

深圳大学 计算机与软件学院

College of Computer Science and Software Engineering of Shenzhen University



# 系统编程

## 基于TaiShan服务器/openEuler OS 的实践

### 第四讲：文件控制 – 文件I/O函数

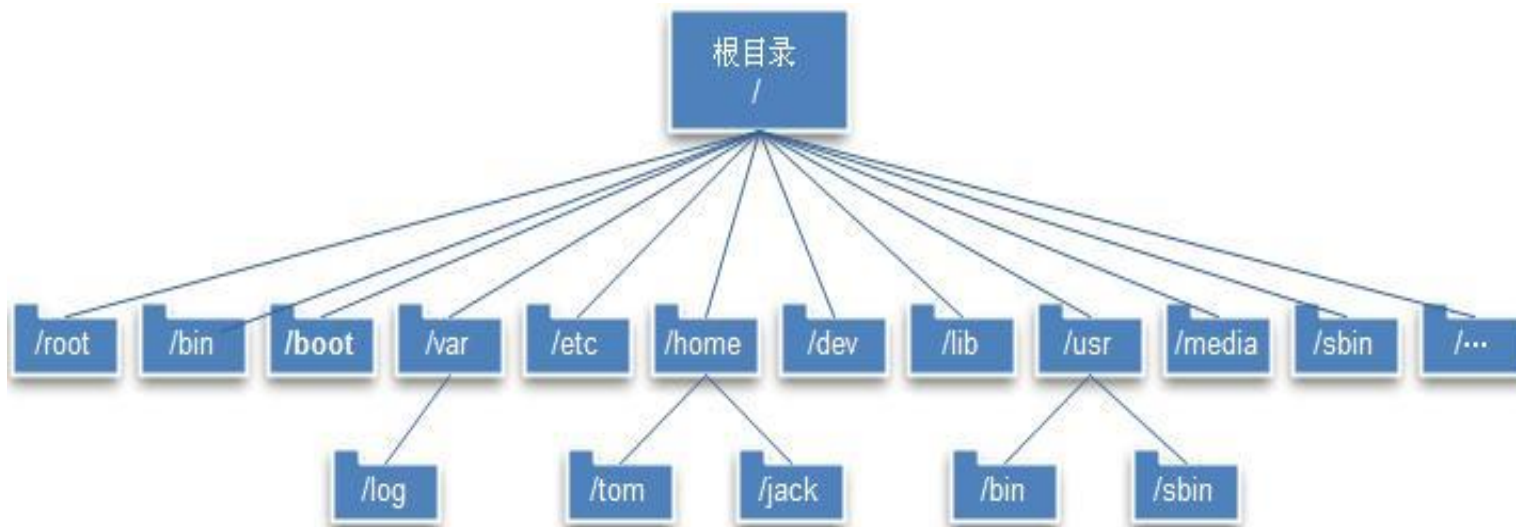
# 文件系统

## ■ 文件系统提供对磁盘的高层应用访问

- 屏蔽底层基于扇区的I/O操作细节
- 提供对数据（文件或目录）的结构化访问
- 在内存建立对最近访问数据的高速缓存

## ■ 层次化文件系统：最常用的类型

- 以树结构组织目录和文件



# 文件系统

## ■ 文件系统提供对磁盘的高层应用访问

- 屏蔽底层基于扇区的I/O操作细节
- 提供对数据（文件或目录）的结构化访问
- 在内存建立对最近访问数据的高速缓存

## ■ 层次化文件系统：最常用的类型

- 以树结构组织目录和文件

## ■ 面向字节的文件 vs. 面向记录的文件

- UNIX、Windows等提供面向字节的文件访问
  - ◆ 可一次读/写文件的一个字节，简单、灵活
- 很多老一点的操作系统仅提供面向记录的文件访问

# 文件系统操作

## ■ 文件系统提供对文件或目录的标准接口

- 创建/删除/打开/关闭一个文件/目录
- 读/写/追加文件内容
- 增加或删除目录条目

## ■ 文件系统还可以提供什么特别的服务

- 会记和限额：防止某些用户霸占磁盘
- 备份：一些文件系统的  
“\$HOME/. backup” 包含自动快照
- 索引和查找能力
- 文件版本控制
- 加密
- 不常用文件的自动压缩

## 这些功能

■ 文件系统的一部分？

■ 架构在文件系统之上的上层软件的一部分？

经典OS社区争论

■ 哪才是最好归宿？

# 日志文件管理与备份

## 1. rsyslog.conf

如： 将日志信息级别大于或等于info级别的信息记录到/var/log/messages

```
# Log cron stuff
cron.*                                /var/log/cron

# Everybody gets emergency messages
*.emerg                               *

# Save news errors of level crit and higher in a special file.
uucp,news.crit                        /var/log/spooler

# Save boot messages also to boot.log
local7.*                              /var/log/boot.log
*.info                                /var/log/messages
```

## 2. 日志信息的备份与转储logrotate: message.conf

如： 日志权限： 664, 拥有者： root, 群组： utmp; 每周转储一次， 最大保留多少份： 4

```
[root@fedora17 httpd]# vi /etc/logrotate.d/message.conf

/var/log/messages(
    weekly
    create 0664 root utmp
    rotate 4
)
```

# 基本的文件系统结构

## ■ 文件和目录都由其inode 表示

- “index node”，索引节点

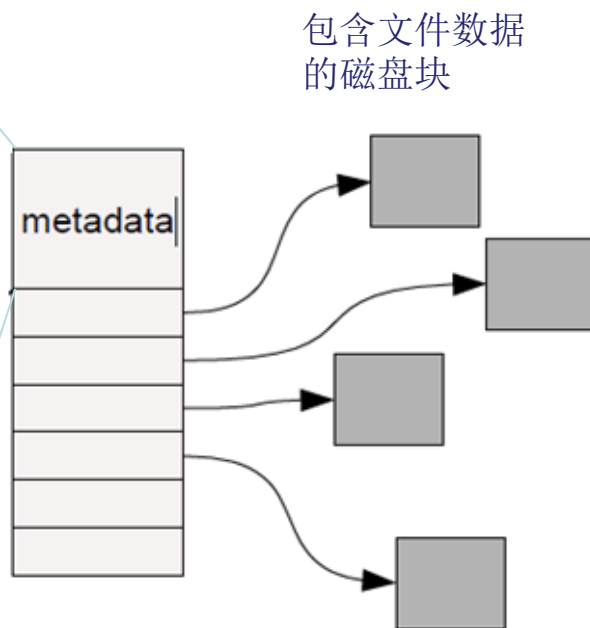
## ■ inode包含两类信息

- Metadata: 元数据，描述文件的所有者、访问权限等
- 文件块在磁盘的位置

这里缺了啥？

- 文件大小
- 文件所有者 **UID**
- 文件组 **GID**
- 权限
- 创建时间
- 修改时间
- 访问时间
- 连接数

....



```
[yuhong@FedoraDVD13 semaphore]$ ls
```

```
sem sem.c
```

```
[yuhong@FedoraDVD13 semaphore]$ stat sem.c
```

```
File: "sem.c"
```

```
Size: 788          Blocks: 8          IO Block: 4096   普通文件
```

```
Device: fd00h/64768d    Inode: 153035    Links: 1
```

```
Access: (0664/-rw-rw-r--)  Uid: ( 500/  yuhong)   Gid: ( 500/  yuhong)
```

```
Access: 2017-11-19 22:03:09.597016818 +0800
```

```
Modify: 2017-11-19 22:03:07.940935139 +0800
```

```
Change: 2017-11-19 22:03:07.951920141 +0800
```

# 基本的文件系统结构

- 文件和目录都由其inode 表示
- inode包含两类信息
- inode的大小

```
[yuhong@FedoraDVD13 semaphore]$ ls -i sem.c
153035 sem.c
[yuhong@FedoraDVD13 semaphore]$
```

- 文件大小
- 文件所有者 **UID**
- 文件组 **GID**
- 权限
- 创建时间
- 修改时间
- 访问时间
- 连接数

... ..



包含文件数据的  
磁盘块

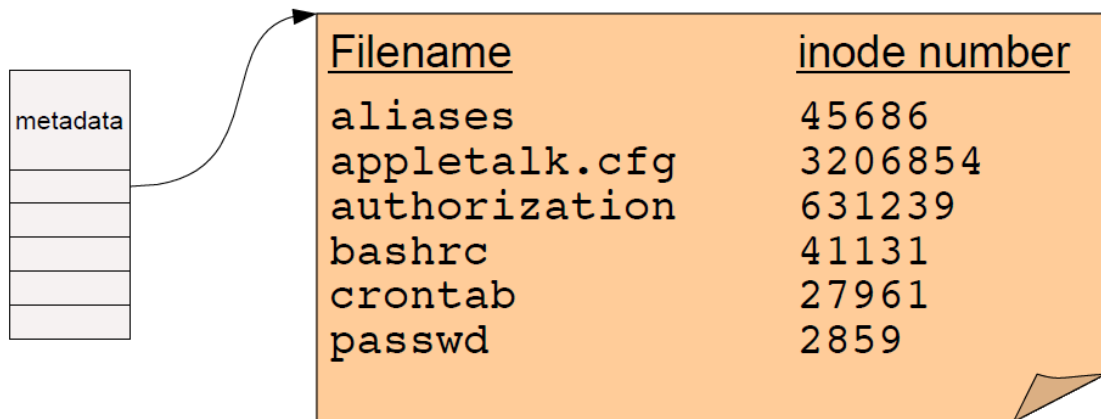
```
[yuhong@FedoraDVD13 semaphore]$ df -i
文件系统                Inode  已用 (I)  可用 (I)  已用 (I)%%  挂载点
/dev/mapper/vg_fedoradvd13-lv_root
425984 175371 250613    42% /
tmpfs                63718      6  63712     1% /dev/shm
/dev/sda1            128016     36 127980     1% /boot
/dev/sr0              0         0      0      - /media/Fedora 13 i386 DVD
```

```
[yuhong@FedoraDVD13 semaphore]$ df -h
文件系统                容量  已用  可用  已用%%  挂载点
/dev/mapper/vg_fedoradvd13-lv_root
6.4G  5.1G 1012M  84% /
tmpfs                249M  420K  249M   1% /dev/shm
/dev/sda1            485M   28M  433M   6% /boot
/dev/sr0             3.1G   3.1G    0 100% /media/Fedora 13 i386 DVD
```

```
[root@FedoraDVD13 semaphore]# dumpe2fs -h /dev/sda1 | grep "Inode size"
dumpe2fs 1.41.10 (10-Feb-2009)
Inode size:                128
```

# 目录

- 目录是一种特殊的文件，内容是系列（文件名，inode号）对

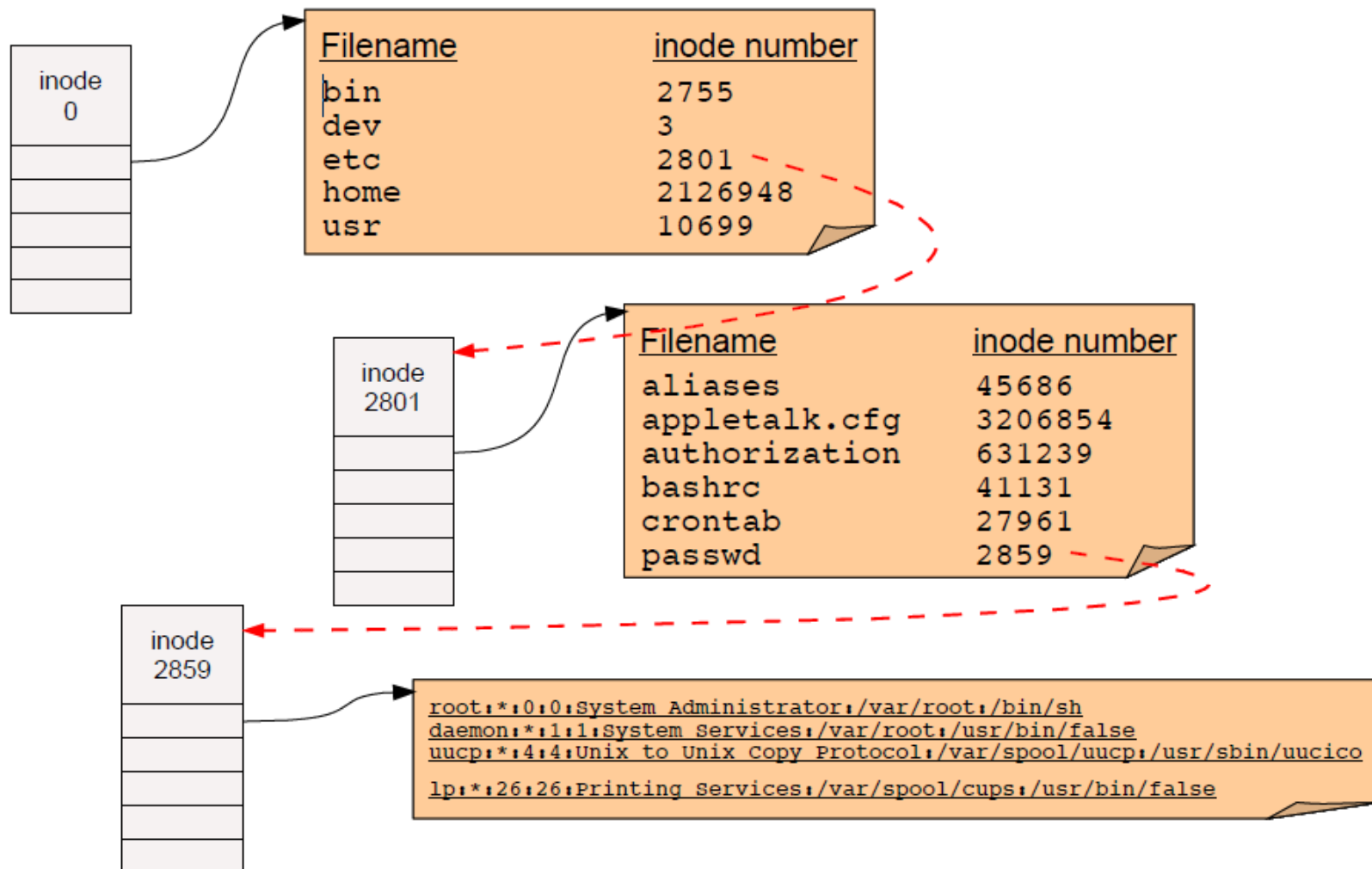


- 目录内容
  - 在UNIX 和Linux中，文件名不存在inode中
- 两个开放性问题
    - 如何查找根目录？如UNIX系统的 “/”
    - 如何从一个inode号码到达磁盘的inode所在位置？



# 路径名解释

## ■ 查找路径名/etc/passwd, 从根目录开始, 沿着inode链表查

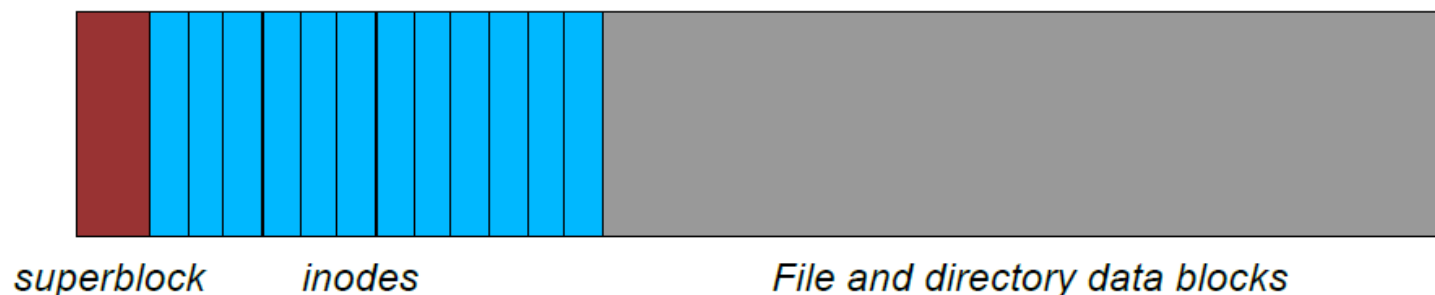


# 在磁盘上查找inodes

## ■ 查看目录可获得一个文件的inode号

- 如何在磁盘上找到inode本身？

## ■ 基本想法：文件系统顶部包含所有的inodes



- inode号仅是inode的一个索引
- 给定一个inode，容易计算其块地址

$$\text{block\_addr}(\text{inode\_num}) = \text{block\_offset\_of\_first\_inode} + (\text{inode\_num} * \text{inode\_size})$$

- 这意味着一个文件系统的**潜在inode节点数**是固定的？

最大inode数量通常在创建文件系统时设定

- 超级块存储文件系统布局的重要元数据，空闲块链表等

# 愚笨的目录小技巧

- 目录将文件名映射到inode号，这意味着什么？

- 我们可在不同目录创建指向同一inode的不同指针

- 或者同一目录下不同文件名

- 硬链接 vs. 软链接

- “file1” 和 “file1\_hl” 指向磁盘上的同一文件

不是拷贝，不管你用哪个名字去读或写文件，你看到的都是同样的内容

- 这跟“符号链接不一样”

```
[yuhong@FedoraDVD13 inode1]$ touch file1
[yuhong@FedoraDVD13 inode1]$ ln file1 file1_hl
[yuhong@FedoraDVD13 inode1]$ ln -s file1 file1_sl
[yuhong@FedoraDVD13 inode1]$ ls -il
总用量 0
153048 -rw-rw-r--. 2 yuhong yuhong 0 11月 24 19:59 file1
153048 -rw-rw-r--. 2 yuhong yuhong 0 11月 24 19:59 file1_hl
153058 lrwxrwxrwx. 1 yuhong yuhong 5 11月 24 19:59 file1_sl -> file1
[yuhong@FedoraDVD13 inode1]$ echo "Hello world" >> file1
```

```
[yuhong@FedoraDVD13 inode1]$ cat file1
Hello world
[yuhong@FedoraDVD13 inode1]$ cat file1_hl
Hello world
[yuhong@FedoraDVD13 inode1]$ cat file1_sl
Hello world
[yuhong@FedoraDVD13 inode1]$ rm file1_sl
[yuhong@FedoraDVD13 inode1]$ ls -il
总用量 8
153048 -rw-rw-r--. 2 yuhong yuhong 12 11月 24 20:00 file1
153048 -rw-rw-r--. 2 yuhong yuhong 12 11月 24 20:00 file1_hl
[yuhong@FedoraDVD13 inode1]$ rm file1_hl
[yuhong@FedoraDVD13 inode1]$ ls -il
总用量 4
153048 -rw-rw-r--. 1 yuhong yuhong 12 11月 24 20:00 file1
[yuhong@FedoraDVD13 inode1]$
```

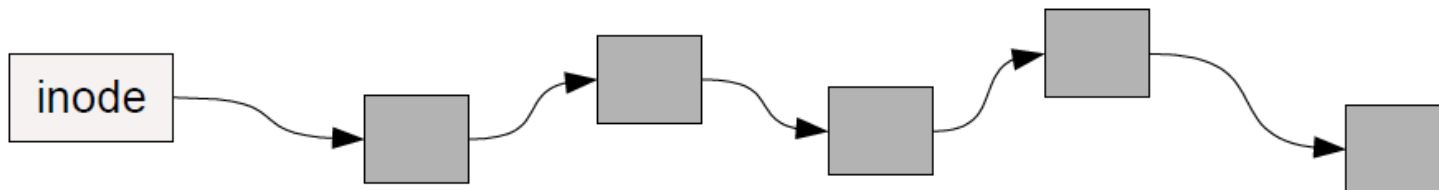
```
[yuhong@FedoraDVD13 inode1]$ ls
file1
[yuhong@FedoraDVD13 inode1]$ ln -s file1 file1_sl
[yuhong@FedoraDVD13 inode1]$ ln file1 file1_hl
[yuhong@FedoraDVD13 inode1]$ ls -il
总用量 8
153048 -rw-rw-r--. 2 yuhong yuhong 12 11月 24 20:00 file1
153048 -rw-rw-r--. 2 yuhong yuhong 12 11月 24 20:00 file1_hl
153058 lrwxrwxrwx. 1 yuhong yuhong 5 11月 24 20:05 file1_sl -> file1
[yuhong@FedoraDVD13 inode1]$ rm file1
[yuhong@FedoraDVD13 inode1]$ ls -l
总用量 4
-rw-rw-r--. 1 yuhong yuhong 12 11月 24 20:00 file1_hl
lrwxrwxrwx. 1 yuhong yuhong 5 11月 24 20:05 file1_sl -> file1
[yuhong@FedoraDVD13 inode1]$ cat file1_hl
Hello world
[yuhong@FedoraDVD13 inode1]$ cat file1_sl
cat: file1_sl: 没有那个文件或目录
[yuhong@FedoraDVD13 inode1]$
```

# 在磁盘上如何组织块？

## ■ 非常简单的策略：一个文件包含链接的块

- inode指向该文件的首块.
- 每一块都指向文件的下一块（磁盘上的一个链表）

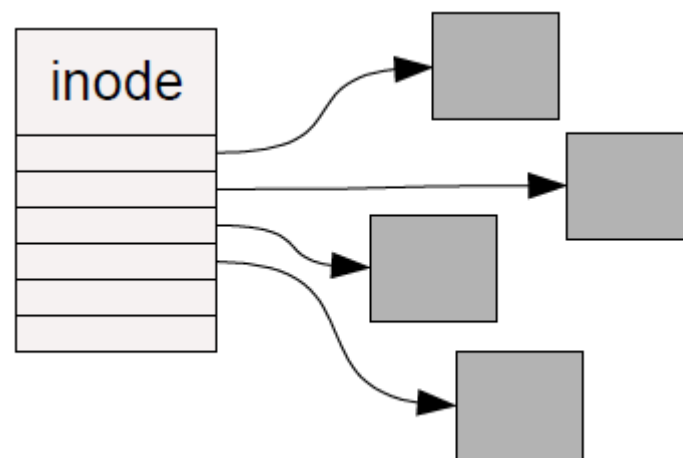
◆ 优缺点分别是？



## ■ 索引文件

- inode包含一系列包含文件的块号
- 文件创建时就分配了数组

◆ 优点和缺点分别是？

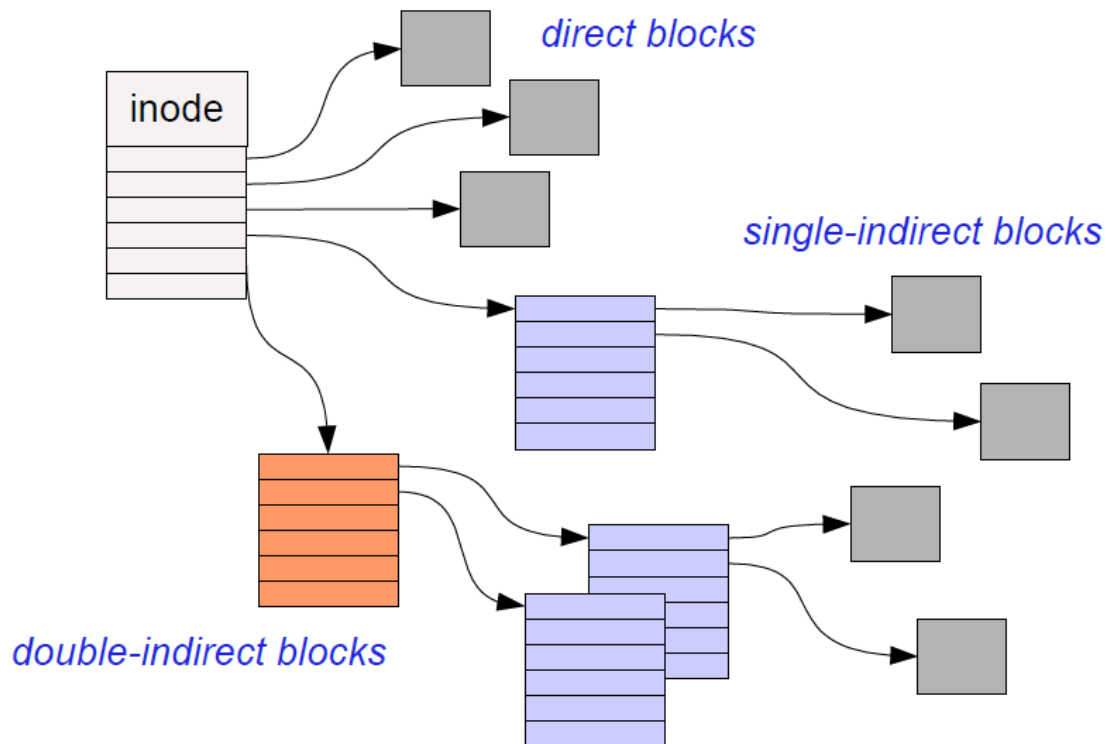


# 多级索引文件

## ■ inode包含一个10-15个直接块指针的列表

- 文件刚开始的几个块可以通过inode节点本身检索到.
- inode同时也包含指向单层、两层或是三层的非直接块指针

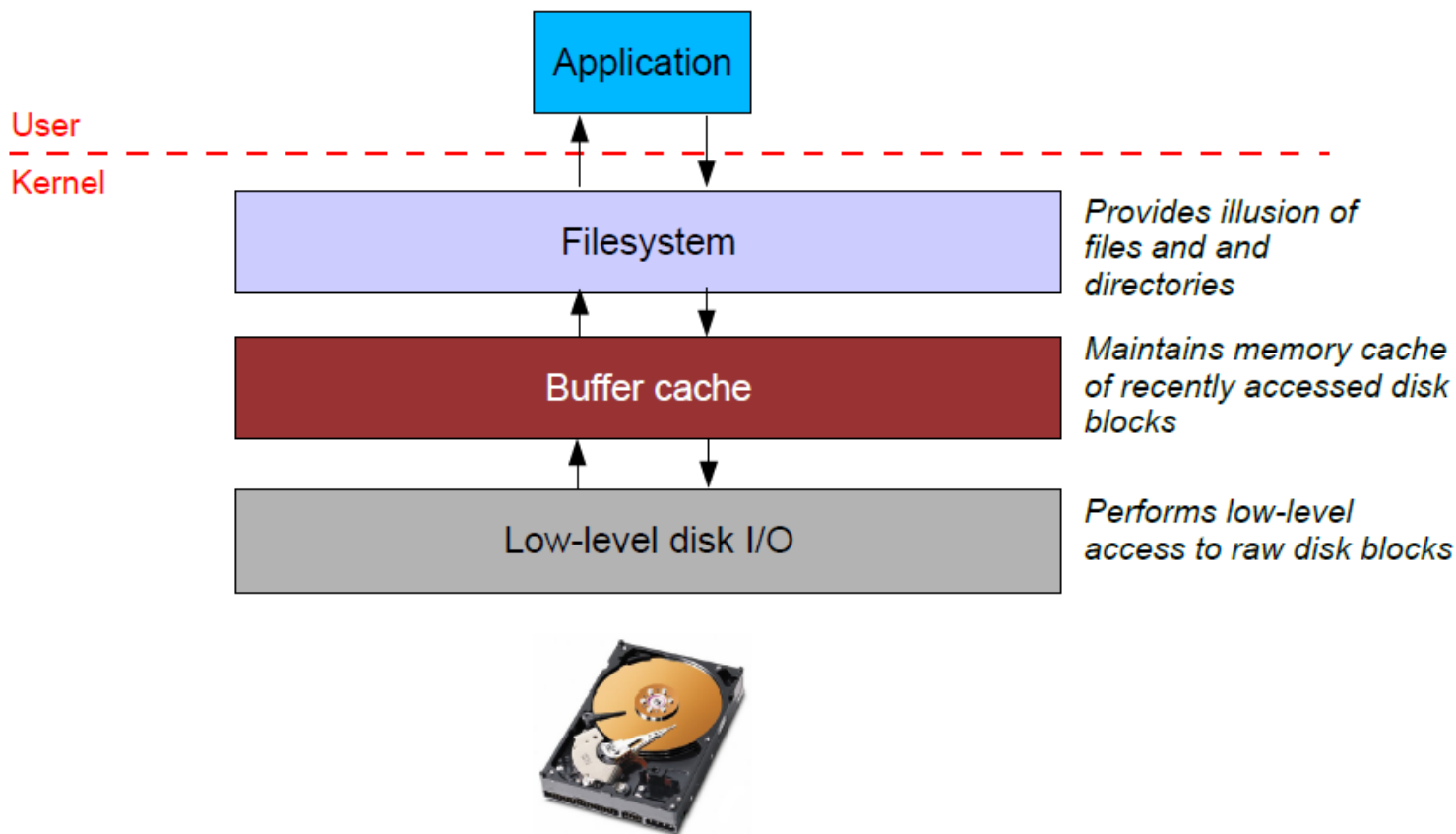
### ◆ 允许大文件



# 文件系统caching

## ■ 大多数文件系统将磁盘数据缓存在内存中

- 例如Linux试图将所有空闲物理空间都看做是一个大的缓存
- 避免对每个I/O引起的磁盘访问运行期开销



# 文件I/O

## ■ Linux SHELLs

- 文件描述符0:标准输入
- 文件描述符1:标准输出
- 文件描述符2: 标准出错

## ■ 在POSIX兼容的应用中

- 0: STDIN\_FILENO
- 1: STDOUT\_FILENO
- 2: STDERR\_FILENO

这些常数在<unistd.h>中定义

## ■ 文件描述符取值范围【0, OPEN\_MAX】

- 系统级进程能打开的文件描述符最大值

```
[root@FedoraDVD13 yuhongf]# cat /proc/sys/fs/file-max
49671
```

```
/proc/sys/fs/file-max
```

This file defines a system-wide limit on the number of open files for all processes. This limit is not applied when a root user (or any user with **CAP\_SYS\_ADMIN** privileges) is trying to open a file. (See also **setrlimit(2)**, which can be used by a process to set the per-process limit, **RLIMIT\_NOFILE**, on the number of files it may open.) If you get lots of error messages about running out of file handles, try increasing this value:

```
echo 100000 > /proc/sys/fs/file-max
```

The kernel constant **NR\_OPEN** imposes an upper limit on the value that may be placed in **file-max**.

If you increase **/proc/sys/fs/file-max**, be sure to increase **/proc/sys/fs/inode-max** to 3-4 times the new value of **/proc/sys/fs/file-max**, or you will run out of inodes.

man 5 proc 或 /etc/sysctl.conf

- 用户级进程能打开的文件描述符最大值

/etc/security/limits.conf

```
[root@FedoraDVD13 etc]# ulimit -n
1024
```

# 常用的I/O 系统调用

## ■ 打开和关闭文件

- `open()`
- `close()`

## ■ 读和写文件

- `read()`
- `write()`

## ■ 创建文件

- `creat()`

## ■ 改变当前文件位置

- `lseek()`

## ■ 删除文件

- `unlink()`

# 进阶级I/O 系统调用

## ■ 获取文件状态信息

- `stat()`
- `fstat()`

## ■ 获取目录条目

- `getdents()`



# open()

第一次课曾说过，不是所有的出错都exit(...)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

受文件名、路径名  
最大长度的限制

```
int open (const char *pathname,
          int flags[, mode_t mode])
```

打开文件的  
方式

*pathname* : 相对路径或绝对路径

*flags* : 打开方式，零个或多个可选标志按位“or”

*mode* : 被打开文件的存取权限

返回值: 成功为文件描述符 ( $\geq 0$ )，**“失败”为-1**

```
/* 文件fn存在，打开文件并返回文件描述符，否则  
创建并返回新文件的文件描述符 */
```

```
fd = open(fn, O_RDWR|O_CREAT, 666);
if (fd == -1) { perror(“Fail... ”); exit(-1); }
```

```
/* 文件fn存在，返回-1，否则创建 */
```

```
fd = open(fn, O_RDWR|O_CREAT|O_EXCL, 666);
if (fd == -1) { fd = open(fn, O_RDWR, 666); ... }
```

flags	含义
O_RDONLY	只读
O_WRONLY	只写
O_RDWR	可读可写

flags	含义
O_APPEND	追加写
O_CREAT	文件不存在则创建文件，此时，第三个参数mode生效
O_EXCL	创建文件，若同时指定O_CREAT且文件已存在，返回-1
O_TRUNC	若文件存在且以可写方式打开，则文件长度清零，内容被新写入内容覆盖

# 以下两个代码片段等价嘛？你选哪一种？Why？

```
/* 文件fn存在，返回-1，否则创建文件、打开文件并返回新文件描述符 */  
fd = open(fn, O_RDWR|O_CREAT|O_EXCL, 666);  
/* 返回-1，则打开已存在的文件 */  
if (fd == -1) {  
    fd = open(fn, O_RDWR, 666);  
    ...  
}
```

```
/* 判断文件fn是否存在 */  
if (access(fn, R_OK) == -1) {  
    /* 不存在则创建文件、打开文件并返回新文件描述符*/  
    fd = open(fn, O_RDWR|O_CREAT, 666);  
    ...  
}
```

# creat()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

受文件名、路径名的  
最大长度限制

```
int creat (const char *pathname, mode_t mode)
```

*pathname* : 相对路径或绝对路径

*mode* : 被打开文件的存取权限

返回值: 成功返回文件描述符 ( $\geq 0$ ), 否则 -1

受进程umask  
值的限制

mode	含义
S_IRUSR 或 S_IREAD	0400
S_IWUSR 或 S_IWRITE	0200
S_IXUSR 或 S_IEXEC	0100
S_IRGRP	0040
S_IWGRP	0020
S_IXGRP	0010
S_IRWXO	0007

```
sprintf(fn, ".rev.%d", getpid());
if (fd = creat(fn, S_IRUSR|S_IWUSR)) == -1) {
    perror("Fail ... "); exit(-1);
}
```

```
sprintf(fn, ".rev.%d", getpid());
fd = open(fn, O_RDWR | O_CREAT | O_TRUNC, 0600);
if (fd == -1) {
    perror("Fail ... "); exit(-1);
}
```

# close ()

```
#include <unistd.h>
```

```
int close (int filedes)
```

返回值：成功返回文件描述符 ( $\geq 0$ )，否则 -1

- 当一个进程结束时，内核将自动关闭它打开的所有文件
- 显式关闭不必要的文件描述符是一个好习惯

# read ()

```
#include <unistd.h>
```

```
ssize_t read (int fd, void *buf, size_t count)
```

- 从fd 所指的文件中读取 count字节到起始地址为buf的缓冲区
- 返回值： 成功返回读取的字节数(0表示到达文件结尾), 否则-1

## ■ 读文件时

- 一次一个字符： I/O次数多， 延长程序的运行时间
- 一次最多读取 “BUFFER\_SIZE” 字符

```
chars_read = read(fd, buffer, BUFFER_SIZE);
```

```
if (chars_read == 0) {...}; /* EOF */
```

```
if (chars_read == -1) fatalError(); /* Error */
```

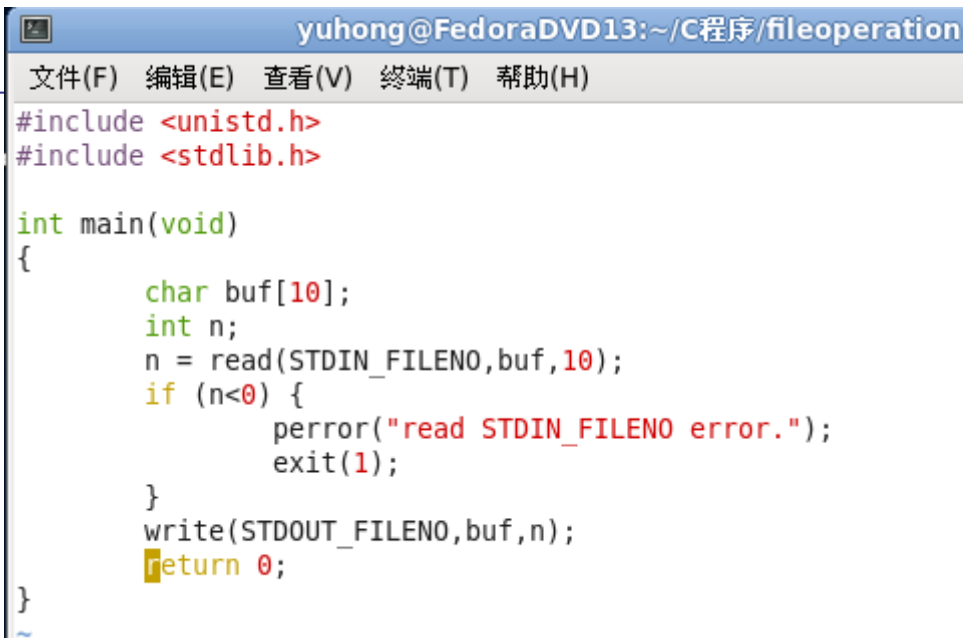
# write ()

```
#include <unistd.h>
```

```
ssize_t write (int fd, void *buf, size_t count)
```

- 从起始地址为buf的缓冲区拷贝count字节到fd所指的文件
- 返回值：成功返回写入的字节数( $\geq 0$ , 0表示什么也没写)  
失败返回-1  
如果count==0, 且fd指向常规文件, 系统视情况返回(0或-1)

此代码有多少种  
出错的可能?



```
yuhong@FedoraDVD13:~/C程序/fileoperation
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
    char buf[10];
    int n;
    n = read(STDIN_FILENO, buf, 10);
    if (n < 0) {
        perror("read STDIN_FILENO error.");
        exit(1);
    }
    write(STDOUT_FILENO, buf, n);
    return 0;
}
```

# lseek() - 设置读或写文件的偏移量

```
#include <sys/types.h>
#include <unistd.h>
```

偏移量值超过文件大小时，  
不改变文件大小

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

whence	含义
SEEK_SET	读写位置 = 文件开头+ offset
SEEK_CUR	读写位置 = 当前位置 + offset
SEEK_END	读写位置 = 文件大小+ offset

返回值：成功返回当前读写位置（距离文件开头），否则 -1

```
lseek(fd, array[i], SEEK_SET );
```

```
chars_read = read(fd, buffer, array[i+1]-array[i] );
```

```
curOffset = lseek(fd, 0, SEEK_CUR );
```

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
main()
{
```

## lseek() 应用例子

```
    int i, fd;
    /* Creates a sparse file */
    fd = open( "sparse.txt", O_CREAT | O_RDWR, 0600 );
    write( fd, "space", 6 );
    lseek( fd, 60006, SEEK_SET );
    write( fd, "file", 4 );
    close(fd);
    /* Create a normal file */
    fd = open( "normal.txt", O_CREAT | O_RDWR, 0600 );
    write( fd, "normal", 6 );
    for ( i=1; i<=60000; i++ )
        write( fd, "/0", 1 );
    write( fd, "file", 4 );
    close(fd);
}
```

```
$ sparse          ---> 运行.
$ ls -l *.txt     ---> 查看相关文件
-rw-r--r--  1  glass      60010 Feb  14  15:06 normal.txt
-rw-r--r--  1  glass      60010 Feb  14  15:06 sparse.txt
$ ls -s *.txt     ---> 列出块使用数.
  60 normal.txt*   ---> 使用60块.
   8 sparse.txt*   ---> 仅使用8块.
$
```



# unlink() – 文件硬链接减1，硬链接为0则删除文件

```
#include <unistd.h>
int unlink (const char *pathname);
```

返回值：成功返回0, 否则-1

- 如果pathname的硬链接数为1
  - 进程open(pathname,...), 还没有close()该文件
  - 进程调用unlink(pathname)
  - 进程继续运行
- 请问此时， pathname所指向的文件还在文件系统中吗？

注意：

执行unlink()函数并不一定会真正的删除文件，它先会检查文件系统中此文件的连接数是否为1，如果不是1说明此文件还有其他链接对象，因此只对此文件的连接数进行减1操作。若连接数为1，并且在此时没有任何进程打开该文件，此内容才会真正地被删除掉。在有进程打开此文件的情况下，则暂时不会删除，直到所有打开该文件的进程都结束时文件就会被删除。

<https://blog.csdn.net/judgejames/article/details/83749669>

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void minuslink(int sig)
{
    printf("to minus 1 link of linkamount.txt\n");
    unlink("./linkamount.txt");
    signal(SIGINT,SIG_DFL);
}

int main()
{
    signal(SIGINT,minuslink);
    int fd = open("./linkamount.txt",O_RDWR,0600);
    while(1){}
```

博客很好，无数人点滴尝试  
的总结与分享。  
但不可尽信。

```
[szu@taishan02-vm-10 fileoperation]$ touch linkamount.txt
[szu@taishan02-vm-10 fileoperation]$ ls -l
总用量 40
-rw-rw-r--. 1 szu szu      0 10月  9 09:39 linkamount.txt
-rwx-----. 1 szu szu 71712  5月  2 17:45 test
-rw-----. 1 szu szu   987  5月  2 17:45 test.c
-rwxrwxr-x. 1 szu szu 71328 10月  9 09:30 unlink
-rw-rw-r--. 1 szu szu   360 10月  9 09:29 unlink.c
[szu@taishan02-vm-10 fileoperation]$ ls -l
总用量 40
-rw-rw-r--. 1 szu szu      0 10月  9 09:39 linkamount.txt
-rwx-----. 1 szu szu 71712  5月  2 17:45 test
-rw-----. 1 szu szu   987  5月  2 17:45 test.c
-rwxrwxr-x. 1 szu szu 71328 10月  9 09:30 unlink
-rw-rw-r--. 1 szu szu   360 10月  9 09:29 unlink.c
[szu@taishan02-vm-10 fileoperation]$ ls -l
总用量 40
-rwx-----. 1 szu szu 71712  5月  2 17:45 test
-rw-----. 1 szu szu   987  5月  2 17:45 test.c
-rwxrwxr-x. 1 szu szu 71328 10月  9 09:30 unlink
-rw-rw-r--. 1 szu szu   360 10月  9 09:29 unlink.c
```

# stat(), fstat(), lstat() – 获取文件信息

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

```
int stat(const char *path, struct stat *buf);
```

■ 通过文件名path获取文件信息，保存到buf所指的 struct stat 中

```
int fstat(int fd, struct stat *buf);
```

■ 通过文件描述符fd获取文件信息，保存到buf所指的 struct stat 中

```
int lstat(const char *path, struct stat *buf);
```

■ 当path为普通文件时，同stat()。

■ 当path为符号链接时，获取符号链接的信息，而不是它所指向的文件的信息

返回值：成功返回0，否则-1，同时设置errno

# stat(), fstat(), lstat() – 文件信息结构体stat

```
struct stat {  
    dev_t    st_dev;        /* 文件的设备编号 */  
    ino_t    st_ino;        /* inode节点号 */  
    mode_t   st_mode;       /* 文件的类型和存取权限*/  
    nlink_t  st_nlink;      /* 硬链接数，刚创建的文件硬链接数为1 */  
    uid_t    st_uid;        /* 用户ID */  
    gid_t    st_gid;        /* 组ID */  
    dev_t    st_rdev;       /* 设备ID（如果是设备类型特殊文件）*/  
    off_t    st_size;       /* 文件大小（字节总数）*/  
    blksize_t st_blksize;   /* 文件系统I/O的块大小 */  
    blkcnt_t st_blocks;     /* 分配的512字节大小的块数 */  
    time_t   st_atime;      /* 最后一次访问的时间 */  
    time_t   st_mtime;      /* 最后一次修改的时间 */  
    time_t   st_ctime;      /* 最后一次属性改变的时间 */  
};
```

# 文件信息结构体stat中st\_mode的取值

```
struct stat {  
    dev_t    st_dev;        /* 文件的设备编号 */  
    ino_t    st_ino;        /* inode节点号 */  
    mode_t   st_mode;       /* 文件的类型和存取权限*/  
    nlink_t  st_nlink;      /* 硬链接数，刚创建的文件硬链接数为1 */  
    uid_t    st_uid;        /* 用户ID */  
    gid_t    st_gid;        /* 组ID */  
    dev_t    st_rdev;       /* 设备ID（如果是设备类型特殊文件）*/  
    off_t    st_size;       /* 文件大小（字节总数）*/  
    blksize_t st_blksize;   /* 文件系统I/O的块大小 */  
    blkcnt_t st_blocks;     /* 分配的512字节大小的块数 */  
    time_t   st_atime;       /* 最后一次访问的时间 */  
    time_t   st_mtime;       /* 最后一次修改的时间 */  
    time_t   st_ctime;       /* 最后一次属性改变的时间 */  
};
```

# stat()应用例子

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    struct stat buf;
    stat("./stat.c", &buf);
    printf( "./stat.c file size = %d,
            hard link= %d\n", buf.st_size, buf.st_nlink);
}
```

```
[szu@taishan02-vm-10 fileoperation]$ gcc -o stat stat.c
[szu@taishan02-vm-10 fileoperation]$ ./stat
./stat.c file size = 218, hard link= 1
```