操作系统

第8章 文件管理、第9章 磁盘存储器管理

朱小军, 教授
https://xzhu.info
南京航空航天大学
计算机科学与技术学院
2025年春

文件系统接口

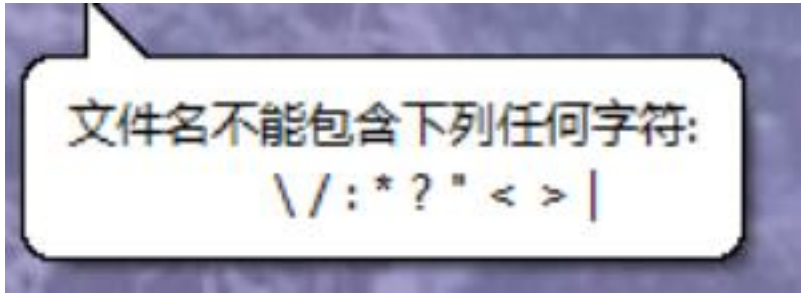
你知道哪些文件系统?

文件（file），是操作系统对磁盘（等永久性存储介质）的抽象。

- 进程是对CPU的抽象
- 地址空间是对物理内存的抽象

文件名

- 长度有限制
  - NTFS 为255个字符
  - FAT12, FAT16, FAT32 为11个字符
  - 为什么?
- 字符有限制
  - 有一些字符不允许使用
  - 常见的，比如 / \ ? :
  - 为什么要有这种限制?



减号其实也有问题

- 如果文件名以“-”开头，则在命令行会有问题

```
xzhu@DESKTOP-9QI0LGD:/mnt/c/Users/xzhu/Downloads$ rm *.pdf
rm: invalid option -- '2'
Try 'rm ./-2050750996_8a80943a747c210e01748f6436000ec6.pdf' to remove the file '-2050750996_8a80943a747c210e01748f6436000ec6.pdf'.
Try 'rm --help' for more information.
xzhu@DESKTOP-9QI0LGD:/mnt/c/Users/xzhu/Downloads$ rm ./*.pdf
```

扩展名

.exe, .c, .gif, .html, .jpg

- Unix系统对扩展名不做要求，扩展名仅用来帮助用户和应用程序识别文件
- Windows系统充分使用扩展名识别文件，扩展名标识了文件的打开方式（存在注册表中）

文件类型

- 普通文件、目录文件、特殊文件
  - 普通文件是用户文件
  - 目录文件是系统文件
  - 特殊文件用于管理 I/O 设备
- 每个操作系统至少应当识别一种文件类型（?）
- 普通文件是ASCII文件或二进制文件
  - ASCII文件便于显示、打印、重定向
  - 二进制文件（即，非ASCII文件）exe，obj，rar
  - ASCII文件中每个字节的第0位为0（因ASCII字符只需用7个比特表示），而二进制文件无此约束

(普通)文件的逻辑结构

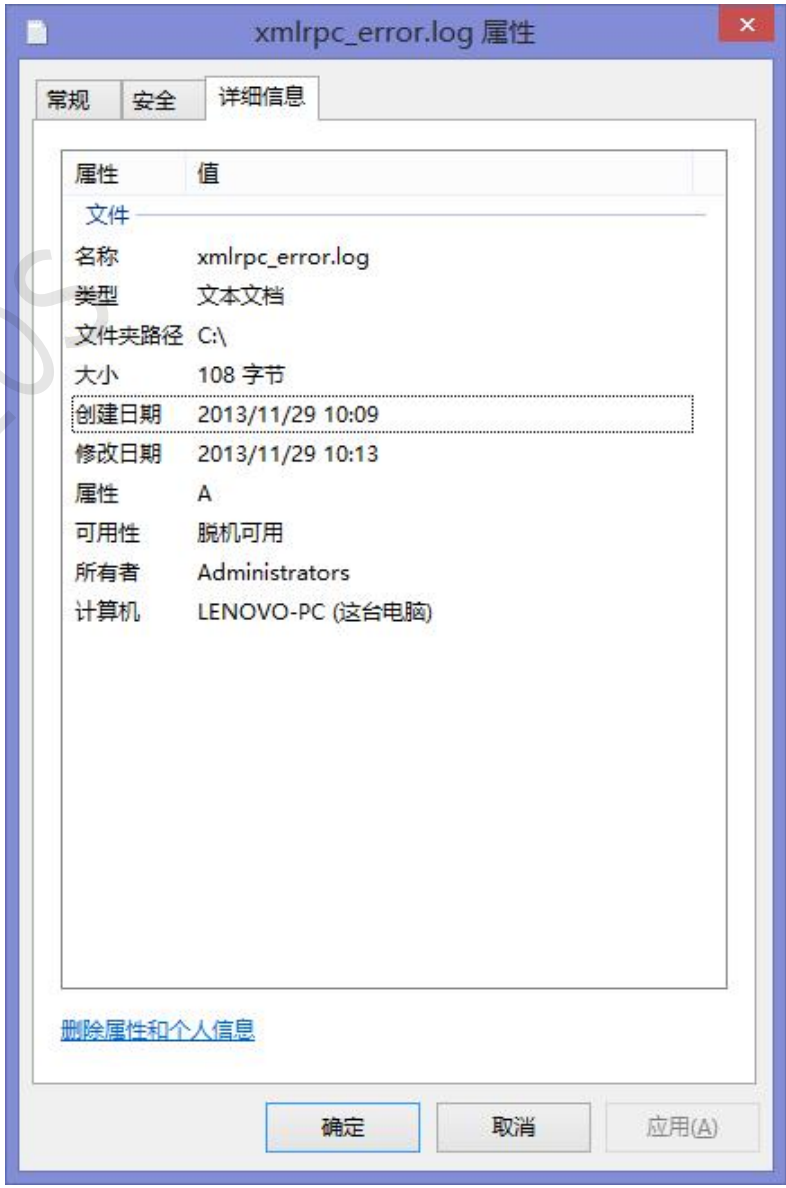
- 字节串（byte sequence）
  - 文件是一串字节
  - 用户可以将任意内容放入文件
  - OS 不关心文件的具体内容
  - windows，unix均采用了这种文件结构

文件存取方式

- 顺序存取
  - 从头开始读，顺序读，直到结尾
  - “磁带”
- 直接存取
  - 应用需要，如，数据库系统
  - 两种实现方式
    - 每个read 指明地址
    - 先seek，然后开始顺序读（windows、linux的方式）
  - “磁盘”

文件的属性

- 在文件名和内容之外，OS 还为每个文件维护一些其他信息
- 每个OS维护的信息有所差别
- 有一些软件会利用这部分信息
  - 如，Unix下的make工具



文件的操作

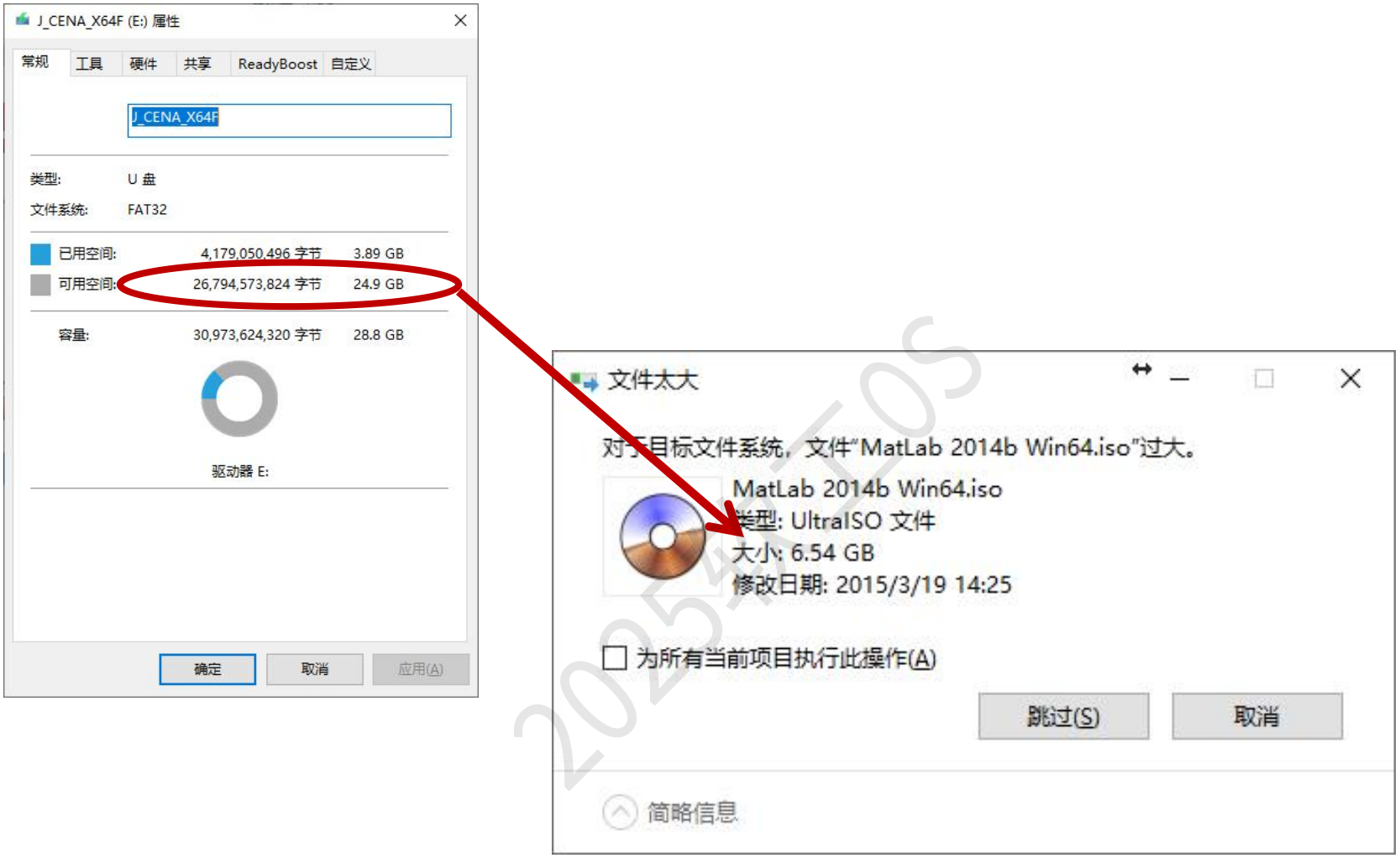
- 基本操作
  - 创建、删除
  - 读、写
  - 设置读写位置（seek）
  - 设置和获取文件属性
- 其他操作
  - 可由基本操作组合而成
  - 添加
  - 重命名
  - …

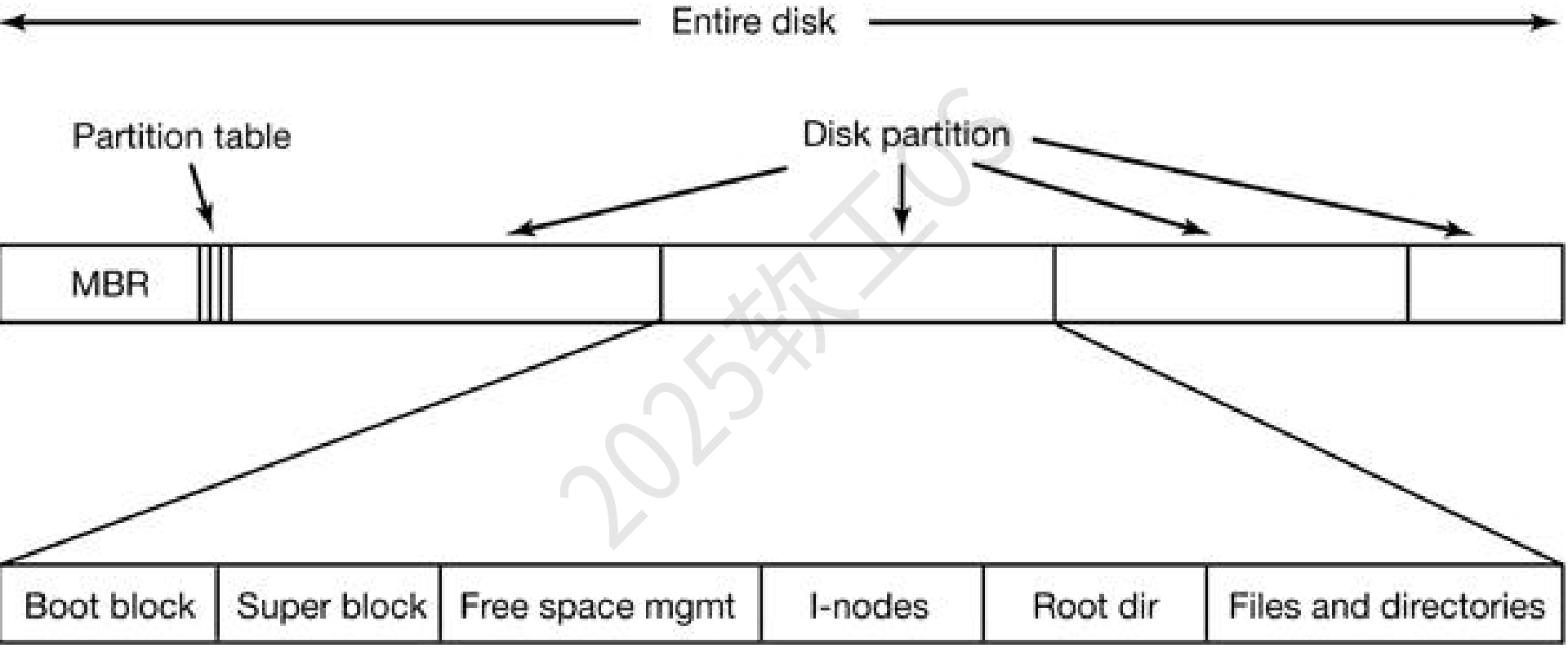
文件目录

- 文件目录本身就是文件，但属于系统文件，用户不可直接操作
- 单级文件目录系统
  - 仅有一个目录
  - 所有文件都在此目录下
  - 世界上第一台超级计算机 CDC 6600 采用此法
- 单级文件目录的优缺点
  - 优点：简单易实现
  - 缺点：查找速度慢、不允许重名

- 多级目录（树型结构目录）
- 路径名
  - 每个文件有唯一的路径，c:\Users\, or /root/
  - 绝对路径与相对路径
- 当前目录（工作目录、working directory)
  - 每个进程都有自己的工作目录
  - 相对路径，. 与 ..
- 目录操作
  - 搜索文件、创建文件、删除文件、遍历目录、重命名文件
- 如何访问目录中特定的节点?
- 如何存储目录树?

文件系统实现


J_CENA_X64F (E:) 属性
常规
工具
硬件
共享
ReadyBoost
自定义
J_CENA_X64F
类型: U 盘
文件系统: FAT32
已用空间: 4,179,050,496 字节 3.89 GB
可用空间: 26,794,573,824 字节 24.9 GB
容量: 30,973,624,320 字节 28.8 GB
驱动器 E:
确定
取消
应用(A)
文件太大
对于目标文件系统，文件"MatLab 2014b Win64.iso"过大。
MatLab 2014b Win64.iso
类型: UltraISO 文件
大小: 6.54 GB
修改日期: 2015/3/19 14:25
为所有当前项目执行此操作(A)
跳过(S)
取消
简略信息


Entire disk
Partition table
Disk partition
MBR
Boot block
Super block
Free space mgmt
I-nodes
Root dir
Files and directories

文件的实现

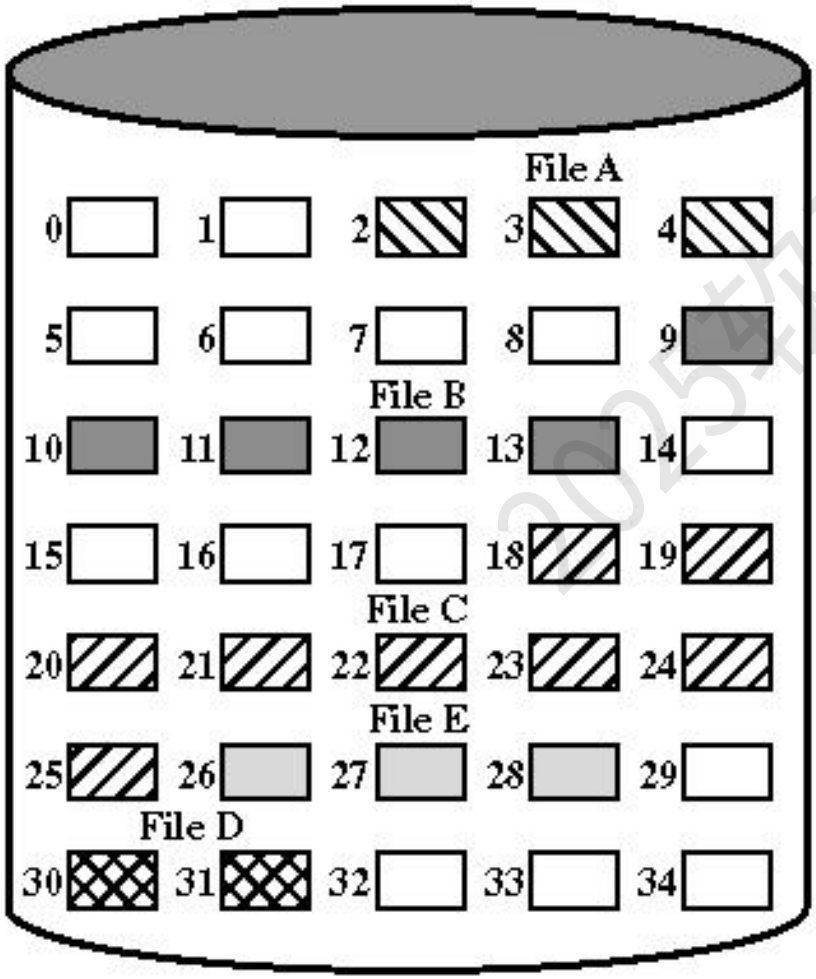
如何为文件分配磁盘空间？

注意在不同实现方式下，如何删除文件？

（若无特殊说明，则假设使用bitmap管理空闲盘块）

连续分配

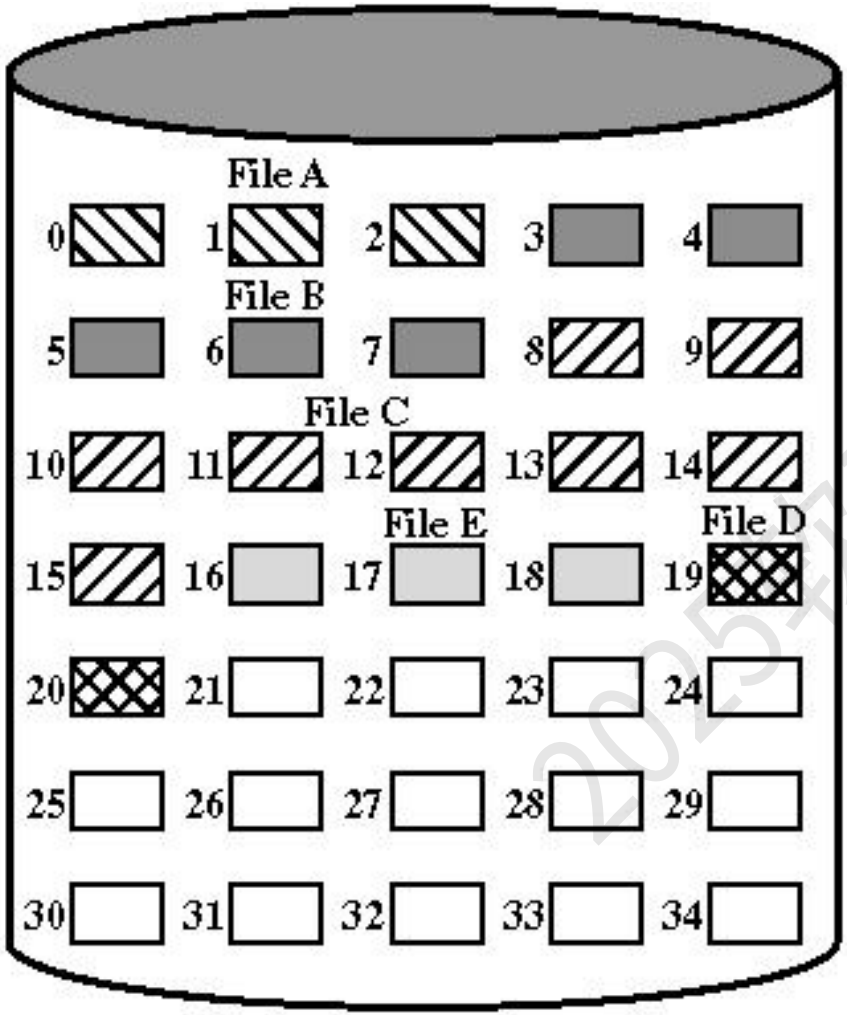
- 每一个文件占用一个连续的磁盘块的集合,从而成为一种物理上的顺序文件



File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

连续分配方法会造成外部碎片



File Allocation Table

File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

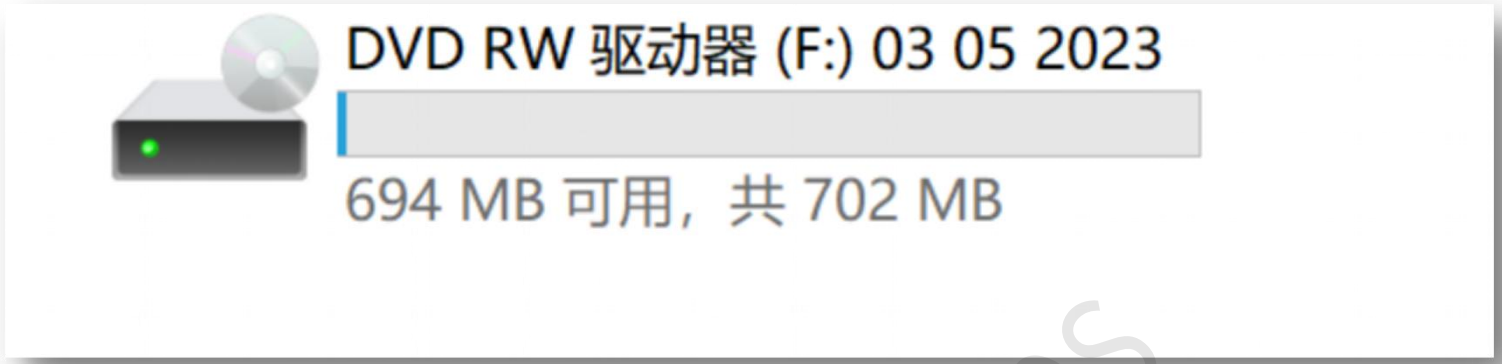
缓解碎片问题的策略：紧凑

连续分配的优缺点

- 优点
  - 简单，一个文件只需要记录起始块号和长度
  - 访问速度快--读连续块的效率高
  - 支持随机存取(Random access)
- 缺点
  - 文件不能动态增长
  - 浪费空间：几次动态存储分配后就出现碎片问题
    - 可采用“紧凑”来解决

由于需要提前声明文件大小，连续分配曾经一度被抛弃，后来出现一种特殊的存储介质，这种介质上的数据事先知道大小，并且不会动态删除，于是连续分配方法又被应用。知道这种介质是什么吗？

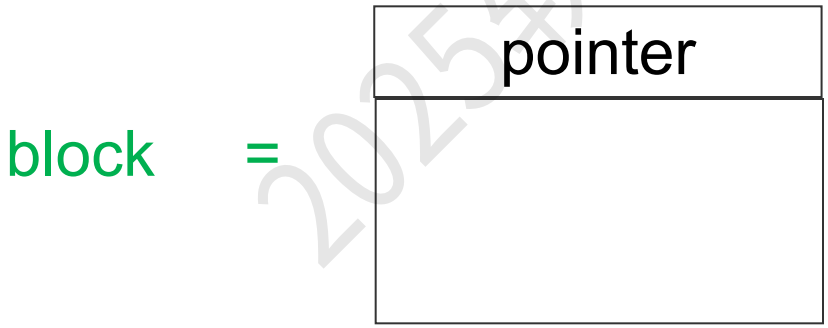
“history often repeats itself”

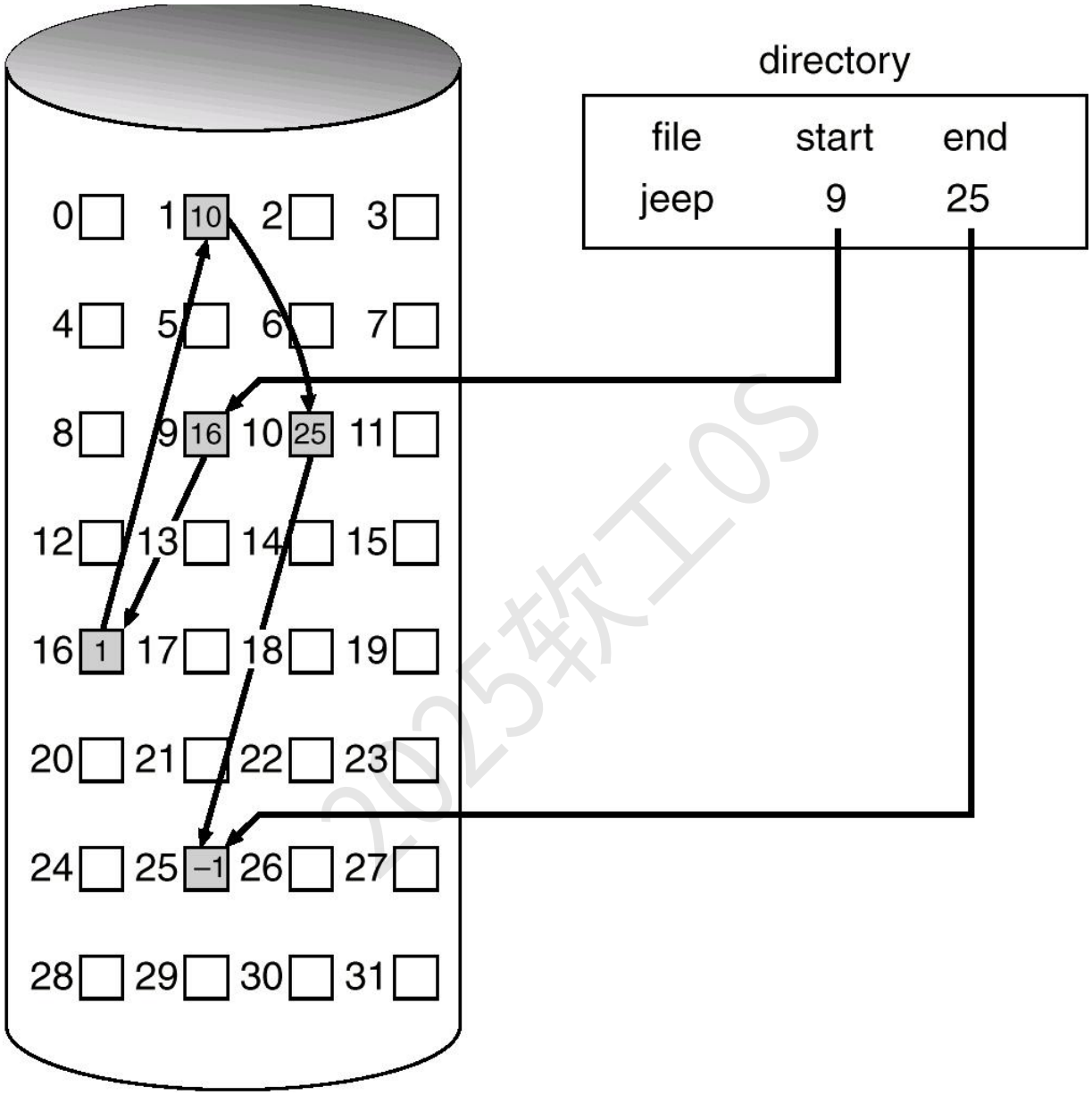


为什么一个文件没有，但是可用空间少了8MB?

（隐式）链接分配

- 文件用链表表示
- 每个磁盘块存储下一个磁盘块的地址




directory
file
start
end
jeep
9
25
0
1
10
2
3
4
5
6
7
8
9
16
10
25
11
12
13
14
15
16
1
17
18
19
20
21
22
23
24
25
−1
26
27
28
29
30
31

- 如何删除文件？

隐式链接分配的优缺点

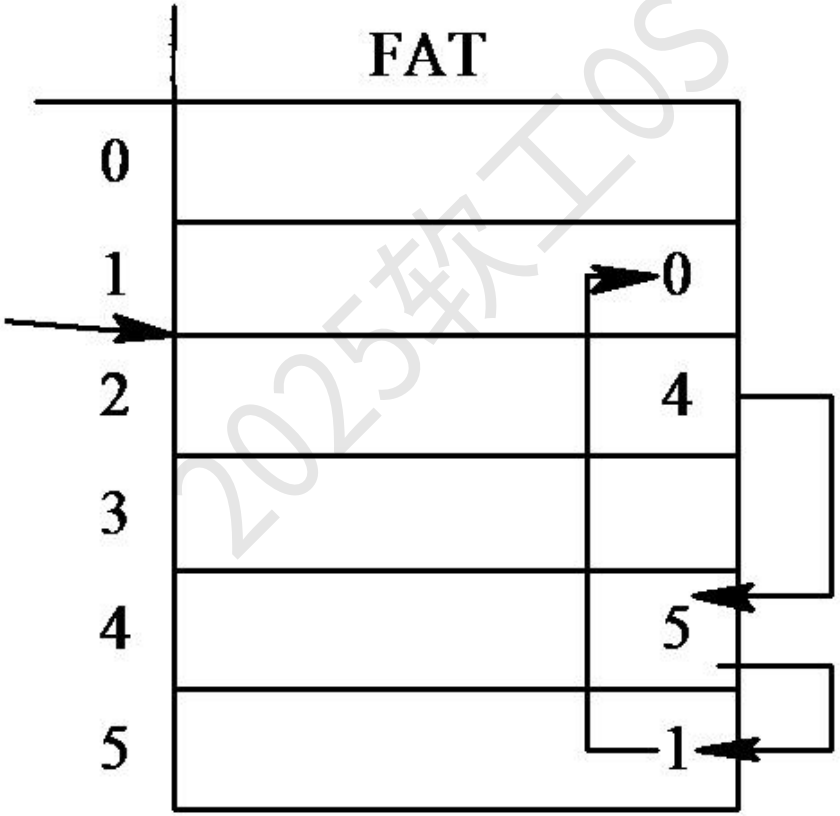
- 优点
  - 消除外部碎片，磁盘利用率高
  - 文件大小可动态增长，不必事先声明文件大小
- 缺点（哪个是致命缺点？）
  - 指针占用空间，致可用字节数不是2的幂次方（会造成什么问题？）
  - seek操作：访问特定盘块需要多次读盘

课程回顾

- 文件系统接口
  - 文件
    - 普通文件（字节流）、特殊文件（目录、设备，等）
    - 操作：创建、删除、读、写、设置位置等
  - 目录
    - 一种特殊的文件，只有文件系统能访问
    - 目录的操作？搜索文件、创建文件、删除文件、遍历目录、创建目录
- 文件的实现
  - 连续分配（文件无法动态增长、碎片问题）
  - 隐式链接分配（seek操作、删除操作费时）

显式链接

- 文件分配表FAT，MS-DOS采用的技术
- 整个磁盘仅一张



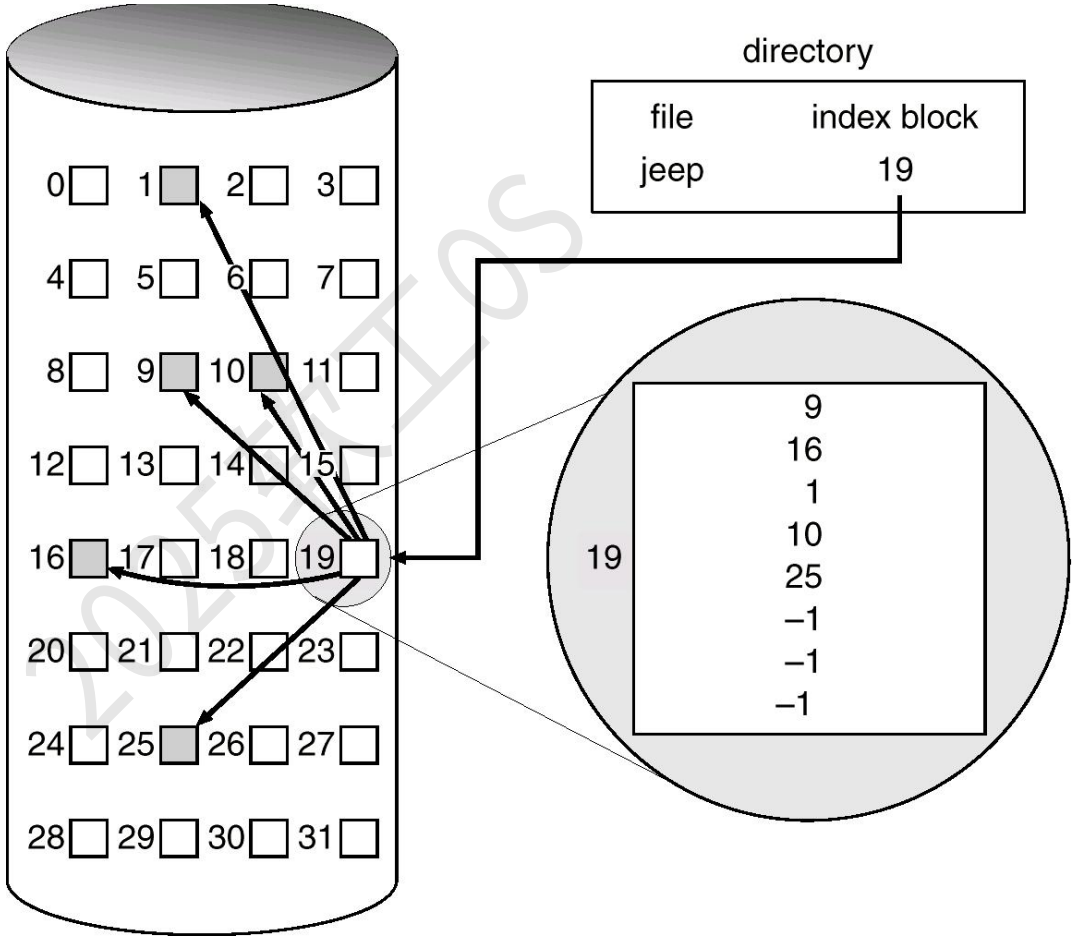
- 优点
  - 隐式链接方式的所有优点，外加
  - 查找指定盘块的速度比隐式链接快（文件分配表一次读入内存，常驻内存）
- 缺点
  - 磁盘较大时，文件分配表大，占用内存过多
- 例子：200GB硬盘，block大小为1KB。问
  - FAT表需要多少项?
  - 若描述每个FAT表项需要的比特数为32的整数倍，则FAT表需要消耗多少内存?
  - 缓解方案?
    - 放磁盘上，or，使用较大block，分别有何缺点?

例题

- 在某FAT16文件系统中，FAT表的每个表项用16位表示，每簇（即，块）64扇区，扇区的大小为512字节。
  - 该分区最大可为多少字节?
  - 其FAT占用多少存储空间?

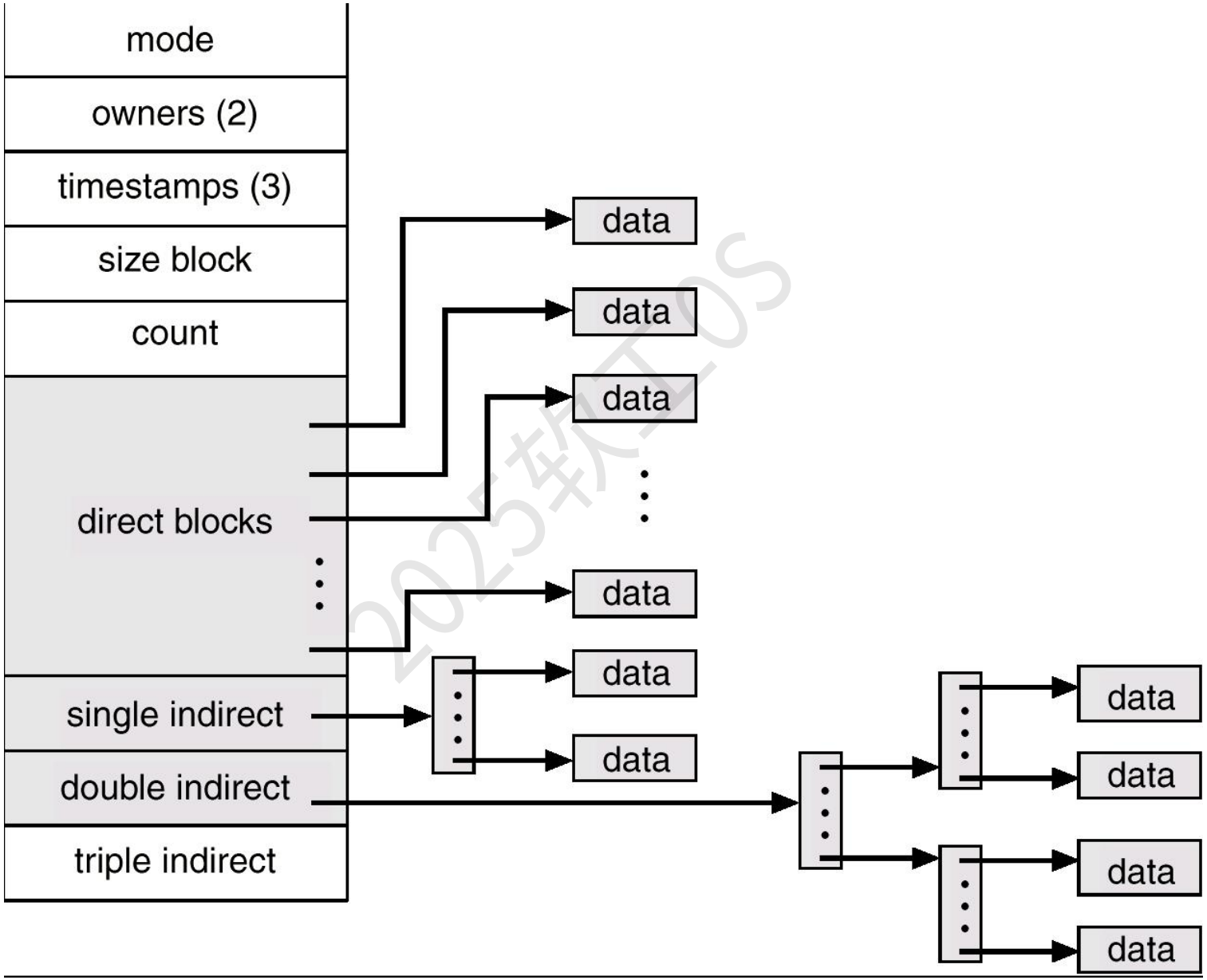
索引分配

- 节省内存（相对于FAT）

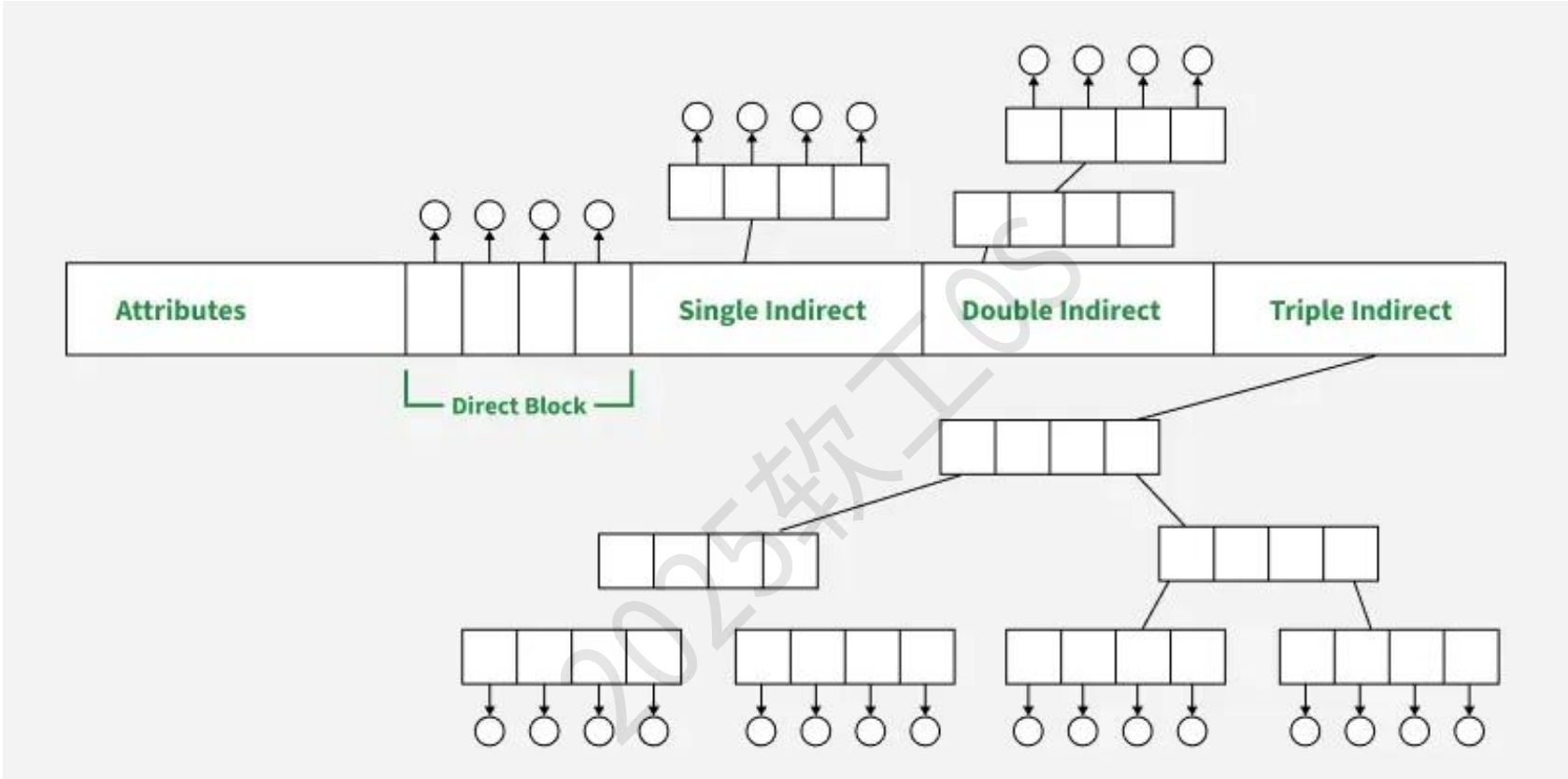


- 如何删除文件？

示例：Unix下的i-node



示例：Unix下的i-node



这张图跟前面的是一个意思，看懂一张即可

思考：在i-node系统中，对于任意盘块，能否从它自身的内容中判断它是triple indirect，double indrect，single indirect，还是direct block?

计算最大文件

- 假设块大小为512字节
- 直接地址有12个
- single indirect 地址1个
- double indirect 地址1个
- 地址长度均为32位
- 问：文件大小最大为多少?

以上就是文件的实现方式

请对比

- 内存分配
- 问题
  - 将内存分配给进程
- 分配方案
  - 连续分配
    - 外部碎片问题
  - 离散分配：分页存储管理
    - 页面大小的问题

- 磁盘分配
- 问题
  - 将磁盘分配给文件
- 分配方案
  - 连续分配
    - 外部碎片问题
  - 离散分配：链接分配，索引分配
    - 块大小的问题

总结：文件的实现

- 分配磁盘空间给文件
  - 连续分配（简单、效率高、需要提前声明文件大小，目前在光盘存储上使用）
  - 隐式链接分配，每个磁盘块中存下一个磁盘块的地址（文件可动态增长，无外部碎片，seek操作速度慢，无外存使用此方案）
  - 显式链接分配。将链接信息集中存放在一张表格中，命名为FAT，启动时一次读入内存。（文件可动态增长，无外部碎片，seek操作速度快。MS-DOS使用，目前FAT32还在使用）
  - 索引分配，用inode记录文件实际位置。（文件可动态增长，访问文件时只需载入inode）

目录如何实现？

目录管理的要求

- 按名存取：根据名字找到外存中的位置
  - 名字通常为ASCII字符串表示的文件位置，如：/home/xzhu/1.txt, C:\\Windows\1.txt
  - OS需要根据名字查找到文件在磁盘中的地址
  - 进程管理中是否有类似的需求?
- 文件属性
- 文件共享（同一个文件有不同名字）

按名存取

- 最简单的做法？维护一张名字地址映射表
  - 此处的名字包含目录地址
  - 单级目录时可以。（查找的复杂度是多少？）
  - 多级目录时。。。
- 一般的做法，将目录作为特殊的文件
  - 某个目录的文件中，记录了子目录文件、以及普通文件的地址（直接地址或间接地址）
  - 两个特殊目录: . 和 ..
  - 在FAT文件系统中，存盘块地址即可
  - 在索引文件系统中，存inode编号

目录查找（FAT）

- C:\\Windows\1.txt
- 找到驱动器C对应的目录文件地址
- 从目录文件中查找名为“Windows”的子目录
- 根据地址读盘块进入内存
- 从中查找名为“1.txt”的文件
- 找到盘块地址

目录查找（inode）

- /home/xzhu/1.txt
- 读入根目录的inode（如何知道inode编号？）
- 将根目录文件读入内存
- 从文件中查找名为“home”的子目录，找到对应的inode编号
- 读入inode（可能已在内存中）
- 读入目录文件
- 从文件中查找名为“xzhu”的子目录，找到inode编号
- 读入inode（可能已在内存中）
- 读入目录文件
- 从文件中查找名为“1.txt”的文件，找到inode编号
- 读入inode（可能已在内存中）
- 找到盘块地址！

文件属性存放在哪里?

两种实现方式

- 存放在目录条目中（windows）
  - 目录条目（directory entry）的例子

```
struct dient{
        char name[12];
        attr_t attributes;
        uint addr;
        ... }
```

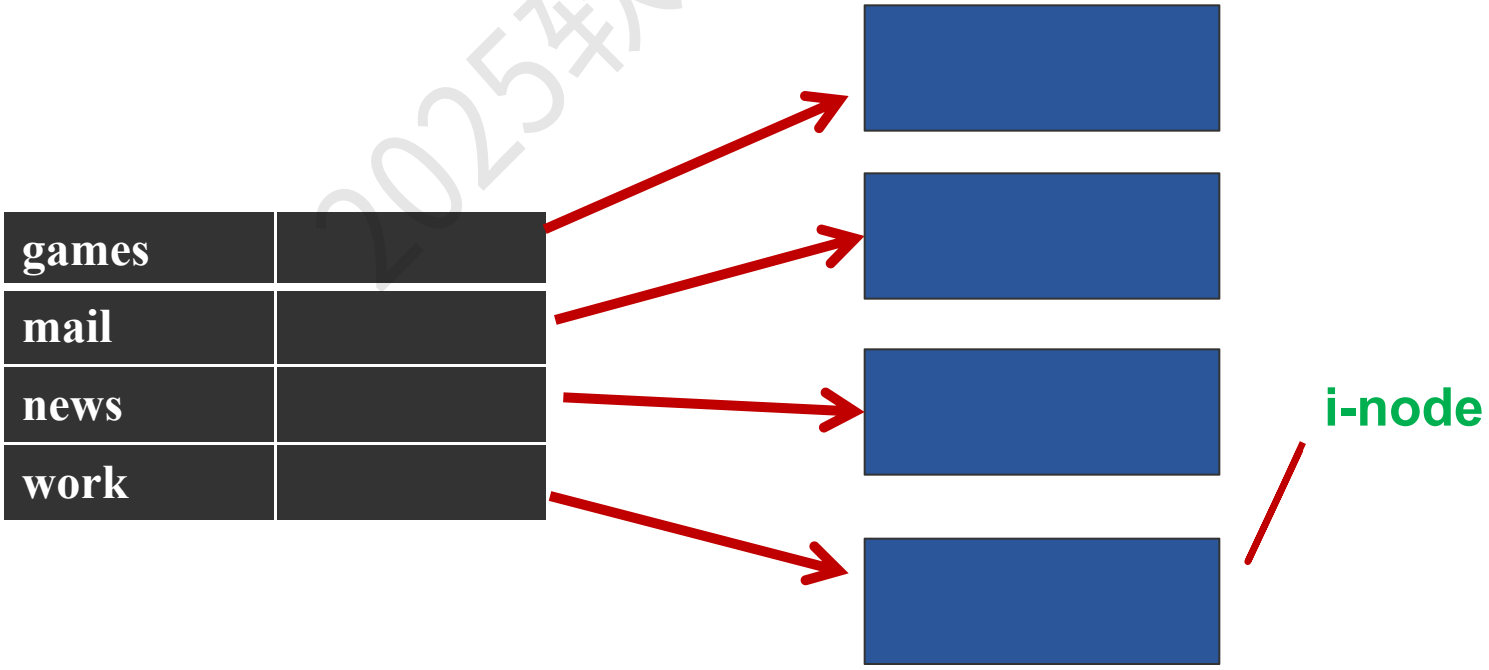
games	属性	地址
mail	属性	地址
news	属性	地址
work	属性	地址

两种实现方式

- 存放在i-node中（unix）
  - 目录条目的例子

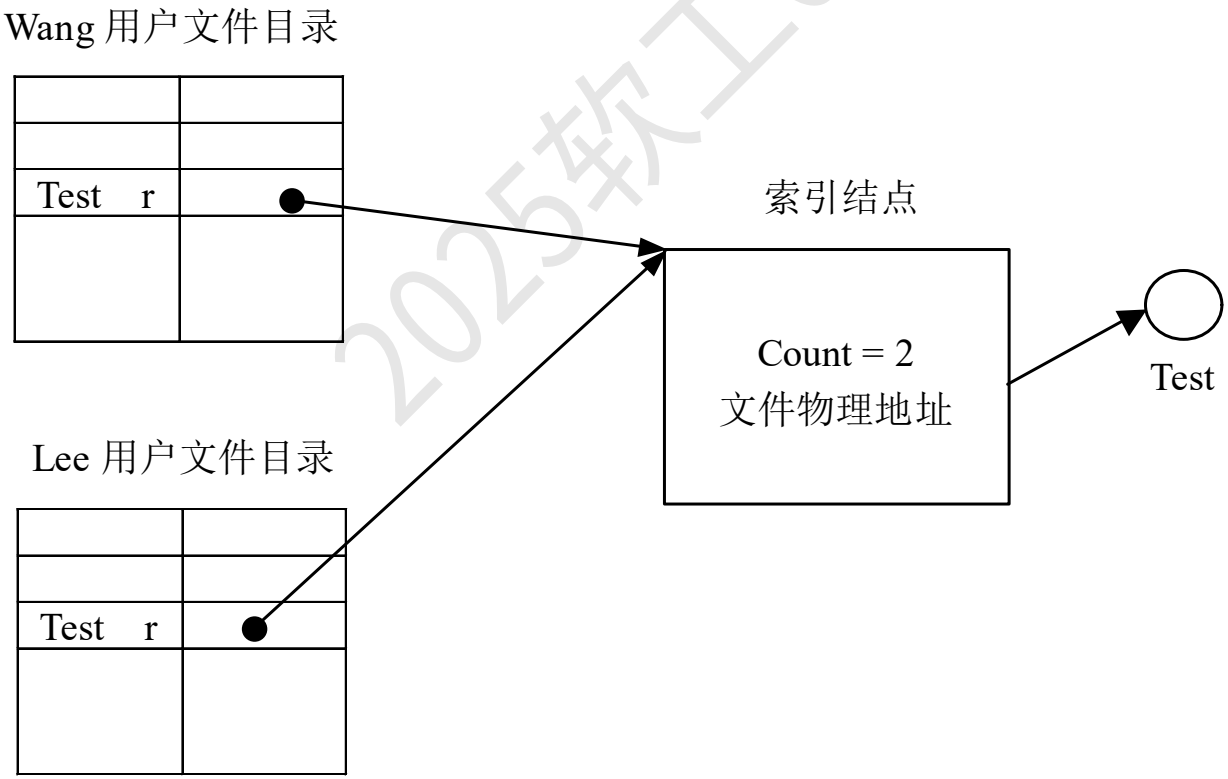
struct dient{ char name[12];

uint i-num;}

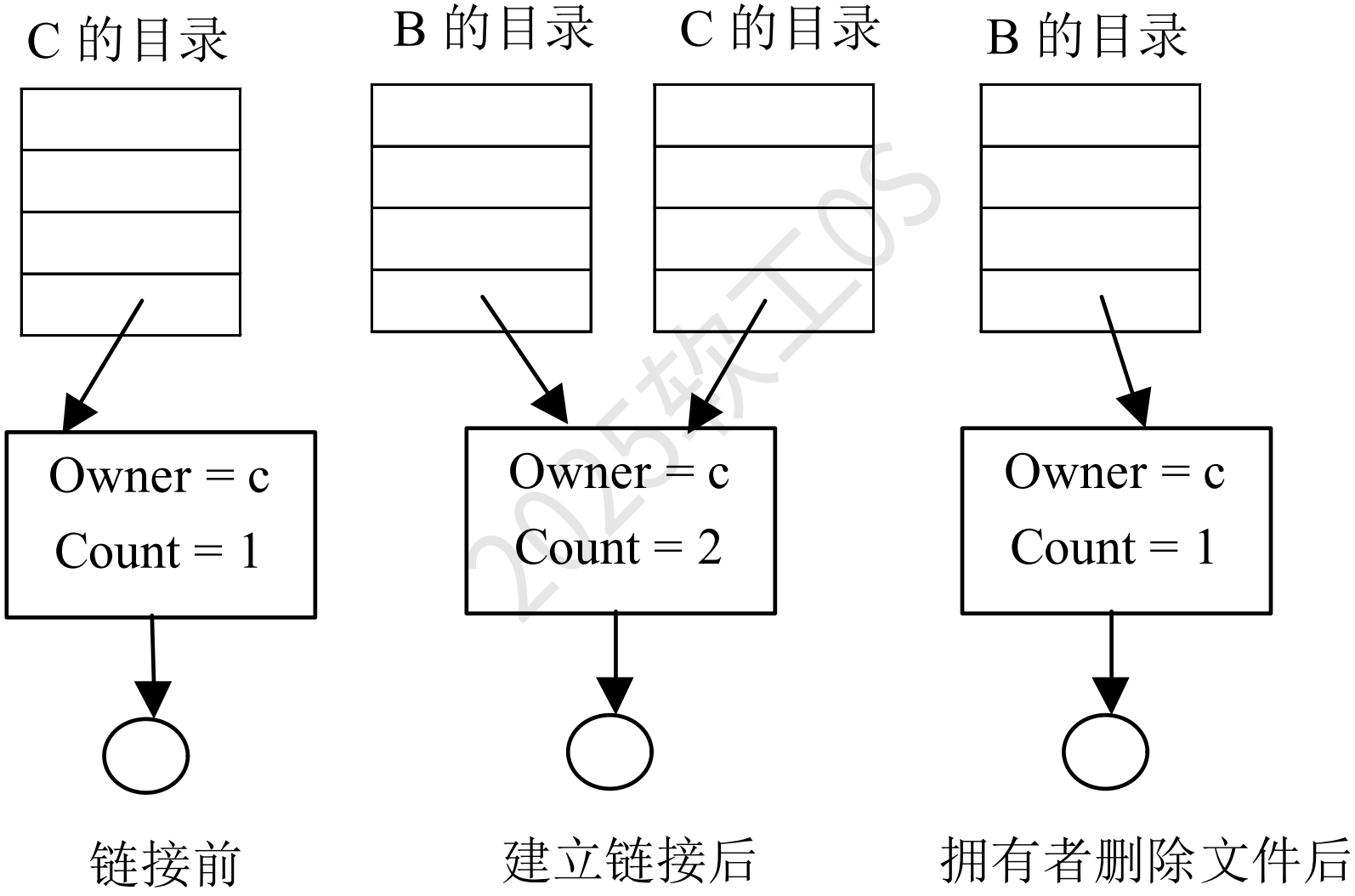


文件共享

- 同一台计算机上的多个用户共享同一个文件
  - 共享之后，树形结构变为有向无环图
- 基于索引节点的共享方式



●缺点：拥有者无法彻底删除文件



- 基于符号链接的共享方式
  - 创建一个LINK类型的文件A
  - 此文件仅包含文件B的路径地址（非盘块地址）
  - OS 自动将访问A的请求转为访问B
  - 类似于 windows 快捷方式
  - 使用案例：
    - Linux下，将需要开机执行的脚本的符号链接放入init.d目录下
- 优点
  - 拥有者可彻底删除文件
- 缺点
  - 多次读盘

- **思考：**
  - 树莓派制作SD卡，要求将操作系统刷到SD卡上，而不是解压到SD卡上。
  - 但是，通过这两种操作后的两张SD卡，插到windows机器上，双击打开，显示的内容完全一样，甚至用文件对比工具也分析不出区别。
  - 为何刷上去的可以启动，而解压的不可以启动?
  - 用你所学解释为何一个可以启动，一个不可以。

日志文件系统

从删除文件说起

- 如何删除一个文件?
  - 从目录文件里删除
  - 将 inode归还（unix）
  - 将磁盘空间归还（里面的数据呢。。。？）
  - 正常情况下，上述几个操作的次序无所谓
- 如果在此操作过程中突然断电。。。
  - 如果只把目录里的条目删除。。。
  - 如果次序被打乱，只归还了i-node。。。
  - 如果次序被打乱，只释放了磁盘空间。。。

解决思路

- 基本步骤
  - 操作之前先将操作记录到磁盘
  - 然后执行操作
  - 执行完操作后，删除磁盘中的记录
- 如何预防前述错误的?
  - 系统启动时，如果有记录没被删除，重新执行记录中的操作，然后将记录删除
  - 注意这些操作具有idempotent特性（幂等）
- 日志文件系统(journaling file system)采用此方案（NTFS、ext3等）

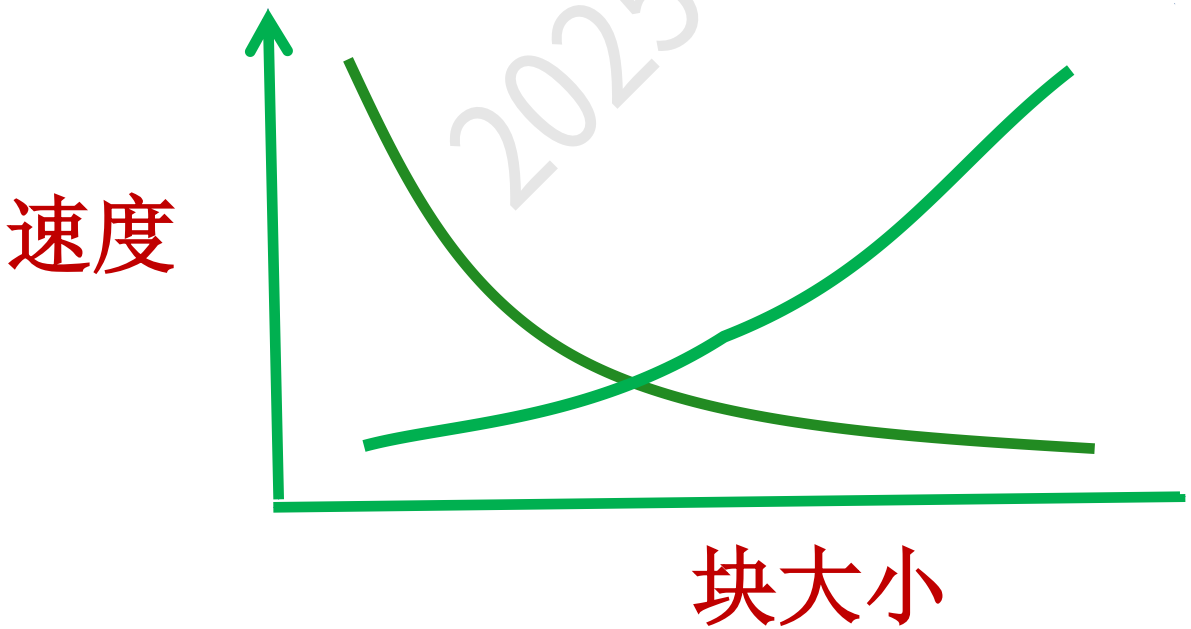
总结

- 文件的实现
  - 分配磁盘块给文件
  - 现在常用的为显示链接分配、索引分配
- 目录的实现
  - 目录也是文件
  - 文件属性的存放
  - 长文件名的支持
  - 目录查找技术
  - 文件共享
- 日志文件系统

（空闲）磁盘管理方法

背景：磁盘的分块管理

- 磁盘按块（block）进行分配
- 块如果过大，则内部碎片较大，导致。。。
- 块如果太小，则文件读取太慢（why？）
- 到底应该设多大？如果让你调研，你打算怎么做？



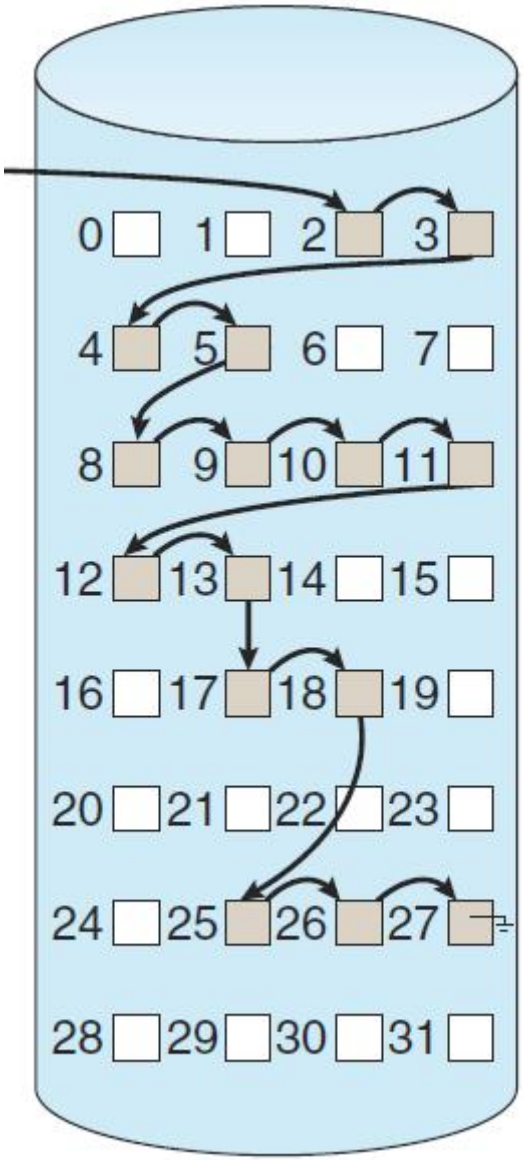
Google File System

Google's design philosophy is readily evident in the architecture of the Google File System (GFS) [3]. GFS serves as the foundation of Google's cloud software stack and is intended to resemble a Unix-like file system to its users. Unlike Unix or Windows file systems, GFS is optimized for storing very large files (> 1 GB) because Google's applications typically manipulate files of this size. One way that Google implements this optimization is by changing the smallest unit of allocated storage in the file system from the 8 KB block size typical of most Unix file systems to 64 MB. Using a 64 MB block size (Google calls these blocks *chunks*) results in much higher performance for large files at the expense of very inefficient storage utilization for files that are substantially smaller than 64 MB.

根据应用需求优化文件系统

空闲磁盘管理方法：链表法

- 每个空闲块存下一个空闲块的编号
  - 不占空间
  - 若要遍历，则读盘次数多
- 或者，将链表信息集中存放到一个空闲块
  - 不占空间
  - 遍历速度快
- 如何分配?
- 如何回收?



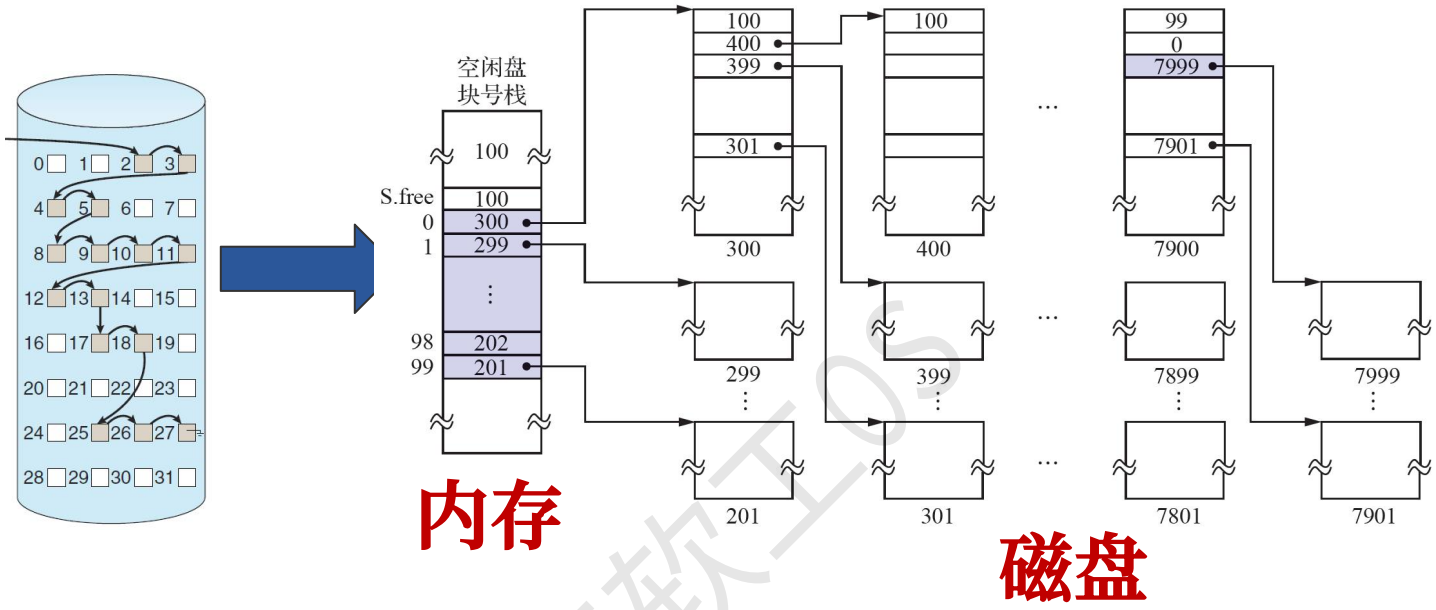
空闲磁盘管理方法：位示图法

- 每个块用一个比特表示
  - 为0表示空闲
  - 为1表示占用

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	1	0	0	0	1	1	1	0	0	1	0	0	1	1	0
2	0	0	0	1	1	1	1	1	1	0	0	0	0	1	1	1
3	1	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0
…																
16																

- 需要占用多少磁盘空间?
  - 设磁盘有1TB，每块大小为4KB，则位示图需要占用多少磁盘空间?
- 如何分配?
- 如何回收?

空闲磁盘管理方法：成组链接法



- 每个空闲盘块不要只存一个地址，存满！
- 保留一个地址存储下一个类似结构（s.free(0)）。
- 增加一个域记录当前存储了多少空闲块地址(N)。
- “空闲盘块号栈”在内存中。

讨论：哪种方法好？位示图？链表？

更多话题

- 提高磁盘IO的速度
  - 磁盘高速缓存
  - 提前读
  - 延迟写
  - 优化物理块的分布
- 提高磁盘可靠性
  - 备份
- 文件保护
  - 域、访问矩阵

文件系统总结

- 文件系统基本情况
  - 文件名、扩展名、文件类型、文件的逻辑结构、文件属性、文件目录、文件的操作
- 实现文件系统
  - 文件的实现：连续、离散（隐式链接、显式链接、索引）
  - 目录的实现
  - 文件共享
  - 日志文件系统
  - 空闲磁盘管理方法：链表法、位示图法

自学

案例：xv6文件系统

xv6磁盘组织结构

```
// Block 0 is unused.
// Block 1 is super block.
// Blocks 2 through sb.ninodes/IPB hold inodes.
// Then free bitmap blocks holding sb.size bits.
// Then sb.nblocks data blocks.
// Then sb.nlog log blocks.

#define ROOTINO 1  // root i-number
#define BSIZE 512  // block size

// File system super block
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
};
```

```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

dinode中为何不存储i-number？

```
// Directory is a file containing a sequence of dirent structures.
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

文件File （file.h）

```c
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

struct file内核结构体；file可以指向inode，也可以指向pipe

file.c > ...

```c
struct devsw devsw[NDEV];
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;

void
fileinit(void)
{
  initlock(&ftable.lock, "ftable");
}

// Allocate a file structure.
struct file*
filealloc(void)
{
  struct file *f;

  acquire(&ftable.lock);
  for(f = ftable.file; f < ftable.file + NFILE; f++){
    if(f->ref == 0){
      f->ref = 1;
      release(&ftable.lock);
      return f;
    }
  }
  release(&ftable.lock);
  return 0;
}
```

预定义NFILE个file结构体常驻内存

用户打开文件时用filealloc申请file结构体

文件描述符File Descriptor是什么？

```
38  struct proc {
39    uint sz;                     // Size of process memory (bytes)
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for this process
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52  };
```

PCB中有一个域指向一系列文件对象；FD是此数组的下标

标准流程：创建文件、为文件分配FD

```
423 int
424 sys_pipe(void)
425 {
426   int *fd;
427   struct file *rf, *wf;
428   int fd0, fd1;
429
430   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
431     return -1;
432   if(pipealloc(&rf, &wf) < 0)
433     return -1;
434   fd0 = -1;
435   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
436     if(fd0 >= 0)
437       myproc()->ofile[fd0] = 0;
438     fileclose(rf);
439     fileclose(wf);
440     return -1;
441   }
442   fd[0] = fd0;
443   fd[1] = fd1;
444   return 0;
445 }
446
```

sysfile.c

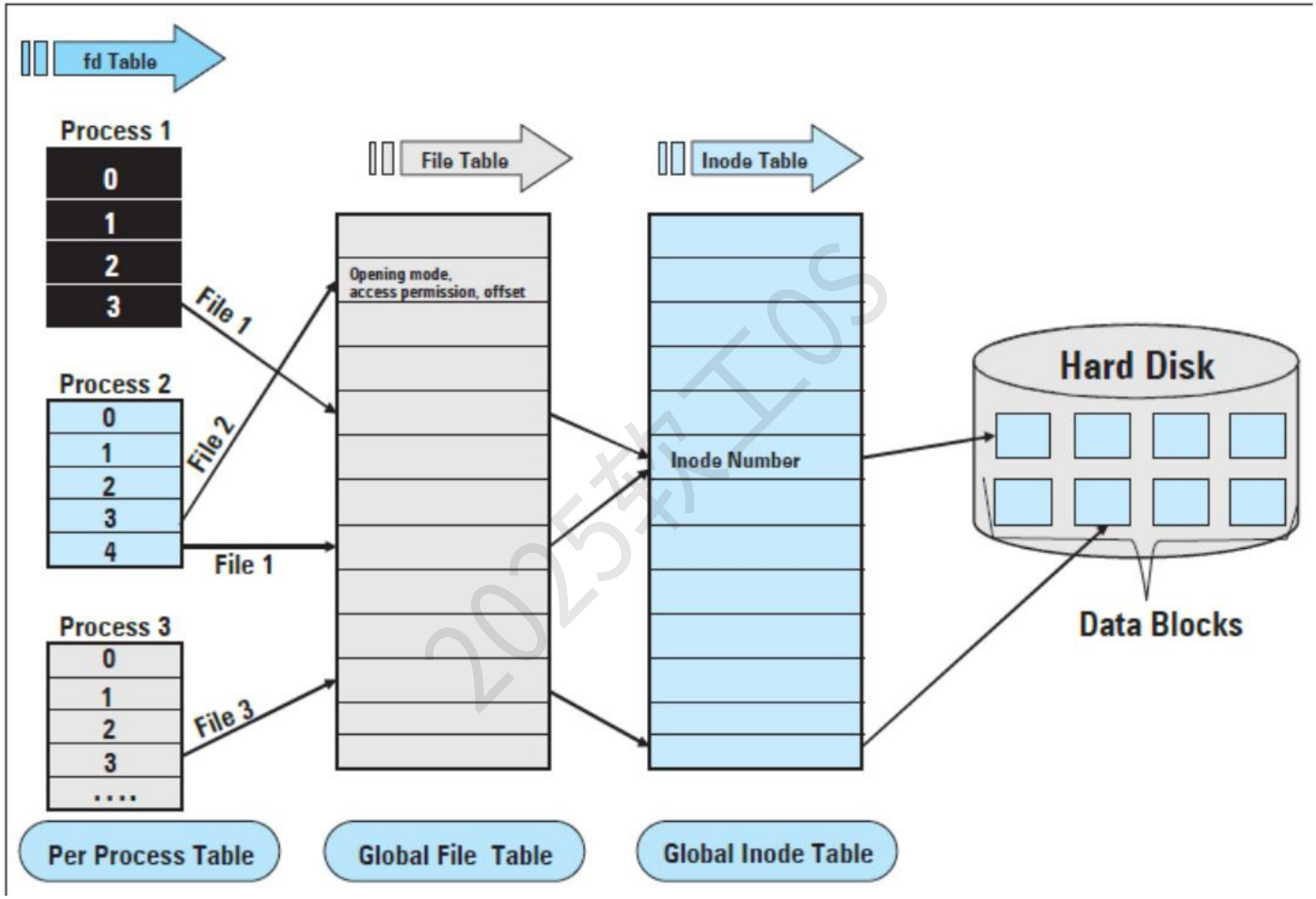
```
38  // Allocate a file descriptor for th
39  // Takes over file reference from ca
40  static int
41  fdalloc(struct file *f)
42  {
43    int fd;
44    struct proc *curproc = myproc();
45
46    for(fd = 0; fd < NOFILE; fd++){
47      if(curproc->ofile[fd] == 0){
48        curproc->ofile[fd] = f;
49        return fd;
50      }
51    }
52    return -1;
53  }
```

再看fork

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

```
for(i = 0; i < NOFILE; i++)
  if(curproc->ofile[i])
    np->ofile[i] = filedup(curproc->ofile[i]);
```

逐个FD复制，但指向了同样的File


fd Table
Process 1
0
1
2
3
File 1
Process 2
0
1
2
3
4
File 2
File 1
Process 3
0
1
2
3
....
File 3
File Table
Opening mode,
access permission, offset
Inode Table
Inode Number
Hard Disk
Data Blocks
Per Process Table
Global File Table
Global Inode Table