操作系统

Operating Systems

夏文 副教授

xiawen@hit.edu.cn

哈尔滨工业大学（深圳）

2019年12月

第9章 文件系统

主要内容

9.1 文件的目录结构

9.2 文件系统的实现

9.3 EXT2文件系统实现

9.4 Windows的FAT文件系统实现

思考几个小问题

- 拷贝一个1GB的文件 vs 拷贝1M个1KB的小文件？
- 删除一个1GB的文件 vs 删除1M个1KB的小文件？
- 无数个小文件总大小和占用磁盘空间大小不一致？
- 磁盘寻道和旋转耗时，优化方法？
- 同时剪切很多文件，那种时间快？：
  - 1）存放到当前磁盘分区某个文件夹中；
  - 2）存放到其他磁盘分区某个文件夹中
- 针对磁盘的分区或U盘进行格式化是干什么？
- 磁盘碎片清理是干什么？

9.1 文件的目录结构

- 文件的引入回顾
- 文件盘块的三种分配方式回顾
- 文件的目录结构

Disk VS Memory

	Disk	MLC NAND Flash	DRAM
Smallest write	sector	sector	byte
Atomic write	sector	sector	byte/word
Random read	8 ms	3-10 μs	50 ns
Random write	8 ms	9-11 μs*	50 ns
Sequential read	100 MB/s	550–2500 MB/s	$>$ 10 GB/s
Sequential write	100 MB/s	520–1500 MB/s*	$>$ 10 GB/s
Cost	$0.03/GB	$0.32/GB	$10/GiB
Persistence	Non-volatile	Non-volatile	Volatile

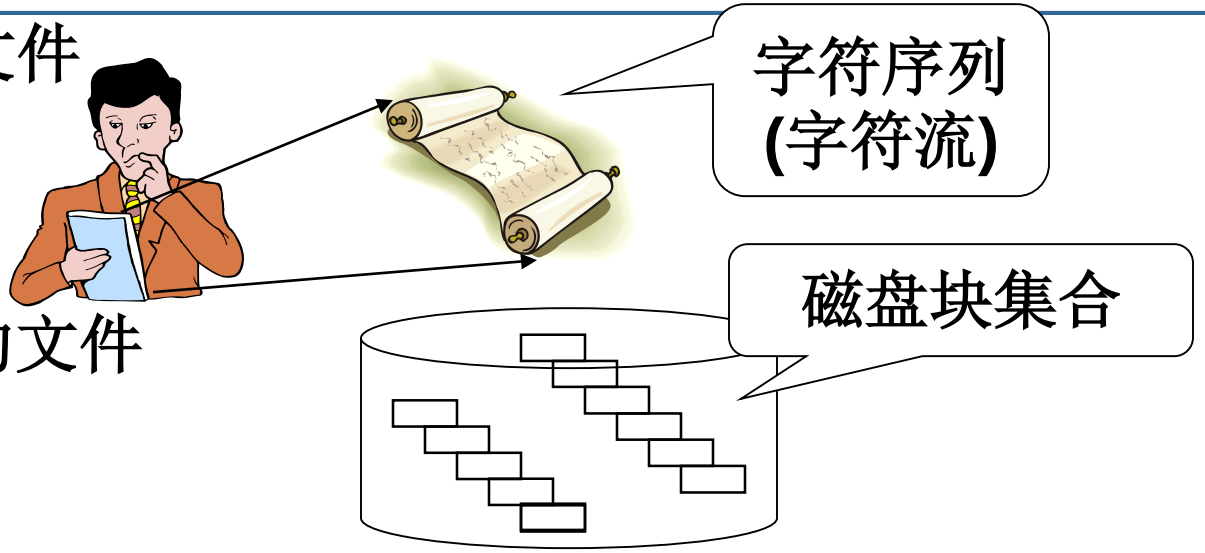
*Flash write performance degrades over time

几个发展趋势

- **Disk bandwidth and cost/bit improving exponentially**
  - Similar to CPU speed, memory size, etc.
- **Seek time and rotational delay improving *very* slowly**
  - Why? require moving physical object (disk arm)
- **Disk accesses a huge system bottleneck & getting worse**
  - Bandwidth increase lets system (pre-)fetch large chunks for about the same cost as small chunk.
  - Trade bandwidth for latency if you can get lots of related stuff.
- **Desktop memory size increasing faster than typical workloads**
  - More and more of workload fits in file cache
  - Disk traffic changes: mostly writes and new data
- **Memory and CPU resources increasing**
  - Use memory and CPU to make better decisions
  - Complex prefetching to support more IO patterns
  - Delay data placement decisions reduce random IO

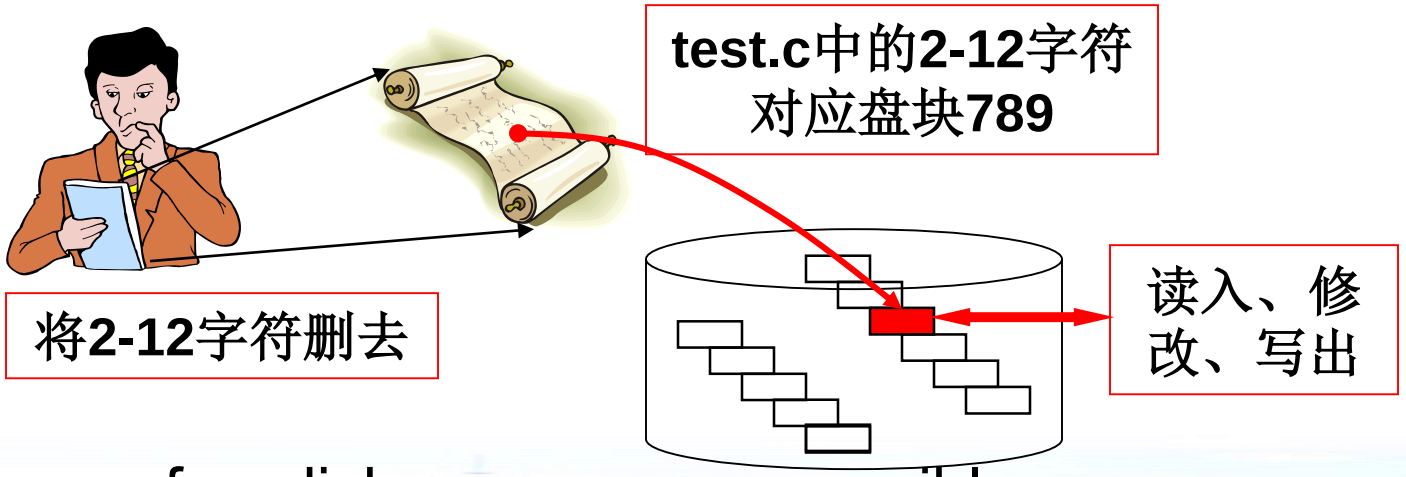
文件: named bytes on disk

- 用户眼里的文件



- FS眼中的文件

❿ **FS的任务: 文件名字和偏移——>磁盘块地址**



FS: to have as few disk accesses as possible
& have minimal space overhead (group related things)

File 类型

Contiguous logical address space

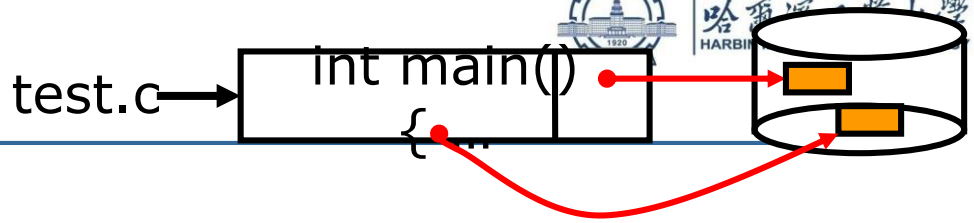
Types:

- Data
  - numeric
  - character
  - binary
- Program

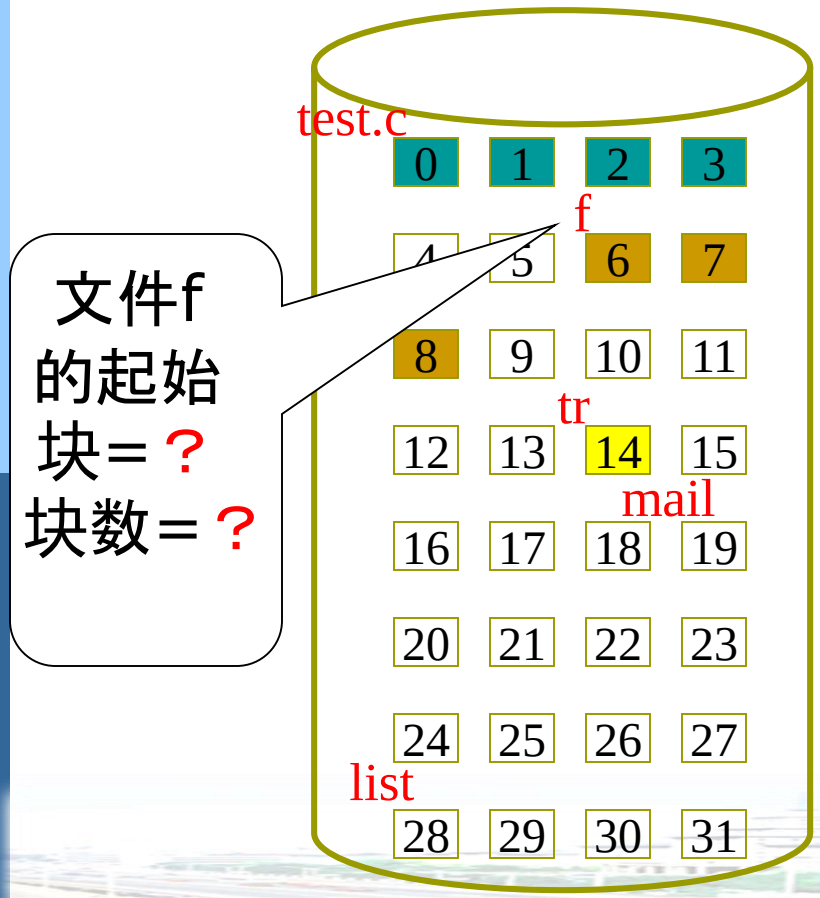
Contents defined by file's creator

- Many types
  - Consider **text file, source file, executable file**

文件的实现



- 文件抽象概念的实现关键：描述这一映射关系
  - 文件实现1：物理盘块连续分配



- 需存放什么信息？

  起始盘块和盘块个数

- 存放在哪里？

  文件描述信息节点中

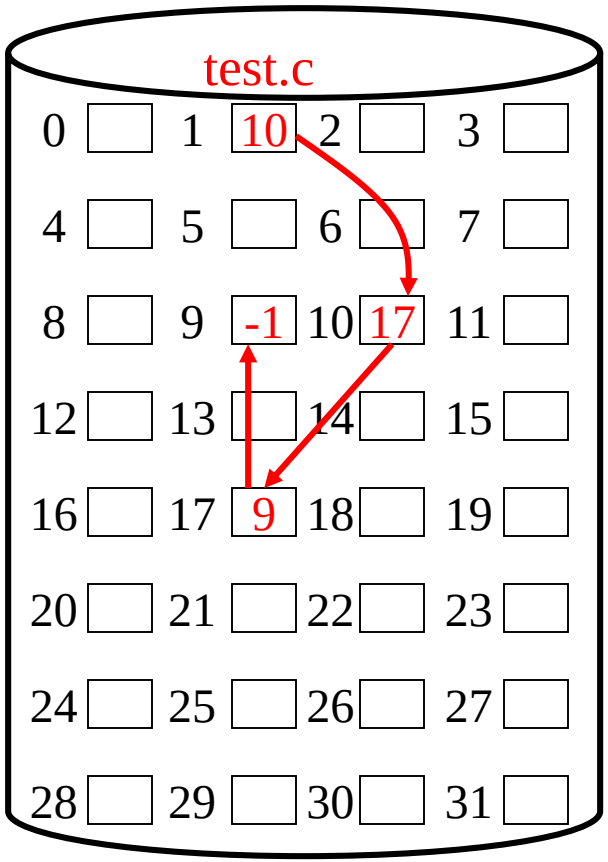
名字很多：FCB, File Header等

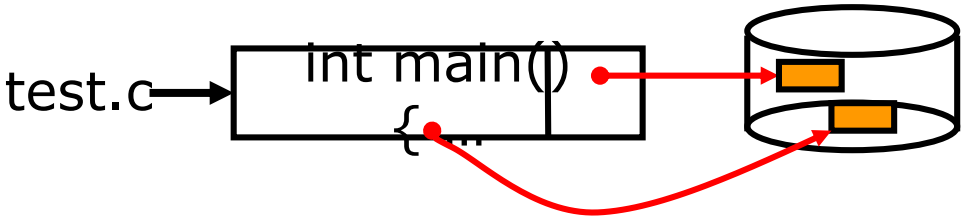
test.c的File Header

文件名	始址	块数
test.c	0	4

- 优点简单快速，缺点？和连续内存分配相比一下！

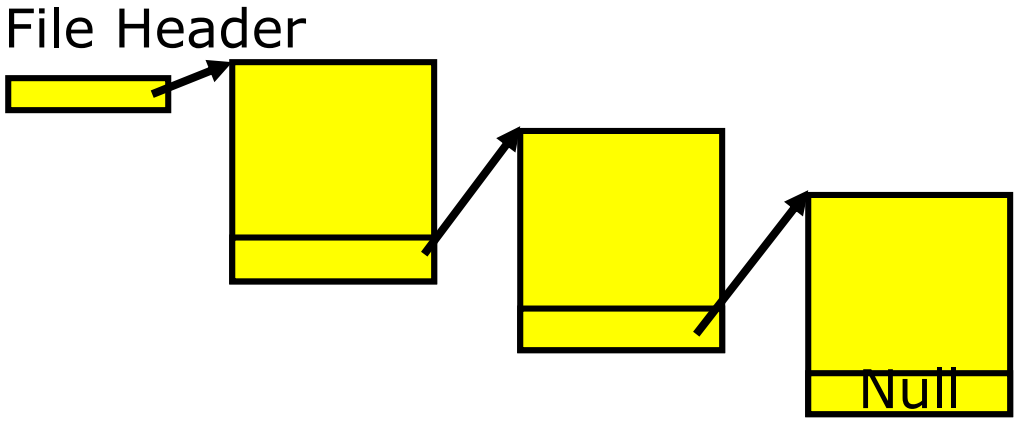
文件实现2: 链式分配





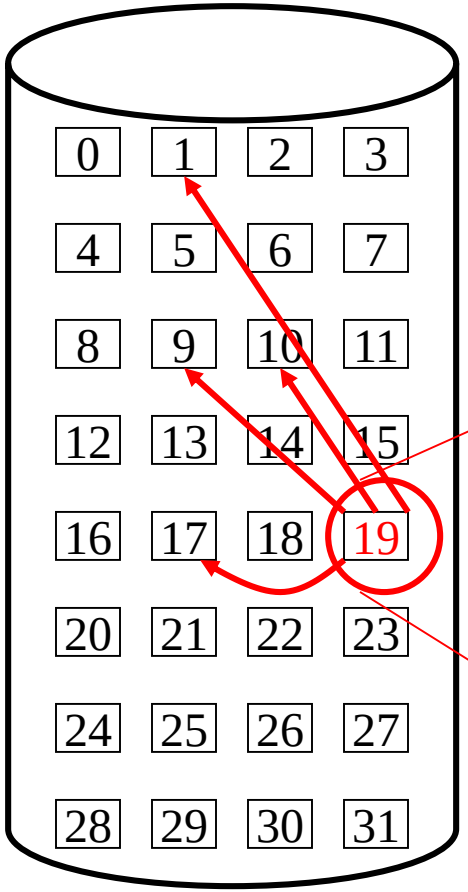
test.c的File Header

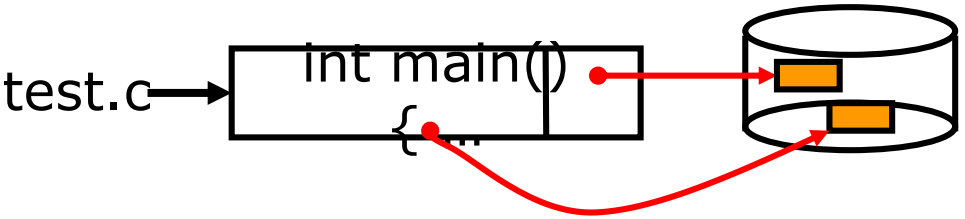
文件名	始址
test.c	1



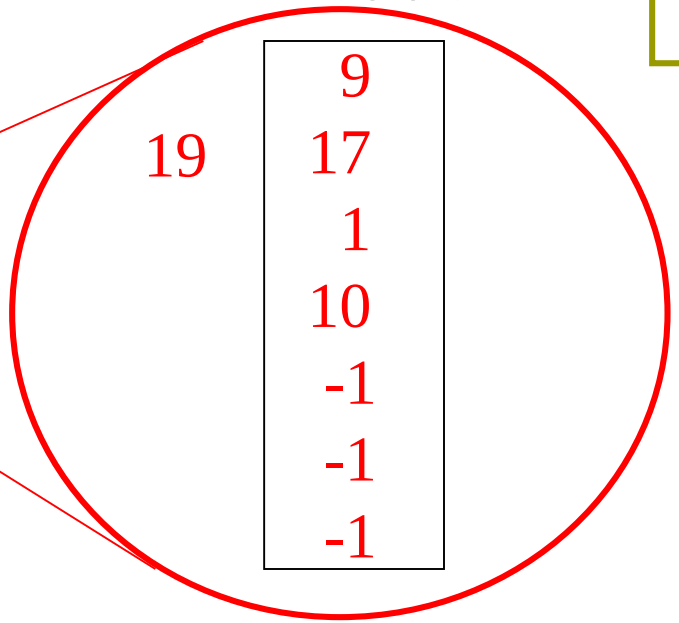
- 优点：文件长度增减容易
- 缺点：顺序访问、效率低

文件实现3: 索引分配





test.c的File Header



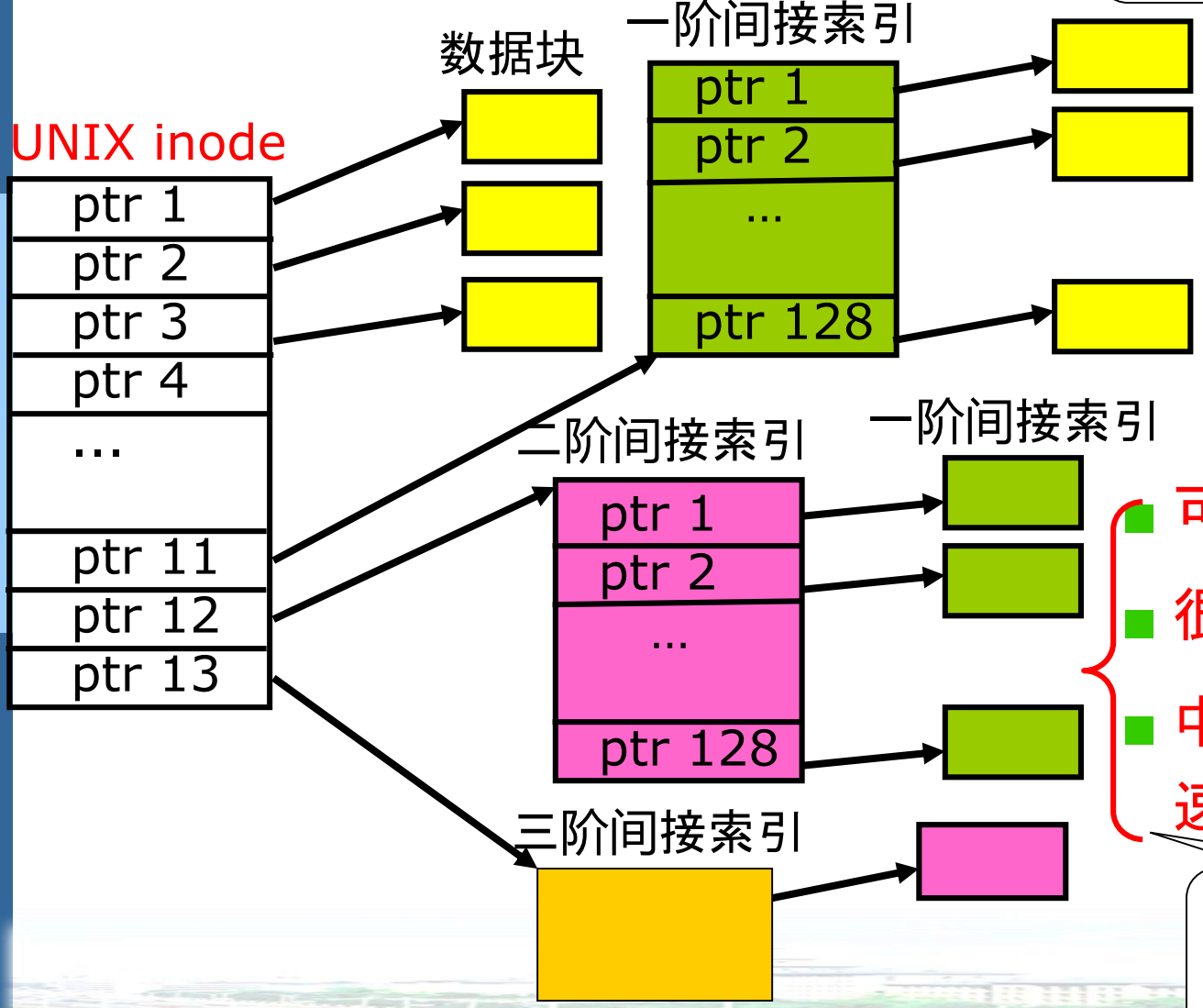
文件名	索引块
test.c	19

在实际操作系统中较常用!

- 优点：是连续和链式分配的有效折衷

UNIX的索引节点(inode)

根据名字就知道是索引分配!

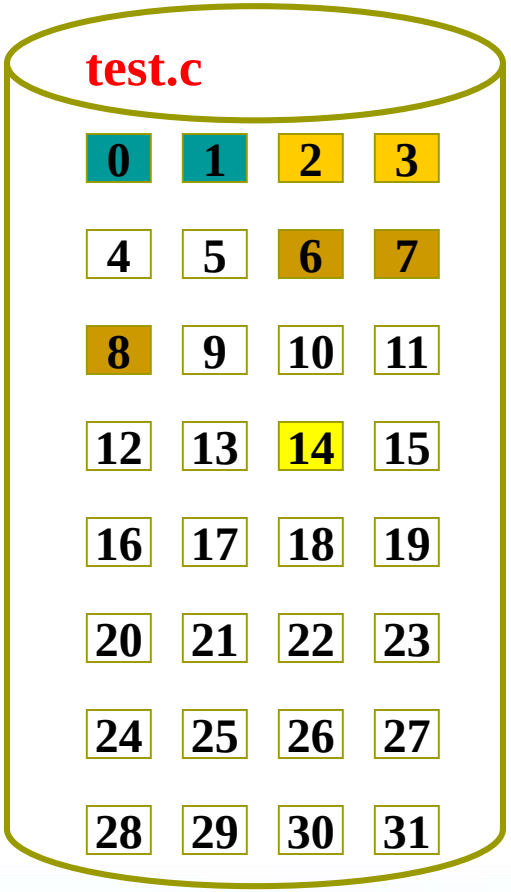


- 可以表示很大的文件
- 很小的文件高效访问
- 中等大小的文件访问速度也不慢!

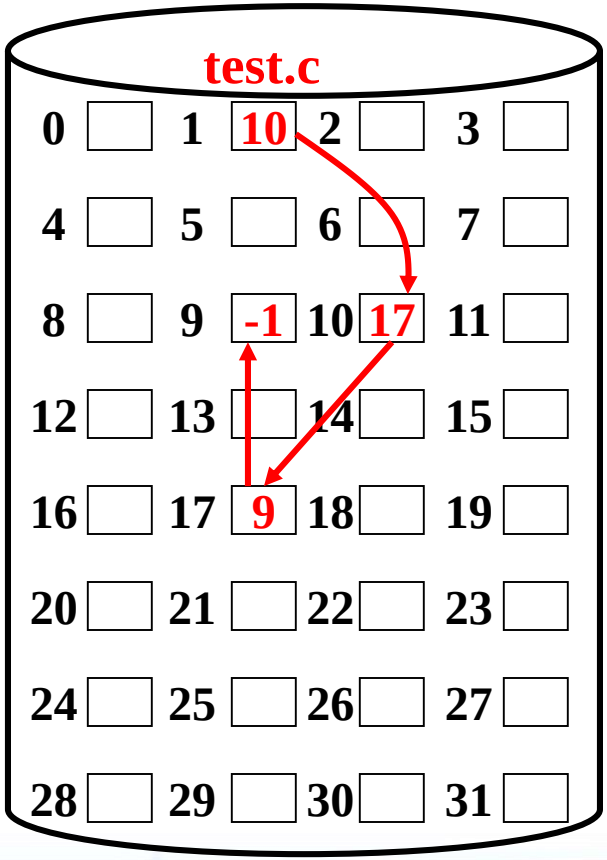
这就是通用操作系统的魅力!

三种基本映射关系

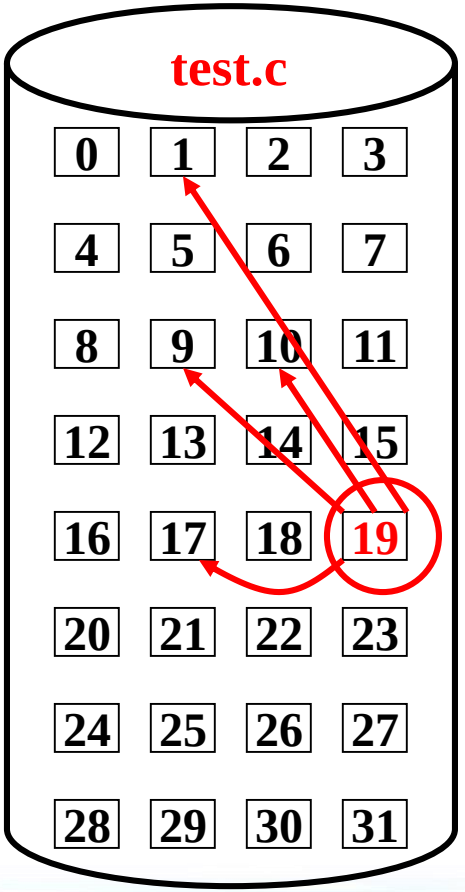
盘块连续分配



盘块链式分配



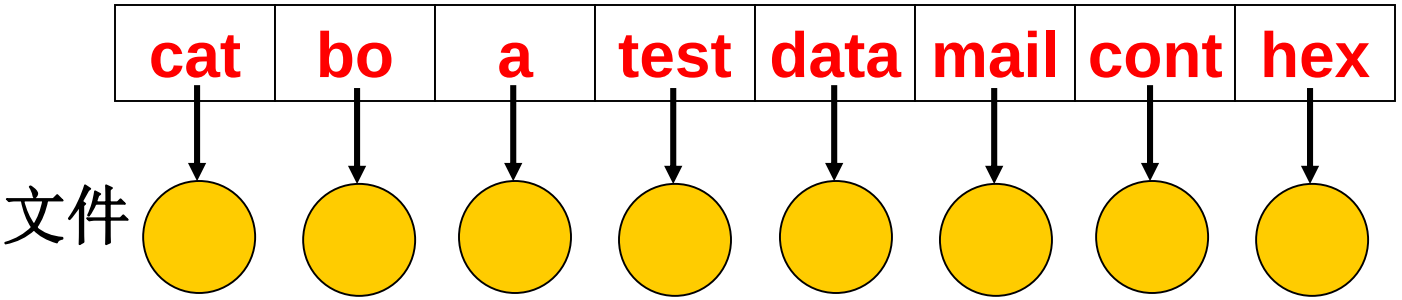
盘块索引分配



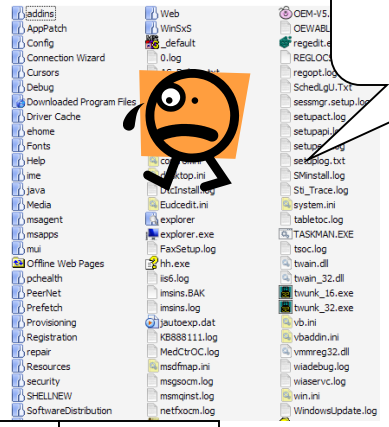
文件和文件系统的差别在哪里？

从字面上理解：文件系统-管理文件的系统
为文件的各类操作和存储（增删改查）提供简单、高效、安全和可靠的管理功能。

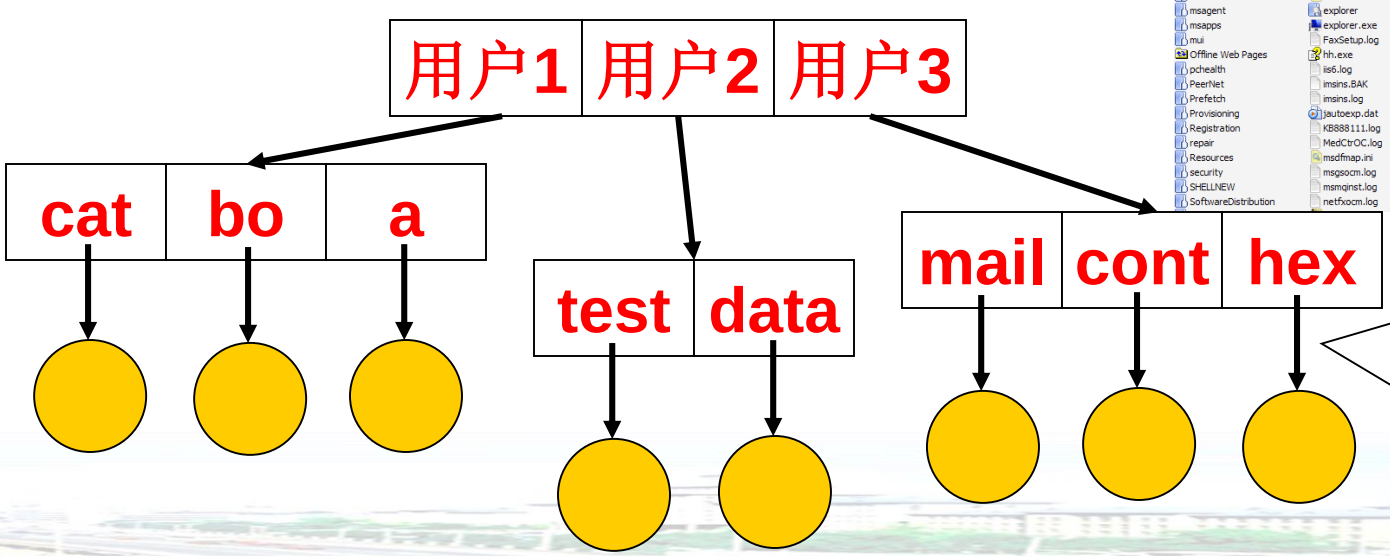
文件系统中有很多文件 怎么管理?



- 所有文件放在一层(一个大集合)
- 怎么办? 集合划分



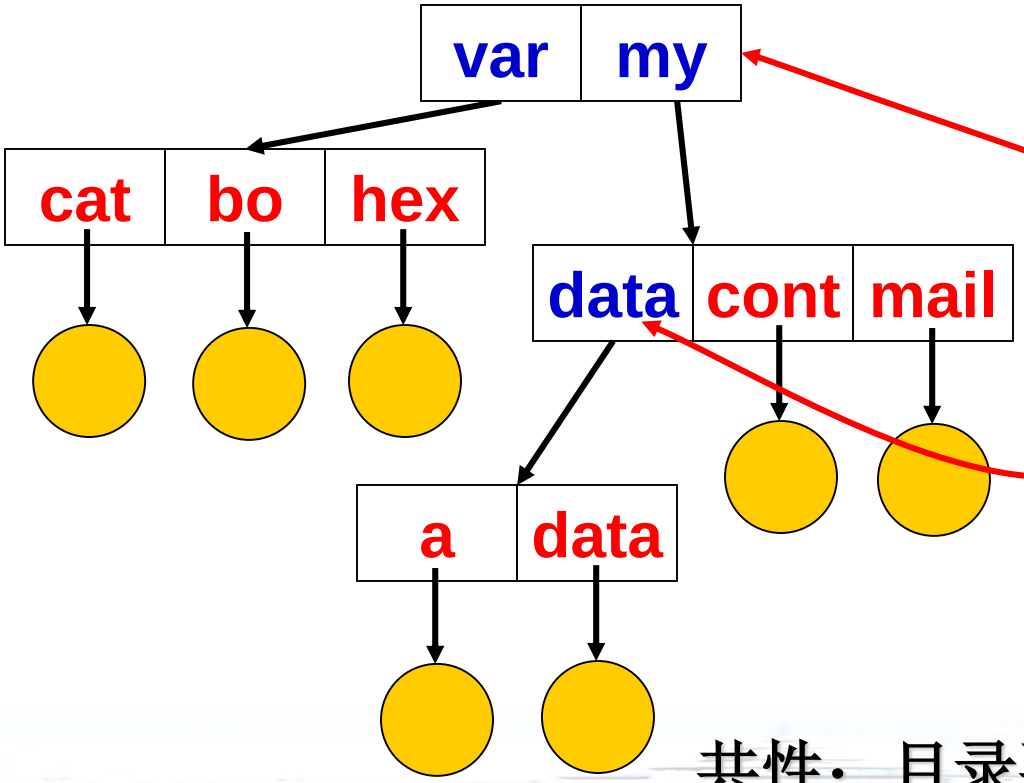
查找困难



问题依然存在: N/U，扩展性仍然差

划分的基础上继续划分 树状目录

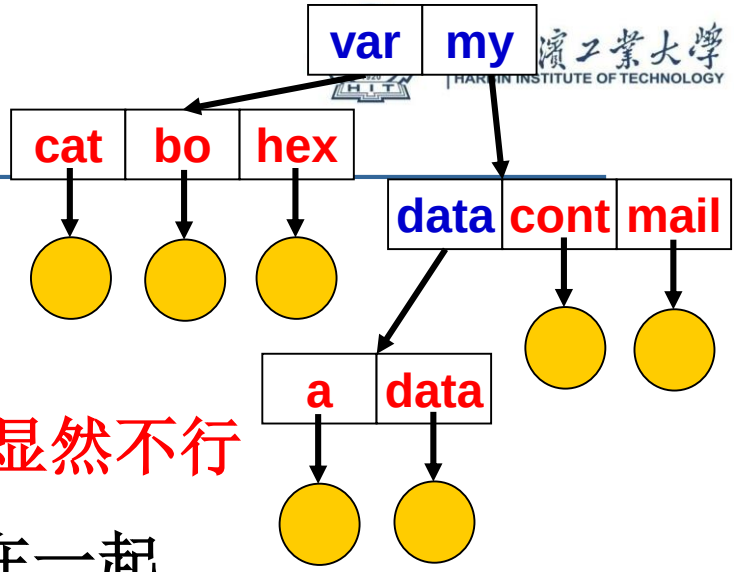
⑩ 将划分后的集合再进行划分: k次划分后，每个集合中的文件数可以达到为$O(\sqrt[k]{N})$



- 此种树状结构扩展性好、表示清晰，最常用
- 描述“文件的集合”，需要引入概念: 目录
- 怎么进行“文件集合”物理表述?

共性：目录要存储信息，文件要存储信息

目录的实现



- 存放“文件集合”
  - 将文件内容(盘块)放在一起… 显然不行
  - 将文件内容指针(即文件头)放在一起
  - 应该是可以的，取决于文件系统如何处理…
- 思考: 有了树状目录后会出现什么问题?
  - 出现了路径名: /my/data/a用来定位文件a
  - 路径名 ⇒ 路径的解析: 再根据文件头定位文件内容

输入/my/data/a，获得文件a的文件头

路径的解析(Name Resolution)

⑩ 输入/my/data/a，获得文件a的文件头

想一想?

- 从哪里开始? 顶层目录(根目录/)
- 目录是什么? 应该也是一个文件，文件存放的内容是该目录中所有文件的文件头!

文件系统中全是文件!

- 解析/my/data/a：

放在磁盘确定位置，可在OS初始化时读入!

```
res("/my/data/a")
{  fh=FileHeader("/");  //根目录的文件头
   data=ReadData(fh);fh=Find(data,"my");
   data=ReadData(fh);fh=Find(data,"data");
   data=ReadData(fh);fh=Find(data,"a");
   return fh; }
```

从路径解析来看目录内容

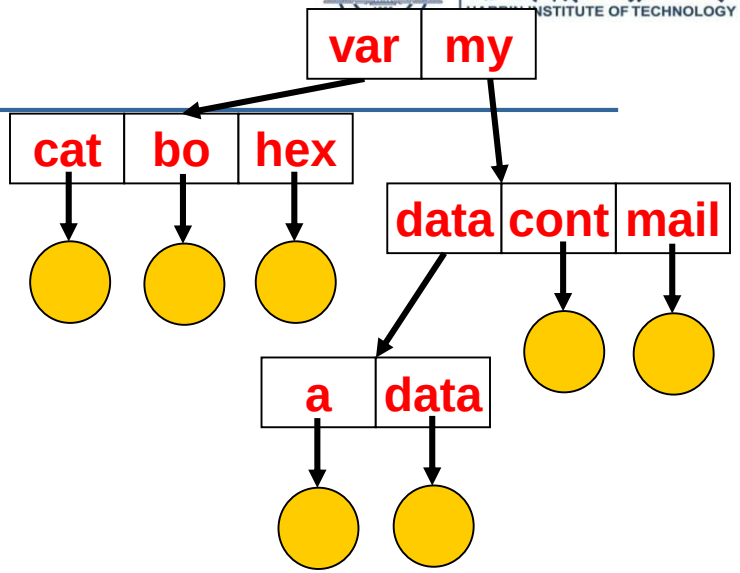
- 显然路径解析的使用频率高，因此效率很重要
  - 如何提高路径解析的效率?
  - 要使语句data=ReadData(fh);fh=Find(data,"??");
    效率高，data应该尽可能短!
  - 所以目录文件中不应该存放完整的文件头，可以存放指向文件头的指
  - 文件头指针? 可将文件头连续存放(形成了数组)在磁盘的固定位置，文件头指针就是其数组项标号!

文件头尺寸也并不小

任何文件的文件头结构相同

基址已知、偏移已知就能找到文件头

树状目录的完整实现


var
my
cat
bo
hex
data
cont
mail
a
data

磁盘
文件头数组
数据盘块集合
“/”的文件头
“/”的数据块
<var, 13>
<my, 82>
目录项!
文件头数组索引号!
“my”的文件头
“my”的数据块
<data, 103>
<cont, 225>
<mail, 77>

9.2 文件系统的实现

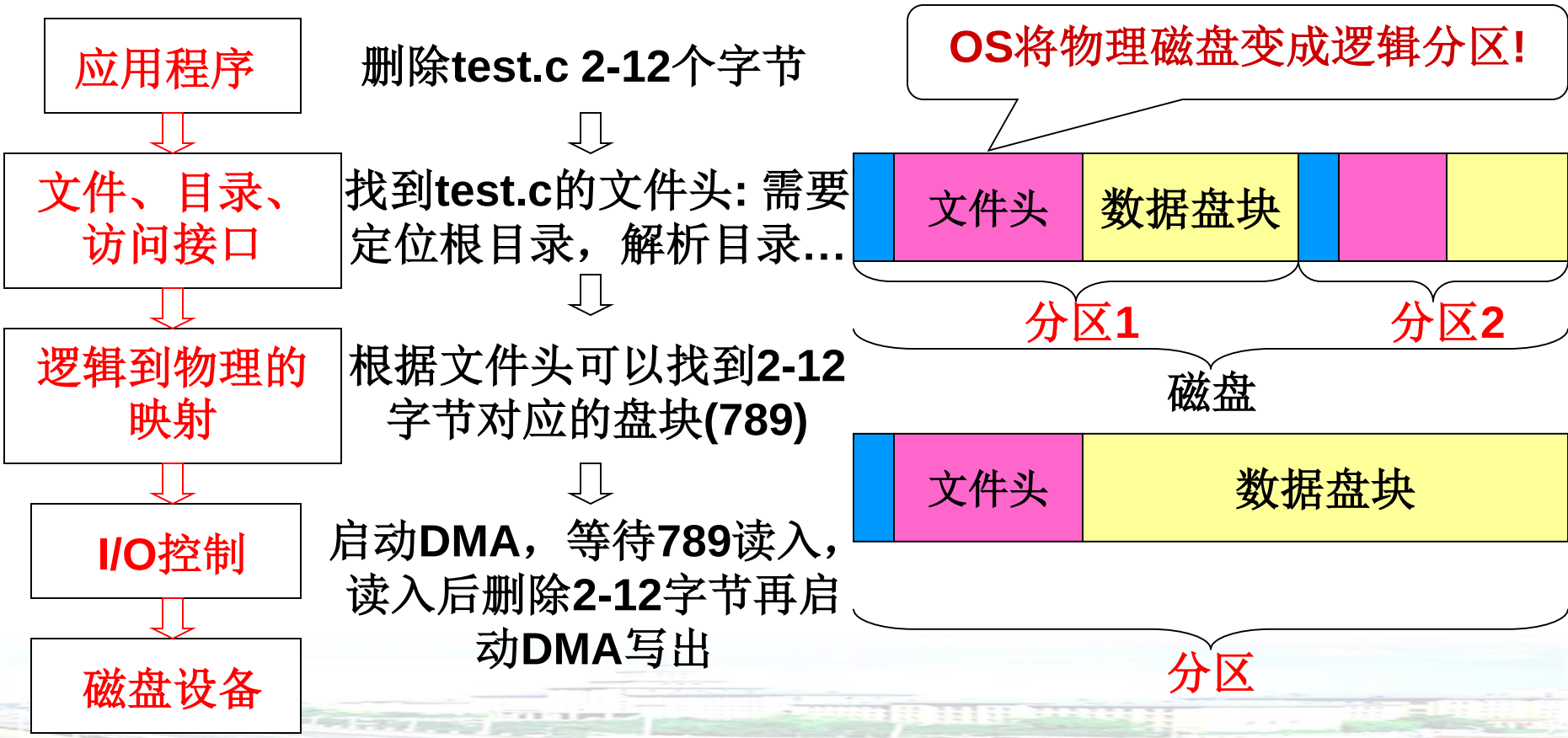
- 文件系统定义
- 典型的文件系统结构
- 文件分区空闲块的管理

描述文件系统的实质(定义)

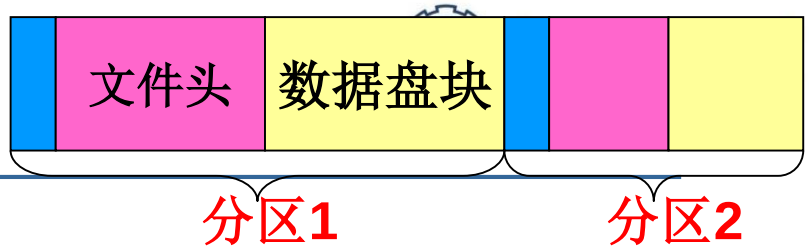
就像将CPU资源和地址空间封装成进程一样!

⑩ 文件系统: 将盘块“变”成文件集，方便用户访问

- 文件系统是“抽象盘块”的一层软件!



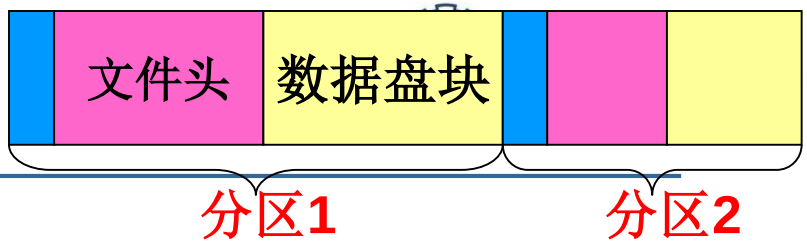
分区的详细结构



❿ 典型分区结构：UNIX分区的基本结构

- 引导块存放引导OS的信息，如果该分区中没有OS，则该块为空
- 超级块记录分区基本信息：分区块数；块大小；空闲块数量、指针；空闲文件头数量、指针等
- 索引节点数组存放所有文件的文件头，UNIX root目录的索引节点号为2
- 数据块，文件内容

分区空闲盘块的管理

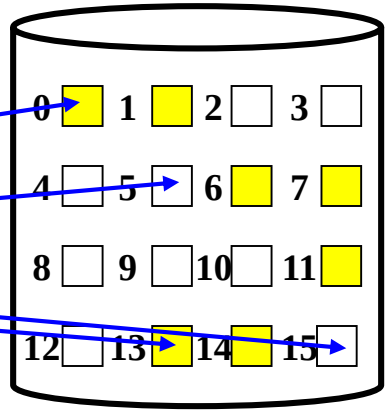


- ⑩ 有的盘块被文件使用，其它盘块如何管理?
  - 组织起来等待文件的使用! 怎么组织?
  - 方法1：空闲位图(位向量)…

    0011110011101001

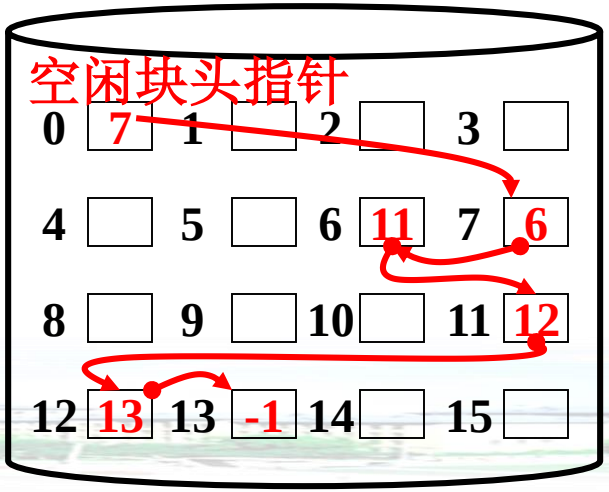
    表示磁盘块2,3,4,5,8,9,10,12已被占用

    可快速分配连续盘块组，但位向量很大(1G/1k=?)



  - 方法2：空闲链表

    分配一个(或少量的)空闲盘块是可以高效工作，但分配多个则慢!



良好运转的文件系统应该高效

- ⑩ 相比CPU和内存，磁盘读写非常慢!

```
int main(int argc, char* argv[])
{
    int  i , to, *fp, sum = 0;
    to = atoi(argv[1]);
    for(i=1; i<=to; i++)
    {
        sum = sum + i;
        fprintf(fp,"%d", sum);
    }
}
```

fprintf用一条其他计算语句代替

```
C:\>sum 10000000
0.015000 seconds
```

$0.015 * 10^{-7}$ S

用fprintf

```
C:\>sum 1000
0.859000 seconds
```

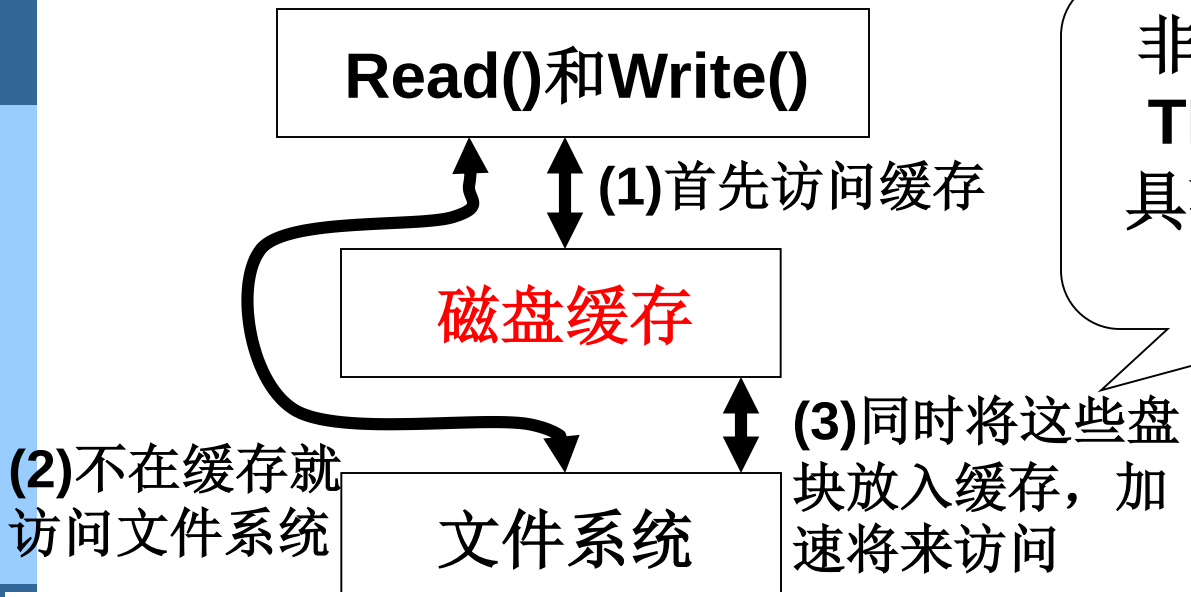
$0.859 * 10^{-3}$ S

$5.7 \times 10^5 : 1$

解决速度差异问题的基本手段是?
引入缓存!(因为局部性)

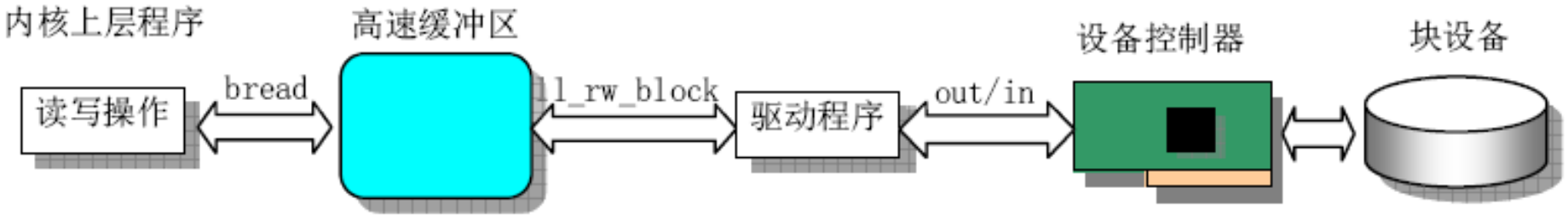
磁盘缓存

⑩ 在内存中缓存磁盘上的部分(很少部分)盘块



非常熟悉的过程?
TLB、虚拟内存!
具有局部性性质才有意义!

Linux中的sync命令
用于强制将修改的文件的内容立刻写入磁盘



多细节

Delayed Write or Write Through

其他的提高文件访问效率的技术…

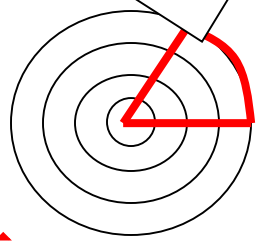
- 某些目录文件的FCB可以常驻内存
  - /的FCB常驻内存，因为许多目录解析都从此处开始
  - 当前目录(pwd)，如gcc 1.c。其FCB可驻内存?
- ls -l的使用非常频繁

一段时间大多数文件访问集中在一个目录中(局部性)

  - 怎么才能快速执行?

    同一目录中的文件inode在一个柱面(组)!

每个柱面组都有inode，空闲盘块…



  - 需要重新设计inode、目录文件分配算法…
- 显然，还有许多提高文件系统效率的技术……
  - 需要自己去整理，这是操作系统课程显著特点之一

良好运转的文件系统应该提供保护

- 文件用来存放用户的信息，用户应该能控制对文件的访问: 如只允许读
  - 文件关联权限，哪些权限? 放在哪里?
  - 权限: 读/写/执行(r/w/x)，当然是放在文件头中!
  - 一实例: drwxrwxrwx root staff test/

不同用户具有不同权限 | owner id | group id

  - 如何强制执行(enforce)?
  - 访问文件是由进程发起的，进程是由用户启动的:

(1)PCB中有uid和gid; (2)fork时设置; (3)这些信息在登录时收集(从tty); (4)文件访问时校验

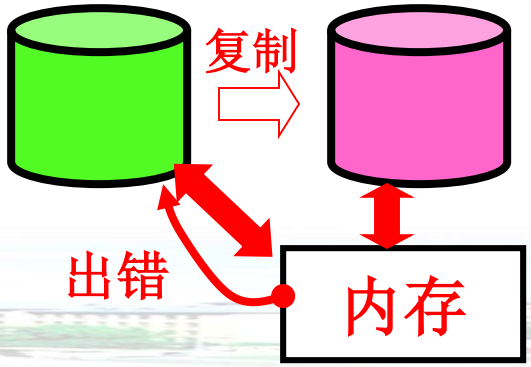
又将许多东西联系在了一起…OS魅力所在!

回忆:进程控制块PCB

```
struct task_struct {
/*-----------------------------------------------*/
    long state;    // 进程运行状态（-1 不可运行，0 可运行，>0 以停止）
    long counter; // 任务运行时间片，递减到 0 是说明时间片用完
    long priority;     // 任务运行优先数，刚开始是 counter＝priority
    long signal;  // 任务的信号位图，信号值=偏移+1
    struct sigaction sigaction[32];  //信号执行属性结构，对应信号将要执行的操作和标志信息
    long blocked; // 信号屏蔽码
/*--------------- various fields---------------------- */
    int exit_code;     // 任务退出码，当任务结束时其父进程会读取
    unsigned long start_code, end_code, end_data, brk, start_stack;
        // start_code      代码段起始的线性地址
        // end_code    代码段长度
        // end_data    代码段长度+数据段长度
        // brk         代码段长度+数据段长度+bss 段长度
        // start_stack     堆栈段起始线性地址
    long pid, father, pgrp, session, leader;
        // pid 进程号，father 父进程号，pgrp 父进程组号，session 会话号，leader 会话首领
    unsigned short uid, euid, suid;
        // uid 用户标 id，euid 有效用户 id，suid 保存的用户 id
    unsigned short gid, egid, sgid;
        // gid 组 id，egid 有效组 id，sgid 保存组 id
    long alarm;    // 报警定时值
```

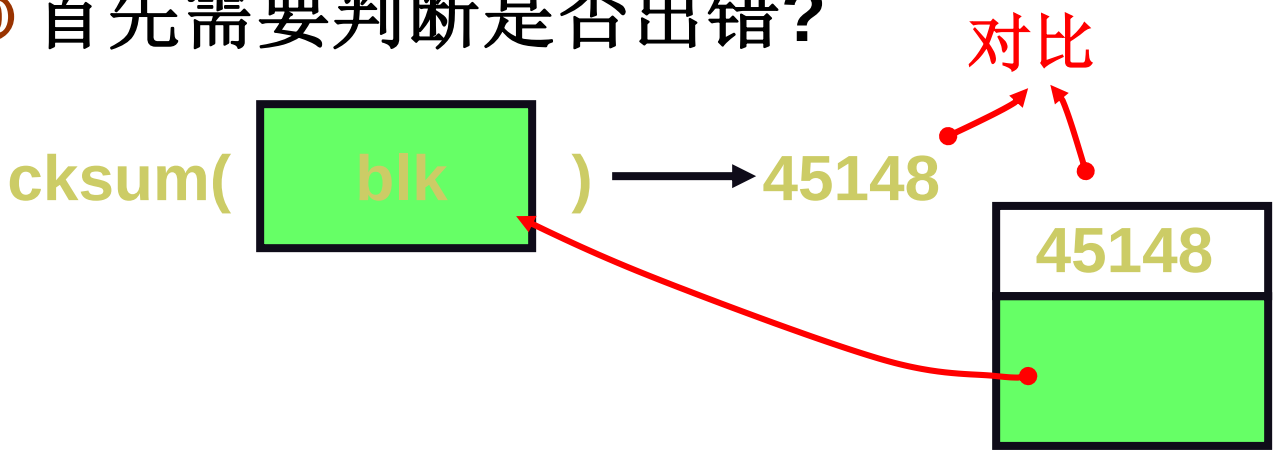
良好运转的文件系统似乎也应该容错

- 有过这样的经历?
  - 用户当然不希望: 早晨起来发现写了数月论文打不开了(如因下一块的link断了)…
  - 错误是难免的: 误操作、突然断电、无处不在的电磁干扰… 怎么办? 错误避免还是错误恢复…
  - **RAID(Redundant Arrays of Inexpensive Disks)**
  - **RAID基本思想就是冗余(R):**
    如在镜像磁盘上备份数据,
    发现错误时拷贝镜像磁盘(恢复)

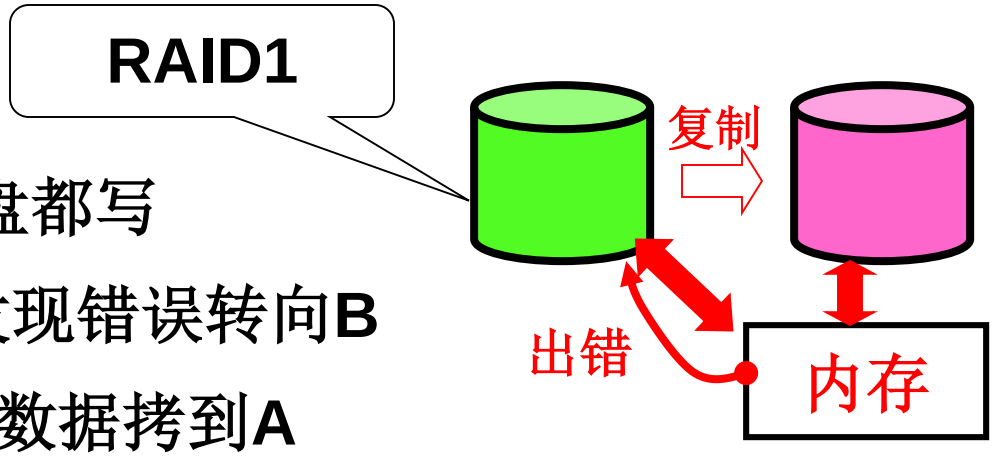


RAID的简单实现

- 首先需要判断是否出错?



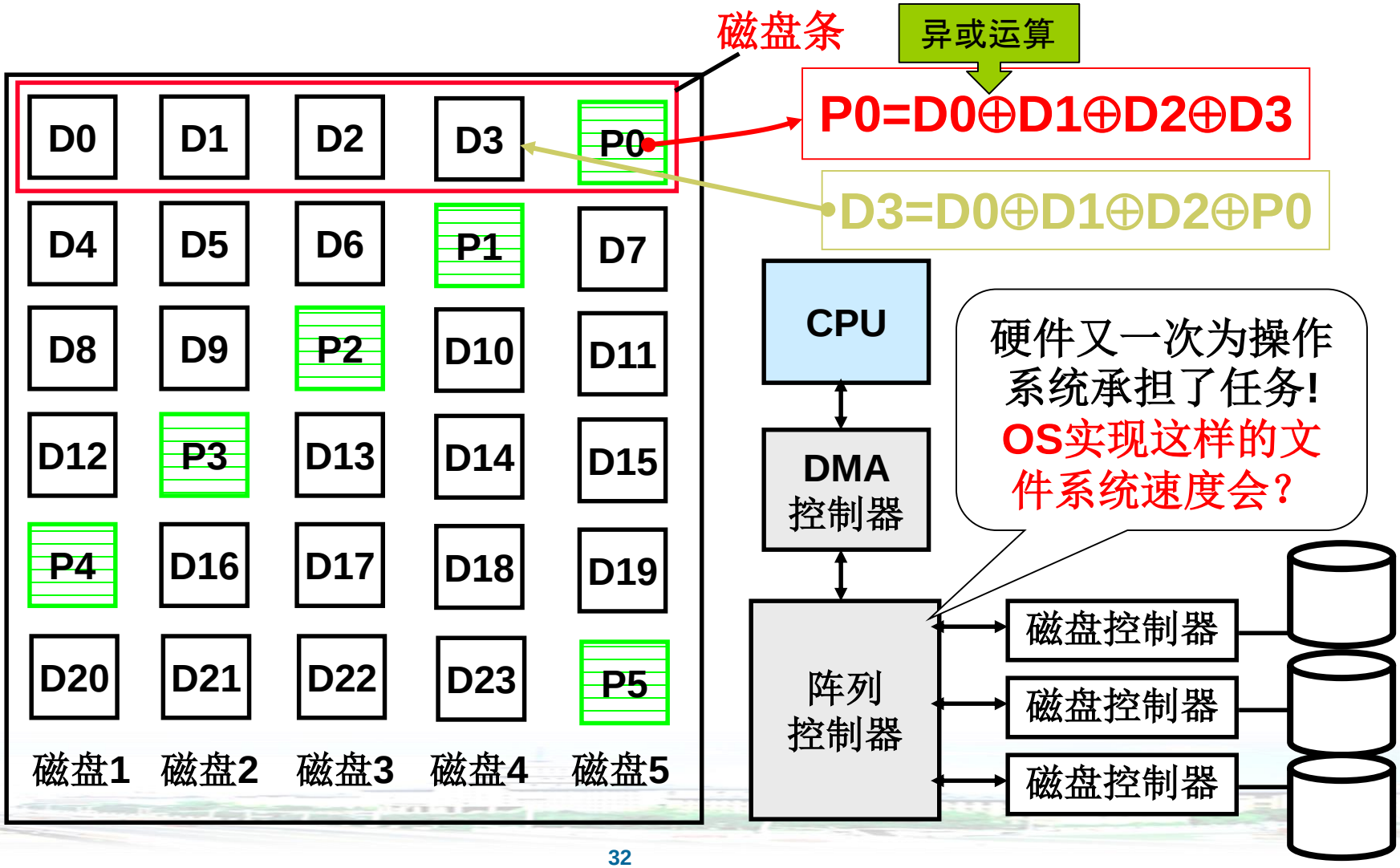
- 此时的磁盘读写



  - (1)每次写两个磁盘都写
    (2)从磁盘A读，发现错误转向B
    (3)A有错时将B的数据拷到A
  - 磁盘利用率50%

RAID5+

⑩ 校验数据分布在多个盘上，磁盘互相恢复数据



拓展阅读

PRO: A popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems @ FAST 2007

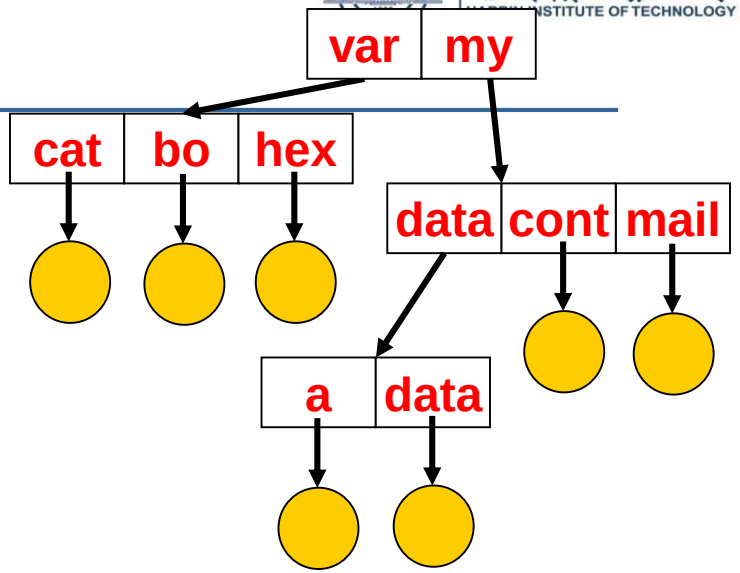
Workout: I/O workload outsourcing for boosting RAID reconstruction performance @ FAST 2009

HUST从事信息存储系统与技术的相关研究
于1974建立，是我国第一个可授予存储领域硕士学位(1984)和博士学位(1986)的实验室.
“计算机系统结构”全国重点学科
信息存储系统教育部重点实验室
武汉光电国家实验室(筹)信息存储功能实验室
数据存储系统与技术教育部工程研究中心

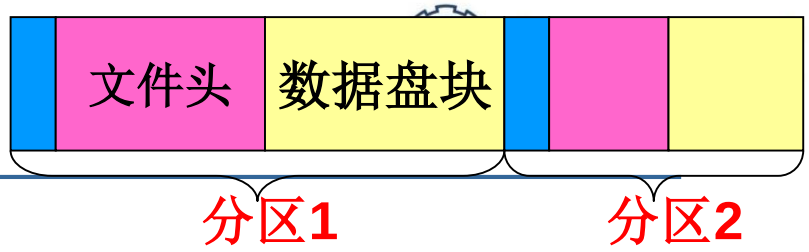
9.3 EXT2文件系统实现

- EXT2文件系统(实验四)

回顾：树状目录的完整实现



回顾：分区的详细结构



⑩ 典型分区结构：UNIX分区的基本结构

- 引导块存放引导OS的信息，如果该分区中没有OS，则该块为空
- 超级块记录分区基本信息：分区块数；块大小；空闲块数量、指针；空闲文件头数量、指针等
- 索引节点数组存放所有文件的文件头，UNIX root目录的索引节点号为2
- 数据块，文件内容

EXT2文件系统

EXT2文件系统把逻辑分区划分为块组，并且从0开始编号。每个块组包含的等量的物理块(即块组大小是相同的；物理分区最后一个块组可能小些)；在块组的数据块中存储文件或目录；

图9.2 Ext2磁盘布局在逻辑空间的映像

上图中启动块(Boot Block)的大小是确定的，用来存储磁盘分区信息和启动信息，任何文件系统都不能使用启动块。启动块之后才是ext2文件系统的开始。

EXT2文件系统的磁盘组织

除了引导扇区之外，EXT2磁盘分区被顺序划分为若干个**磁盘块组**（Block Group）。每个块组按照相同的方式组织，具有相同的大小。

EXT2磁盘块组中的磁盘块按顺序被组织成：

一个用作**超级块**的磁盘块：存放了文件系统超级块的一个拷贝；

多个记录**组描述符**的磁盘块；

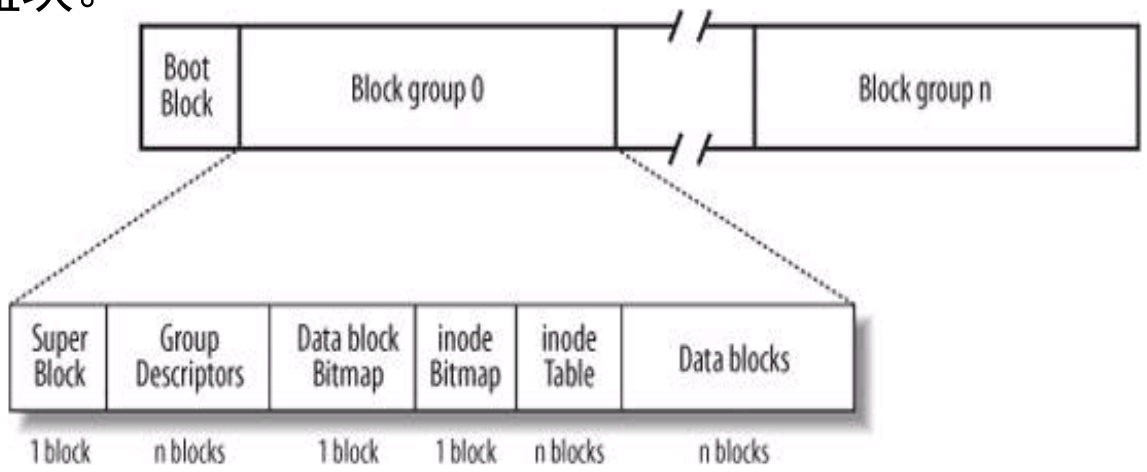
1个记录**数据块位图**的磁盘块；

1个记录**索引结点位图**的磁盘块；

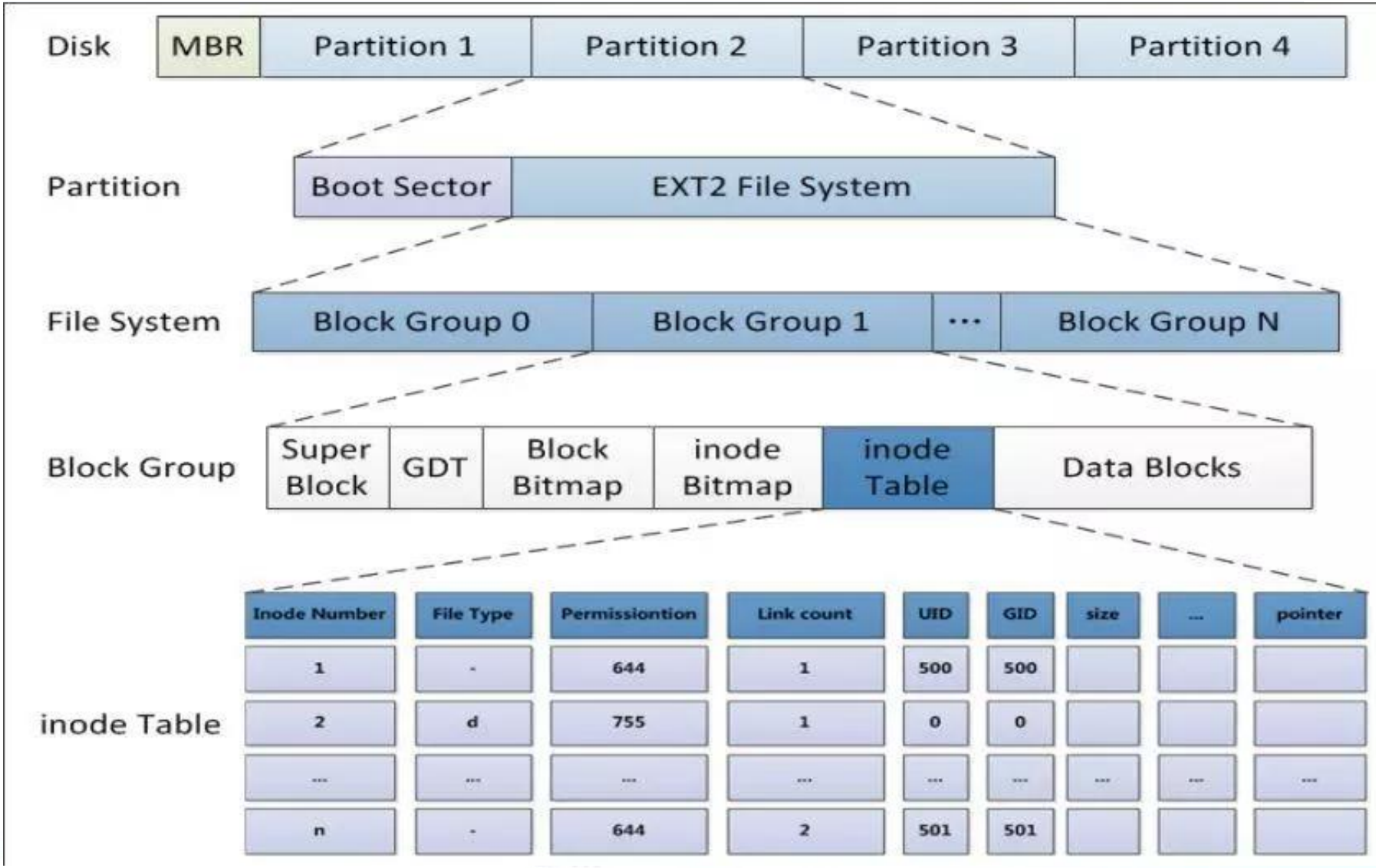
多个用作**索引结点表**的磁盘块；

多个用作**数据块**的磁盘块。

Layouts of an Ext2 partition and of an Ext2 block group



Ext2全貌图


Disk
MBR
Partition 1
Partition 2
Partition 3
Partition 4
Partition
Boot Sector
EXT2 File System
File System
Block Group 0
Block Group 1
···
Block Group N
Block Group
Super Block
GDT
Block Bitmap
inode Bitmap
inode Table
Data Blocks
inode Table
Inode Number
File Type
Permissiontion
Link count
UID
GID
size
...
pointer
1
-
644
1
500
500
2
d
755
1
0
0
...
...
...
...
...
...
...
...
...
n
-
644
2
501
501

EXT2的超级块

- 描述整个分区的文件系统信息，如块大小、版本号、上次mount时间等。
- 每个块组的第一个磁盘块用来保存所在EXT2 fs的超级块
- 多个块组中的超级块形成冗余
  - 在某个或少数几个超级块被破坏时，可用于恢复被破坏的超级块信息。
  - 系统运行期间，把超级块复制到系统缓冲区内，只需把块组0的超级块读入内存，其它块组的超级块做为备份

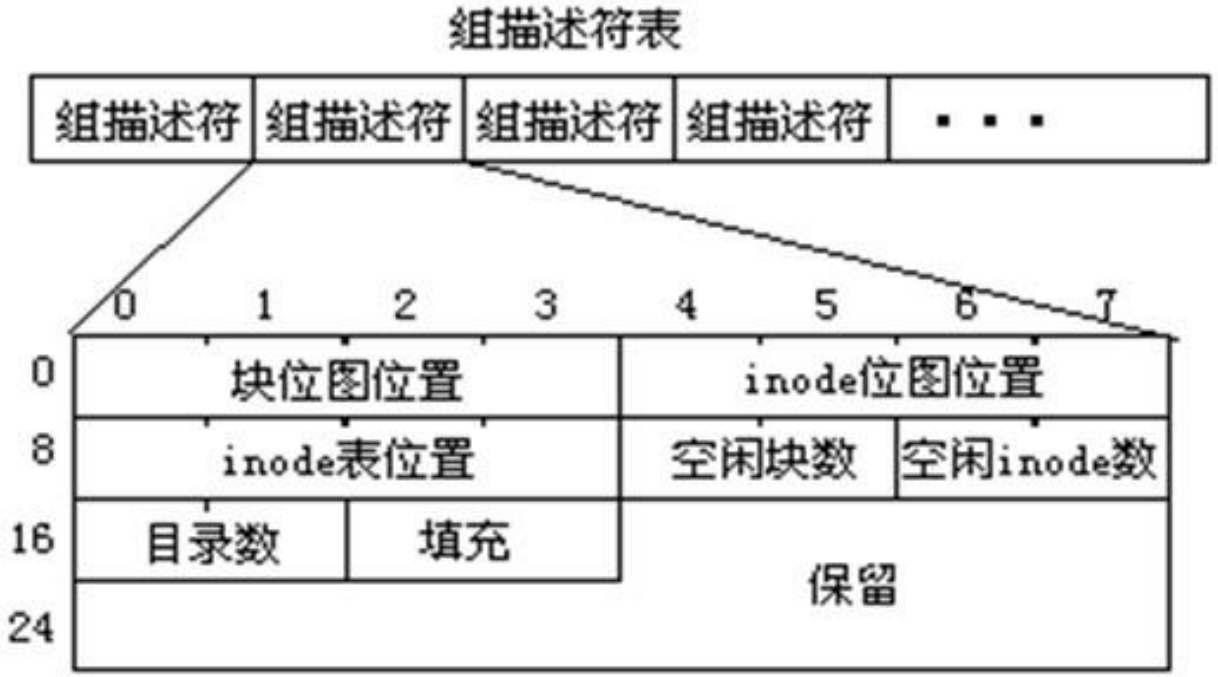
超级块数据结构

EXT2文件系统超级块

块组描述符

块组描述符用来描述一个磁盘块组的相关信息

块组描述符组由若干块组描述符组成，描述了文件系统中所有块组的属性，存放于超级块所在块的下一个块中。



EXT2文件系统组描述符

块组描述符

一个块组描述符的结构如下：

```
struct ext2_group_desc
{
__le32 bg_block_bitmap; // 块位图所在的第一个块的块ID
__le32 bg_inode_bitmap; // inode位图所在的第一个块的块ID
__le32 bg_inode_table; // inode表所在的第一个块的块ID
__le16 bg_free_blocks_count; // 块组中未使用的块数
__le16 bg_free_inodes_count; // 块组中未使用的inode数
__le16 bg_used_dirs_count; // 块组分配的目录的inode数
__le16 bg_pad;
__le32 bg_reserved[3];
};
```

数据块位图和索引结点块位图

- EXT2的空闲盘块分配算法采用了位图法
- 位图:为便于查找数据块或索引结点的分配信息
- 每个位(bit)都对应了一个磁盘块:
  - 0，表示对应的磁盘块(或索引结点)空闲
  - 1，表示占用。
- 2个位图(数据块和索引节点)分别占用一个专门的磁盘块;位于组描述符表之后
- 根据磁盘块的大小，可以计算出每个块组中最多能容纳的数据块个数和索引节点块个数

索引结点

EXT2中所有的索引结点大小相同，都是128个字节。

一个inode的结构如下

```
struct ext2_inode {
__le16 i_mode; // 文件格式和访问权限
__le16 i_uid; // 文件所有者ID的低16位
__le32 i_size; // 文件字节数
__le32 i_atime; // 文件上次被访问的时间
__le32 i_ctime; // 文件创建时间
__le32 i_mtime; // 文件被修改的时间
__le32 i_dtime; // 文件被删除的时间（如果存在则为0）
__le16 i_gid; // 文件所有组ID的低16位
__le16 i_links_count; // 此inode被连接的次数
__le32 i_blocks; // 文件已使用和保留的总块数（以512B为单位）
__le32 i_flags; // 此inode访问数据时ext2的实现方式
union {
struct {
__le32 l_i_reserved1; // 保留
} linux1;
struct {
__le32 h_i_translator; // “翻译者”标签
} hurd1;
struct {
__le32 m_i_reserved1; // 保留
} masix1;
} osd1; // 操作系统相关数据
```

续

```
__le32 i_block[EXT2_N_BLOCKS]; // 定位存储文件的块的数组，前
块号，第14个为二级间接块号，第15个为三级间接块号
__le32 i_generation; // 用于NFS的文件版本
__le32 i_file_acl; // 包含扩展属性的块号，老版本中为0
__le32 i_dir_acl; // 表示文件的“High Size”，老版本中为0
__le32 i_faddr; // 文件最后一个段的地址
union {
struct {
__u8 l_i_frag; // 段号
__u8 l_i_fsize; // 段大小
__u16 i_pad1;
__le16 l_i_uid_high; // 文件所有者ID的高16位
__le16 l_i_gid_high; // 文件所有组ID的高16位
__u32 l_i_reserved2;
} linux2;
struct {
__u8 h_i_frag; // 段号
__u8 h_i_fsize; // 段大小
__le16 h_i_mode_high;
__le16 h_i_uid_high; // 文件所有者ID的高16位
__le16 h_i_gid_high; // 文件所有组ID的高16位
__le32 h_i_author;
} hurd2;
struct {
__u8 m_i_frag; // 段号
__u8 m_i_fsize; // 段大小
__u16 m_pad1;
__u32 m_i_reserved2[2];
} masix2;
} osd2; // 操作系统相关数据
};
```

关于索引节点中的i_block[]

ext2的索引结点中使用了组合索引方式。

```
00241:      __le32    i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
```

```
00159: /*
00160:  * Constants relative to the data blocks
00161:  */
00162: #define  EXT2_NDIR_BLOCKS        12
00163: #define  EXT2_IND_BLOCK              EXT2_NDIR_BLOCKS
00164: #define  EXT2_DIND_BLOCK             (EXT2_IND_BLOCK + 1)
00165: #define  EXT2_TIND_BLOCK             (EXT2_DIND_BLOCK + 1)
00166: #define  EXT2_N_BLOCKS           (EXT2_TIND_BLOCK + 1)
```

前12项用作直接索引

第13项用作间接索引

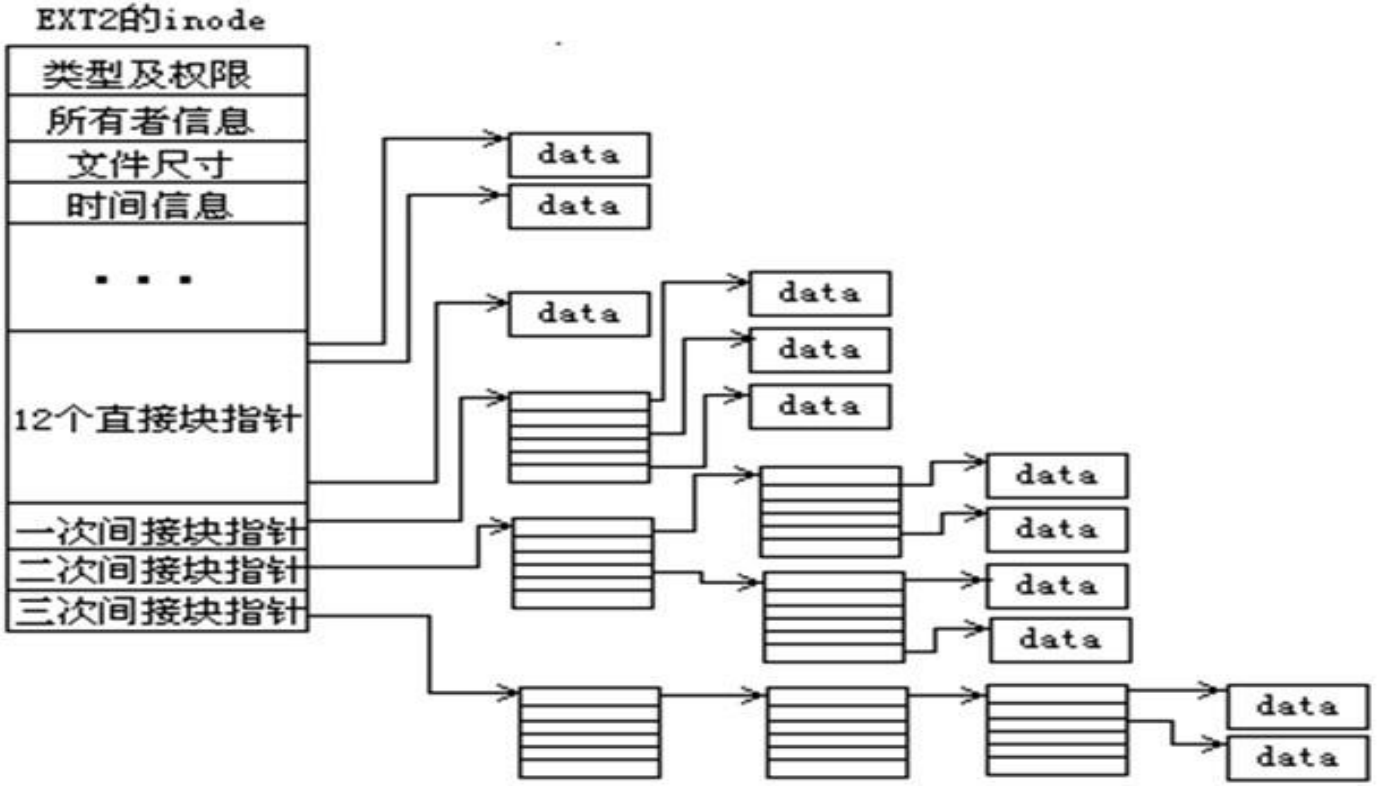
第14项用作二次间接索引

第15项用作三次间接索引

多级索引

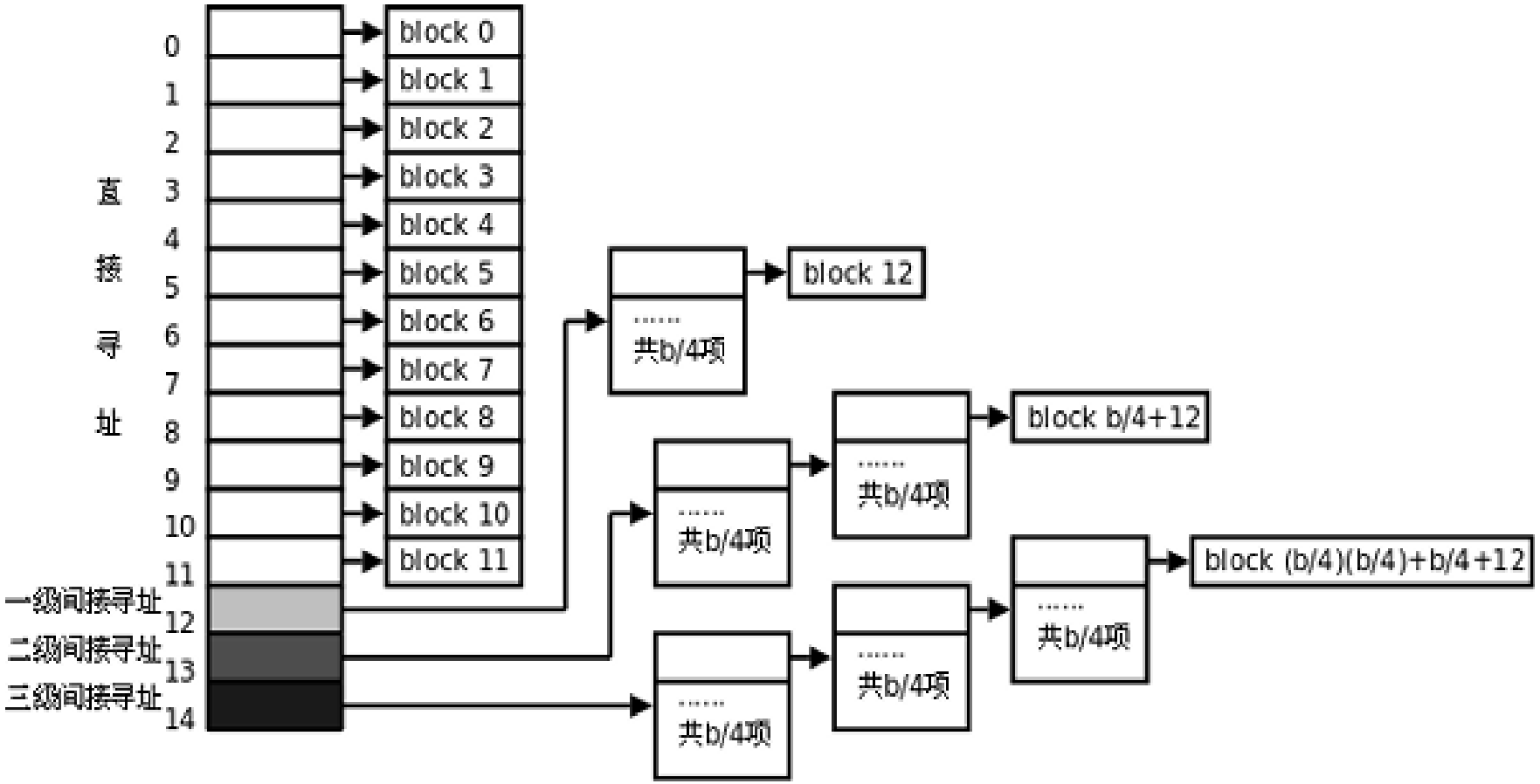
EXT2文件系统中的每个文件由一个inode描述，且只能由一个inode描述。

inode与文件一起存放在外存，系统运行时，把inode写入内存建立映像，加快文件系统速度。



EXT2的inode中物理块指针示意图

EXT2中的数据块寻址



假如采用磁盘块大小为4KB，索引表项中块号信息占四个字节，那么三级索引能够支持的最大文件多大？

EXT2中的目录项和文件类型

在ext2文件系统中，目录是作为文件存储的。

这种文件的数据块中存放了该目录下的所有目录项

```
struct ext2_dir_entry_2 {
__le32 inode; // 文件入口的inode号，0表示该项未使用
__le16 rec_len; // 目录项长度
__u8 name_len; // 文件名包含的字符数
__u8 file_type; // 文件类型
char name[255]; // 文件名
};
```

目录项在include/linux/ext2_fs.h文件中定义.

EXT2支持的文件类型

EXT2在目录项中存放了文件的类型信息。文件类型可以是0～7中的任意一个整数。它们分别代表如下含义：

0：文件类型未知；

1：普通文件类型；

2：目录；

3：字符设备；

4：块设备；

5：有名管道FIFO；

6：套接字；

7：符号链接

```
00539: /*
00540:  * Ext2 directory file types.  Only the low 3 bits are used.  The
00541:  * other bits are reserved for now.
00542:  */
00543: enum {
00544:     EXT2_FT_UNKNOWN,
00545:     EXT2_FT_REG_FILE,
00546:     EXT2_FT_DIR,
00547:     EXT2_FT_CHRDEV,
00548:     EXT2_FT_BLKDEV,
00549:     EXT2_FT_FIFO,
00550:     EXT2_FT_SOCK,
00551:     EXT2_FT_SYMLINK,
00552:     EXT2_FT_MAX
00553: };
```

管理ext2的磁盘空间

- 存储在磁盘上的文件与用户所“看到”的文件有所不同：
  - 用户感觉，文件在逻辑上是连续的
  - 而在磁盘上，存储文件数据的磁盘块可能分散在磁盘各处
- 磁盘碎片整理

随堂习题

【考研真题】某XXX文件系统最大容量为16TB，以磁盘块为基本分配单位，磁盘块大小为4 KB。文件控制块(FCB)包含一个1024B的索引表区。请回答下列问题：

(1)假设索引表区仅采用直接索引结构，索引表区存放文件占用的磁盘块号。索引表项中块号最少占多少字节？可支持的单个文件最大长度是多少字节？

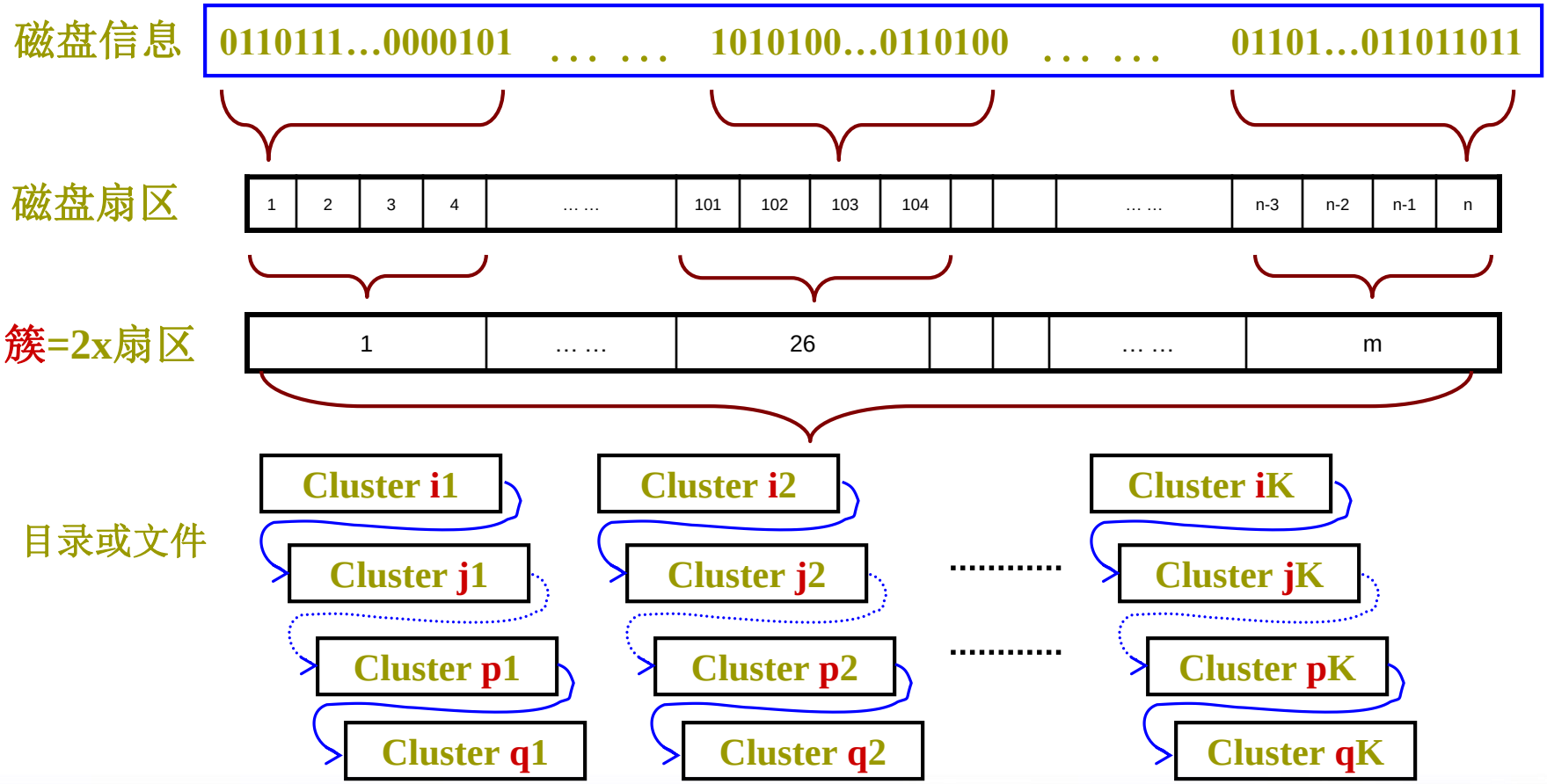
(2)假设索引表区采用如下结构：第0～7字节采用＜起始块号，块数＞格式表示文件创建时预分配的连续存储空间，其中起始块号占5B，块数占3B；剩余1016字节采用直接索引结构，一个索引项占8B，则可支持的单个文件最大长度是多少字节？为了使单个文件的长度达到最大，请指出起始块号和块数分别所占字节数的合理值并说明理由。

9.4 Windows的FAT文件系统实现

- **Windows的FAT文件系统实现**

Windows的FAT文件系统实现

数据组织逻辑结结构



Windows的FAT文件系统实现

FAT卷结构示意图

引导代码

介质标志
磁头数
每磁道扇区数
扇区大小
扇区数/簇

总扇区数
保留扇区数
隐藏扇区数

FAT数目
每个FAT占扇区数
根目录项数

如果是引导分区,则存在该代码,否则空闲

软盘或硬盘等

该参数不是固定不变的, 一般卷越大簇越大

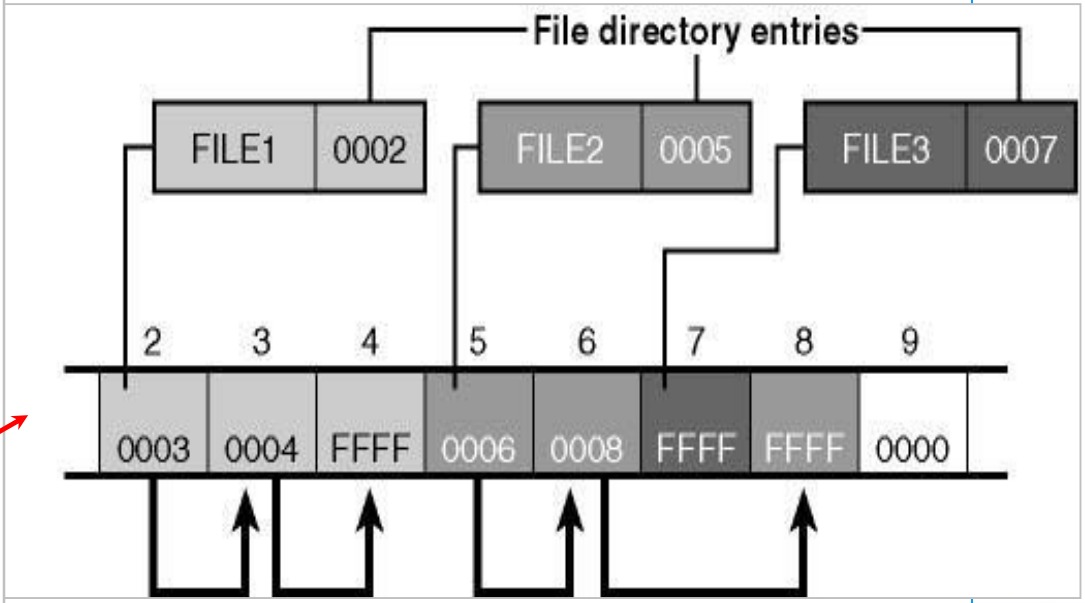
在FAT32中，引导扇区有备份

Windows的FAT文件系统实现

FAT卷结构示意图

功能： 记录和描述整个卷使用情况

1. FAT文件系统格式信息：
   FAT12/FAT16/FAT32
2. 卷上每一簇对应FAT中一项
   记录该簇使用情况
   包含：簇地址号和使用标志信息
   使用标志信息＝**0** — 该簇空闲未用
   ≠0 — 该簇被占用
3. 每个目录/文件的文件分配链(簇链)
   链尾标志信息：
   0xFFF/0xFFFF/0xFFFFFFFF



Windows的FAT文件系统实现

FAT卷结构示意图

FAT2是FAT1的“镜像”备份

文件分配表对卷非常重要
它的内容破坏会导致部分文件无法访问，甚至导致整卷瘫痪

对FAT表备份是十分必要的
若FSD（文件系统驱动程序）不能正常访问FAT1，则会访问FAT2

Windows的FAT文件系统实现

FAT卷结构示意图

根目录区保存卷中根目录项内容

FAT12和FAT16卷中预留256个目录项空间(是1个文件)
即指定了根目录可以容纳的文件和目录数上限
FAT32卷中没有预留根目录空间，也无文件/目录数限制

FAT目录项大小为32字节(文件名遵循8.3命名规则)，保存：
文件名、文件尺寸、文件属性、起始簇号、
创建日期和时间、最后访问日期、最后修改日期和时间

如果目录/文件名为长文件名（非8.3规则），则通过增加若干个目录项的方法解决

Windows的FAT文件系统实现

FAT卷结构示意图

引导区	文件分配表 FAT1	文件分配表 FAT2	根目录	其他目录和所有文件

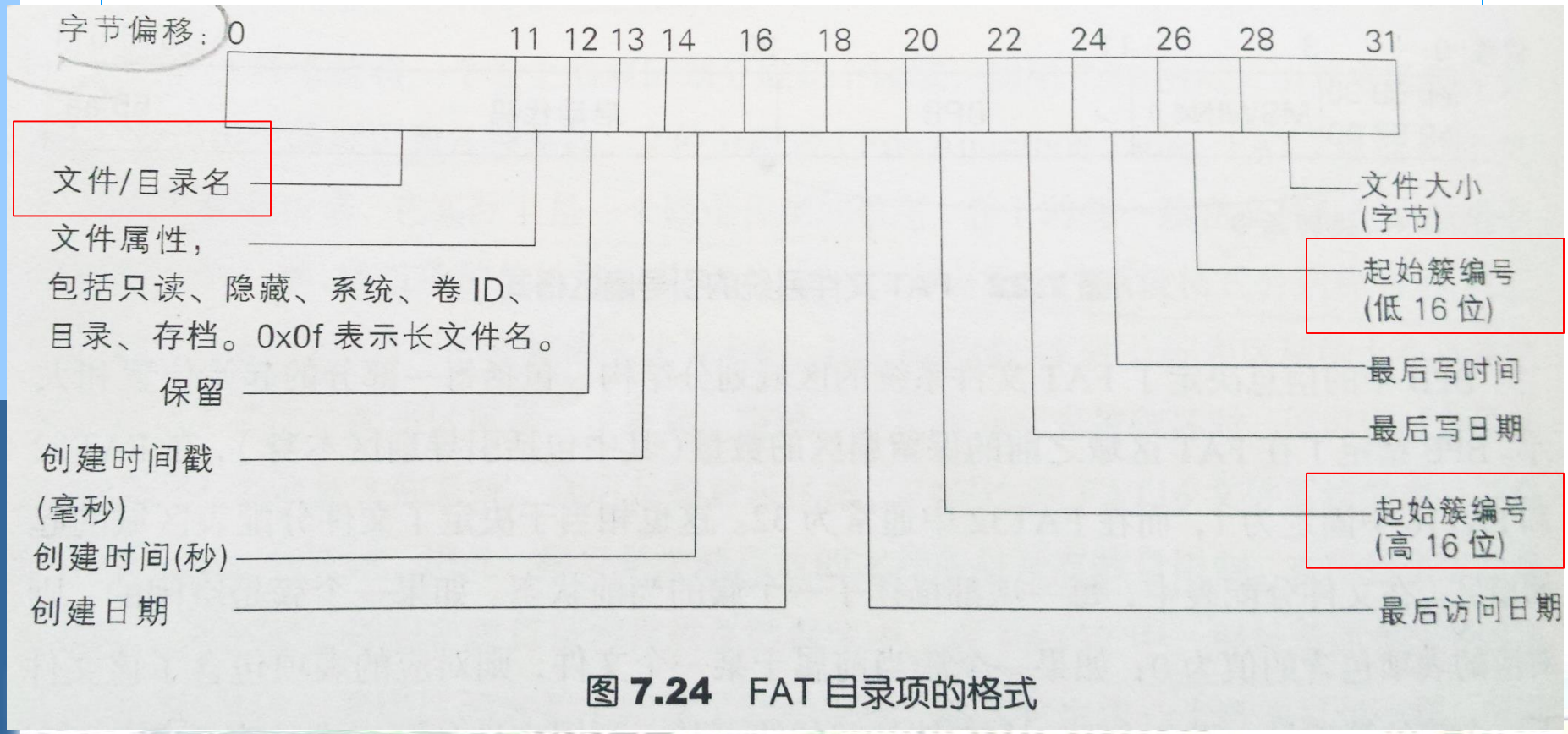
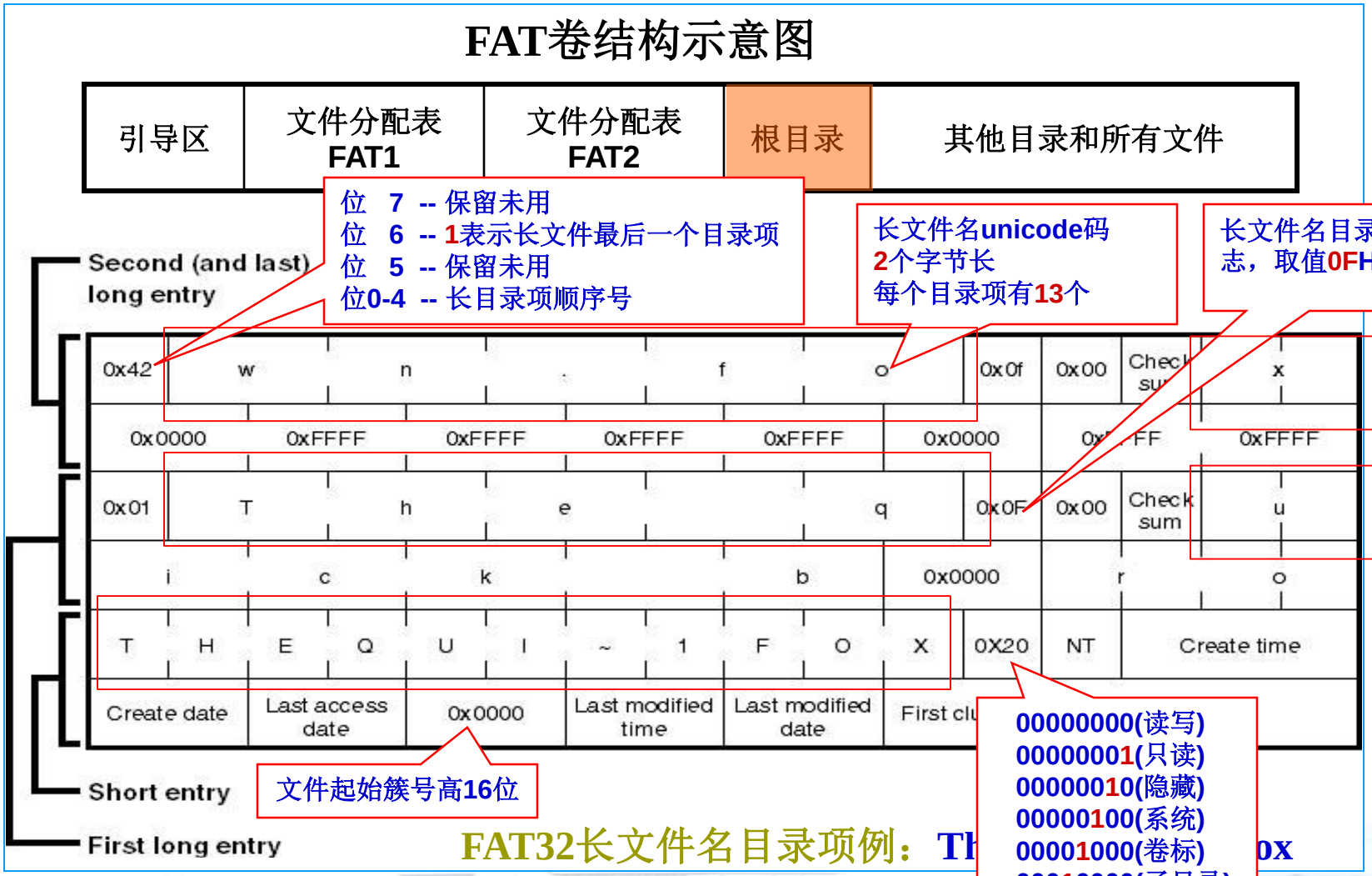
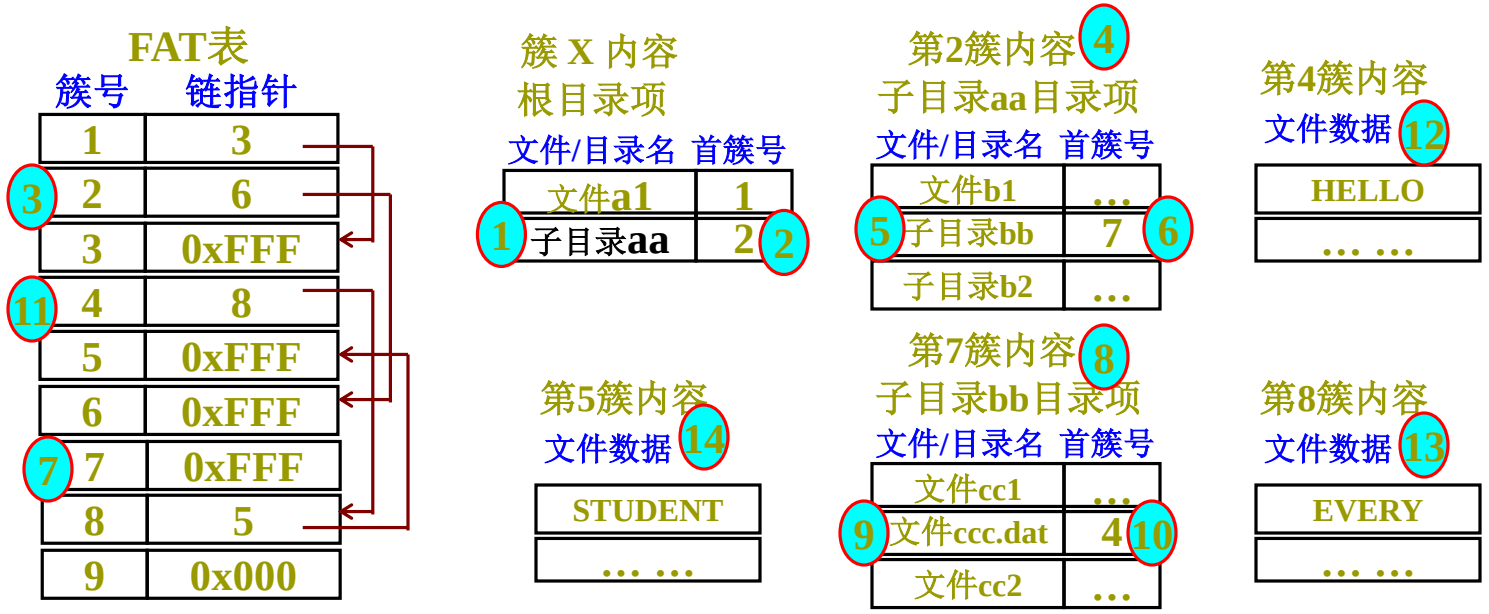


图 7.24　FAT 目录项的格式

Windows的FAT文件系统实现



Windows的FAT文件系统实现



文件访问过程：读出文件\aa\bb\ccc.dat的内容

1. 查“根目录”中目录项：找到含目录名=“aa”的目录项
2. 从aa目录项中查出该目录文件的首簇号＝2
3. 查FAT1中以第2簇为头的文件分配簇链，检索相应簇内容找出含目录名=“bb”的目录项
4. 从bb目录项中查出该目录文件的首簇号=7
5. 查FAT1中以第7簇为头的文件分配簇链，检索相应簇内容找出含文件名=“ccc.dat”的目录项
6. 从ccc.dat目录项中查出该文件的首簇号=4
7. 查FAT1中以第4簇为头的文件分配簇链，读出相应簇内容，得到了文件ccc.dat的内容

Windows的FAT文件系统实现

FAT表

簇号	链指针
1	3
2	6
3	0xFFF
4	8
5	0xFFF
6	0xFFF
7	0xFFF
8	5
9	0x000

簇 X 内容
根目录项

文件/目录名	首簇号
文件a1	1
子目录aa	2

第2簇内容
子目录aa目录项

文件/目录名	首簇号
文件b1	…
子目录bb	7
子目录b2	…

第4簇内容
文件数据

HELLO
… …

第5簇内容
文件数据

STUDENT
… …

第7簇内容
子目录bb目录项

文件/目录名	首簇号
文件cc1	…
文件ccc.dat	4
文件cc2	…

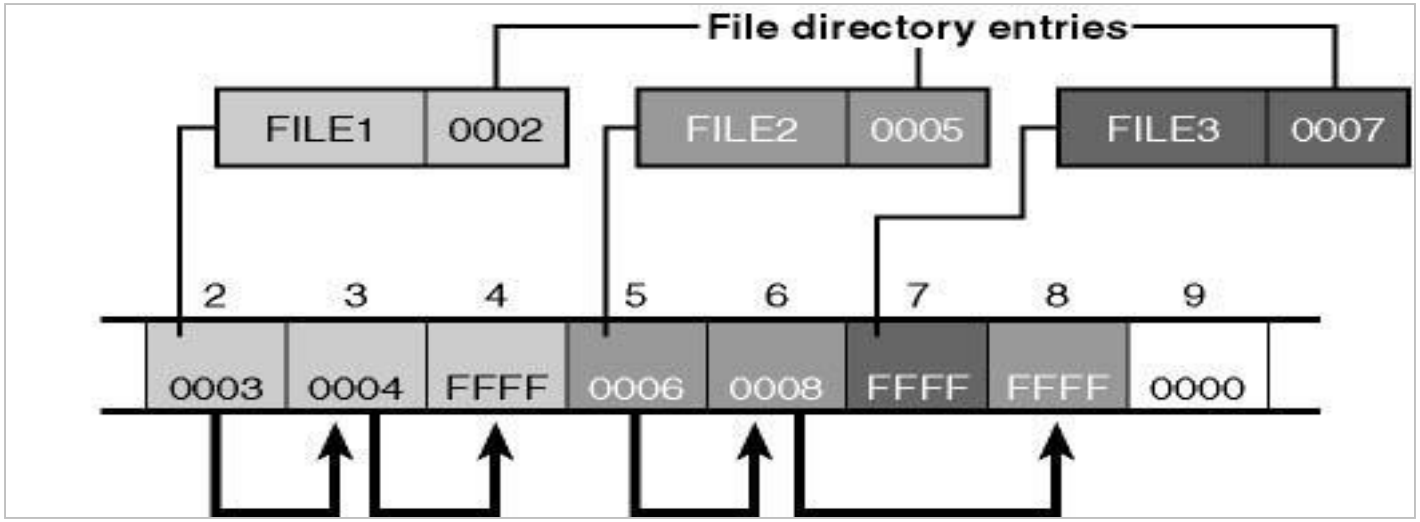
第8簇内容
文件数据

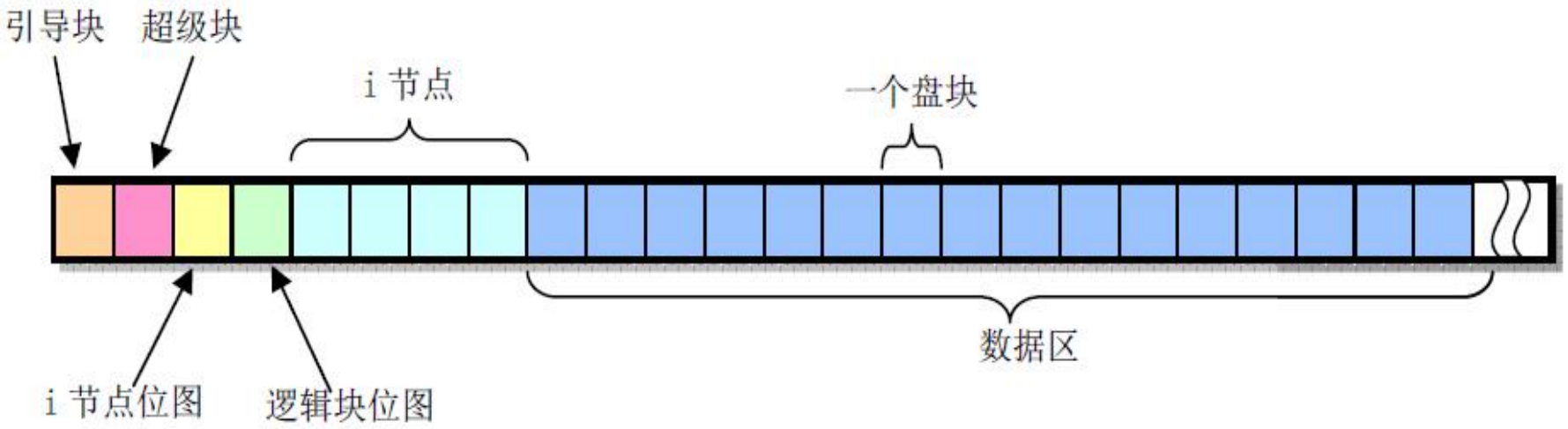
EVERY
… …

文件删除过程：删除文件\aa\bb\ccc.dat
文件建立过程：建立新文件\aa\fgx.txt

比较两种文件系统

引导区	文件分配表 **FAT1**	文件分配表 **FAT2**	根目录	其他目录和所有文件





Windows的FAT文件系统实现

FAT卷容量与簇大小的关系

- **FAT12** 最大簇数 = 2^{12}个=4K个
  规定簇大小=0.5KB～8KB
  FAT12卷大小≤32MB
- **FAT16** 最大簇数 = 2^{16}个=64K个
  规定簇大小=0.5KB～64KB
  FAT16卷大小≤4GB
- **FAT32** 最大簇数 = 2^{32}个=4G个
  限定使用2^{28}个=0.25G个
  规定簇大小=4KB～32KB
  FAT32卷大小 ≤ 1TG~8TB

FAT16/FAT32限制单个文件最大为4GB

W2K系统默认FAT16卷缺省簇大小

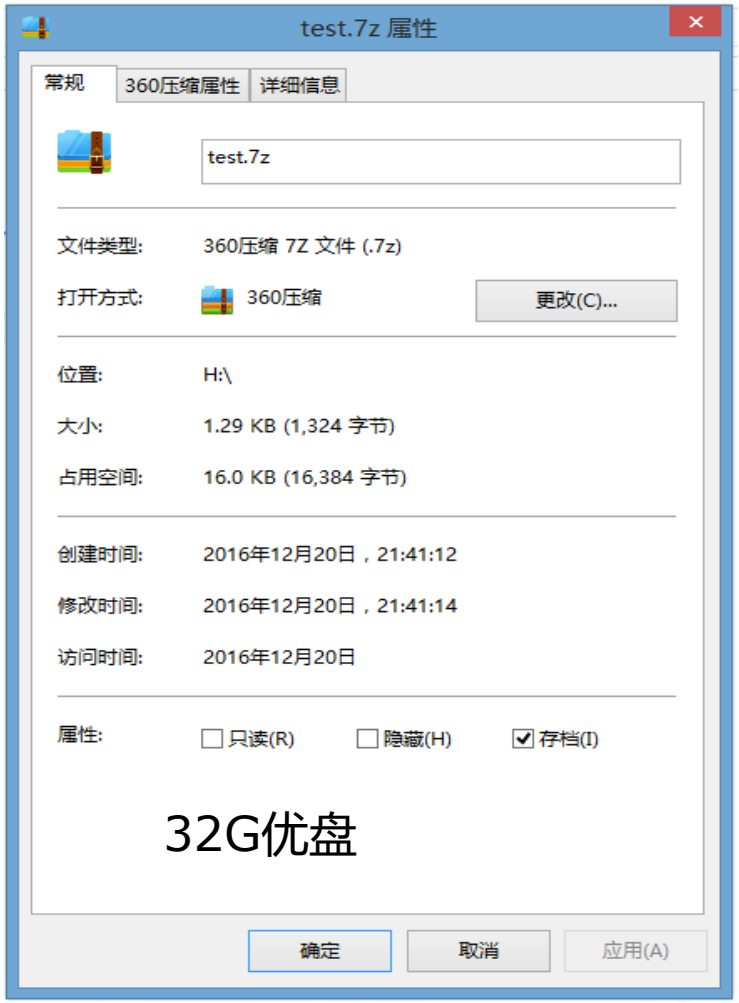
Volume Size	Cluster Size
0-32MB	512bytes
33MB-64MB	1KB
65MB-128MB	2KB
129MB-256MB	4KB
257MB-511MB	8KB
512MB-1023MB	16KB
1024MB-2047MB	32KB
2048MB-4095MB	64KB

FAT32卷缺省簇大小

Volume Size	Cluster Size
32MB-8GB	4KB
8GB-16GB	8KB
16GB-32GB	16KB
>32GB	32KB

为什么在同一种文件系统中簇的大小不统一？

Windows的FAT文件系统实现



32G优盘

128G优盘

随堂习题

- 假设某文件系统的硬盘空间为500MB，盘块大小为1KB，采用显示链接分配，请回答以下问题：
  - (1)其FAT表（文件分配表）需占用多少存储空间？
  - (2)如果文件A占用硬盘的盘块号依次为120、130、145、135、125共五个盘块，请画图示意文件A的FCB与FAT表的关系以及FAT表中各盘块间的链接情况。

文件系统总结

- 文件 ⇒ 帮助用户将信息放在磁盘上
- 一个长长的字符序列 ⇒ 盘块集合 ⇒ 文件系统完成映射
- 操作(文件名,偏移) ⇒ 找到文件头(FCB)，找到磁盘盘块
- 系统中会有很多文件 ⇒ 为方便寻找，组织成树状目录
- 目录表明某些文件在一个集合中 ⇒ 目录也是文件，其内容是<文件名，FCB的指针>
- 树状目录 ⇒ 文件路径 ⇒ 路径解析
- 目录+FCB+文件盘块+空闲盘块 = 文件系统
- 从可以运转到良好运转 ⇒ 高效、保护、容错…

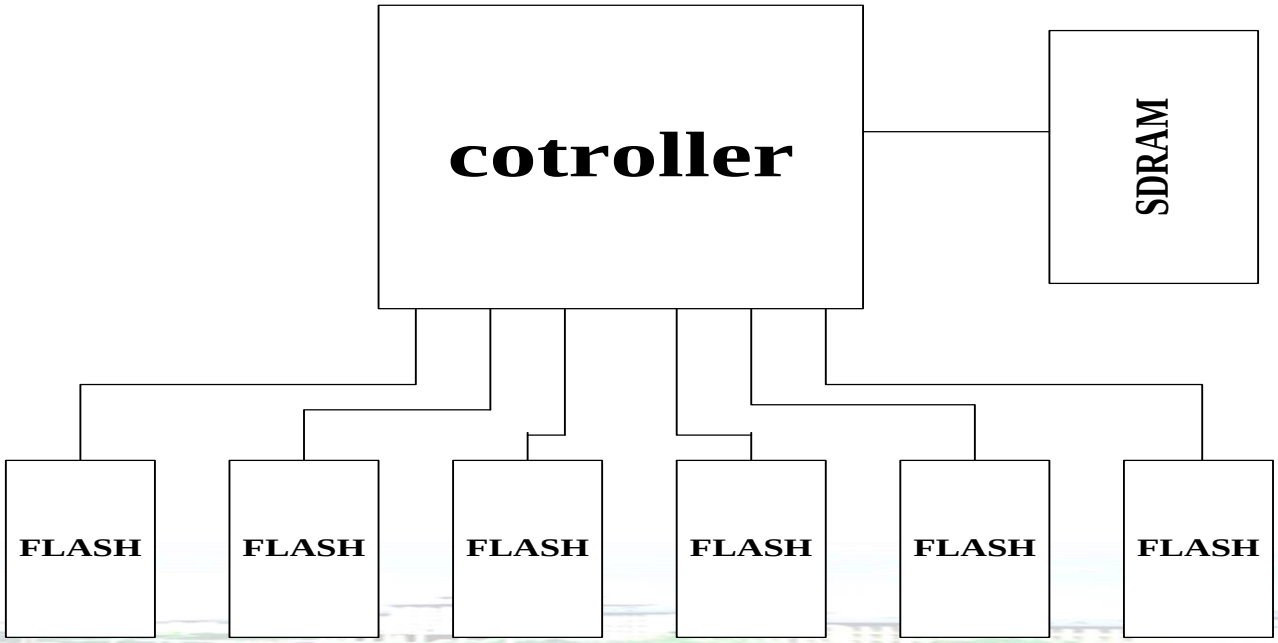
固态盘以及相关技术

- 固态盘以及相关技术

固态盘(SSD:Solid State Disk/Drive)

由NAND FLASH作为存储介质，由一个嵌入式控制器控制NAND FLASH的操作，RAM作为buffer，通过IDE,SATA,PCI-e等总线对外提供块接口

分为SLC、TLC、MLC、QLC等；



SSD 性能特性

顺序读吞吐量	550 MB/s	顺序写吞吐量	470 MB/s
随机读吞吐量	365 MB/s	随机写吞吐量	303 MB/s
平均顺序读访问时间	50 us	平均顺序写访问时间	60 us

- **顺序访问比随机访问快**
  - 典型存储器层次结构问题
- **随机写较慢**
  - 擦除块需要较长的时间(~1ms)
  - 修改一页需要将块中所有页复制到新的块中
  - 早期SSD 读/写速度之间的差距更大

资料来源: Intel SSD 730 产品详细说明书

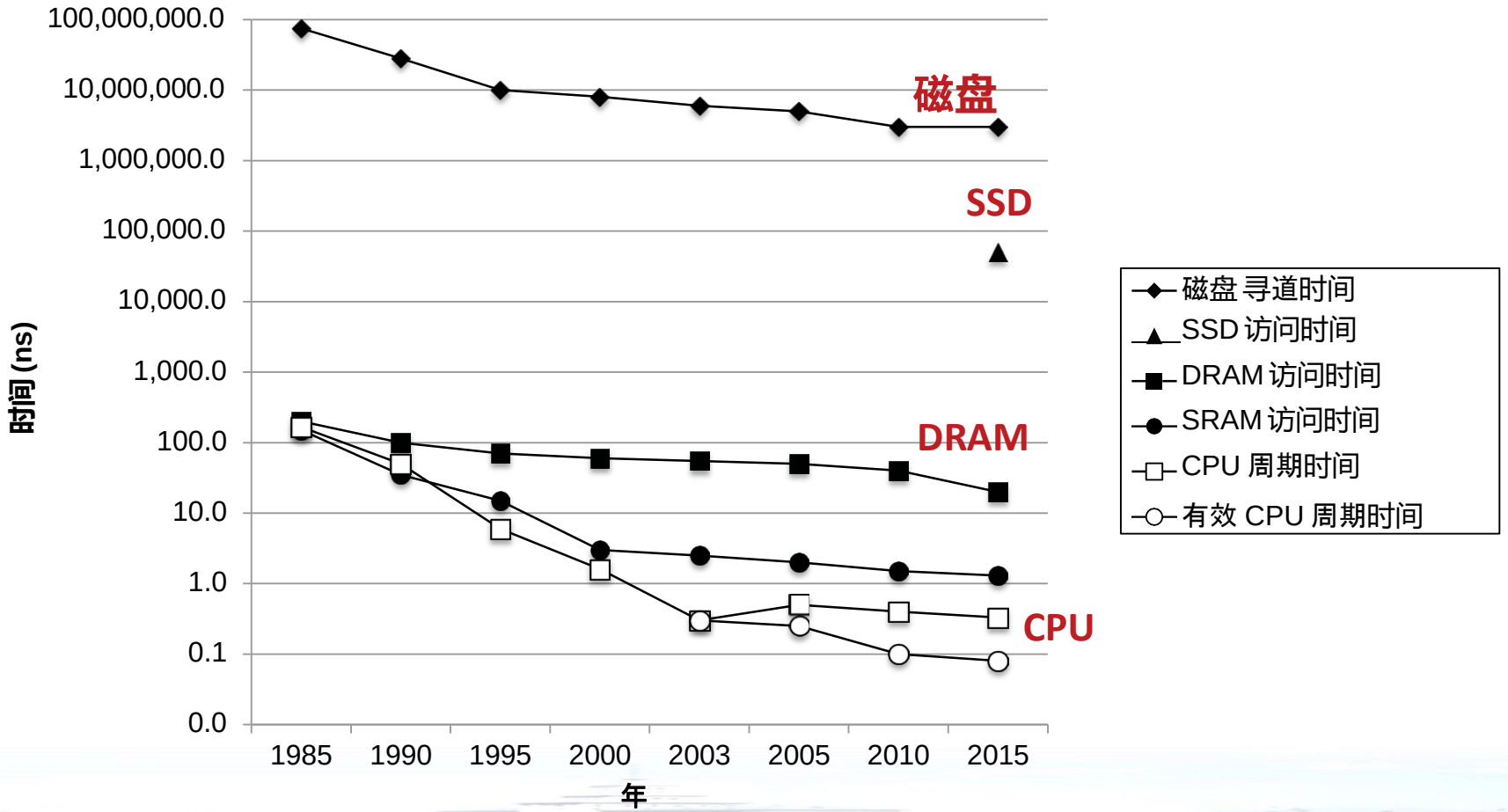
SSD vs 机械磁盘

- **优点**
  - 没有移动部件 → 更快、能耗更低、更结实
- **缺点**
  - 会磨损
    - 闪存翻译层中的平均磨损逻辑试图通过将擦除平均分布在所有块上来最大化每个块的寿命
    - 比如，Intel SSD 730 保证能经得起 128 PB (128×10^{15} 字节)的写
  - 2015年，SSD每字节比机械磁盘贵大约30倍
- **应用**
  - MP3播放器、智能手机、笔记本电脑
  - 开始在台式机和服务器中应用

72

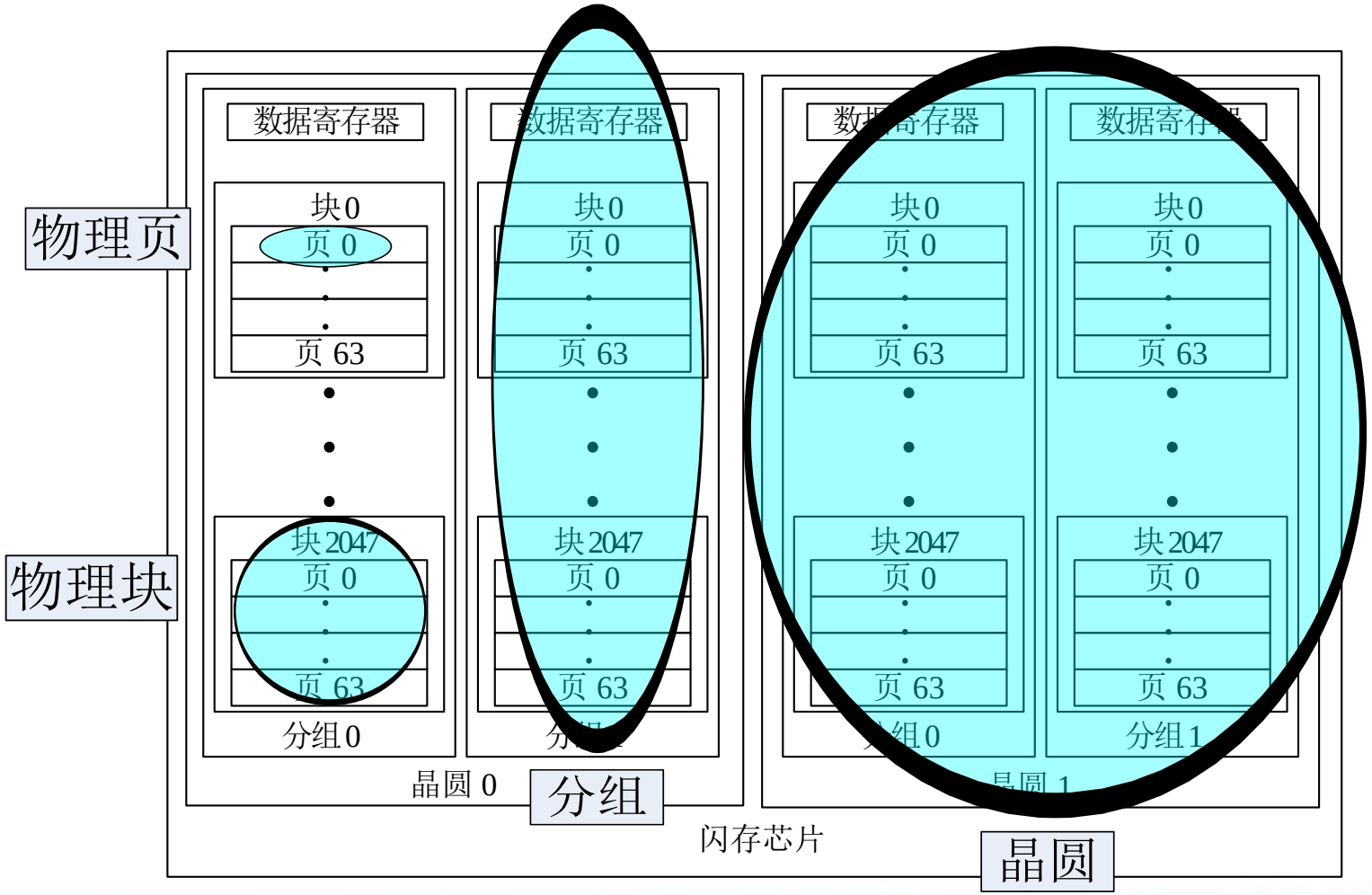
CPU-储存器 之间的差距

DRAM、磁盘和CPU速度之间的差距

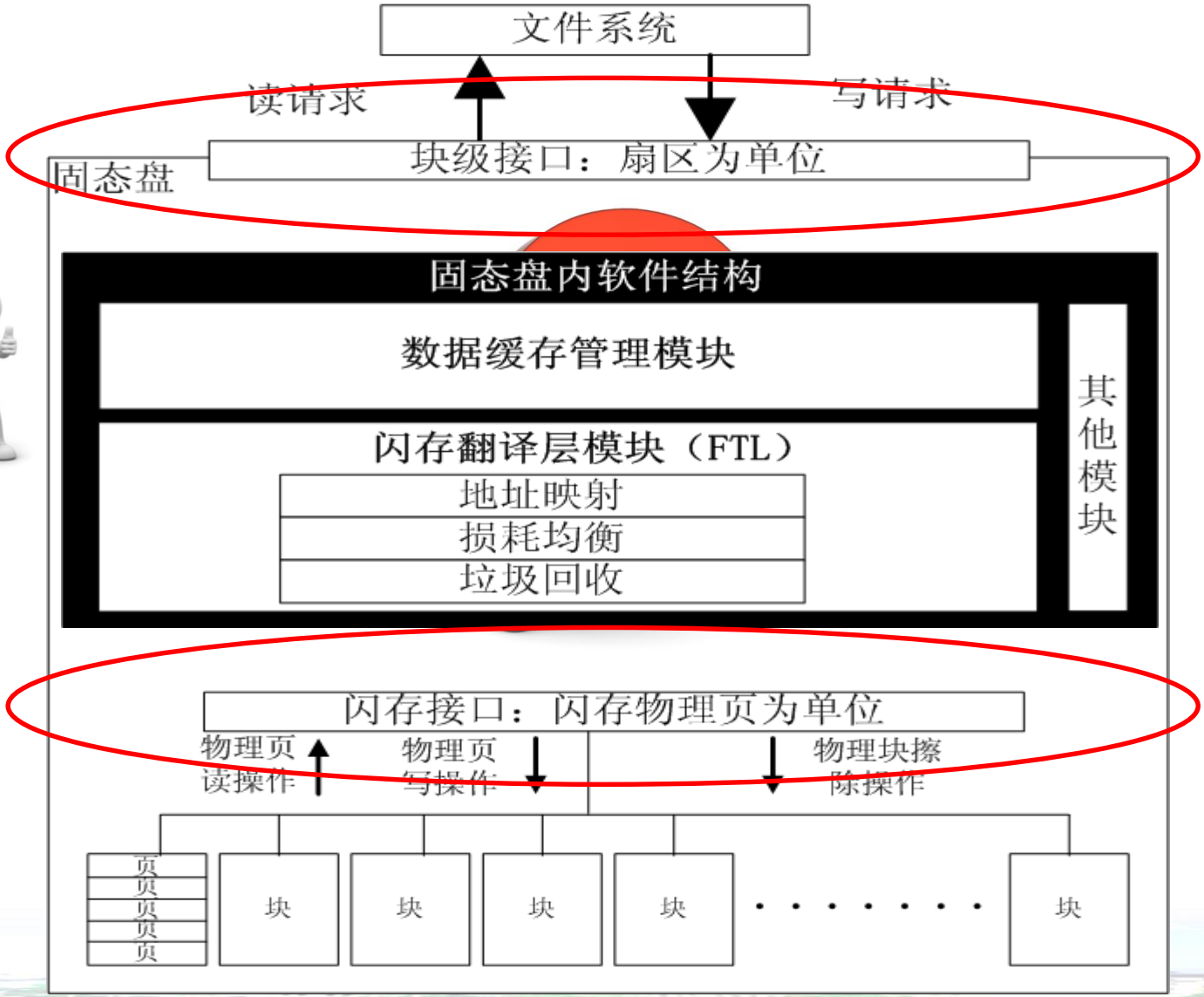


73

Flash芯片内部结构



SSD软件结构


文件系统
读请求
写请求
块级接口：扇区为单位
固态盘
固态盘内软件结构
数据缓存管理模块
闪存翻译层模块（FTL）
地址映射
损耗均衡
垃圾回收
其他模块
闪存接口：闪存物理页为单位
物理页读操作
物理页写操作
物理块擦除操作
页
块

FTL(Flash Translation Layer)

SSD是以硬盘的替代者的姿态出现，为了与现有系统无缝对接，SSD必须对外提供的是块接口，作为主机端，所看到的SSD是一个和HDD一样的块设备。

为了达到模拟块设备的目的，SSD中需要FTL作为中间层

FTL: flash translation layer

FTL从主机文件系统接收块级请求（LSN, size），经过FTL的处理，产生flash的各种控制命令

76

FTL

FTL由三部分组成:

- Address mapping (地址映射)
- Wear leveling (损耗平衡)
- Garbage collection (垃圾回收)

77

Address mapping (地址映射)

上层文件系统发送给SSD的任何读写命令包括两个部分（LSN，size）

LSN是逻辑扇区号，对于文件系统而言，它所看到的存储空间是一个线性的连续空间。例如，读请求（260，6）表示的是需要读取从扇区号为260的逻辑扇区开始，总共6个扇区。

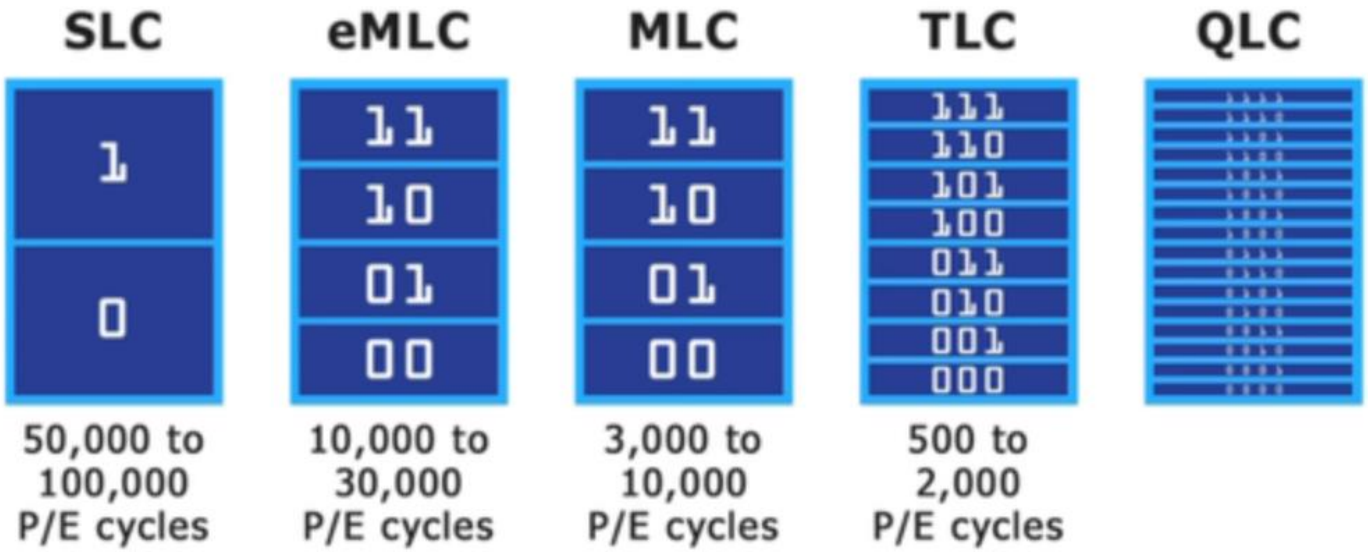
请求到达SSD后，需要经过地址转换，将逻辑扇区转换成NAND FLASH中的物理页号

<package, die, plane, block, page>

78

损耗平衡(Wear-Leveling)

- Flash中每个块都有一定的擦写次数限制。故不能让某一个块被写次数较多，而其他块被写的次数较少。
- 需要找一种方法：使flash中每个块被擦写的次数基本相同。



79

WL的基本方法

动态损耗平衡

在请求到达时，选取擦除次数较少的块作为请求的物理地址。

静态损耗平衡

在运行一段时间后，有些块存放的数据一直没有更新（冷数据），而有些块的数据经常性的更新（热数据）。那些存放冷数据的块的擦除次数远小于存放热数据的块。将冷数据从原块取出，存放在擦除次数过多的块，原来存放冷数据的块被释放出来，接受热数据的擦写。

80

垃圾回收(Garbage Collection)

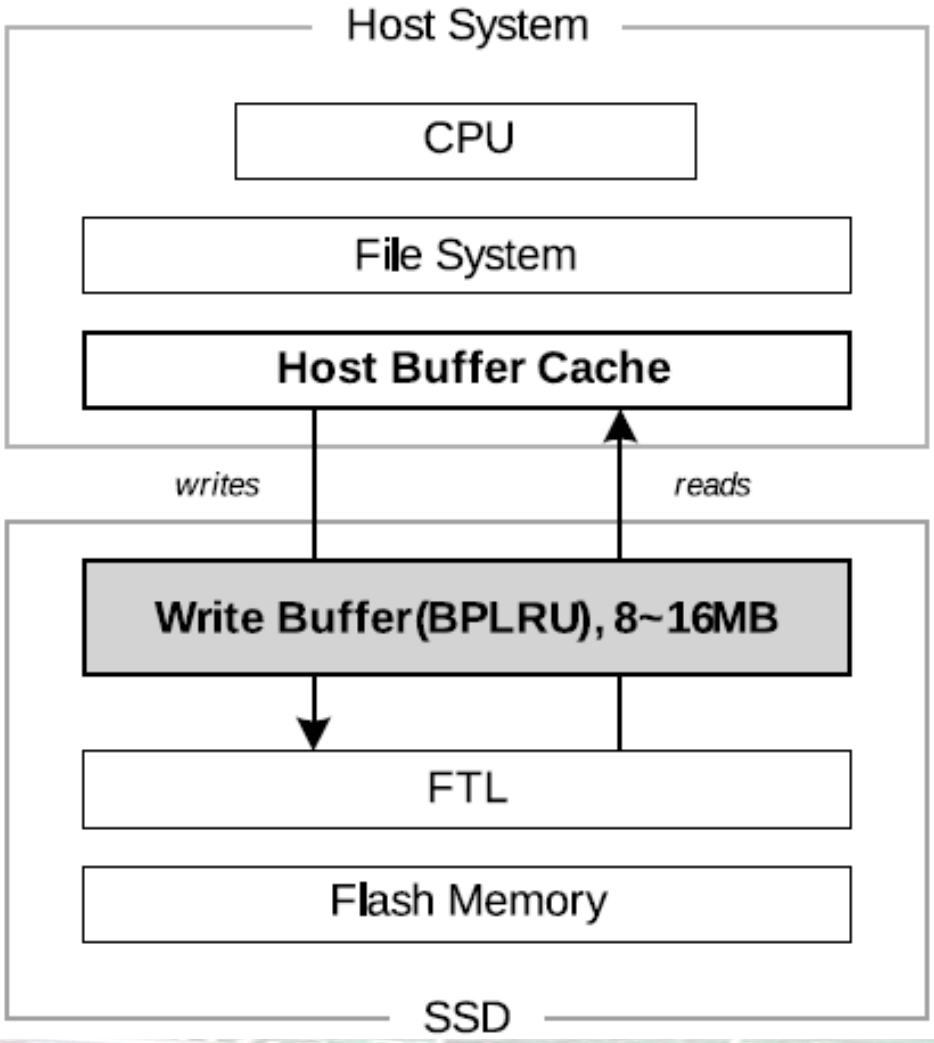
垃圾回收的目的

SSD在使用过程中，会产生大量失效页，在SSD的容量到达一定阈值时，需要调用GC函数，清除所有失效页，以增加可用空间。

81

SSD中buffer策略

好的buffer策略能够提高SSD的整体性能。



82

拓展阅读：F2FS：Flash-Friendly File System

传统的文件系统，比如EXT系列是针对磁盘介质设计的，因为磁盘是可以原地更新的，但是nand flash因为擦除太慢（擦除单元很大），为了速度只能异地更新。如果不考虑这个特性，会导致性能低下。

F2FS主要技术：基于日志追加写，数据冷热分离、区分顺序写和随机写的数据布局等

F2FS的提升主要在随机读写性能中，比EXT4性能好2倍；

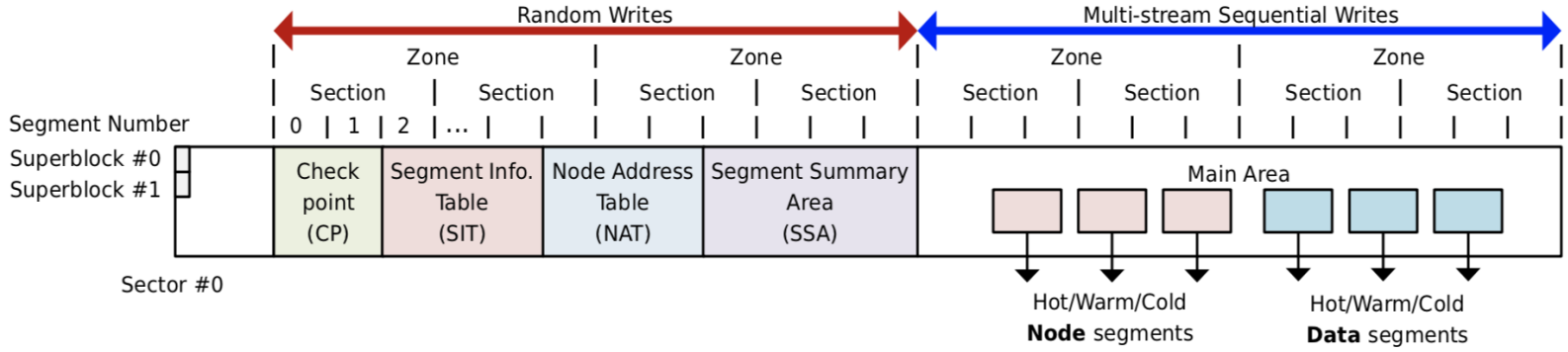


Figure 1: On-disk layout of F2FS.

Hope you enjoyed the OS course!

84