File System Implementation

Liu yufeng

Fx_yfliu@163.com

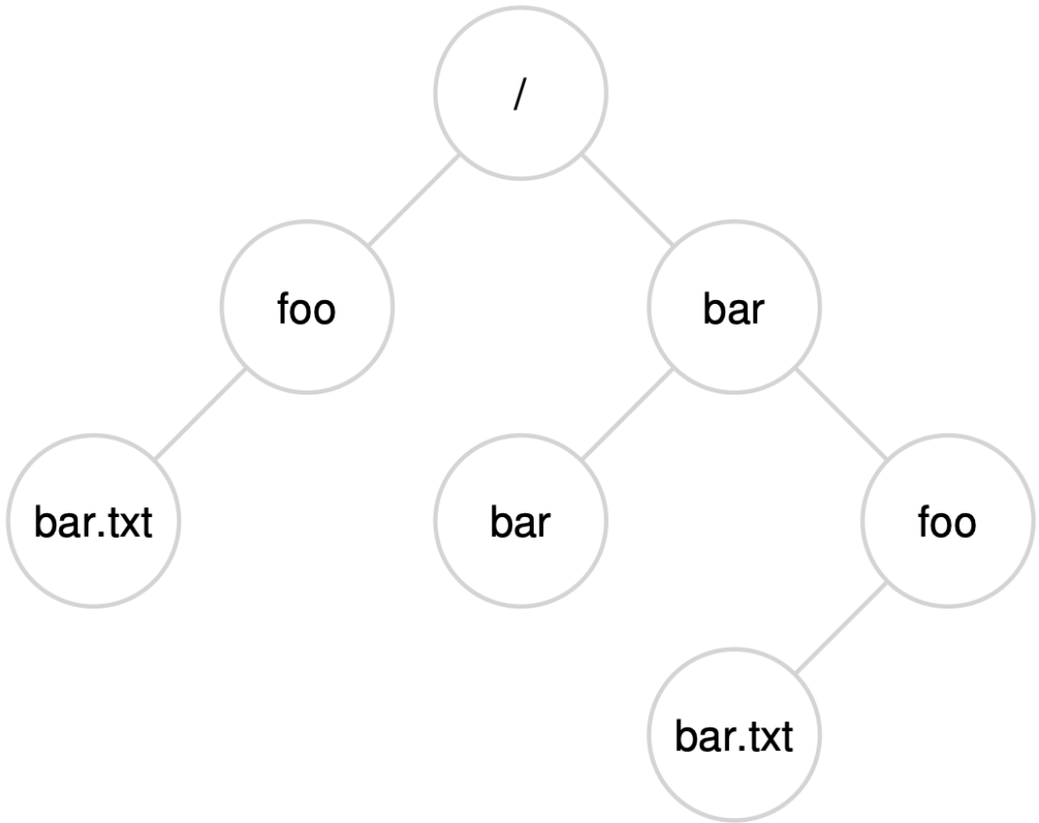
Hunan University

Persistent Storage

- Keep a data intact even if there is a power loss.
  - Hard disk drive
  - Solid-state storage device

- Two key abstractions in the virtualization of storage
  - 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
    - Each file has low-level name as inode number
      - The user is not aware of this name.
  - 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
  - A directory has an entry（“foo”，“10”）
    - A file “foo” with the low-level name “10”

Files and Directories



Valid files (absolute pathname) :
/foo/bar.txt
/bar/foo/bar.txt

Valid directory :
/
/foo
/bar
/bar/bar
/bar/foo/

The File System Interface

Creating Files

```
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()` system call returns file descriptor.

File descriptor is an integer, and is used to access files.

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()。

Reading and Writing Files

```
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。

How does the `cat` program access the file foo ?

We can use `strace` to trace the system calls made by a program.

Reading and Writing Files

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 cat foo
…
open(“foo”, O_RDONLY|O_LARGEFILE) = 3
read(3, “hello\n”, 4096)      = 6 // read()的调用返回读取的字节数
write(1, “hello\n”, 6)        = 6 // file descriptor 1: standard out
hello
read(3, “”, 4096)             = 0 //文件中没有剩余字节，read()返回 0
close(3)                      = 0
…
prompt>
```

`open` (file descriptor, flags)

Return file descriptor (3 in example)

File descriptor 0, 1, 2, is for standard input/ output/ error.

`read` (file descriptor, buffer pointer, the size of the buffer)

Return the number of bytes it read

`write` (file descriptor, buffer pointer, the size of the buffer)

Return the number of bytes it write

Reading And Writing, But Not Sequentially

```
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

Writing Immediately with `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。

- off_t fsync(int fd)
  - Filesystem forces all dirty (i.e., not yet written) data to disk for the file referred to by the file description.
  - fsync() returns once all of theses writes are complete.

Writing Immediately with `fsync()`

- An Example of `fsync()`.

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
assert (fd > -1)
int rc = write(fd, buffer, size);
assert (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。

Renaming Files

- `rename(char* old, char *new)`
  - Rename a file to different name.
  - 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) Change from `foo` to `bar`:

```
prompt> mv foo bar // mv uses the system call 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 fint fd = open("foo.txt.tmp",O_WRONLY|O_CREAT|O_TRUNC);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

将文件的新版本写入临时名称（foot.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

Getting Information About Files

- `stat(), fstat()`: Show the file metadata
  - **Metadata** is information about each file.
  - Ex) Size, Low-level name, Permission, …
  - `stat` structure is below:

```
struct stat {
    dev_t st_dev;     /* ID of device containing file */
    ino_t st_ino;     /* inode number */
    mode_t st_mode;   /* protection */
    nlink_t st_nlink;    /* number of hard links */
    uid_t st_uid;     /* user ID of owner */
    gid_t st_gid;     /* group ID of owner */
    dev_t st_rdev;    /* device ID (if special file) */
    off_t st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t st_blocks;  /* number of blocks allocated */
    time_t st_atime;     /* time of last access */
    time_t st_mtime;     /* time of last modification */
    time_t st_ctime;     /* time of last status change */
};
```

Getting Information About Files (Cont.)

- To see stat information, you can use the command line tool `stat`.

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

File: ‘file’
Size: 6 Blocks: 8 IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084 Links: 1
Access: (0640/-rw-r-----) Uid: (30686/ root) Gid: (30686/ remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

- File system keeps this type of information in a `inode` structure.

Removing Files

- `rm` is Linux command to remove a file
  - `rm` call `unlink()` to remove a file.

```
prompt> strace rm foo
…
unlink("foo")        = 0 // return 0 upon success
…
prompt>
```

Why it calls `unlink()`? not "`remove` or `delete`"
We can get the answer later.

Making Directories

- `mkdir()`: Make a directory

```
prompt> strace mkdir foo
…
mkdir("foo", 0777)      = 0
prompt>
```

- When a directory is created, it is empty.
- Empty directory have two entries: . (itself), .. (parent)

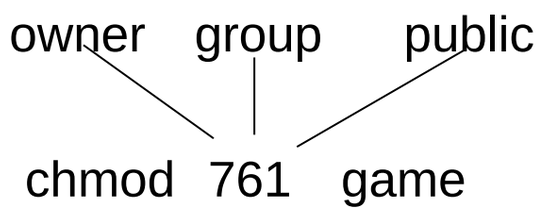
```
prompt> ls -a
./  ../
prompt> ls -al
total 8
drwxr-x---  2 remzi remzi    6 Apr 30 16:17 ./
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

访问（控制）列表和组

- Mode of access: read, write, execute
- Three classes of users

			RWX
a) owner access	7	⇒	1 1 1
			RWX
b) group access	6	⇒	1 1 0
			RWX
c) public access	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

chgrp G game

Reading Directories

- 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

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");          // open current directory
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL)    // read one directory entry
    {
        // print outthe name and inode number of each file
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp);                // close current directory
    return 0;
}
```

- The information available within struct dirent

```
struct dirent {
    char      d_name[256];  /* filename */
    ino_t          d_ino;        /* inode number */
    off_t          d_off;        /* offset to the next direct */
    unsigned short     d_reclen;         /* length of this record */
    unsigned char d_type;       /* type of file */
}
```

Deleting Directories

- `rmdir()`: Delete a directory.
  - Require that the directory be **empty**.
    - If you call `rmdir()` to a non-empty directory, it will fail.
      - I.e., Only has “.” and “..” entries.

Hard Links

- `link(old pathname, new one)`
  - **Link** a new file name to an old one
  - Create another way to refer to *the same file*

- The command-line link program : `ln`

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2  // create a hard link, link file to file2
prompt> cat file2
hello
```

Hard Links (Cont.)

- The way `link` works:
  - **Create** another name in the directory.
  - **Refer** it to the same inode number of the original file.
    - The file is not copied in any way.
- Then, we now just have two human names (`file` and `file2`) that both refer to the same file.

Hard Links (Cont.)

- The result of `link()`

```
prompt> ls -i file file2
67158084 file  /* inode value is 67158084 */
67158084 file2 /* inode value is 67158084 */
prompt>
```

- Two files have **same inode** number, but two human name (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。

Hard Links (Cont.)

```
prompt> rm file
removed 'file'
prompt> cat file2        // Still access the file
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。

Hard Links (Cont.)

- The result of `unlink()`
- `stat()` shows the reference count of a file.

```
prompt> echo hello > file              /* create file*/
prompt> stat file
... Inode: 67158084 Links: 1 ...       /* Link count is 1 */
prompt> ln file file2                  /* hard link file2 */
prompt> stat file
... Inode: 67158084 Links: 2 ...       /* Link count is 2 */
prompt> stat file2
... Inode: 67158084 Links: 2 ...       /* Link count is 2 */
prompt> ln file2 file3                 /* hard link file3 */
prompt> stat file
... Inode: 67158084 Links: 3 ...       /* Link count is 3 */
prompt> rm file                        /* remove file */
prompt> stat file2
... Inode: 67158084 Links: 2 ...       /* Link count is 2 */
prompt> rm file2                       /* remove file2 */
prompt> stat file3
... Inode: 67158084 Links: 1 ...       /* Link count is 1 */
prompt> rm file3
```

Symbolic Links (Soft Link)

- Symbolic link is more useful than Hard link.
  - Hard Link cannot create to a directory（防止出现环）.
  - Hard Link cannot create to a file to other partition.
    - Because inode numbers are only unique within a file system.

```
prompt> echo hello > file
prompt> ln -s file file2  /* option -s : create a symbolic link, */
prompt> cat file2
hello
```

- Create a symbolic link: `ln -s`

Symbolic Links (Cont.)

- What is different between Symbolic link and Hard Link?
  - 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 file
 ... regular file ...
prompt> stat file2
 ... symbolic link ...     // Actually a file it self of a different type
```

- The size of symbolic link (`file2`) is **4 bytes**.

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May 3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May 3 15:14 ../              // directory
-rw-r-----  1 remzi remzi    6 May 3 19:10 file             // regular file
lrwxrwxrwx  1 remzi remzi    4 May 3 19:10 file2 -> file    // symbolic link
```

  - A symbolic link holds the pathname of the linked-to file as the data of the link file.

Symbolic Links (Cont.)

- If we link to a longer pathname, our link file would be bigger.

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

Symbolic Links (Cont.)

- Dangling reference
  - When remove a original file, symbolic link points noting.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file     // remove the original file
prompt> cat file2
cat: file2: No such file or directory
```

Making and Mounting a File System

- `mkfs` tool : Make a file system
  - 在一个磁盘分区上写入一个空文件 系统.
  - Input:
    - A device (such as a disk partition, e.g., `/dev/sda1`)
    - A file system type (e.g., `ext3`)

Making and Mounting a File System (Cont.)

- 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
- **Example)**

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

  - The pathname /home/users/ now refers to the root of the newly-mounted directory.

Making and Mounting a File System (Cont.)

- `mount` program: show **what is mounted** on a system.

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

- `ext3`: A standard disk-based file system
- `proc`: A file system for accessing information about current processes
- `tmpfs`: A file system just for temporary files
- `AFS`: A distributed file system

Implementing File System

- 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（但我们需要注意设备特性，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?

The Way To Think

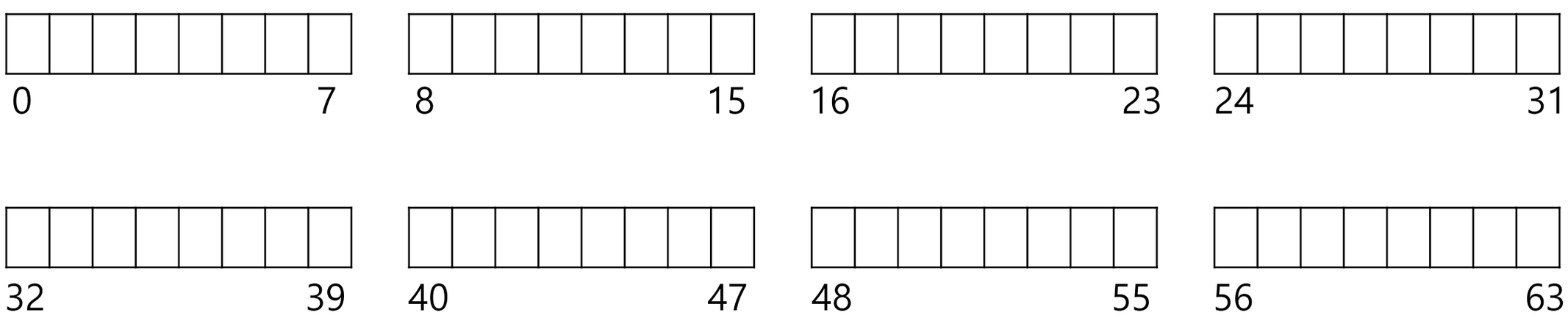
- 第一个方面是文件系统的数据结构（data structure）。换言之，文件系统在磁盘上使用哪 些类型的结构来组织其数据和元数据?

- 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?

Overall Organization

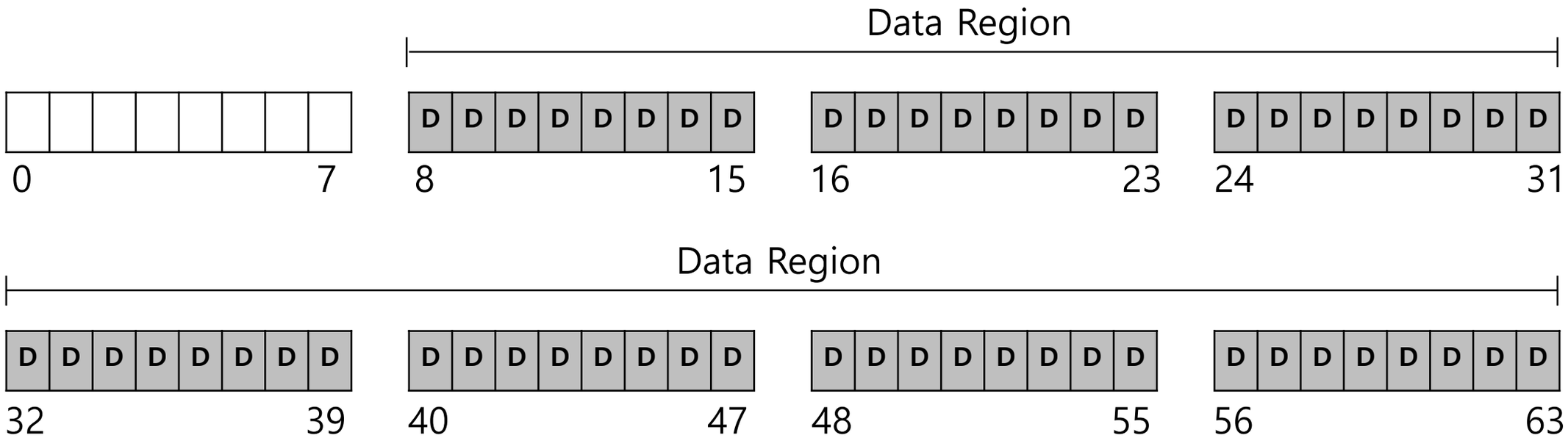
- VSFS 文件系统在磁盘上的数据结构的整体组织。
- 第一件 事是将磁盘分成块（block）。 选择常用的 4KB。

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



Data region in file system

- 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



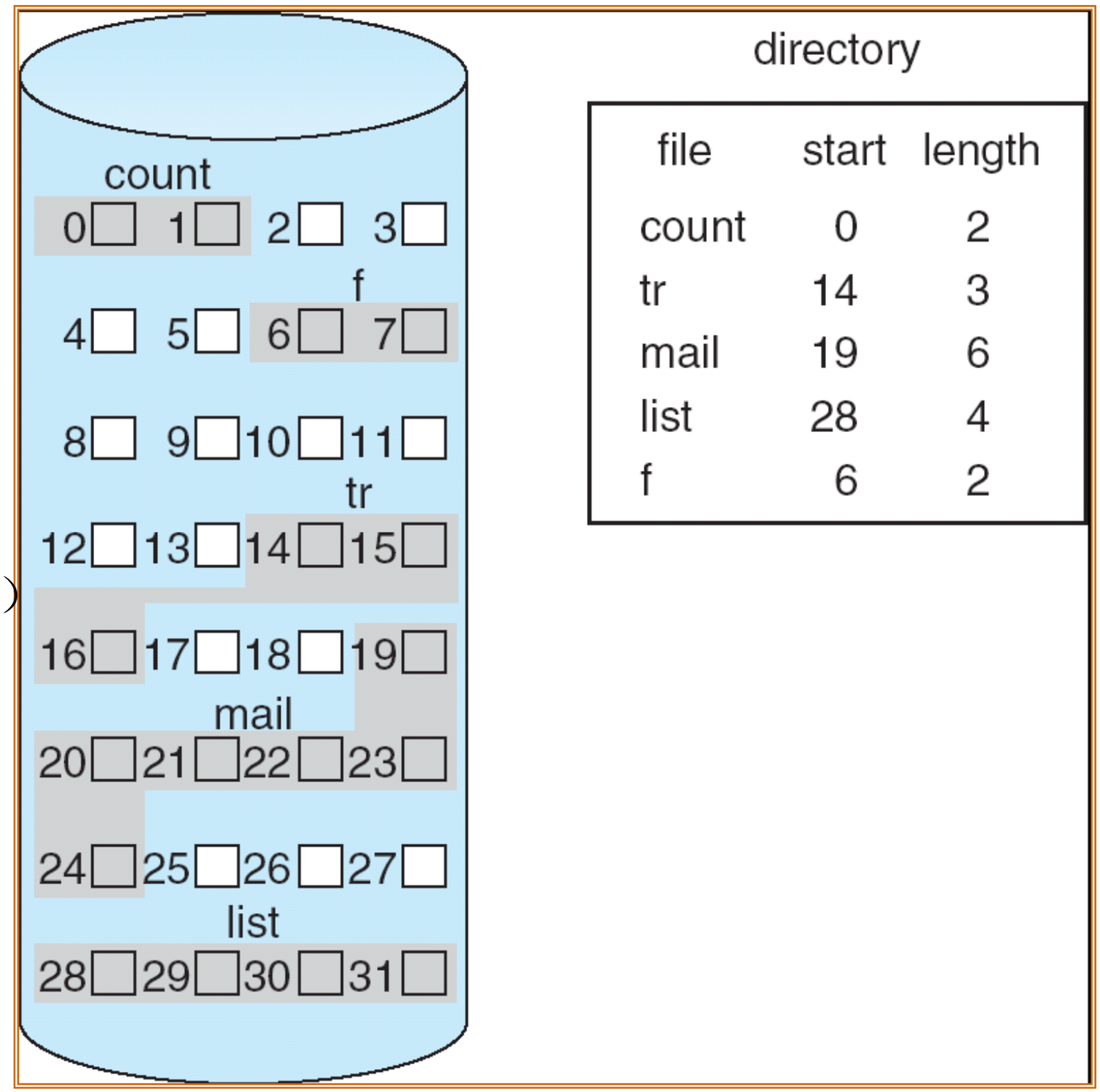
How we store these inodes in file system?

文件物理结构之连续分配 (contiguous allocation)

- 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

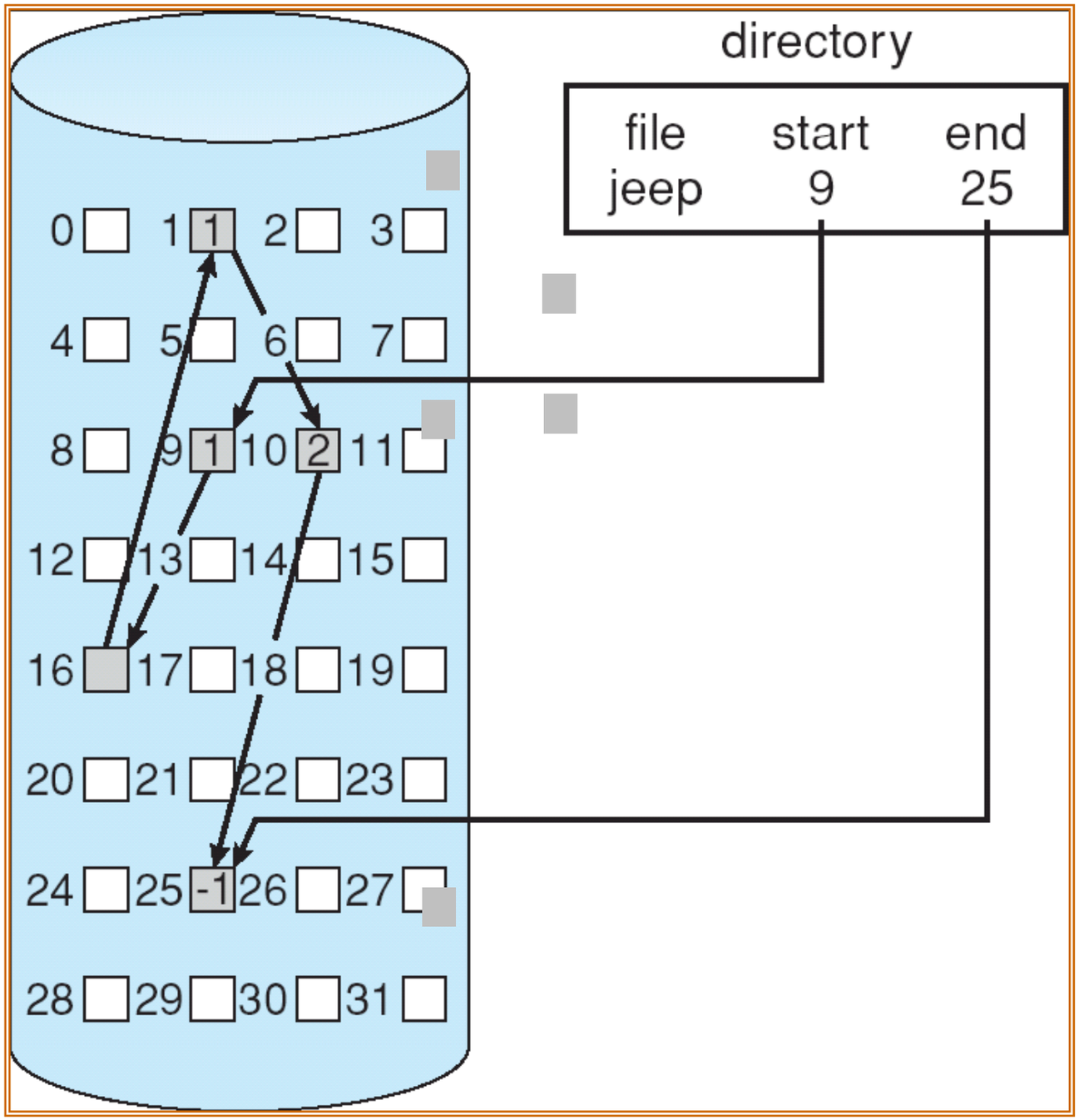
- 许多新的文件系统采用一种修正的连续分配方法
- 该方法开始分配一块连续空间，当空间不够时，另一块被称为扩展的连续空间会添加到原来的分配中。
- 文件块的位置就成为开始地址、块数、加上一个指向下一扩展的指针。

文件物理结构(二) 链接分配 (linked allocation)

- 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

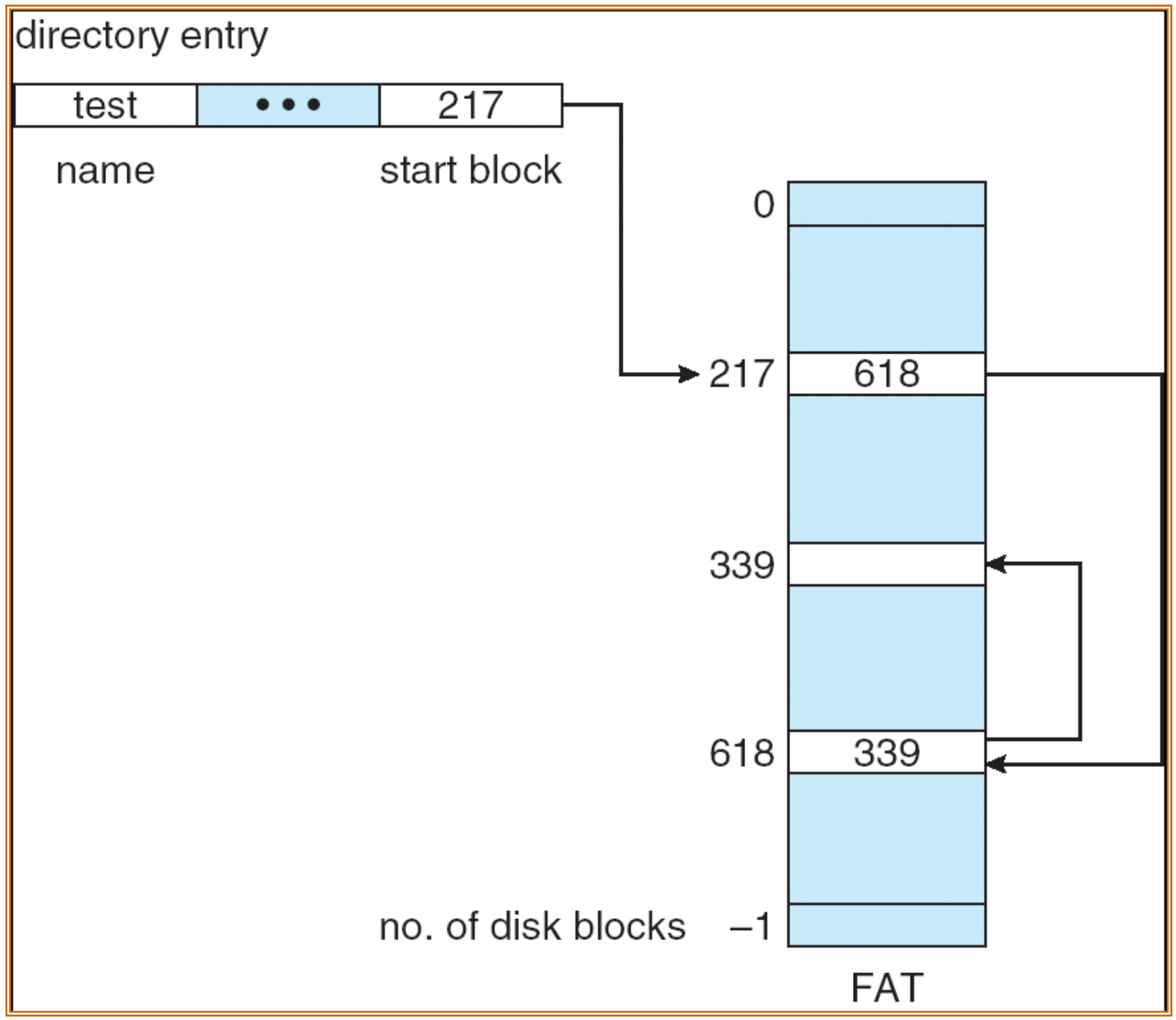


链接分配方法的变种-文件分配表(FAT)

- 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？

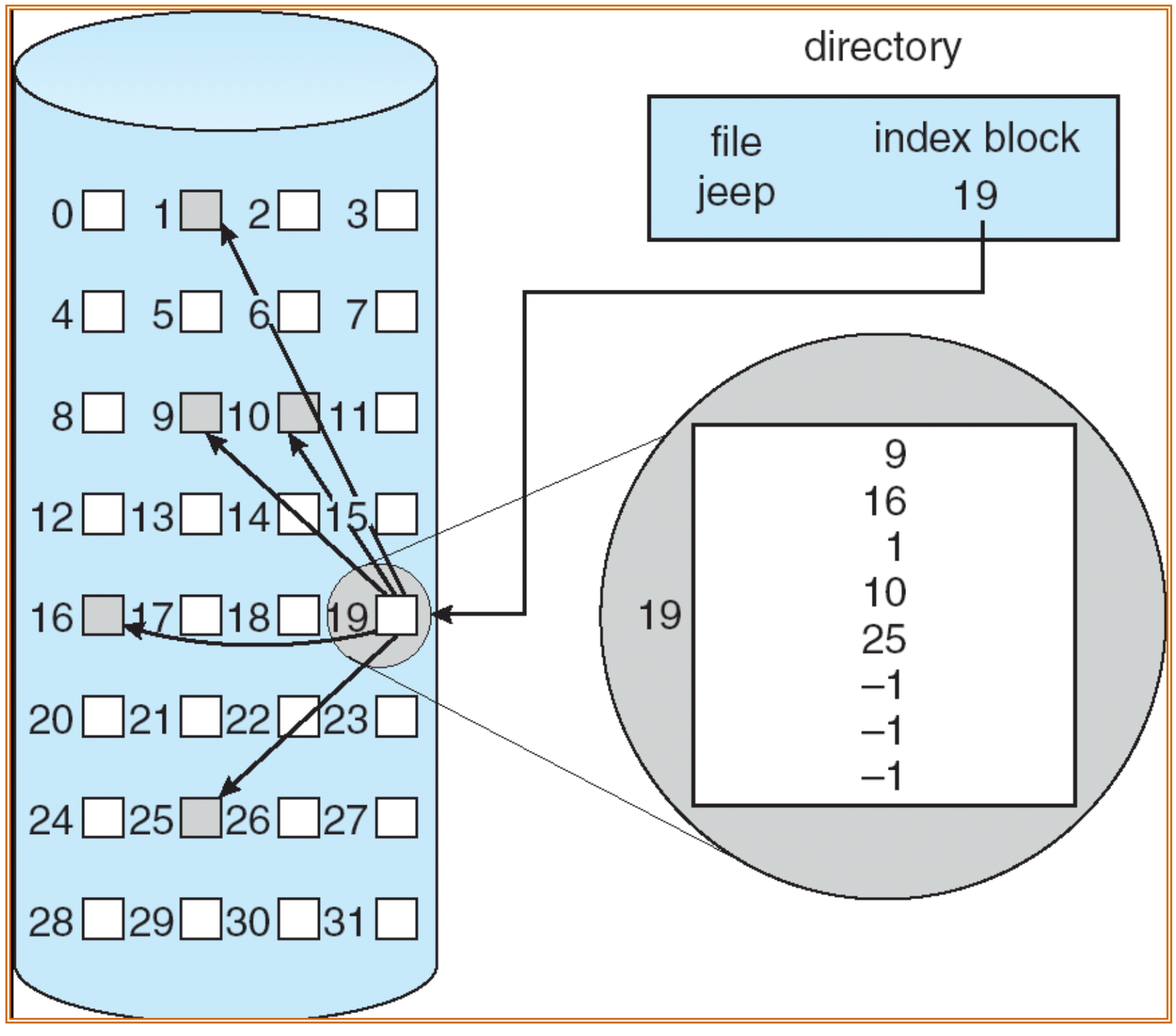
答:有512MB/512B = 1M个盘块，64K个指针不够

文件物理结构(三) 索引分配(indexed allocation)

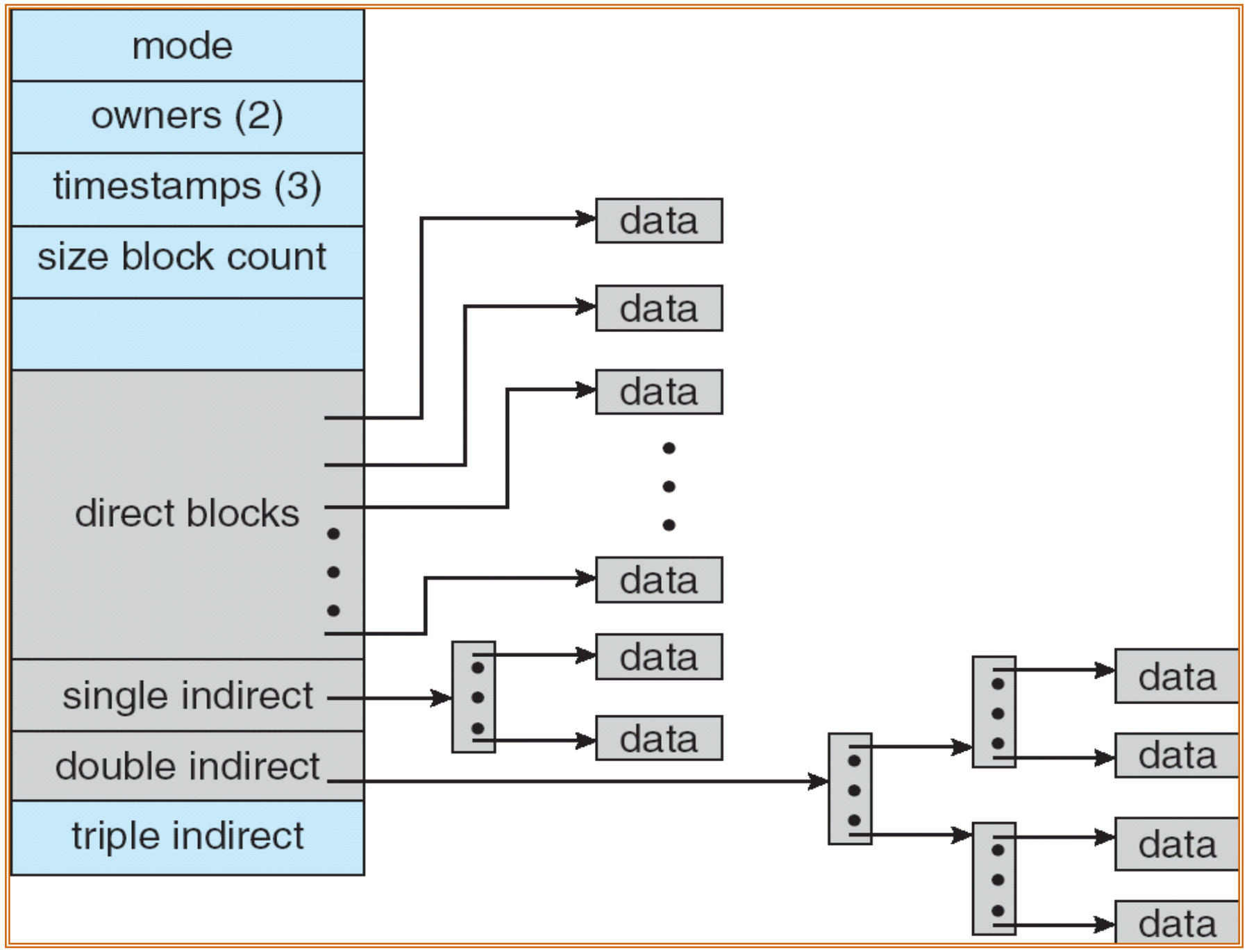
- 将所有的数据块指针集中到索引块中
  - 索引块中的第i个条目指向文件的第i块。
  - 目录条目包括索引块的地址
- 索引分配支持直接访问，且没有外部碎片问题
- 索引块本身可能会浪费空间
  - 链接方案：一个索引块通常为一个磁盘块。对于大文件，可以将多个索引块链接起来。
  - 多层索引：类似于内存的间接寻址方式（一级、二级间接…）
  - 组合方案：如Unix的inode

索引分配

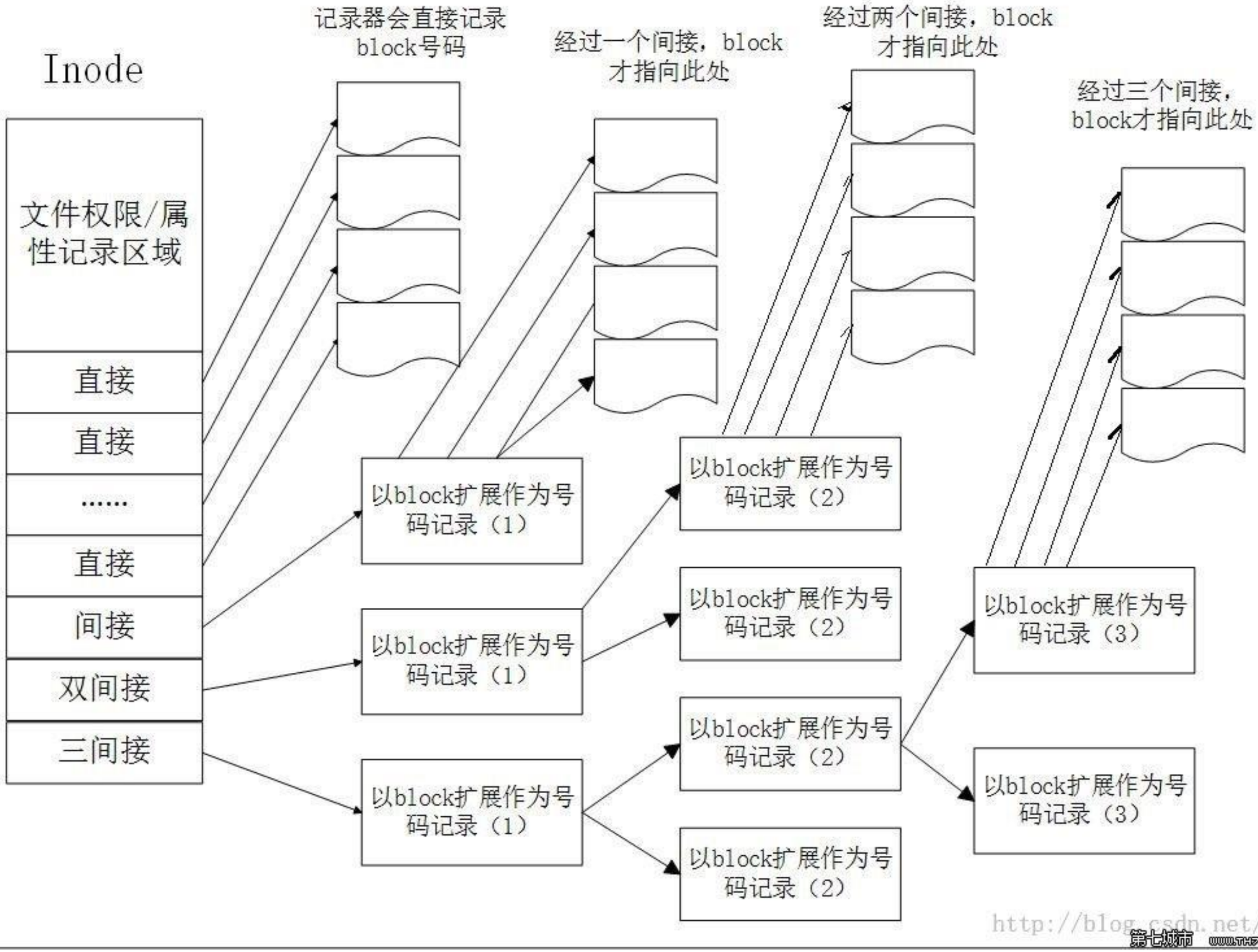
1、需要索引表
2、支持直接访问
3、无外部碎片
4、索引有开销



UNIX inode (4K bytes per block)



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

由于索引结点为128字节，指针为4个字节，而状态信息占68个字节，且每块大小为8kB

```
用于指针的空间大小为：
索引节点字节数-状态信息字节数=60字节
```

一次间接指针、二次间接指针和三次间接指针将占用索引节点中的三个指针项，
因此直接指针项数为： 60/4-3=12（个）

“且每块大小为8kB”
这里题目让我这钻牛角尖的人思考太多，其实说的就是指针指向的盘块大小为8KB，
8kB=8*1024B = 8192 B
(网上有一个博主写了这个同一个题的，但是没有解释作用且写错答案，具体就不说了)
奔主题：

```
直接指针有12个，一个指向的盘块的字节数是8192B，
所以 使用直接指针可以表示的文件大小是
12 * 8192B =98304 B
所以大小不超过98304字节的文件使用直接指针即可表示。
```

复制

```
一次间接指针指向的磁盘块大小是8192B，
全是指针，所以一块磁盘块内的指针的指针项数量为8192B/4B=2048(个)
2048个指针指向的总磁盘大小是2048*8192B =16777216B = 16384KB=16MB
一次间接指针 表示的文件大小为16MB
一次间接指针提供了对附加16M字节信息的寻址能力。
```

```
一次间接指针指向文件存放的总磁盘大小是2048*8192B =16777216B = 16384KB=16MB
二次间接指针 可以提供的指针项为 16MB /4B  =4M个
4M个指针指向的总磁盘大小是4M*8192B = 32GB
二次间接指针 表示的文件大小为 4M*8192B = 32GB
二次间接指针提供了对附加32G字节信息的寻址能力。
```

```
二次间接指针 指向文件存放的总磁盘大小是 4M*8192B = 32GB
三次间接指针 可以提供的指针项为 32GB/4B=8G个
8G个指针指向的总磁盘大小是8G*8192B =  65536GB = 64TB
三次间接指针 表示的文件大小为8G*8192B =  65536GB = 64TB
三次间接指针提供了对附加64T字节信息的寻址能力。
```

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

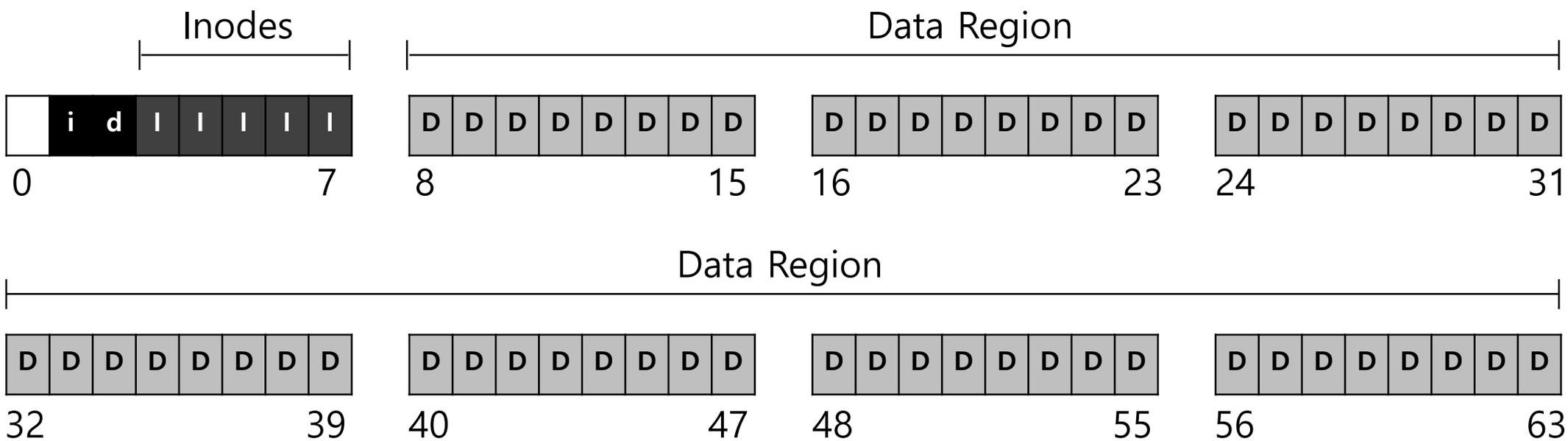
一次间接指针:（8K/4）*8K=16MB

二次间接指针:2K*2K*8K=32G

三次间接: 2K*2K*2K*8K=16TB

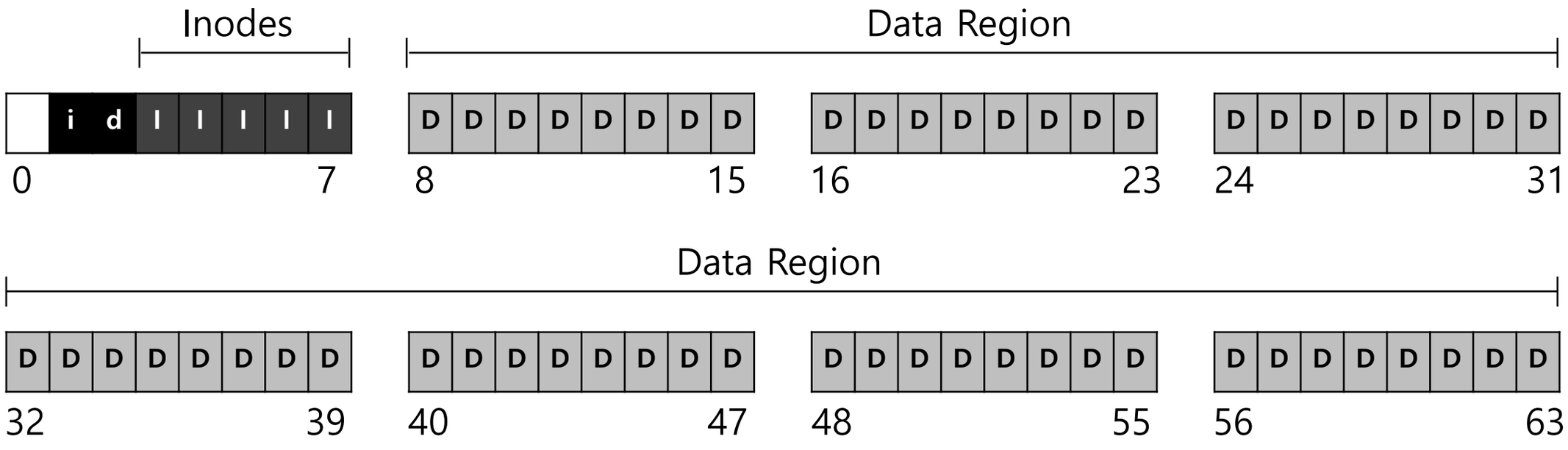
Inode table in file system

- 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）。



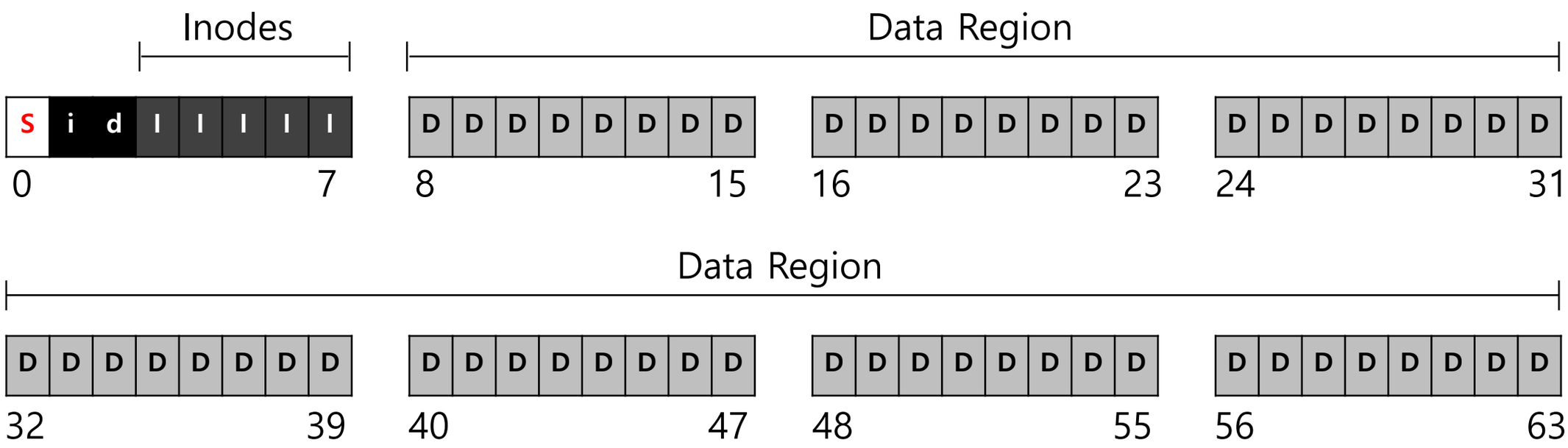
Inode table in file system

- 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。



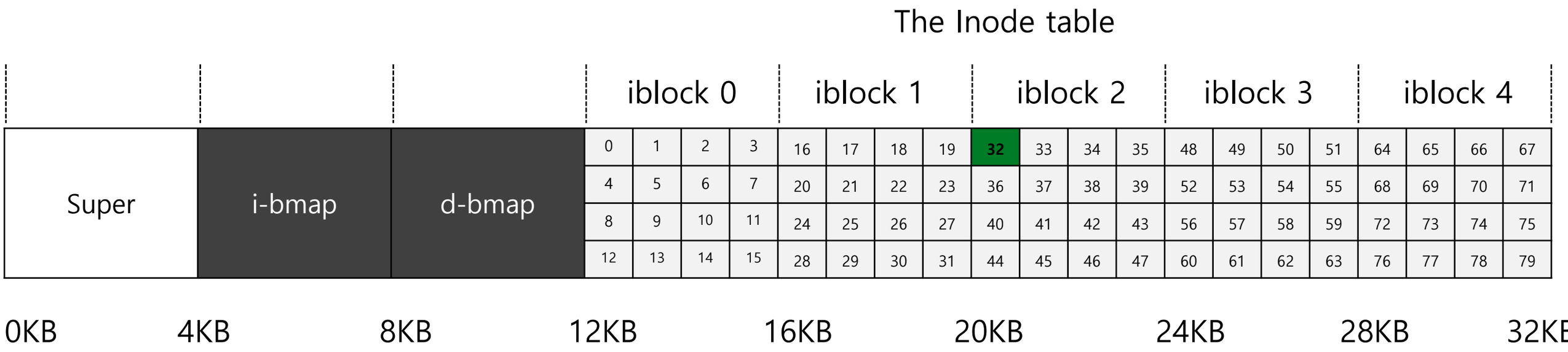
Inode table in file system

- 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）



File Organization: The 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×inode 的大小，即 8192）
  - 它加上磁盘 inode 表的起始地址（inodeStartAddr = 12KB），从而得到希望的 inode 块的正确字节地址：20KB



File Organization: The 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 = (inumber * sizeof(inode_t)) / blockSize;
  - sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;

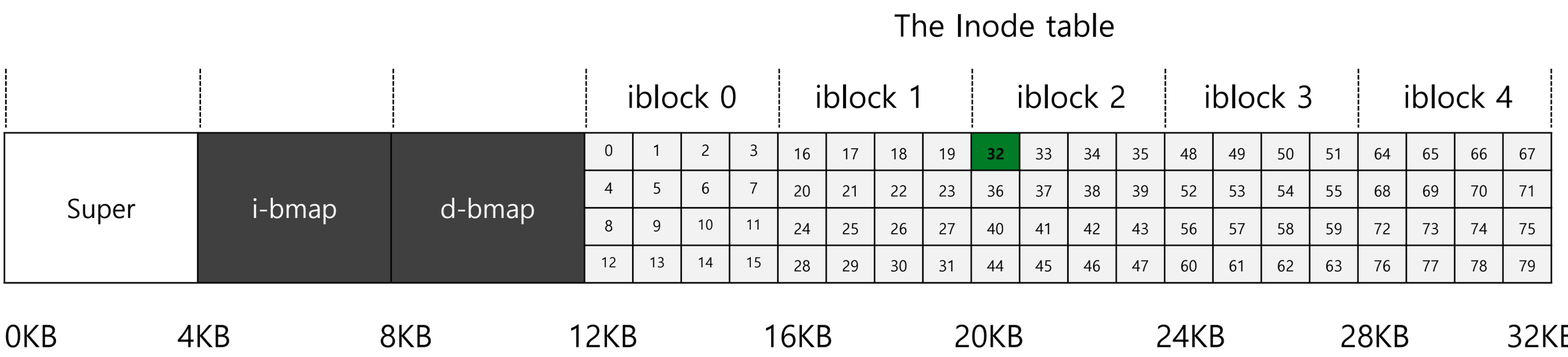


表 40.1　　ext2 的 inode

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

The Multi-Level Index

How to support bigger files? 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*512B/1024 = 5KB
一个盘块中可放的盘块地址数为：512B/4B = 128
一次间接索引存放的容量为：128*512B/1024 = 64KB
二次间接索引存放的容量为：128*128*512B/1024 = 8192KB
三次间接索引存放的容量为：128*128*128*512B/1024 = 1048576KB

1MB为1024KB，1024KB-64KB-5KB=955KB，需要二次间接盘块为
955×1024B/512B=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

25MB为：25×1024KB--64KB-5KB-8192KB=17339KB,
17339×1024B/512B=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

The Multi-Level Index?

- 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 ） × 4KB。
- 增加一个三重间接块，你是否能弄清楚支持多大的文件?

Directory Organization

- Directory contains a list of (entry name, inode number) pairs.
- Each directory has two extra files ."dot" for current directory and .."dot-dot" for parent directory
  - For example, `dir` has three files (`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

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

on-disk for dir

Free Space Management

- 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。

- Thus **free space management** is important for all file systems. We have two simple **bitmaps** for this task.

空闲空间管理

- 位向量（n块）
  - bit[i] = 0 ➔ block[i]空闲
  - bit[i] = 1 ➔ 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 DIV n

j=b MOD n

Access Paths: Reading and Writing

- 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**					
					write			**read**		
read()					**read**					
					write				**read**	
read()					**read**					
					write					**read**

File Read Timeline (Time Increasing Downward)

Access Paths: Reading and Writing

- 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]
			read							
						read				
				read						
							read			
create (/foo/bar)		**read write**								
							write			
					read write					
				write						
write()	**read write**				**read**					
								write		
					write					
write()	**read write**				**read**					
									write	
					write					
write()	**read write**				**read**					
										write
					write					

File Creation Timeline (Time Increasing Downward)

Caching and Buffering

- Reading and writing files are expensive, incurring many I/Os.
  - For example, long pathname(/1/2/3/…./100/file.txt)
    - One to read the inode of the directory and at least one read its data.
    - Literally perform hundreds of reads just to open the file.

- In order to reduce I/O traffic, file systems aggressively use system memory(DRAM) to cache.
  - 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.
- Read I/O can be avoided by large cache.

Caching and Buffering (Cont.)

- 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，离散的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。

Problem of Unix operating system

- Unix file system treated the disk as a **random-access memory.**
  - Example of random-access blocks with Four files.
    - Data blocks for each file can accessed by going back and forth the disk, because they are are **contiguous.**

A1	A2	B1	B2	C1	C2	D1	D2

    - File b and d is deleted.

A1	A2			C1	C2		

    - File E is created with free blocks. (**spread across** the block)

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

Other Problem is the original block size was too small(512 bytes)
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”？

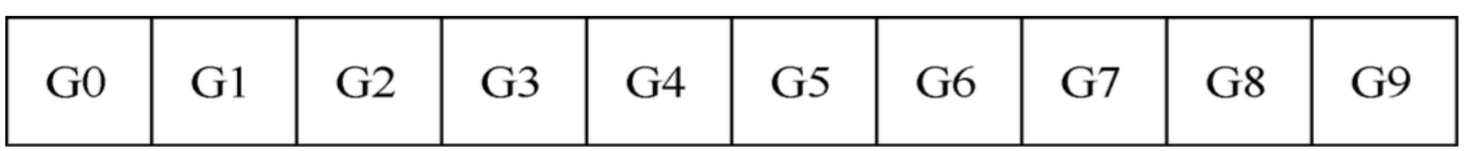
FFS:磁盘意识是解决方案

- 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）

End