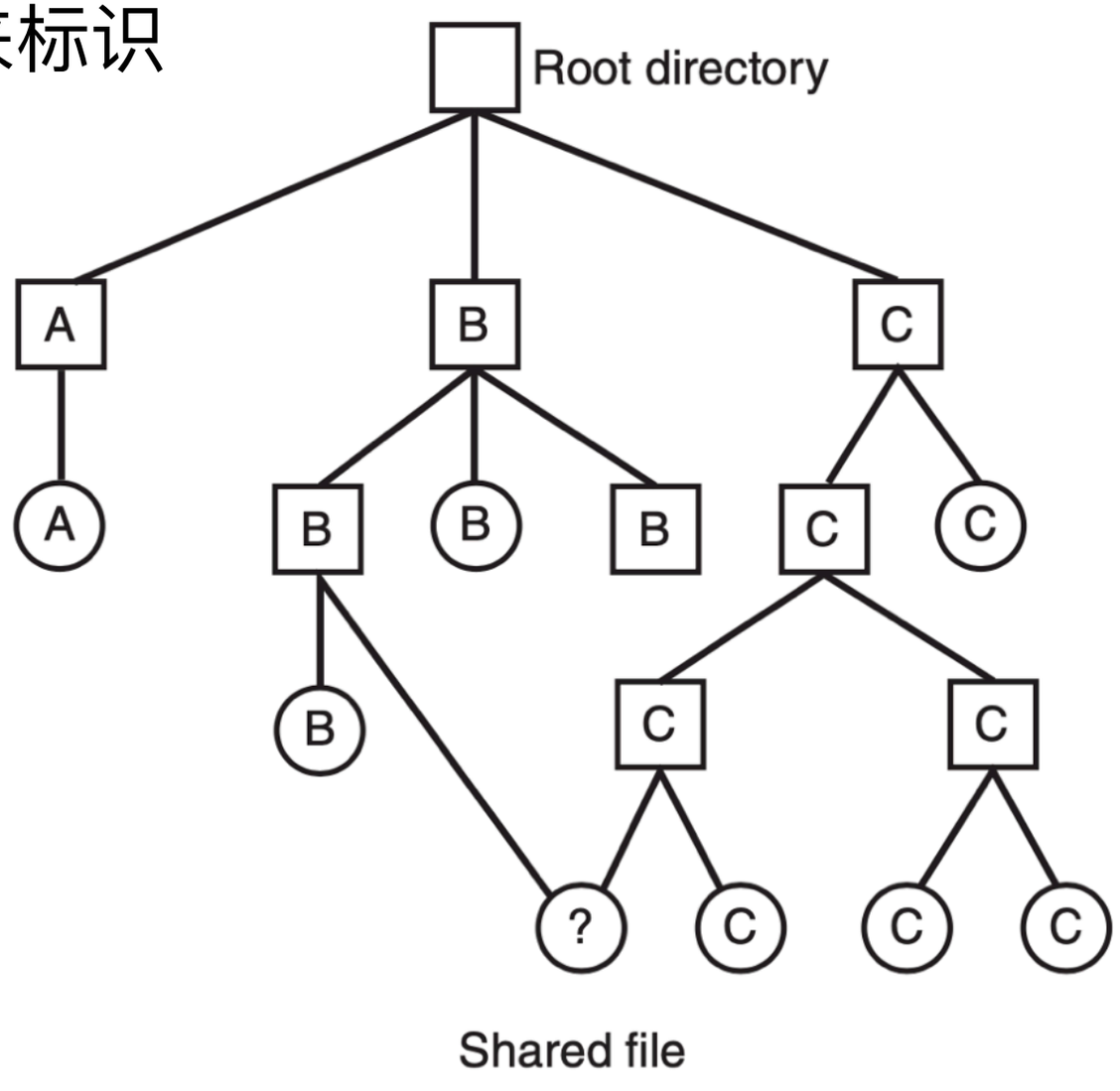
目录文件的内容可被视为文件系统的元数据 (metadata)，因而通常不允许用户直接读写目录文件

- `mkdir()`: 创建一个目录
- `rmdir()`: 删除一个目录
- `opendir()`:打开一个目录文件
- `readdir()`:返回已打开目录中的下一个目录项
- `link()`:将一个已有文件链接到一个路径名 (目录项)
- `unlink()`:删除目录中的一个目录项
- ...

文件共享

通过将同一个文件 link 到不同的路径名 (path names)，文件系统允许以多种不同名称来引用该文件

- 同样的信息可以由多个文件名来标识
- 硬链接 (Hard link)
- 符号链接 (Symbolic link)



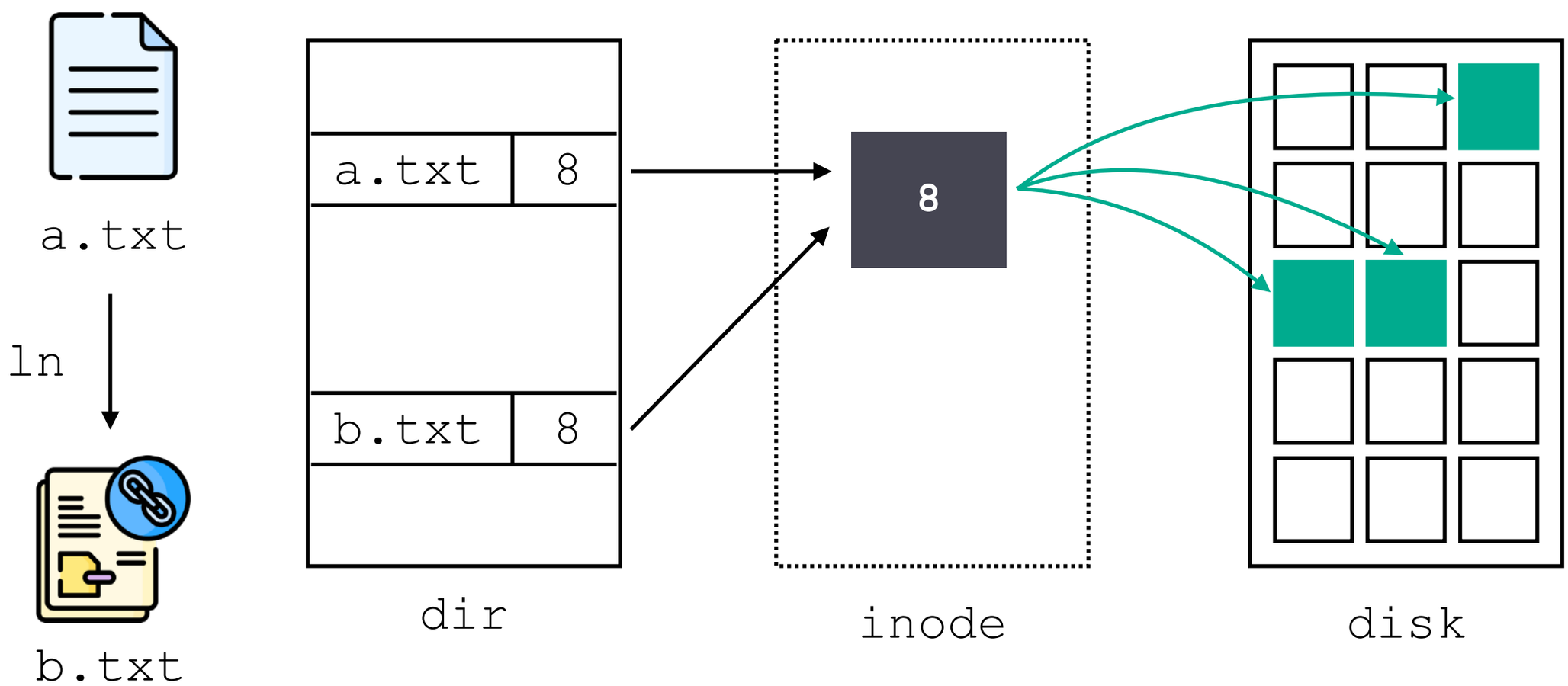
Shared file

文件共享

Hard Link

将一个新的文件名 link 到 target file 的 inode (alias for inode number)

- 两个不同的文件名指向磁盘中存储的同一个文件



文件共享

Hard Link

将一个新的文件名 link 到 target file 的 inode (alias for inode number)

- 在创建文件时，实际上就是先准备好文件的 inode，然后通过 `link()` 将其链接到某个目录中的一个 human friendly name
- 对应地，使用 `unlink()` 来删除一个文件
- 文件系统为每个 inode 记录一个 reference count (link counter)
- 在每次 `unlink()` 时对该 count 值减一
- 如果 count 值等于 `0`，则可以从磁盘中删除该文件

文件共享

Hard Link

将一个新的文件名 link 到 target file 的 inode (alias for inode number)

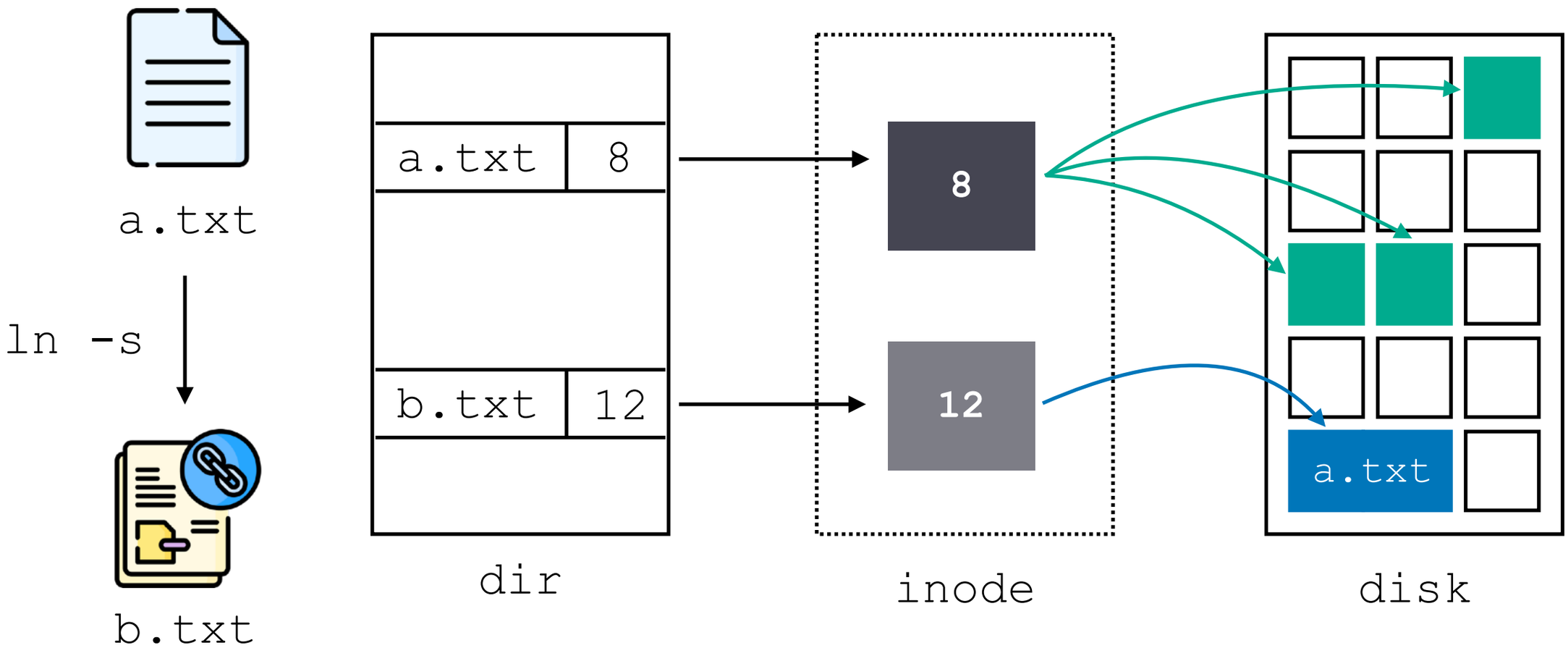
- 不能跨文件系统创建 Hard Link
  - 每个文件系统有各自的 inode 编号
- 为了简化管理，通常不允许创建目录文件的 Hard Link
  - 避免在目录层次结构中引入环 (同一个文件有无数的 path names)
  - 破坏父目录的唯一性 (如果多个父目录都有指向同一个子目录的 link，则该子目录的 . . 目录项应指向哪个目录？)

文件共享

Symbolic Link

创建一个 link type 类型的新文件 (alias for path names)

- 该符号链接文件中存储 target file 的 path name



文件共享

Symbolic Link

创建一个 link type 类型的新文件 (alias for path names)

- 能 link 到不同的文件系统、或目录文件
- 在打开文件时需要额外解析符号链接 (resolve symbolic links)
  - 读取符号链接文件中存储的 target file 的路径，使用该路径打开
    - 但 target file 可能也是一个符号链接文件
      - resolve recursively
- 当删除一个符号链接文件时，不影响 target file
- 当删除 target file 时，会造成 dangling reference
  - symbolic link 指向的文件不存在

文件保护

防止对文件的意外 (accidental) 或恶意 (malicious) 破坏行为

- 文件的所有者应该能控制
  - Access Right: 能对该文件做什么操作 (what can be done)
  - Domain: 谁能做 (by whom)
- 一些典型的访问权限
  - 与文件相关的权限: read / write / execute
  - 与目录相关的权限: list / modify / delete
  - 与 Access Right 相关的权限: 修改当前的 access right / 给某人某种 access right / 撤销某人的某种 access rights …

文件保护

防止对文件的意外 (accidental) 或恶意 (malicious) 破坏行为

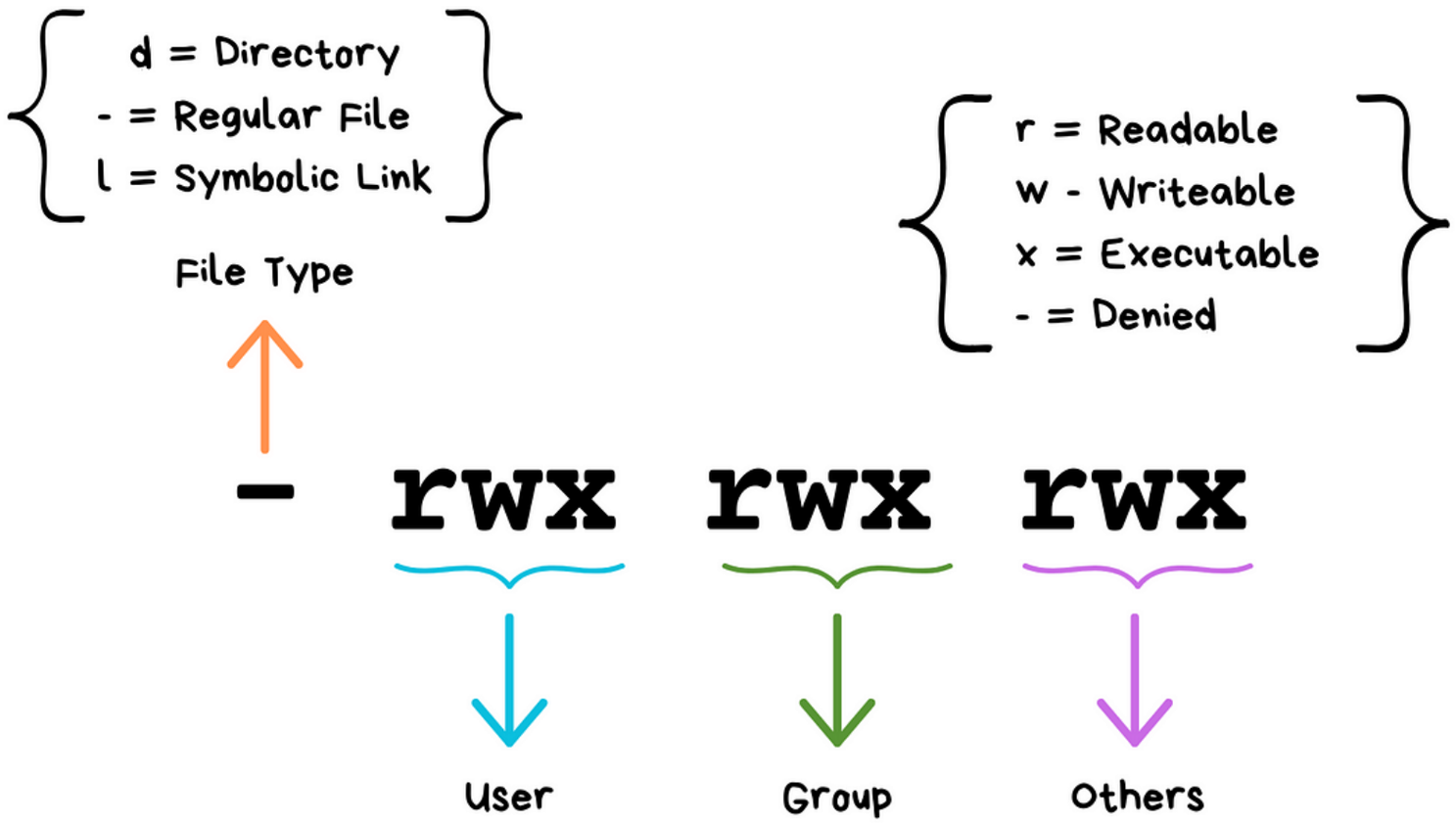
- 理论上可以利用一个巨大的 Access Control Matrix 来记录系统中每个用户对每个文件的权限

	File1	File2	File3	Dir1	Dir2	...
UserA	rw	r	rwx	lmd	l	...
GroupB		r	rw		lm	...
...	...	...	...	...	...	...

文件保护

Unix 文件系统中的访问控制

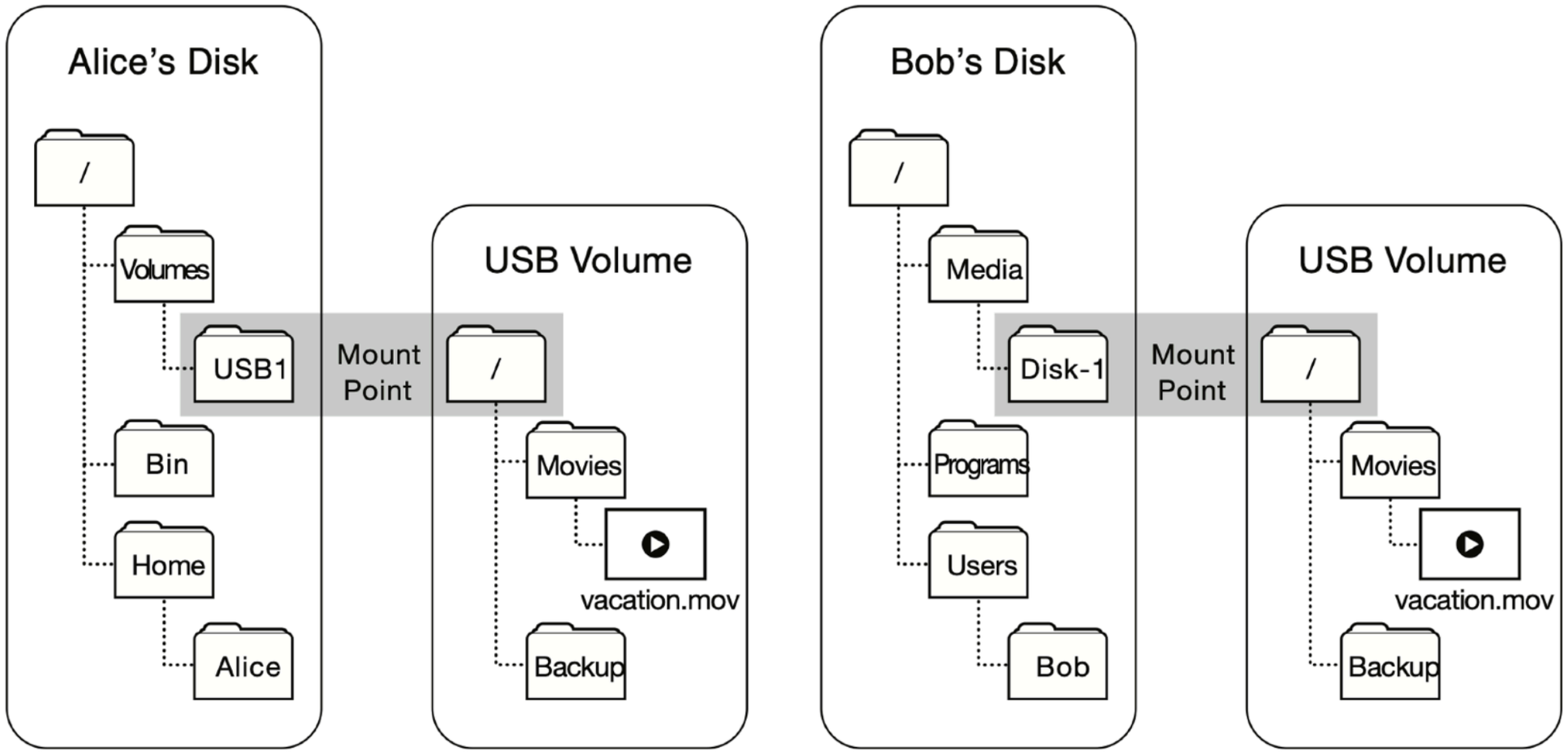
- 为每个用户分配 `UserID` 和 `GroupID`
- 为每个文件记录不同类型用户 (domain) 的访问权限 (access rights)



文件系统挂载

`mount()`：一个文件系统必须先挂载才能进行访问

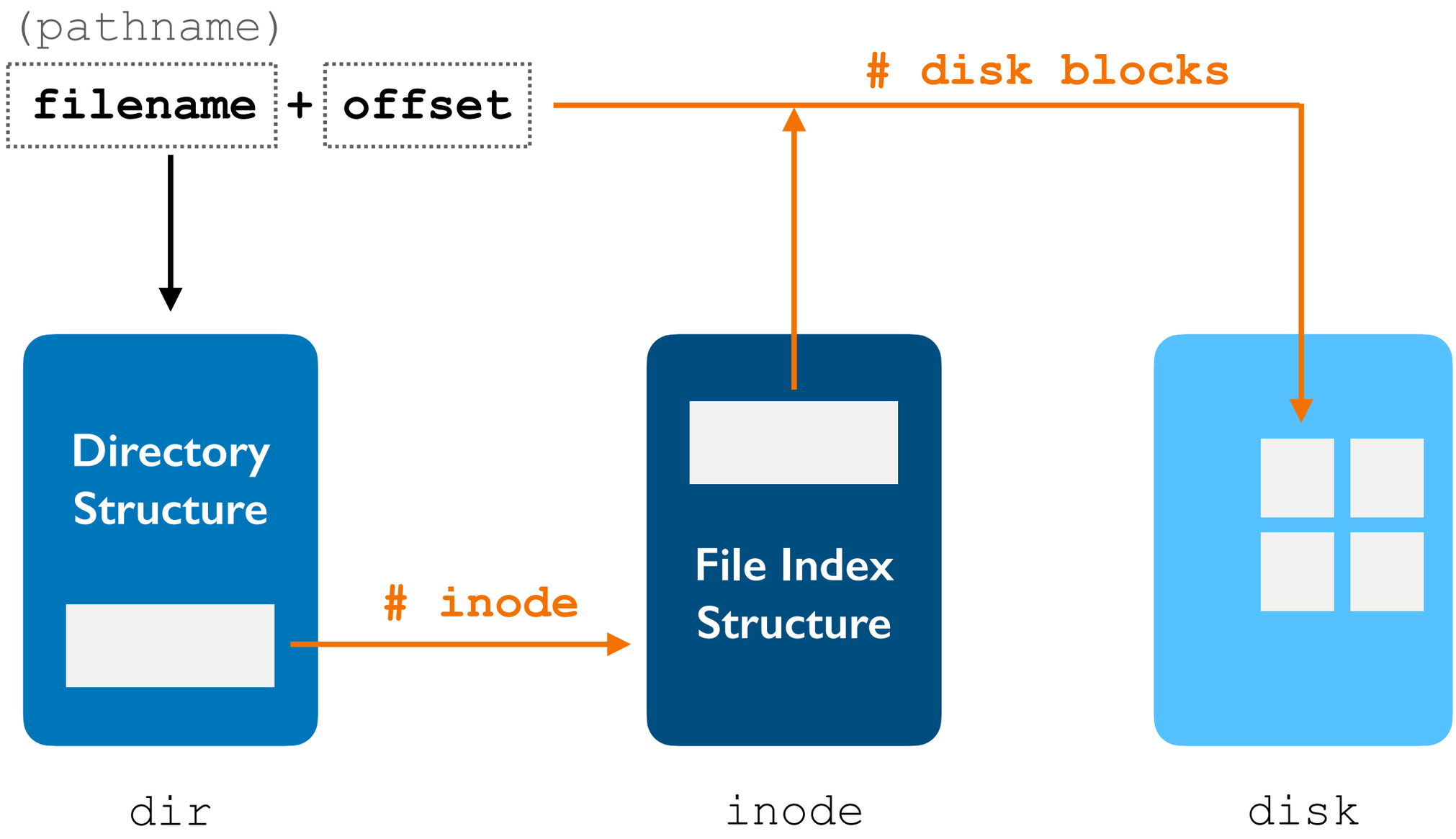
- 形成一个统一的目录树结构



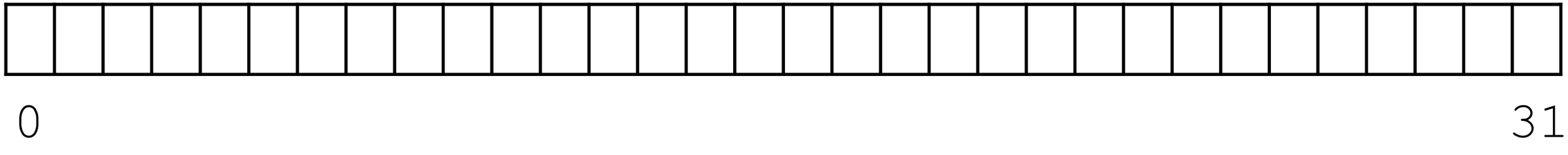
File System Implementation

Two Key Abstractions

File and Directory (Unix File System)



File System Layout

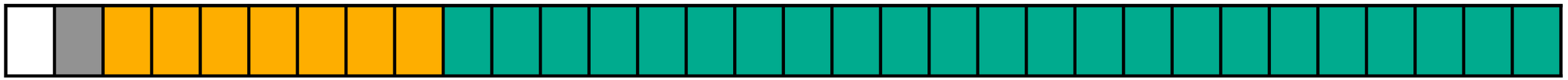


将整个磁盘划分为若干固定大小的 blocks

File System Layout

Data Region (Data Blocks)

分配一定数量的 blocks 用于存储文件数据



0

31

File Attributes and Allocation Structure

分配一定数量的 blocks 用于存储文件的 metadata 、以及空闲空间管理相关的信息

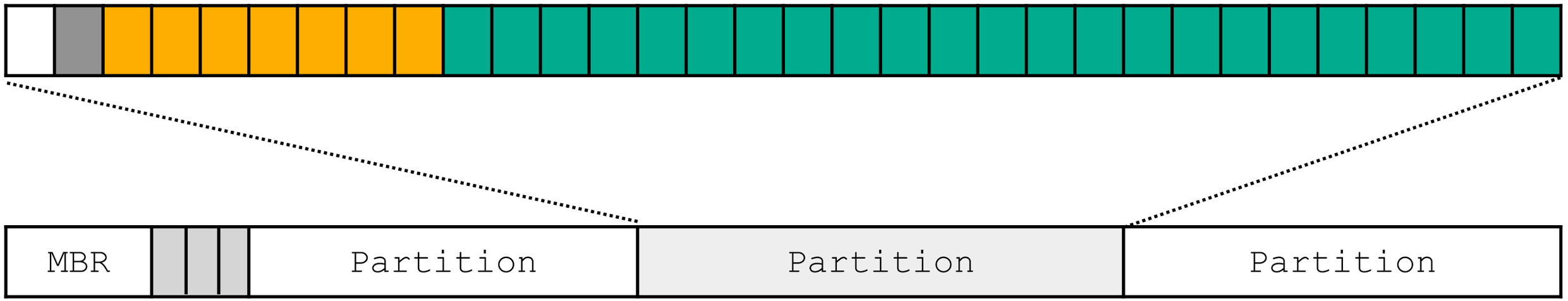
Super Block

存储文件系统的 metadata 信息 (first block to read when mounting)

Boot Block

存储 Boot OS 相关的信息 (empty if no OS)

File System Layout



Disk Partitions

不同 partition 可格式化为不同文件系统

Partition Table

存储每个 partition 的属性信息 (one is marked as active)

Master Boot Record (MBR)

存储 Boot Computer 相关的信息

文件的实现方式

文件的第 *N* 个 bytes (第 *N* 个 logical block) 存储在哪个 disk block？

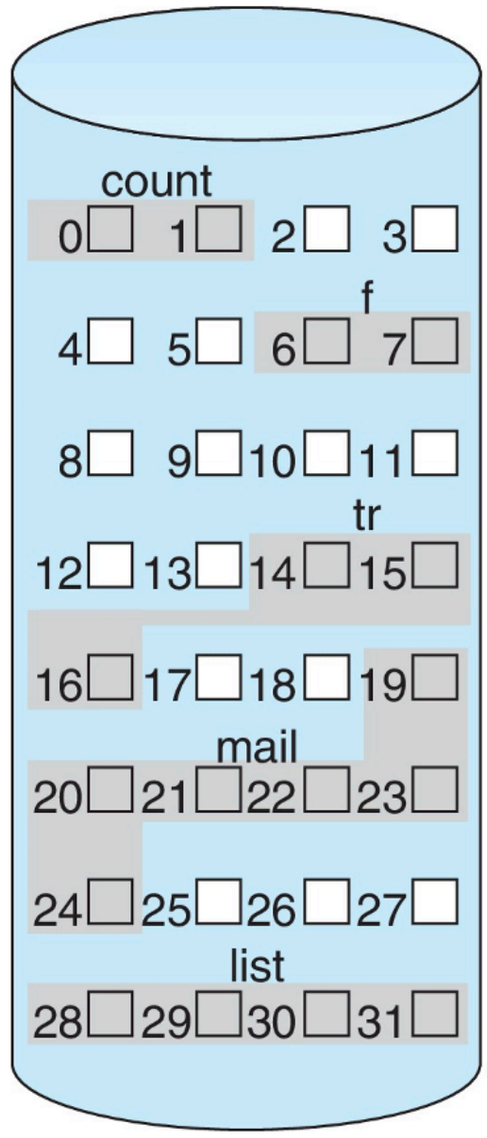
- 文件数据存储在磁盘的某些 data blocks 中
- 需要设计特定的数据结构来记录每个文件 data blocks 的分配信息
  - 连续方式 (Contiguous Allocation)
  - 链表方式 (Linked List Allocation)
    - FAT 文件系统的 File Allocation Table 结构
  - 索引方式 (Index Allocation)
    - Unix 文件系统的 inode 索引结构

文件的实现方式

Contiguous Allocation

每个文件实现为磁盘上的若干连续 data blocks

- 仅需在文件属性中记录 first block + total number of blocks 信息
- 有较好的顺序读写性能、且能快速定位随机数据块
- 但显然不够灵活
  - 在创建文件时需确定文件大小
  - 还会存在严重的外部碎片问题
  - 在特定场景下仍是很好的选择 (e.g., CD-ROM)



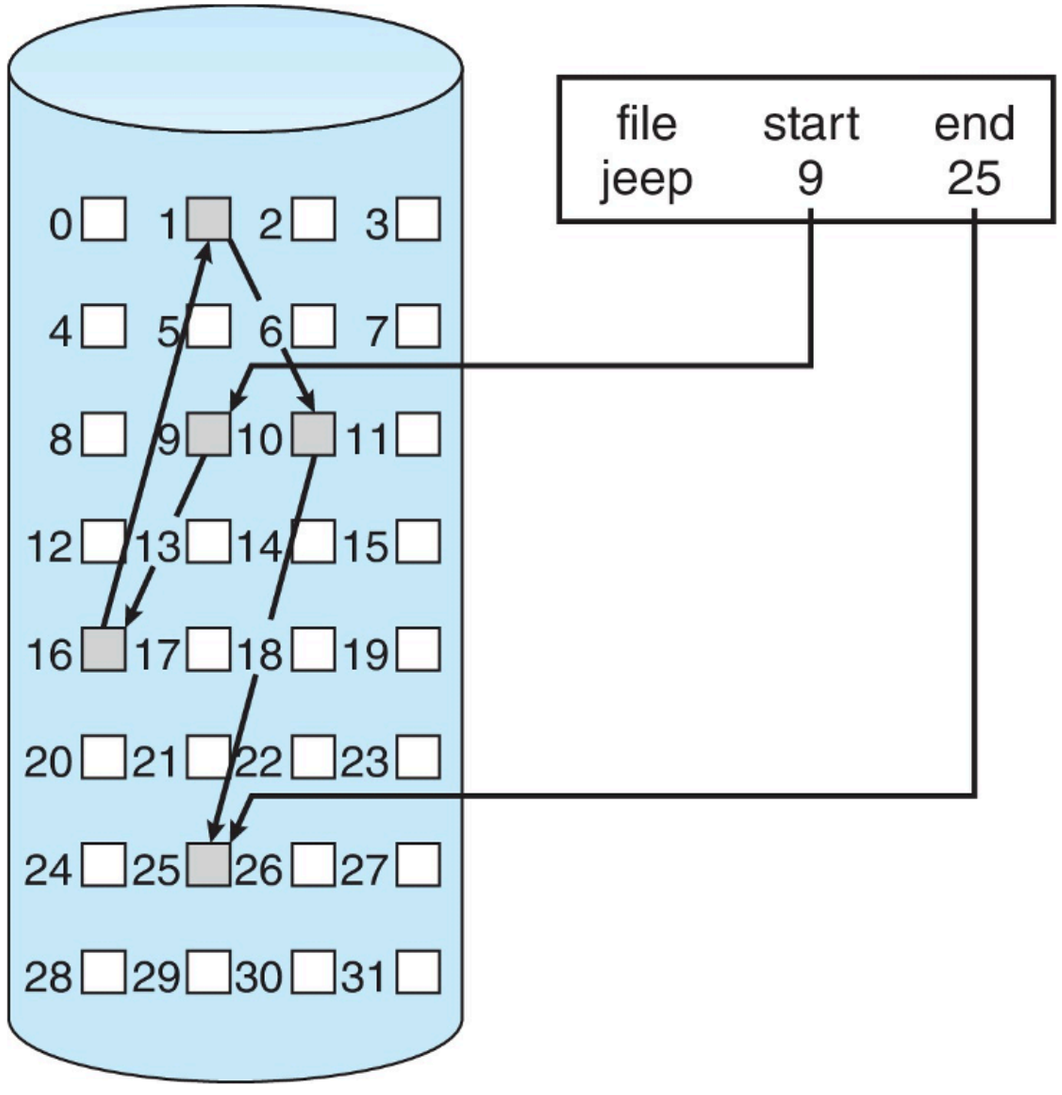
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

文件的实现方式

Linked List Allocation

每个文件实现为 data blocks 的一个链表

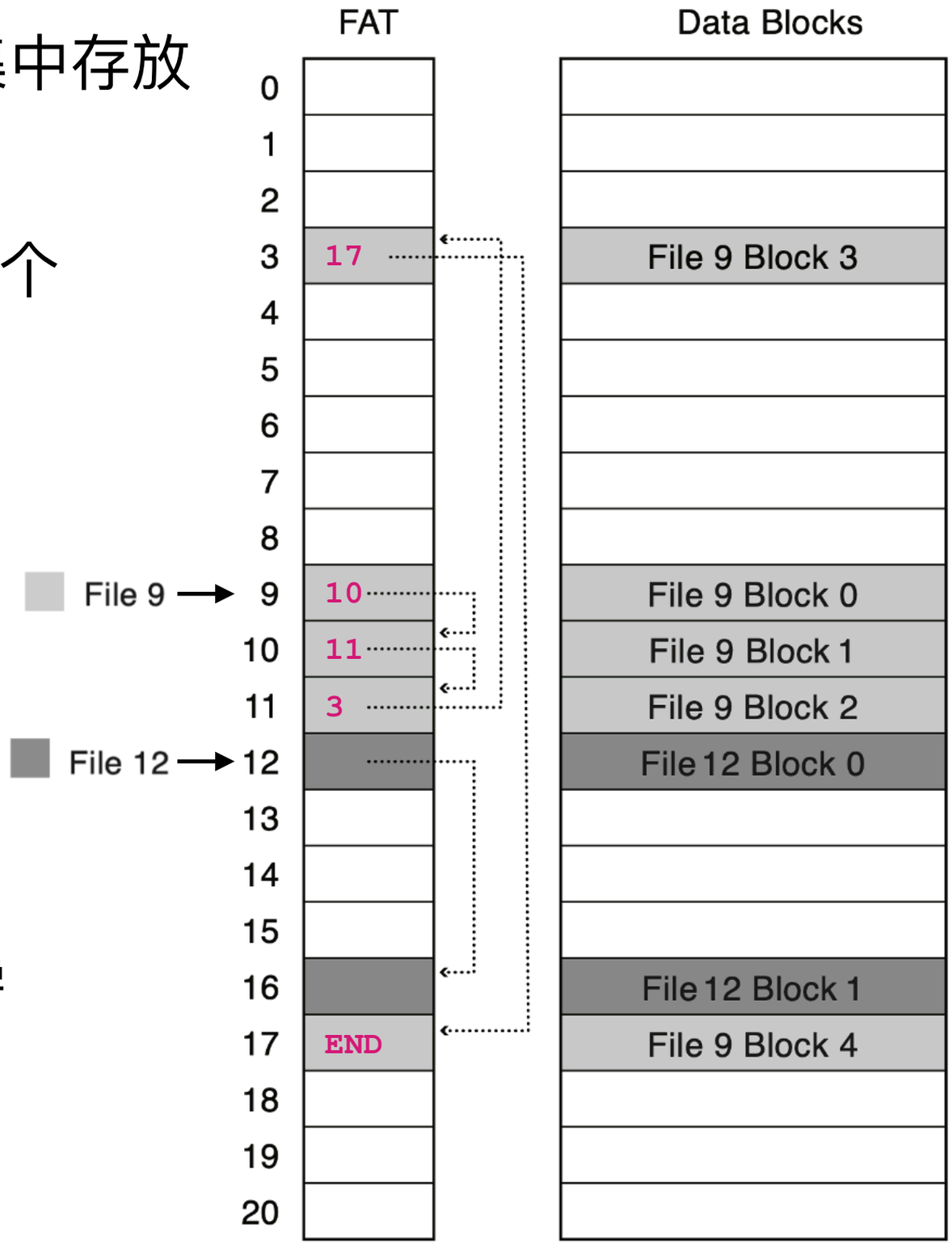
- 仅需在文件属性中记录 first block number 信息
- 可以把所有 data blocks 都利用起来
  (没有外部碎片)
- 比较适合小文件，但是
  - 定位一个大文件的随机数据块
    需要遍历链表
  - 每个数据块中需要存储一个指针
    (数据块可用大小不再是 2^k)
  - 指针损坏导致文件丢失



File Allocation Table

将 data blocks 链表中的所有指针集中存放到一个 File Allocation Table (FAT) 中

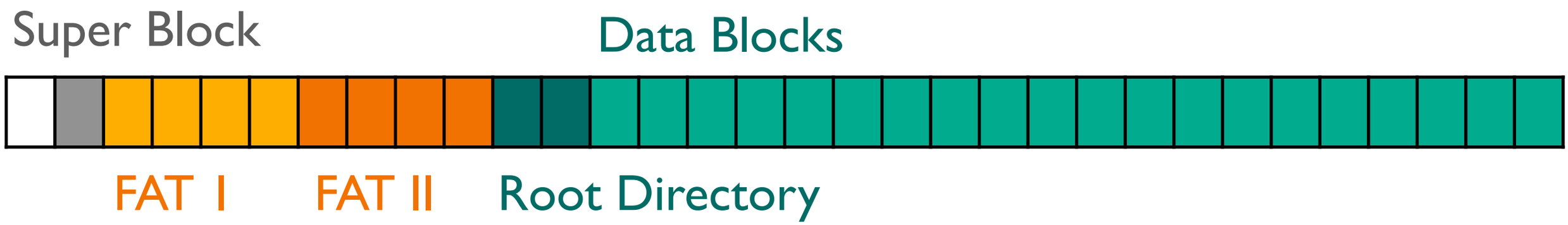
- 每个 data block 对应 FAT 中的一个表项，其存储
  - 文件下一个 data block 的编号 (pointer to the next FAT entry of the file)
  - 文件终止标记 (end of file)
  - 空闲空间标记 (free space)
- 此时定位文件的随机数据块只需访问 FAT 结构 (可缓存在内存)



FAT File System

FAT 文件系统的磁盘布局

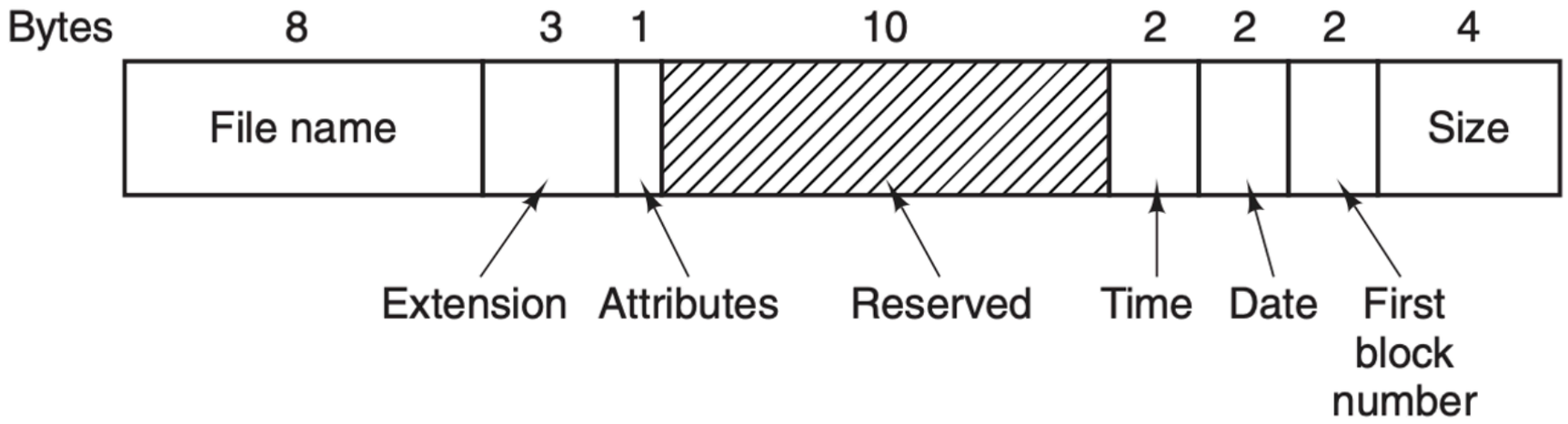
- FAT 结构和根目录存储在磁盘的特定 blocks 中
- 存储两个 FAT 来增加可靠性 (redundancy)
- FAT 中同时也记录了空闲空间信息



FAT File System

FAT 文件系统的目录项 (directory entry) 结构

- 文件的属性信息存储在目录项中 (32 bytes)



FAT File System

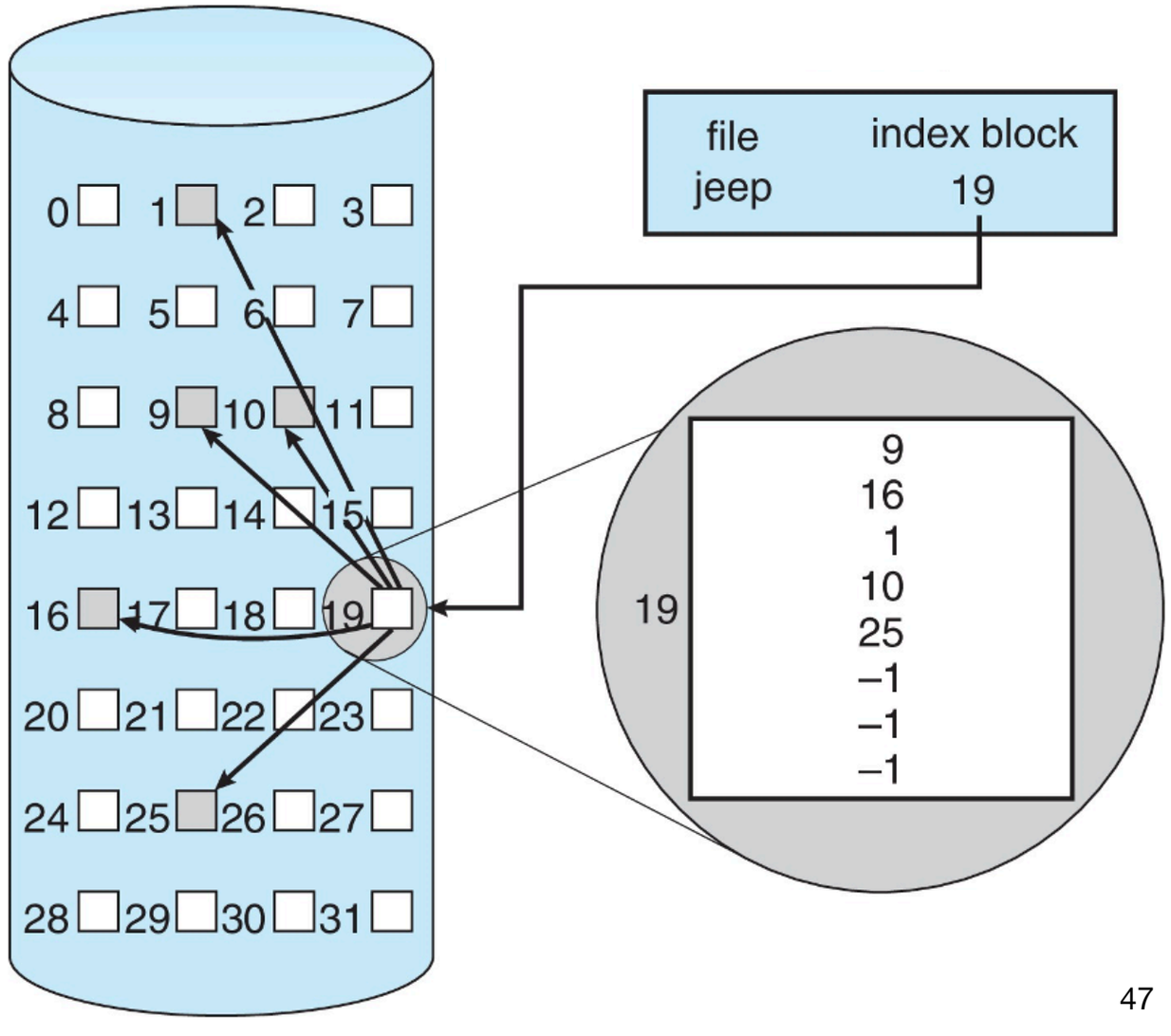
- FAT 是一个设计和实现较为简单的文件系统，广泛应用于许多便携和嵌入式设备 (including USB flash drives)
- 但在内存中缓存整个 FAT 结构会导致较高的存储开销
  - 对于 1TB (2^{40}) 大小的磁盘和 4KB (2^{12}) 大小的 Disk Block
  - 总共需要 2^{28} 个 FAT 表项
  - 如果每个表项占用 4 bytes (FAT 32)，则整个 FAT 表大小为 1GB

文件的实现方式

Indexed Allocation

每个文件实现为一个记录 data blocks 编号的数组 (the i-th pointer refers to the i-th data block of the file)

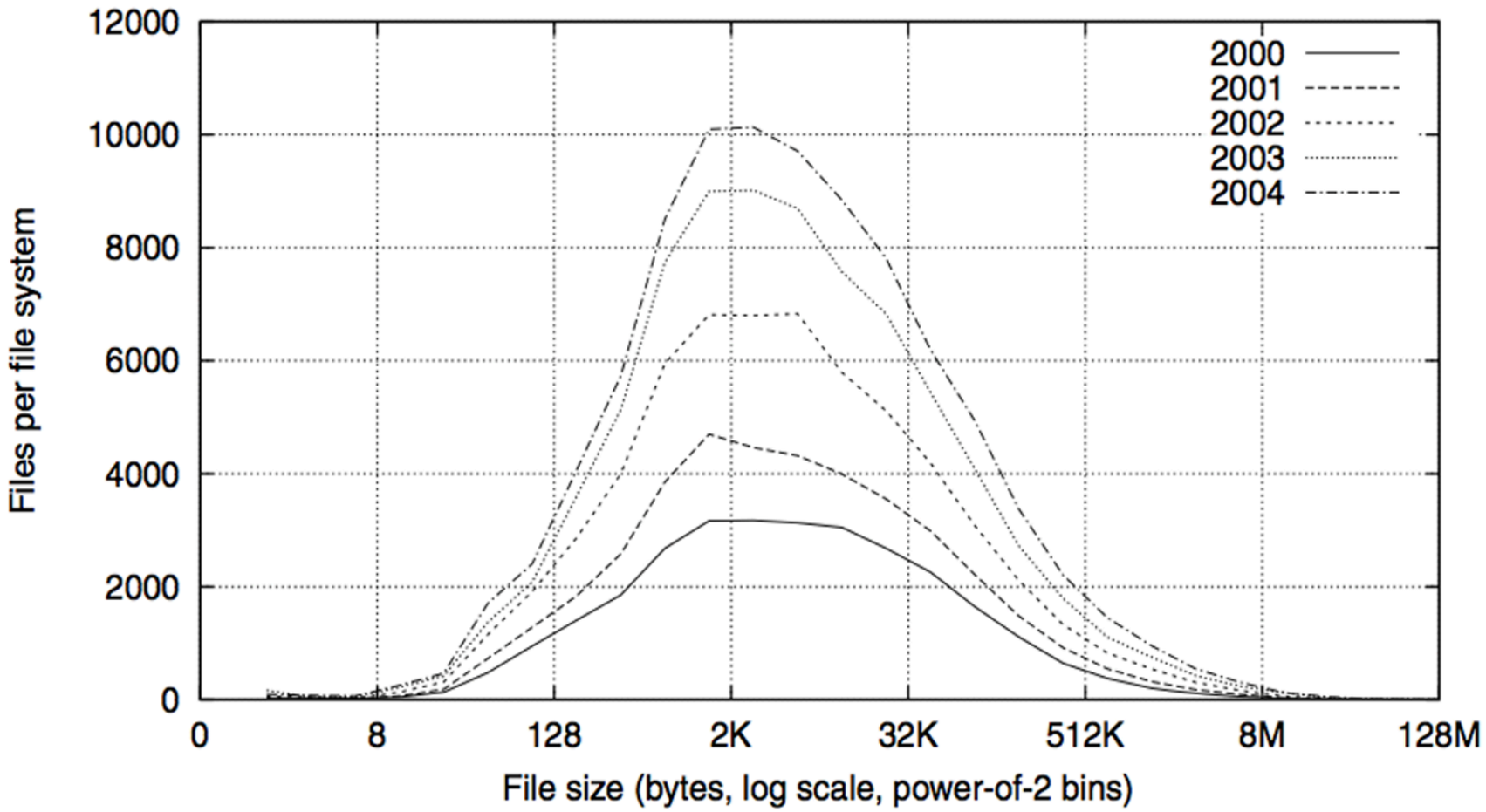
- 仅需在文件属性中记录存储 data block 编号的 block number
- 通过在内存中缓存索引结构来提高随机访问速度
  - 需要缓存的内容只和当前已打开文件数相关 (而不是整个磁盘的大小)
- 但是，如何决定所需索引结构的大小 (number of pointers required)？



文件的实现方式

Characteristics of Real-World File Systems

- Most files are small: ~2K is the most common size



Agrawal N, Bolosky W J, Douceur J R, et al. A five-year study of file-system metadata. ACM Transactions on Storage (TOS), 2007, 3(3): 9-es.

文件的实现方式

Characteristics of Real-World File Systems

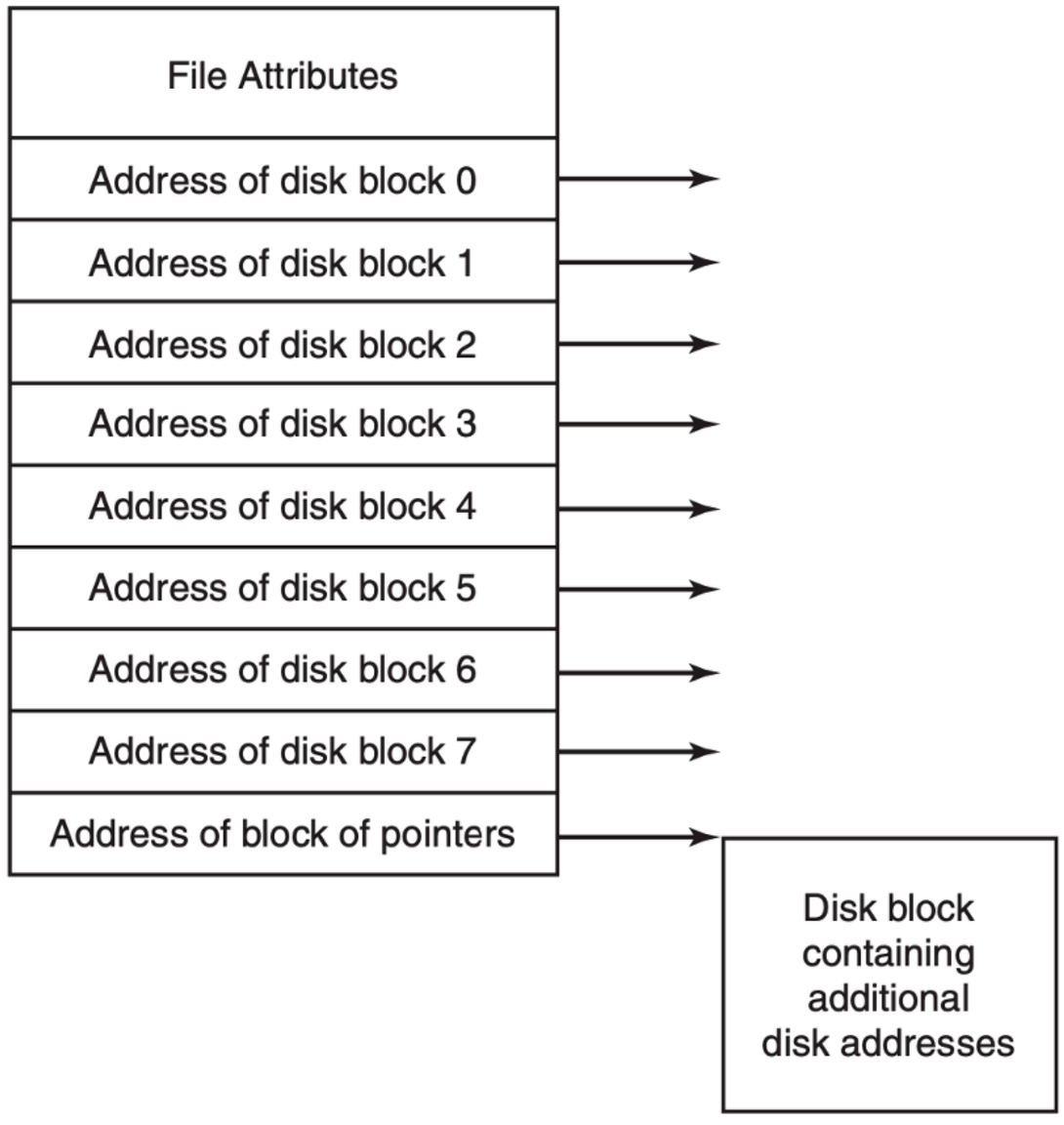
- Most files are small: ~2K is the most common size
- Average file size is growing: ~200K is the average
- Most bytes are in large files: a few big files use most of space
- File systems contains lots of files: almost 100K on average
- File systems are roughly half full: even as disks grow, file systems remain about 50% full
- Directories are typically small: many have few entries; most have 20 or fewer

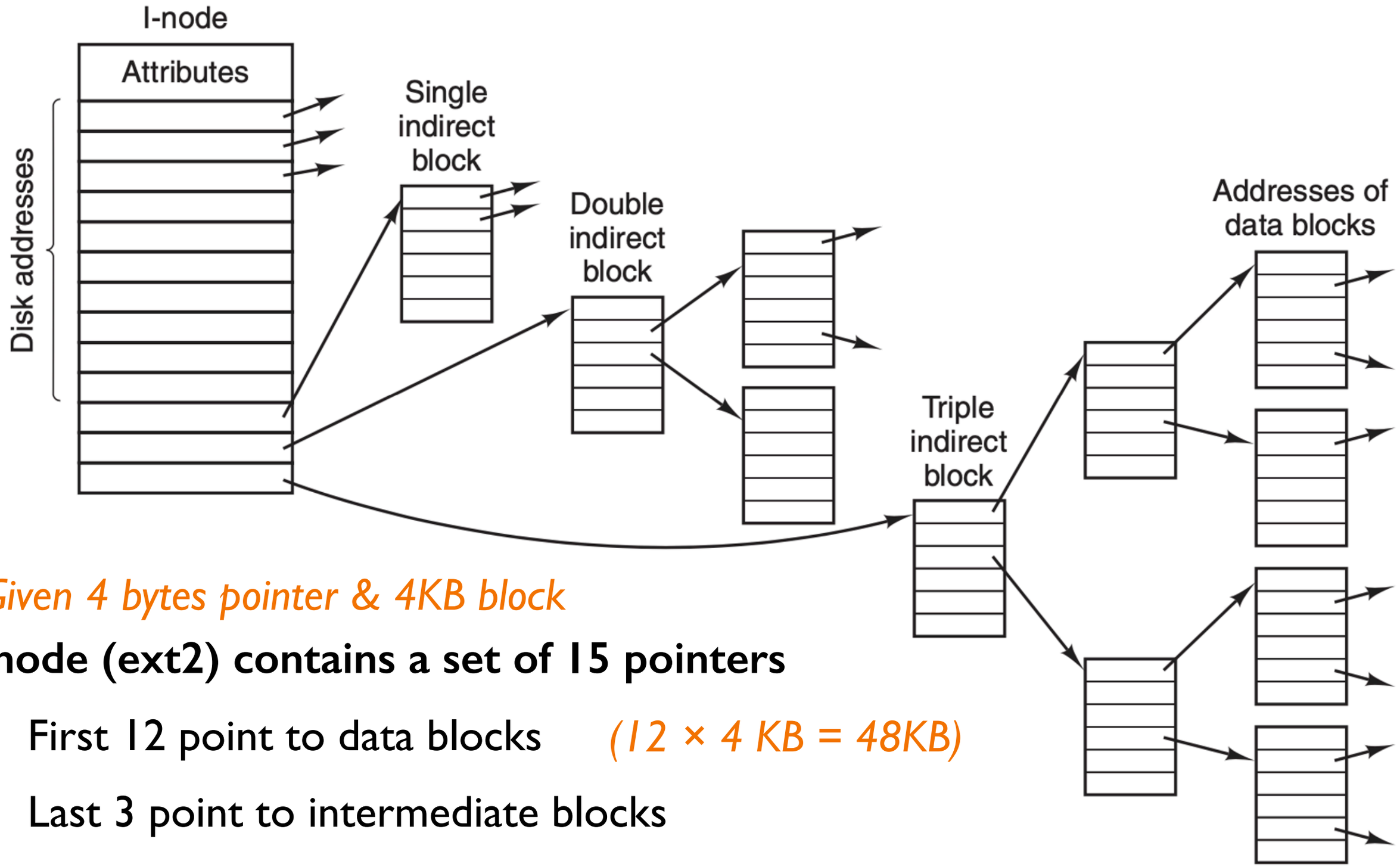
Agrawal N, Bolosky W J, Douceur J R, et al. A five-year study of file-system metadata. ACM Transactions on Storage (TOS), 2007, 3(3): 9-es.

Unix File System

对小文件 (fast path) 和大文件 (slow path) 都提供良好支持

- 在 inode 中维护一组固定数目的 pointers
  - Direct pointers: 直接指向包含文件数据的 data blocks
  - Indirect pointers: 指向一个包含 direct pointers 的 data block
- 支持快速随机访问
  - 小文件直接使用 direct pointers (array structure)
  - 大文件额外使用 indirect pointers (tree structure)





Given 4 bytes pointer & 4KB block

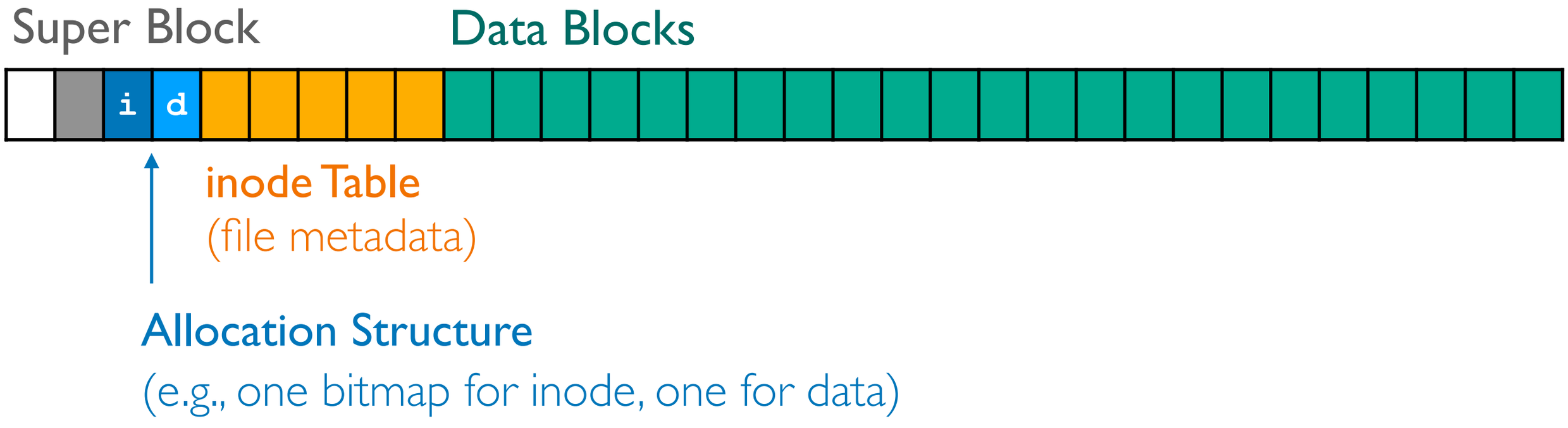
inode (ext2) contains a set of 15 pointers

- First 12 point to data blocks *(12 × 4 KB = 48KB)*
- Last 3 point to intermediate blocks
  - #13: single indirect pointer *(2^{10} × 4 KB = 4MB)*
  - #14: double indirect pointer *(2^{10} × 2^{10} × 4 KB = 4GB)*
  - #15: triple indirect pointer *(2^{10} × 2^{10} × 2^{10} × 4 KB = 4TB)*

Unix File System

Unix 文件系统的磁盘布局

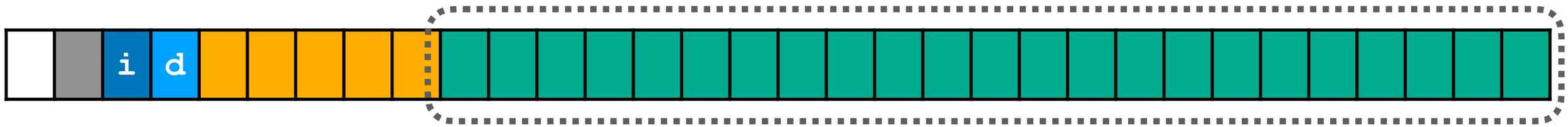
- 分配若干 disk blocks 用于存储 inode (每个 block 可存储多个 indoes)
  - 根据 inode number 可以容易定位该 inode 在哪个 disk block
- 分配额外的数据结构 (e.g., bitmap) 进行空闲空间管理
  - 当前哪些 inodes 以及哪些 data blocks 空闲



Unix File System

文件的大小 (size) 和文件占用的磁盘空间 (size on disk)

- Size (`ls -lh`) = 文件字符序列的长度
- Size on disk (`du -h`) = 文件占用了多少 data blocks
  - data block 是文件存储空间分配的基本单位
  - 对于大文件，还需要分配 data block 来存储 indirect pointers

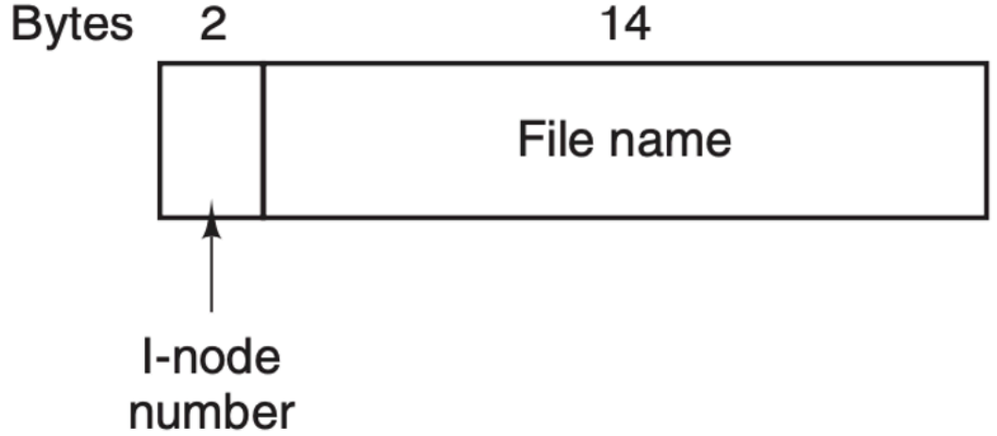


disk space (# data blocks)
that a file occupies

Unix File System

FAT 文件系统的目录项 (directory entry) 结构

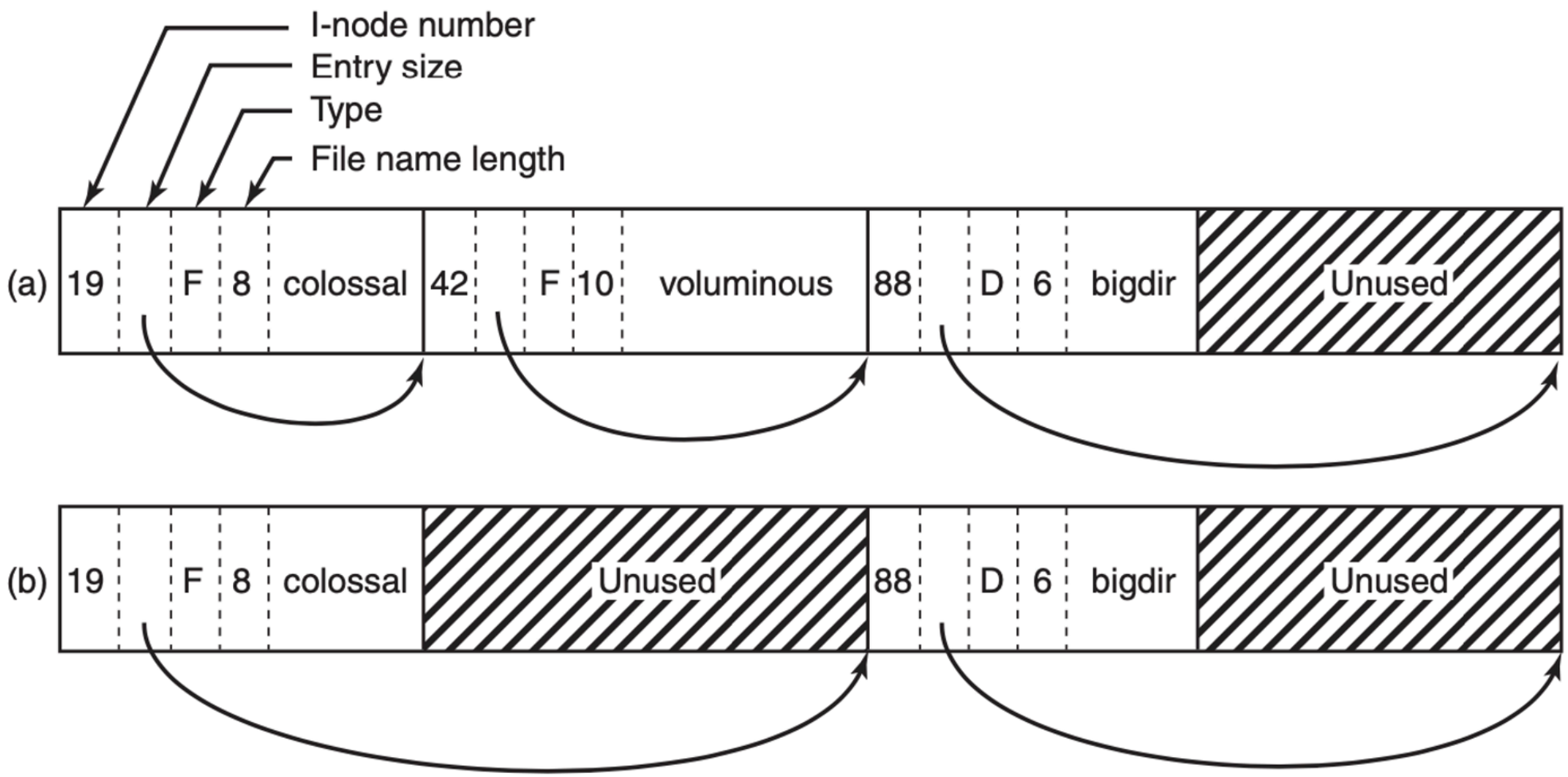
- 在最简单的形式下，每个目录项仅需包含文件名和 inode number
- 文件的属性信息存储在 inode 结构中



Unix File System

FAT 文件系统的目录项 (directory entry) 结构

- 可在目录项中额外记录文件名长度来支持不定长文件名



the ext2 example

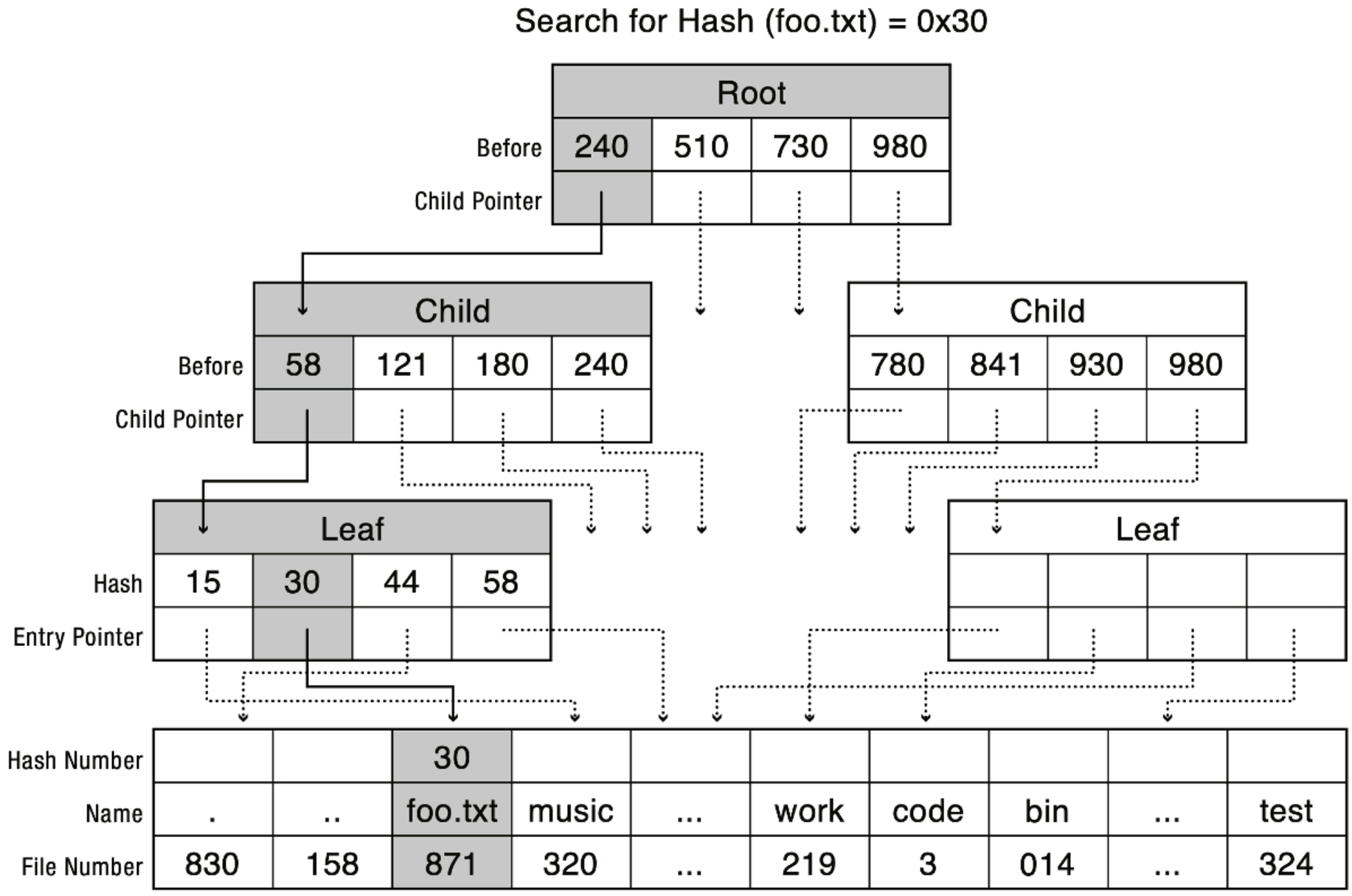
Unix File System

如何在目录文件中查找文件名对应的 inode number (resolve file name) ?

- 对于线性的 ⟨file name, inode number⟩ 目录项存储方式
  - 需要遍历数组
  - 如果每个目录中文件数量不是很多，已经足够高效
- 现代文件系统中会设计更加复杂的数据结构来加速查找

Unix File System

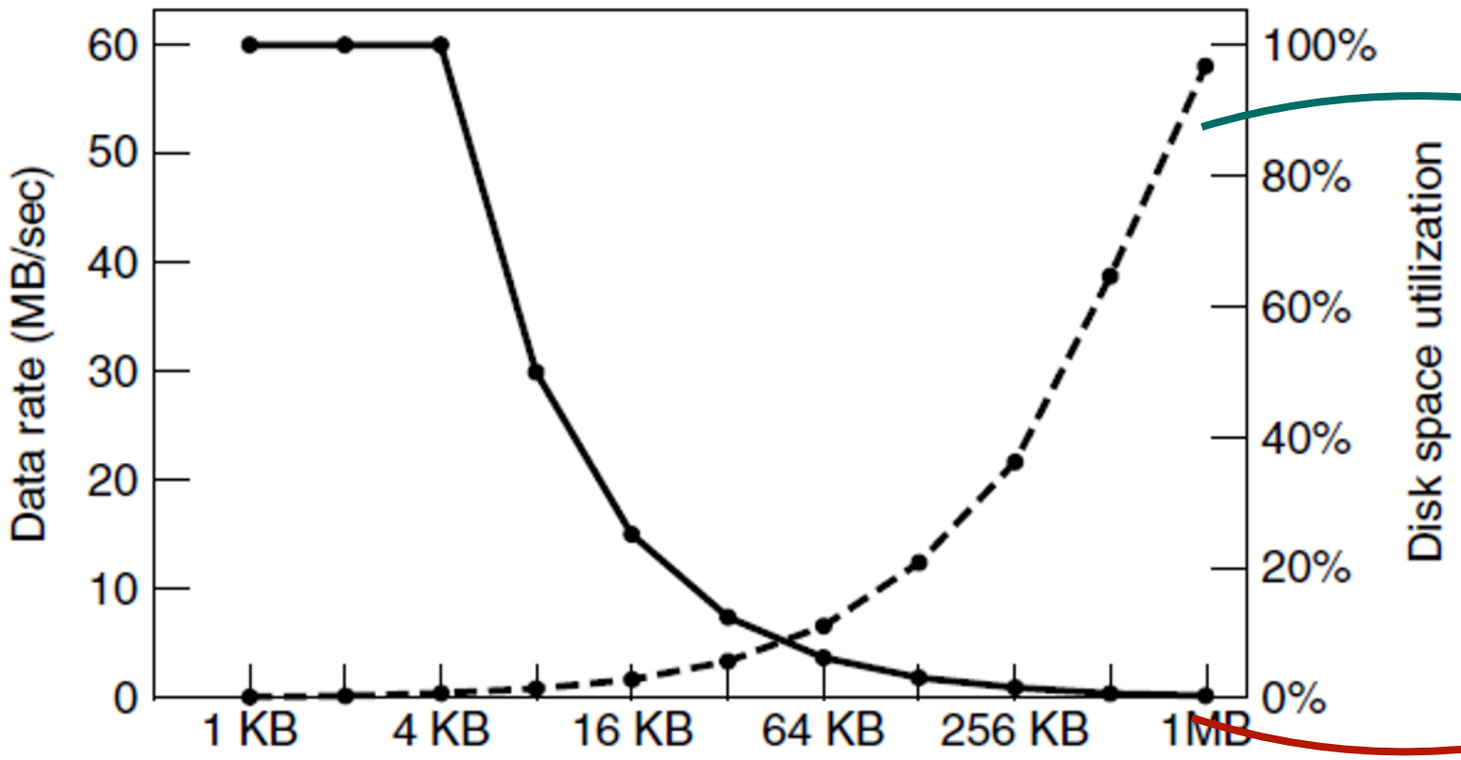
- 使用 B+ tree 来存储 ⟨file name, inode number⟩， 其中用文件名的 hash value 作为索引



磁盘块大小

文件系统可按不同的磁盘块大小 (block size) 来进行格式化

- 设置太大会浪费磁盘空间 (存储小文件时的内部碎片)
- 设置太小会导致较高的磁盘读写开销 (随机读写多个磁盘块)



To store a 4KB file

Data rate goes up almost linearly with block size (the more data fetched, the better)

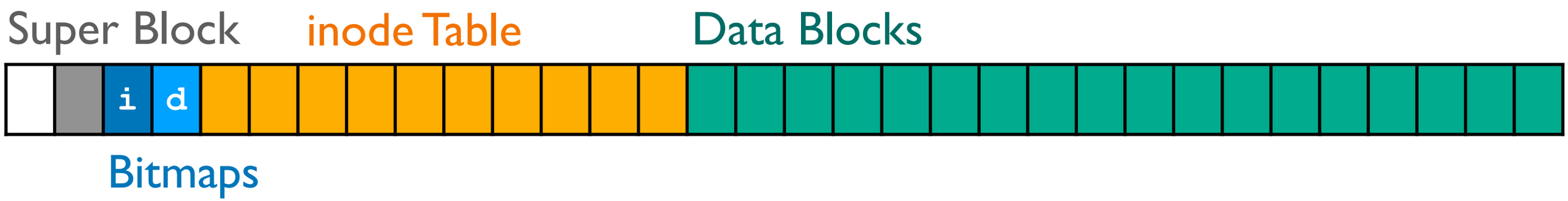
More spaces are wasted for larger blocks

Quiz

如下 Unix 文件系统所能支持的最大文件大小 (maximum file size) 是多少？单个目录中最多能包含多少文件 (maximum number of files in a single directory)?

- 磁盘 block 大小为 1 KB，其中划分 4096 个 blocks 用于存储 inodes
- inode 结构中存储 user ID (2 bytes)、时间戳 (12 bytes)、文件类型和保护信息 (4 bytes)、link count (2 bytes)、文件大小 (4 bytes)、以及文件 data blocks 索引结构
  - 索引结构中包含 8 个 direct pointer、1 个 indirect pointer、以及 1 个 double indirect pointer (4 bytes for each pointer)
- 每个目录项 (directory entry) 存储文件名 (14 bytes) 和 inode number

文件操作



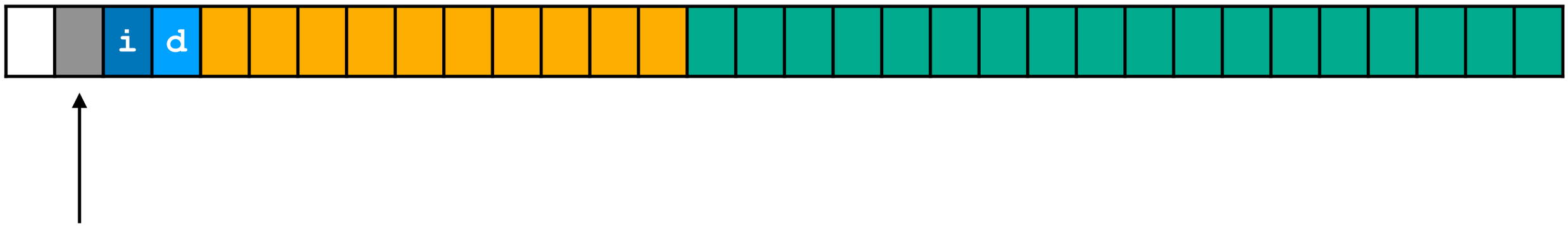
在执行各种文件操作时需分别访问磁盘上的哪些数据结构?

(涉及多少次磁盘读写操作)

文件操作

`mount()`

Super Block

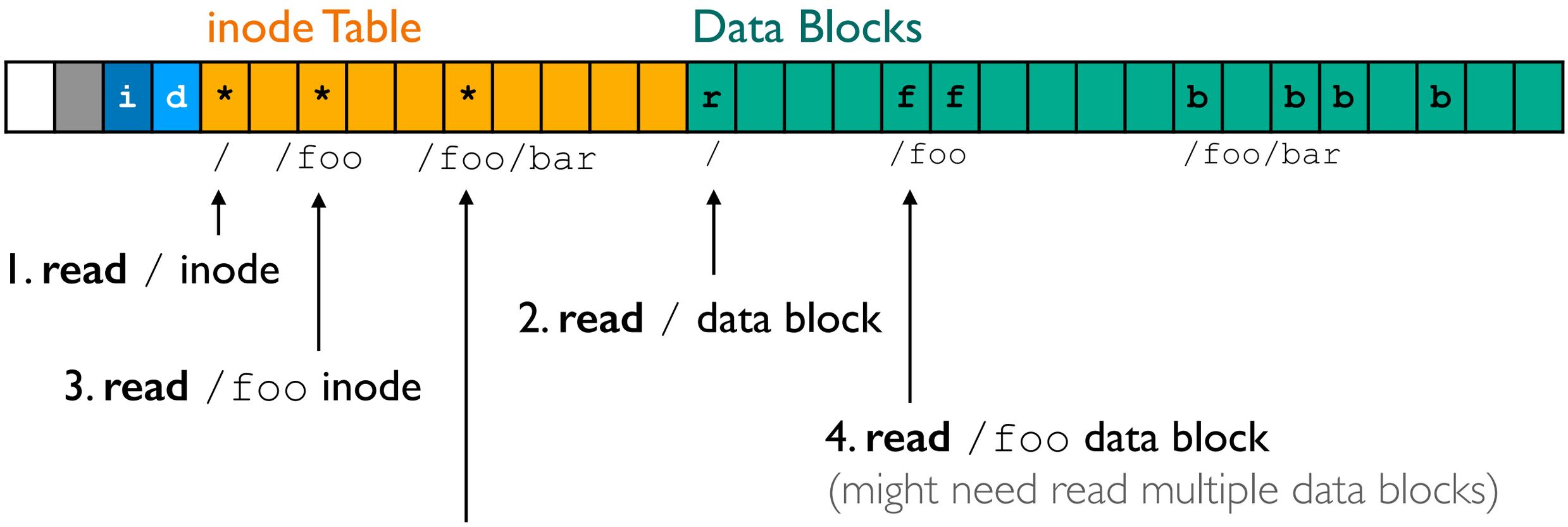


read super block

(metadata of the file system)

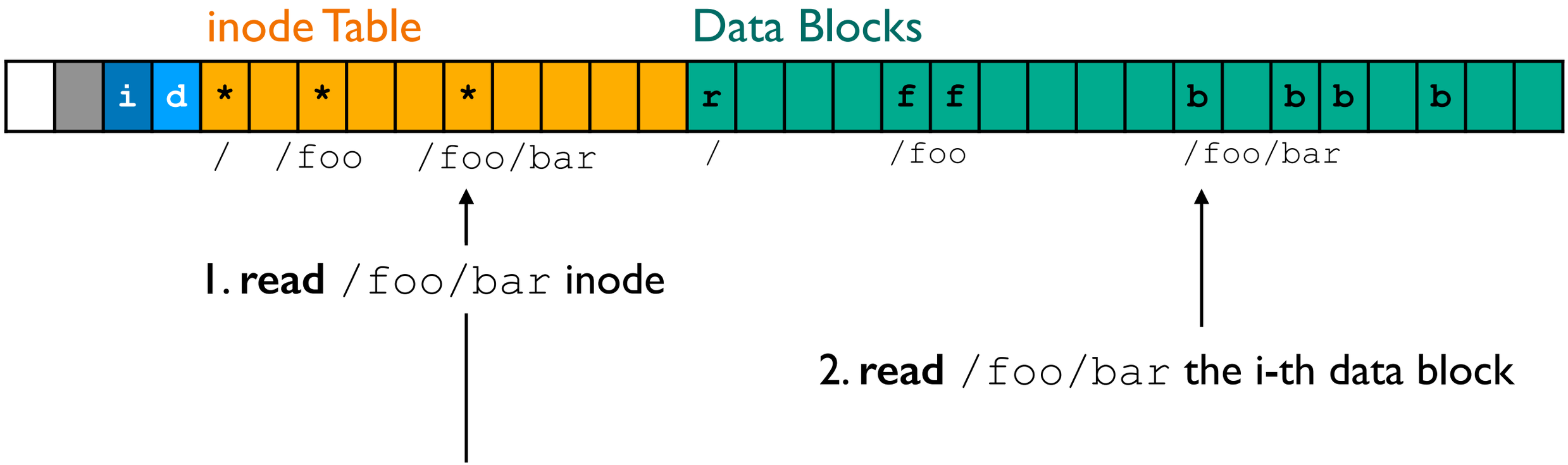
文件操作

`open("/foo/bar")`



文件操作

`read()`

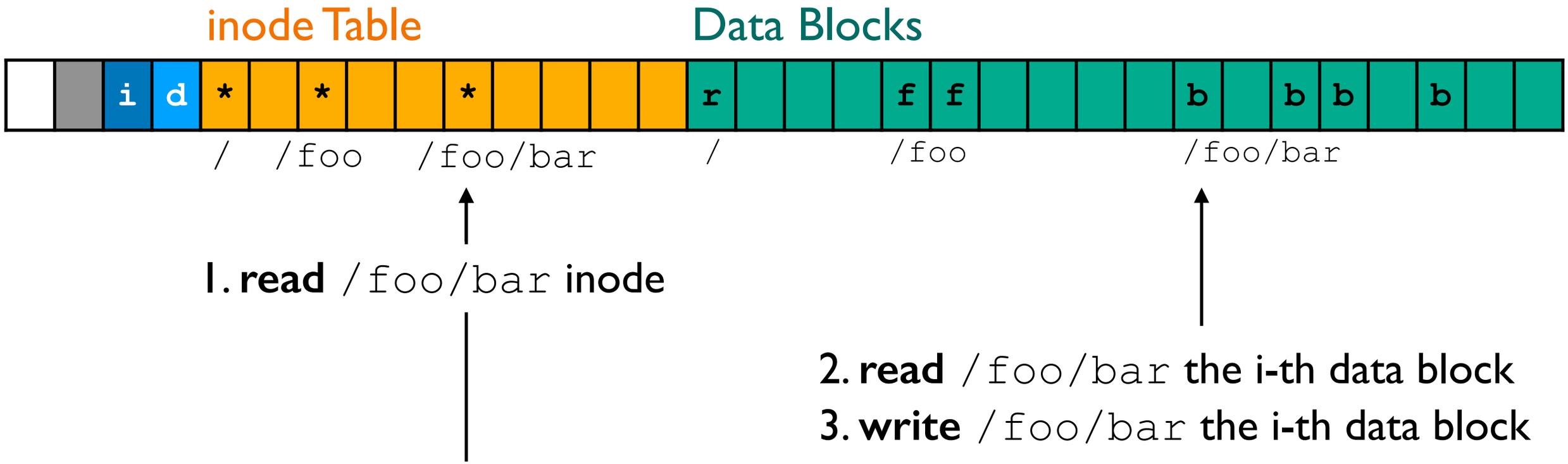


The Read-Copy-Update Pattern

(the block interface of disk)

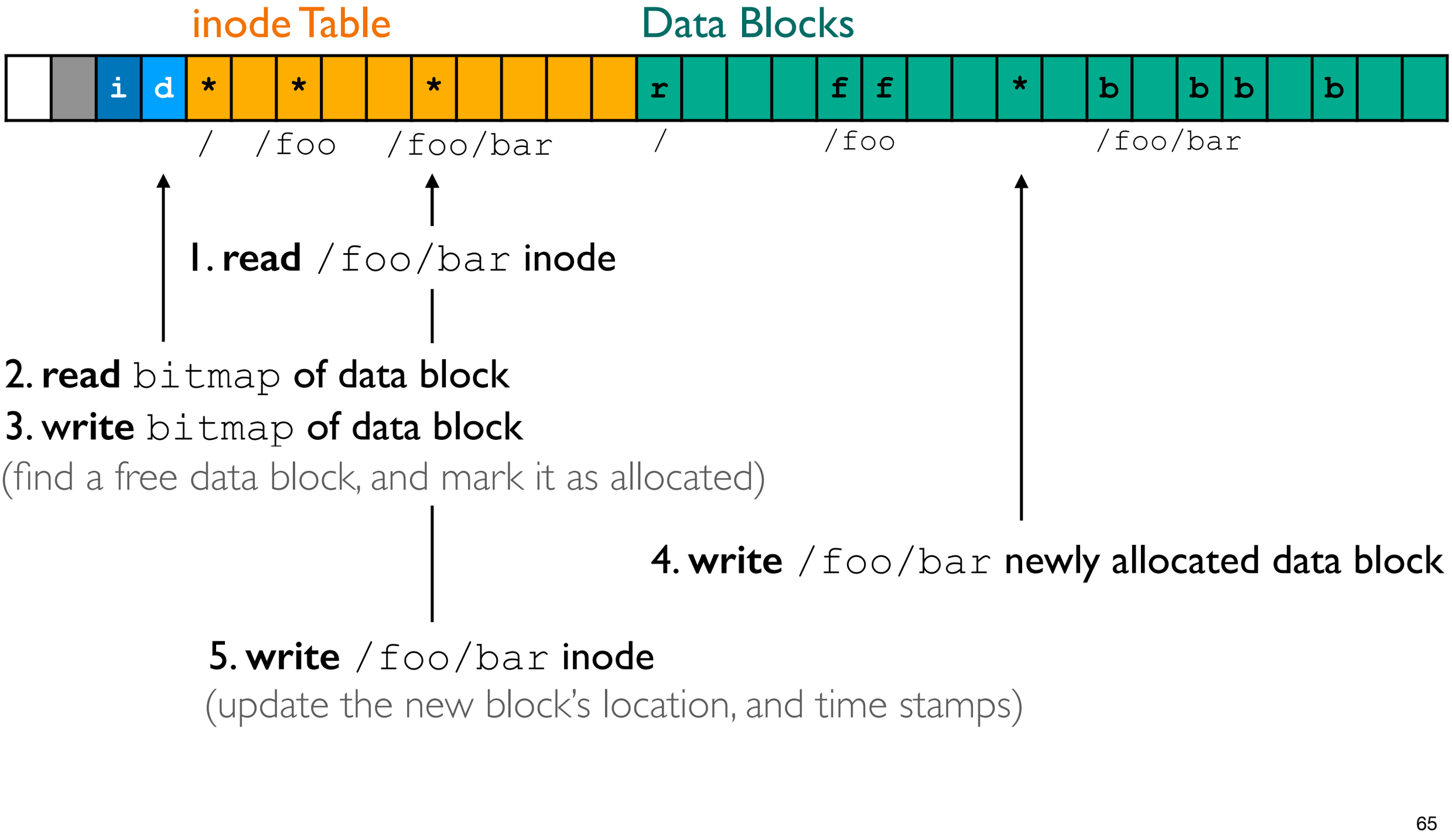
文件操作

write(over-written)



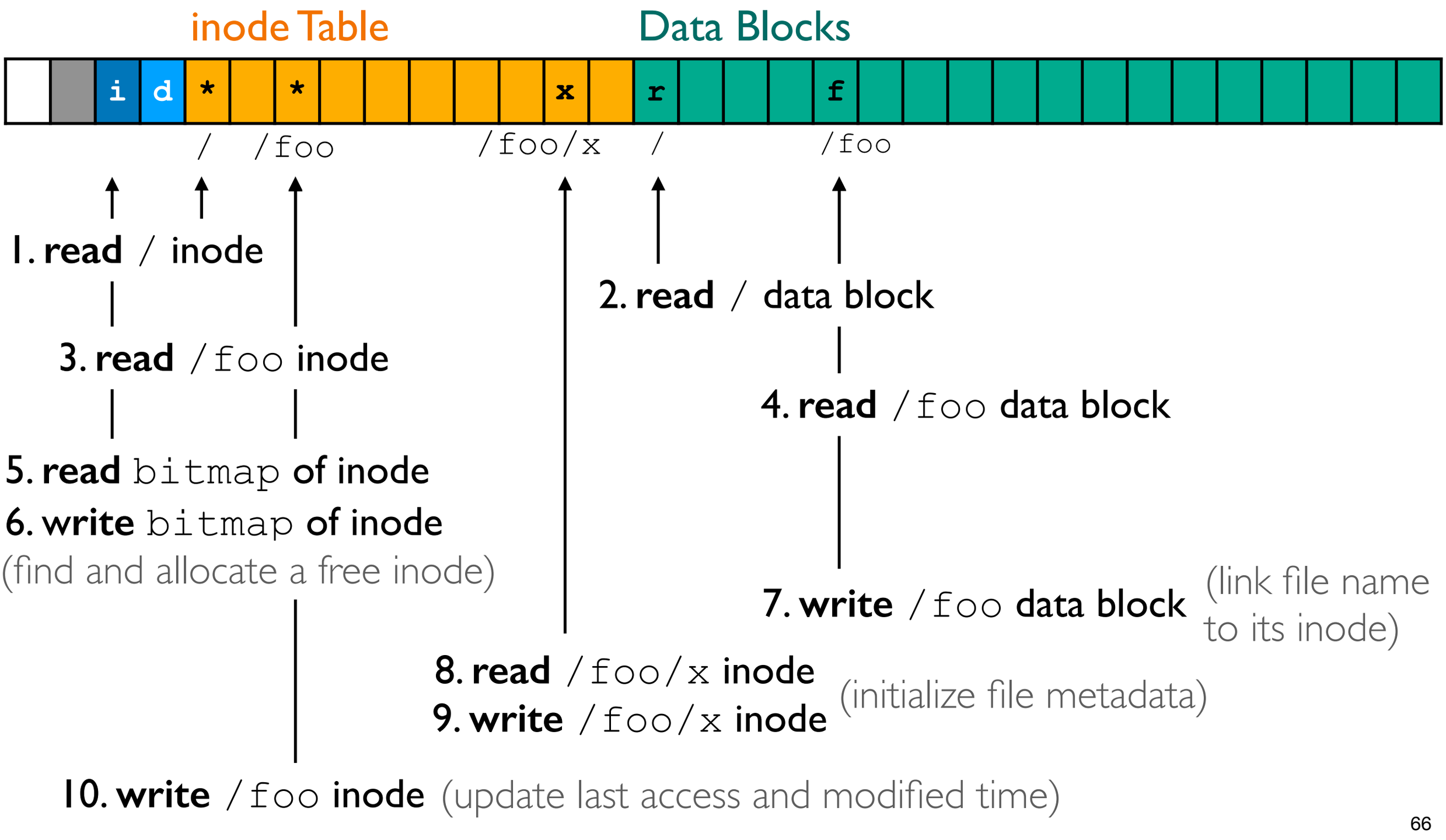
文件操作

`write(append)`



文件操作

`create("/foo/x")`



文件系统的性能问题

每次对磁盘数据块的 read 和 write 都是磁盘 I/O 操作 (比直接读写内存要慢很多)，需要尽量降低这些操作的开销

- 考虑局部性
  - Caching and Buffering
- 考虑存储介质 (磁盘) 的特点
  - Fast File System (FFS)
  - Log-Structured File System (LFS) *

Caching and Buffering

将频繁访问的数据缓存在内存中 (caching)

- 例如，对于文件的存储位置信息
  - FAT 将所有 data block 的 pointer 缓存在内存中
  - Unix 打开文件表中缓存了当前已打开文件的 inode 数据
- 缓存区满时需要考虑替换策略
  - 相比于物理页面替换，可以在文件系统中维护精确的 LRU 信息
    (文件系统是纯软件实现的系统，且涉及 I/O 操作)
- 可以通过预取 (pre-fetching) 技术来在 I/O 空闲时提前读取数据
  - 尤其在顺序访问文件时非常有效

Caching and Buffering

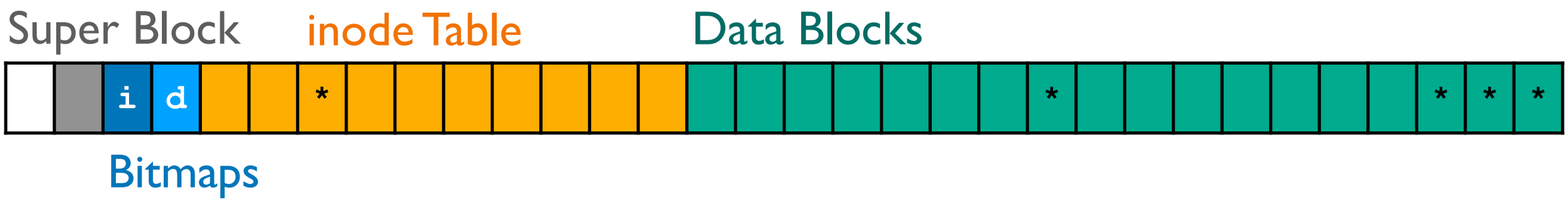
将需要写回磁盘的数据先暂存在内存缓冲区中 (delay writes)

- Write-through Policy: 修改后立即写回
  - 可能导致磁盘的大量随机写操作
  - 而且有些写操作是可以避免的 (e.g., create a file and then delete it)
- Write-back Policy: 修改后仅标记为 dirty 并延迟写回
  - 在超过一定时间间隔、缓冲区满、关闭文件、或显式调用 `flush()` 时才写回磁盘
  - 以牺牲潜在一致性的代价来提高性能
    - 系统奔溃时丢失尚未写回的数据

文件系统的性能问题

在设计文件系统布局 (file system layout) 时考虑存储介质的物理特性

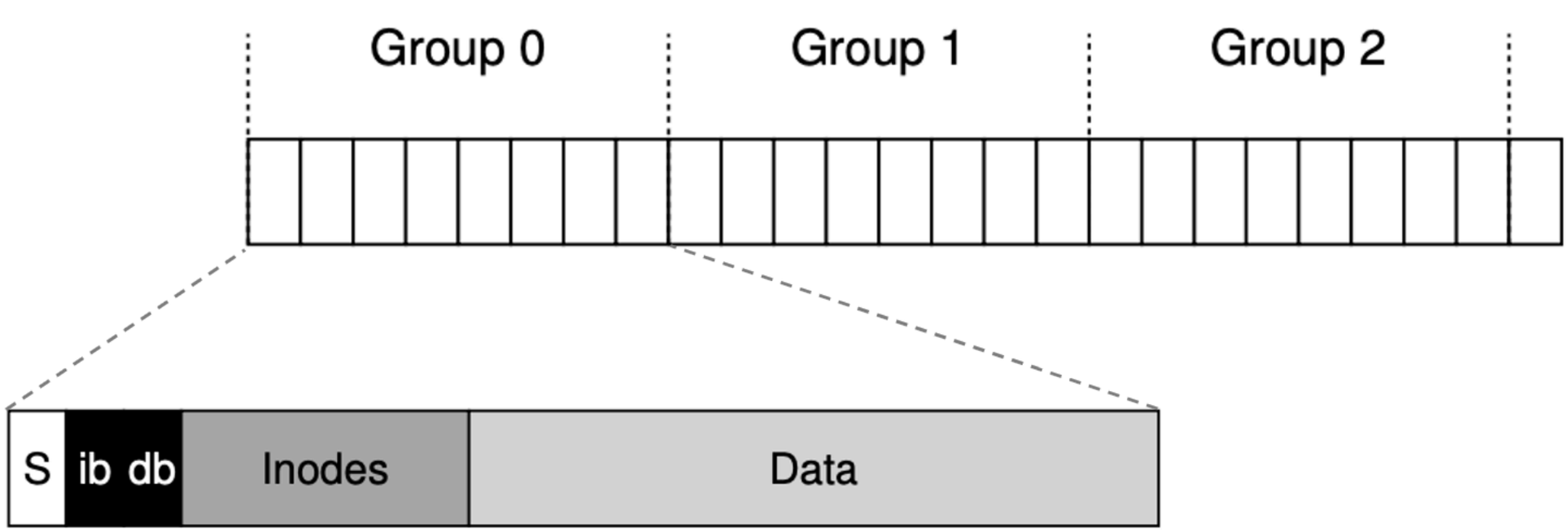
- 之前考虑的文件系统布局其实把磁盘看作了一种 random access 存储介质 (data is spread all over the place)
  - 所有 inodes 放在磁盘开始的一段空间中
  - 任意空闲的 data block 都可用于分配
  - 在文件操作时会导致太多的磁头移动 (disk arm motion) 开销



Fast File System (FFS)

FFS 将整个磁盘分为若干 block groups (Berkeley, 1984)

- 每个 group 有各自 allocation structure、inode table 和 data region



Fast File System (FFS)

FFS 将整个磁盘分为若干 block groups (Berkeley, 1984)

- 每个 group 有各自 allocation structure、inode table 和 data region
- 将可能依次访问的相关数据放在相邻位置 (keep related stuff together)
  - 在创建文件时
    - 同一文件的 inode 和 data 放在同一个 group
    - 同一目录下的文件放在同一个 group
  - 在创建目录时
    - 找一个已存储目录相对较少 (balance directories across groups) 且空闲 inode 相对较多 (be able to allocate a bunch of files) 的 group
  - 还可以通过碎片整理 (defragmenting) 来把数据放到一起

文件系统的性能问题

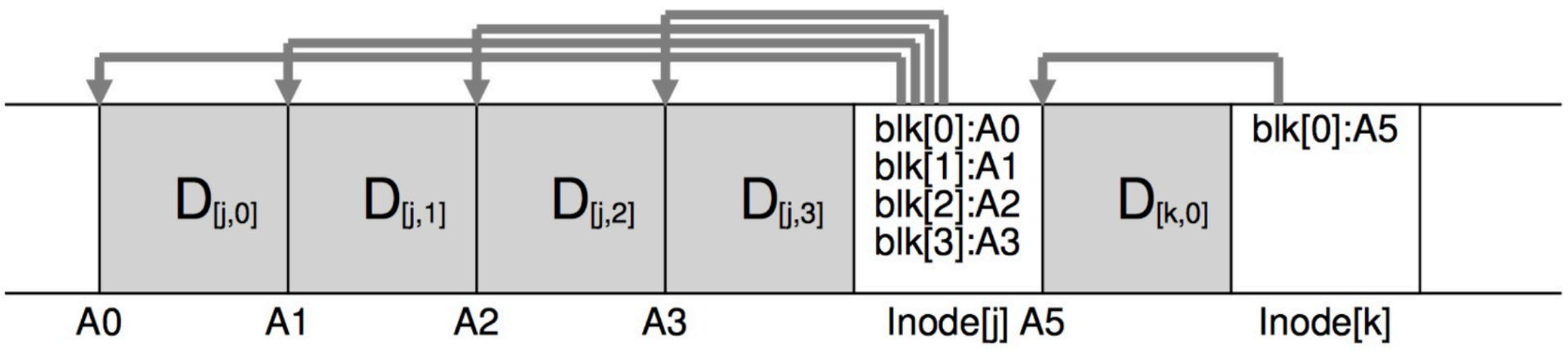
在设计文件系统布局 (file system layout) 时考虑存储介质的物理特性

- 硬件的发展带来更快的 CPU 和更大的内存
  - 可以通过缓存来加速对磁盘的读操作
  - 性能瓶颈主要集中于对磁盘的写操作
    - 随机写磁盘数据块性能很差
    - 尝试把很多小的随机写 (small random writes) 变成一个长的顺序写 (a long sequential write)

Log-Structured File System *

LFS 将整个磁盘看作一个巨大的 Logging 空间

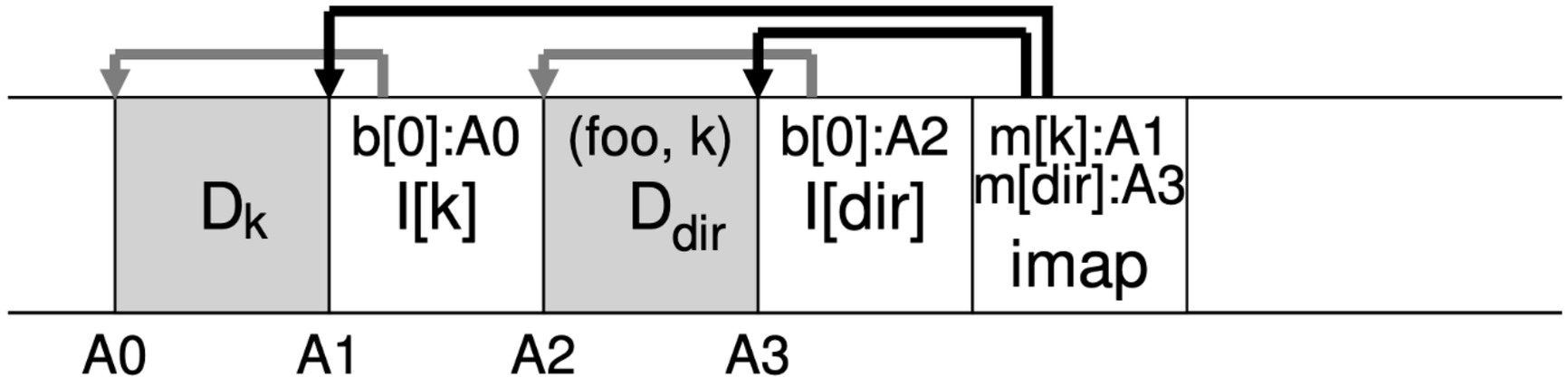
- 对 inode 和 data block 的写都先缓存在内存中 (mixed together)
- 缓冲区满时将其中内容以追加的方式顺序写入磁盘 (never overwrite existing data but always write to free locations)



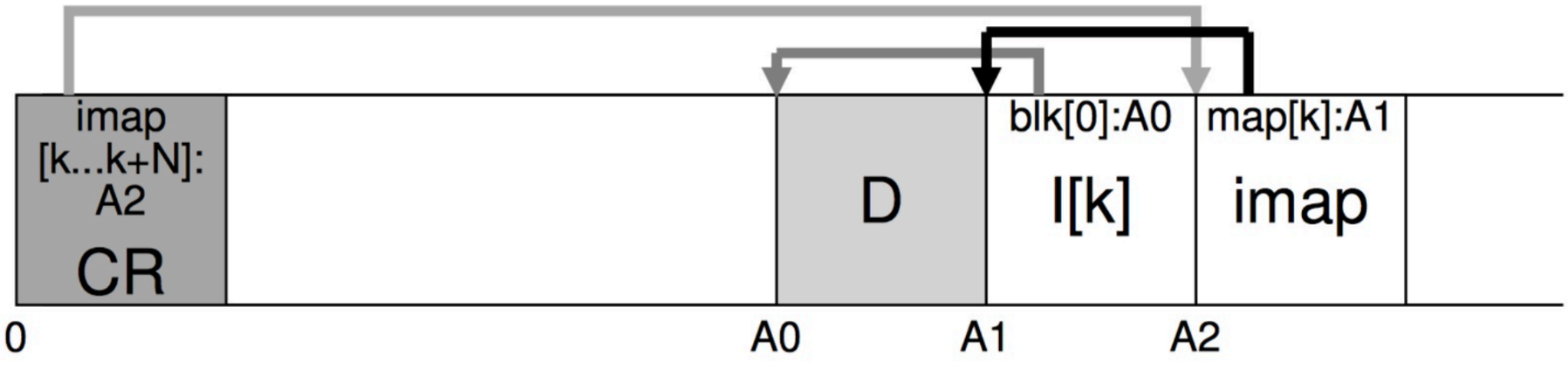
Log-Structured File System *

虽然避免了随机写的性能瓶颈，但是

- 对读操作带来了巨大的不便
  - 如何知道某个文件的 inode 在磁盘的什么地方?



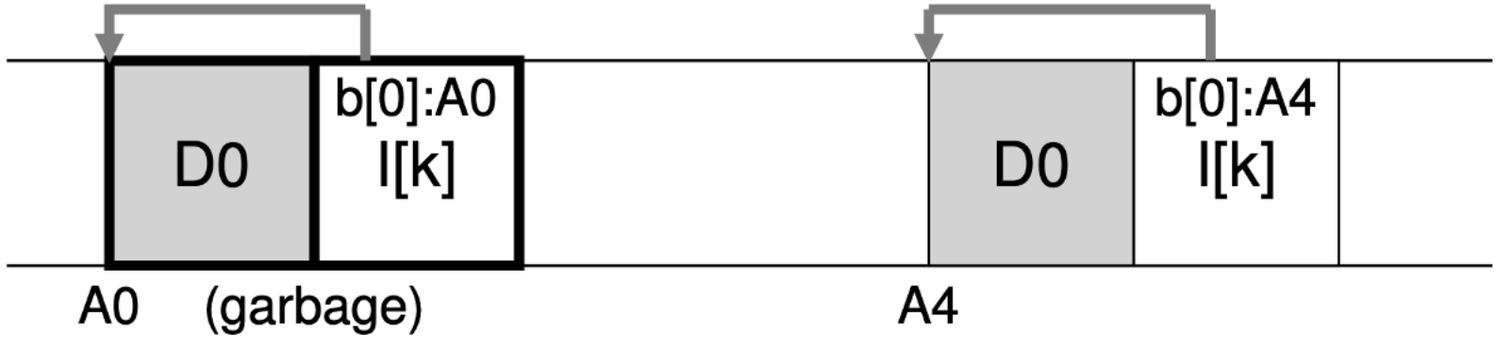
- 如何知道最新的 inode map 在磁盘的什么地方?



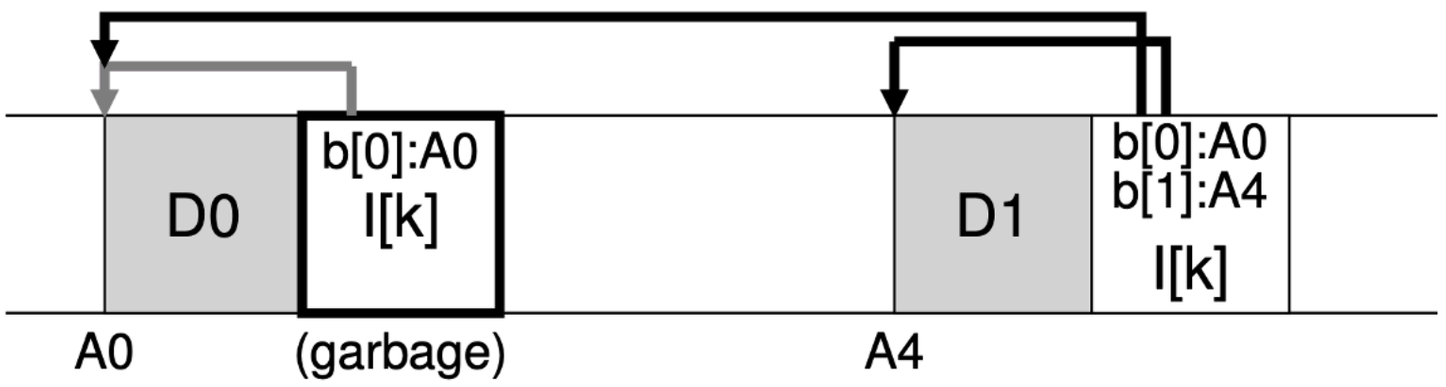
Log-Structured File System *

虽然避免了随机写的性能瓶颈，但是

- 对读操作带来了巨大的不便
- 对资源回收带来了巨大的不便 (garbage collection)



Rewrite an existing file



Append a data block to an original file

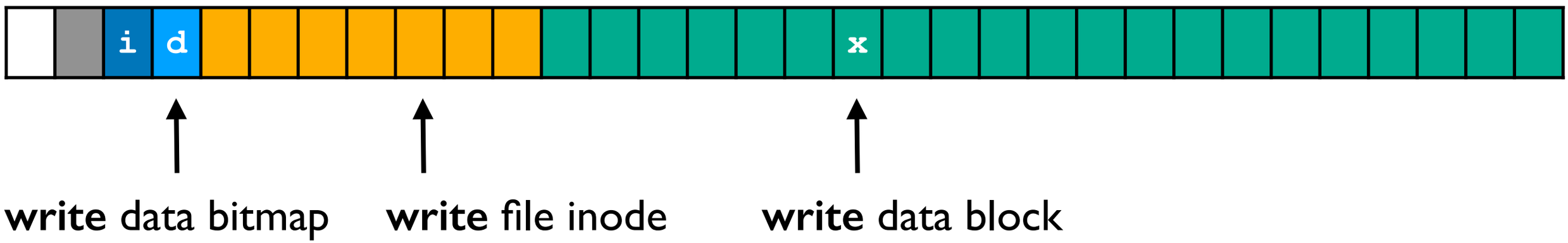
文件系统的性能问题

在设计文件系统布局 (file system layout) 时考虑存储介质的物理特性

- FFS 和 LFS 是为磁盘 (hard disk) 设计的
  - 存在磁头的物理移动
- 对于固态硬盘 (solid-state disk)
  - 接近 random access 的读写速度
    - 文件的 inode 和 data 放不放在一起就没那么重要了
  - 但每个 block 仅能读写一定的次数
    - 此时频繁将数据移动到相邻 blocks 甚至是有害的
    - 需要解决的问题变成 “如何确保尽可能均匀地使用 disk blocks“ ?

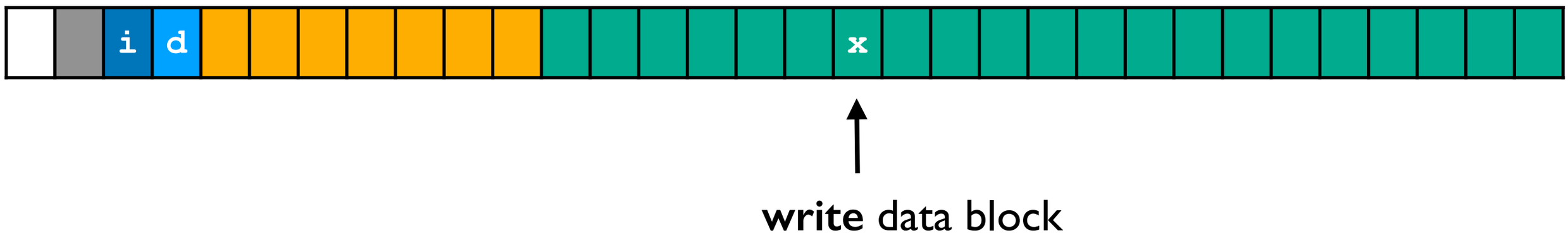
文件系统的一致性问题

对文件追加写一个新的 data block 共涉及三个磁盘写操作



文件系统的一致性问题

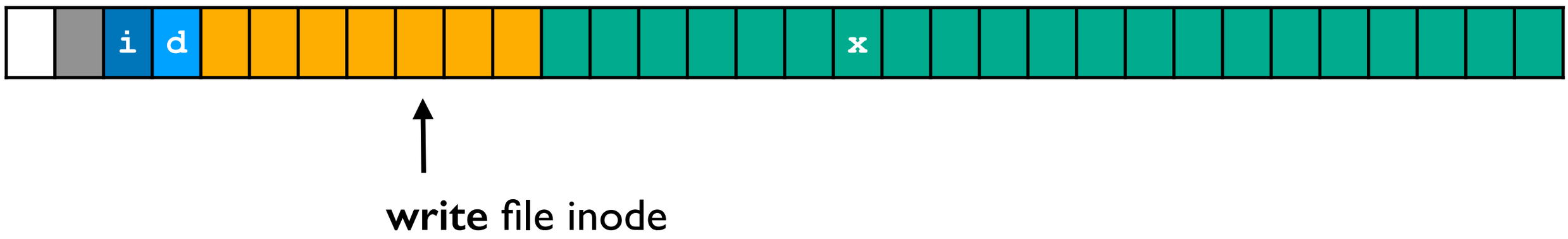
对文件追加写一个新的 data block 共涉及三个磁盘写操作



写丢失，但文件系统的 metadata 仍然是一致的

文件系统的一致性问题

对文件追加写一个新的 data block 共涉及三个磁盘写操作



文件系统 metadata 的不一致

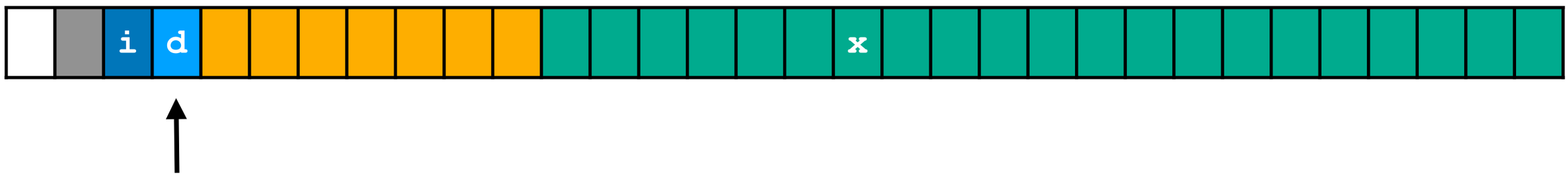
根据 data bitmap: data block *x* 是空闲块

根据 inode: data block *x* 属于某个文件

(read garbage data)

文件系统的一致性问题

对文件追加写一个新的 data block 共涉及三个磁盘写操作



write data bitmap

文件系统 metadata 的不一致

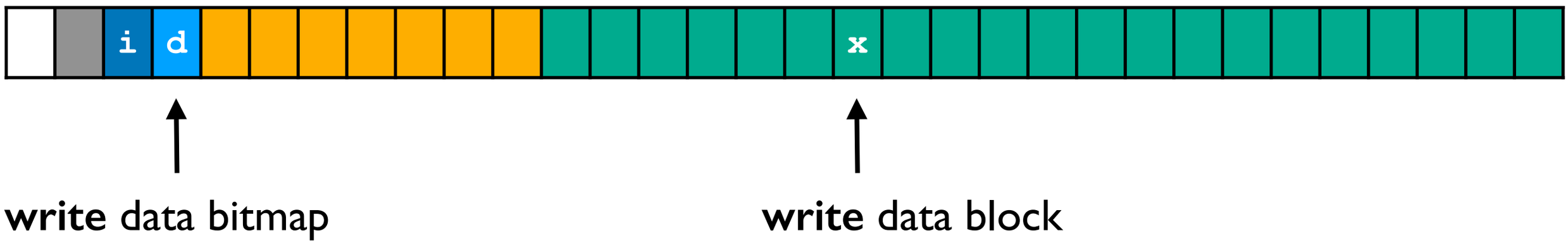
根据 data bitmap: data block *x* 已分配

根据 inode: data block *x* 不属于任何文件

(space leak)

文件系统的一致性问题

对文件追加写一个新的 data block 共涉及三个磁盘写操作

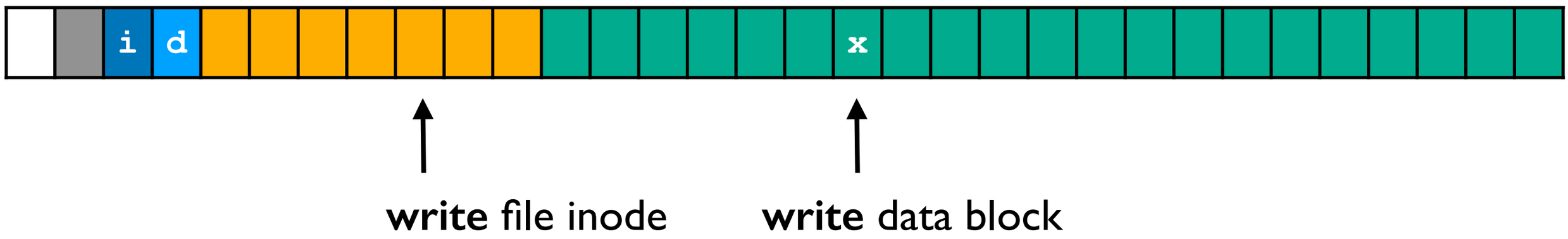


文件系统 metadata 的不一致

不知道 data block *x* 属于哪个文件

文件系统的一致性问题

对文件追加写一个新的 data block 共涉及三个磁盘写操作

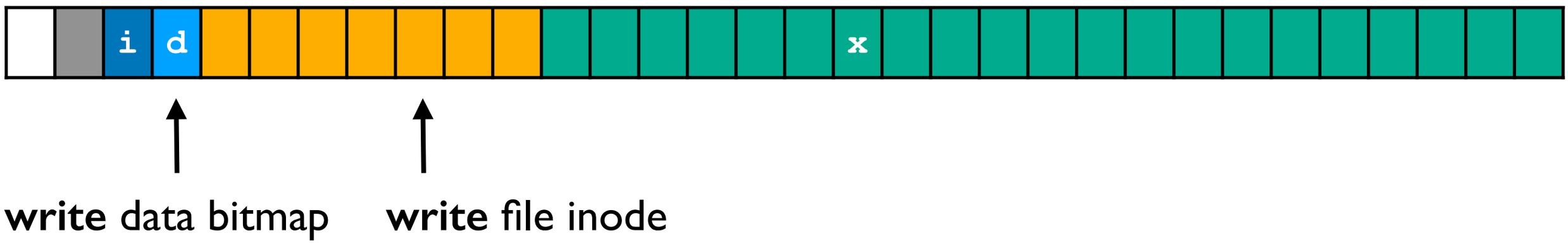


文件系统 metadata 的不一致

虽然 inode 指向了正确的 data block

文件系统的一致性问题

对文件追加写一个新的 data block 共涉及三个磁盘写操作



文件系统 metadata 保持一致，但会读到 garbage data

文件系统的一致性问题

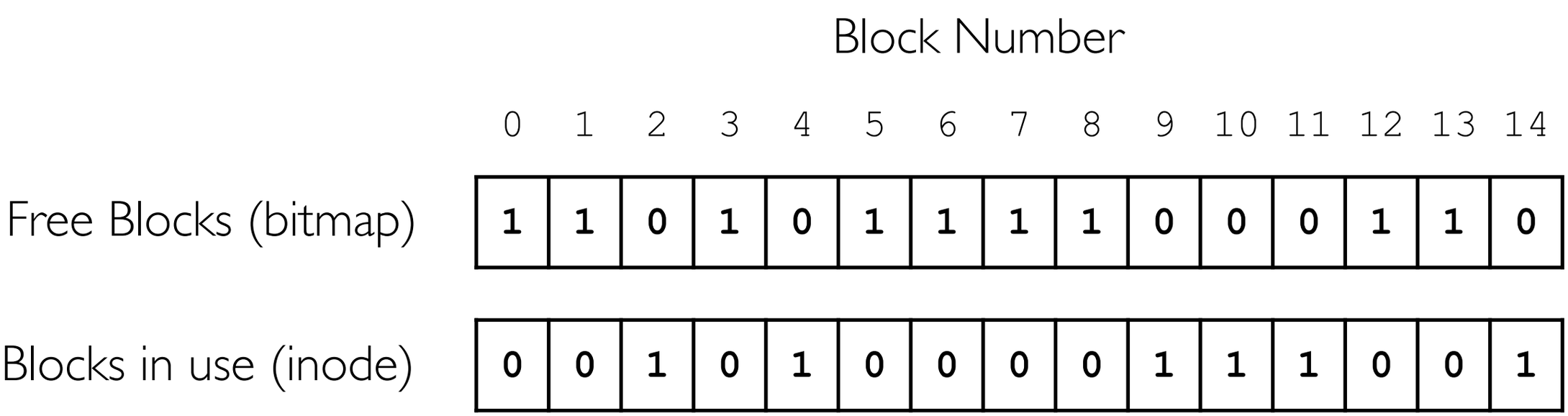
奔溃一致性 (crash-consistency): 确保文件系统总是从一个一致性状态原子性 (atomically) 地进入另一个一致性状态

- 对文件系统的更新往往涉及对磁盘中多个数据结构的修改
  - 在系统崩溃时，可能完成了一些写操作，但另一些没完成
  - 此时会导致文件系统的不一致状态 (inconsistent state)
- 不能假设
  - 对文件系统更新操作 (update operation) 是原子的
  - 多个磁盘写操作总是按预期顺序执行

文件系统一致性检查

在重启时检查文件系统是否存在不一致，并尝试进行修复

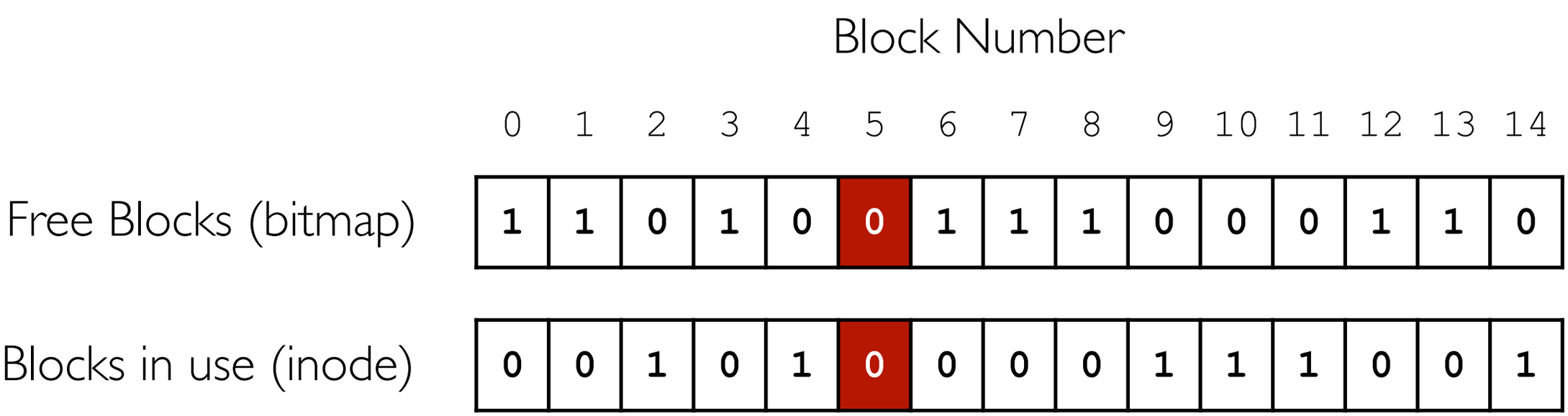
- 比较从 allocation structure 和 inode table 中读取的信息是否一致



文件系统一致性检查

在重启时检查文件系统是否存在不一致，并尝试进行修复

- 比较从 allocation structure 和 inode table 中读取的信息是否一致

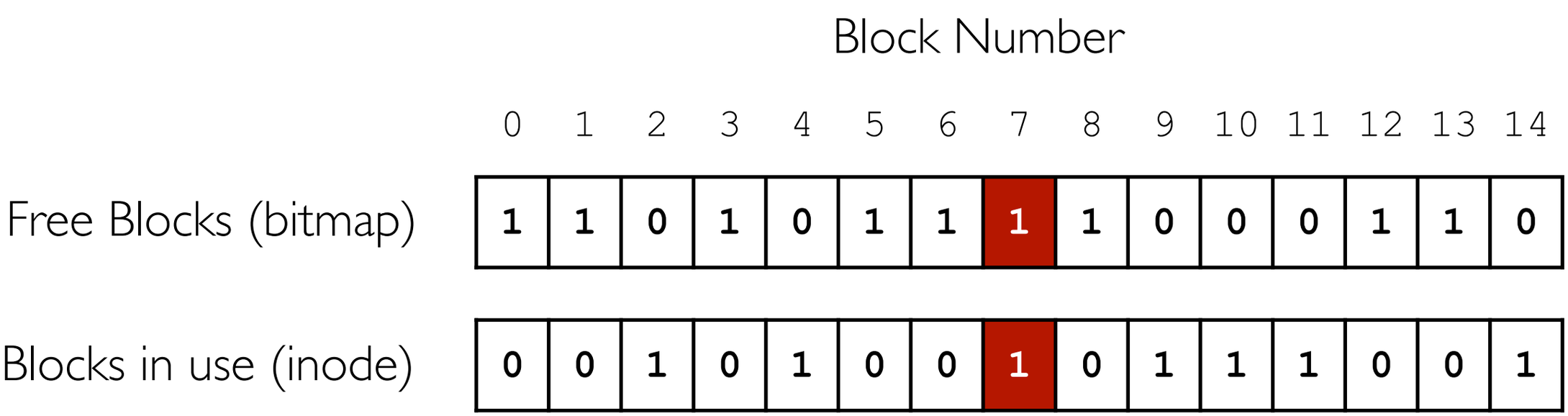


可以在 bitmap 中将该 data block 标记为 free block
(可能丢失了对 data block 的写操作)

文件系统一致性检查

在重启时检查文件系统是否存在不一致，并尝试进行修复

- 比较从 allocation structure 和 inode table 中读取的信息是否一致



可以在 bitmap 中将该 data block 标记为 allocated
(虽然确保一致，但 inode 可能指向 garbage data)

文件系统一致性检查

在重启时检查文件系统是否存在不一致，并尝试进行修复

- 比较从 allocation structure 和 inode table 中读取的信息是否一致
- Linux 中的 `fsck` 工具 (check and repair a Linux filesystem)
  https://docs.oracle.com/cd/E19683-01/806-4073/6jd67r9ln/index.html
- 文件系统一致性检查的实现：
  - 需要了解文件系统内部的各种底层细节信息 (require intricate knowledge of the file system)
  - 需要扫描整个文件系统 (very slow to run)
  - 仅能依赖启发式规则进行修复，无法保证修复所有可能问题

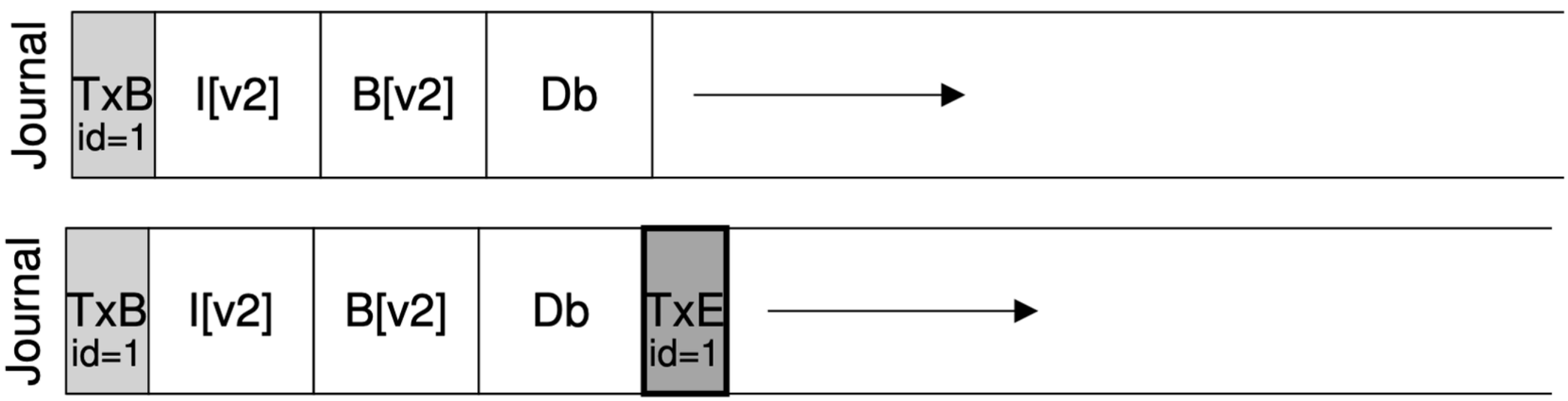
日志文件系统

通过在更新文件系统时额外多做一些事情来避免扫描整个磁盘的开销

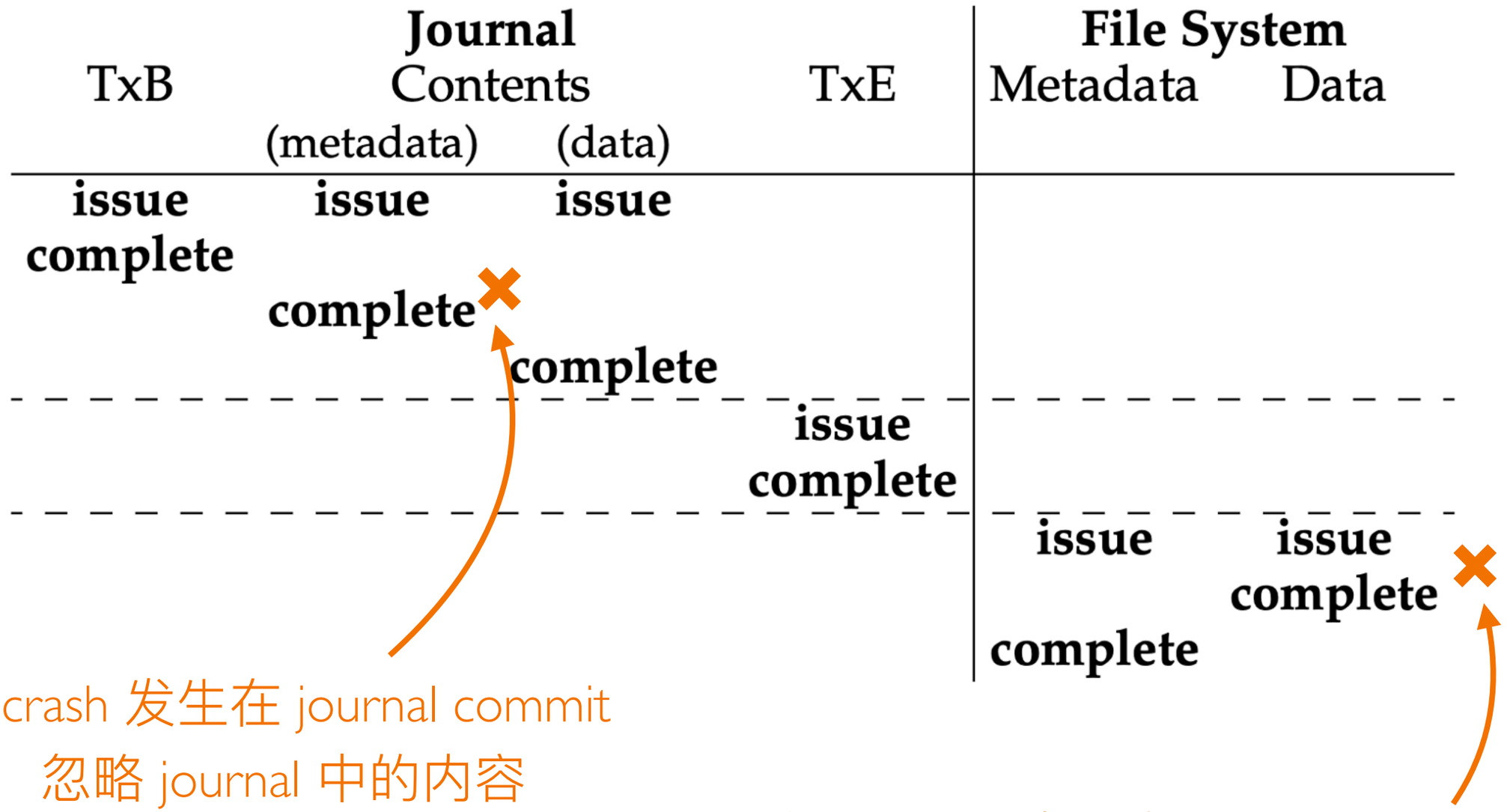
- 在更新数据之前，先把更新的意图记在日志中 (journal)
- 如果系统发生了 crash
  - 如果 crash 发生在 journal 之前：不需要做任何事情
  - 如果 crash 发生在 journal 之后：重做 journal 中记录的操作
- 一种 write-ahead logging 的思想 (from database systems)

日志文件系统

- 在 journal 中写 `TxB` (transaction start) **(journal write)**
- 在 journal 中写 inode `I[v2]`, bitmap `B[v2]` 和 data block `Db`
- 在 journal 中写 `TxE` (transaction end) **(journal commit)**
- 向磁盘中相应位置写 `I[v2]`, `B[v2]` 和 `Db` **(checkpoint)**



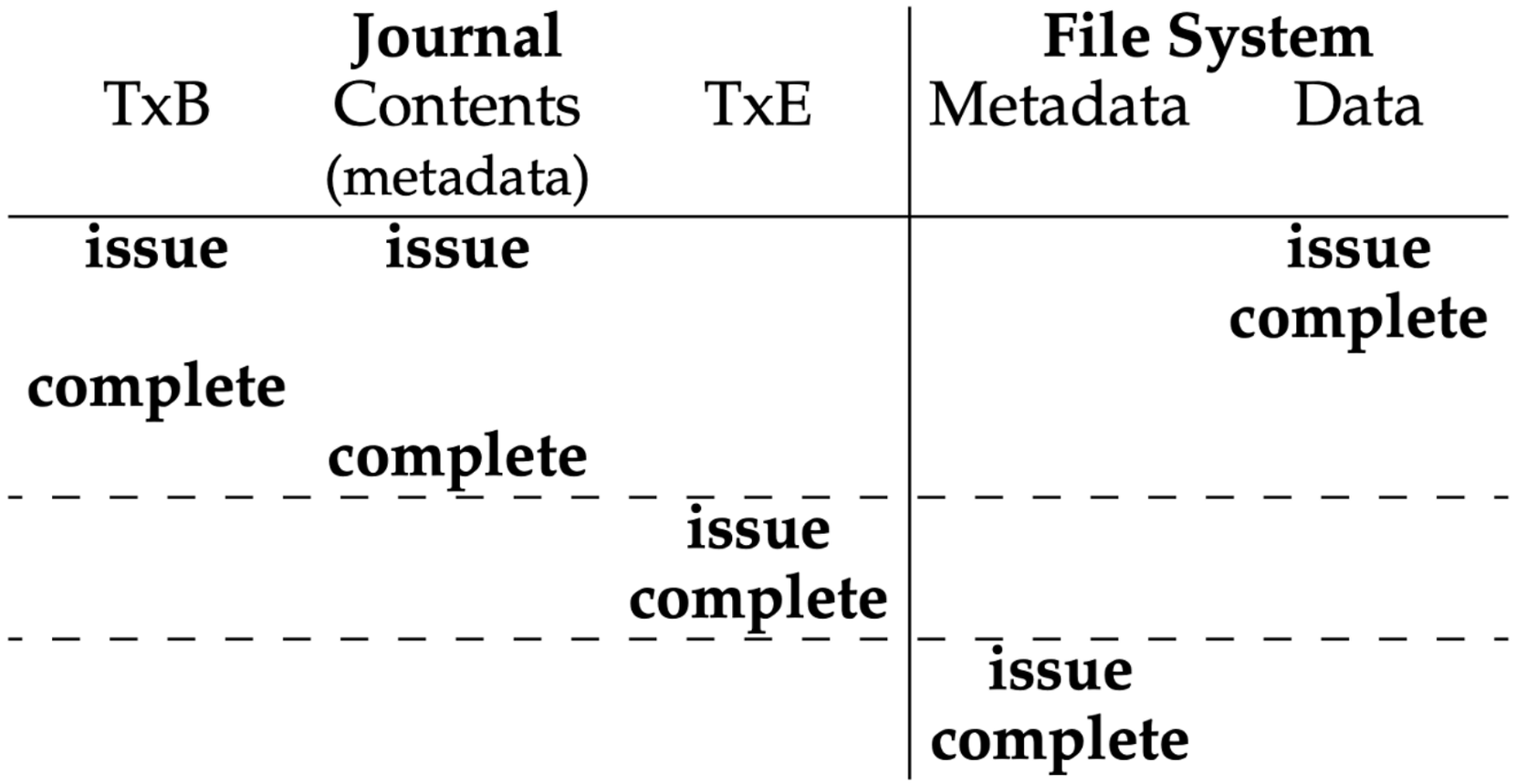
日志文件系统


Journal
File System
TxB
Contents
TxE
Metadata
Data
(metadata)
(data)
issue
complete
issue
complete
issue
complete
issue
complete
issue
complete
issue
complete
如果 crash 发生在 journal commit 之前，忽略 journal 中的内容
如果 crash 发生在 journal commit 之后，重做 journal 中记录的操作 (可能同一份数据写了两次)

日志文件系统

- 用户数据 `Db` 在 journal 和实际磁盘块中各写了一次 (Data Journaling)
- 为了降低开销，在 journal 中不记录用户数据 (Metadata journaling)
  - 什么时候向磁盘写 data block？
  - 向 journal 中写完 metadata 后再写 data block
    - 虽然确保了一致性，但 inode 可能指向 garbage data
  - 向磁盘写完 data block 后再 journal commit
    (先准备好数据，再更新指向该数据的 pointer)
    - 确保 inode 不会指向 garbage data

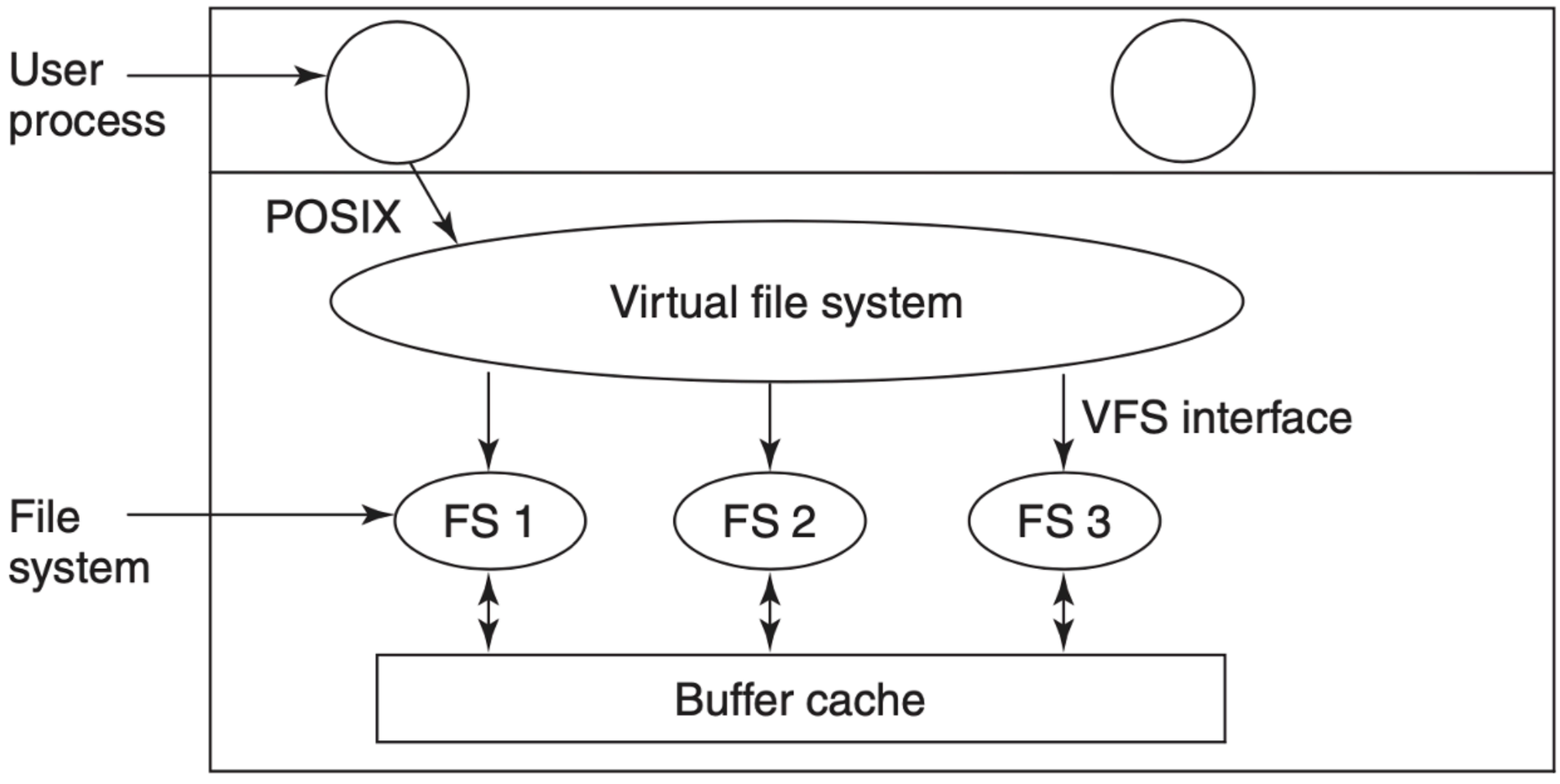
日志文件系统


Journal
File System
TxB
Contents
(metadata)
TxE
Metadata
Data
issue
issue
issue
complete
complete
complete
issue
complete
issue
complete

虚拟文件系统

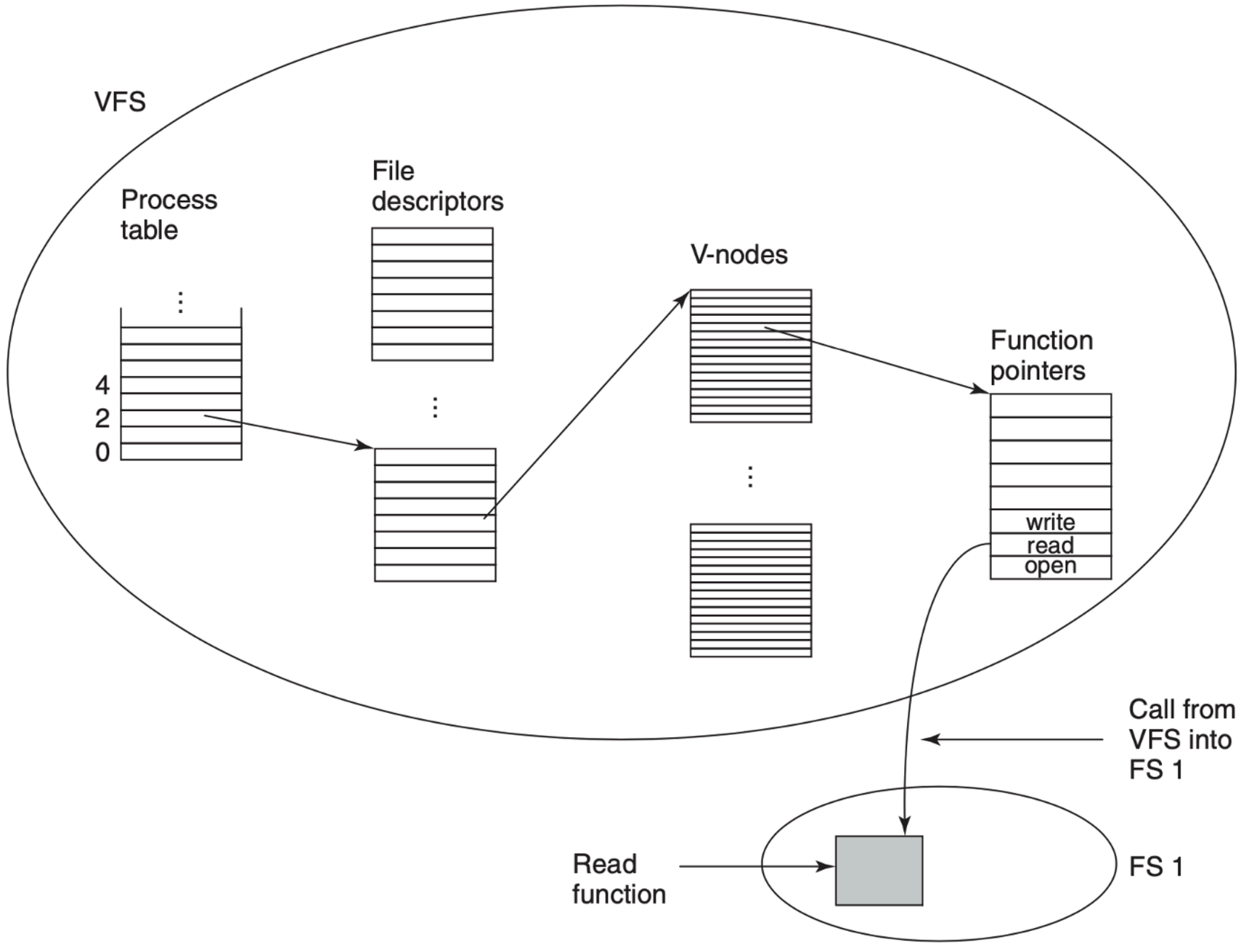
将不同文件系统所需实现的共性功能抽象为一组通用 APIs (a common interface)，不同的文件系统以不同的方式进行实现

- 以同一个文件系统层次视角使用多个 (互不兼容的) 的文件系统



虚拟文件系统

在内存中创建一个 v-node 结构来记录文件的属性信息，并记录该文件对应文件系统实现各指定 APIs 的函数地址



总结

- 文件和目录 
  - 文件描述符、inode 和打开文件表 (Open File Table)
  - 解析文件名 (Resolve File Name)
  - 硬链接 (Hard Link) 和符号链接 (Symbolic Link)
- 文件系统的布局 (Layout)
- 文件的实现方式 
  - FAT 文件系统的 File Allocation Table (FAT) 结构及其目录项
  - Unix 文件系统的 inode 索引结构及其目录项
  - 文件大小和文件占用磁盘空间

总结

- 文件不同操作涉及的磁盘访问情况 
- 文件系统的性能问题
  - 快速文件系统 (Fast File System)
  - 日志结构文件系统 (Log Structured File System)
- 文件系统的一致性问题 
  - 文件系统一致性检查 (File System Consistency Check)
  - 日志文件系统 (Journaling)
- 虚拟文件系统