

# 文件系统实施

刘玉峰

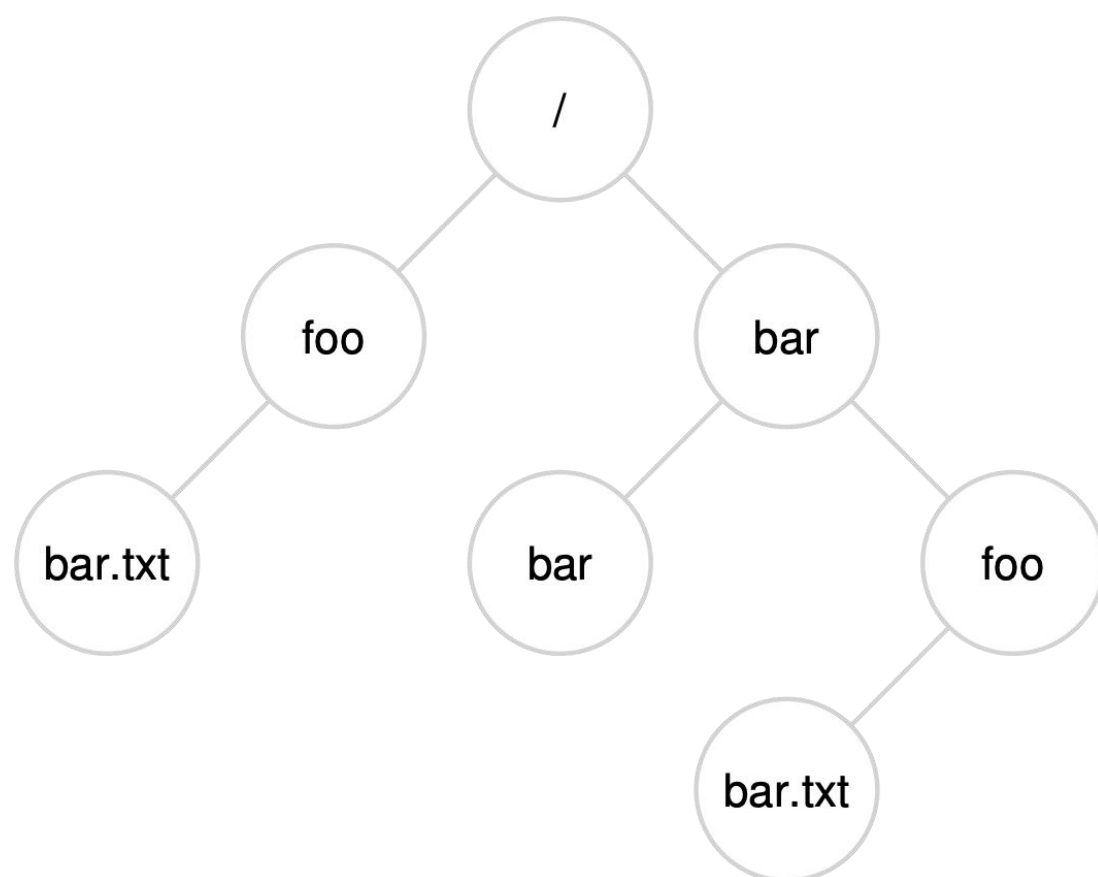
[Fx\\_163.com](http://Fx_163.com)湖南

大学

# 第三十九章——永久存储

- 即使断电也能保持数据完整。
  - 硬盘驱动器
  - 固态存储装置
- 存储虚拟化中的两个关键抽象
  - File: 文件就是一个线性字节数组，每个字节都可以读取或写入
  - 每个文件的低级名称为inode编号
    - 用户不知道此名称。
  - Directory: 目录通常是一个列表，列表中每个条目（用户可读名字，低级名字），每个条目是一个文件或其他目录的入口
  - 一个目录有一个条目（“foo”，“10”）
    - 一个文件“foo”，低层名称为“10”

# 文件和目录



有效文件（绝对路径名）：

/foo/bar.txt  
/bar/foo/bar.txt

有效目录：

/  
/foo  
/bar  
/bar/bar  
/bar/foo

# 文件系统接口

# 创建文件

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
```

程序创建文件（O\_CREAT）

只能写入该文件，因为以（O\_WRONLY）这种方式打开。

如果该文件已经存在，则首先将其截断为零字节大小，删除所有现有内容（O\_TRUNC）。

open（）系统调用返回文件描述符。

文件描述符是一个整数，用于访问文件。

文件描述符只是一个整数，是每个进程私有的，在 UNIX 系统中用于访问文件，类似于 windows 中的句柄。文件描述符可以认为是作为指向文件类型对象的指针。一旦你有这样的对象，就可以调用其他“方法”来访问文件，如 read() 和 write()。

# 读写文件

```
prompt> echo hello > foo  
prompt> cat foo  
hello  
prompt>
```

在这段代码中，我们将程序 `echo` 的输出重定向到文件 `foo`，然后文件中就包含单词 “hello”。然后我们用 `cat` 来查看文件的内容。

**cat**程序如何访问文件**foo**？

我们可以使用**strace**来跟踪程序的系统调用。

# 读写文件

**strace** 工具可以跟踪程序生成的系统调用，查看参数和返回代码

```
prompt> strace catfoo
...
open ("foo", O_RDONLY| O_LARGEFILE) =
read (3, "hello\n", 4096)           = 6 // read() 的调用返回读取的字节数
write (1, "hello\n", 6)              = 6 //文件描述符1: 标准      出
你好
read (3, "", 4096)                  = 0 //文件中没有剩余字节, read() 返回 0
关闭 (3)                            = 0
...
提示>
```

`open`（文件描述符，标志）

返回文件描述符（示例中为3）

文件描述符0、1、2用于标准输入/输出/错误。

`read`（文件描述符，缓冲区指针，缓冲区大小）

返回它读取的字节数

`write`（文件描述符，缓冲区指针，缓冲区大小）返回  
写入的字节数

# 阅读和写作，但不是顺序

```
off_t lseek(int fildes, off_t offset, int whence);
```

第一个参数是熟悉的（一个文件描述符）。

第二个参数是偏移量，它将文件偏移量定位到文件中的特定位置。

第三个参数，由于历史原因而被称为 **whence**，明确地指定了搜索的执行方式。

If whence is SEEK\_SET, the offset is set to offset bytes.

If whence is SEEK\_CUR, the offset is set to its current location plus offset bytes.

If whence is SEEK\_END, the offset is set to the size of the file plus offset bytes.

对于每个进程打开的文件，操作系统都会跟踪一个“当前”偏移量，这将决定在文件中读取或写入时，下一次读取或写入开始的位置



# 立即撰写

## `fsync()`

- 当程序调用 `write()` 时，它只是告诉文件系统：请在将来的某个时刻，将此数据写入持久存储。出于性能的原因，文件系统会将这些写入在内存中缓冲（buffer）一段时间（例如 5s 或 30s）。在稍后的时间点，写入将实际发送到存储设备。
- 当进程针对特定文件描述符调用 `fsync()` 时，文件系统通过强制将所有脏（dirty）数据（即尚未写入的）写入磁盘。一旦所有这些写入完成，`fsync()` 例程就会返回。
- `int fsync(int fd)`
  - 文件系统强制所有脏（即，还未写入）数据到磁盘，以用于文件描述所引用的文件。
  - `fsync()` 在所有这些写入完成后返回。

# 立即撰写

## fsync ()

- fsync () 的一个例子。

```
int fd = open ("foo", O_CREAT| O_错误  
|O_TRUNC); assert (fd>-1)  
int rc = write (fd, buffer, size);  
断言 (rc ==size);  
rc = fsync  
    (fd); assert (rc  
    ==0);
```

- 代码打开文件 foo，向它写入一个数据块，然后调用 fsync() 以确保立即强制写入磁盘。一旦 fsync() 返回，应用程序就可以安全地继续前进，知道数据已被保存。

# 重命名文件

- `rename (char* old, char *new)`
- 将文件重命名为其他名称。
- `rename()`调用提供了一个有趣的保证：它（通常）是一个原子（atomic）调用，不论系统是否崩溃。如果系统在重命名期间崩溃，文件将被命名为旧名称或新名称，不会出现奇怪的中间状态。
- **Ex)** 从foo更改为bar:

```
prompt> mv foo bar //mv使用系统调用rename ()
```

- **Ex)** 编辑器更新文件并确保新文件包含原有内容 和插入行的方式如下How to update a file atomically:

```
int fd = open ("foo.txt.tmp", O_WRONLY | O_CREAT | return
(int, int, int, int) ; //写出新版本的文件 fsync (fd) ;
return () ;
rename ("foo.txt", "foo.txt") ;
```

将文件的新版本写入临时名称（foo.txt.tmp），使用 `fsync()` 将其强制写入磁盘。然后，当应用程序确定新文件的元数据和内容在磁盘上，就将临时文件重命名为原有文件的名称。最后一步自动将新文件交换到位，同时删除旧版本的文件，从而实现原子文件更新

# 获取文件信息

- `stat()` , `fstat()` : 显示文件元数据
- 元数据是关于每个文件的信息。
- Ex) 大小、低级名称、权限、...
- `stat`结构如下:

```
struct stat {dev_t
    st_dev; int_t; /* 包含文件的设备ID*/
    mode_t st_mode; /* inode number*/
                    /* 保护*/

    nlink_t; /* 硬链接数量*/
    id_t id; gid_t /* 所有者的用户ID*/
    t_gid; dev_t /* 所有者的组ID*/
    st_rdev; off_t /* 设备ID (如果是特殊文件) */
    st_size; /* 总大小, 以字节为单位*/
    blksize_t st_blksize; /* 文件系统I/O的块大小 */ blkcnt_t
    st_blocks; /* 分配的块数*/
    time_t st_atime; /* 时间 的 最后 访问*/
    time_t st_mtime; /* 时间 的 最后 修改*/
    time_t st_ctime; /* 时间 的 最后 状态更改*/
};
```

# 获取有关文件的信息（续）

- 要查看stat信息，可以使用命令行工具马上

```
prompt> echo hello >file  
prompt> stat文件
```

文件: 'file'

大小: 6块: 8个IO块: 4096常规文件

设备: 811 h/2065 d输入节点: 67158084友情链接: 1

访问: (0640/-rw-r-----) Uid: (30686/ root) GID: (30686/ remzi)

访问: 2011-05-03 15: 50: -0500  
20.157594748

修改: 2011-05-03 15: 50: -0500  
20.157594748

变更: 2011-05-03 15: 50: -0500  
20.157594748

- 文件系统将此类信息保存在一个inode中结构。

# 删除文件

- rm是删除文件的Linux命令
- rm调用`unlink()`删除文件。

```
prompt> strace rmfoo
...
unlink("foo")      = 0//成功后返回0
...
提示>
```

为什么调用`unlink()`？而不是“删除”  
我们可以稍后得到答案。

# 制作目录

- `mkdir()`：创建一个目录

```
prompt> strace mkdirfoo
...
mkdir("foo", 0777)      =0
提示符>
```

- 创建目录时，该目录为[空](#)。
- 空目录有两个条目：`.`（本身），`..`（父）

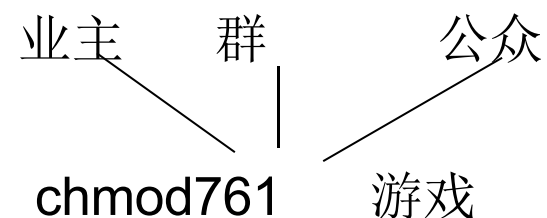
```
提示符> ls-a
./ ../
prompt> ls-al
total 8
drwxr-x--- 2 remzi remzi      4月30日16: 17
drwxr-x--- 26 remzi remzi 4096 Apr 30 16: 17../
```

# 访问（控制）列表和组

- 访问模式：读、写、执行
- 三类用户

			RWX
a) 所有者访问	7	⇒	111 RWX
b) 组访问	6	⇒	— — 零 RWX
C. 公众访问	1	⇒	0 0 1

- 要求manager创建一个组（唯一的名称），说G，并添加一些用户到组。
- 对于一个特定的文件（比如游戏）或子目录，定义一个合适的访问权限。



将组附加到文件

CHG    G    游戏  
RP



# 正在读取目录

- 该程序使用了 opendir()、readdir()和 closedir()这 3 个调用来完成工作，使用一个简单的循环一次读取一个目录条目，并打印目录中每个文件的名称和 inode 编号

```
public int findDuplicate (int [] nums) {
    DIR *dp= opendir (".");           //打开当前目录assert (dp!=
    NULL) ;
    struct *d;
    while ( (d = readdir (dp) ) != NULL)    //读取一个目录项
    {
        //打印出每个文件的名称和inode号
        printf ("%d %s", (int) d->d_ino, d->d_name) ;
    }
    return () ;                       //关闭当前目录返回0;
}
```

- structdireent中可用的信息

```
struct dirent {
    焦      d_name[256]; /*文件名 */
    ino_t    d_ino;      /* inodenumbr */
    关闭t    d_off;      /*到下一个直接地址的偏移量 */unsigned
    short    d_reclen;    /* 记录长度 */unsigned char d_type;
    /* 文件类型*/
}
```

# 删除目录

- `rmdir()`：删除目录。
- 要求目录为空。
- 如果对一个非空目录调用`rmdir()`，它将失败。
- 即，只有“。”还有“...”条目。

# 硬链接

- link (旧路径名, 新路径名)
  - 将新文件名链接到旧文件名
  - 创建引用 *同一文件的另一种方式*
  - 命令行链接程序: ln

```
prompt> echo hello > file
prompt> catfile
你好
prompt> ln file file2 //创建一个硬链接, 将文件链接到file 2
prompt> cat file2
Hello
```

# 硬链接（续）

- link的工作方式：
  - 在目录中创建另一个名称。
  - 将其引用到与原始文件相同的inode号。
  - 不会以任何方式复制文件。
- 然后，我们现在只有两个人名（file和file2），它们都引用同一个文件。

# 硬链接（续）

- link（）的结果

```
prompt> ls -i filefile2
67158084文件/* inode值为67158084*/
67158084 file 2/* inode值为67158084*/
提示>
```

- 两个文件有相同的inode号，但有两个人名（file，file2）。
- 在创建文件的硬链接之后，在文件系统中，原有文件名（file）和新创建的文件名（file2）之间没有区别。实际上，它们都只是指向文件底层元数据的链接，可以在inode 编号 67158084 中找到。

# 硬链接（续）

```
prompt> rm file  
removed 'file'  
prompt> catfile2 //仍然访问文件  
hello
```

- 当文件系统取消链接文件时，它检查 inode 号中的引用计数（reference count）。该引用计数（有时称为链接计数，link count）允许文件系统跟踪有多少不同的文件 名已链接到这个 inode。
- 调用 `unlink()` 时，会删除可读的名称（正在删除的文件）与给定 inode 号之间的“链接”，并减少引用计数。只有当引用计数达到零时，文件系统才会释放 inode 和相关数据块，从而真正“删除”该文件。

# 硬链接（续）

- `unlink()` 的结果
- `stat()` 显示文件的引用计数。

```
prompt> echo hello > file /* 创建文件 */
prompt> statfile
...联系电话: 67158084 一... /* 链接计数为% 1 */
prompt> ln filefile2 /* 硬链接文件2 */
prompt> stat文件
...联系电话: 67158084 二... /* 链接 计数为2 */
prompt> statfile2
...联系电话: 67158084 二... /* 链接 计数为2 */
prompt> ln file2file3 /* 硬的 链接文件3 */
prompt> stat文件
...电话: 67158084 友情链 3 ... /* 链接 计数为3 */
接:
prompt> rmfile /* 删除文件*/
prompt> statfile2
...电话: 67158084 友情链 二... /* 链接计数为2 */
接:
prompt> rmfile2 /* remove file2*/
prompt> statfile3
...电话: 67158084 友情链 一... /* 链接计数为% 1 */
prompt> rmfile3 接:
```

# 软链接（SoftLink）

- 符号链接比硬链接更有用。
  - Hard Link **cannot create to a directory**（防止出现环）。
  - 硬链接**无法创建到其他分区的文件**。
    - 因为inode号只在文件系统中是唯一的。

```
prompt> echo hello > file
prompt> ln -s file file2 /* option -s: 创建一个符号链接, */
prompt> catfile2
你好
```

- 创建符号链接: `ln -s`



# 符号链接（续）

- 符号链接和硬链接有什么区别？
  - 第一个区别是符号链接本身实际上是一个不同类型的文件。我们已经讨论过常规文件和目录。符号链接是文件系统知道的第三种类型。

```
prompt> stat文件
...常规文件...
prompt> stat file2
...符号链接
```

//实际上是一个不同类型的文件

- 符号链接（file2）的大小为4字节。

```
prompt>ls -
铝
drwxr-x--  2 雷姆齐 雷姆齐    29 五月 3 19点10分 ./
drwxr-x-- 27 雷姆齐 雷姆齐 4096 五月 3 十五点十四分 ../
-rw-r----- 1 雷姆齐 雷姆齐    6 五月 3 19点10分 锉
```

// 目录

// 正规档按

- 符号链接将链接到的文件的路径名作为链接文件的数据。

# 符号链接（续）

- 如果我们链接到一个更长的路径名，我们的链接文件会更大。

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi 5月6日3 19: 17alongerfilename
remzi remzi 15 May 3 19: 17 file3->alongerfilename
```

# 符号链接（续）

- 悬挂参考
  - 删除原始文件时，符号链接点要注意.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> catfile2
你好
prompt> rmfile          //删除原始文件
prompt> catfile2
cat: file2: 没有这样的文件或目录
```

# 创建和装载文件系统

- `mkfs`工具：创建文件系统
  - 在一个磁盘分区上写入一个空文件 系统.
  - 输入：
    - 设备（诸如磁盘分区，例如，`/dev/sda1`）
    - 文件系统类型（例如，`ext3`）

# 制作和装载文件系统（续）

- mount（）
  - mount 的作用很简单：以现有目录作为目标挂载点（mount point），本质上是将新的文件系统粘贴到目录树的这个点上
  - 我们有一个未挂载的 ext3 文件系统，存储在设备分区/dev/sda1 中，它的内容包括：一个根目录，其中包含两个子目录 a 和 b，每个子目录依次包含一个名为 foo 的文件。假设希望在挂载点 /home/users 上挂载此文件系统
  - **EX样品）**

```
prompt> mount -t ext3 /dev/sda1 /home/users
prompt> ls/home/users
ab
```

- 路径名/home/users/现在指向新挂载目录的根目录。

# 创建和装载文件系统 (续)

- `mountprogram`: 显示系统上安装的内容。

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/tmp类型ext 3 (rw) 上的/dev/sda 5
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs类型afs (rw)
```

- `ext3`: 标准的基于磁盘的文件系统
- `proc`: 用于访问当前进程信息的文件系统
- `tmpfs`: 仅用于临时文件的文件系统
- `AFS`: 分布式文件系统

# 第四十章——实施文件系统

- 介绍一个简单的文件系统实现，称为 VSFS（Very Simple File System，简单文件系统）。它是典型 UNIX 文件系统的简化版本。
- 文件系统是纯软件。与 CPU 和内存虚拟化的开发不同，我们不会添加硬件功能来使文件系统的某些方面更好地工作（但我們需要注意设备特性，以确保文件系统运行良好）。



## THE CRUX: HOW TO IMPLEMENT A SIMPLE FILE SYSTEM

How can we build a simple file system? What structures are needed on the disk? What do they need to track? How are they accessed?

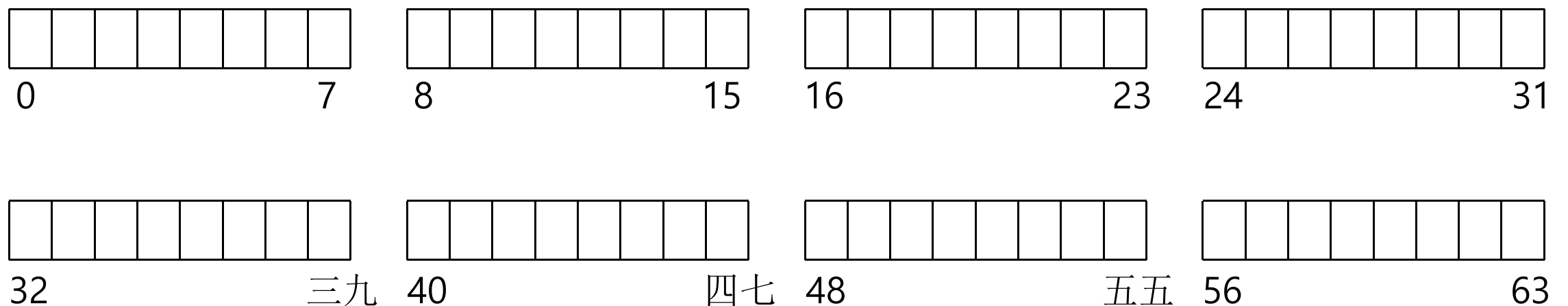
# 思考之道

- 第一个方面是文件系统的数据结构（data structure）。换言之，文件系统在磁盘上使用哪些类型的结构来组织其数据和元数据？
- 文件系统的第二个方面是访问方法（access method）。如何将进程发出的调用，如 `open()`、`read()`、`write()` 等，映射到它的结构上？在执行特定系统调用期间读取哪些结构？改写哪些结构？所有这些步骤的执行效率如何？

# 总体组织

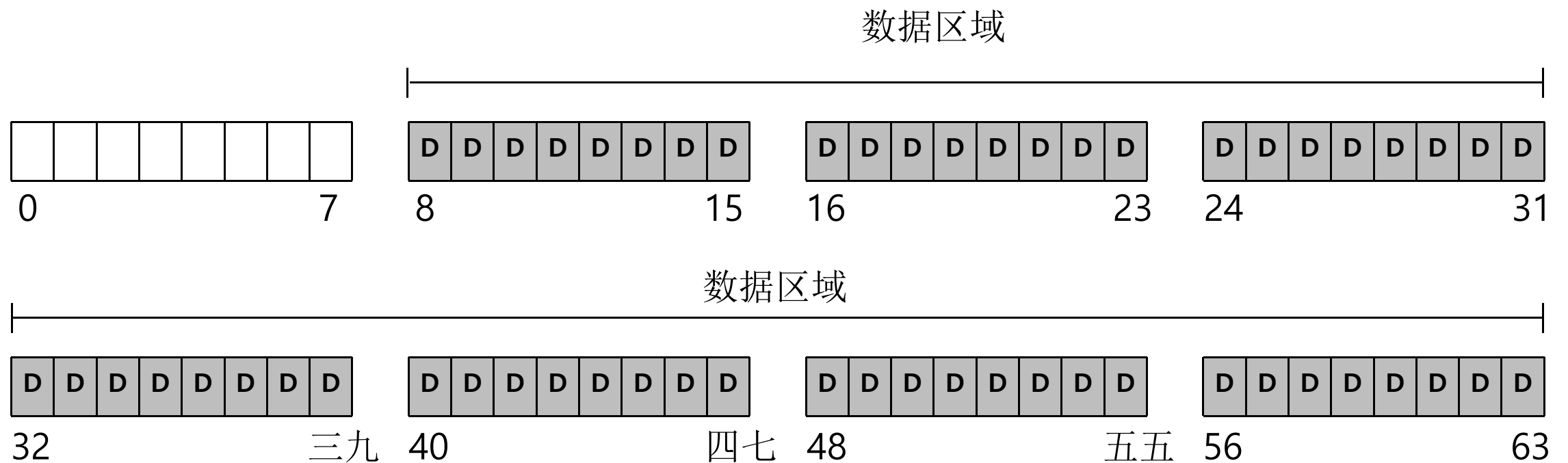
- VSFS 文件系统在磁盘上的数据结构的整体组织。
- 第一件事是将磁盘分成块（block）。选择常用的 4KB。

磁盘分区被认为是一系列块，每块大小为 4KB。在大小为 N 个 4KB 块的分区中，这些块的地址为从 0 到 N-1。  
目前假设我们有一个非常小的磁 盘，只有 64 块



# 文件系统中的数据区域

- 任何文件系统中的大多数空间都是（并且应该是）用户数据。我们将用于存放用户数据的磁盘区域称为数据区域（data region），简单起见，将磁盘的固定部分留给这些块，例如磁盘上 64 个块的最后 56 个



我们如何在文件系统中存储这些**inode**?

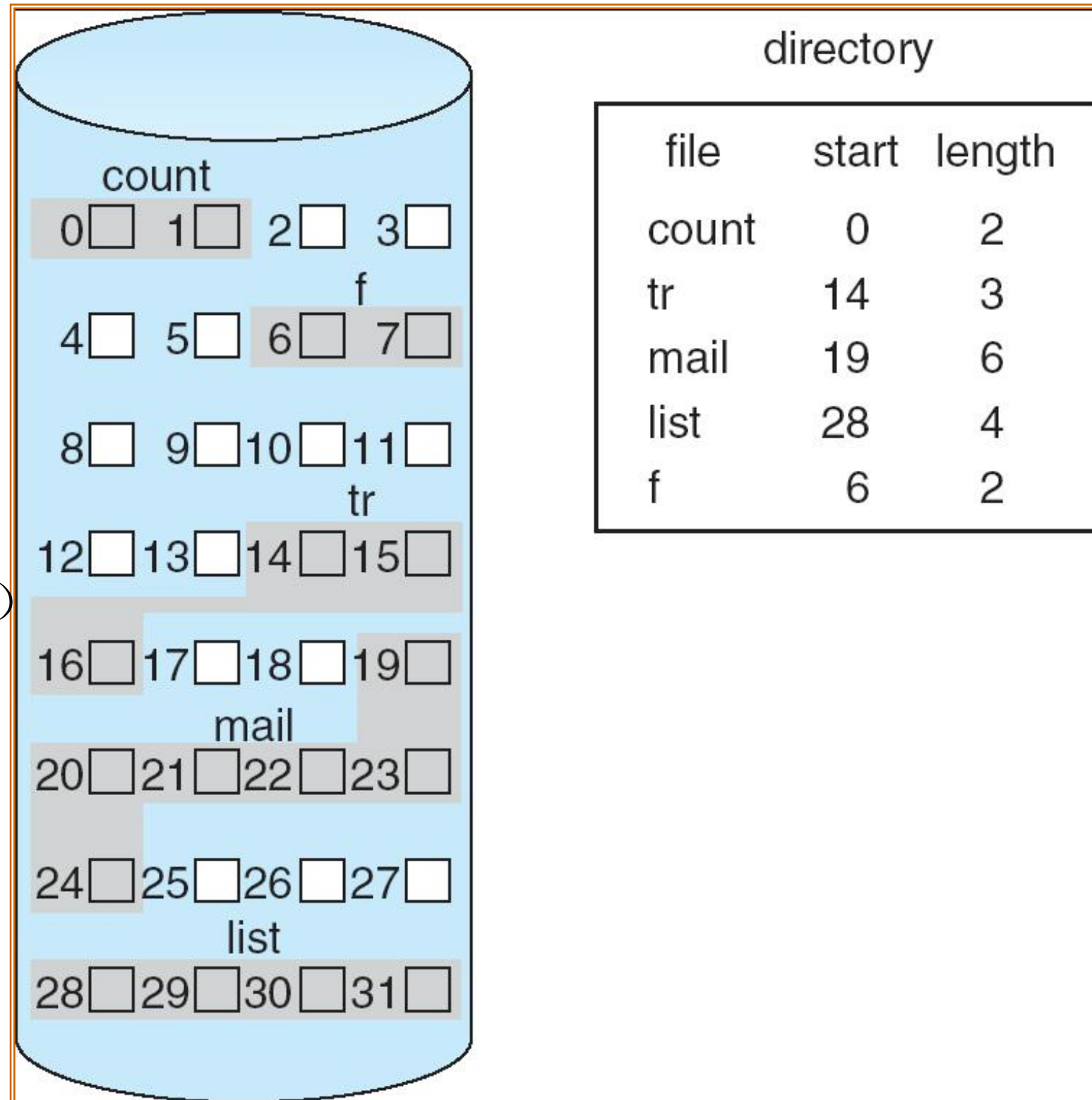
# 文件物理结构之连续分配

## （连续分配）

- 每个文件占据磁盘上的一组连续的块
- 特点：
  - 简单 - 只需要记录文件的起始位置（块号）及长度。
  - 访问文件很容易，所需的寻道时间也最少
- 存在的问题
  - 为新文件找空间比较困难（类似于内存分配中的连续内存分配方式）
  - 文件很难增长

# 连续分配

- 1、简单
- 2、支持直接访问
- 3、动态存储分配  
(首次, 最佳, 最差)



# 基于扩展的连续分配方法

- 许多新的文件系统采用一种修正的连续分配方法
- 该方法开始分配一块连续空间，当空间不够时，另一块被称为扩展的连续空间会添加到原来的分配中。
- 文件块的位置就成为开始地址、块数、加上一个指向下一扩展的指针。

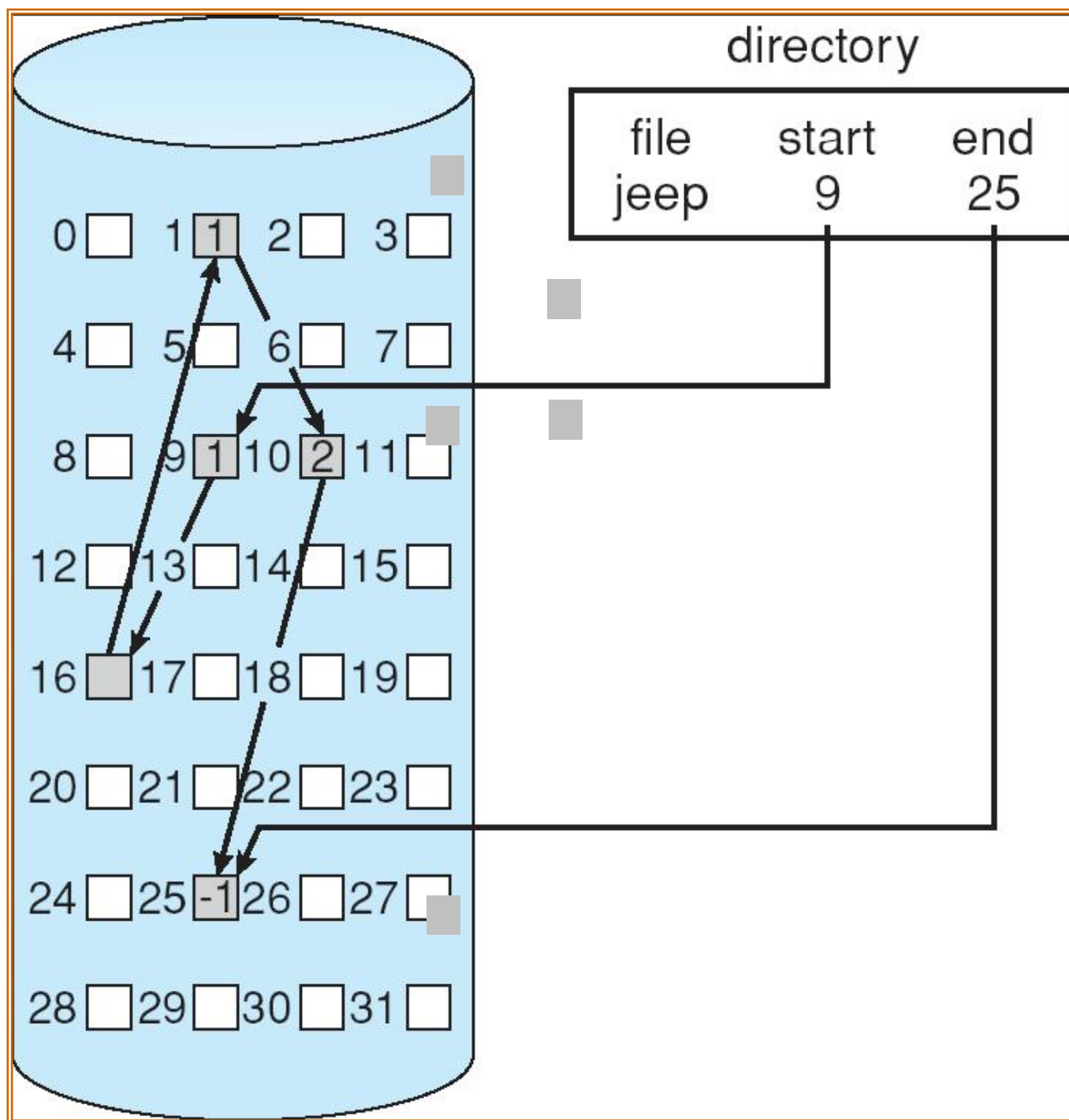
# 文件物理结构(二) 链接分配 (联系分配)

- 每个文件是磁盘块的**链表**;磁盘块分布在磁盘的任何地方。
- 优点:
  - 简单 - 只需起始位置
  - 文件创建与增长容易
- 缺点:
  - 不能随机访问
  - 块与块之间的链接指针需要占用空间
    - 簇: 将多个连续块组成簇, 磁盘以簇为单位进行分配
  - 存在可靠性问题



# 链接分配

- 1、简单
- 2、不会有外部碎片
- 3、不支持直接访问
- 4、链接分配方案的变种——FAT

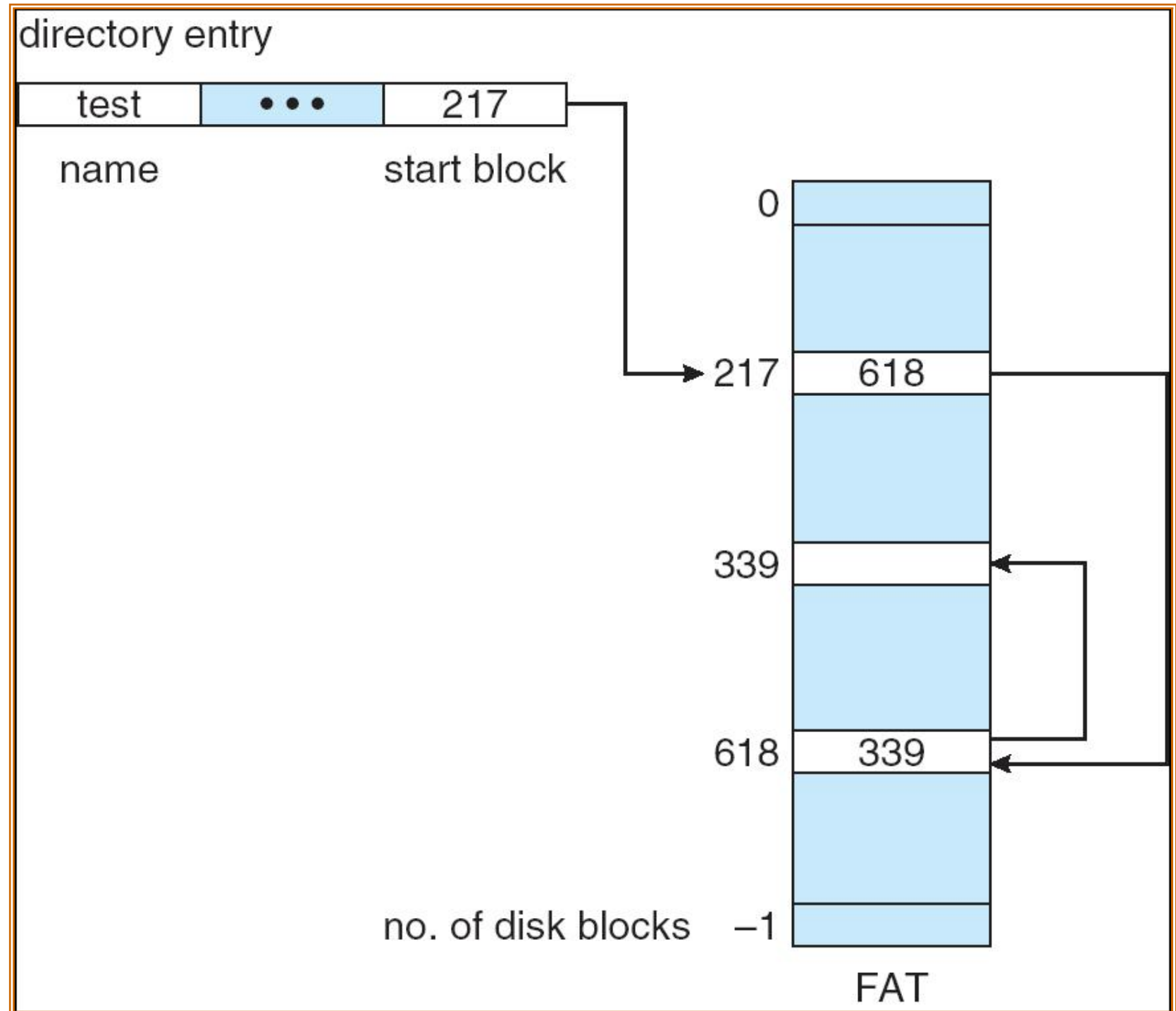


# 链接分配方法的变种-文件分配表 (FAT)

- 每个分区的开始部分用于存储该FAT表。
- 每个磁盘块在该表中有一项，该表可以通过块号来索引。
- 目录条目中含有文件首块的块号码。根据块号码索引的FAT条目包含文件下一块的号码。这种链会一直继续到最后一块，该块对应FAT条目的值为文件结束值。未使用的块用0值来表示。
- 为文件分配一个新的块只要简单地找到第一个值为0的FAT条目，用新块的地址替换前面文件结束值，用文件结束值替代0。

# 文件分配表FAT

如果不对FAT采用缓存，FAT分配方案可能导致大量的磁头寻道时间。但通过读入FAT信息，磁盘能找到任何块的位置，从而实现随机访问。



假定一个文件系统组织方式与MS-DOS相似，在FAT中可有64K个指针，磁盘的盘块大小为512B，试问该文件系统能否用于一个512MB的磁盘？

假定一个文件系统组织方式与MS-DOS相似，在FAT中可有64K个指针，磁盘的盘块大小为512B，试问该文件系统能否用于一个512MB的磁盘？

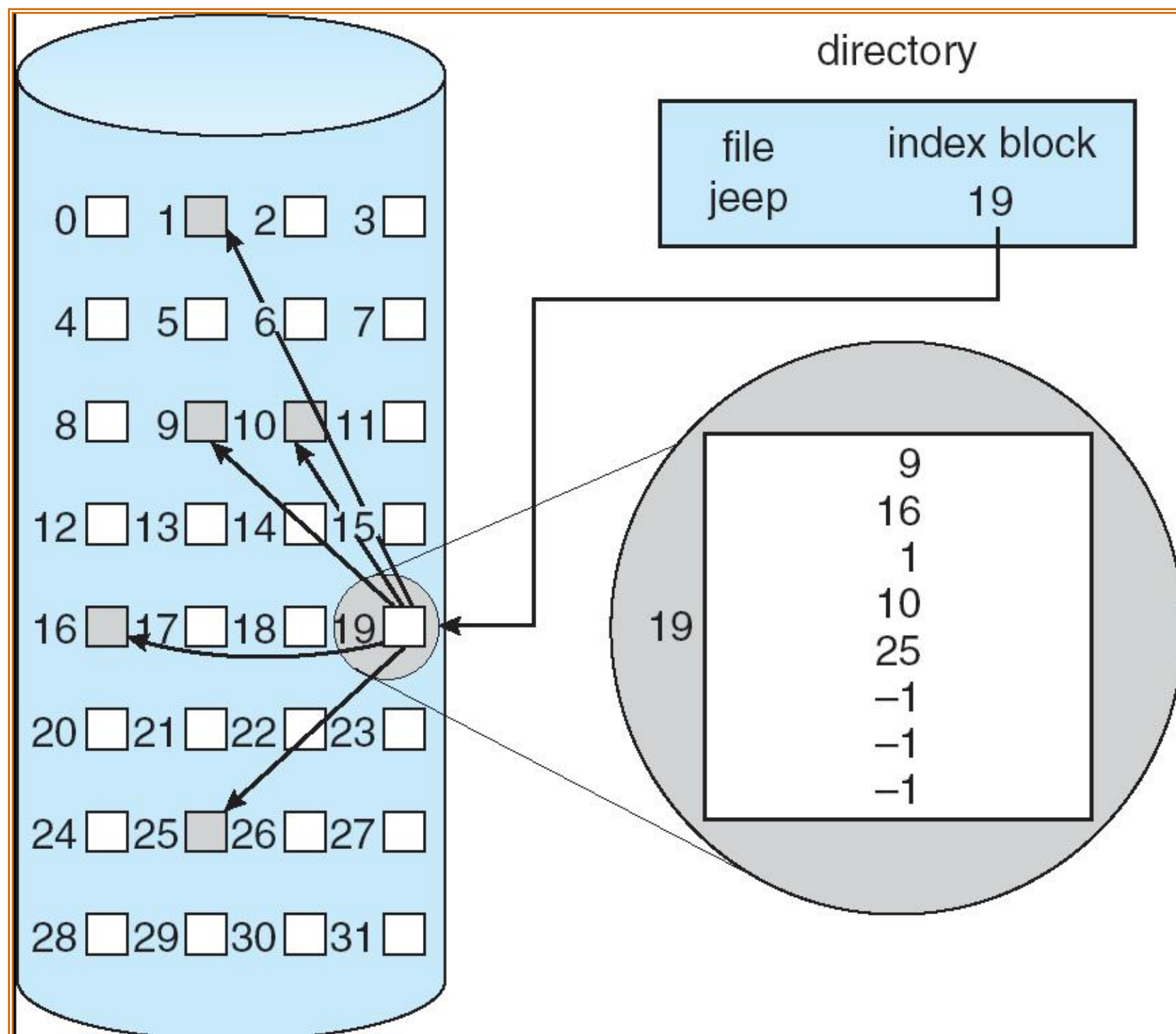
答:有 $512\text{MB}/512\text{B} = 1\text{M}$ 个盘块，64K个指针不够

# 文件物理结构(三) 索引分配(indexed allocation)

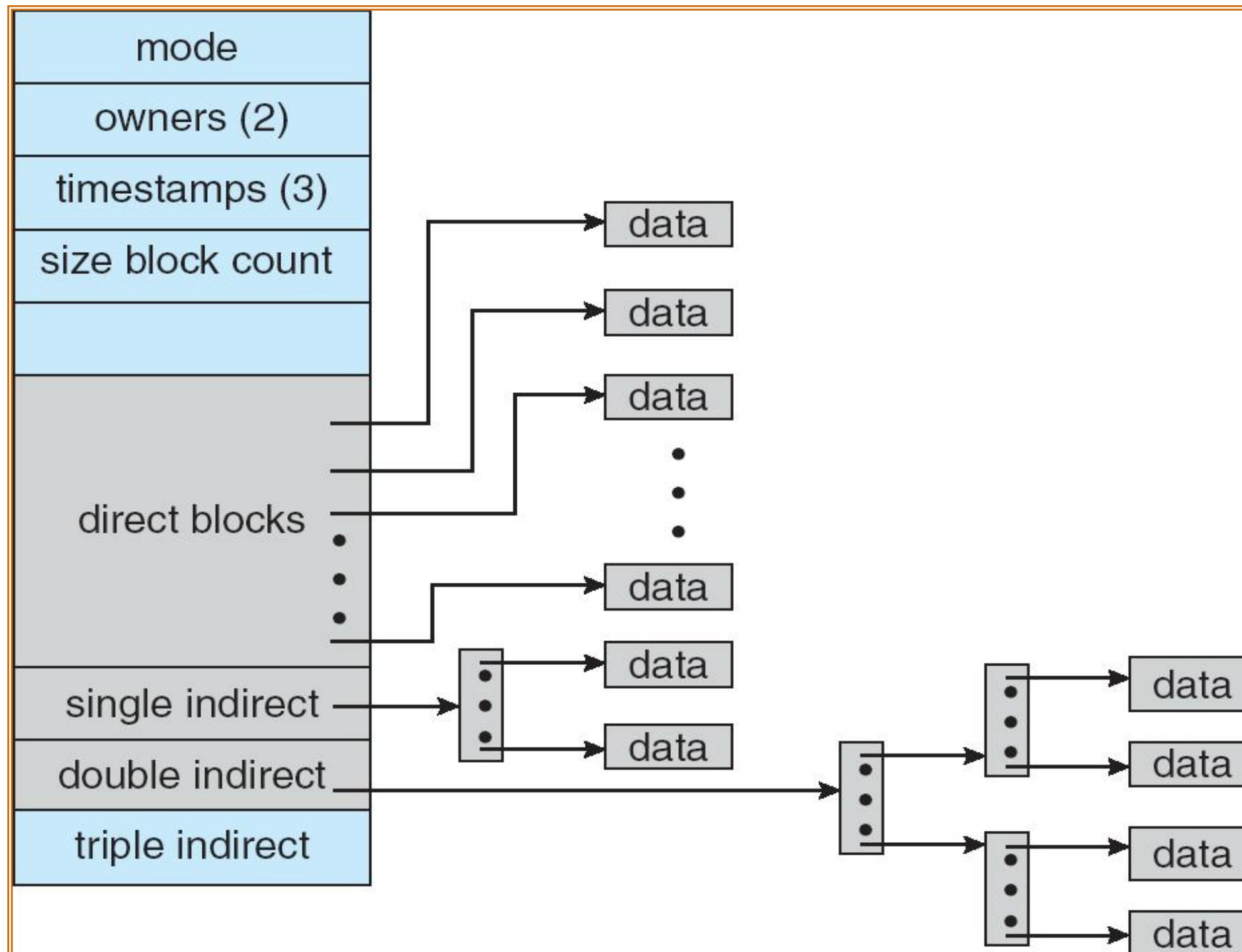
- 将所有数据块指针集中到索引块中
  - 索引块中的第i个条目指向文件的第i块。
  - 目录条目包括索引块的地址
- 索引分配支持直接访问，且没有外部碎片问题
- 索引块本身可能会浪费空间
  - 链接方案：一个索引块通常为一个磁盘块。对于大文件，可以将多个索引块链接起来。
  - 多层索引：类似于内存的间接寻址方式（一级、二级间接…）
  - 组合方案：如Unix的inode

# 索引分配

- 1、需要索引表
- 2、支持直接访问
- 3、无外部碎片
- 4、索引有开销

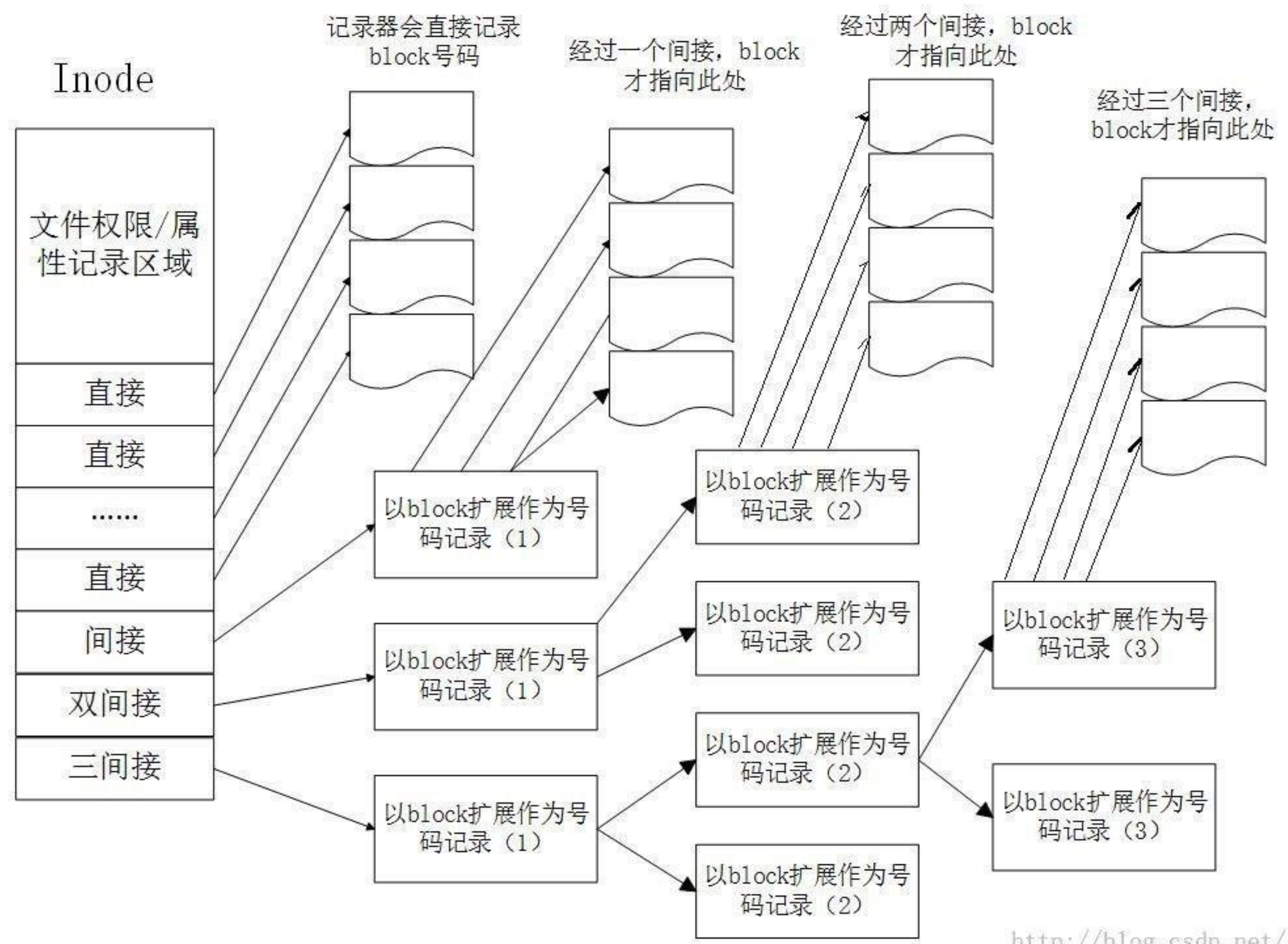


# UNIX信息节点（每个数据块4K字节）





inode结构示意图



# 索引文件

- 优点：
  - 保持了链接结构的优点，又解决了其缺点：
  - 即能顺序存取，又能随机存取
  - 满足了文件动态增长、插入删除的要求
  - 能充分利用外存空间
- 缺点：
  - 索引表本身带来了系统开销
  - 如：内外存空间，存取时间

设定一个文件的i节点为128字节，文件的状态信息占用了68个字节;一个盘块指针为4字节长，每块的大小为8K。使用直接指针、一次间接指针、二次间接指针、三次间接指针分别可以表示多大的文件?

设定一个文件的i节点为128字节，文件的状态信息占用了68个字节;一个盘块指针为4字节长，每块的大小为8K。使用直接指针、一次间接指针、二次间接指针、三次间接指针分别可以表示多大的文件?

答案:

直接指针项数:  $(128-68)/4-3=12$  (个)  $12*80K=960KB$

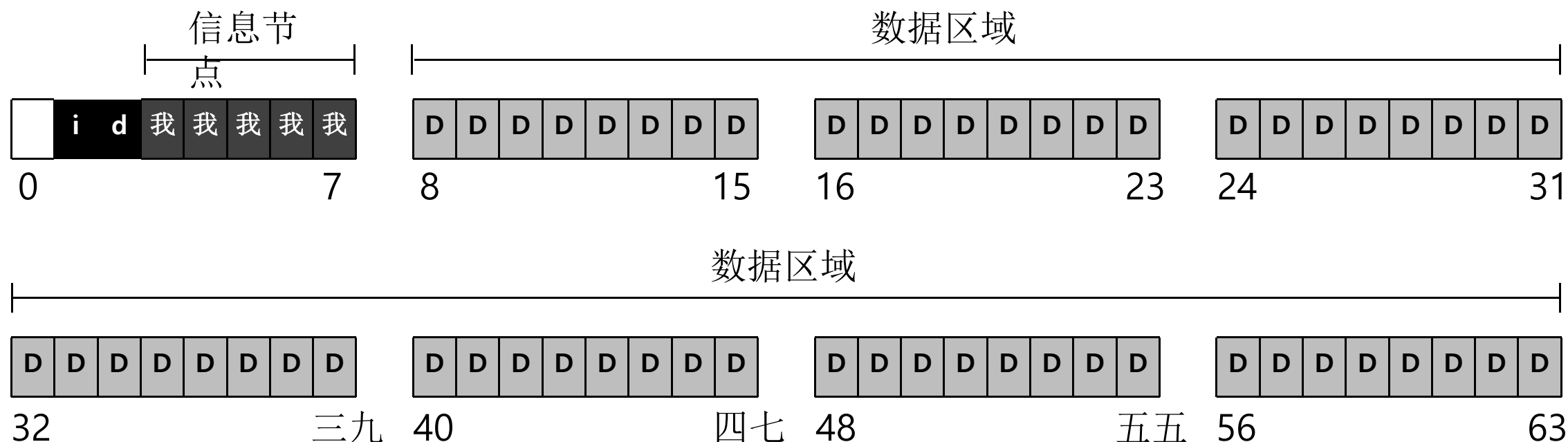
一次间接指针:  $(8K/4) * 8K=16MB$

二次间接指针:  $2K*2K*8K=32G$

三次间接:  $2K*2K*2K*8K=16TB$

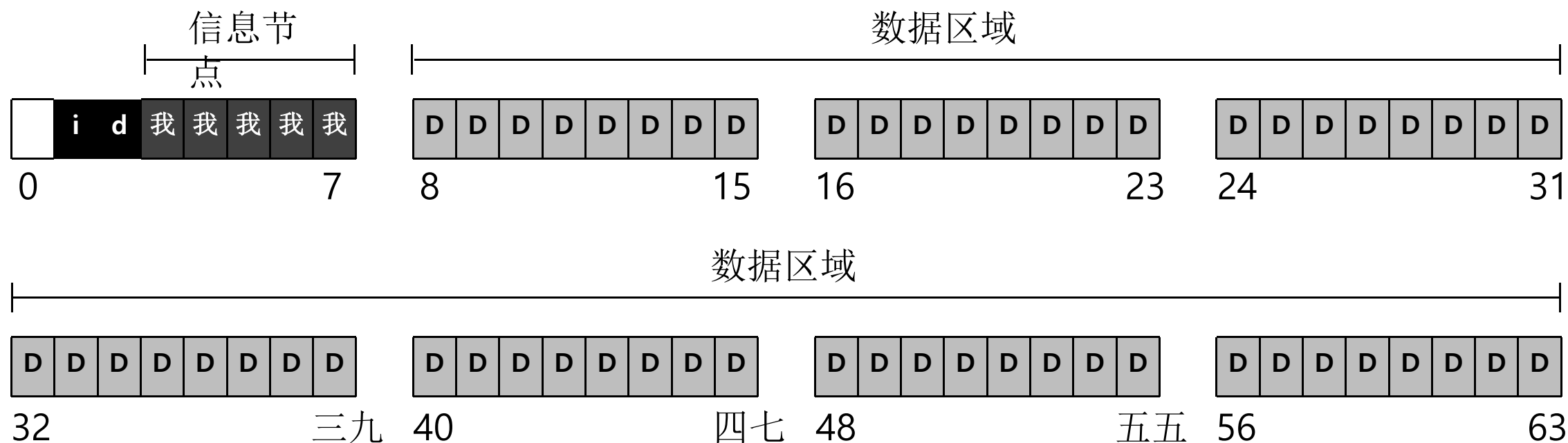
# 文件系统中的索引节点表

- 文件系统必须记录每个文件的信息。该信息是元数据（metadata）的关键部分，并且记录诸如文件包含哪些数据块（在数据区域中）、文件的大小，其所有者和访问权限、访问和修改时间以及其他类似信息的事情。为了存储这些信息，文件系统通常有一个名为 inode 的结构
- 为了存放 inode，我们还需要在磁盘上留出一些空间。我们将这部分磁盘称为 inode 表（inode table），它只是保存了一个磁盘上 inode 的数组。因此，假设我们将 64 个块中的 5 块用于 inode
- inode 通常不是那么大，例如，只有 128 或 256 字节。假设每个 inode 有 256 字节，一个 4KB 块可以容纳 16 个 inode，而我们上面的文件系统则包含 80 个 inode（这个数字表示文件系统中可以拥有的最大文件数量）。



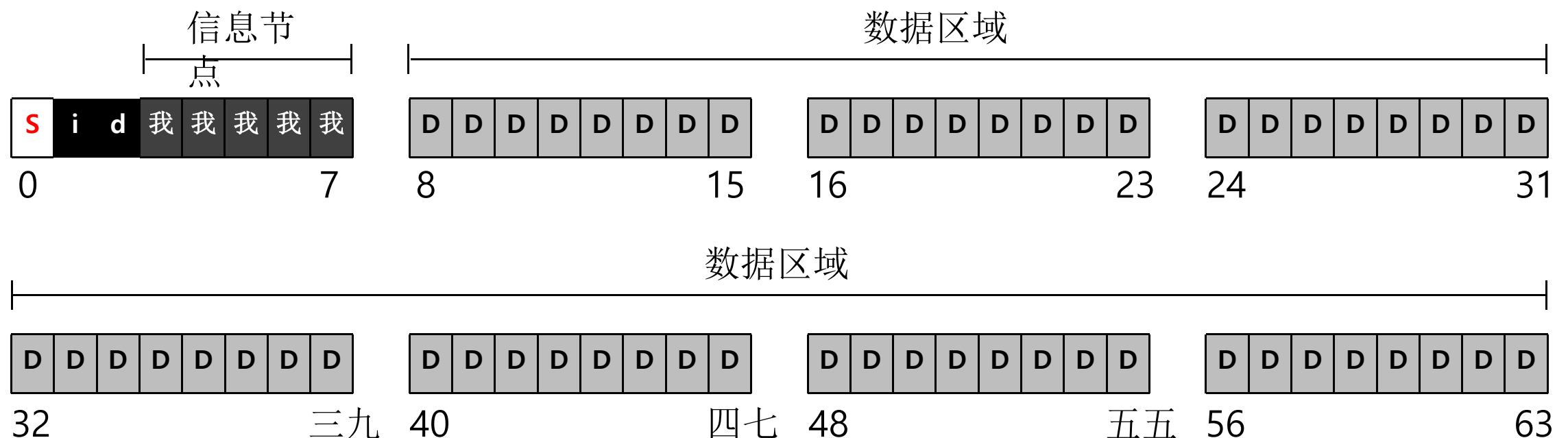
# 文件系统中的索引节点表

- 需要某种方法来记录 inode 或数据块是空闲还是已分配。因此，这种分配结构（allocation structure）是所有文件系统中必需的部分
- 位图是一种简单的结构：每个位用于指示相应的对象/块是空闲（0）还是正在使用（1）。因此新的磁盘布局如下，包含 **inode 位图（i）** 和 **数据位图（d）**
- 这样的位图可以记录（4K \* 8）对象是否分配，但我们只有 80 个 inode 和 56 个数据块。



# 文件系统中的索引节点表

- 超级块包含关于该特定文件系统的信息，包括例如文件系统中有多少个 inode 和数据块（在这个例子中分别为 80 和 56）、inode 表的开始位置（块 3）等等。下图中用 S 表示。
- 可能还包括一些幻数，来标识文件系统类型（在本例中为 VSFS）



# 文件组织: inode

- 每个 inode 都由一个数字（称为 inumber）隐式引用，我们之前称之为文件的低级名称（low-level name）。在 VSFS（和其他简单的文件系统）中，给定一个 inumber，你应该能够直接计算磁盘上相应节点的位置。
- VSFS 的 inode 表：大小为 20KB（5 个 4KB 块），因此由 80 个 inode（假设每个 inode 为 256 字节）组成
- 进一步假设 inode 区域从 12KB 开始（即超级块从 0KB 开始，inode 位图在 4KB 地址，数据位图在 8KB，因此 inode 表紧随其后）。
- Ex) inode number: 32
  - 文件系统首先会计算 inode 区域的偏移量（ $32 \times \text{inode 的大小}$ ，即 8192）
  - 它加上磁盘 inode 表的起始地址（ $\text{inodeStartAddr} = 12\text{KB}$ ），从而得到希望的 inode 块的正确字节地址：20KB

## 索引节点表

			iBlock0				iBlock1				iBlock2				iBlock3				iBlock4			
超级	i-bmap	d-bmap	0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67
			4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71
			8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75
			12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79
0KB	4KB	8KB	1 2KB	1 6KB	2 0KB	2 4KB	2 8KB	3 2KB														



# 文件组织：inode

- 磁盘不是按字节可寻址的，而是由大量可寻址扇区 组成，通常是 512 字节。因此，为了获取包含索引节点 32 的索引节点块，文件系统将向节点（即 40）发出一个读取请求，取得期望的 inode 块
- inode 块的扇区地址 iaddr 可以计算如下：
  - blk = (innumber \* sizeof (inode\_t) ) / blockSize;
  - sector = ( (blk \* blockSize) + inodeStartAddr) / sectorSize;

索引节点表

			iBlock0				iBlock1				iBlock2				iBlock3				iBlock4			
超级	i-bmap	d-bmap	0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67
			4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71
			8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75
			12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79
0KB	4KB	8KB	1 2KB	1 6KB	2 0KB	2 4KB	2 8KB	3 2KB														

表 40.1

ext2 的 inode

大小 (字节)	名称	inode 字段的用途
2	mode	该文件是否可以读/写/执行
2	uid	谁拥有该文件
4	size	该文件有多少字节
4	time	该文件最近的访问时间是什么时候
4	ctime	该文件的创建时间是什么时候
4	mtime	该文件最近的修改时间是什么时候
4	dtime	该 inode 被删除的时间是什么时候
2	gid	该文件属于哪个分组
2	links count	该文件有多少硬链接
4	blocks	为该文件分配了多少块
4	flags	ext2 将如何使用该 inode
4	osd1	OS 相关的字段
60	block	一组磁盘指针 (共 15 个)
4	generation	文件版本 (用于 NFS)
4	file acl	一种新的许可模式, 除了 mode 位
4	dir acl	称为访问控制列表
4	faddr	未支持字段
12	i osd2	另一个 OS 相关字段

# Linux磁盘文件系统布局

MBR即主引导记录，它记录了整个硬盘的分区信息。在硬盘做分区动作时，保存在被激活的分区（一般是将C区激活）



- 磁盘看成是以块为单位的连续存储空间。



在磁盘上，文件系统可能包括如下信息：

如何启动所存储的操作系统

总的块数

空闲块的数目和位置

目录结构以及各个具体文件等

# 多级索引

如何支持更大的文件？ 多级索引

在UNIX中，每个i节点中有10个直接地址和一、二、三级间接索引。若每个盘块512B，每个盘块地址4B，则一个1MB的文件分别占用多少间接盘块？25MB的文件呢？

在UNIX中，每个i节点中有10个直接地址和一、二、三级间接索引。若每个盘块 512B，每个盘块地址4B，则一个1MB的文件分别占用多少间接盘块？ 25MB的文件呢？

10个直接盘块存放的容量为： $10 \times 512\text{B} / 1024 = 5\text{KB}$

一个盘块中可放的盘块地址数为： $512\text{B} / 4\text{B} = 128$

一次间接索引存放的容量为： $128 \times 512\text{B} / 1024 = 64\text{KB}$

二次间接索引存放的容量为： $128 \times 128 \times 512\text{B} / 1024 = 8192\text{KB}$

三次间接索引存放的容量为： $128 \times 128 \times 128 \times 512\text{B} / 1024 = 1048576\text{KB}$

1MB为1024KB， $1024\text{KB} - 64\text{KB} - 5\text{KB} = 955\text{KB}$ ，需要二次间接盘块为  
 $955 \times 1024\text{B} / 512\text{B} = 1910$

1MB的文件分别占用1910个二次间接盘块，128个一次间接块和10个直接块

61

25MB为： $25 \times 1024\text{KB} - 64\text{KB} - 5\text{KB} - 8192\text{KB} = 17339\text{KB}$ ,

$17339 \times 1024\text{B} / 512\text{B} = 34678$

所以25MB的文件分别占用128个一次间接盘块和 $128^2 = 16384$ 个二次间接盘块、  
34678个三次间接盘块

# 多层次指数？

- 使用少量的直接指针（12 是一个典型的数字），inode 可以直接指向 48KB 的数据，需要一个（或多个）间接块来处理较大的文件

表 40.2

文件系统测量汇总

大多数文件很小	大约 2KB 是常见大小
平均文件大小在增长	几乎平均增长 200KB
大多数字节保存在大文件中	少数大文件使用了大部分空间
文件系统包含许多文件	几乎平均 100KB
文件系统大约一半是满的	尽管磁盘在增长，文件系统仍保持约 50% 是满的
目录通常很小	许多只有少量条目，大多数少于 20 个条目

- 有 12 个直接指针，以及一个间接块和一个双重间接块。假设块大小为 4KB，并且指针为 4 字节，则该结构可以容纳一个刚好超过 4GB 的文件，即  $(12 + 1024 + 1024^2) \times 4\text{KB}$ 。
- 增加一个三重间接块，你是否能弄清楚支持多大的文件？



# 目录组织

- 目录包含（**条目名称、信息节点号**）对的列表。
- 每个目录都有两个额外的文件。**"点"表示当前目录，  
.."dot-dot"表示父目录**
  - 例如，`dir`有三个文件（`foo`，`bar`，`foobar`）
- 删除一个文件（例如调用 `unlink()`）会在目录中间留下一段空白空间，因此应该有一些方法来标记它（例如，用一个保留的 `inode` 号，比如 0）。
- XFS以 B 树形式存储目录
- 文件系统将目录视为特殊类型的文件。因此，目录有一个 `inode`

伊努姆 | 雷克伦 | 斯特伦 | 姓名

5	4	2	.
2	4	3	..
12	4	4	福
13	4	4	酒吧
24	8	7	福巴

目录的磁盘上



# 自由空间管理

- 文件系统必须记录哪些 inode 和数据块是空闲的，哪些不是，这样在分配新文件或目录时，就可以为它找到空间。因此，空闲空间管理（free space management）对于所有文件系统都很重要。
- 当我们创建一个文件时，我们必须为该文件分配一个 inode。文件系统将通过位图搜索一个空闲的内容，并将其分配给该文件。文件系统必须将 inode 标记为已使用（用 1），并最终用正确的信息更新磁盘上的位图。
- 因此，自由空间管理对所有文件系统都很重要。我们有两个简单的位图来完成这个任务。

# 空闲空间管理

- 位向量 (n块)
  - $\text{bit}[i] = 0 \rightarrow \text{block}[i]$  空闲
  - $\text{bit}[i] = 1 \rightarrow \text{block}[i]$  被占用

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	0	0	1	1	1	0	0	1	0	1	1	1	0
1	0	0	0	1	1	1	1	1	1	0	0	0	0	1	1	1
2	1	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0
3																
⋮																
15																
位示图																

盘块号转换成位于图中的行号和列号。转换公式为：

$$i = b \text{ DIV } n$$

$$j = b \text{ MOD } n$$

# 访问路径：读和写

- 我们假设文件系统已经挂载，因此超级块已经在内存中。其他所有内容（如 inode、目录）仍在磁盘上。
- 发出一个 `open("/foo/bar", O_RDONLY)` 调用时，文件系统首先需要找到文件 bar 的 inode，从而获取关于该文件的一些基本信息（权限信息、文件大小等等）
- 所有遍历都从文件系统的根开始，即根目录（root directory），它就记为/。因此，文件系统的第一次磁盘读取是根目录的 inode。
- 根的 inode 号必须是“众所周知的”。在挂载文件系统时，文件系统必须知道它是什么。在大多数 UNIX 文件系统中，根的 inode 号为 2。因此，要开始该过程，文件系统会读入 inode 号 2 的块（第一个 inode 块）。
- 一旦 inode 被读入，文件系统可以在其中查找指向数据块的指针，数据块包含根目录的内容。因此，文件系统将使用这些磁盘上的指针来读取目录，在这个例子中，寻找 foo 的条目。通过读入一个或多个目录数据块，它将找到 foo 的条目。一旦找到，文件系统也会找到下一个需要的 foo 的 inode 号（假定是 44）
- 递归遍历路径名，直到找到所需的 inode。在这个例子中，文件系统读取包含 foo 的 inode 及其目录数据的块，最后找到 bar 的 inode 号。
- `open()` 的最后一步是将 bar 的 inode 读入内存。然后文件系统进行最后的权限检查，在每个进程的打开文件表中，为此进程分配一个文件描述符，并将它返回给用户。
- 程序可以发出 `read()` 系统调用，从文件中读取。

	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			
read()					read			read		
read()					read					read
					write					

**File Read Timeline (Time Increasing Downward)**

# 访问路径：读和写

- 首先，文件必须打开（如上所述）。其次，应用程序可以发出 `write()` 调用以用新内容更新文件。
- 考虑简单和常见的操作（例如文件创建）。一个读取 `inode` 位图（查找空闲 `inode`），一个写入 `inode` 位图（将其标记为已分配），一个写入新的 `inode` 本身（初始化它），一个写入目录的数据（将文件的高级名称链接到它的 `inode` 号），以及一个读写目录 `inode` 以便更新它。如果目录需要增长以容纳新条目，则还需要额外的 I/O（即数据位图和新目录块）
- 创建了 `/foo/bar`，并且向它写入了 3 个块

	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	read	read write	read	read write							
			write											
			read											
			write											
write()	read write				read	write								
					write									
write()	read write				read	write								
					write									
write()	read write				read	write								
					write									

文件创建时间轴（时间向下递增）

# 缓存和缓冲

- 读取和写入文件的成本很高，会产生许多I/O。
  - 例如，长路径名（`/1/2/3/.../ 100/file.txt`）
    - 一个用于读取目录的inode，至少一个用于读取其数据。
    - 仅仅为了打开文件，就需要执行数百次读取。
- 为了减少I/O流量，文件系统积极地使用系统内存（DRAM）缓存。
  - 早期的文件系统因此引入了一个固定大小的缓存（fixed-size cache）来保存常用的块。这个固定大小的缓存通常会在启动时分配，大约占总内存的 10%。这种静态的内存划分（static partitioning）可能导致浪费。
  - 现代操作系统将虚拟内存页面和文件系统页面集成到统一页面缓存中。
- 读I/O可以通过大缓存来避免。

# 缓存和缓冲（续）

- 写操作必须写到磁盘才能持久化，因此，缓存不会减少写I/O，写操作采用写缓冲。
- 写缓冲（write buffering）的优点。
  - 通过延迟写入，文件系统可以将一些IO编成一批处理，从而节省IO。例如，如果在创建一个文件时，inode 位图被更新，稍后在创建另一个文件时又被更新，则文件系统会在第一次更新后延迟写入，从而节省一次 I/O。
- 通过将一些写入缓冲在内存中，系统可以调度（schedule）后续的 I/O，从而提高性能。
- 大多数现代文件系统将写入在内存中缓冲 5~30s，这代表了另一种折中：如果系统在更新传递到磁盘之前崩溃，更新就会丢失。
- 某些应用程序(如数据库)不喜欢这种折中。因此，为了避免由于写入缓冲导致的意外数据丢失，它们就强制写入磁盘，通过调用 `fsync()`，使用绕过缓存的直接 I/O(direct I/O)接口，或者使用原始磁盘(raw disk)接口并完全避免使用文件系统。

# 局部性和快速文件系统



- Ken Thompson 编写了第一个文件系统，它的数据结构在磁盘上看起来如下图所示。
- 超级块(S)包含有关整个文件系统的信息: 卷的大小、有多少 inode、指向空闲列表块的头部的指针等。
- 磁盘的 inode 区域包含文件系统的所有 inode。最后，大部分磁盘都被数据块占用



# 问题:性能不佳

- 性能很糟糕。随着时间的推移变得更糟，文件系统仅提供总磁盘带宽的 2%。
- 主要问题是将磁盘当成随机存取内存。数据遍布各处，没有考虑介质是磁盘。例如，文件的数据块通常离其 inode 非常远，导致昂贵的寻道成本。
- 更糟糕的是，文件系统最终会变得非常碎片化。空闲列表最终会指向遍布磁盘的一堆块。最后导致，逻辑上连续的文件，离散分布在磁盘的各个位置，磁头需要在磁盘上来回访问逻辑上连续的文件，大大降低了性能。

# Unix操作系统问题

- Unix文件系统将磁盘视为随机存取存储器。
- 具有四个文件的随机访问块的示例。
  - 每个文件的数据块可以通过在磁盘上来回访问来访问，因为它们  
是连续的。

A1	A2	B1	B2	C1	C2	D1	D2
----	----	----	----	----	----	----	----

- 文件b和d被删除。

A1	A2			C1	C2		
----	----	--	--	----	----	--	--

- 文件E是用空闲块创建的。（散布在整个街区）

A1	A2	E1	E2	C1	C2	E3	E4
----	----	----	----	----	----	----	----

- E 分散在磁盘上，因此，在访问 E 时，无法从磁盘获得峰值（顺序）性能。你  
首先读取 E1 和 E2，然后寻道，再读取 E3 和 E4。这个碎片问题一直发生在老  
UNIX 文件系统中，并且会影响性能

其他问题是原始块大小太小（512字节）

较小的块最大限度地减少了内部碎片（internal fragmentation，块内的浪费），但是由于  
每个块可能需要一个定位开销来访问它，因此传输不佳

### 关键问题：如何组织磁盘数据以提高性能

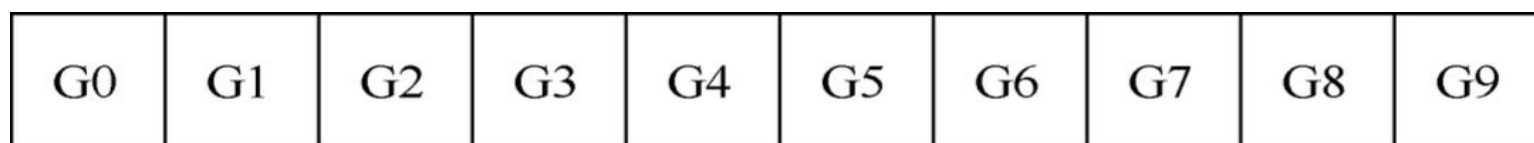
如何组织文件系统数据结构以提高性能？在这些数据结构之上，需要哪些类型的分配策略？如何让文件系统具有“磁盘意识”？

# FFS: 磁盘意识是解决方案

- 伯克利的一个小组决定建立一个更好、更快的文件系统，称之为快速文件系统 (Fast File System, FFS)。
- 思路是让文件系统的结构和分配策略具有“磁盘意识”，从而提高性能。

# 组织结构:柱面组

- FFS 将磁盘划分为一些分组，称为柱面组 (cylinder group, Linux ext2 和 ext3 称它们为块组，即 block group)。
- 这些分组是 FFS 用于改善性能的核心机制。通过在**同一组中放置两个文件，FFS 可以确保先后访问两个文件不会导致长时间寻道。**
- 每个柱面组中都有超级块 (super block) 的一个副本 (例如，如果一个被损坏或划伤，你仍然可以通过使用其中一个副本来挂载和访问文件系统)。
- 每个柱面组中，需要记录该组的 inode 和数据块是否已分配。每柱面组的 inode 位图 (inode bitmap, ib) 和数据位图 (data bitmap, db) 起到了这个作用，分别针对每组中的 inode 和数据块。
- 位图是管理文件系统中可用空间的绝佳方法，因为很容易找到大块可用空间并将其分配给文件，这可能会避免旧文件系统中空闲列表的某些碎片问题。最后，inode 和数据块区域就像之前的极简文件系统一样。像往常一样，每个柱面组的大部分都包含数据块。



FFS 在每个组中分配文件和目录。每个组看起来像下图，出于可靠性原因，每个组中都有超级块



# 策略:如何分配文件和目录

- 如何在磁盘上放置文件和目录以及相关的元数据, 以提高性能: **相关的东西放在一起** (无关的东西分开放)。
- 什么是“相关的”?
- 目录的放置: 找到分配数量少的柱面组(因为我们希望跨组平衡目录)和大量的空闲 inode(因为希望随后能够分配一堆文件), 将**目录数据和 inode 放在该分组中**。
- 对于文件: 首先, 它确保将**文件的数据块分配到与其 inode 相同的组中, 从而防止长时间寻道**;其次, 它将位于同一目录中的所有文件, 放在它们所在目录的柱面组中。
- 例如: 如果用户创建了 4 个文件, /dir1/1.tXt、/dir1/2.tXt、/dir1/3.tXt 和 /dir99/4.tXt, FFS 会尝试将前 3 个放在一起(同一组), 与第四个远离(它在另外某个组中)

E  
ND