文件系统
File System

钮鑫涛
南京大学
2026春

本门课由蒋炎岩和钮鑫涛共同构建

文件系统

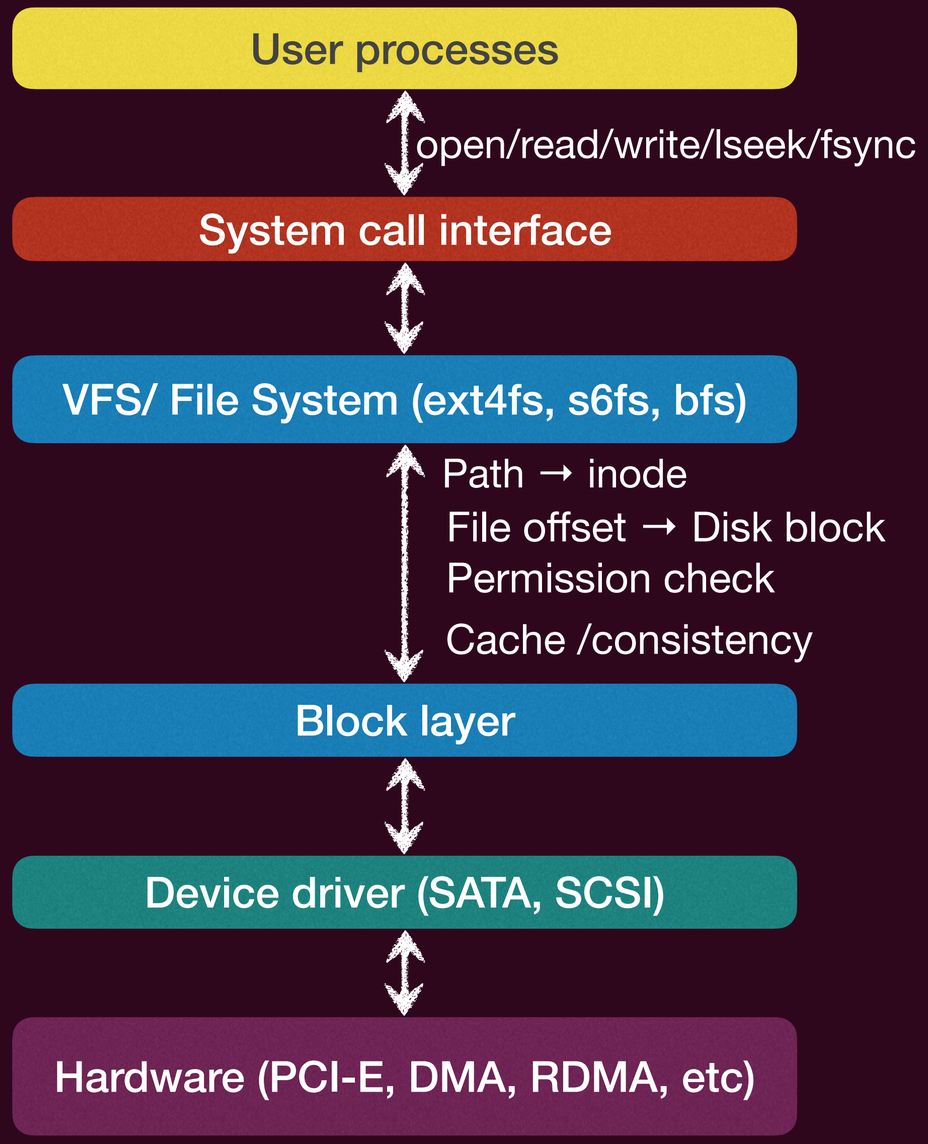
- 磁盘中存储的数据
  - 程序数据
    - 可执行文件、动态链接库、应用数据
  - 用户数据
    - 文档、下载、截图
  - 系统数据
    - 配置文件 (/etc)

文件系统

- 存储和读取本身没有问题，磁盘驱动器就是做这个的
- 但让应用程序直接通过驱动访问存储设备 (1950s)?
  - 程序出bug了（不可避免），完全可能弄坏整块磁盘
  - 连带着所有数据，包括操作系统都直接损坏
- 因此应用程序应该“有限制”地访问数据
  - 1.提供合理的 API 使多个应用程序能共享数据
  - 2.提供一定的隔离，使恶意/出错程序的伤害不能任意扩大
  - 这就是文件系统！

文件系统

- 文件系统是在操作系统内核中实现的持久化存储抽象。
  - 用户程序通过系统调用(open/read/write)与文件系统进行交互。
  - 文件系统负责维护路径、目录、inode、权限、空闲空间和文件到磁盘块的映射
  - 设备驱动程序负责把文件系统发出的块读写请求转换为具体设备操作。



文件系统

- 文件系统解决的四个问题：
  - **持久性和命名数据**：文件和目录
    - 存储在系统中直到显式删除为止
    - 可以通过文件系统关联的可读标识符访问
  - **访问和保护**：提供打开、读取、写入和其他操作；调节不同用户对文件的访问。
  - **磁盘空间管理**：公平有效地利用磁盘空间
    - 分配空间给文件，并跟踪空闲空间
    - 快速访问文件
  - **可靠性**：不得丢失文件数据

文件（File）

- 文件是操作系统创建的逻辑存储单元，用于存储信息
  - 可以是数据库、音频、视频、网页等内容。
  - 它是一组数据集合（类型由用户定义）
  - 可以创建、读取、写入和删除
  - 提供了一种在磁盘上存储信息并随后读取的抽象机制。

文件命名

- 命名是文件系统抽象的一个重要特征
  - 提供人类可读的名称
    - 用一个具有意义的单一名称来引用任意大小的数据
  - 帮助用户组织大量的存储空间。
  - 信息存储的细节（低级结构）以及磁盘的实际工作方式被屏蔽了。

文件命名

- 文件的命名规则因系统而异
  - 长度和特殊字符， 字母大小写
- 文件扩展名：例如.txt，.c
  - 表示文件内容的某些类型
  - 为应用程序或操作系统提供文件合理操作的提示

文件类型

- 许多操作系统支持多种类型的文件
  - 普通文件(‘-’)：用于存储实际的用户信息
    - 如文本、源码、图像或可执行程序
  - 目录(‘d’)：用于维护文件系统结构的系统文件
  - 符号链接文件（‘l’）
  - 命名管道文件或简称管道文件（‘p’）
  - 块文件（‘b’）
  - 字符设备文件（‘c’）
  - 套接字文件（‘s’）

文件元数据（属性）

- 除了文件的名称和数据外，操作系统还会保留文件的额外信息：
  - 数据块位置（文件对应哪些磁盘块）
  - 大小：文件的大小（当前大小或最大大小）
  - 时间：文件的创建时间、最近访问时间和最近修改时间
  - 所有者：文件的当前所有者
  - 保护信息
  - 链接信息
  - …

文件元数据（属性）

- 文件系统应该将文件元数据保存在一个结构中（文件控制块）：
  - 存储在磁盘上，并且缓存在内存中以加快访问速度。
- 在UNIX中，这个文件控制块就是 inode（index node）

Simplified ext2 Inode

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	osdl	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

文件访问

- 顺序访问：按顺序读取或写入数据
  - 读取下一个/写入下一个
  - 最常见的访问模式（例如，复制文件，编译器读取和写入文件）
  - 速度快（可以达到磁盘的峰值传输速率）
- 随机（直接）访问：随机寻址任意块
  - 读取[n] 写入[n] 寻址[n]
  - 文件操作包括块号作为参数
  - 速度慢（寻址时间和旋转延迟）

文件的常见API（分别对应了系统调用）

- `open` (or `create`)：打开（或创建）具有给定路径（目录和名称）的文件，并将文件指针设置为文件的开头
- `read`：从打开的文件中读取最多一定数量的字节，并为下一次读取移动文件指针
- `write`：将字节数组写入打开的文件（并移动指针）
- `close`：关闭打开的文件
- `lseek`： 移动文件指针到文件中的某个索引处
- `fsync`：立即将更改推送到磁盘（刷新脏数据）

拷贝文件

```c
#include <sys/types.h> /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);  /* ANSI prototype */
#define BUF_SIZE 4096    /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700  /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];
    if (argc != 3) exit(1);  /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY); /* open the source file */
    if (in_fd < 0) exit(2);   /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
    if (out_fd < 0) exit(3); /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
        if (rd_count <= 0) break;  /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4); /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0) /* no error on last read */
        exit(0);
    else
        exit(5);  /* error on last read */
}
```

文件描述符（File descriptor）

- 文件描述符（句柄）：操作系统分配给一个进程打开的文件的一个唯一数字（每个进程私有），用于引用该文件。
  - 持有该文件描述符，可以对对应的文件执行特定操作
  - 避免在每次访问时解析文件名（在目录中搜索文件名）和检查权限
  - 一个文件可以以不同方式多次打开

文件描述符

- 在Unix系统中，一切皆是文件（字节流）。
  - 常规文件、目录、管道、设备、套接字等都是文件的一种形式。
- 通常对从Shell运行的程序可用的三个特殊文件描述符

FD	Purpose	Posix name	stdio stream
0	Standard input	`STDIN_FILENO`	*stdin*
1	Standard output	`STDOUT_FILENO`	*stdout*
2	Standard error	`STDERR_FILENO`	*stderr*

文件偏移

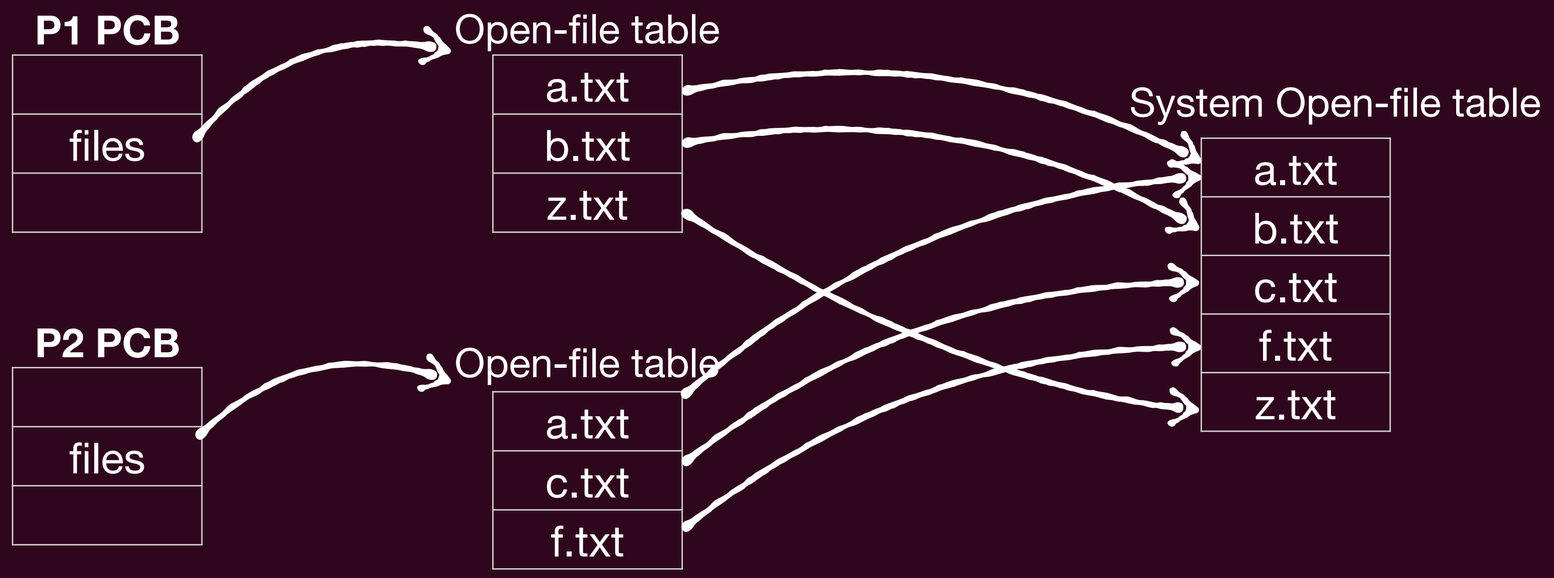
- 对于进程打开的每个文件，操作系统都会跟踪一个文件偏移量，该偏移量决定下一次读取或写入将从何处开始。
  - 隐式更新：当进行N字节的读取或写入时，N会被添加到当前偏移量。
  - 显式更新：使用lseek()函数。

打开文件表

- 当进程打开一个文件时，操作系统应该创建一些额外的数据结构（在内存中），用于存储关于进程打开文件的信息。
- 每个进程都维护一个打开文件表
  - 一个由文件描述符索引的数组
  - 表中的每个条目跟踪文件描述符所引用的底层文件，当前偏移量以及其他相关细节（例如文件大小、位置、权限等）

打开文件表

- 多个进程可能同时打开同一个文件
  - 每个进程都有自己的打开文件表：跟踪进程打开的所有文件
  - 系统范围的打开文件表：跟踪与进程无关的信息（如文件属性，大小和位置）



Multiple entries of per-process open-file table can point to the same entry of system open-file table

打开文件表

- 要打开一个文件，搜索系统范围的打开文件表，以查看文件当前是否正在使用
  - 如果没有，搜索目录以查找文件名，并在系统范围的打开文件表中添加一个条目
- 在属于进程的打开文件表中创建一个打开文件的条目，并指向系统的打开文件表
- 增加系统的打开文件表中的打开计数
  - 只有当所有进程关闭文件（或退出）时，才可以删除表条目
- 返回指向每个进程打开文件表中条目的指针（文件描述符）

文件描述符和打开文件表

- 在Unix中，有三种内核数据结构用于描述文件描述符和打开文件之间的关系：
  - 文件描述符表（每个进程）：为进程打开的每个文件描述符创建一个条目
    - 指向打开文件表的指针
  - 打开文件表（系统范围）：为每个打开的文件创建一个条目
    - 文件偏移量、访问模式、与文件打开相关的标志
    - 指向inode的指针
  - inode表（系统范围）：为每个inode点创建一个条目
    - 文件属性（类型、大小、权限、时间戳等）

打开文件表

Process A
File descriptor table

FD	File ptr
0	
1	
2	
…	
20	

Process B
File descriptor table

FD	File ptr
0	
1	
2	
3	
…	

Open File Table
(System-wide)

ID	File offset	Status flags	inode ptr
0	…	…	
…	…	…	…
23	…	…	
…	…	…	…
73	…	…	
…	…	…	…
86	…	…	
…	…	…	…

Inode table
(System-wide)

ID	file type	file locks	…
…	…	…	…
224	…	…	…
…	…	…	…
1976	…	…	…
…	…	…	…
5139	…	…	…
…	…	…	…

两个进程打开相同文件

Process A
File descriptor table

FD	File ptr
0	
1	
2	
…	
20	

Process B
File descriptor table

FD	File ptr
0	
1	
2	
3	
…	

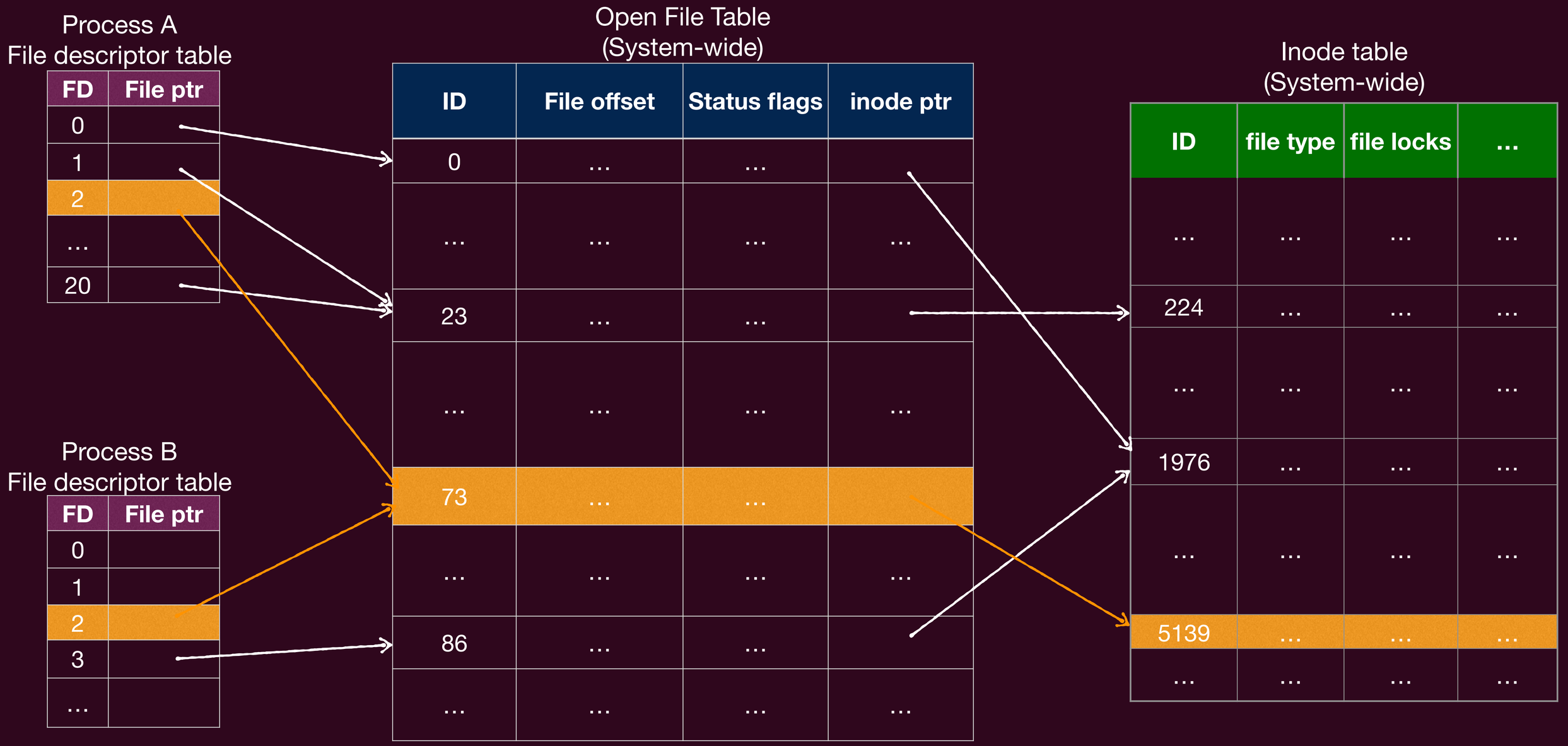
Open File Table
(System-wide)

ID	File offset	Status flags	inode ptr
0	…	…	
…	…	…	…
23	…	…	
…	…	…	…
73	…	…	
…	…	…	…
86	…	…	
…	…	…	…

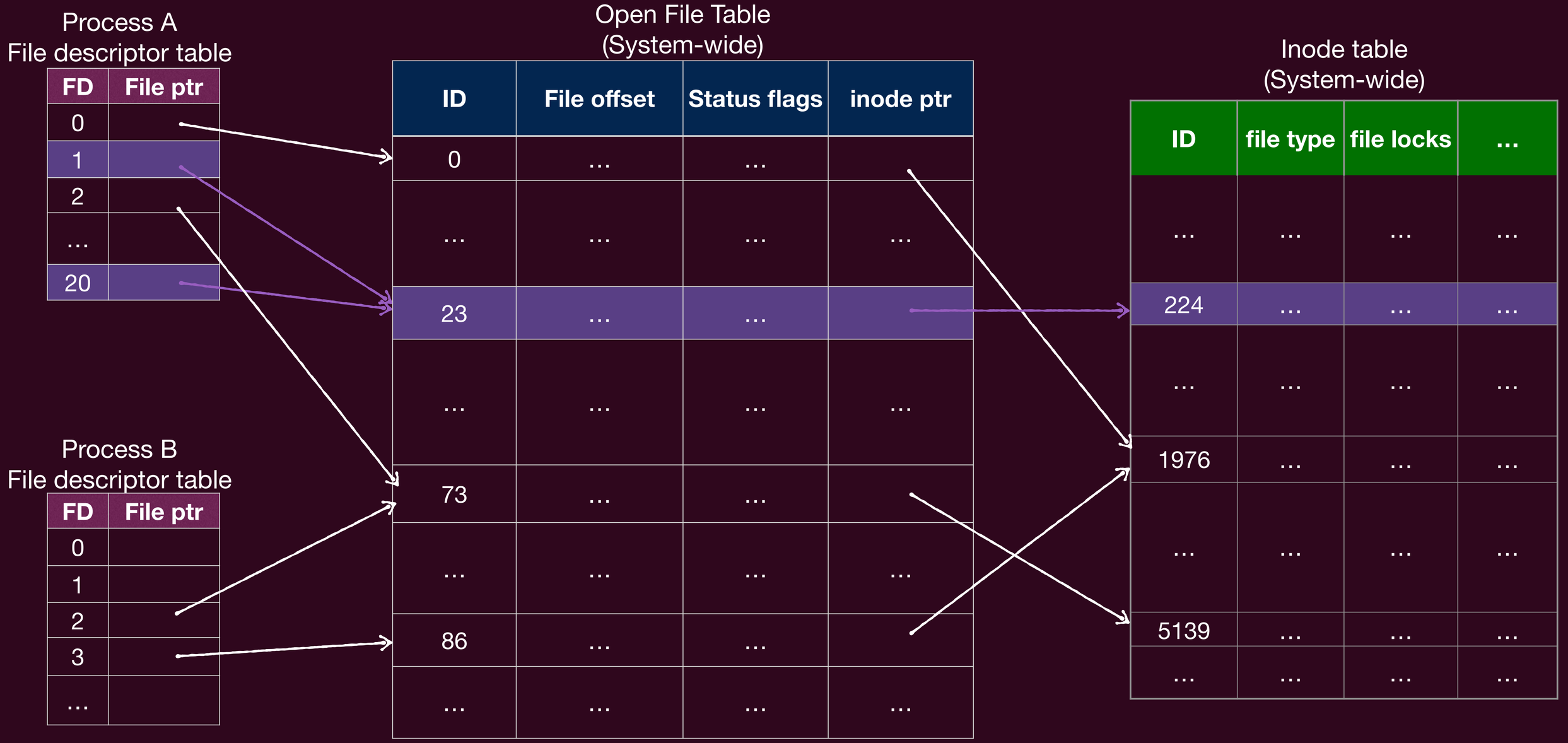
Inode table
(System-wide)

ID	file type	file locks	…
…	…	…	…
224	…	…	…
…	…	…	…
1976	…	…	…
…	…	…	…
5139	…	…	…
…	…	…	…

两个进程可以指向同一个全局的打开项（如Fork）

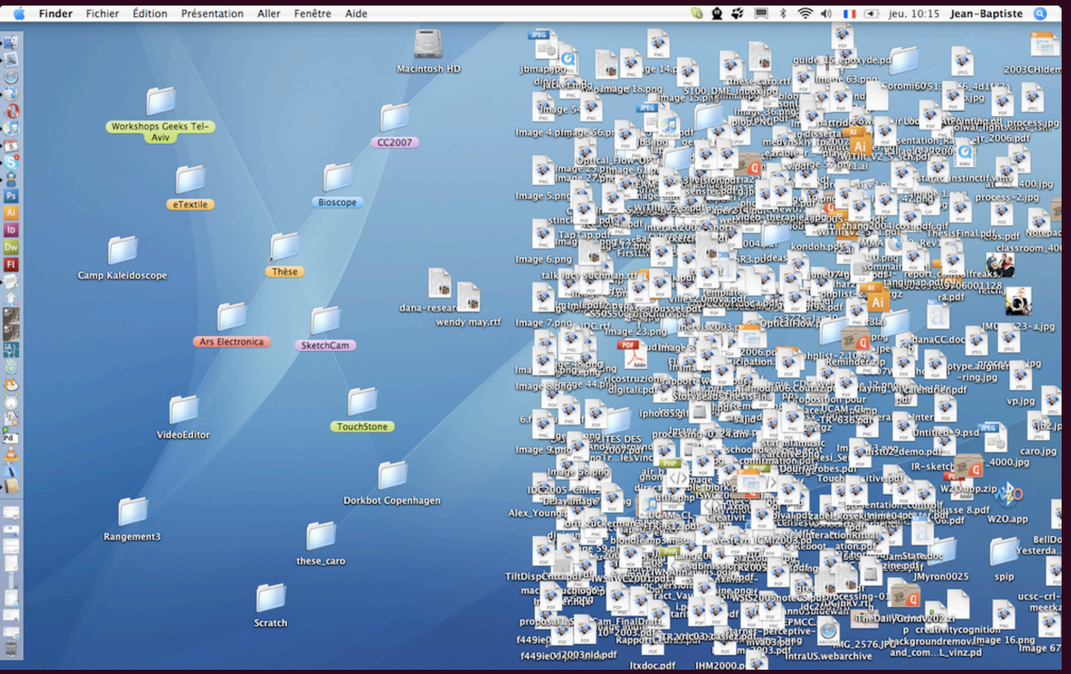


一个进程可以有多个指向同一个打开项的文件描述符（Dup）

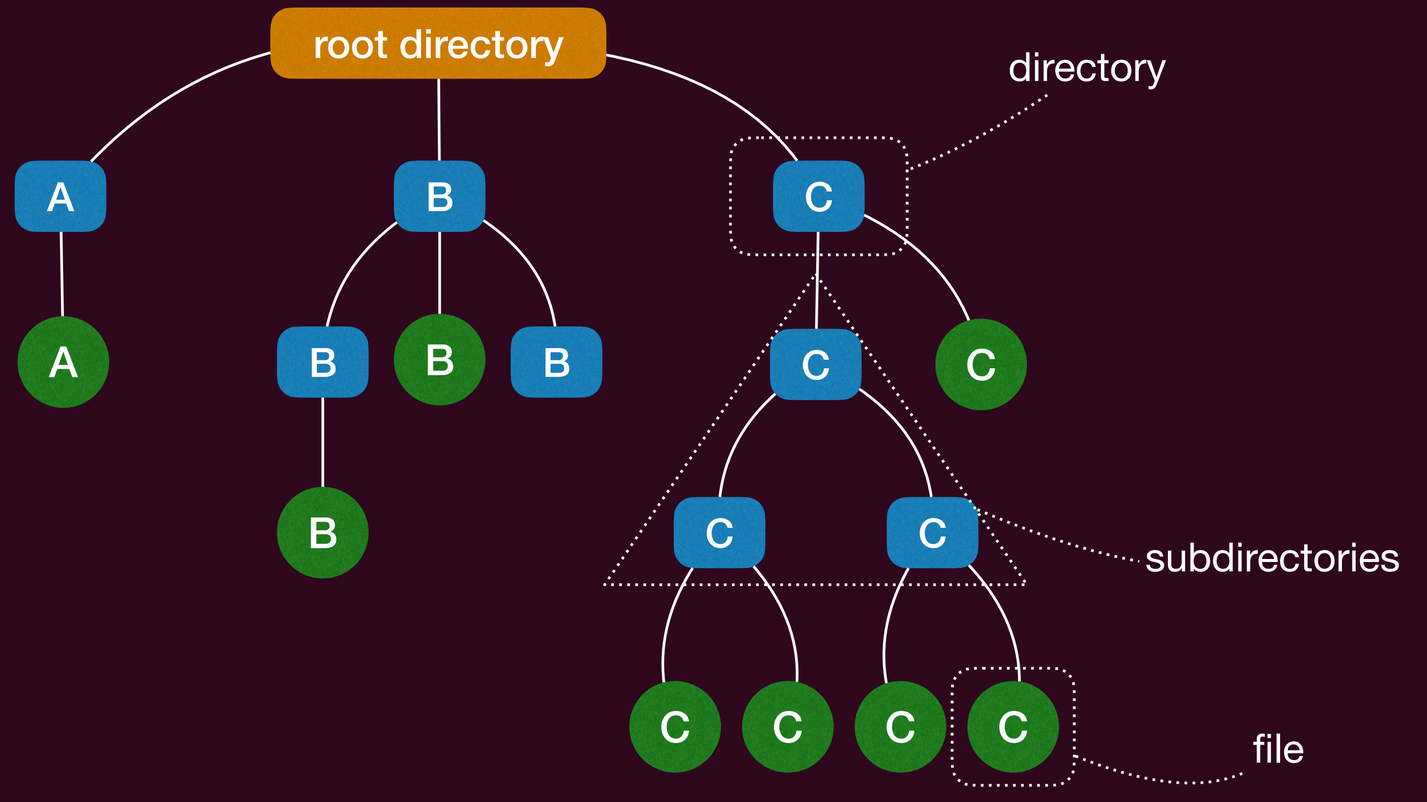


目录（Directory）

- 文件系统通常有目录来跟踪文件
  - 通过将目录放在其他目录内，用户可以构建任意的目录树（或目录层次结构）

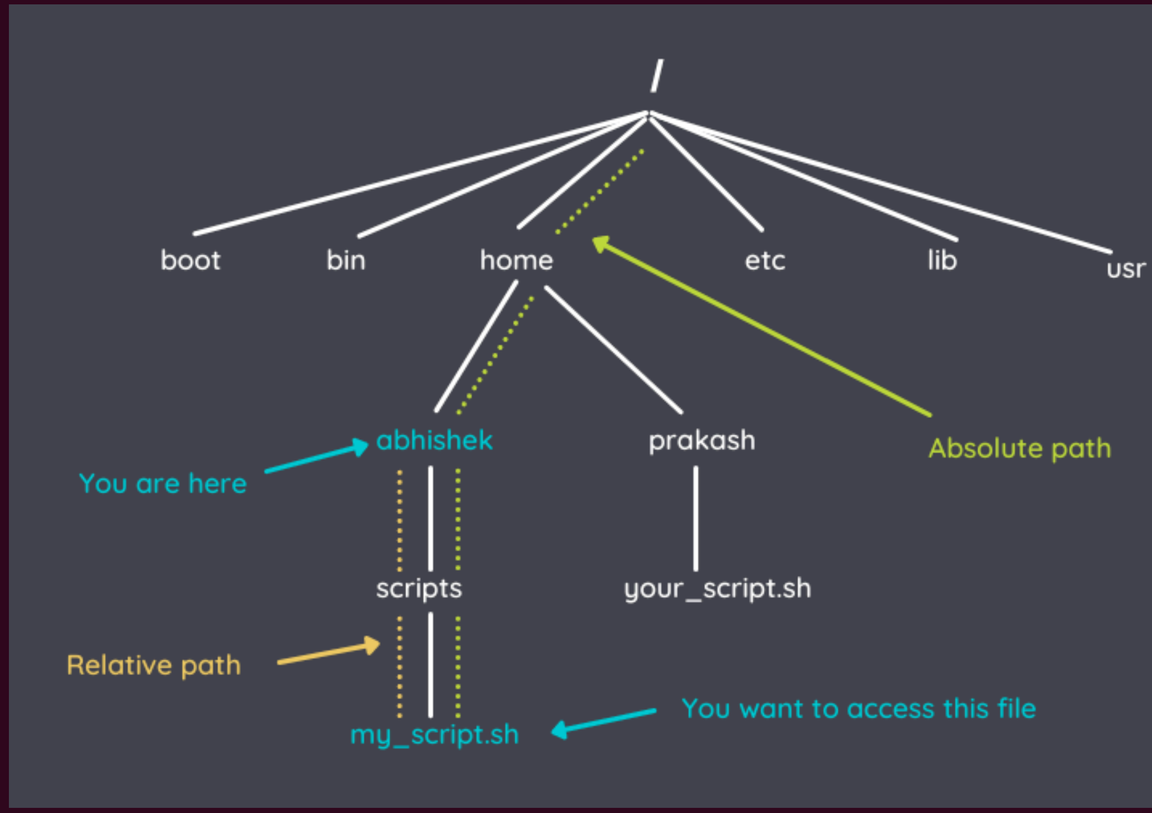


没有目录的话....



目录

- 标识文件或目录的字符串称为路径
  - **绝对路径**：从根目录到文件的路径（使用某种分隔符）
  - **相对路径**：为每个进程提供一个当前（工作）目录
    - 在 Unix 中，每个目录中有两个特殊条目：“.” 和 “..”分别代表当前目录和上一层目录




```
ls -l /home/abhishek/scripts/my_script.sh
```

```
ls -l scripts/my_script.sh
```

```
cd ../prakash
```

目录

- 目录存储了文件名与低级别结构（文件控制块inode）之间的映射
  - 在 Unix 中，每个目录条目只是一个〈文件名，inode 号〉对
  - 目录被存储为一个文件
  - 要查找一个文件，需要找到包含该映射的目录
  - 根目录是特别的：需要为根目录分配一个固定的 inode 号

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

目录

- 寻找 /usr/ast/mbox 的步骤

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

从根目录找到 usr
得到其inode为6

Inode 6 is for /usr
mode
size
time
132
...

inode 6找到/usr 在
磁盘的132块

Block 132 is /usr directory	
6	.
1	..
19	dic
30	erik
51	jim
26	ast
45	bal

找到/usr/ast 的inode为26

Inode 26 is for /usr/ast
mode
size
time
406
...

inode 6找到/usr/ast 在
磁盘的406块

Block 406 is /usr/ast directory	
26	.
6	..
64	grants
92	books
60	mbox
81	minix
17	src

找到/usr/ast/mbox 的inode为60

目录的操作

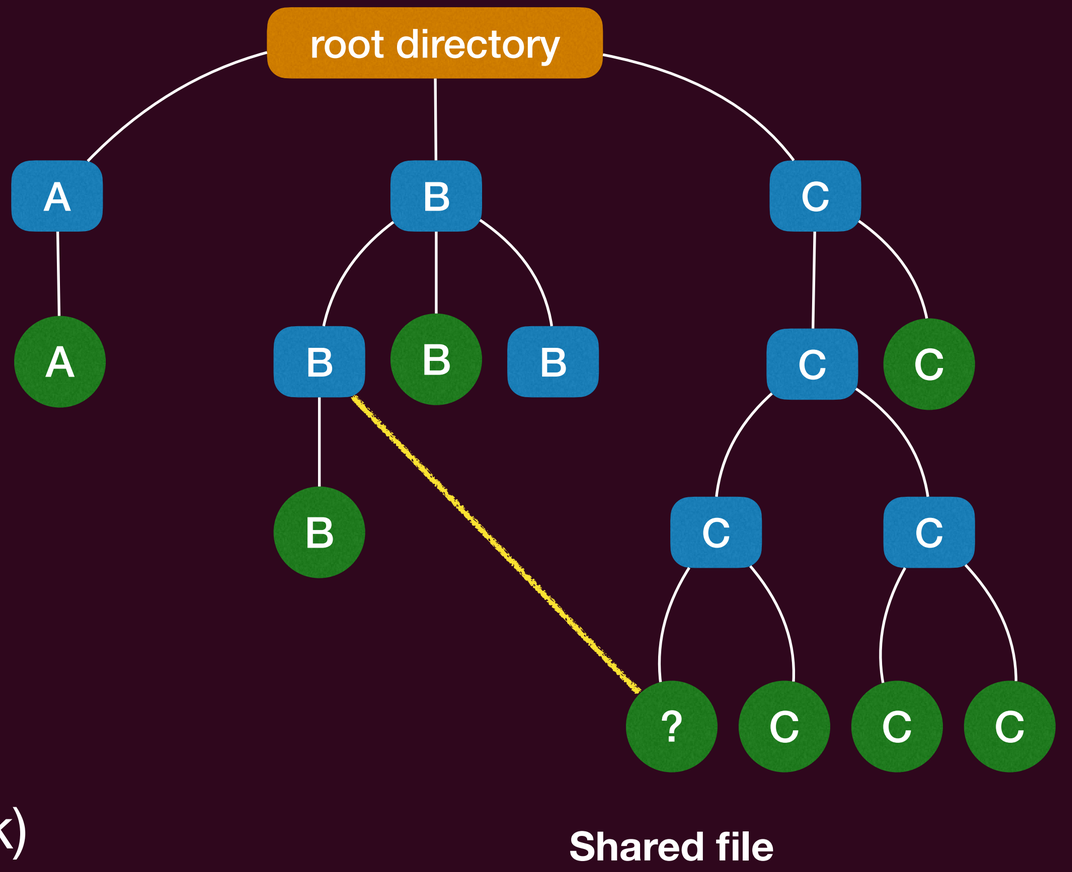
- `create`：创建一个目录（除了 . 和 .. 之外是空的）
- `delete`：删除一个目录（仅在目录为空时）
- `opendir`：可以读取目录（例如，列出所有文件）
- `closedir`：释放内部表空间
- `readdir`：返回打开目录中的下一个条目
- `rename`：更改目录的名称
- `link`：从现有文件创建一个链接(硬链接)到路径名（允许文件出现在多个目录中）
- `unlink`：删除一个目录条目（在 Unix 中删除文件）

目录的操作

- 提供的具体操作随系统而异
  - 目录的格式通常被视为文件系统的元数据，文件系统认为自己有责任维护目录数据的完整性
  - 因此，用户只能通过在目录中创建文件、目录或其他对象类型来间接更新目录

共享文件

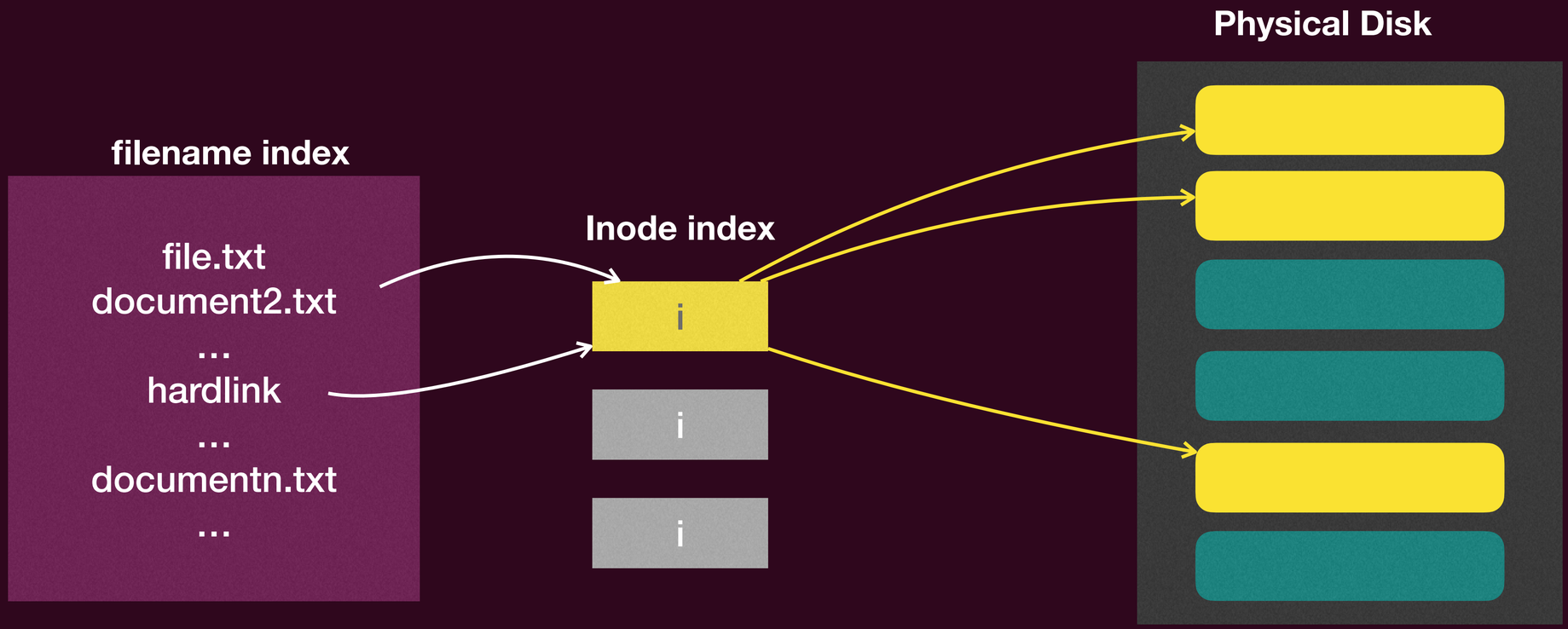
- 通过将一个新文件名链接到一个旧文件名，我们可以创建另一种引用同一文件的方式
  - 可以为同一个文件创建多个不同的名称
  - 目录结构变成一个有向无环图
  - 有两种链接
    - 硬链接（Hard link）
    - 符号链接（Symbolic link）或 软链接(Soft link)



Shared file

硬链接（Hard Link）

- 硬链接：在目录中创建另一个名称，并将其指向原始文件的相同 inode 号
  - linux下：`ln file linkname`



文件没有被复制；只是有两个名称（两个绝对路径）都指向同一个文件

硬链接

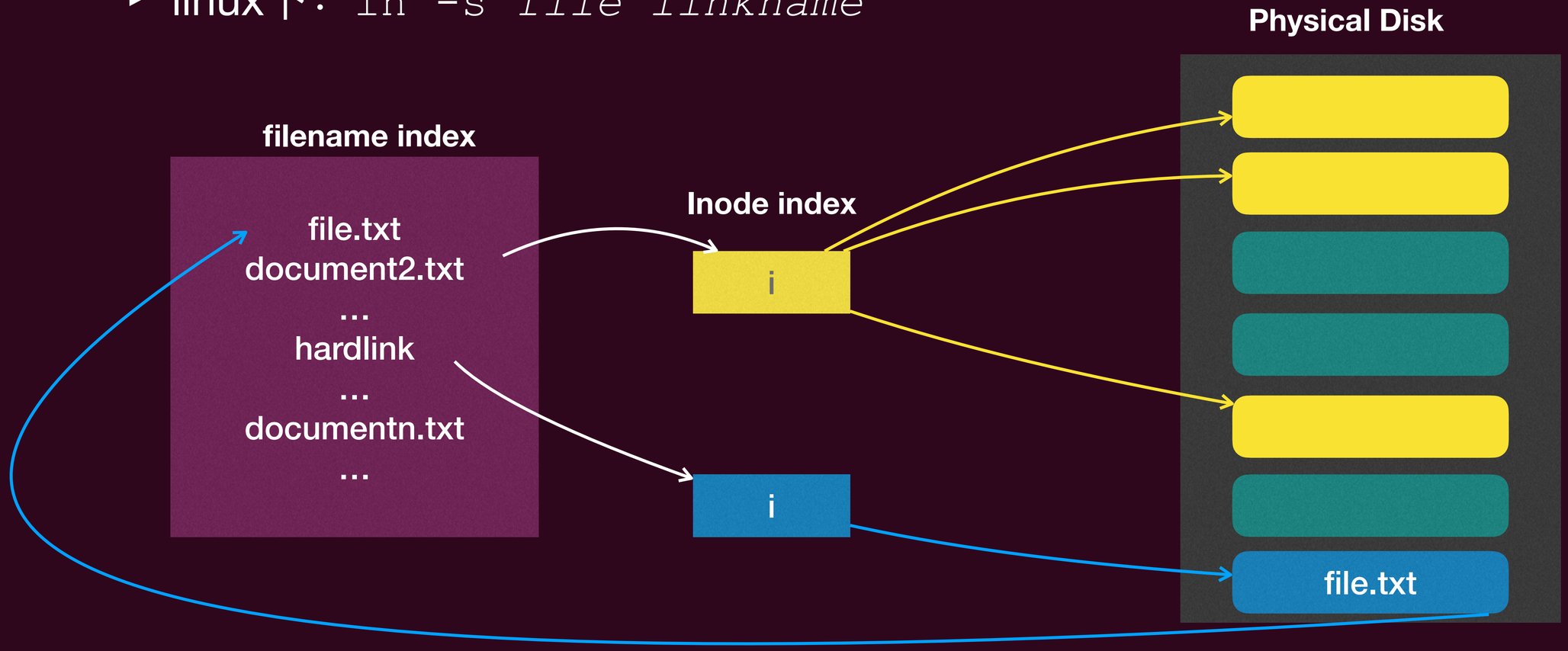
- 硬链接本质上是 inode 号的别名
- 因此，当创建一个文件时，
  - 首先，创建一个结构（inode），该结构将跟踪关于文件的所有相关信息
  - 其次，将一个人类可读的名称链接到该文件，并将该链接放入目录中
- 要从文件系统中删除文件，调用 unlink()
  - 每个 inode 有一个引用计数（链接计数器）
  - 删除链接时，引用计数减一；如果计数达到 0，目标文件将被删除

硬链接

- 但不能链接到另一个文件系统上的文件
  - 因为 inode 号只在一个文件系统内是唯一的
- 不允许链接到目录（简化管理）
  - 这防止了在目录层次结构中创建循环
  - 避免了父目录的不明确性
    - 比如如果多个父目录有指向同一子目录的链接，那么该目录中的 “..” 指向什么?

软链接（Soft or Symbolic Link）

- 符号（软）链接：创建一种不同类型的文件（链接类型）
  - linux下：`ln -s file linkname`



Holding the pathname of the linked-to file as the data of the link file

软链接

- 符号（软）链接是路径名的别名
  - 可以链接到目录，或跨文件系统链接
- 当需要解析路径名时符号链接被解析
  - 找到目标文件的名称，并使用新名称打开
  - 目标可以是另一个符号链接（递归解析）
  - 比硬链接效率低
- 当删除符号链接时，目标文件保持不变
  - 当目标文件被删除时，产生引用悬空（ dangling reference）
    - 该链接指向一个已不存在的路径名

文件保护

- 文件保护是为了防止意外和恶意破坏行为
- 文件所有者应该能够控制
  - 可以做什么?
  - 谁可以做?
- 访问权限的类型
  - 对于文件：read / write / execute
  - 对于目录：list / modify / delete
  - 对于访问权限本身：更改访问权限 / 给予某人访问权限 / 撤销某人的访问权限

文件保护

- 访问控制矩阵（Access Control Matrix）：系统访问控制的实现可以视为基于一个巨大表格，该表格编码了系统中每个用户或者用户组的所有访问权限

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

文件保护

- Unix中的访问控制：为每个用户分配（用户ID，组ID）相应的权限
  - ID与该用户发出的每个操作请求相关联
  - 3个用户类别：所有者、组、其他人
  - 3种访问模式：读取、写入、执行（编码为3位）

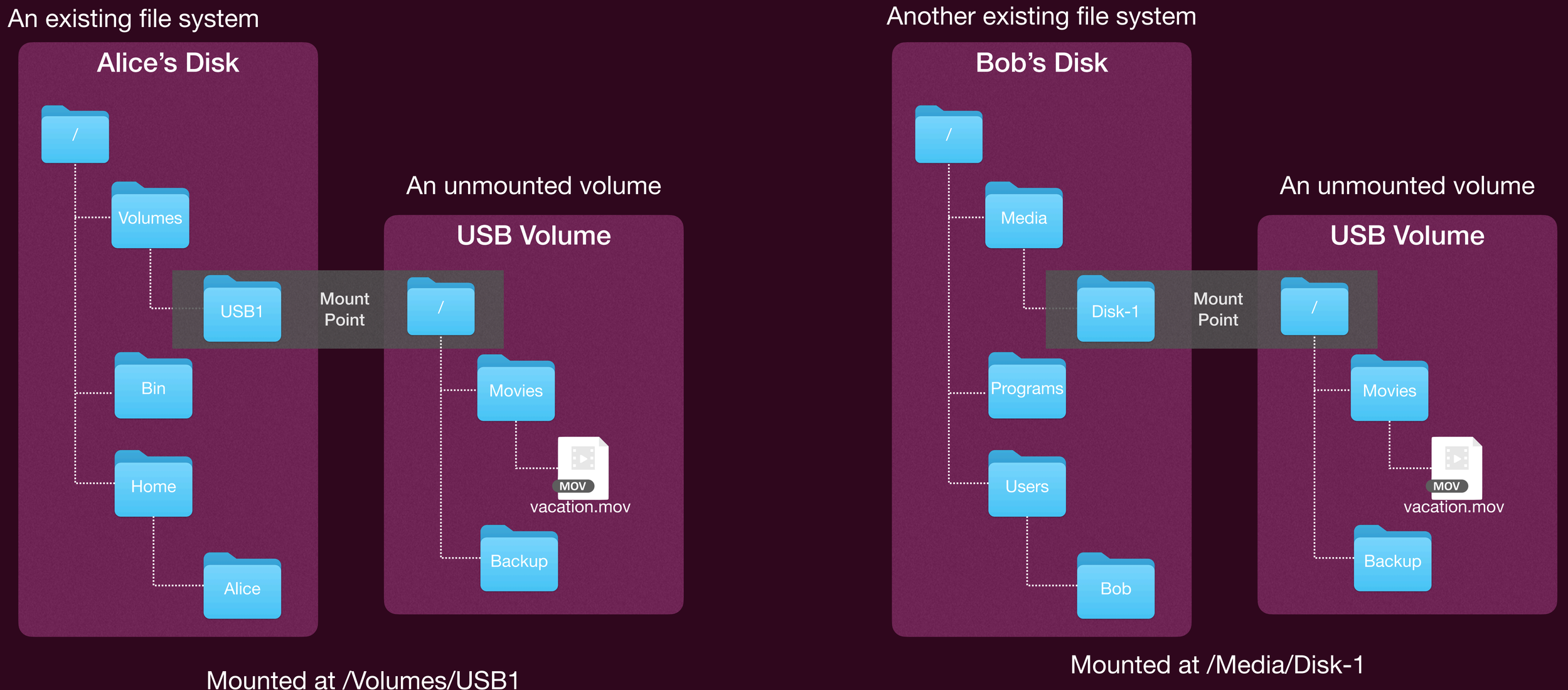
例：

		RWX
owner access:	7	1 1 1
group access:	6	1 1 0
other access	1	0 0 1

文件系统挂载（Mount）

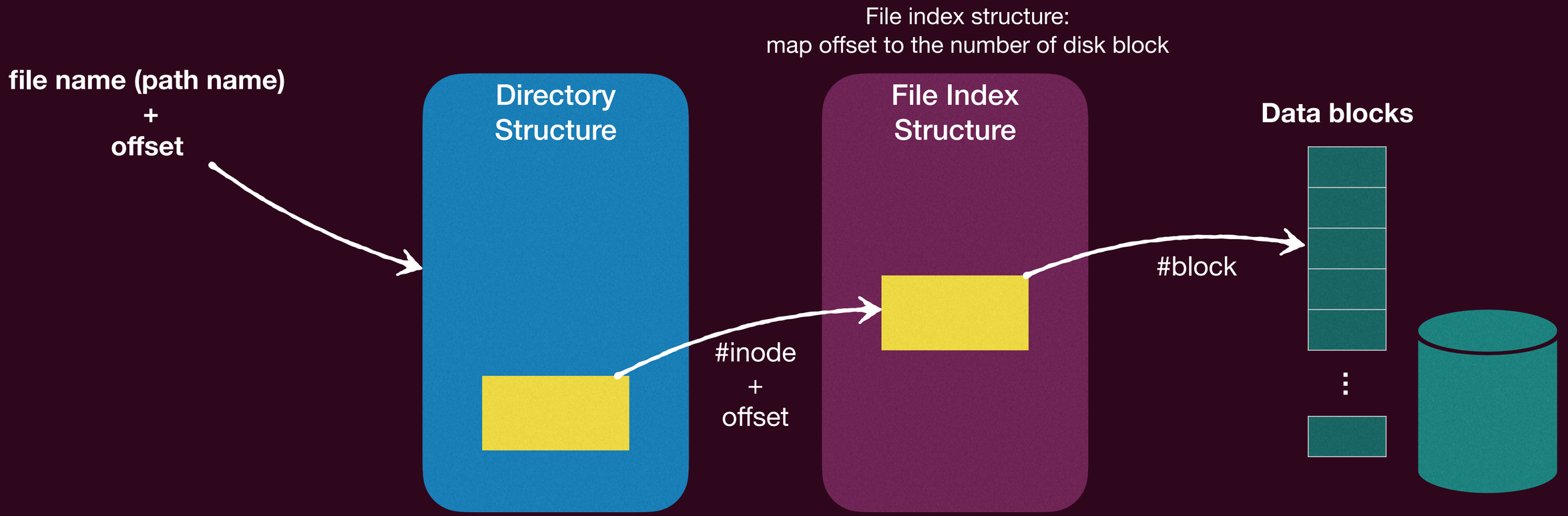
- 一个文件系统在能被访问之前，必须先进行挂载
  - 从现有文件系统中的某个路径（挂载点）创建到挂载文件系统的根目录的映射
    - 将多个文件系统统一到一棵树中
  - 在 Linux 上，mkfs 命令可以在块设备上创建一个新的文件系统，mount 命令可以在当前文件系统中的某个目录下挂载一个文件系统。没有任何参数时，mount命令会显示当前的挂载点.

文件系统挂载


An existing file system
Alice's Disk
/
Volumes
USB1
Bin
Home
Alice
An unmounted volume
USB Volume
Mount Point
/
Movies
MOV
vacation.mov
Backup
Mounted at /Volumes/USB1
Another existing file system
Bob's Disk
/
Media
Disk-1
Programs
Users
Bob
An unmounted volume
USB Volume
Mount Point
/
Movies
MOV
vacation.mov
Backup
Mounted at /Media/Disk-1

两个关键的抽象

- 目录结构：将文件路径名映射到相应的文件控制块



文件系统 & 虚拟内存

- 文件元信息
  - 文件到数据块的映射
  - 访问权限和非法地址检测
  - 完全的软件实现
- 页表
  - 地址空间到物理内存的映射
  - 访问权限和非法地址检测
  - 实现需要硬件的支持，如TLB

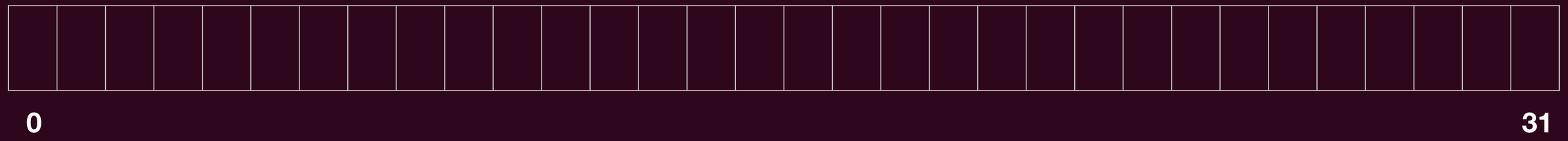
文件系统实现

文件系统实现

- 我们需要回答如下几个问题：
  - 文件和目录是怎么存储和组织的?
  - 磁盘空间如何管理?
  - 文件系统的实现是否高效和可靠?

文件系统的布局

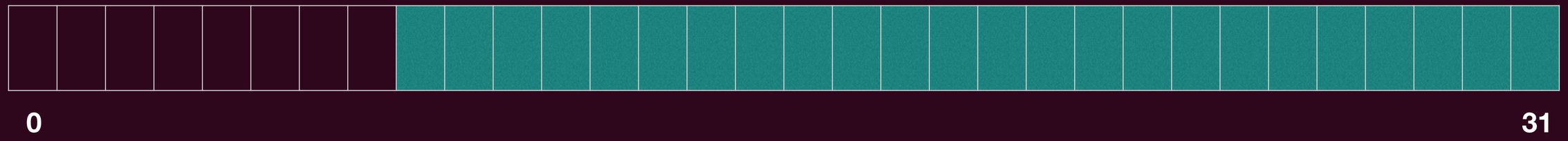
- 文件系统需要为需要存储的数据维护一个其在磁盘上的数据结构
- 磁盘被分割为等大小的数据块
  - 从0到N-1进行编号



A simplified version of a typical UNIX file system

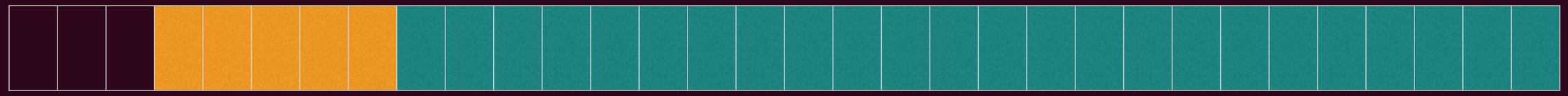
文件系统的布局

- 文件系统需要为需要存储的数据维护一个其在磁盘上的数据结构
  - 数据区域(数据块): 磁盘保留的一个固定数据块部分用来存储数据本身



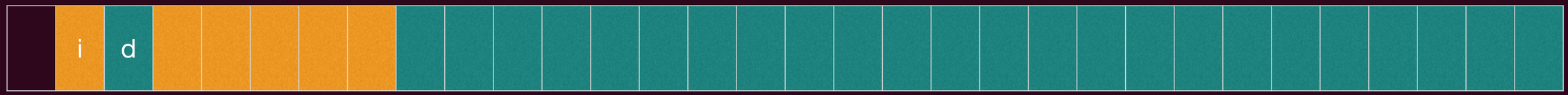
文件系统的布局

- 文件系统需要为需要存储的数据维护一个其在磁盘上的数据结构
  - inode表：每个文件的元信息
    - 文件对应哪些数据块、文件的大小，拥有者，时间戳，访问权限…
    - 一个数据块可以存储多个元信息（inodes），一个4KB的数据块可以包含16个256字节的inode
    - 一个inode在哪个数据块存储是容易查询的



文件系统的布局

- 文件系统需要为需要存储的数据维护一个其在磁盘上的数据结构
  - 分配数据结构: 对于已分配和空闲的空间的信息 (空闲空间分配)
    - 决定一个inode或一个数据块是否是空闲的
    - 有两个空闲数据结构（基于bitmap），一个为了inode，一个为了数据块

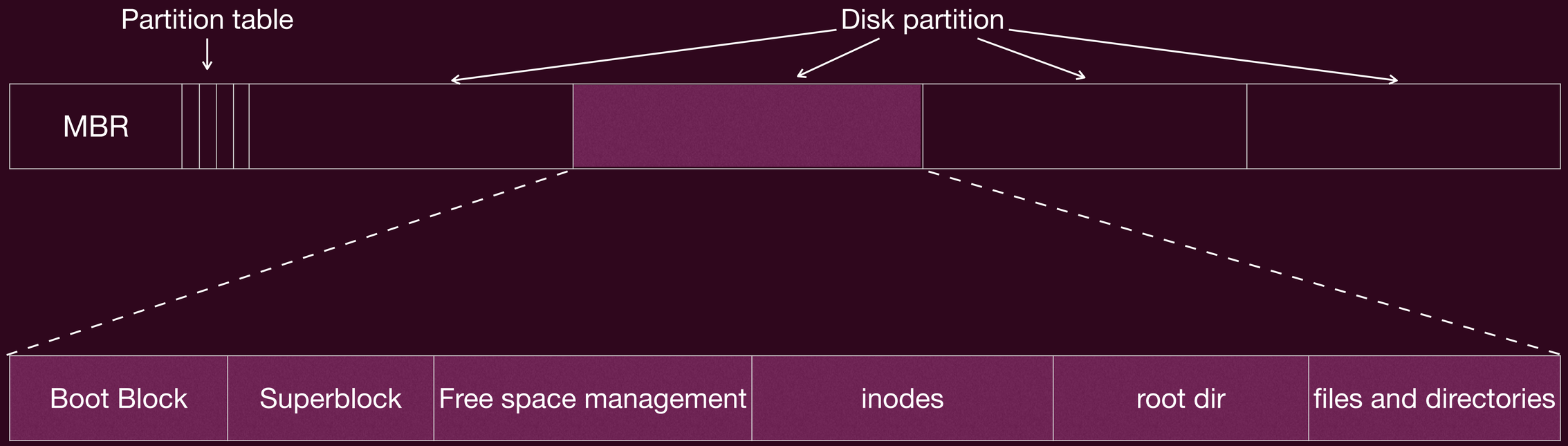


文件系统的布局

- 文件系统需要为需要存储的数据维护一个其在磁盘上的数据结构
  - 超级块（Superblock）：文件系统本身的信息
    - 文件系统的类型、inodes数量、数据块数量、inode表开始的块地址…
    - 当挂载一个文件系统时会读取超级块的信息

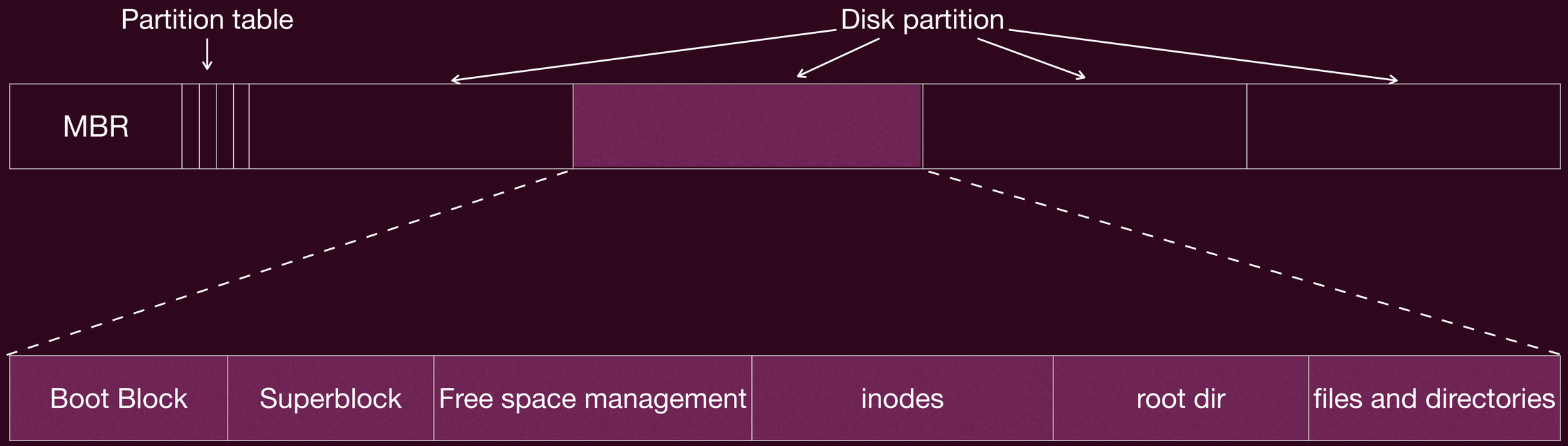
文件系统的布局

- 文件系统需要为需要存储的数据维护一个其在磁盘上的数据结构
  - 启动块：启动OS所需要的信息
    - 一般来说是一个分区的第一个数据块 (没有OS的话为空)



文件系统的布局

- 文件系统需要为需要存储的数据维护一个其在磁盘上的数据结构
  - 主引导记录，或主引导扇区 (Master Boot Record，MBR): 启动计算机所需的信息
  - 分区表（Partition Table）：每个分区的起始和结束地址（其中一个标记为活跃，就是启动OS的分区)
  - 每个分区是磁盘上连续的一块区域，可以格式化以存储文件系统（每个分区的文件系统类型可以不同），比如Linux三个不同格式的分区/boot, swap, /



文件系统的布局

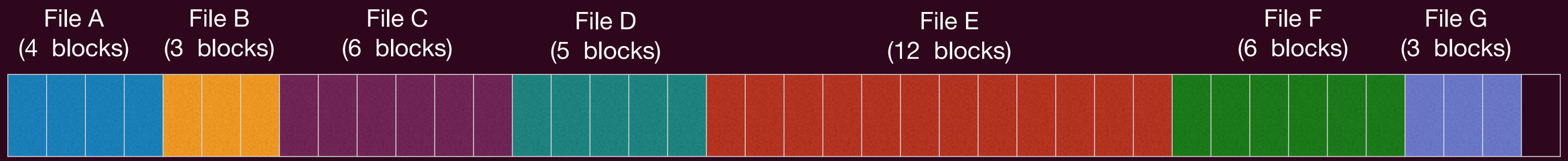
- 文件系统还需要在内存中维护相应的数据结构用来方便的对磁盘数据进行访问（用来反映和拓展磁盘中的结构）
  - 挂载表（Mount table）:关于文件系统的挂载信息（挂载点、文件系统类型）
  - 打开文件表: 系统范围的和每个进程的
  - 目录结构：最近访问的目录信息
  - I/O内存缓冲: 读写磁盘时需要处理主存和磁盘之间速度差异的“缓冲带”

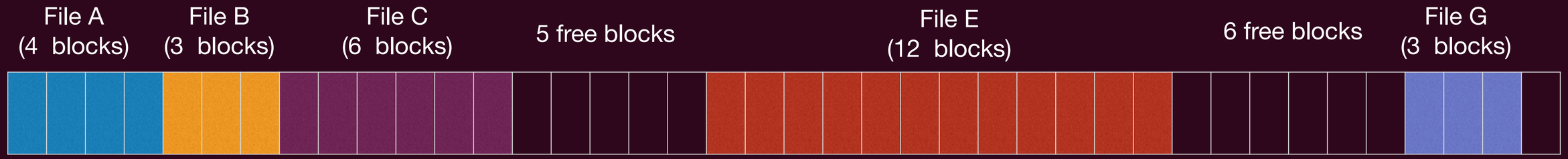
文件组织

- 文件需要磁盘给定相应的数据块进行存储，此外还需要有一些数据结构来反映该文件用的数据块在哪
- 很多种组织方式
  - 连续存储（Contiguous）
  - 链表（Linked List）
  - 文件分配表（File Allocation Table）
  - 索引式配置（Indexed Allocation）

连续分配

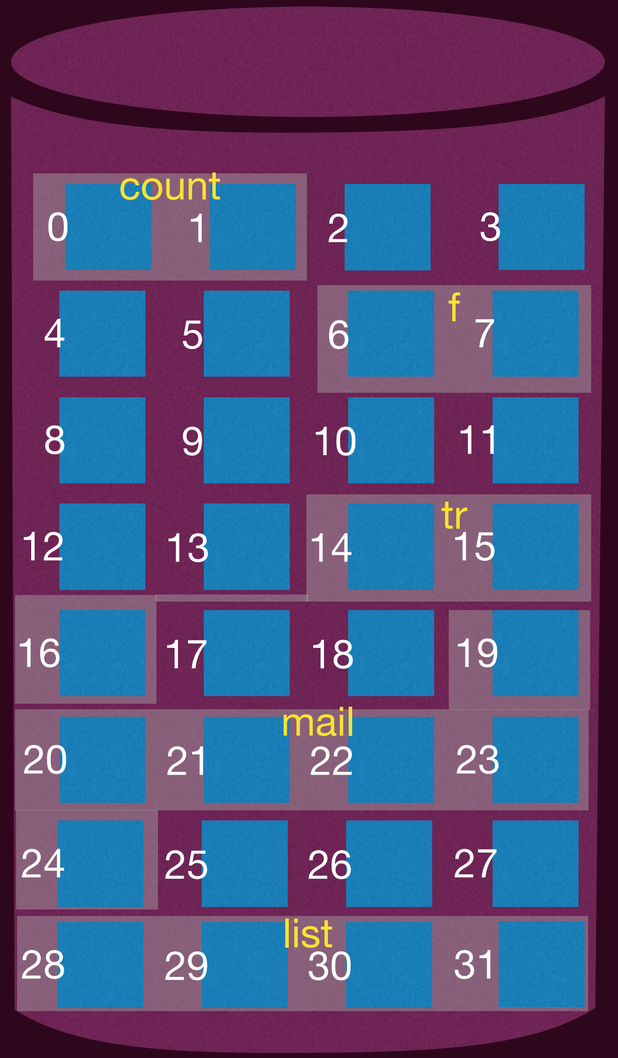
- 每个文件占据一个连续的数据块集合
  - 比如对于磁盘数据块大小为1KB的情况，一个20KB的文件就是占据了20个连续的数据块





连续分配

- 在连续分配下：
  - 只有文件的第一个数据块的地址和需要的数据块总数需要记录
  - 线性的访问是高效的
  - 随机访问的数据地址也是容易计算的



Directory

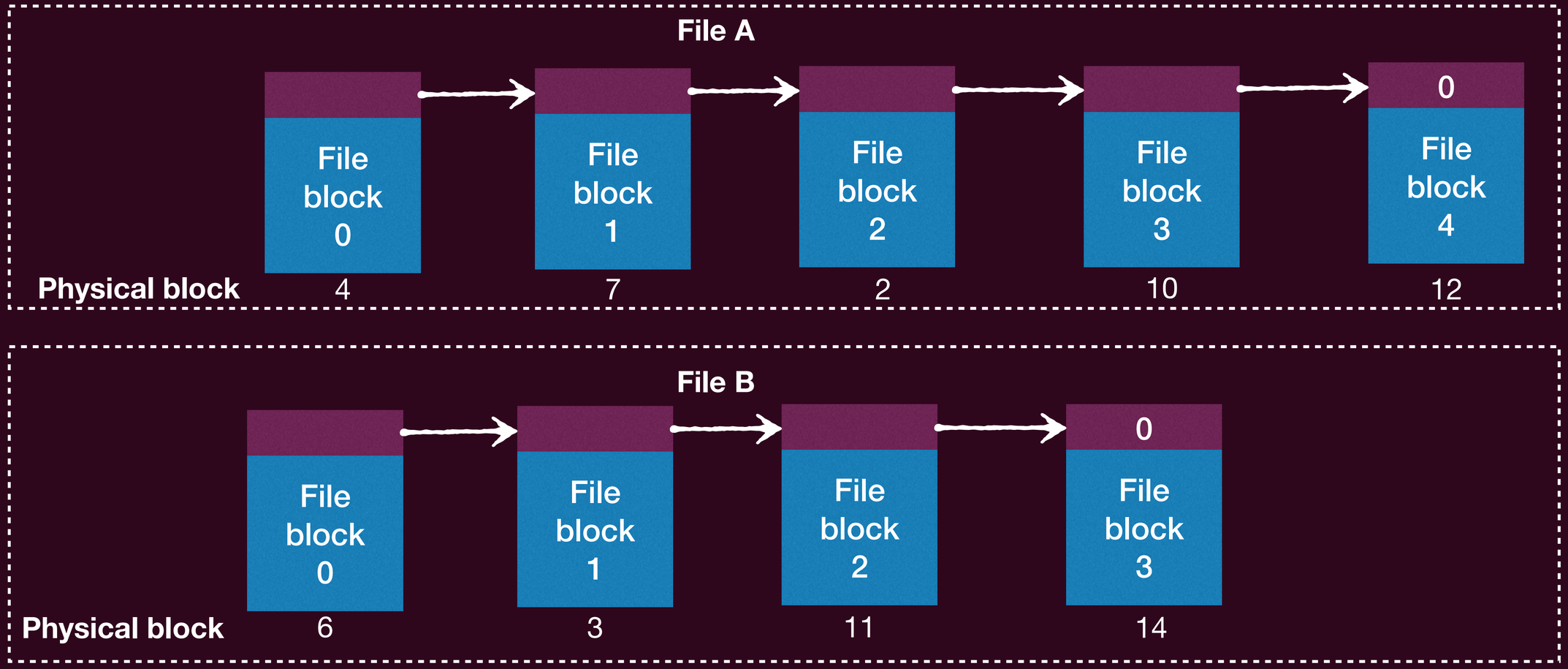
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

连续分配

- 但连续分配不够灵活：
  - 在文件创建时就需要知道文件的大小，之后如何增加和减少文件的大小也是麻烦
  - 有外部碎片：需要进行收缩操作（compaction）来减少这种碎片
- 但对于CD-ROM和DVD这样的介质是好的（烧录之后不再改变）
- 有些文件系统使用连续分配的一种变体：基于extent的连续分配
  - extent是一个连续块集合
  - 一个文件包含了一系列的extents，extents之间不必相邻(以链表连接)
    - 可以支持文件的增长和减少
  - 比整体连续分配灵活一点，但还是存在碎片

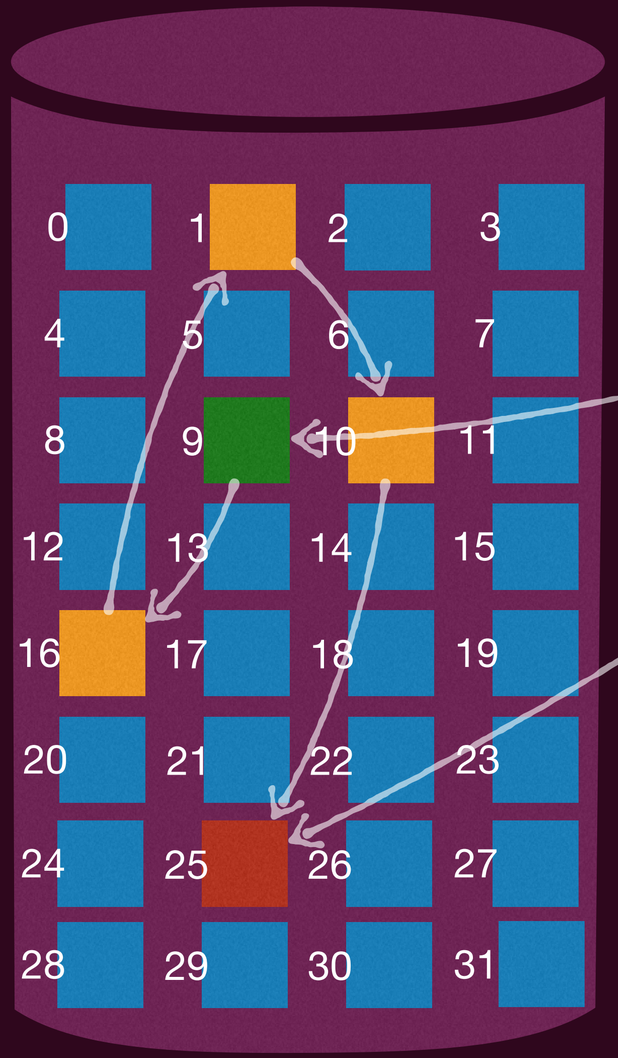
基于链表的分配

- 每个文件是一个数据块的链表
  - 每个数据块包含指向下一个块的指针



基于链表的分配

- 每个文件是一个数据块的链表
  - 磁盘上的每个数据块都可以被使用
    - 没有外部碎片
  - 仅仅需要存储第一个块的地址
  - 然而，
    - 随机读取会产生很多磁盘I/O
    - 每个数据块都需要为指针留取空间
    - 可靠性问题：如果某个指针损坏…

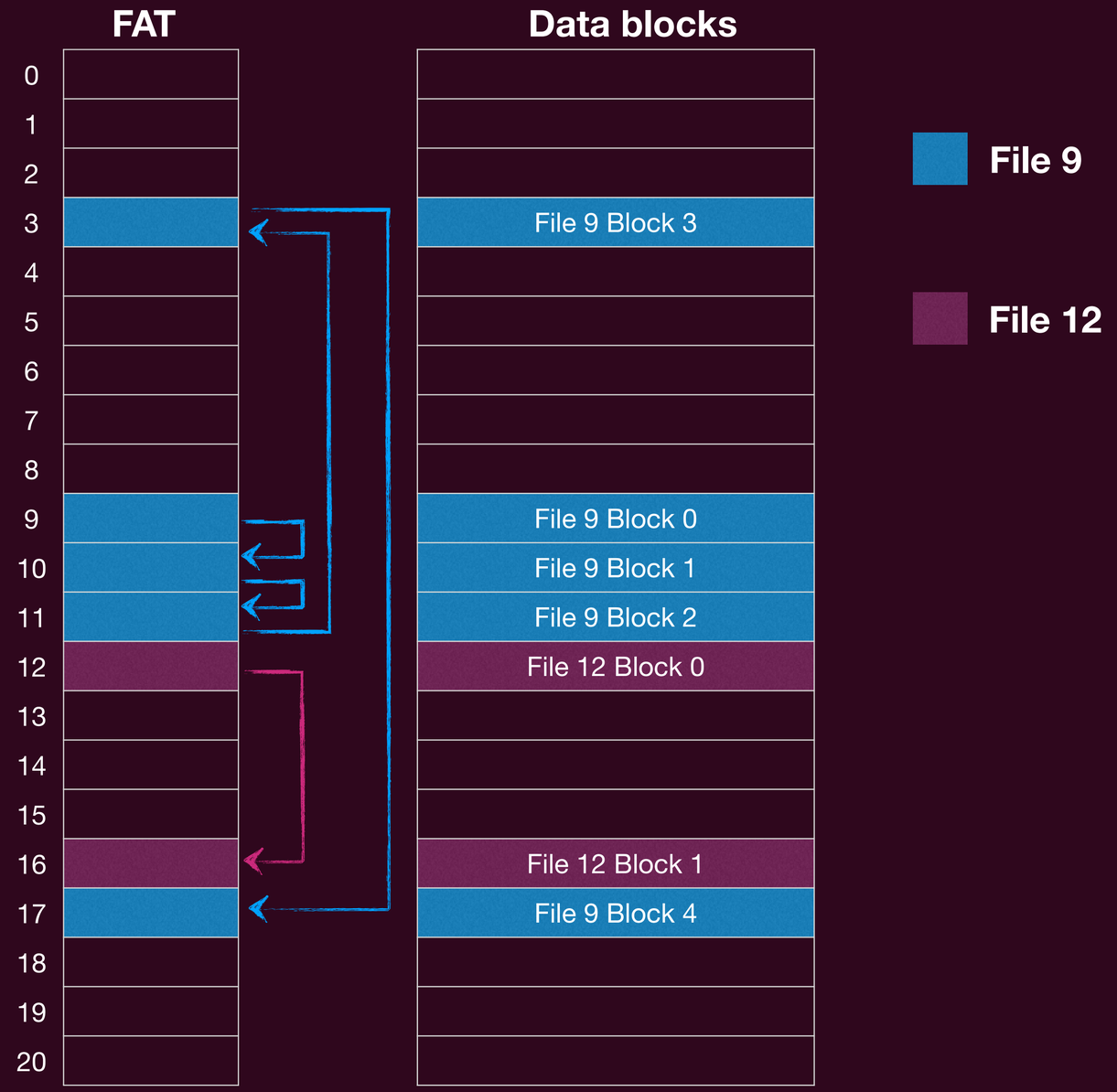


Directory

file	start	end
jeep	9	25

文件分配表 (File Allocation Table,FAT)

- 文件分配表（FAT）：一个基于链表分配的变种，其将所有的指针放到同一个表格中
  - 对于每个磁盘数据块都有一个FAT的项
  - 每个FAT项包含一个指向下一个FAT项的指针，或者文件终止符号
  - 对于随机访问的操作，只需要访问该FAT表即可，该表可以直接存储进主存从而减少需要I/O的次数

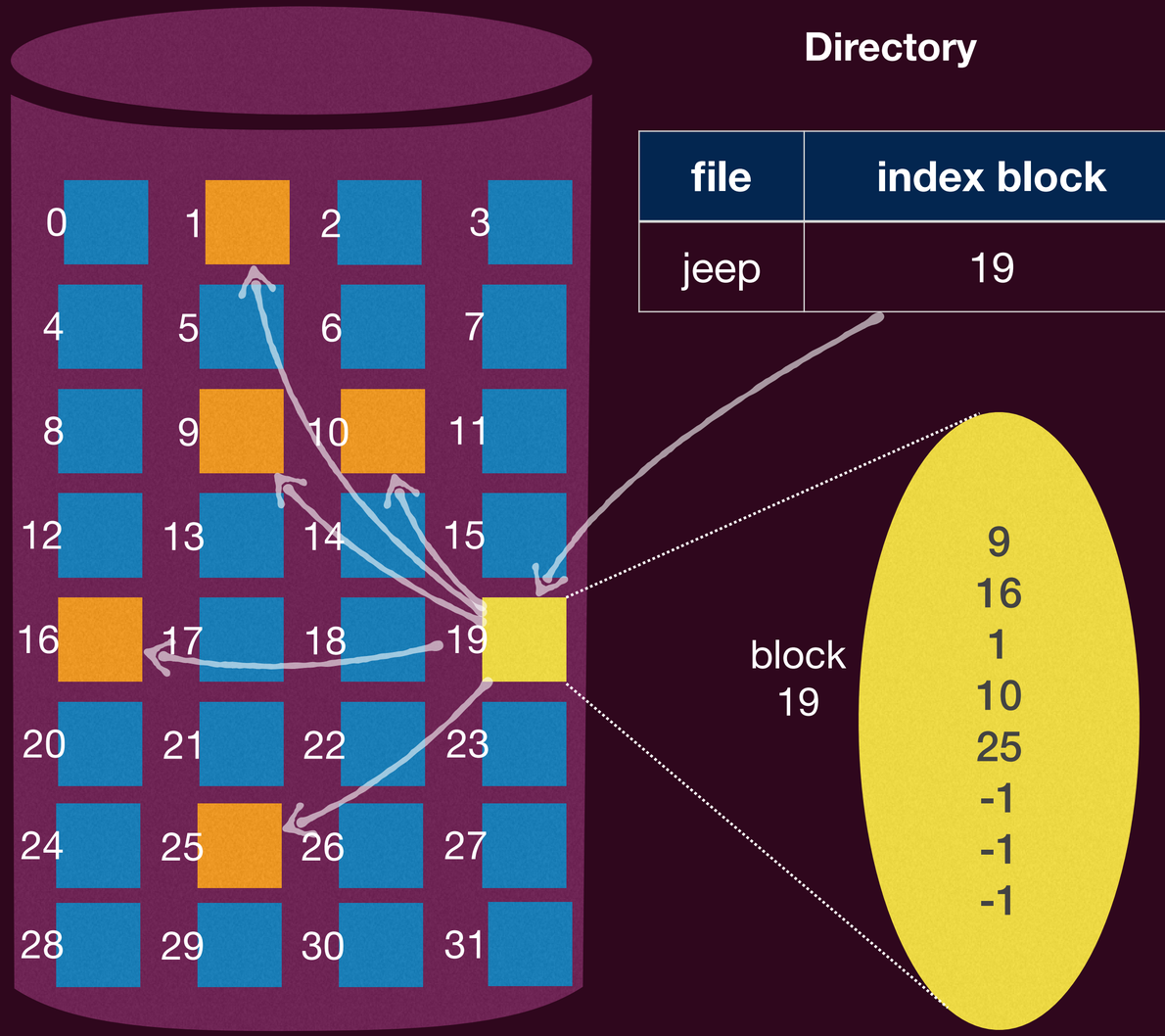


文件分配表

- FAT文件系统布局
  - FAT表还可以用来进行空闲空间追踪
  - 可以使用两个FAT表来增加可靠性
- 但FAT需要存储在主存中，这会带来巨大的开销
  - 对于容量为一个1TB (2^{40}) 的和块大小为4KB (2^{12}) 的磁盘来说, 需要2^{28}的表项;
    - 如果每个表项需要4个字节，那么一个FAT表格就需要1GB

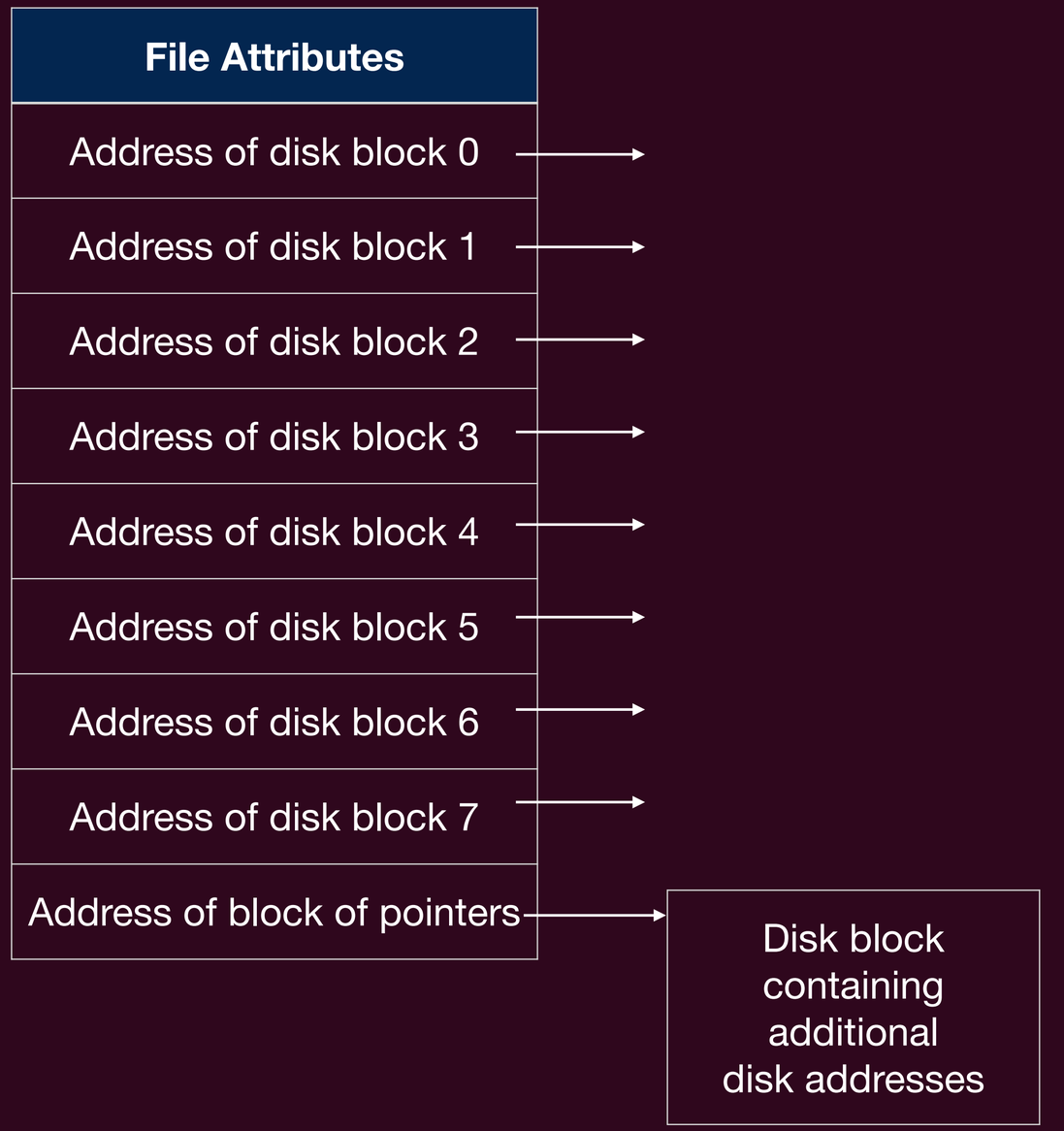
索引式分配（Indexed Allocation）

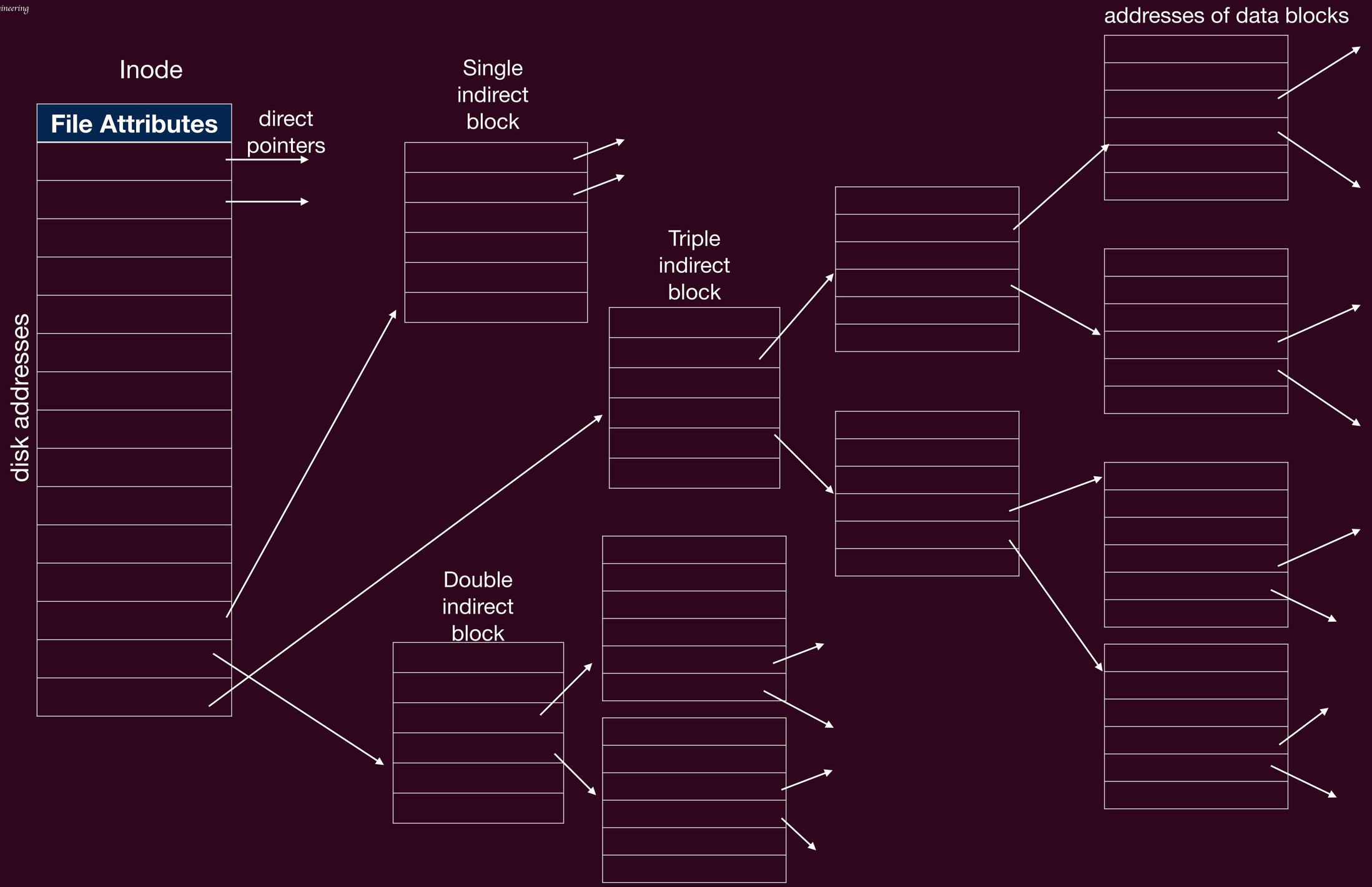
- 每个文件有一个对应的指针数组块（索引数据块），其中的每个指针指向该文件的一个数据块
  - 第i个指针指向文件的第i个磁盘数据块
  - 当文件打开时，该索引块才会被加载进内存
  - 因此内存中的开销只和打开文件数相关，而不是整个磁盘
  - 但有一个问题：每个文件所需要的指针个数不同（占据的数据块的个数不同），从而导致索引块大小难以确定



多级索引

- 使用间接指针：可以将索引块中的指针指向一一个由指针组成的数据块，其中每个指针再指向数据块
  - 一个索引块中可以赋予固定数量的直接指针（指向数据块）和固定数量的间接指针
  - 此时索引的结构变成了一个树（非平衡）
  - 对于小文件和大文件都能很好支持
  - 一般有2级和3级的非间接指针



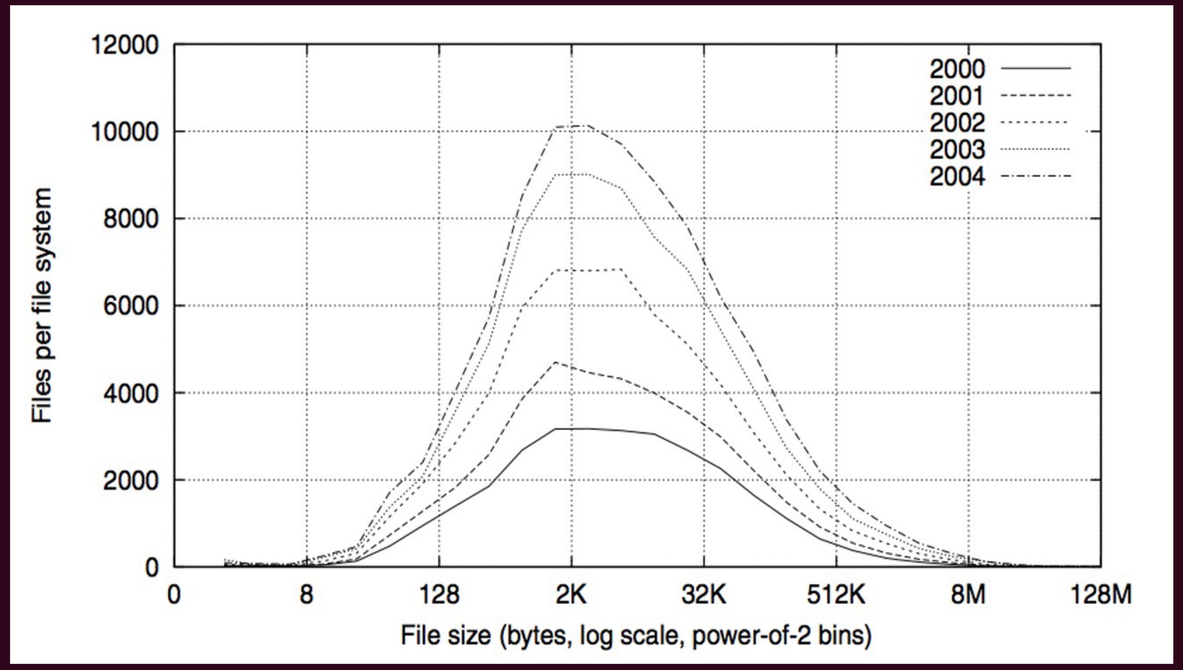

addresses of data blocks
Inode
File Attributes
direct pointers
Single indirect block
Triple indirect block
Double indirect block
disk addresses

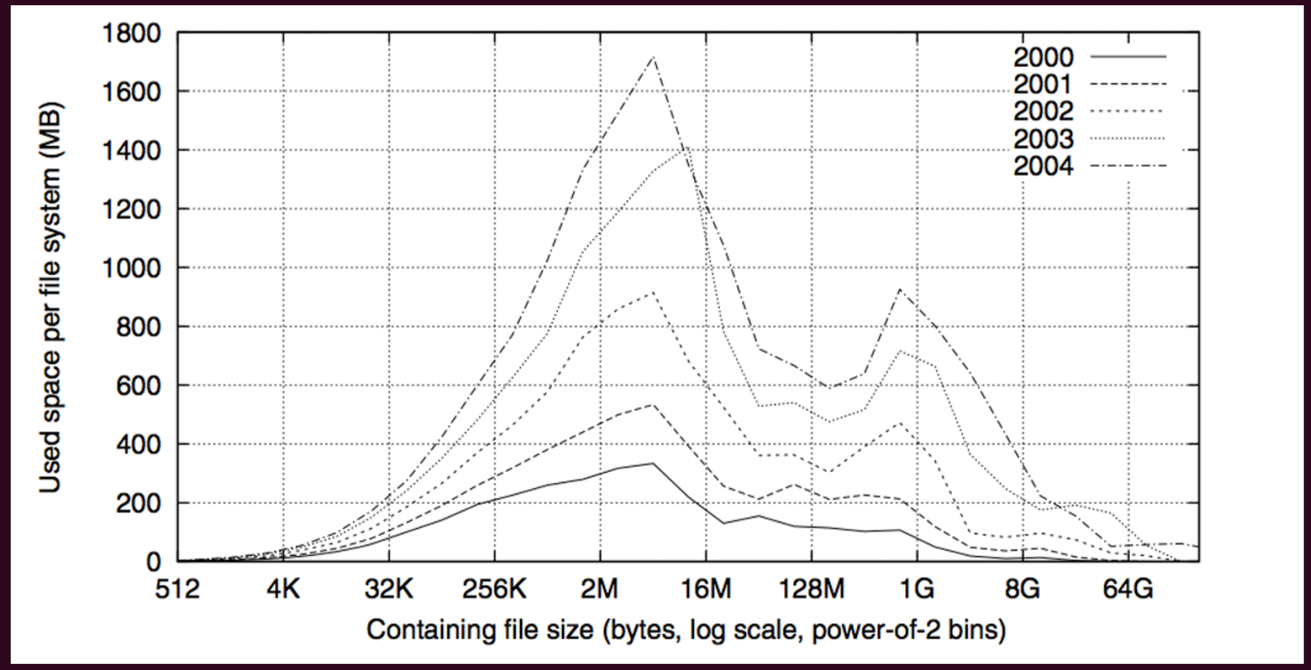
例子

- 假定数据块为4KB，一个表项为4字节
- inode中的索引数据结构包含15个指针
  - 开始的12指针指向数据块 (12 * 4 KB = 48KB)
  - 最后3个指向间接数据块
    - #13: 单级间接指针 (2^{10} * 4 KB = 4MB)
    - #14: 双级间接指针 (2^{10} * 2^{10} * 4 KB = 4GB)
    - #15: 三级间接指针 (2^{10} * 2^{10} * 2^{10} * 4 KB = 4TB)

文件组织

- 现实中的文件系统的特性： A five-year study of file system metadata [FAST 2007]
  - 大部分文件很小: ~2K 是最为常见的文件大小
  - 大部分的数据字节存储在大文件当中





文件组织

- 好的文件组织需要考虑现实中文件系统的特性
- 几个关键属性：
  - 是否能够支持快速的顺序和随机访问
  - 内部和外部碎片
  - 实现的难易程度
  - 文件大小增长的支持
  - 存储的开销
  - …

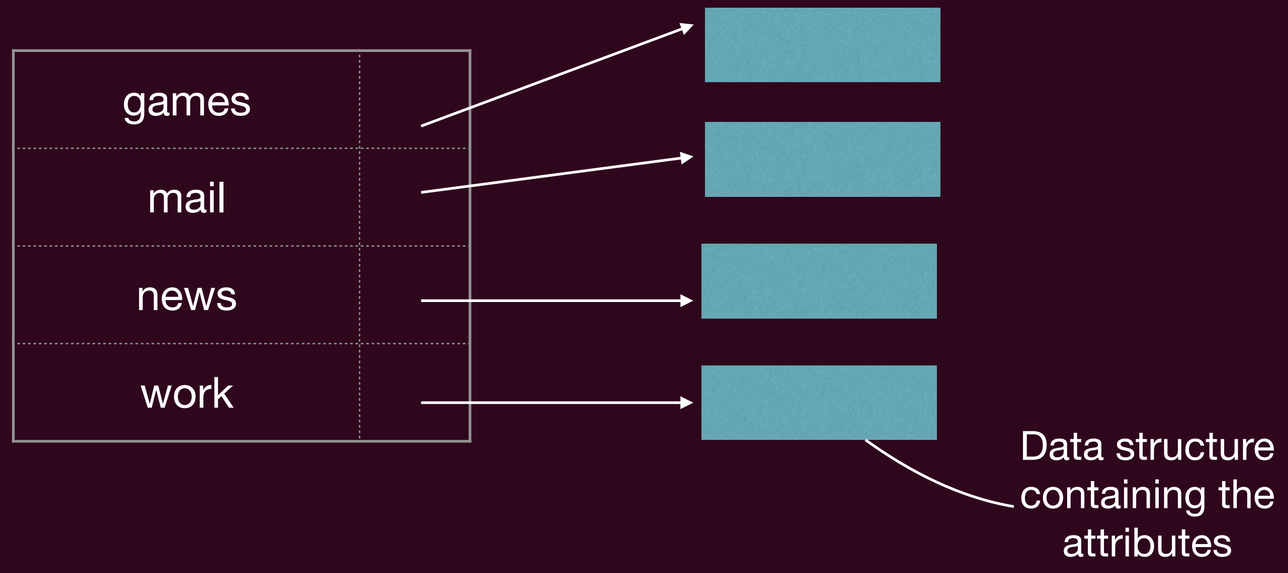
目录组织

- 目录提供了找到文件名和其在磁盘上的数据块的映射信息〈文件名, 文件索引〉
  - 当需要打开一个文件时，OS首先找到路径名，并根据路径名找到磁盘上的目录项
  - 目录项提供了该文件对应的磁盘数据块，可以是
    - 整个文件所在的磁盘地址
    - 第一个数据块的编号
    - 文件的元信息inode号

目录组织

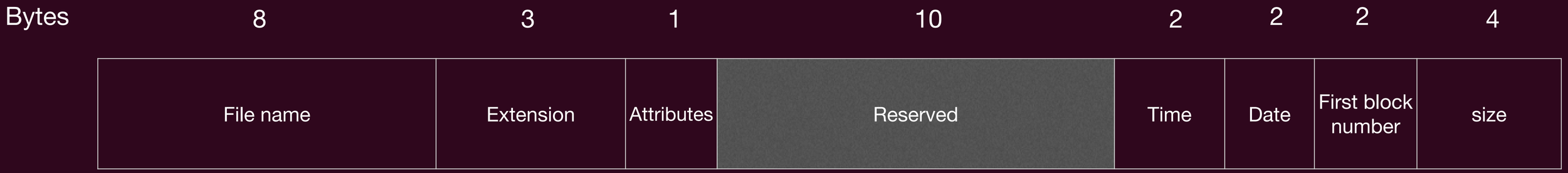
- 文件的信息可以直接存储到目录的一项当中
- 目录的一项也可以只存储指向该文件元信息inode的指针

games	attributes
mail	attributes
news	attributes
work	attributes



目录组织

- MS-DOS (FAT)的目录项



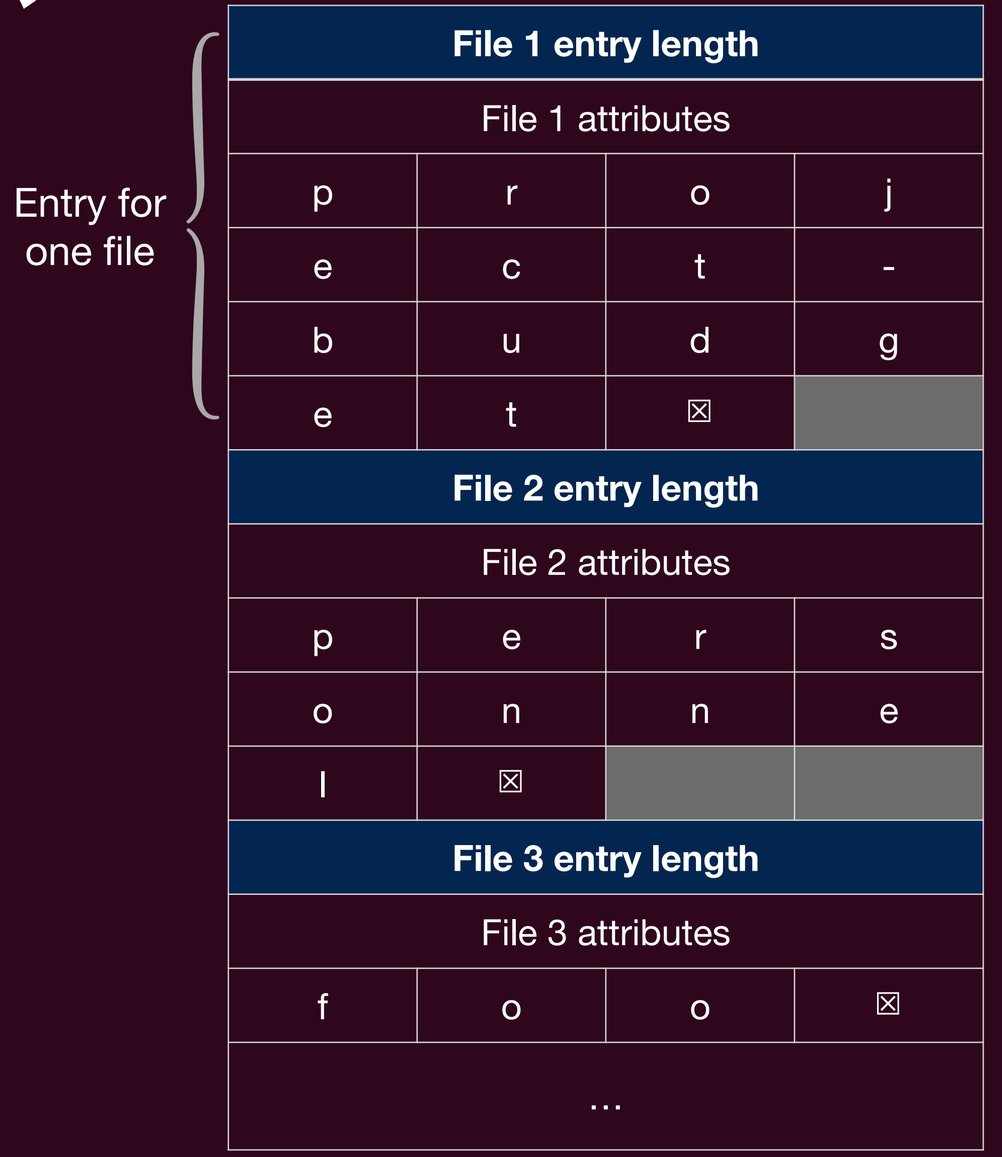
- Unix的目录项

目录组织

- 如果文件名的长度过于长?
  - 给文件名的长度设置一个上限，使其最多含有N个字符
    - 每个文件的文件名项都是一个固定大小的字符数组
  - 简单，但是问题是，不是每个文件名都很长
    - 对于那些大部分拥有短的名字的文件，会造成大量的浪费

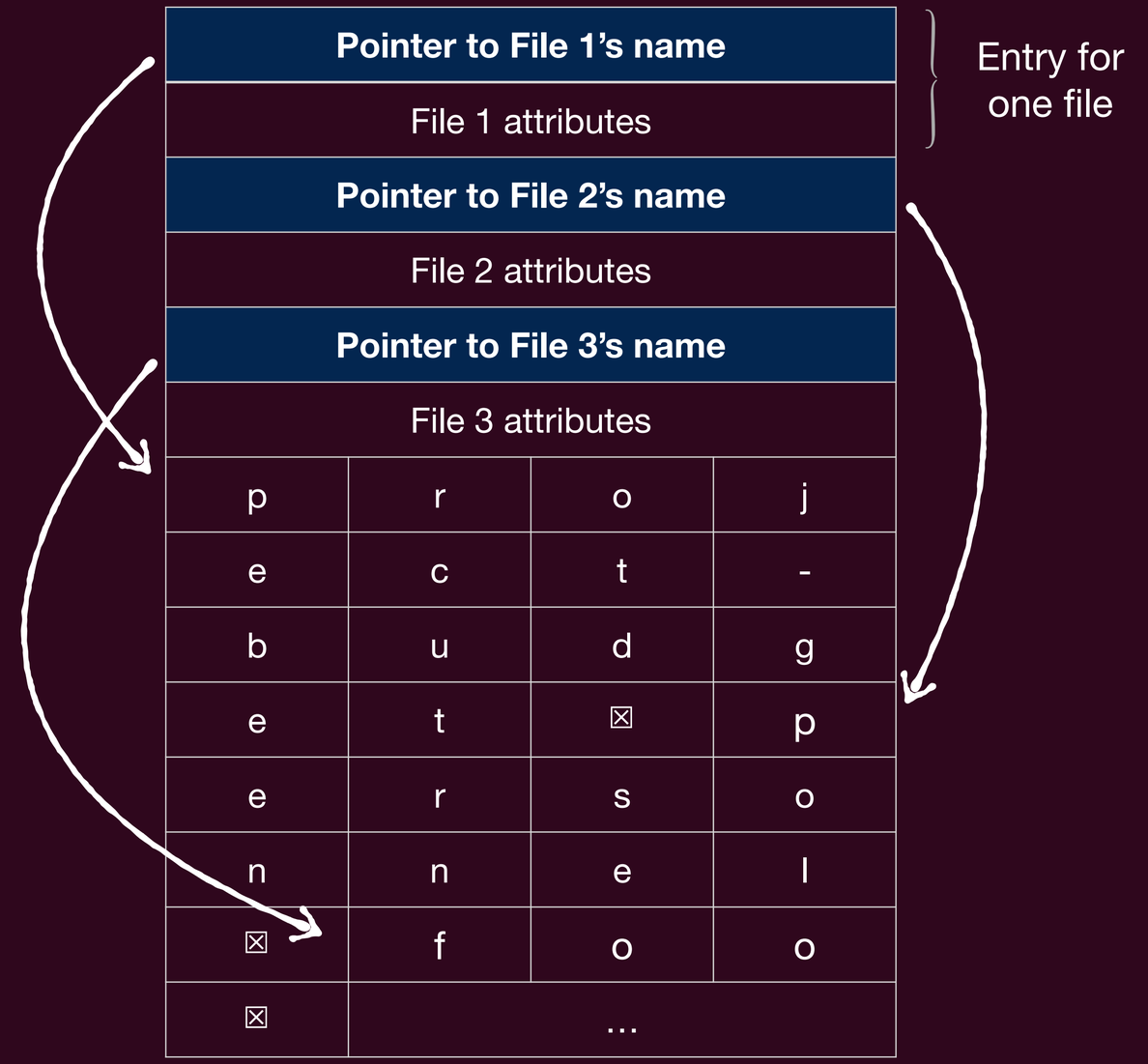
目录组织

- 使得目录项可以有不同的长度
- 对于每一项
  - 该项总共占有的数据长度 (该数据只要定长的空间)
  - 该文件的属性 (定长)
  - 文件名 (不固定长，以一个特殊符号终止)
- 但问题是，如果文件被移除了，就产生了一个不定大小的空洞



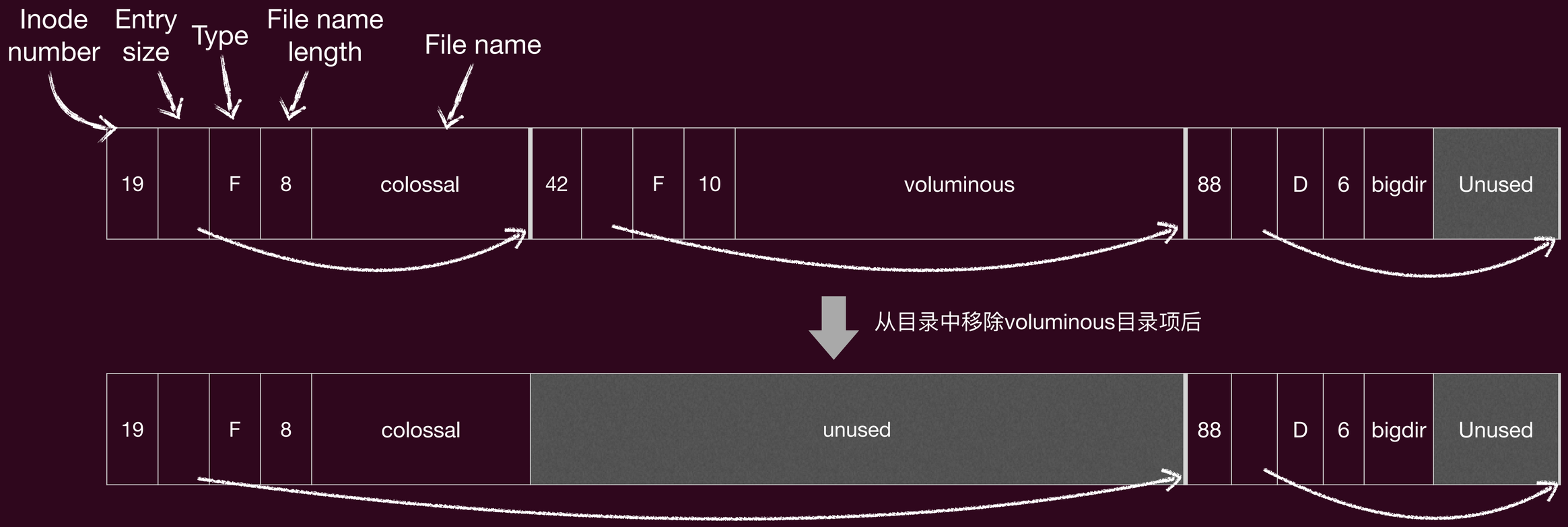
目录组织

- 所有项占的空间都是定长
  - 文件名统一放到一个区域管理
- 当一个项被移出的时候，留下的空间完全可以分配给下一个需要存储的目录项
- 文件名不需要在一个字的开头存储
- 但需要管理这个存放文件名的区域



目录组织

- 文件系统ext2例子
  - 文件名最多为255个字符
  - 记录目录项所占的大小

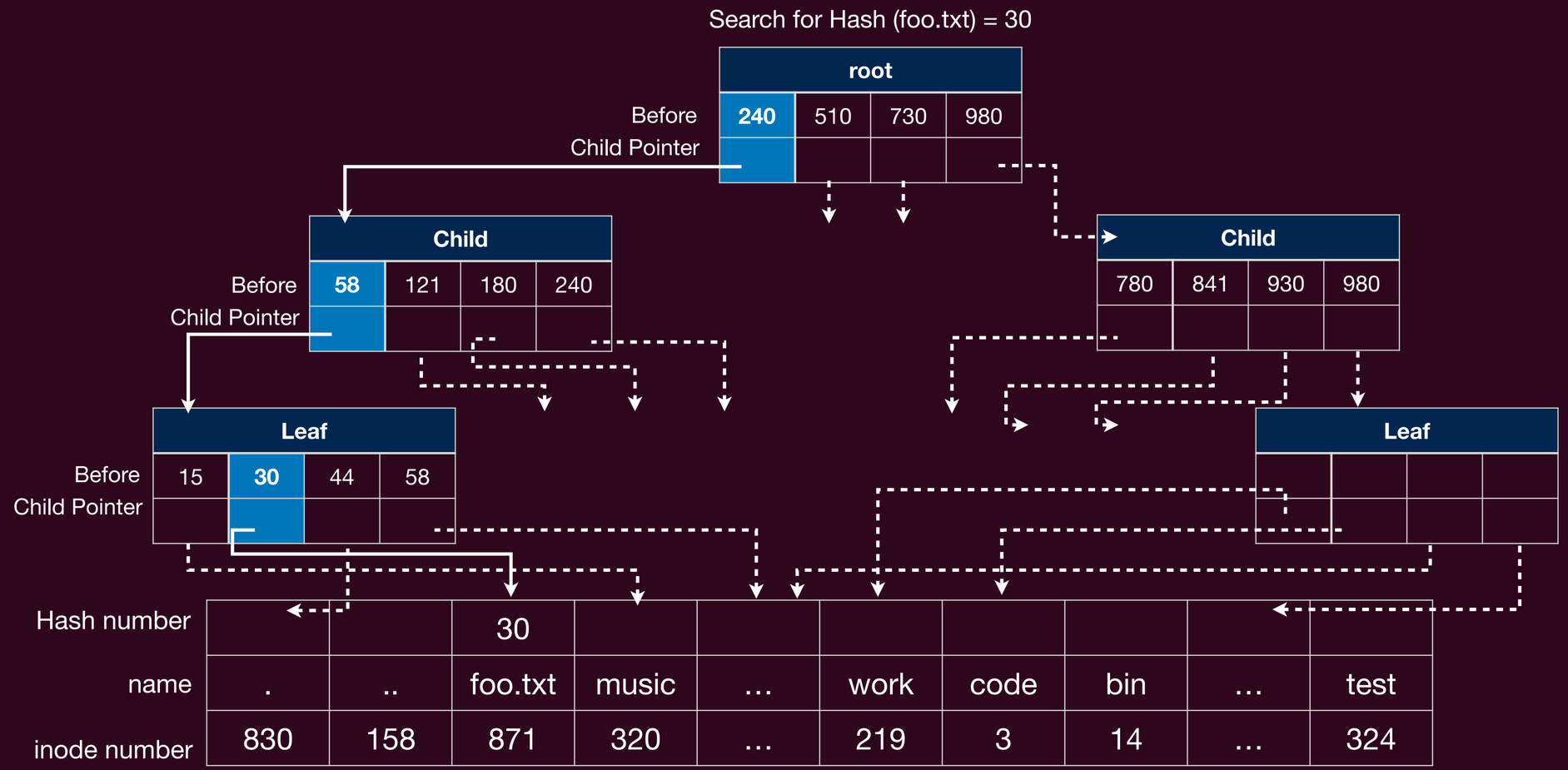


目录组织

- 给定目录实现下的文件查找：
  - 如果目录存储的是〈file name, inode number〉的列表
    - 从头到尾开始搜索列表中的项
    - 如果目录中的项很少没问题（对于现实中的大部分情况都很好），但如果一个目录中出现大量的项，这个做法就很低效了
  - 解决方案：增加一个额外的hash表(以filename为key)
    - 使用链表来处理碰撞
    - 更快的查询，但也需要更为复杂的管理
  - 此外，无论哪种方案，都可以利用cache来进一步加快搜索

目录组织

- 使用搜索树（如B+树）来存储〈file name, inode number〉对, 利用文件名的hash值来进行索引



空闲空间管理

- 如何管理空闲的inode块和数据块?
  - 每当文件和目录需要被创建的时候需要找到空闲的空间存储这些数据
  - 两种常见的管理结构：Bitmap（位图） 和空闲列表
  - 也有采用更复杂结构如B树

Bitmap

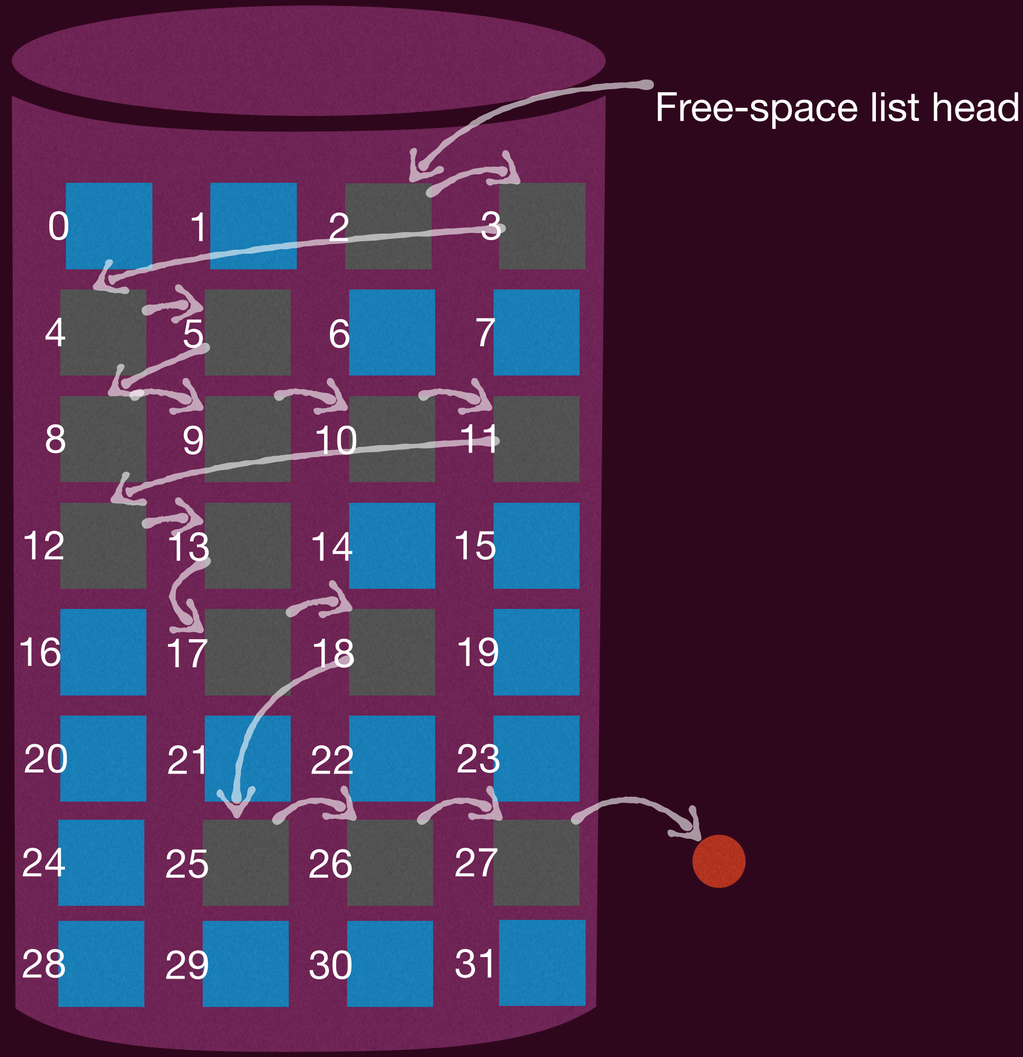
- 每个数据块用一个bit位来表达，0和1代表占据或者空闲
  - 很容易找到连续的空闲空间
  - Bitmap需要额外的空间
    - Block size = 4 KB = 2^{12} bytes
    - Disk size = 1 TB = 2^{40} bytes
    - 需要$2^{40}/2^{12}= 2^{28}$ bits = 32 MB空间的位图

1001101101101100
0110110111110111
1010110110110110
0110110110111011
1110111011101111
1101101010001111
0000111011010111
1011101101101111
1100100011101111
⋮
0111011101110111
1101111101110111

A bitmap

空闲列表

- 用一个链表来维护空闲块
- 没有空间浪费，只需要利用空闲空间块来存储指针即可
- 内存中只需要存储一个指针（head）
- 但分配的过程比较难
  - 需要搜索这个链表来分配多个空闲块(需要额外的磁盘I/O)
  - 难易得到连续的空间

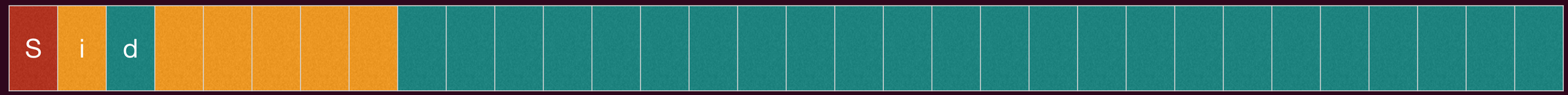


块大小

- 磁盘的块到底多大合适?
  - 过于大: 由于大部分都是小文件，磁盘会产生大量的浪费（内部碎片）
  - 过于小: 读文件时会产生大量的寻道和延迟(每个文件都会包含大量的块)
- 历史上文件系统基本都选择1~4 KB的块大小
  - 当磁盘超过1TB（现在已经很普遍），块大小设置为64 KB更为合适
  - 磁盘空间如今已经很少会出现不足的状

回顾文件系统布局

- 超级块: 文件系统信息
- 分配数据结构: 决定一个inode或者一个数据块是否空闲
- inode表: 每一个存储着一个文件的元信息
- 数据区域: 一系列的数据块



读文件

- 假设一个文件系统被挂载，其中超级块在内存中,但其他 (inodes, directories) 都在磁盘上
- 读一个文件首先需要打开这个文件：
  - 首先需要循着路径名找到相应的inode
  - 读取这个inode，做权限检验，合法就返回文件描述符（对应的操作会反映到打开文件表中）
- 然后对每个发起的读操作：
  - 读相应的inode
  - 读相应的数据块
  - 写inode (更新上次访问时间)
  - 更新内存中打开文件表中的offset

读文件

- 例子：读取文件/foo/bar的开头3个数据块

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

1. Read root inode (must be known) to locate root data
2. Read root data to find foo inode
3. Read foo inode to locate foo data
Read foo data to find bar inode
5. Read bar inode into memory
6. Read bar inode to locate data
7. Read data block of bar
8. Update timestamp of bar inode

The read will further update the in-memory open file table to maintain the file offset for this file descriptor.

写文件

- 首先打开文件（过程与读文件类似）
- 但与读文件不一样的地方在于，写文件可能需要分配一个新的数据块(除非覆写旧数据块)
  - 读和写空闲数据块结构比如bitmap (找到一个空闲的数据块标记为占有)
  - 读和写文件的inode (更新inode表中这个新数据块的位置)
  - 写这个新的数据块

写文件

- 当创建一个文件时需要更多的操作
  - 读和写空闲数据块结构比如bitmap (找到一个空闲的数据块标记为占有)
  - 读和写文件的inode (初始化)
  - 读和写目录的数据 (将文件名和其对应inode项写入)
  - 读和写目录的 inode (更新目录)
  - 如果目录需要增加容量来添加这个新的项，需要更多的I/O (访问空闲数据结构，增加新的目录的数据块)

写文件

- 创建 `/foo/bar` 并写3入个数据块

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read Find a free inode Make it allocated	read	write Initialization the inode	read write	read	write link the file name and the inode in dir		
				read Locate the bar inode						
				write update the directory inode last access and modified time						
write()	read write find a free data block update data bitmap				read Read the inode to get data block location					
					write update with the new data block location			write write the actual data block		
write()	read write				read write				write	
write()	read write				read write					write

The file must be opened as well

文件系统性能

- 一个文件系统如何减少I/O带来的开销?
  - 访问磁盘 (I/O 操作) 比访问主存慢得多
  - Read-Modify-Write 过程
    - 几乎所有的文件系统都将文件分割成固定大小的块
    - 将磁盘块复制到主存以访问其内容，并只有发生修改才将其写回

缓存和缓冲（Caching and Buffering）

- 显然无法将磁盘的所有块都放进内存，应该只放入需要频繁访问的块（类似缓存机制），这样就可以读取该块就可以极大的减少I/O的操作了
  - 利用哈希表来记录一个给定块是否在主存里
  - 如果cache满了就需要移除一些块 (类似页面替换机制，比如LRU)
- 现代系统将虚拟内存页和文件系统缓存页集成到统一的页缓存中
  - 这样，内存可以在虚拟内存和文件系统之间更灵活地分配

缓存和缓冲

- 预取: 可以将需要的数据块提前存入缓存中来提高缓存击中率
  - 很多读文件操作是线性的，这时预取的效率很高
  - 但如果是随机访问，预取则无效
  - 因此可能会造成加载无用的数据块和替换需要用的数据块这样低效的操作
  - 文件系统可以追踪每个打开文件的访问形式

缓存和缓冲

- 读的I/O可以通过的充分的缓存来避免，但还是需要写磁盘的I/O来达成文件数据的持久性
  - 写缓冲（Buffering）
    - 可以延迟写操作：文件系统可以批处理一些I/O写的操作（避免多次频繁的少量的I/O的操作，比如可以合并bitmap的很多位运算）
    - 缓冲写操作，文件系统还可以合理的调度I/O来提高性能（比如让相近的数据块一并写入）
    - 有些写操作甚至可以完全避免 (创建了一个文件然后删除)
    - 但更好的性能也会带来潜在的不一致性：一旦发生crash, 没有提交到磁盘的写操作会丢失
      - 一般每隔一段时间会写缓冲，时间间隔大小是一个trade-off

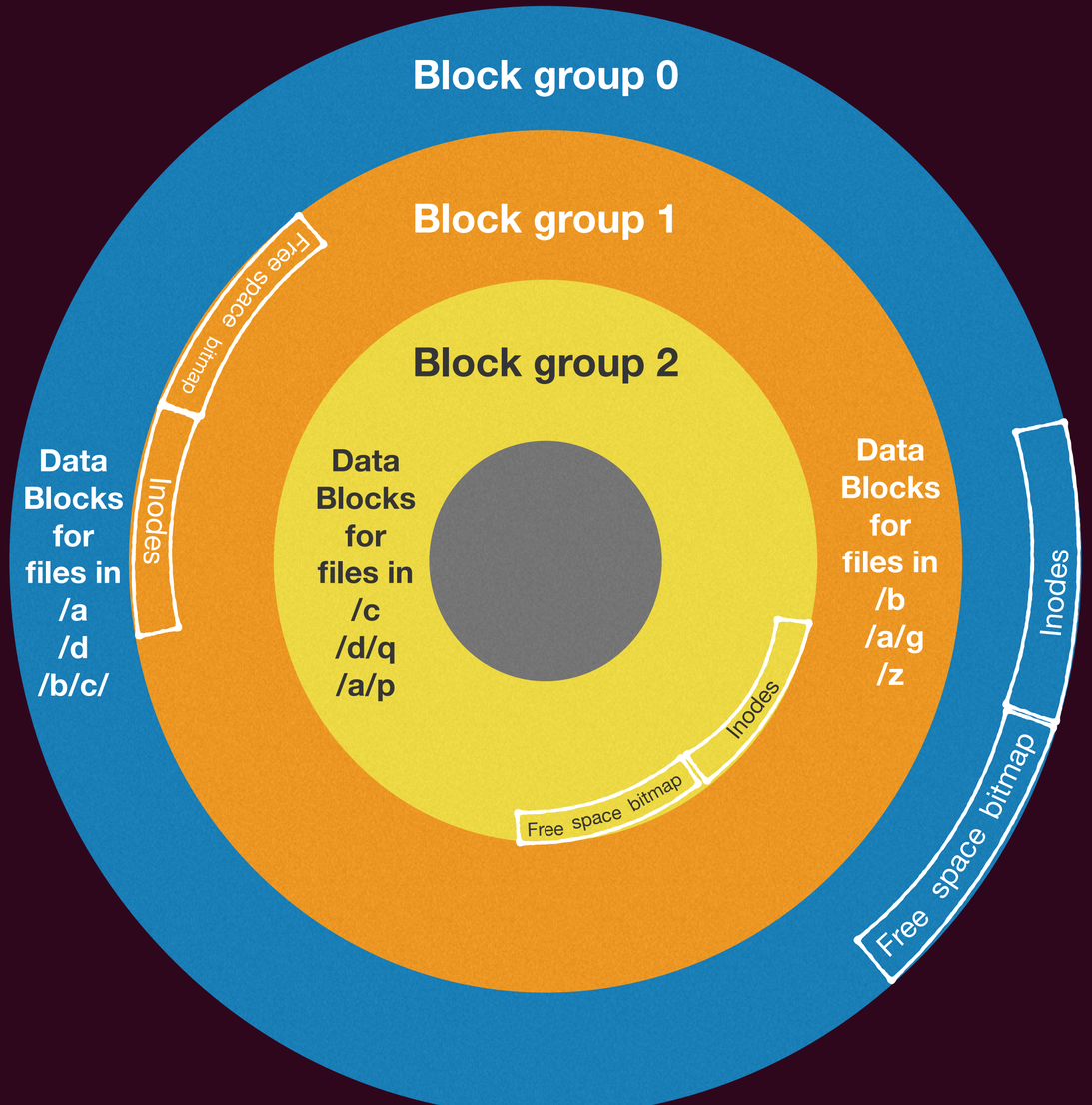
快速文件系统（Fast File System，FFS)

- 设计文件系统需要将磁盘的特性考虑进去
  - 原始的Unix文件系统只利用了大概2%的磁盘带宽（所有时间都浪费在寻道上）
    - 其将磁盘完全看成是一个随机访问的存储器
    - 空闲块没有很好管理，数据在磁盘上随意分布（一个线性的文件只能随机的分配数据块）

- 要提高性能，则需要应用碎片整理将分布在磁盘上的每个文件的块重新复制到相邻的块

快速文件系统(FFS)

- 一个好的文件系统是有磁盘意识的
- FFS将文件系统被划分为一系列块组
  - 每个组由一组相邻的磁道构成（即柱面组）
  - FFS将可能顺序访问的块放在同一个组中，以减少磁臂移动的量



快速文件系统 (FFS)

- 在 FFS 中实现数据局部性（data locality）：将相关内容放在一起（将不相关内容分开）
  - 对于文件：
    - 将数据块分配到与其 inode 相同的组中
    - 将同一目录中的所有文件放置在该目录所在的组中
  - 对于目录：
    - 找到已分配目录数量较少（以平衡各组间的目录）且空闲 inode 数量较多（以便能分配大量文件）的组。
    - 将目录数据和 inode 放在该组中

快速文件系统 (FFS)

- 磁臂移动和旋转时间只有在磁盘具备这些特性时才需要考虑
- 对于固态硬盘（SSD）：
  - 随机访问和顺序访问一样快
  - 每个块只能写入有限次数
    - 因此，SSD反而不应该进行碎片整理
  - 新的问题：需要均匀分散磁盘的磨损

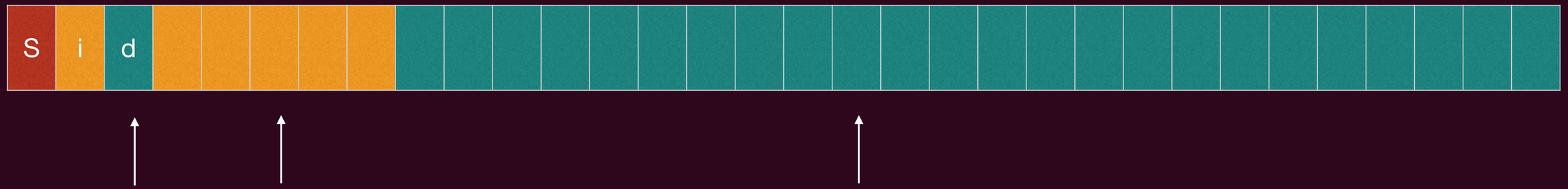
文件系统可靠性

一致性

- 崩溃一致性问题（Crash-consistency problem）：崩溃故障可以在任何时刻发生（例如，电源故障），文件系统可能会受到这种崩溃的影响
  - 对文件系统的更新操作需要多次 I/O 操作（更新多个数据结构）
    - 崩溃时，某些操作可能完成，而某些操作可能丢失
  - 这会导致文件系统处于不一致的状态

一致性

- 当需要为一个文件增加一个数据时
  - 需要写一个新的数据块
  - 需要写这个文件的inode
  - 需要写数据的bitmap
- 如果这中间发生了一个crash?



一致性

- 如果这三个只有一个写成功了...
  - 只有数据块被写进磁盘
    - 这个写会被丢失，但是文件系统的元信息一致
  - 如果只有inode在磁盘上更新成功
    - 不一致: 数据的bitmap认为该块是空闲的，但inode认为该块已经被使用了 (会读到一个垃圾数据)
  - 只有空闲bitmap被更新成功
    - 不一致: 数据的bitmap认为该块已经被使用了，但没有inode指向它 (磁盘空间泄露)

一致性

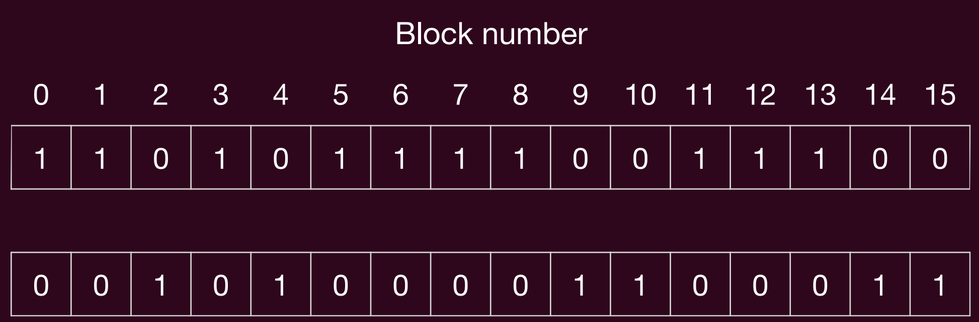
- 如果两个写成功了 …
  - inode 和 data bitmap 被正确写回磁盘，更新成功
    - 文件系统的元信息是一致的
    - 但读取时会是一个垃圾信息
  - inode 和 data block 被正确写回磁盘
    - inode 指向磁盘里正确的数据块
    - 但inode 和 data bitmap不一致
  - Data bitmap 和 data block 被正确写回磁盘
    - inode 和 data bitmap不一致
    - 此时无法获知这个被使用的数据块属于哪个文件（因为没有相应的inode）

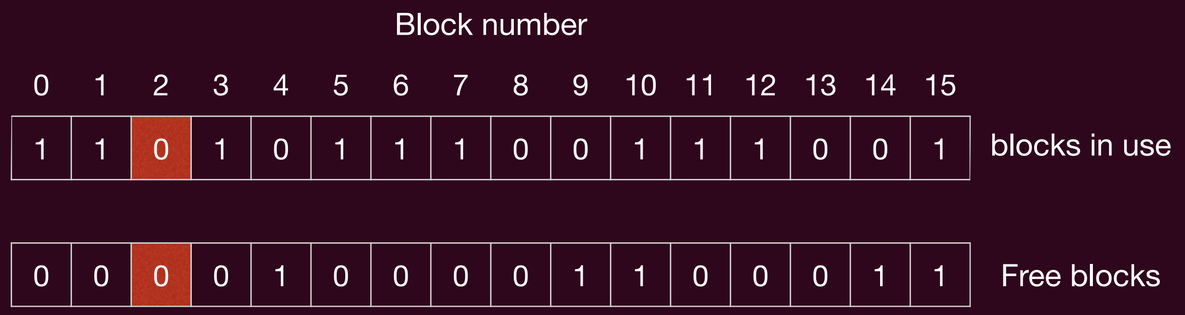
一致性

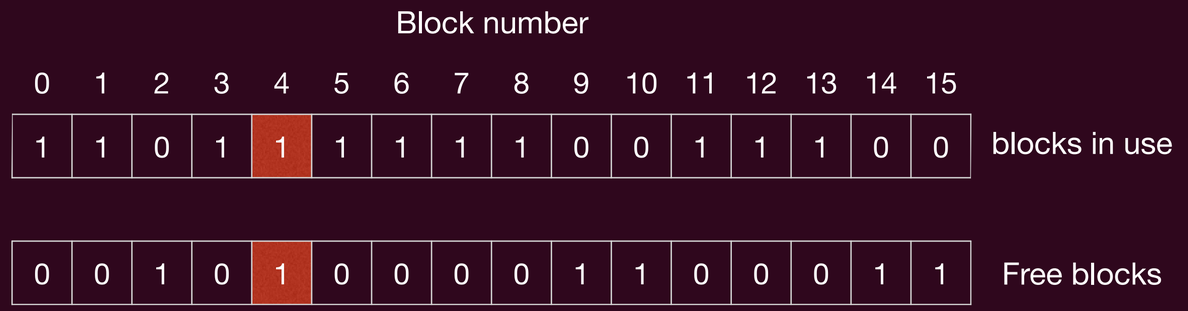
- 理想的方案
  - 一个文件系统原子的从一个一致的状态进入另一个一致的状态
  - 但磁盘只能支持一个写操作是原子的（而两个状态之间的迁移往往涉及多个写操作）
- 实际的方案
  - 进行文件一致性检查（File System Consistency Check, fsck）
  - 日志化

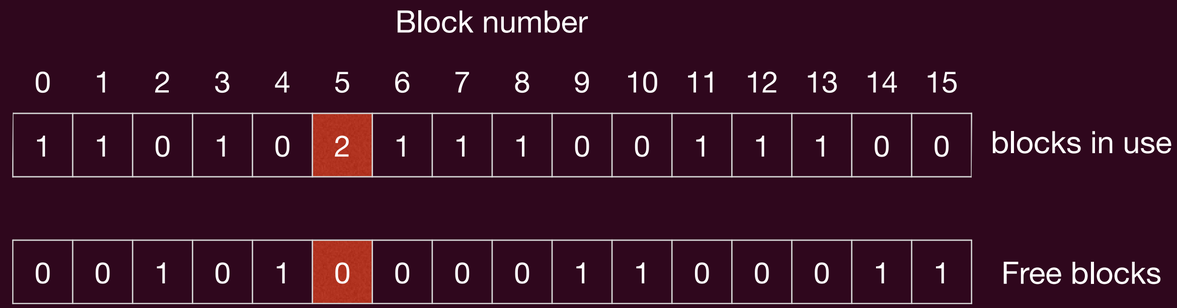
进行文件一致性检查

- 让不一致发生，并在事后修复（在重启期间）









比较使用和空闲块的不一致

进行文件一致性检查

- fsck：在挂载文件系统之前运行（fsck运行期间需要没有相关的文件系统活动），确保文件系统元数据在内部一致
  - 对超级块进行完整性检查：
    - 文件系统大小是否大于已分配的总块数?
    - 发现不一致时，使用超级块的另一份副本

进行文件一致性检查

- 完整性检查超级块
- 检查空闲块和位图的有效性
  - 扫描 inode 以确定哪些块已分配
    - 生成数据位图的正确版本
    - 发现不一致时，覆盖位图（信任 inode）
  - 进行类似的检查以更新 inode 位图

进行文件一致性检查

- 完整性检查超级块
- 检查空闲块和位图的有效性
- 检查 inode 是否未损坏
  - 例如，每个 inode 应该有一个有效的类型字段（普通文件、目录、符号链接等）
  - 如果问题无法修复，清除 inode 并更新 inode 位图

进行文件一致性检查

- 完整性检查超级块
- 检查空闲块和位图的有效性
- 检查 inode 是否未损坏
- 检查 inode 链接
  - 扫描整个目录树，计算每个文件和目录的硬链接数
  - 发现不一致时，修正 inode 中的链接计数
  - 如果没有目录引用已分配的 inode，将其移动到名为 lost+found 的目录中

进行文件一致性检查

- 完整性检查超级块
- 检查空闲块和位图的有效性
- 检查 inode 是否未损坏
- 检查 inode 链接
- 检查重复的指针和损坏的块
  - 两个inode如果指向同一个数据块
    - 清除一个inode (如果其明显损坏), 或者复制那个数据块 (使的每个inode指向不同的数据块（内容拷贝）)
  - 如果一个inode指向分区外的数据块
    - 直接移除这个inode

进行文件一致性检查

- 完整性检查超级块
- 检查空闲块和位图的有效性
- 检查 inode 是否未损坏
- 检查 inode 链接
- 检查重复的指针和损坏的块
- 检查目录
  - 检查 . 和 .. 是否是第一和第二个项
  - 检查其中的每个inode所指向的地方是否已经被分配

进行文件一致性检查

- fsck的缺点
  - 一般来说，只能发现“不一致”，但没有正确的修复手段
    - 不知道“正确”的状态是什么，只知道“一致”的状态是什么，“正确”必定“一致”，但“一致”未必正确
  - 太慢 (需要扫描整个磁盘)

Not full specification!

日志

- 基本思路：预写日志（Write-ahead logging）或日志（Journaling），该思路借鉴自数据库系统
  - 在覆盖结构之前，先写一个小日志（存储在磁盘上），描述将要做的事情
  - 如果在更新过程中发生故障，我们可以在重启时读取日志并重试
    - 在写入意图之前崩溃：没有操作
    - 在写入意图之后崩溃：重做该操作
  - 在更新期间增加的一些工作量，可以大大减少恢复期间所需的工作量
    - 无需扫描整个磁盘，只需要查看奔溃前的日志中的记录即可
  - 被许多文件系统使用，包括 Linux 的 ext3、ext4 和 Windows 的 NTFS

日志

- 文件的更新被作为事务（transaction）保存在日志中（日志写入）
  - 写入 TxB（transaction begin，事务开始），其包括事务标识符和其他相关信息（例如，块I[v2]、B[v2]和Db的地址）
  - 将inode（I[v2]）、位图（B[v2]）和数据块（Db）的具体更新放入日志中（待处理更新）
  - 写入 TxE（transaction end事务结束）

日志

- 一旦事务安全地写入磁盘，将更新内容（元数据和数据）写入其最终的磁盘位置（checkpoint，检查点）
  - 将 I[v2]、B[v2] 和 Db 写入它们的磁盘位置

日志

- 日志每个部分的写入顺序并不确定， 磁盘可以执行调度，并以任意顺序完成写入
  - 比如可能先写入 TxB、I[v2]、B[v2] 和 TxE
  - 再写入 Db
- 那么在再写入Db之时发生崩溃，你是无法感知到这次崩溃的，因为在外观上日志已经完成（TxE已经写入），但Db是垃圾数据，做检查点时会写入垃圾数据

日志

- 解决方案：
  - 将除 TxE 之外的所有块写入日志，并等待这些写入完成（日志写入）
  - 发出 TxE 块（512字节以内，保证原子性）的写入请求，等待写入完成（日志提交），使日志处于安全状态
  - 将更新内容（元数据和数据）写入它们的最终磁盘位置（检查点）

日志

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

日志

- 当需要从崩溃中恢复时，文件系统会扫描日志并查找已提交到磁盘的事务
  - 如果在日志提交之前发生崩溃
    - 则忽略待处理的更新
  - 如果在日志提交之后但在检查点之前发生崩溃
    - 则按顺序重放已提交的事务（重做日志）
    - 可能会有些写冗余（因为有些日志可能已经正确写入，会被重新再写一次），但问题不大，因为崩溃发生不频繁

日志

- 日志在提交之后其实就不需要保留了
- 有限日志：将日志视为循环数据结构（bounded buffer），反复重用（circular log）
  - 一旦事务被检查点记录，释放它在日志中占用的空间
  - 用一个日志超级块记录哪些事务尚未被检查点记录

日志

- 数据日志记录代价较高：数据被写入两次（到日志和实际数据块）
- 元数据日志记录：用户数据不写入日志（在大多数系统中更常见）
  - 先记录元数据日志，然后写入数据块
    - 确保一致性，但文件可能包含垃圾数据
  - 更好的做法：先写入数据块，然后记录元数据日志
    - 确保指针不会指向垃圾数据
    - 等待数据写入和元数据日志写入完成，然后进行日志提交

日志

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

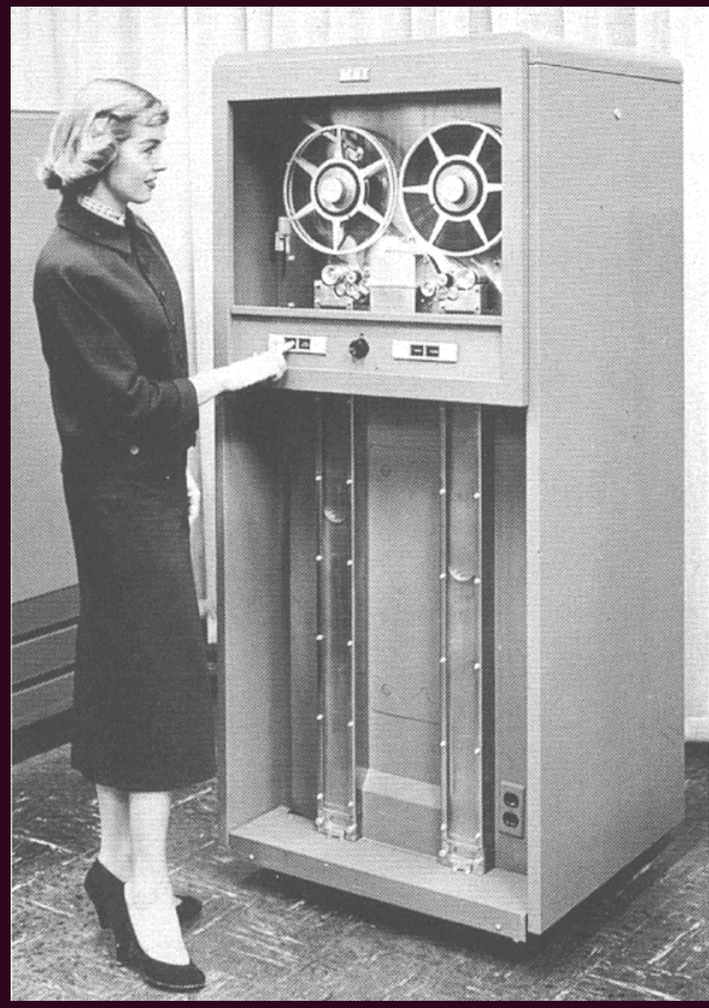
日志结构文件系统

传统文件系统的吞吐量问题

- 通常我们将磁盘抽象地看作是一个数据扇区数组
- 但是，顺序读/写比随机访问要快得多
- 缓存层可以帮助减少对相同位置的重复磁盘 I/O
  - 但缓存无法帮助写入大量新数据
- 如果只写一个大文件，我们可以得到一个大的连续数据块序列
- 但是，如果写入许多小文件，则很难实现良好的性能
  - FFS 块组可以帮助减少寻道的长度。✅
  - 延迟和批量请求允许磁盘的固件重新排序它们 ✅
  - 但我们仍然做了很多寻道😔

日志结构文件系统（Log-structured Filesystems, LFS）

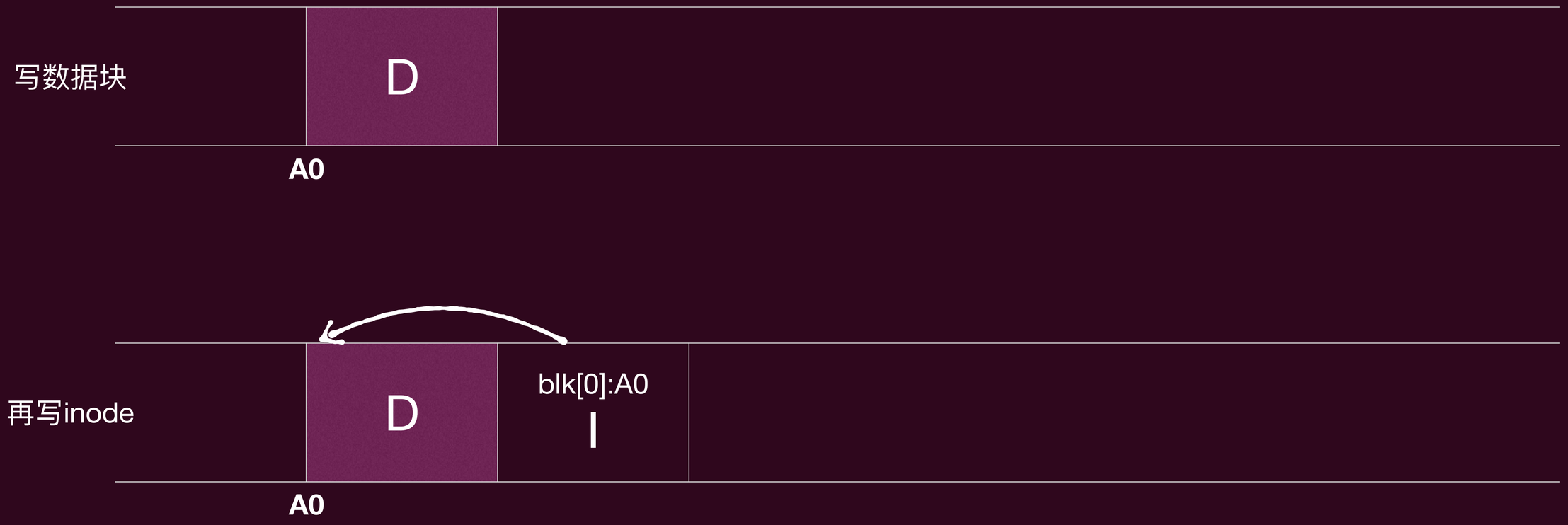
- 一个想法：应该尝试将每次写入都顺序执行！但是如何做到呢?
- 一个思路是将文件系统视为日志（log）
  - 日志是一系列事件，新事件位于末尾
- 当数据发生变化时，不必回去编辑原始数据，只需将新副本存储在末尾（类似磁带）。
  - 在最简单的情况下，假设容量无限



Tape drive circa 1953

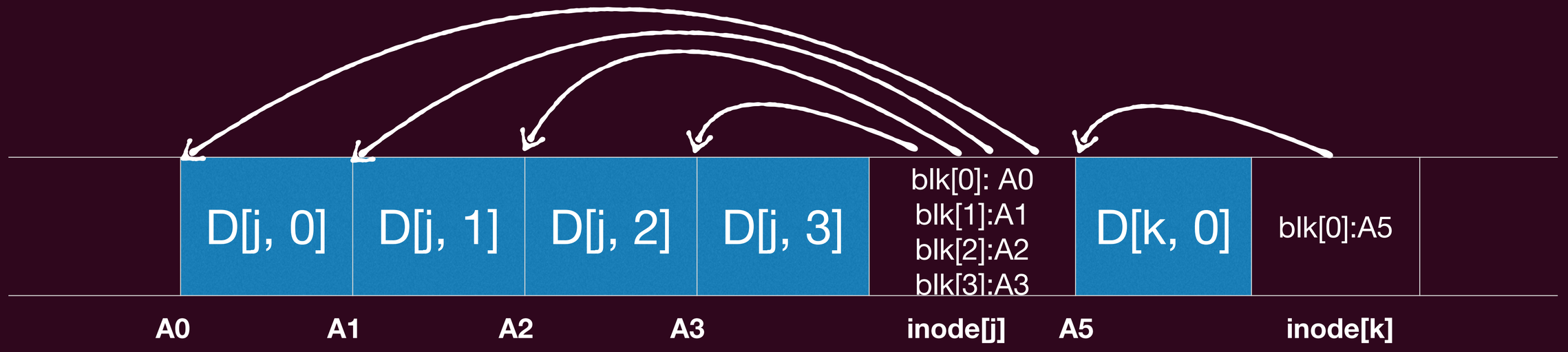
日志结构文件系统

- 日志文件系统（LFS） 仍然有 inode 和数据块，只是将它们放置方式不同。
- 总是写到末尾：例如，当写入一个小文件时



日志结构文件系统

- 一如既往，我们先写入数据块，然后再写入 inode，以最小化中断/崩溃的影响。

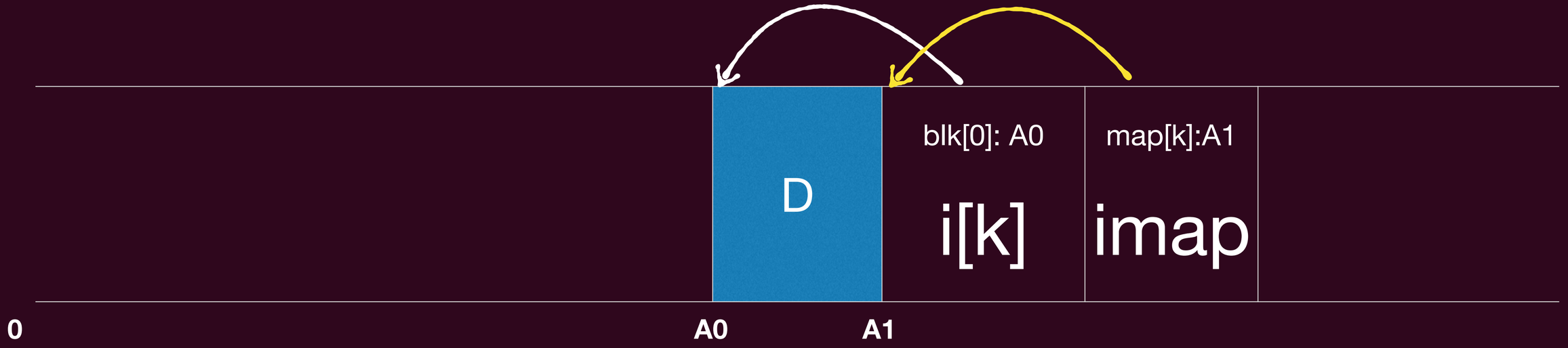


减少旋转延迟

- 即使我们正在顺序递增的位置进行写入，如果请求不是一起发出的话，仍然有可能出现长时间的旋转延迟
  - 例如，写入扇区 25、26、27 在最佳情况下可能非常快，但只有在我们准备好在写入26后立即写入27时才会如此
  - 在写入26和写入27之间存在小延迟可能会导致我们等待整个磁盘旋转
- 因此，LFS 将写入操作先缓冲在内存，等到足够多时（几MB，称为段）再批量写入磁盘

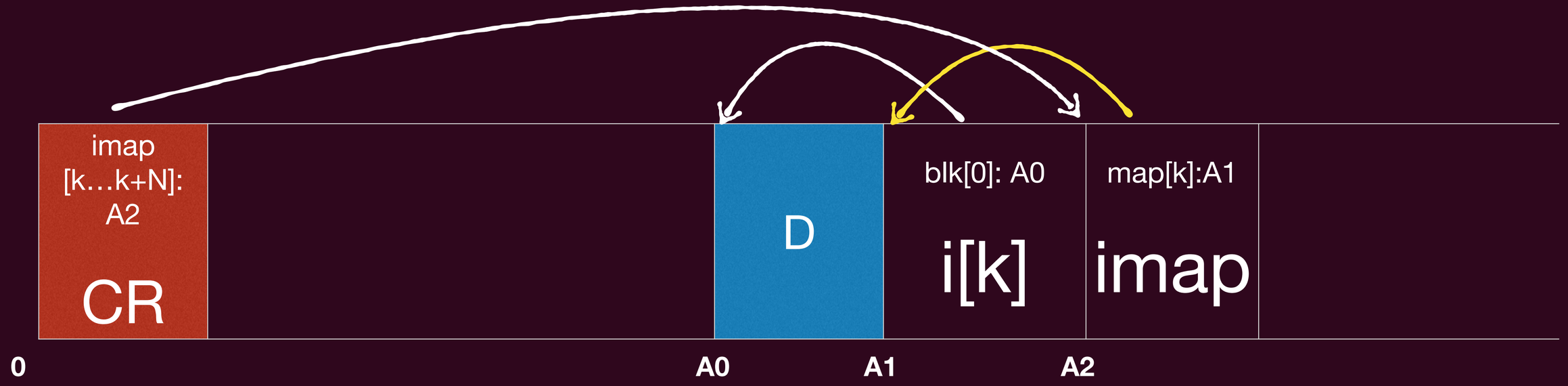
日志结构文件系统

- 之前的文件系统中，根据inode号可以用来在数组中找到 inode 结构。
- 但LFS 使得查找 inode 变得更加困难。它们被放置在磁盘上的任意位置。现在我们如何找到特定的 inode?
- 给每个段一个 inode 映射，其给出了其每个 inode 的地址
  - 段很大，可以包含数百个 inode，即一个映射包含多个inode的地址
- 但这并不是一个完整的解决方案，因为有许多段和许多inode 映射，它们本身位于磁盘上的随机位置



再加一个间接层

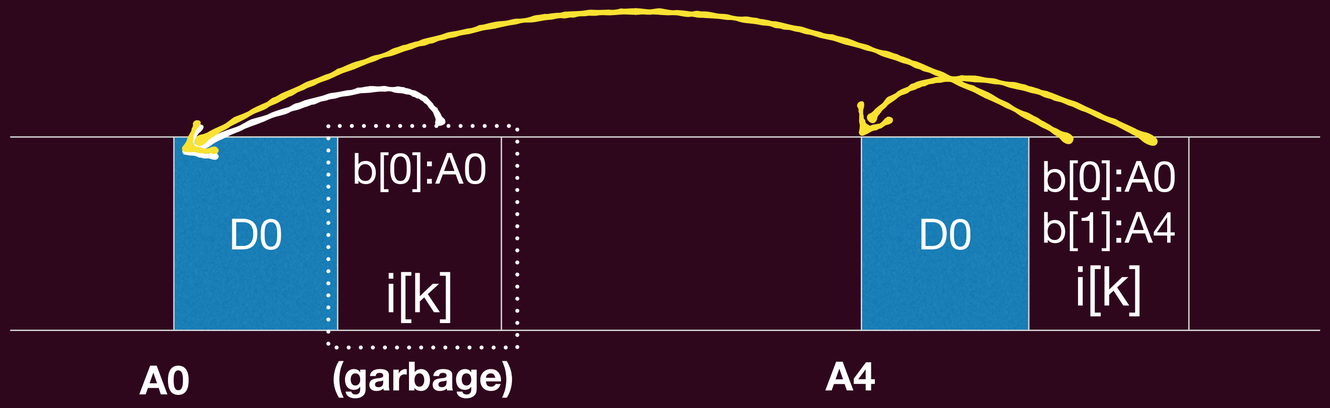
- 在磁盘的开头，存储一个检查点区域（Checkpoint Region, CR），它只是指向磁盘上所有有效的 inode 映射
- i-map分布在磁盘上所有有效的段中。 查找一个 inode 涉及查看整个 imap
- 这可能很慢，但在实践中，我们应该能够将整个imap缓存在内存中
  - 当然，需要定时更新写回该缓存



永远不修改已有数据

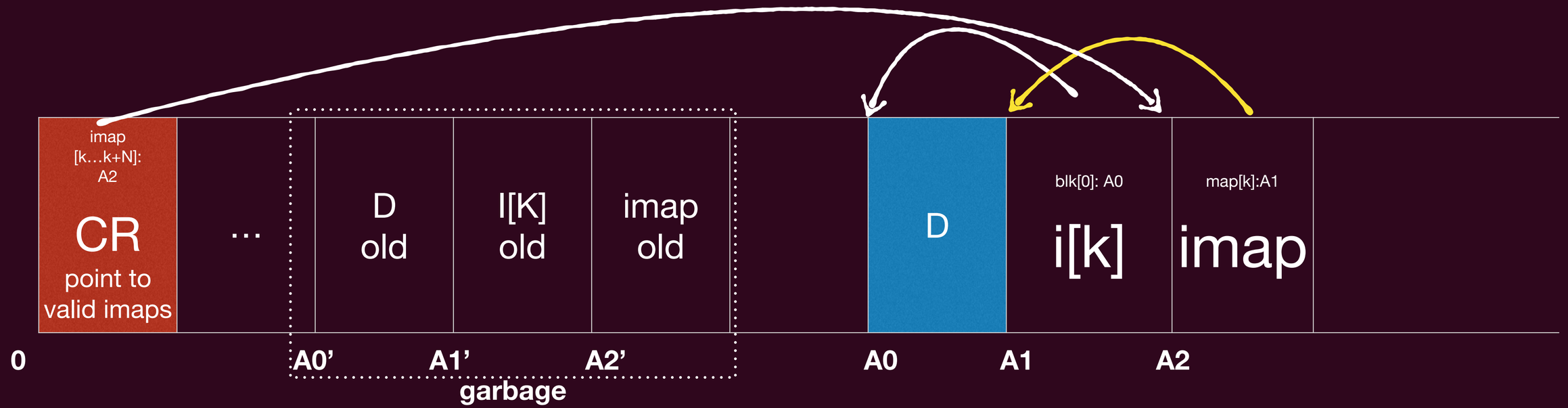
- 始终写入整个块的新副本，如果编辑数据：

- 如果向文件追加数据，需要编辑 inode：



指向新的版本

- 旧数据仍然存在于磁盘上，但内存中的i-map和其磁盘上的持久副本不再引用它
- 如果磁盘空间是无限的，那就足够了
- 此外，如果我们保存旧版本的检查点区域，它还可以用来查看文件系统的旧快照！
  - 能够保留旧快照的文件系统被称为版本化文件系统（a versioning file system）

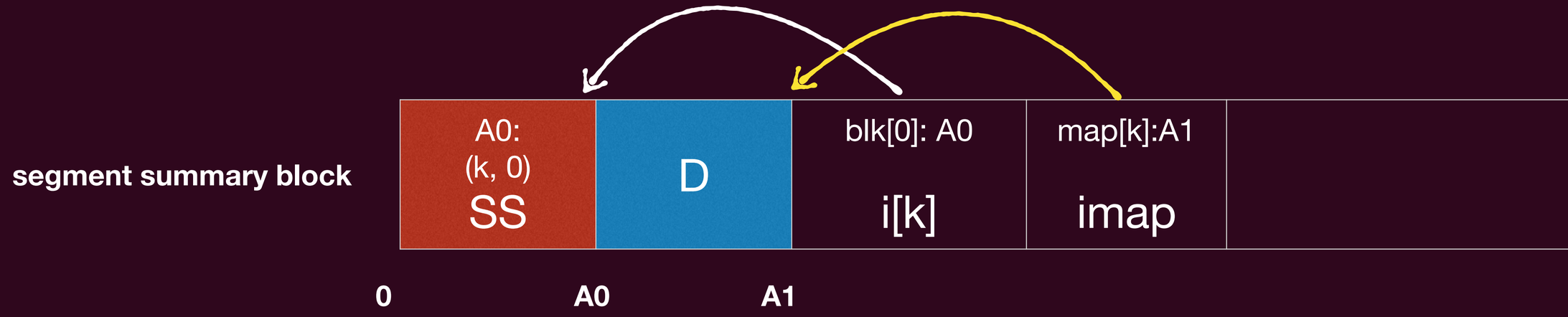


磁盘容量是有上限的

- 受限于磁盘容量，不可能永远顺序写入， 不能无限期地保留旧版本的数据。
- 最终需要对具有可用空间的段进行垃圾回收。
  - 实际上，最好是释放整个段（如果都是垃圾）
  - 但如果遇到部分填充的段，那么可以先释放完整的段，并在日志末尾重新写入一个压缩版本的段（只写入垃圾收集器留下的“空洞”）
  - 垃圾收集器定期扫描磁盘，可能在空闲时进行
  - 但是垃圾收集器如何决定哪些块是活动的，哪些是无用的呢

反向指引的垃圾回收

- 分布在磁盘上的i-map直接告诉我们一个inode 是否有效
- 但数据块难以进行分类是否有效
  - 一个简单的方法是检查磁盘上的每个 inode，寻找对块的引用；
  - 但这太慢了，更好的做法是从数据块到inode有某种反向引用
- 解决方案：添加一个段摘要块，指示段中每个数据块的inode编号和块偏移量
  - 然后检查 SSB 中列出的 inode 是否仍然引用该数据块



总结

- 两个关键抽象：文件和目录
  - 元数据信息：inode
  - 文件描述符File Descriptor, 打开文件表
  - 软/硬链接
- 文件实现：
  - 布局（超级块、空闲位图、inode块、数据块）
  - 性能（FFS、缓存和缓冲）、一致性检查、日志
  - 日志结构文件系统

阅读材料

- [OSTEP] 第39, 40, 41, 42, 43, 44章