第4讲 文件操作

主要内容

- 4.1 概述
- 4.2 文件操作
- 4.3 目录
- 4.4 文件与目录属性
- 4.5 标准文件I/O
- 4.6 处理系统调用中的错误

4.1 概述

- UNIX秉承“一切皆文件”的思想，对设备和资源的访问多以文件的形式进行。因此文件系统是Linux系统的主要功能之一。
- 文件的类型：
  - “ls -l”命令打印的文件属性信息中的第一个字符，便代表文件的类型，该字符有7种取值，分别对应不同的文件：

①d：directory，目录文件；

②l：link，符号链接文件；

③s：socket，套接字文件；

④b：block，块设备文件；

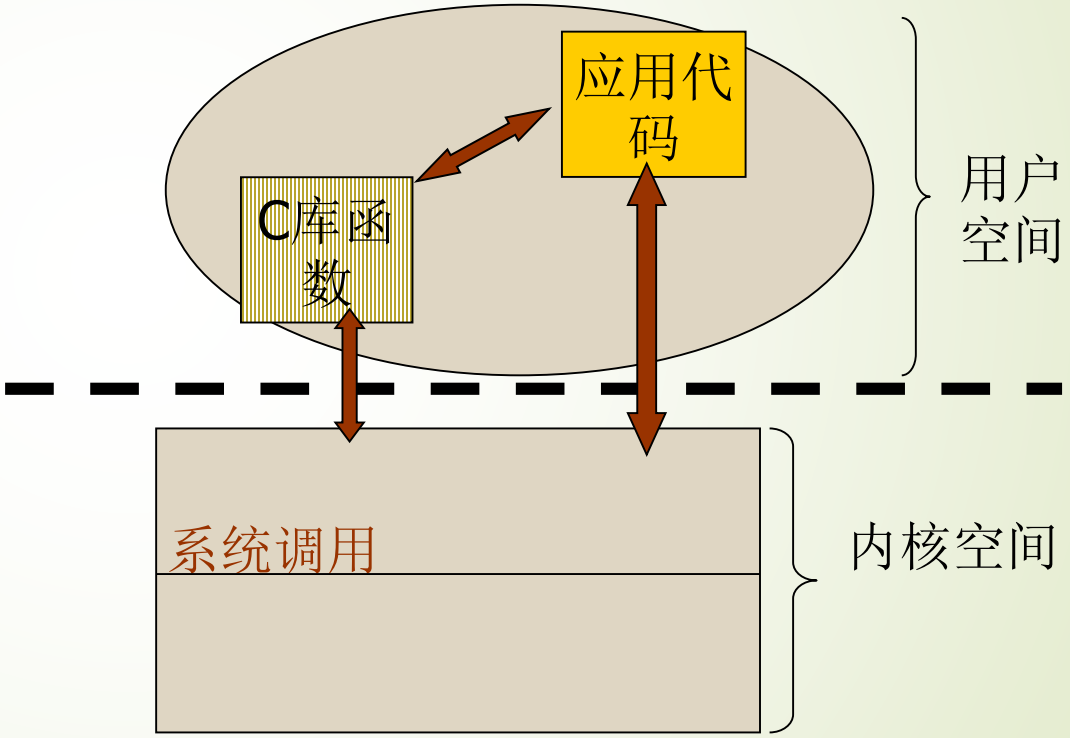
⑤c：character，字符设备文件；

⑥p：pipe，管道文件；

⑦-：不属于以上任一种文件的普通文件。

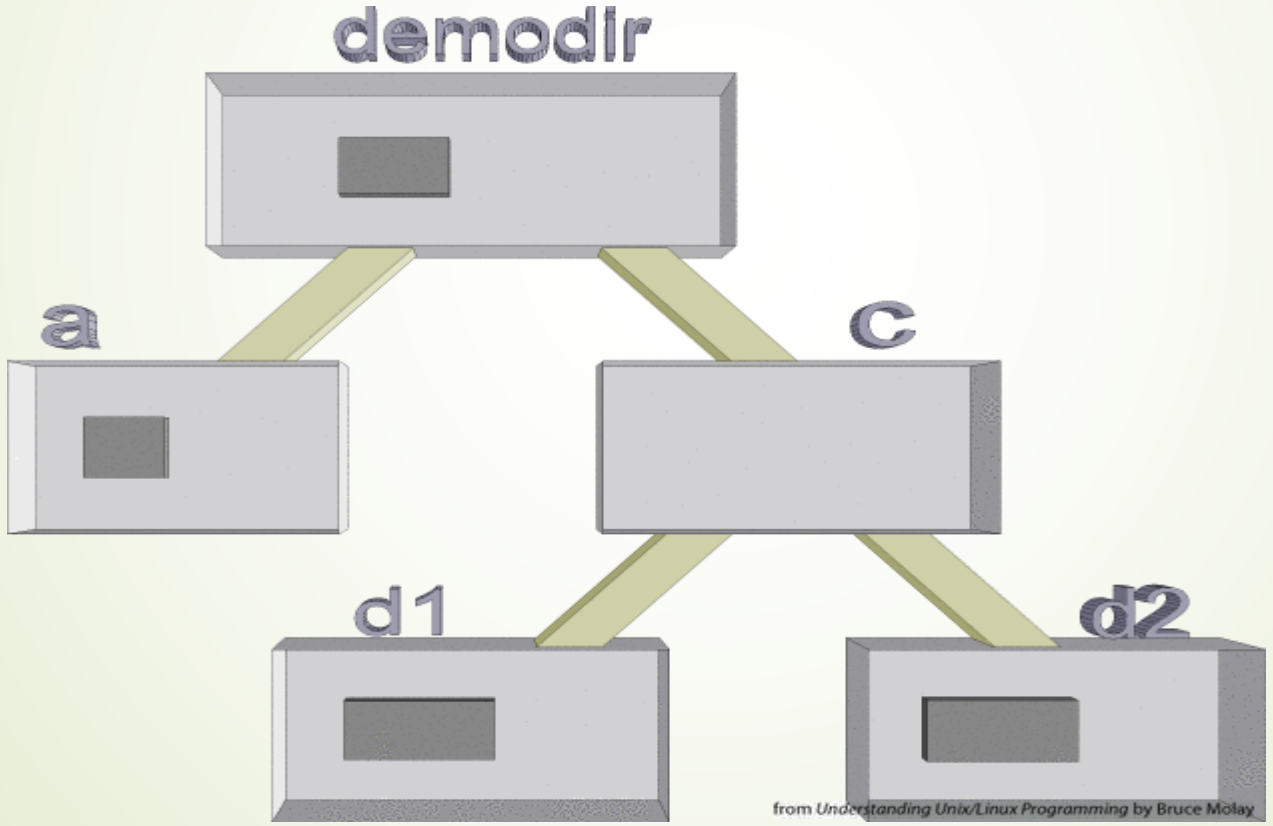
了解系统调用

- 文件操作（系统调用）
  - 打开 open
  - 创建 creat
  - 定位 lseek
  - 读 read
  - 写 write
  - 关闭 close

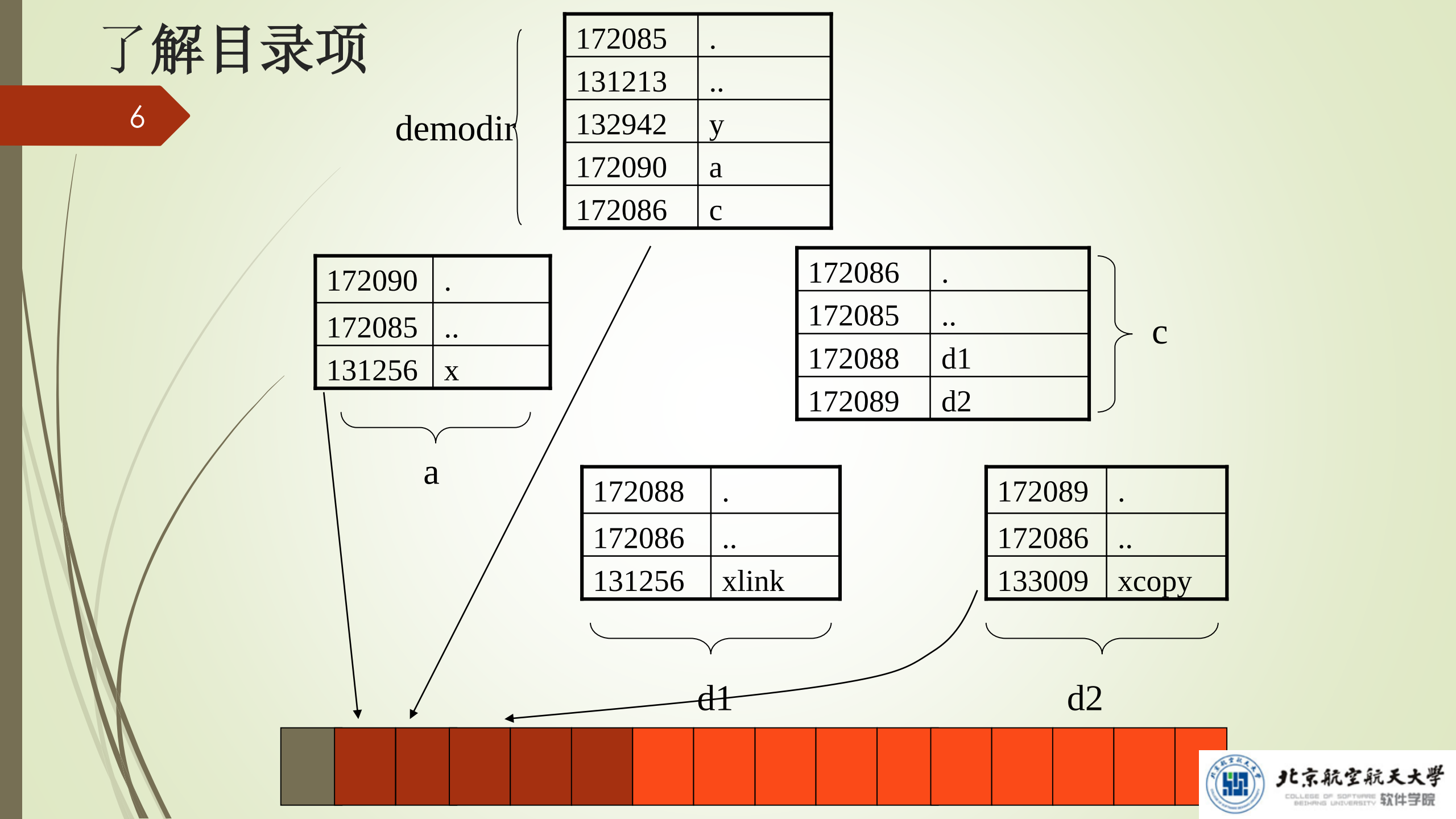


了解目录

- 用户角度所看到的目录结构

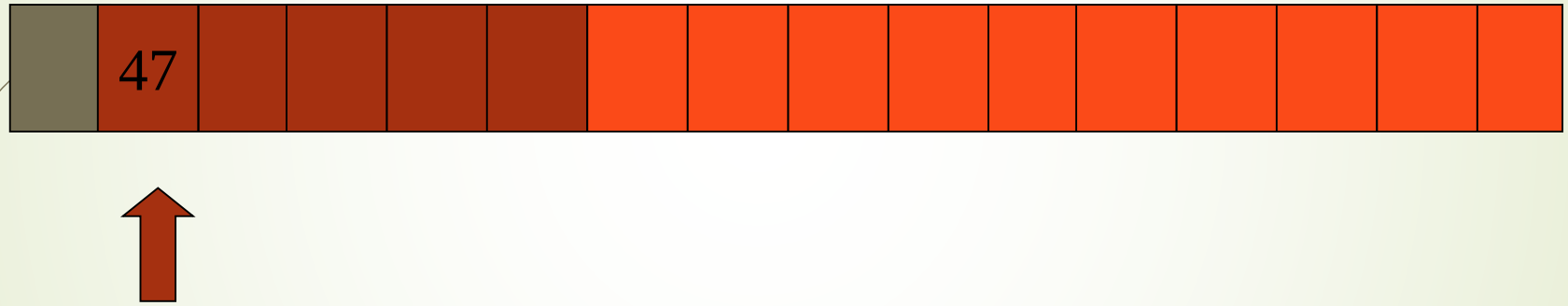


了解目录项



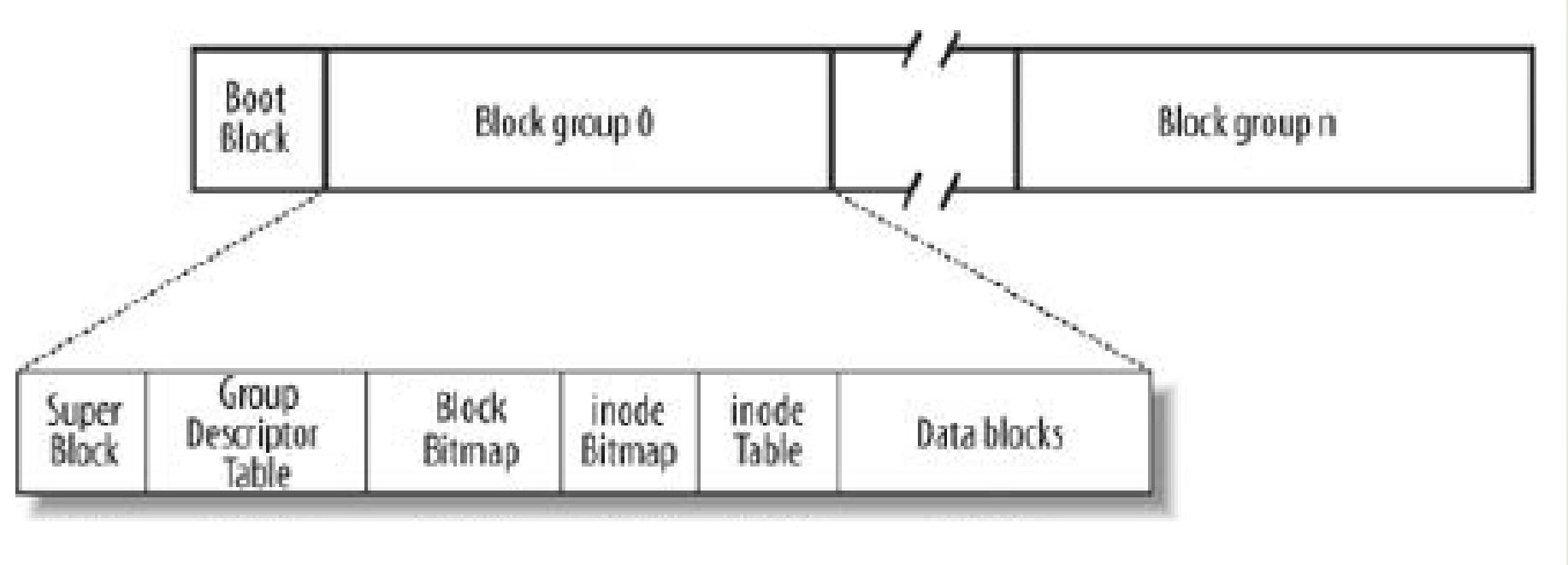
了解inode表

- 内核先找到一个空的i节点，内核将文件信息存储在其中



inode存储文件属性

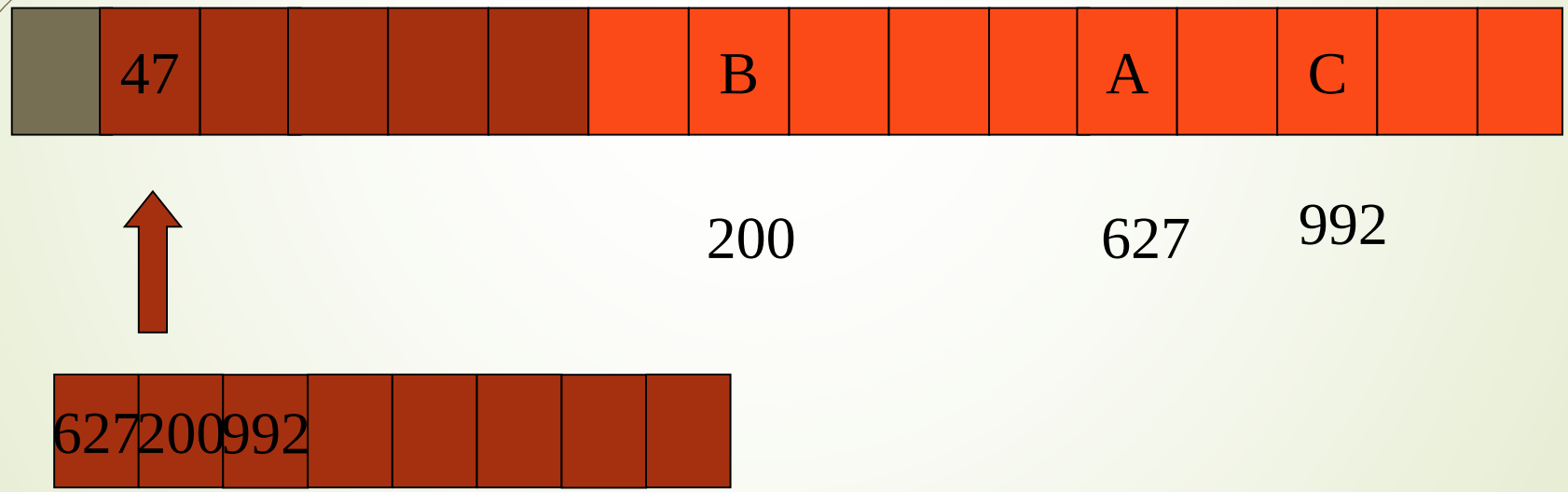
了解ext2文件系统



- 超级块(Super Block)：每个块组中的第一个数据块，这个块存放整个文件系统本身的信息，包括inode 数、块数、空闲块数、空闲inode 数、第一个数据块位置、块长度等信息

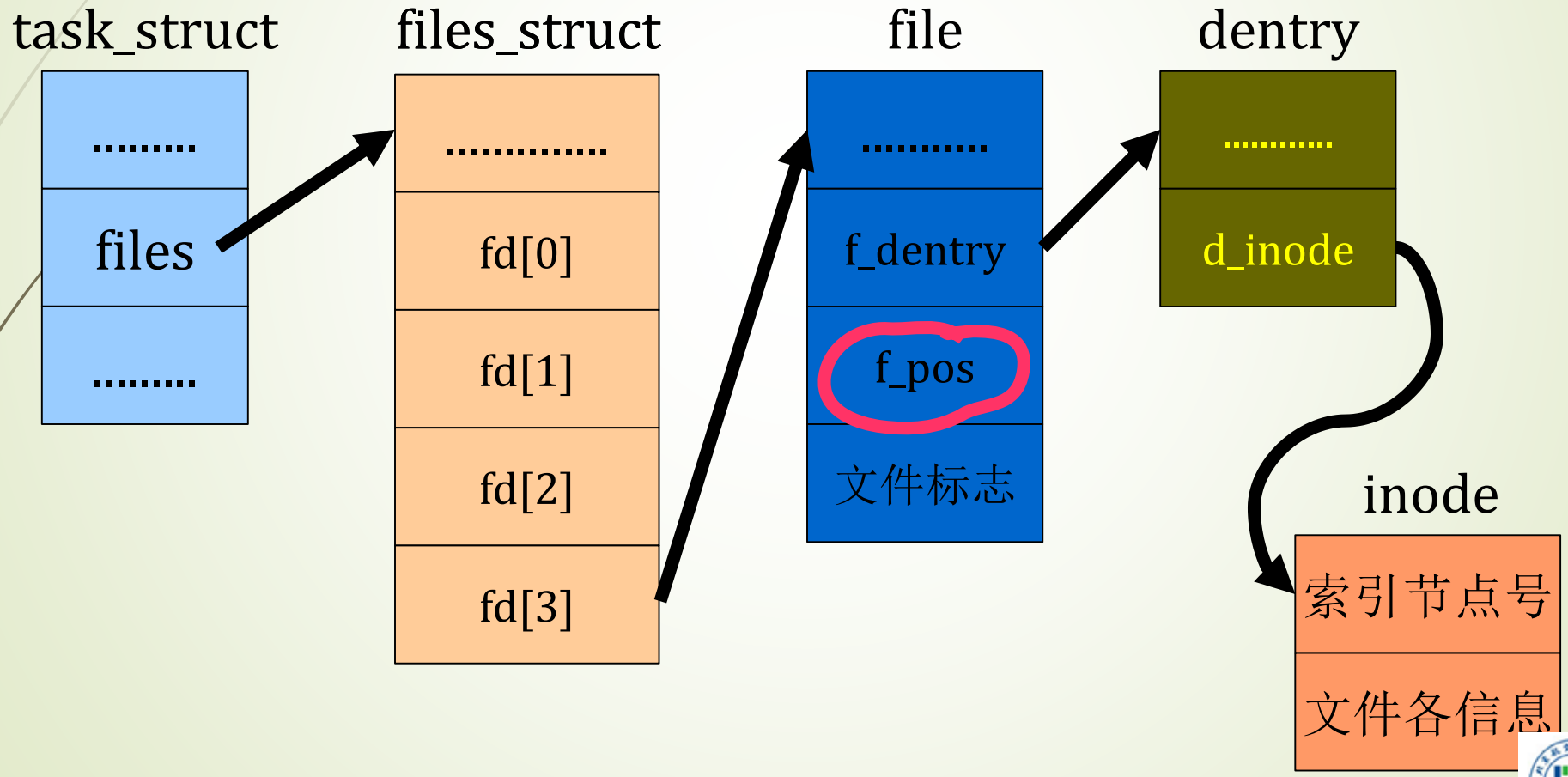
了解Inode 与 块存储

- 假设文件需要3个数据块存储
- 内核从未使用的块列表中找出3个自由块
- 这3个块分别为： 627、 200、992



将文件所使用块记录到i节点中

了解进程及其打开文件的数据结构



主要内容

- 4.1 概述
- 4.2 文件操作
- 4.3 目录
- 4.4 文件与目录属性
- 4.5 标准文件I/O
- 4.6 处理系统调用中的错误

文件操作基本顺序

- 打开　　open
- 创建　　creat
- 定位　　lseek
- 读　　read
- 写　　write
- 关闭　　close

文件的打开 open函数

- 用于打开或者创建一个文件
- 函数原型
  - #include<fcntl.h>
  - int open(const char* pathname, int flags, ...)
- 参数
  - 第一个参数pathname：要打开或者创建的文件名
  - 第二个参数flags：用于指定文件打开模式、标志等信息。

open函数

- 第二个参数flag:
  - Linux头文件已经为文件打开模式、标志等定义了若干的宏
  - flags需要指定这些宏
  - 宏定义在/usr/include/bits/fcntl.h中
  - 在该头文件中，只读打开标志被定义为:

```
#define O_RDONLY        00
```

open函数

- flags：
  - 文件打开模式标志

    以下三个标志必须指定一个且只能指定一个
    - O_RDONLY：　　只读打开
    - O_WRONLY：　　只写打开
    - O_RDWR　　：　　读写打开
  - 其他文件标志

    下面的标志是可以选择的，可通过C语言的或运算与文件打开标志进行组合

open函数

- flags
  - 其他文件标志：

    O_APPEND：每次写的数据都添加到文件尾

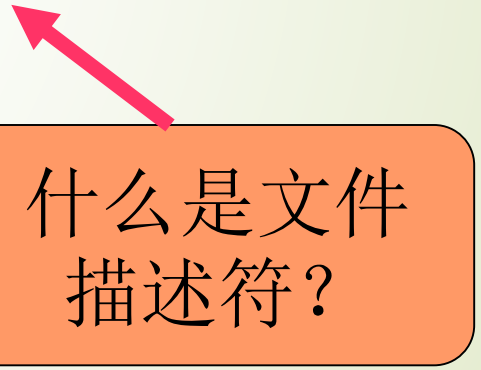
    O_TRUNC：若此文件存在，并以读写或只写打开，则文件长度为0

    O_CREAT：若文件不存在，则创建该文件。此时，open函数需要第三个参数，用于指定该文件的访问权限位（后面描述）

    O_EXCL：若同时指定了O_CREAT标志，而文件已经存在，则会出错。可用于测试文件是否存在

open函数

- 返回值
  - int open(const char* pathname, int oflag, …)
  - 返回值：整型数据
    - 成功时，返回文件描述符
    - 出错时，返回-1



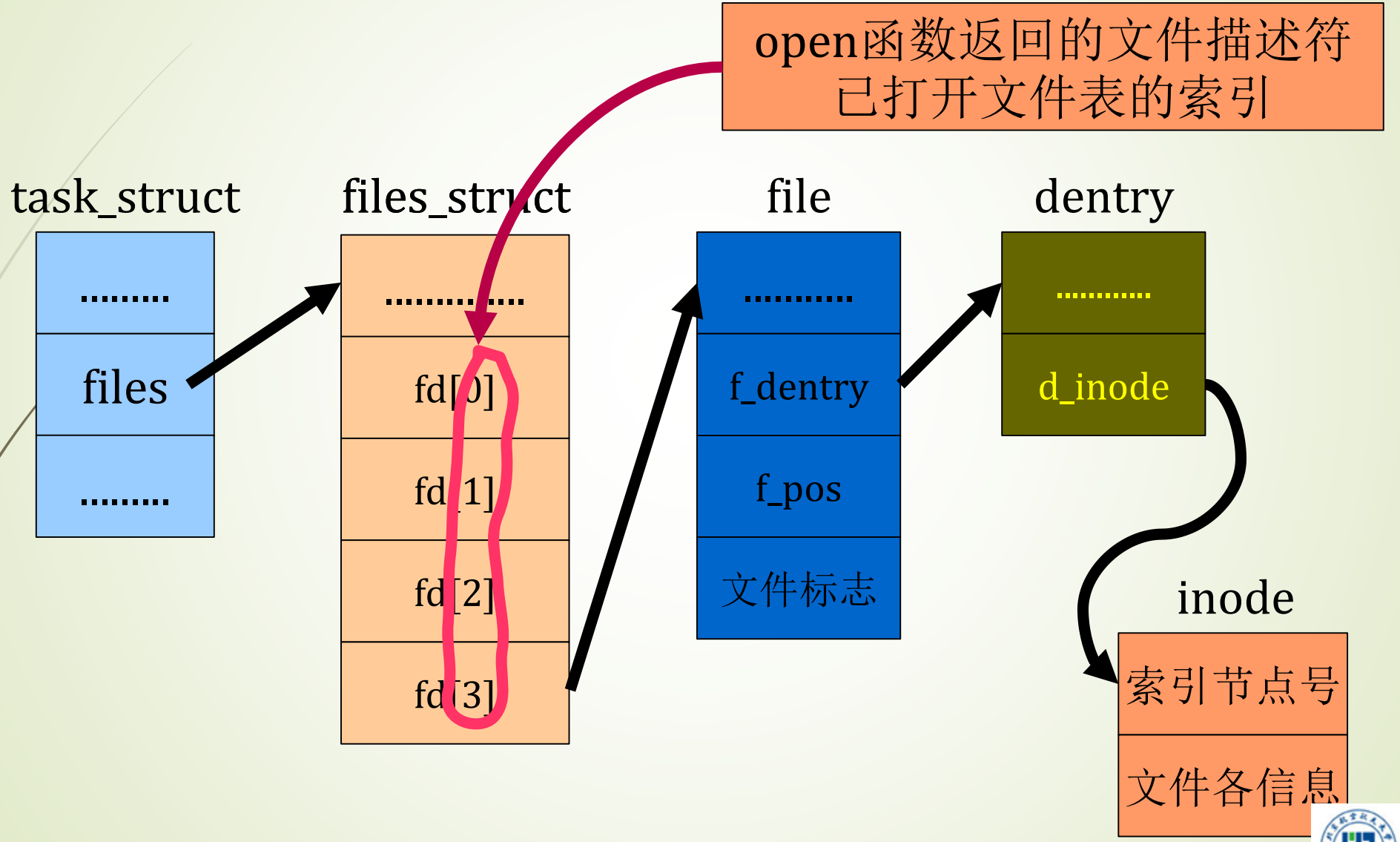
文件描述符

- 文件描述符的本质是什么？
- 通过文件描述符怎么样能找到需访问的文件？
- 需要了解进程打开文件时，内核创建或涉及到的一系列数据结构

文件描述符

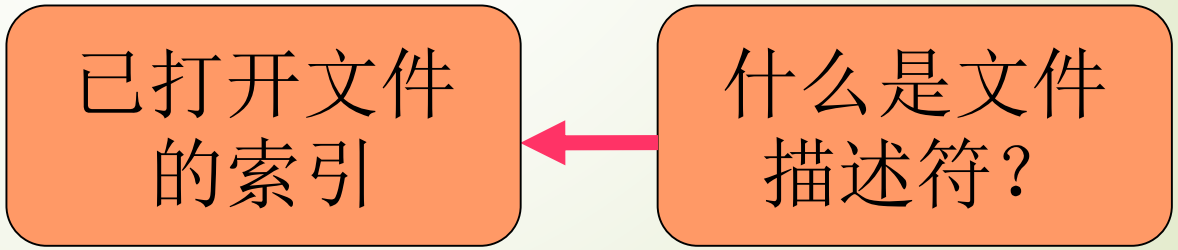
- 文件描述符是已打开文件的索引，通过该值可以在fd_array表中检索相应的文件对象
- 文件描述符是一个非负的整数
- 文件描述符0、1、2分别对应于标准输入、标准输出、标准出错，在进程创建时，已经打开。

文件描述符



open函数

- 返回值
  - int open(const char* pathname, int flags, …)
  - 返回值：整型数据
    - 成功时，返回文件描述符
    - 出错时，返回-1



open函数

- 返回值
  - int open(const char* pathname, int flags, …)
  - 返回值：整型数据
    - 成功时，返回文件描述符
    - 出错时，返回-1

索引节点

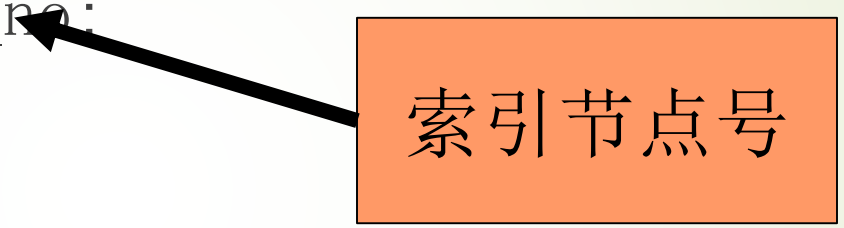
- 文件系统索引节点的信息，存储在磁盘上
- 当需要时，调入内存，填写VFS的索引节点（即inode结构）
- 每个文件都对应了一个索引节点
- 通过索引节点号，可以唯一的标识文件系统中的指定文件

索引节点

```
struct inode{
    ..........................
    unsigned long i_no;
    umode_t i_mode;
    uid_t i_uid;
    gid_t i_gid;
    off_t i_size;
    time_t i_atime;
    time_t i_mtime;
 };
```

索引节点

```
struct inode{
    ..........................
    unsigned long i_no;
    umode_t i_mode;
    uid_t i_uid;
    gid_t i_gid;
    off_t i_size;
    time_t i_atime;
    time_t i_mtime;
 };
```



索引节点

```
struct inode{
    ..........................
    unsigned long i_no;
    umode_t i_mode;
    uid_t i_uid;
    gid_t i_gid;
    off_t i_size;
    time_t i_atime;
    time_t i_mtime;
};
```

文件类型访问权限

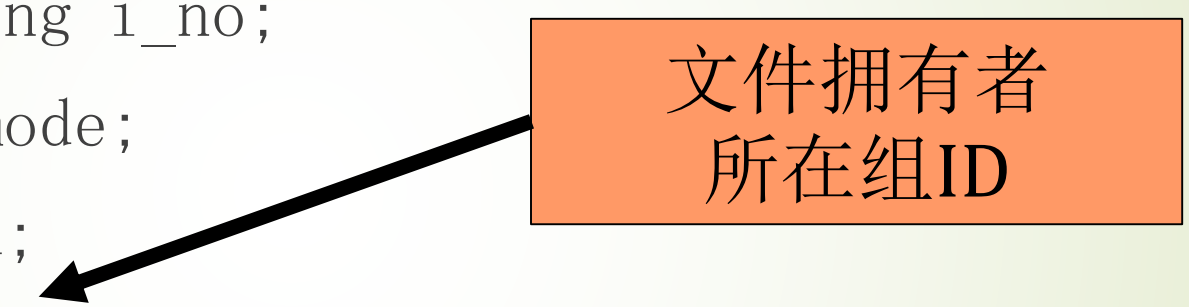
索引节点

```
struct inode{
    ..........................
    unsigned long i_no;
    umode_t i_mode;
    uid_t i_uid;
    gid_t i_gid;
    off_t i_size;
    time_t i_atime;
    time_t i_mtime;
 };
```

文件拥有者ID

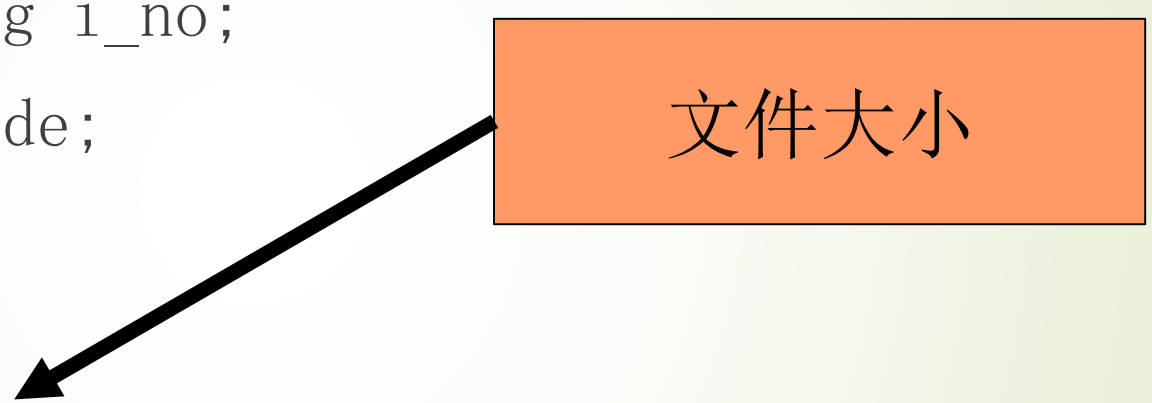
索引节点

```
struct inode{
    ..........................
    unsigned long i_no;
    umode_t i_mode;
    uid_t i_uid;
    gid_t i_gid;
    off_t i_size;
    time_t i_atime;
    time_t i_mtime;
 };
```



索引节点

```
struct inode{
    ..........................
    unsigned long i_no;
    umode_t i_mode;
    uid_t i_uid;
    gid_t i_gid;
    off_t i_size;
    time_t i_atime;
    time_t i_mtime;
 };
```

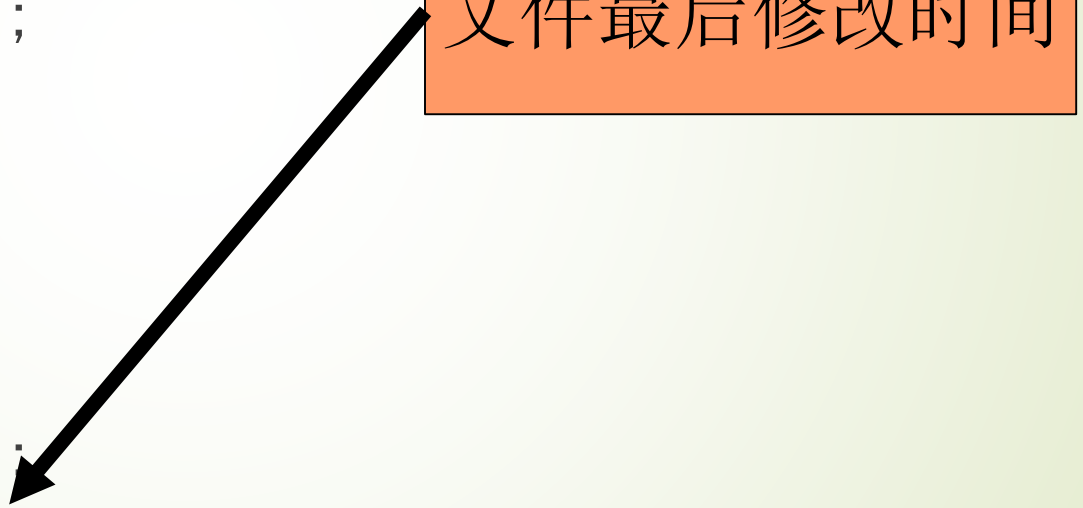


索引节点

```
struct inode{
    ..........................
    unsigned long i_no;
    umode_t i_mode;
    uid_t i_uid;
    gid_t i_gid;
    off_t i_size;
    time_t i_atime;
    time_t i_mtime;
 };
```

索引节点

```
struct inode{
    ..........................
    unsigned long i_no;
    umode_t i_mode;
    uid_t i_uid;
    gid_t i_gid;
    off_t i_size;
    time_t i_atime;
    time_t i_mtime;
 };
```



creat 函数

- 用于创建一个新文件
- 函数原型

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

- 参数
  - pathname：要创建的文件名（包括路径信息）
  - mode：同open的第三个参数，讨论文件的访问权限位时分析
- 返回值
  - 成功返回只写打开的文件描述符
  - 出错返回-1

creat函数

- creat函数的功能可以用open函数实现

```
open(pathname,
        O_WRONLY | O_CREAT | O_TRUNC,
        mode);
```

- 为什么需要指定O_TRUNC标志
- 当文件存在时，调用creat函数，会将文件的大小变为0

creat函数

- creat函数缺点：它以只写方式打开所创建的文件。若要创建一个临时文件，并先写该文件，然后又读该文件，则必须先调用creat，close，然后再open。简便方法：

```
open(pathname,
        O_RDWR | O_CREAT | O_TRUNC,
    mode);
```

文件操作

- 打开 open
- 创建 creat
- 定位 lseek
- 读 read
- 写 write
- 关闭 close

lseek函数

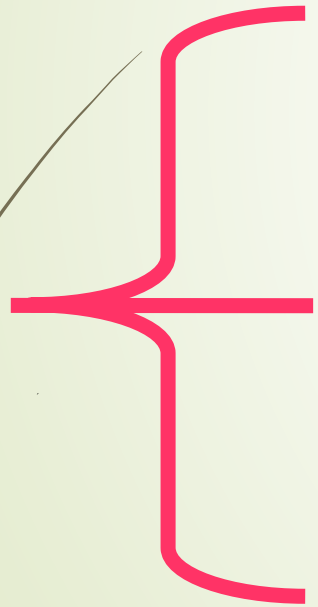
- lseek函数用于修改当前文件偏移量
- 当前文件偏移量的作用
  - 规定了从文件什么地方开始进行读、写操作
- 通常，读、写操作结束时，会使文件偏移量增加读写的字节数
- 当打开一个文件时，除非指定了O_APPEND标志，否则偏移量被设置为0

lseek函数

- 函数原型：
  - off_t lseek(int filedes, off_t offset, int whence)
- 参数
  - 第一个参数filedes：open/creat函数返回的文件描述符
  - 第二个参数offset：
    - 相对偏移量：需结合whence才能计算出真正的偏移量
    - 类型off_t：通常情况下是32位数据类型

lseek函数

- 参数
  - 第三个参数Whence：该参数取值是三个常量之一



SEEK_SET：　当前文件偏移量为：
距文件开始处的offset个字节

SEEK_CUR：　当前文件偏移量为：
当前文件偏移量＋offset(可正可负)

SEEK_END：　当前文件偏移量为：
当前文件长度＋offset(可正可负)

lseek函数

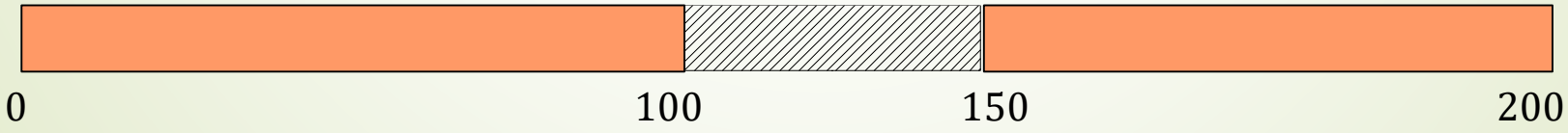
- 返回值：
  - 若成功，返回新的文件偏移量
  - 若出错，返回-1
- 获得当前的偏移量
  - off_t CurrentPosition;
  - CurrentPosition = lseek(fd, 0, SEEK_CUR);
- lseek操作并不引起任何I/O操作，只是修改内核中的记录

空洞文件

- 使用lseek修改文件偏移量后，当前文件偏移量有可能大于文件的长度
- 在这种情况下，对该文件的下一次写操作，将加长该文件
- 这样文件中形成了一个空洞。对空洞区域进行读，均返回0

文件长度是多少？
200？150？

lseek(fd, 50, SEEK_END)
write(fd, buf, 50)



文件操作

- 打开 open
- 创建 creat
- 定位 lseek
- 读 read
- 写 write
- 关闭 close

read函数

- 用于从文件中读出数据
- 函数原型

```
ssize_t read(int fd, void *buff,
                    size_t nbytes)
```

- 参数
  - 第一个参数fd：文件描述符
  - 第二个参数buff：指向缓冲区，用于存放从文件读出的数据
  - 第三个参数nbytes：unsigned int；需要从文件中读出的字节数
    - 缓冲区的大小>=nbytes

read函数

- 返回值
  - 返回值类型：ssize_t，即int
  - 出错：返回-1
  - 成功：返回从文件中实际读到的字节数
    当读到文件结尾时，则返回0

read函数

- 很多情况下，read实际读出的字节数都小于要求读出的字节数
  - 读普通文件，在读到要求的字节数之前，就到达了文件尾端
  - 当从终端设备读时，通常一次最多读一行
  - 当从网络读时，网络中的缓冲机构可能造成read函数返回值小于所要求读出的字节数
  - 某些面向记录的设备，如磁带，一次最多返回一个记录
  - .....

文件操作基本顺序

- 打开　　open
- 创建　　creat
- 定位　　lseek
- 读　　read
- 写　　write
- 关闭　　close

write函数

- 用于向文件里面，写入数据
- 函数原型
  - ssize_t write(int fd, const void *buff,
    size_t nbytes);
- 参数
  - 第一个参数fd：文件描述符
  - 第二个参数buff：指向缓冲区，存放了需要写入文件的数据
  - 第三个参数nbytes：需要写入文件的字节数

write函数

- 返回值
  - 返回值类型：ssize_t，即int
  - 出错：返回-1
  - 成功：返回实际写入文件的字节数
- write出错的原因
  - 磁盘满
  - 没有访问权限
  - 超过了给定进程的文件长度限制
  - ……

文件操作基本顺序

- 打开　　open
- 创建　　creat
- 定位　　lseek
- 读　　read
- 写　　write
- 关闭　　close

close函数

- 用于关闭一个已打开的文件
- 函数原型
  - int close(int filedes)
- 返回值
  - 成功返回0
  - 出错返回-1
- 参数
  - filedes：文件描述符

close函数

- 当close函数关闭文件时，会释放进程加在该文件上的所有记录锁
- 内核会对进程打开文件表、文件对象、索引节点表项等结构进行修改，释放相关的资源
- 当进程退出时，会关闭当前所有已打开的文件描述符

文件删除—unlink函数

- #include <unistd.h>
- int res=unlink(char*path)
- 返回值
  - 成功返回0
  - 出错返回-1
- 参数
  - path：要删除文件的路径

文件描述符属性控制--fcntl函数

- #include <fcntl.h>
- #include <unistd.h>
- #include <sys/types.h>
- int result=fcntl(int fd,int cmd);
- int result=fcntl(int fd,int cmd,long arg,...);

cmd	含义
F_DUPFD	复制文件描述符
F_GETFD	获得文件描述符
F_SETFD	设置文件描述符
F_GETFL	获取文件描述符当前模式
F_SETFL	设置文件描述符当前模式
F_GETOWN	获得异步I/O所有权
F_SETOWN	设置异步I/O所有权
F_GETLK	获得记录锁
F_SETLK	设置记录锁
F_SETLKW	设置记录锁

fcntl的主要常见用法：

(1)增加文件的某个flags，比如文件是阻塞的，想设置成非阻塞：

- flags = fcntl(fd,F_GETFL,0); //首先获取文件描述符属性
- flags |= O_NONBLOCK; //修改文件描述符属性，设置为非阻塞模式
- fcntl(fd,F_SETFL,flags); //设置文件描述符属性

fcntl的主要常见用法：

(2)取消文件的某个flags，比如文件是追加模式的，想设置成为非追加模式：

- flags = fcntl(fd,F_GETFL,0);
- flags &= ~O_APPEND; //对追加模式取非表示取消追加模式。
- fcntl(fd,F_SETFL,flags);

fcntl的文件状态标志

- O_RDONLY , O_WRONLY , O_RDWR
- O_NONBLOCK
  - 非阻塞I/O，如果read(2)调用没有可读取的数据，或者如果write(2)操作将阻塞，则read或write调用将返回-1和EAGAIN错误
- O_APPEND
  - 强制每次写(write)操作都添加在文件大的末尾，相当于open(2)的O_APPEND标志
- O_DIRECT
  - 最小化或去掉reading和writing的缓存影响。系统将企图避免缓存你的读或写的数据。如果不能够避免缓存，那么它将最小化已经被缓存了的数据造成的影响。如果这个标志用的不够好，将大大的降低性能
- O_ASYNC
  - 当I/O可用的时候，允许SIGIO信号发送到进程组，例如：当有数据可以读的时候

cmd值的F_GETOWN和F_SETOWN

- F_GETOWN　取得当前正在接收SIGIO或者SIGURG信号的进程id或进程组id，进程组id返回的是负值(arg被忽略)
- F_SETOWN　设置将接收SIGIO和SIGURG信号的进程id或进程组id，进程组id通过提供负值的arg来说明(arg绝对值的一个进程组ID)，否则arg将被认为是进程id

cmd值的F_GETLK, F_SETLK或F_SETLKW

- 获得／设置记录锁的功能，成功则返回0，若有错误则返回-1，错误原因存于errno。
- F_GETLK
  - 通过第三个参数arg(一个指向flock的结构体)取得第一个阻塞lock description指向的锁。取得的信息将覆盖传到fcntl()的flock结构的信息。
- F_SETLK
  - 按照指向结构体flock的指针的第三个参数arg所描述的锁的信息设置或者清除一个文件的segment锁。F_SETLK被用来实现共享(或读)锁(F_RDLCK)或独占(写)锁(F_WRLCK)，同样可以去掉这两种锁(F_UNLCK)。如果共享锁或独占锁不能被设置，fcntl()将立即返回EAGAIN
- F_SETLKW
  - 除了共享锁或独占锁被其他的锁阻塞这种情况外，这个命令和F_SETLK是一样的。如果共享锁或独占锁被其他的锁阻塞，进程将等待直到这个请求能够完成。当fcntl()正在等待文件的某个区域的时候捕捉到一个信号，如果这个信号没有被指定SA_RESTART, fcntl将被中断

文件锁

- 保护共享资源的一种机制。
- 建议性锁和强制性锁。
- 建议性锁要求每个上锁文件的进程都要检查是否有锁存在，并且尊重已有的锁。
- 强制性锁是由内核执行的锁，当文件上锁进行写入操作时，内核将阻止其他任何文件对其进行读写操作。
- 强制性锁对性能影响很大
- 记录锁：对文件的某一记录进行上锁

记录锁

- 记录锁又分为读取锁和写入锁。
- 读取锁又称共享锁，能使多个进程都在文件的同一部分建立读取锁。
- 写入锁又称为排斥锁，在任何时刻只能有一个进程在文件的某个部分建立写入锁。

当前加上的锁	申请下列锁能否成功？	
	读取锁	写入锁
无	可	可
读取锁	可	不可
写入锁	不可	不可

共享锁和独占锁

- 当一个**共享锁**被set到一个文件的某段的时候，其他的进程可以set共享锁到这个段或这个段的一部分。
  - 共享锁阻止任何其他进程set独占锁到这段保护区域的任何部分。如果文件描述符没有以读的访问方式打开的话，共享锁的设置请求会失败。
- **独占锁**阻止任何其他的进程在这段保护区域任何位置设置共享锁或独占锁。
  - 如果文件描述符不是以写的访问方式打开的话，独占锁的请求会失败。

不同类型加锁情况下的读写

读写方式 / 当前锁类型	阻塞读	阻塞写	非阻塞读	非阻塞写
读取锁	正常读取数据	阻塞	正常读取数据	返回EAGAIN错误
写入锁	阻塞	阻塞	返回EAGAIN错误	返回EAGAIN错误

flock结构体

```
struct flock{
    short  l_type; // 锁的类型
    short  l_whence; //指定偏移量的起始位置
    off_t  l_start;//从l_whence参数指定位置开始的偏移量(以
                   字节为单位)
    off_t  l_len;// 从指定位置开始连续被锁住的字节数，如果
                   为0表示剩余的所有内容上锁
    pid_t  l_pid; //返回在指定位置拥有一个锁的进程ID
}
```

flock结构体

fcntl的命令参数为SETLK时，字段含义如下：

F_RDLCK：请求读锁

F_WRLCK：请求写锁

F_UNLCK： 请求移除该锁

fcntl的命令参数为GETLK时：

F_RDLCK：已存在冲突读锁

F_WRLCK：已存在冲突写锁

F_UNLCK：不存在冲突锁

flock结构体

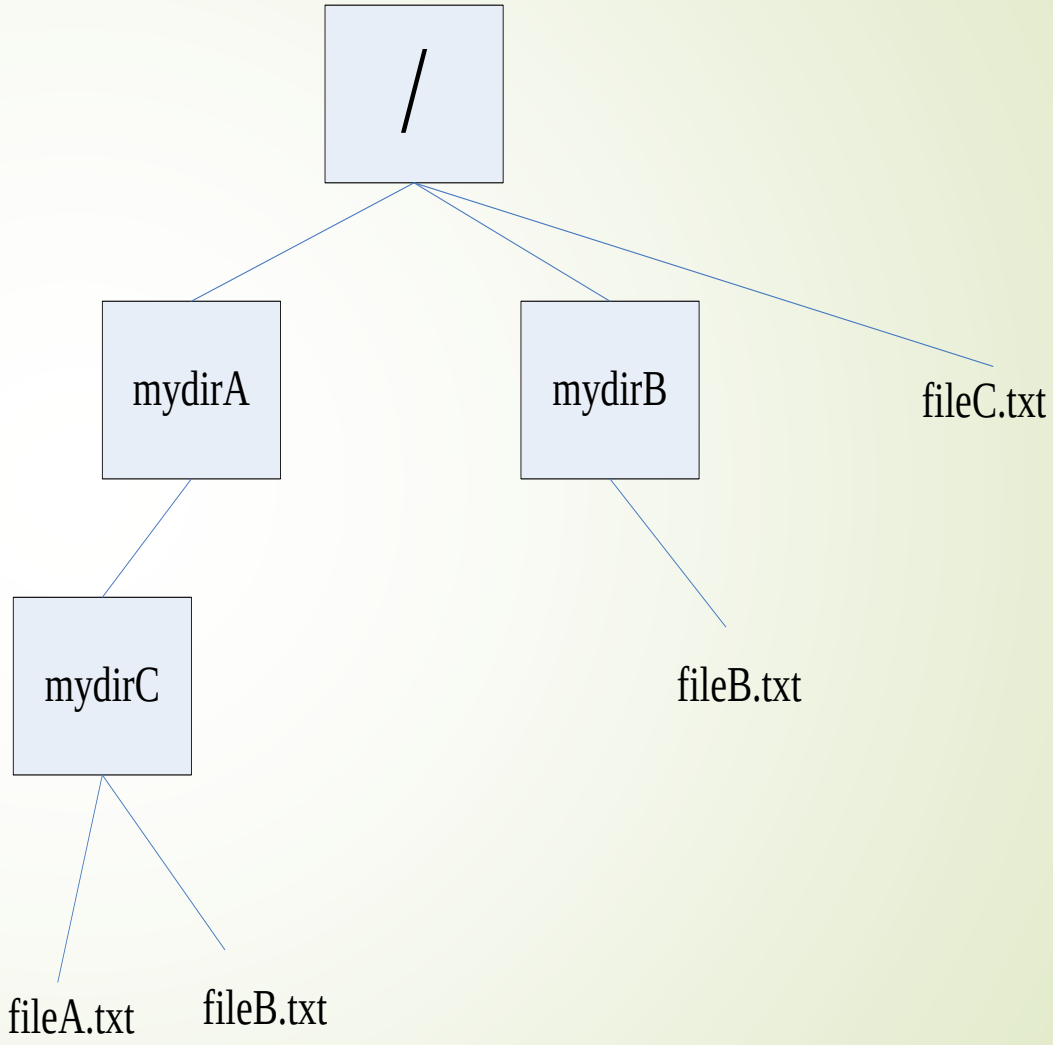
- l_whence指定偏移量的起始位置，类型包括：
  - SEEK_SET，偏移量从文件头部开始
  - SEEK_CUR，偏移量从当前位置开始
  - SEEK_END，偏移量从文件尾部开始

主要内容

- 4.1 概述
- 4.2 文件操作
- 4.3 目录
- 4.4 文件与目录属性
- 4.5 标准文件I/O
- 4.6 处理系统调用中的错误

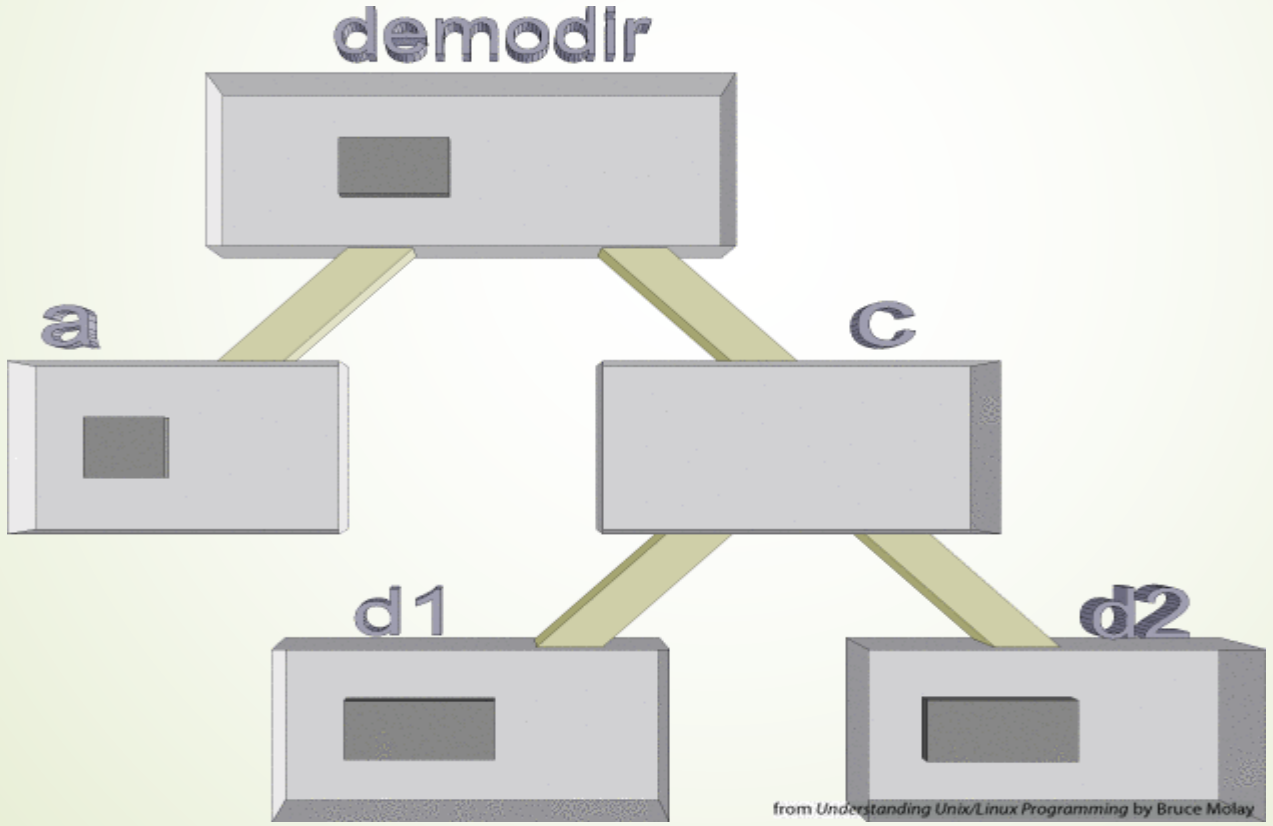
目录

- 目录本身也是一种文件
- 所包含的文件或者目录信息的目录项而不是具体的文件中的数据
- 目录项将文件名与其在磁盘上的物理位置关联起来
- 同一目录下不能有同名文件，而不同目录下可以有同名文件。



理解目录

- 用户角度所看到的目录结构



目录树中所有文件的i节点号

- ls -iaR

demodir:
172085 . 131213 .. 172090 a 172086 c 132942 y

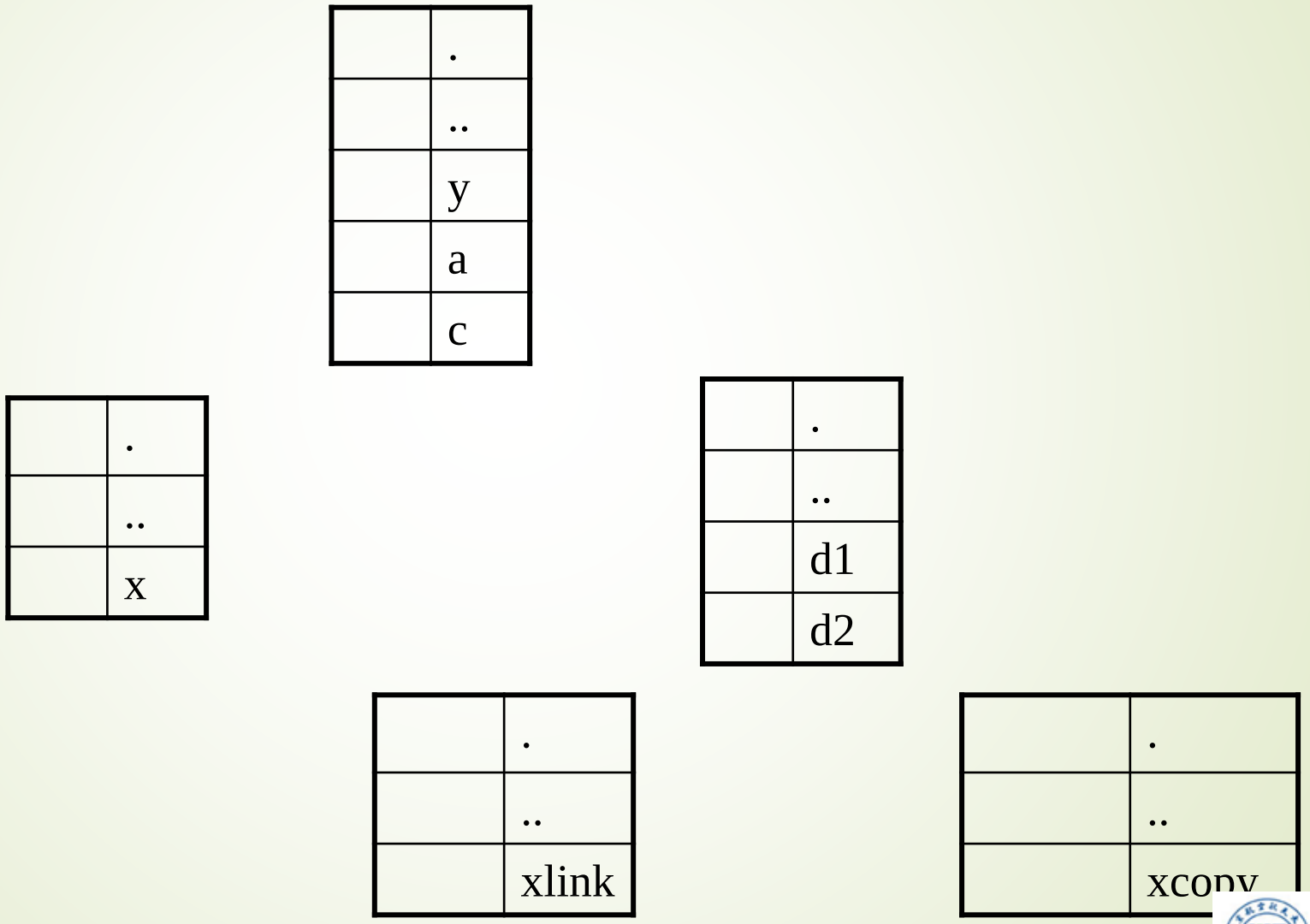
demodir/a:
172090 . 172085 .. 131256 x

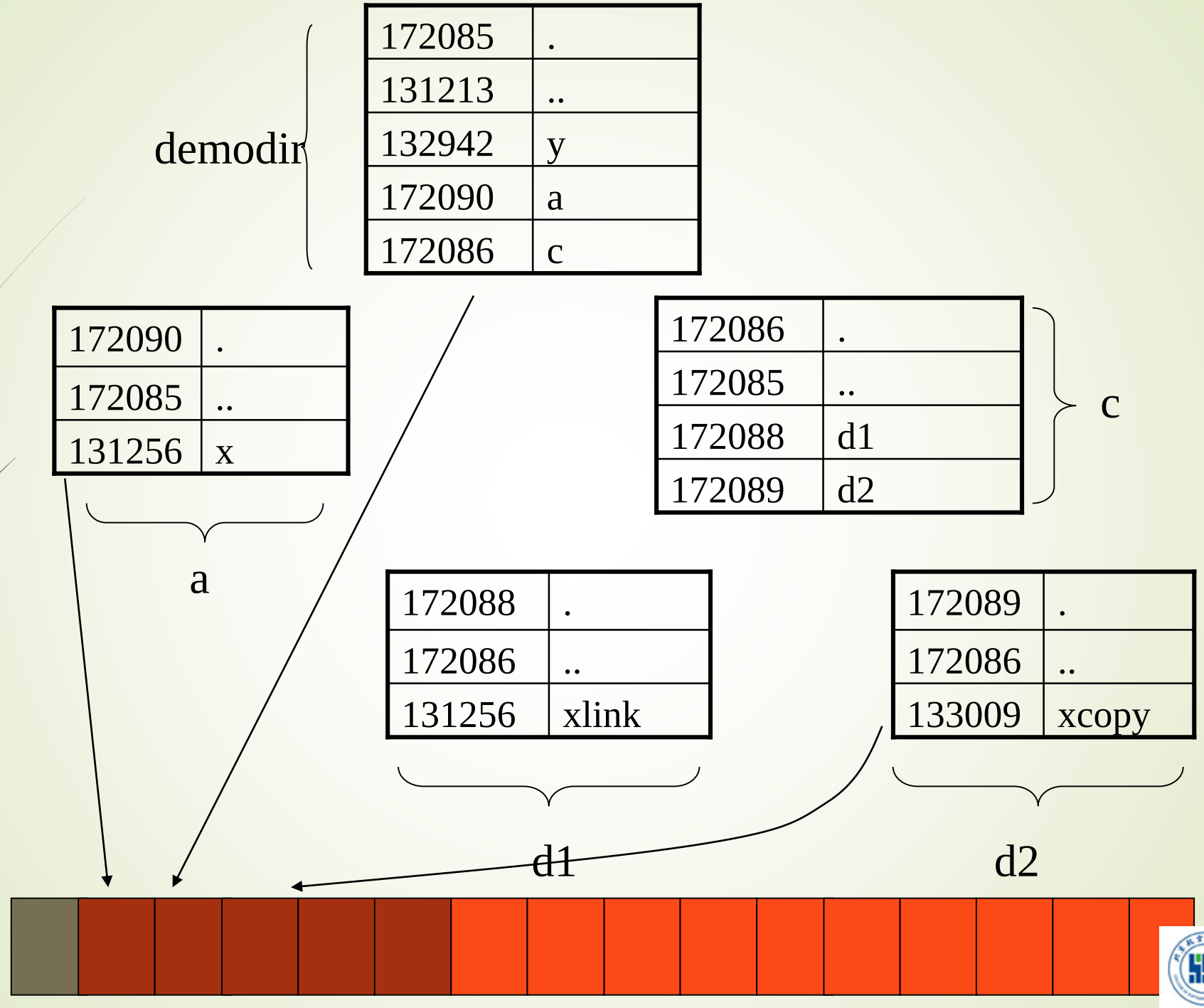
demodir/c:
172086 . 172085 .. 172088 d1 172089 d2 132943 s

demodir/c/d1:
172088 . 172086 .. 131256 xlink

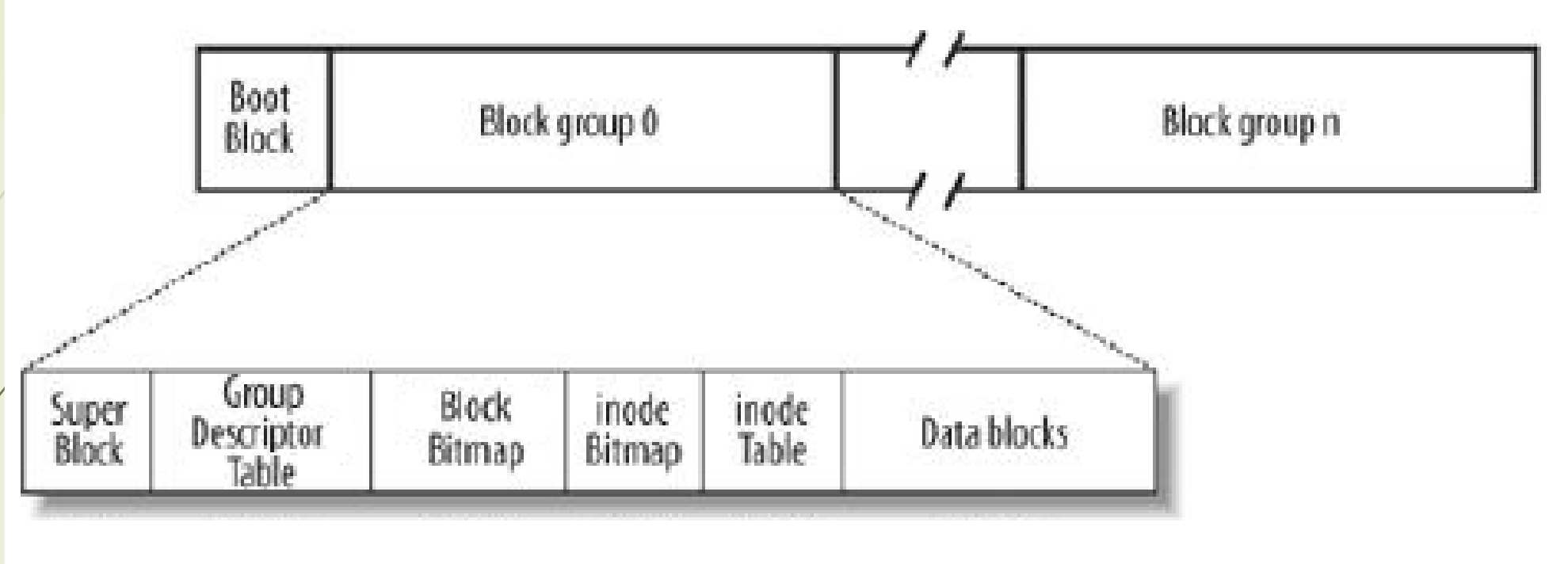
demodir/c/d2:
172089 . 172086 .. 133009 xcopy

系统内部的目录结构





ext2文件结构（了解可选）



- 超级块(Super Block)：每个块组中的第一个数据块，这个块存放整个文件系统本身的信息，包括inode 数、块数、空闲块数、空闲inode 数、第一个数据块位置、块长度等信息

ext2文件结构（了解可选）

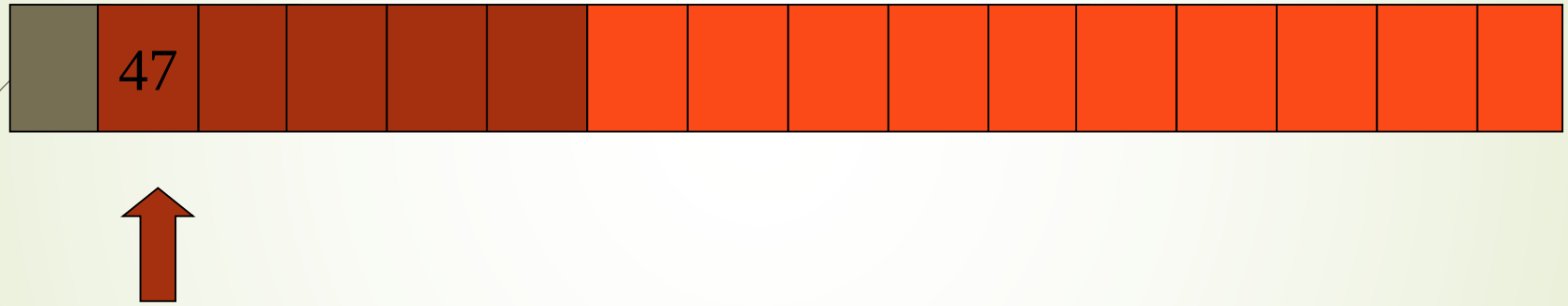
- 组描述符表：存储块组描述符的结构
- 块位图：记录本组数据块中的使用情况，每一块对应一个bit
- inode位图：它记录inode表中inode 的使用情况
- inode表：inode 表保存了本组所有的inode，inode 用于描述文件的属性，一个inode 对应一个文件或目录，有一个唯一的inode 号，并记录了文件在磁盘的存储位置(或者块号)、存取权限、修改时间、类型、链接数等信息。
- 数据块：对于普通文件，数据块存储文件中的数据，对于目录文件，数据块存储该目录下子目录或者文件的名称以及对应的inode信息。

创建文件的过程（了解可选）

- 存储属性
- 存储数据
- 记录分配情况
- 添加文件名到目录

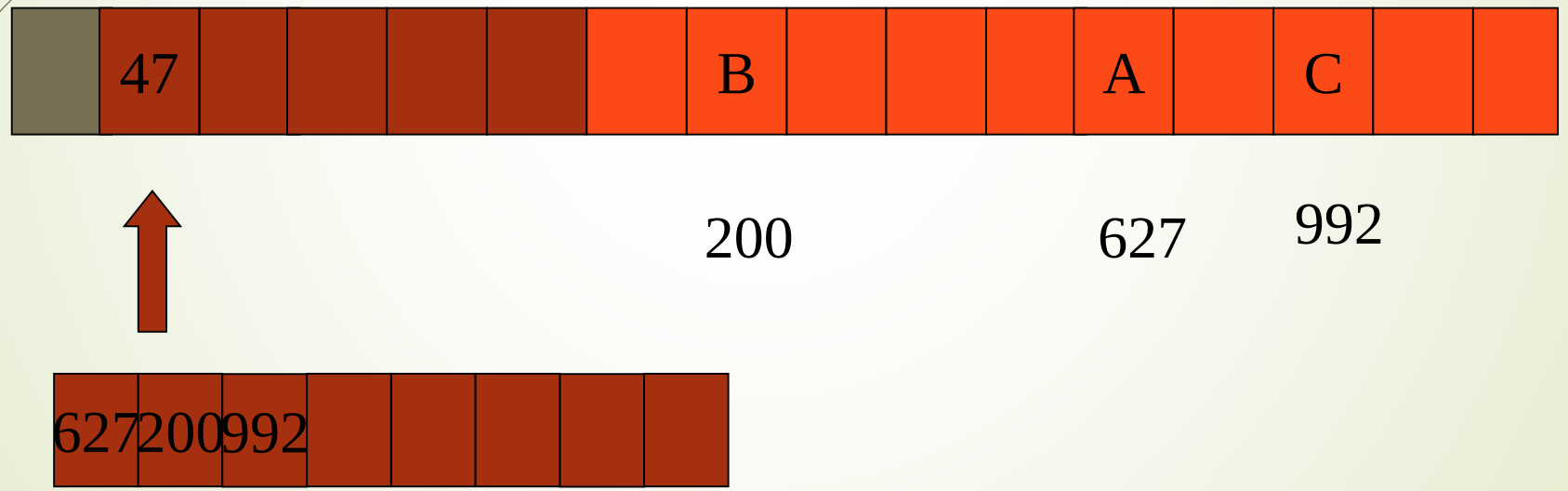
存储属性

- 内核先找到一个空的i节点，内核将文件信息存储在其中



存储数据及记录分配情况（了解可选）

- 假设文件需要3个数据块存储
- 内核从未使用的块列表中找出3个自由块
- 这3个块分别为： 627、 200、992



将文件所使用块记录到i节点中

添加文件名到目录（了解可选））

- 内核将入口(47 文件名)存入到当前的目录文件中
- 文件名与i-node节点号对应起来

47	userlist

节点号	文件名称
123	
833	
4004	Hello.c

目录

与目录相关的系统调用

- (1)目录的打开、读取以及关闭
- #include <sys/types.>
- #include <dirent.h>
- DIR *opendir(const char *dir_name); /*打开目录，返回一个指向DIR的指针，从而创建一个到目录的连接*/

- #include <sys/types.h>
- #include <dirent.h>
- int closedir(DIR *dirp);

目录项的读取

- readdir的用法如下：

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

```
struct dirent  * readdir(DIR *dir); /*每次从DIR中读取目录项信息，该目录信息保存在结构体dirent中*/
```

```
struct dirent{
    char d_name[1];            /* 文件名称 */
    int d_fileno;       /*文件的inode号*/
 };
```

目录的定位

```
#include <dirent.h>
void seekdir(DIR *dir, off_t offset);
```

设置下一个 readdir() 调用的位置

```
#include <dirent.h>
off_t telldir(DIR *dir);
```

返回目录流中的当前位置

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dir);
```

重置目录流

目录的创建/删除/改变/获得

```
#include <sys/stat.h>
#include <sys/types.h>
int res=mkdir(char*pathname,mode_t mode)
#include <unistd.h>
int res=rmdir(char*pathname)
#include <unistd.h>
int res=chdir(const char* path)
char *getcwd(char *buf, size_t size);
```

目录或者文件重命名

#include <unistd.h>

int res=rename(const char*from,const char *to)

例如 rename(“y”,”y.old”)改变文件名称;

rename(“y”,”c/d2/y.old”)改变文件的名字和位置

rename并不移动文件数据本身而本质只是将链接移动到另外一个目录

demodir

172085	.
131213	..
132942	y
172090	a
172086	c

172090	.
172085	..
131256	x

a

172086	.
172085	..
172088	d1
172089	d2

c

172088	.
172086	..
131256	xlink

d1

172089	.
172086	..
133009	xcopy

d2

132942	y.old

rename之前

rename(“y”,”c/d1/y.old”)之后

主要内容

- 4.1 概述
- 4.2 文件操作
- 4.3 目录
- 4.4 文件与目录属性
- 4.5 标准文件I/O
- 4.6 处理系统调用中的错误

4.4 文件与目录的属性

文件与目录的属性存储于其inode中，

Stat函数（2） 获得该属性信息

```
#include <sys/stat.h>
int result=stat(char *fname, struct stat *bufp)
int lstat(const char *restrict path, struct stat *restrict buf);
```

与stat功能相同，区别当符号链接时，返回链接本身

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

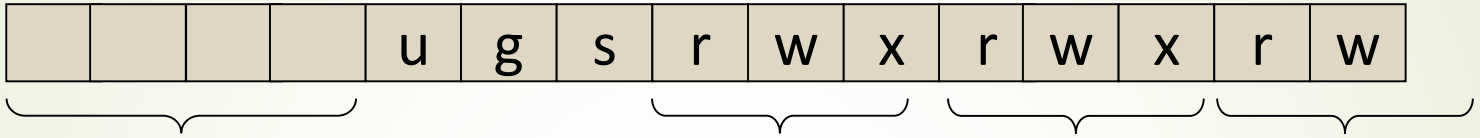
与stat功能相同，由fd指定。

stat结构体

```
struct stat {
    dev_t     st_dev     /*包含该文件的设备ID号*/
    ino_t     st_ino     /*文件的inode*/file serial number
    mode_t    st_mode    /*文件类型及权限模式*/
    nlink_t   st_nlink   /*该文件的链接数*/number of links
    uid_t     st_uid     /*文件所有者的用户ID*/user ID of file
    gid_t     st_gid     /*文件的组ID*/group ID of file
    dev_t     st_rdev    /*如果文件为字符或者块设备时的设备ID*/
    off_t     st_size    /*若文件为普通文件，文件的字节数*/
    time_t    st_atime   /*最近的访问时间*/
    time_t    st_mtime   /*最近数据修改时间*/
    time_t    st_ctime   /*最近文件状态改变的时间*/
}
```

文件/目录的模式

模式是一个长度为16位的二进制数



文本类型　　用户权限　组权限　其它用户权限

最高四位二进制	文件类型常量 (八进制)	文件类型
0100	S_IFDIR 0040000	目录文件
0010	S_IFCHR 0020000	字符设备文件
0110	S_IFBLK 0060000	块设备文件
1000	S_IFREG 0100000	普通文件
1010	S_IFLNK 0120000	符号链接文件
1100	S_IFSOCK 0140000	Socket文件
0001	S_IFIFO 0010000	命名管道文件

判断文件类型

- 文件类型掩码：文件类型掩码将不需要的字段置0，而文件类型部分不变。
- 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
- Linux中使用S_IFMT常量表示
- 文件的属性返回的st_mode值为:100664（八进制），将其与文件类型掩码位于操作：
- st_mode 0 0 1 0 0 0 0 0 0 0 1 1 0 1 1 0 1 0 0
- 掩码 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
- 结果 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

判断文件类型

```
if ( (info.st_mode&0170000) == 0040000)
    printf( “this is a directory” );
if (S_ISDIR(info.st_mode) )
    printf( “this is a directory” );
if ( S_ISREG(info.st_mode) )
    printf(“this is a regular file”);
if ( S_ISCHR(info.st_mode) )
    printf(“this is a character device file”);
……
```

文件权限掩码

```
#define S_IRUSR   0000400       /* 文件所有者读权限 */
#define S_IWUSR   0000200       /* 文件所有者写权限*/
#define S_IXUSR   0000100       /*文件所有者执行权限  */
#define S_IRGRP  (S_IRUSR >> 3)  /* 组用户读权限  */
#define S_IWGRP  (S_IWUSR >> 3)  /*组用户写权限  */
#define S_IXGRP  (S_IXUSR >> 3)  /*组用户执行权限  */
```

文件所有者的读权限的二进制bit为：

0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0

文件链接

- 目录文件中存储的是文件名称及所对应的inode。它被称为链接
  - 硬链接，不同的文件名可以对应同一个inode。
  - 符号链接，有自己的inode
- 文件都有一个属性：链接数
  - 链接数是指向该文件inode的文件数。
- ln命令可建立硬链接　例如：ln Hello.c myHello.bak 命令将建立新文件myHello.bak，它链接到源文件Hello.c的inode。
- ln –s命令建立符号链接

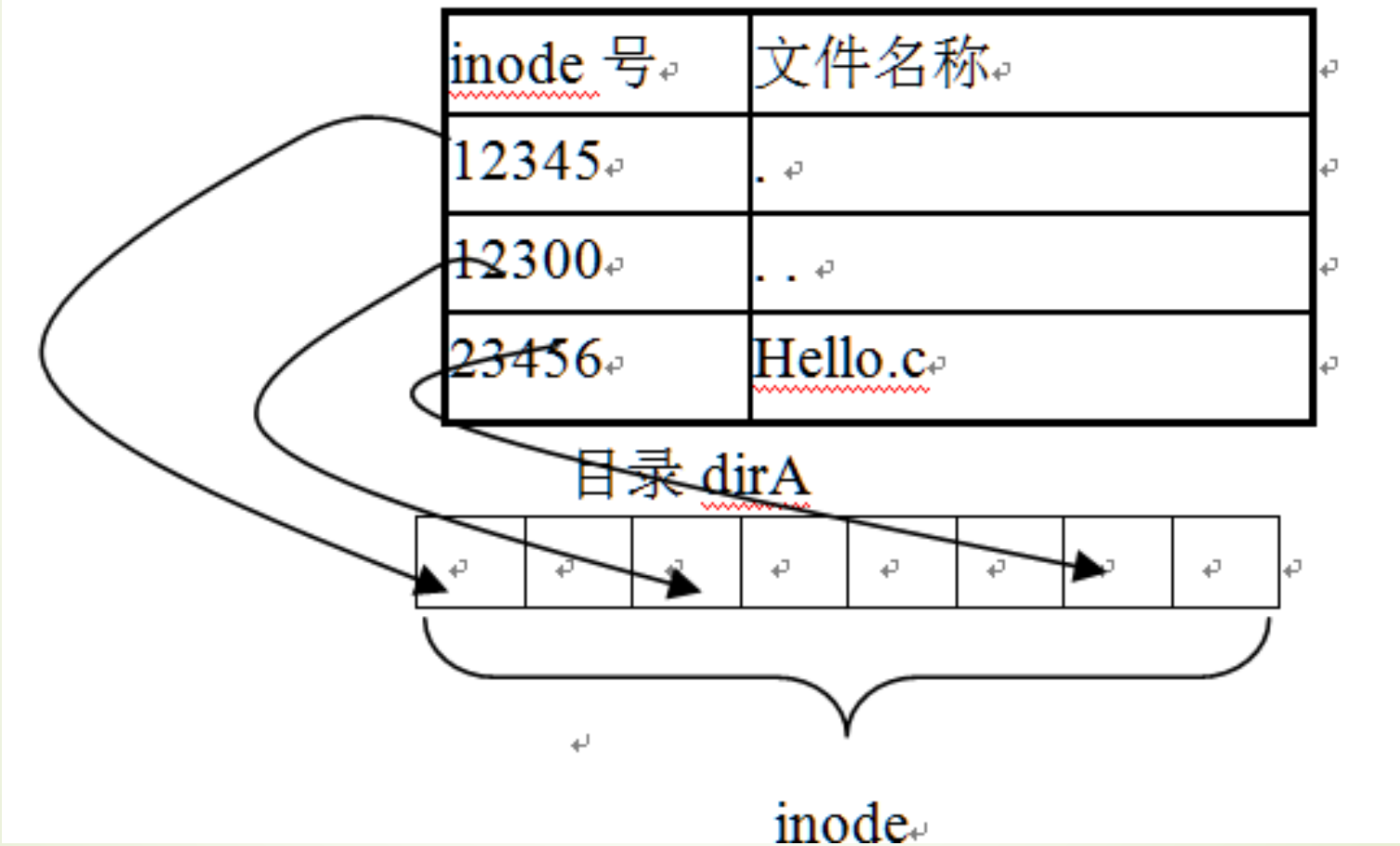
建立硬链接

```
[cosmos@localhost book]$ ls -li Hello.c  /*查看文件Hello.c的inode*/
133369 -rw-rw-r--. 1 cosmos cosmos 9 04-11 05:48 Hello.c

[cosmos@localhost book]$ ln Hello.c myHello.bak

[cosmos@localhost book]$ ls -li Hello.c myHello.bak
133369 -rw-rw-r--. 2 cosmos cosmos 9 04-11 05:48 Hello.c
133369 -rw-rw-r--. 2 cosmos cosmos 9 04-11 05:48 myHello.bak
```

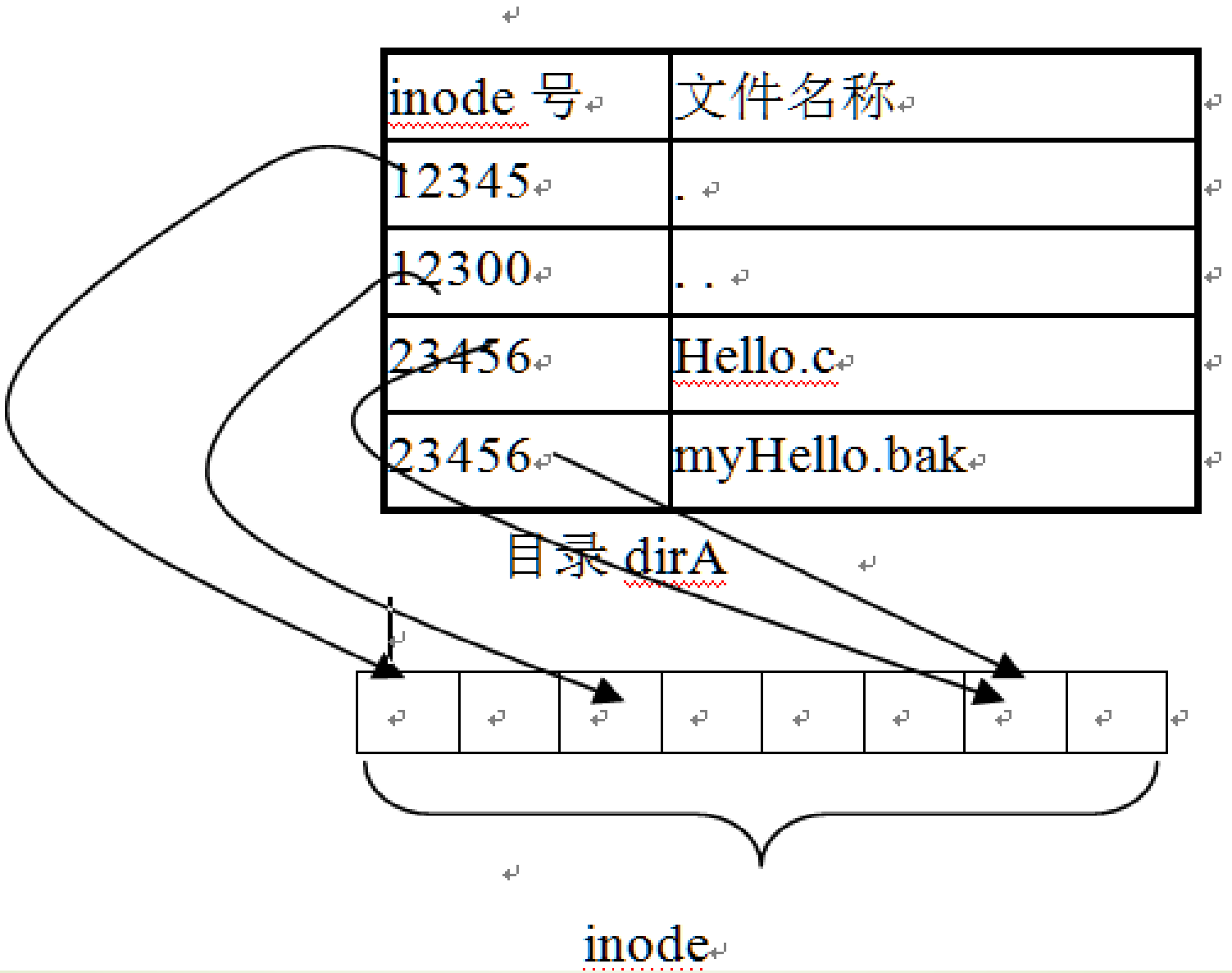
硬链接 删除时，只有减少链接数

```
[cosmos@localhost book]$ diff  Hello.c myHello.bak/*命令查看两个文件内容的差异*/
[cosmos@localhost book]$ cat Hello.c
sfasfasf

[cosmos@localhost book]$ rm -f Hello.c
[cosmos@localhost book]$ cat myHello.bak
sfasfasf
[cosmos@localhost book]$
```

硬链接



硬链接


inode 号
文件名称
12345
.
12300
. .
23456
Hello.c
23456
myHello.bak
目录 dirA
inode

符号链接 定义与创建

- 符号链接有自己的inode，文件属性与源文件不同
  - 是一个新文件
  - 存储的是源文件的路径名称

- ln –s命令创建符号链接文件

```
[cosmos@localhost book]$ ln -s  Hello.c myHellosymbol
```

符号链接 是个独立的文件

[cosmos@localhost book]$ ls -li Hello.c myHellosymbol

133369 -rw-rw-r--. 1 cosmos cosmos 62 04-11 05:39 Hello.c

133293 lrwxrwxrwx. 1 cosmos cosmos 7 04-11 05:40 myHellosymbol -> Hello.c

[cosmos@localhost book]$ vi myHellosymbol

/*在末尾添加数据，例如一行新数据aaaaaaaa*/

[cosmos@localhost book]$ ls -li Hello.c myHellosymbol

133369 -rw-rw-r--. 1 cosmos cosmos ***73*** 04-11 05:43 Hello.c /*源文件内容长度由62变为73*/

133293 lrwxrwxrwx. 1 cosmos cosmos ***7*** 04-11 05:40 myHellosymbol -> Hello.c

/*符号链接文件长度不变*/

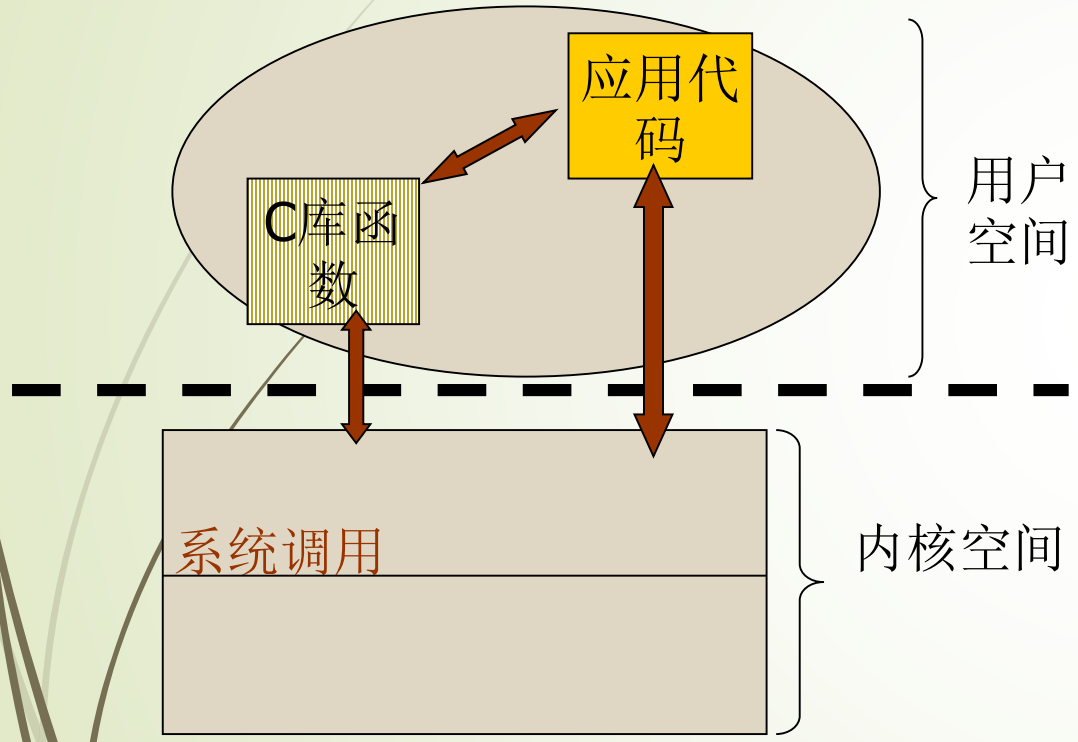
符号链接 引用源文件的内容

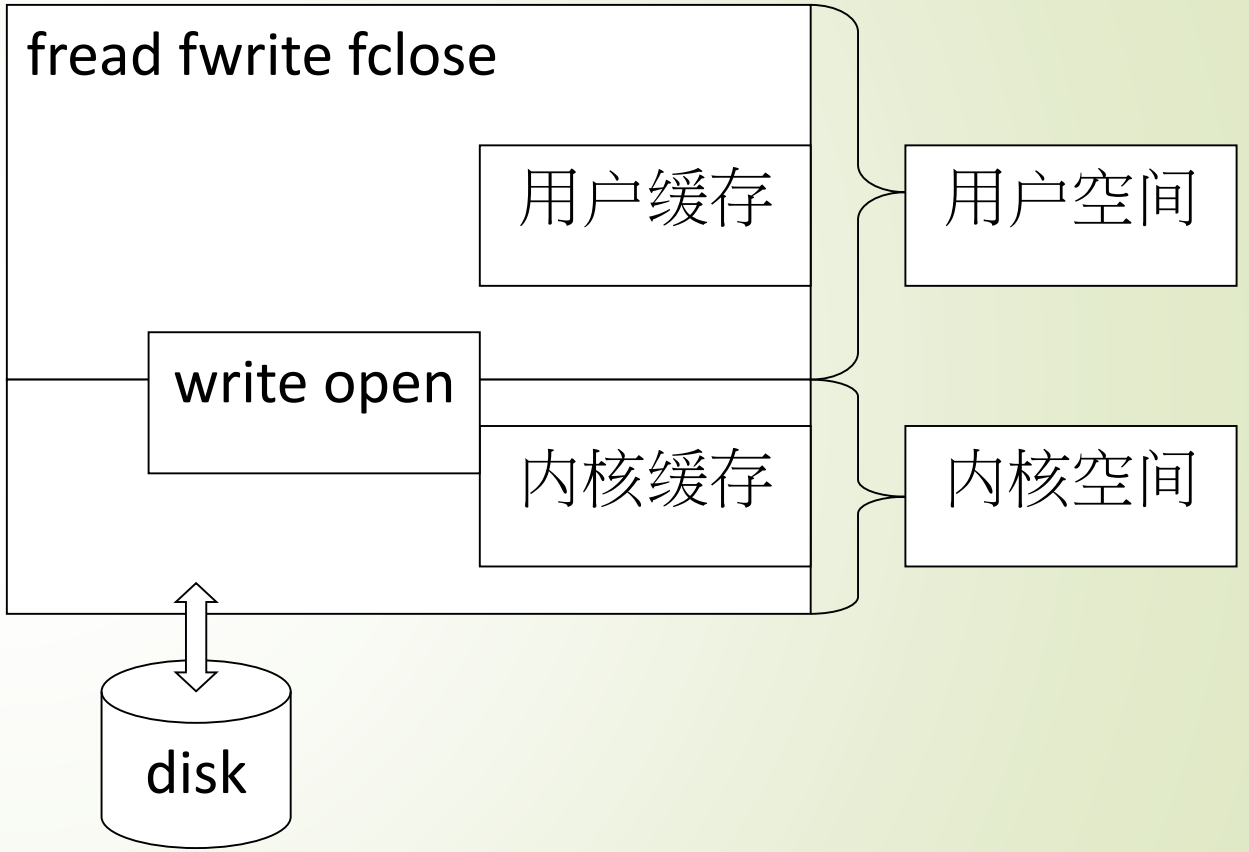
```
[cosmos@localhost book]$ diff Hello.c myHellosymbol
/*比较两个文件内容，结果相同，无差异*/

[cosmos@localhost book]$ rm -f Hello.c

[cosmos@localhost book]$ cat myHellosymbol
cat: myHellosymbol: 没有那个文件或目录
```

主要内容

- 4.1 概述
- 4.2 文件操作
- 4.3 目录
- 4.4 文件与目录属性
- 4.5 标准文件I/O
- 4.6 处理系统调用中的错误

标准文件I/O





文件I/O和标准I/O的区别

- 文件I/O：文件I/O称之为不带缓存的IO（unbuffered I/O)
  - 不带缓存指的是每个read，write都调用内核中的一个系统调用。也就是一般所说的低级I/O——操作系统提供的基本IO服务。
- 标准I/O：标准I/O是ANSI C建立的一个标准I/O模型
  - 是一个标准函数包和stdio.h头文件中的定义，具有一定的可移植性。
- 标准的I/O提供了三种类型的缓存。
  - 全缓存：当填满标准I/O缓存后才进行实际的I/O操作。
  - 行缓存：当输入或输出中遇到新行符时，标准I/O库执行I/O操作。
  - 不带缓存

主要内容

- 4.1 概述
- 4.2 文件操作
- 4.3 目录
- 4.4 文件与目录属性
- 4.5 标准文件I/O
- 4.6 处理系统调用中的错误

处理系统调用中的错误

- 全局变量 errno
- 出错处理
  - 以前的定义：extern int errno;
  - 多线程环境：

```
extern int * __errno_location();

#define errno (*__errno_location())
```

处理系统调用中的错误

头文件：#include<errno.h>

原型：void perror(const char * msg)

- perror函数根据当前的errno，输出一条出错信息
- 该函数输出：

  msg指向的字符串: errno对应的出错信息

第4章 结束