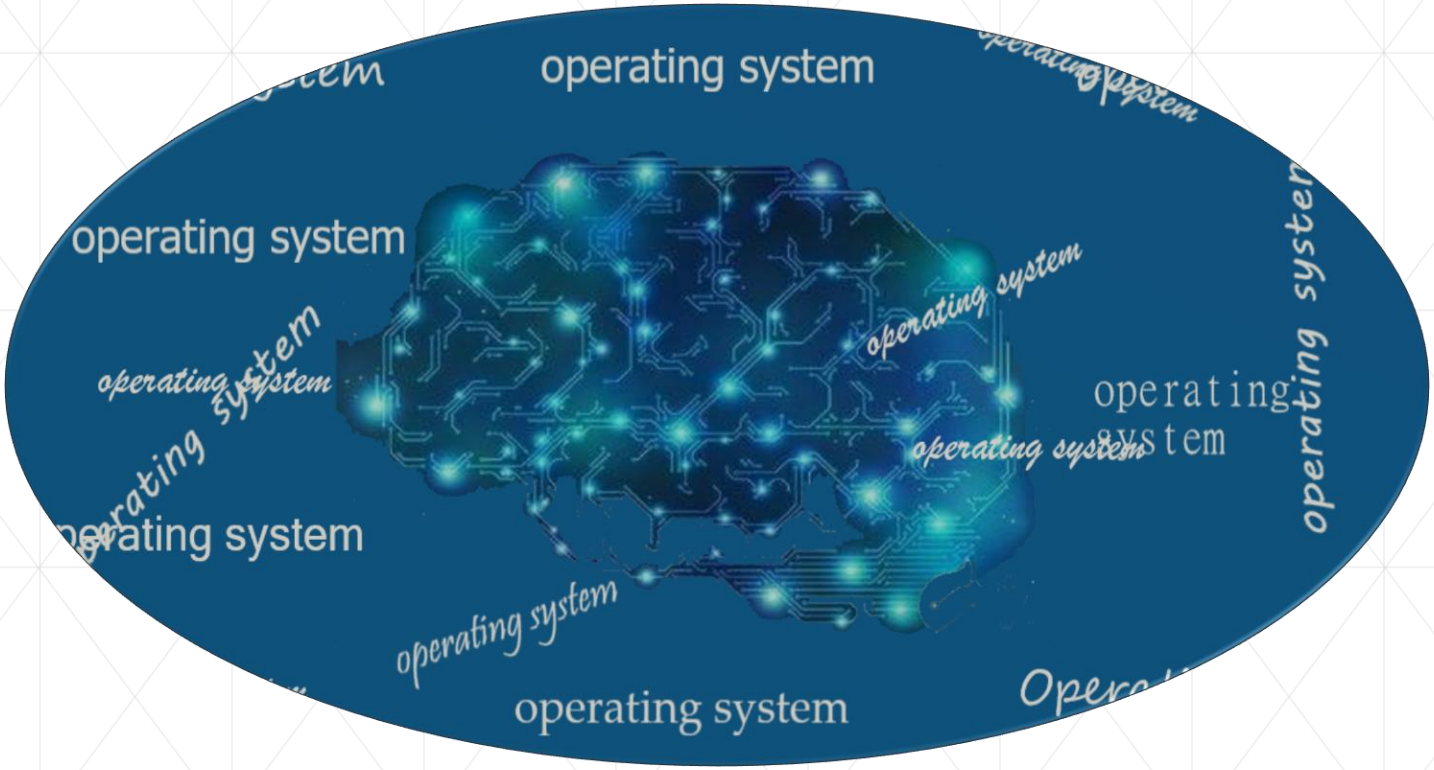


第九章 文件操作与权限管理

授课教师

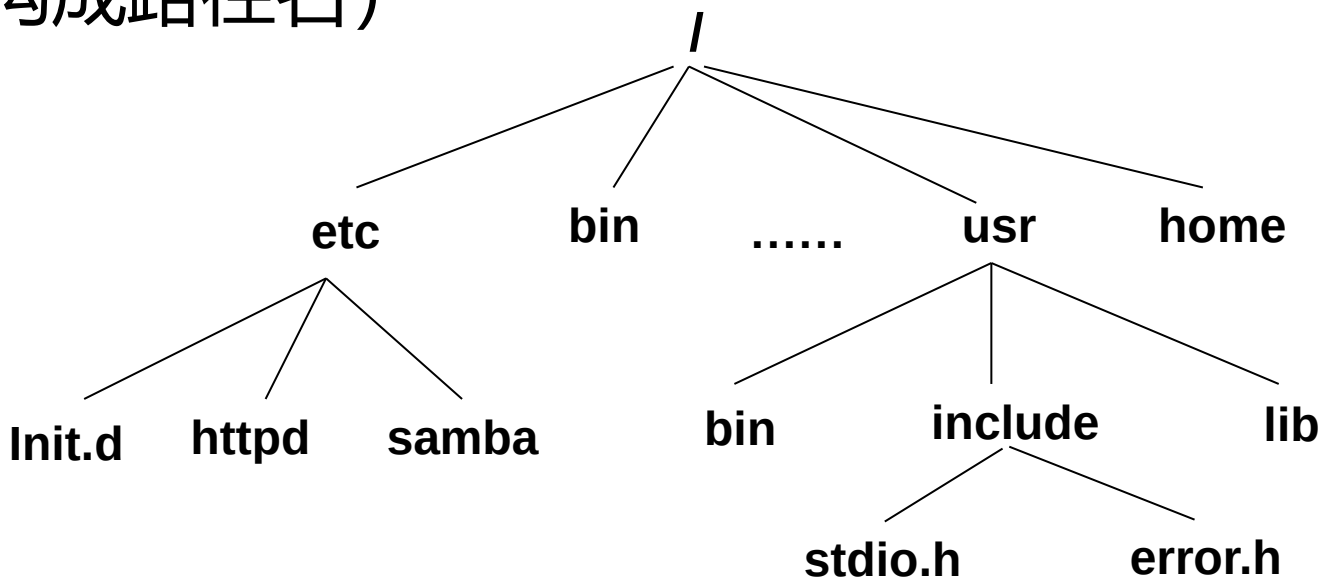
电子邮箱:

主要内容

- UNIX/Linux文件系统的基本结构
- 索引节点的功能
- 文件的分类与权限
- 文件I/O
- 目录操作与文件属性

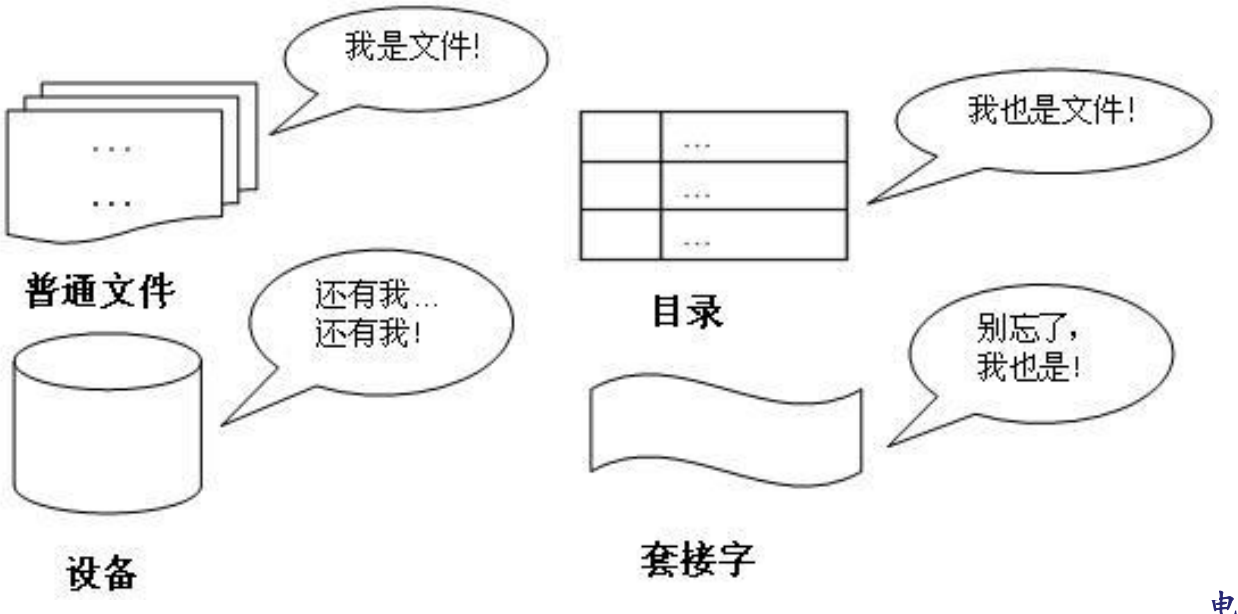
文件的组织形式

- 文件：一组在逻辑上具有完整意义的信息项的系列
- 目录：用来容纳文件，目录可以包含子目录，层层嵌套以形成路径。（以斜线分隔的文件名序列构成路径名）



Unix/Linux文件系统特性

- “一切皆是文件”是 Unix/Linux 的基本哲学之一。普通文件，目录、字符设备、块设备、 套接字等在 Unix/Linux 中都是文件
- 类型不同的文件都是通过相同的API对其进行操作



Unix/Linux文件系统特性

- Unix/Linux 中允许不同的文件系统共存，如 ext2, ext3, vfat 等。

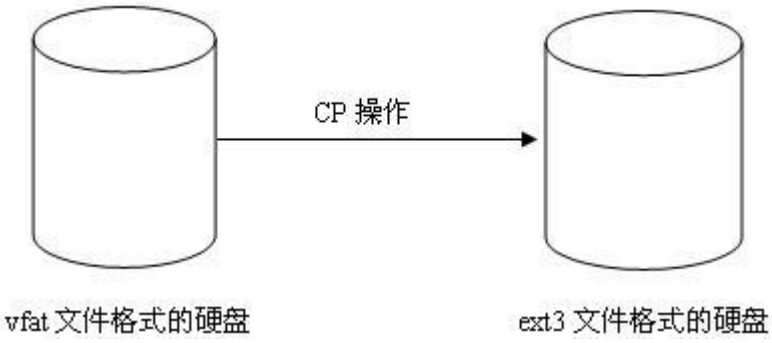
文件系统	适用场景	原因
ext2	U盘	ext2不写日志，对安全性要求不高，兼容FAT
ext3	对稳定性要求高的场景	
ext4	小文件较少	不支持inode动态分配
xfs	小文件多	支持inode动态分配

```
ly@ubuntu:/etc$ findmnt
TARGET                                SOURCE      FSTYPE     OPTIONS
/                                     /dev/sda1   ext4       rw,relatime,errors=remount-ro
├─/sys                                sysfs       sysfs      rw,nosuid,nodev,noexec,relati
│ ├─/sys/kernel/security              securityfs  securit    rw,nosuid,nodev,noexec,relati
│ ├─/sys/fs/cgroup                    tmpfs       tmpfs      ro,nosuid,nodev,noexec,mode=7
│ │ ├─/sys/fs/cgroup/unified          cgroup      cgroup2    rw,nosuid,nodev,noexec,relati
│ │ ├─/sys/fs/cgroup/systemd          cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ │ ├─/sys/fs/cgroup/perf_event       cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ │ ├─/sys/fs/cgroup/net_cls,net_prio cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ │ ├─/sys/fs/cgroup/blkio            cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ │ ├─/sys/fs/cgroup/cpu,cpuacct      cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ │ ├─/sys/fs/cgroup/cpuset           cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ │ ├─/sys/fs/cgroup/freezer          cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ │ ├─/sys/fs/cgroup/devices          cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ │ ├─/sys/fs/cgroup/rdma             cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ │ ├─/sys/fs/cgroup/pids             cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ │ ├─/sys/fs/cgroup/hugetlb          cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ │ └─/sys/fs/cgroup/memory           cgroup      cgroup     rw,nosuid,nodev,noexec,relati
│ ├─/sys/fs/pstore                    pstore      pstore     rw,nosuid,nodev,noexec,relati
│ ├─/sys/kernel/debug                 debugfs     debugfs    rw,relatime
│ ├─/sys/fs/fuse/connections          fusectl     fusectl    rw,relatime
```

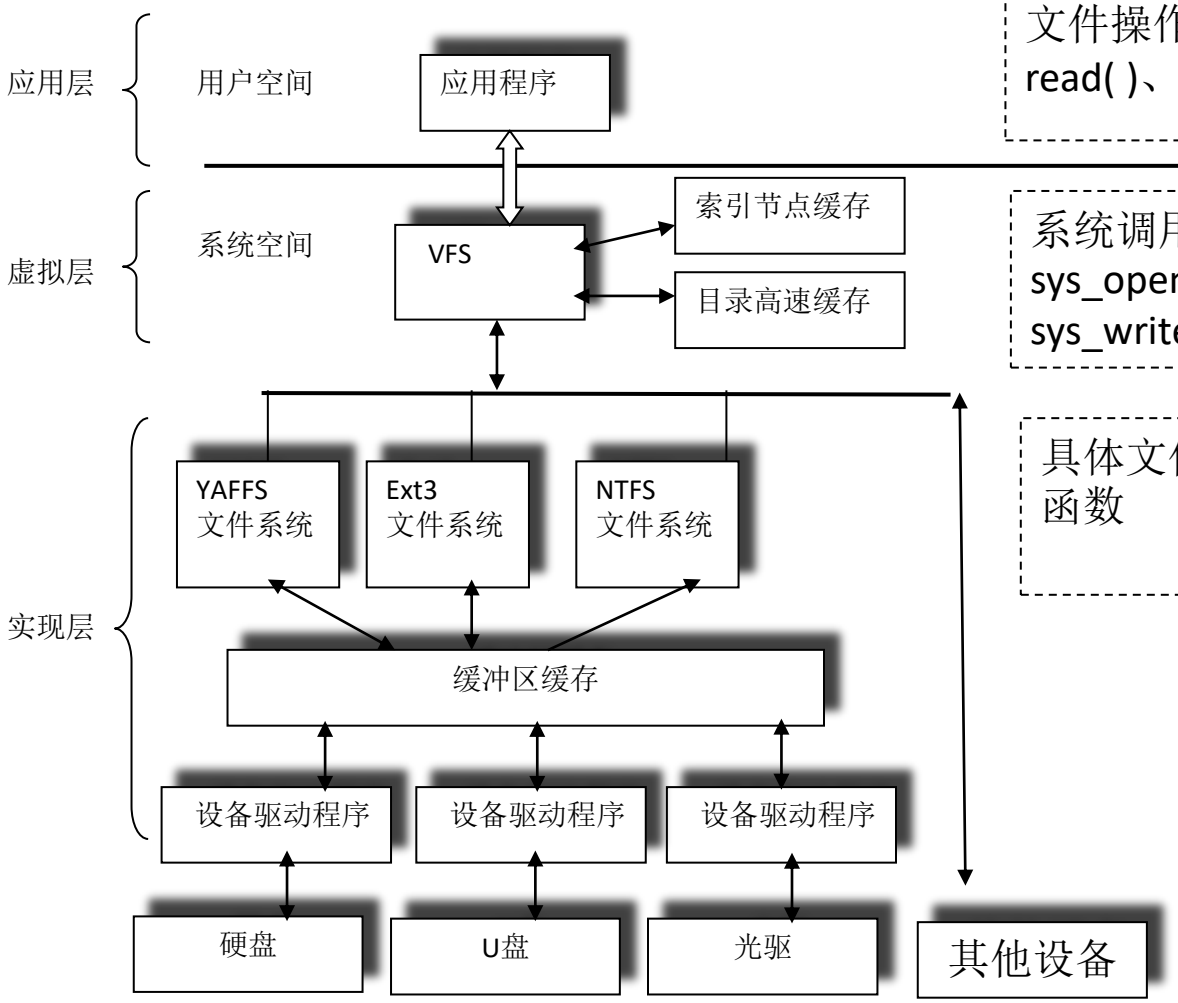
```
ly@ubuntu:~$ df -T
Filesystem     Type      1K-blocks     Used Available Use% Mounted on
udev           devtmpfs     469604        0    469604   0% /dev
tmpfs          tmpfs         98524     1788     96736   2% /run
/dev/sda1      ext4       19478204  6963308  11502416  38% /
tmpfs          tmpfs        492608        0    492608   0% /dev/shm
tmpfs          tmpfs          5120        4      5116   1% /run/lock
tmpfs          tmpfs        492608        0    492608   0% /sys/fs/cgroup
tmpfs          tmpfs         98520       16     98504   1% /run/user/122
tmpfs          tmpfs         98520       32     98488   1% /run/user/1000
ly@ubuntu:~$ lsblk -f
NAME   FSTYPE LABEL UUID                                 MOUNTPOINT
sda
├─sda1 ext4         520fac95-9abf-460c-b6b5-0a7ae147c027 /
├─sda2
└─sda5 swap         05003d6c-f81a-4fb6-8c24-25e7ba73bded [SWAP]
sr0
```

Unix/Linux文件系统特性

- 通过统一的文件操作API/系统调用即可对系统中的任意文件进行操作而无需考虑其所在的具体文件系统格式
- 更进一步，对文件的操作可以跨文件系统而执行：可以使用 cp 命令从 vfat 文件系统格式的硬盘拷贝数据到 ext3 文件系统格式的硬盘；而这样的操作涉及到两个不同的文件系统。



Linux文件系统架构



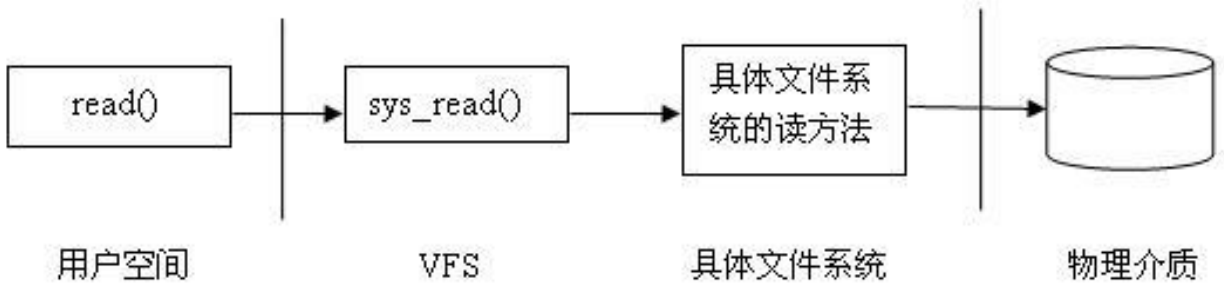
文件操作类系统调用，open( )、read( )、write( )、close( )等

系统调用对应的内核函数，sys_open( )、sys_read( )、sys_write( )、sys_close( )等

具体文件系统对应的文件操作函数

虚拟文件系统

- 虚拟文件系统是Linux 内核中的一个软件层，对内实现文件系统的抽象，允许不同的文件系统共存，对外向应用程序提供统一的文件系统接口
  - **VFS 定义了所有文件系统都支持的基本、抽象接口和数据结构**
  - **实际文件系统实现VFS 定义的抽象接口和数据结构（文件、目录等概念在形式上与VFS的定义保持一致），在统一的接口和数据结构下隐藏了具体的实现细节**
- 虚拟文件系统保证了上述UNIX/Linux文件系统的两点特性



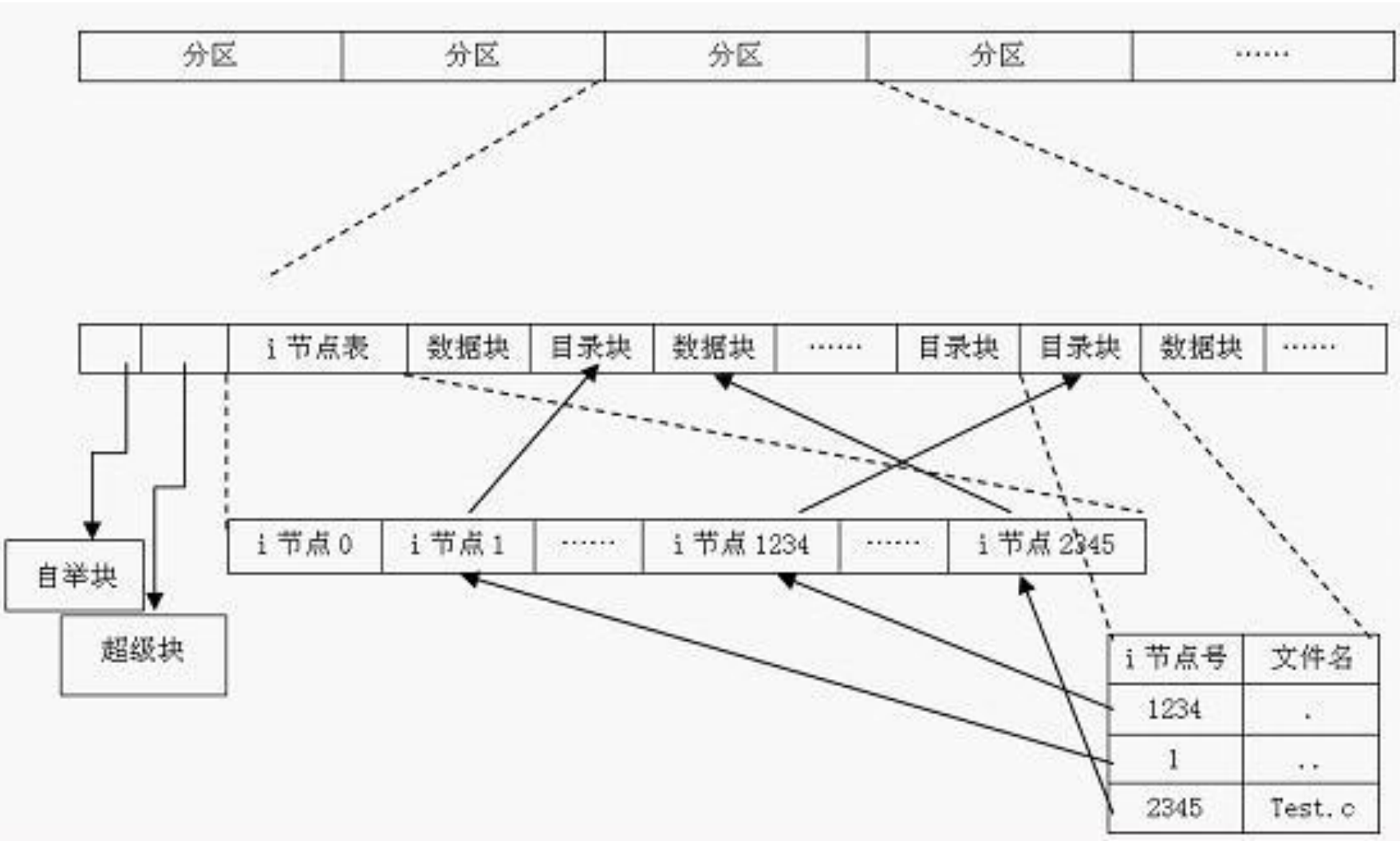
主要内容

- UNIX/Linux文件系统的基本结构
- 索引节点的功能
- 文件的分类与权限
- 文件I/O
- 目录操作与文件属性

文件在磁盘中的表现形式

- 物理表现形式：
  - **超级块**：用于存储文件系统的控制信息的数据结构。描述文件系统的状态、文件系统类型、大小、区块数、索引节点数等，存放于磁盘的特定扇区中。
  - **索引节点（i节点）**：用于存储文件的元数据（文件的基本信息）的一个数据结构，包含诸如文件的大小、拥有者、创建时间、数据块/目录块位置等信息。
  - **目录块**：存放目录文件的内容
  - **数据块**：存放非目录文件的内容

文件在磁盘中的表现形式



目录文件内容

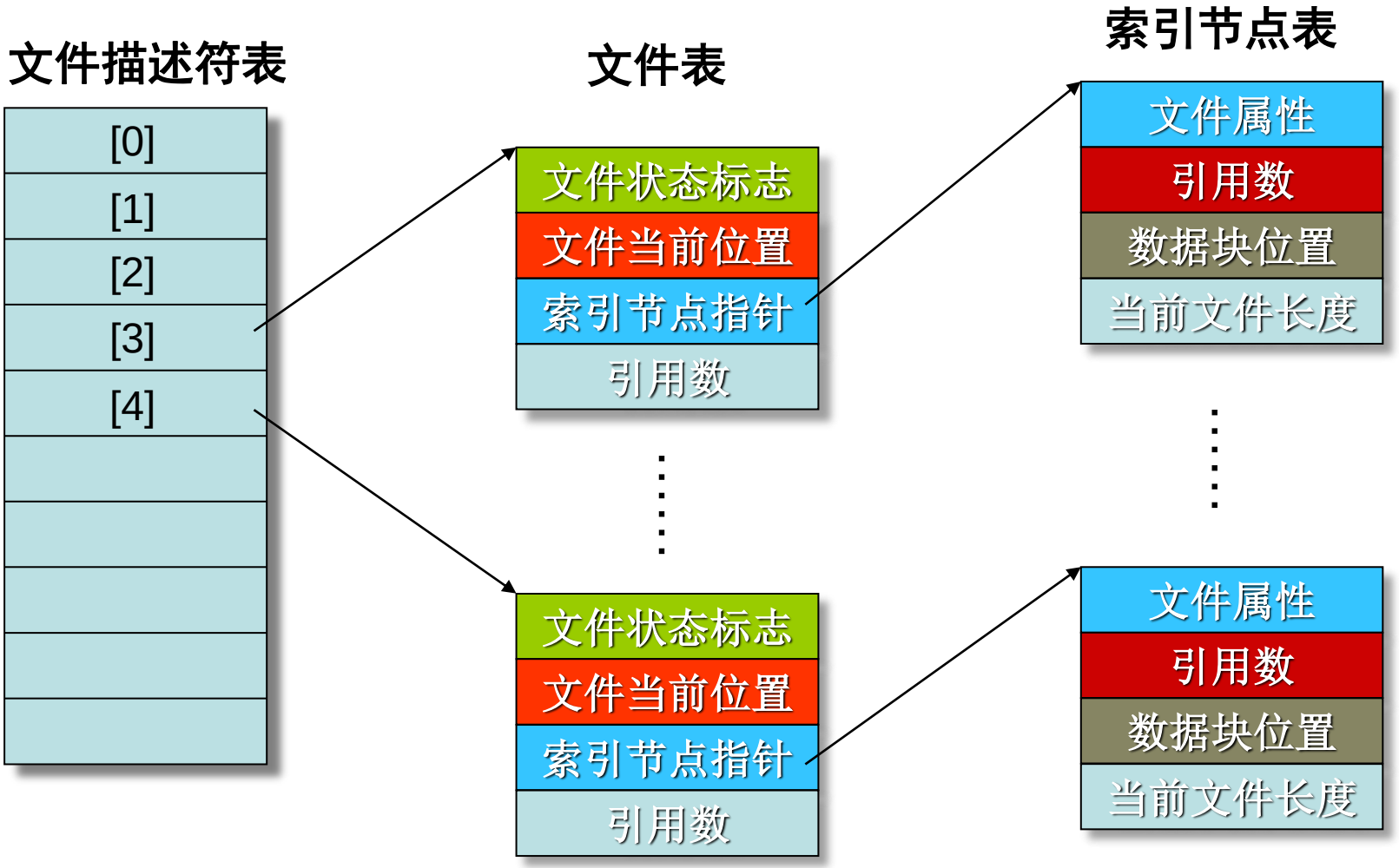
- 一系列目录项（dirent）的列表，每个目录项由两部分组成：
  - **所包含文件的文件名**
  - **文件名对应的索引节点（inode）号**
- ls -i命令列出目录文件内容，即文件名和索引节点号

```
$ ls -i ./code
388776 helloworld.js  389698 imgHash.py   13886 tmp
389031 imgHash.jpg    395507 server.js
$
```

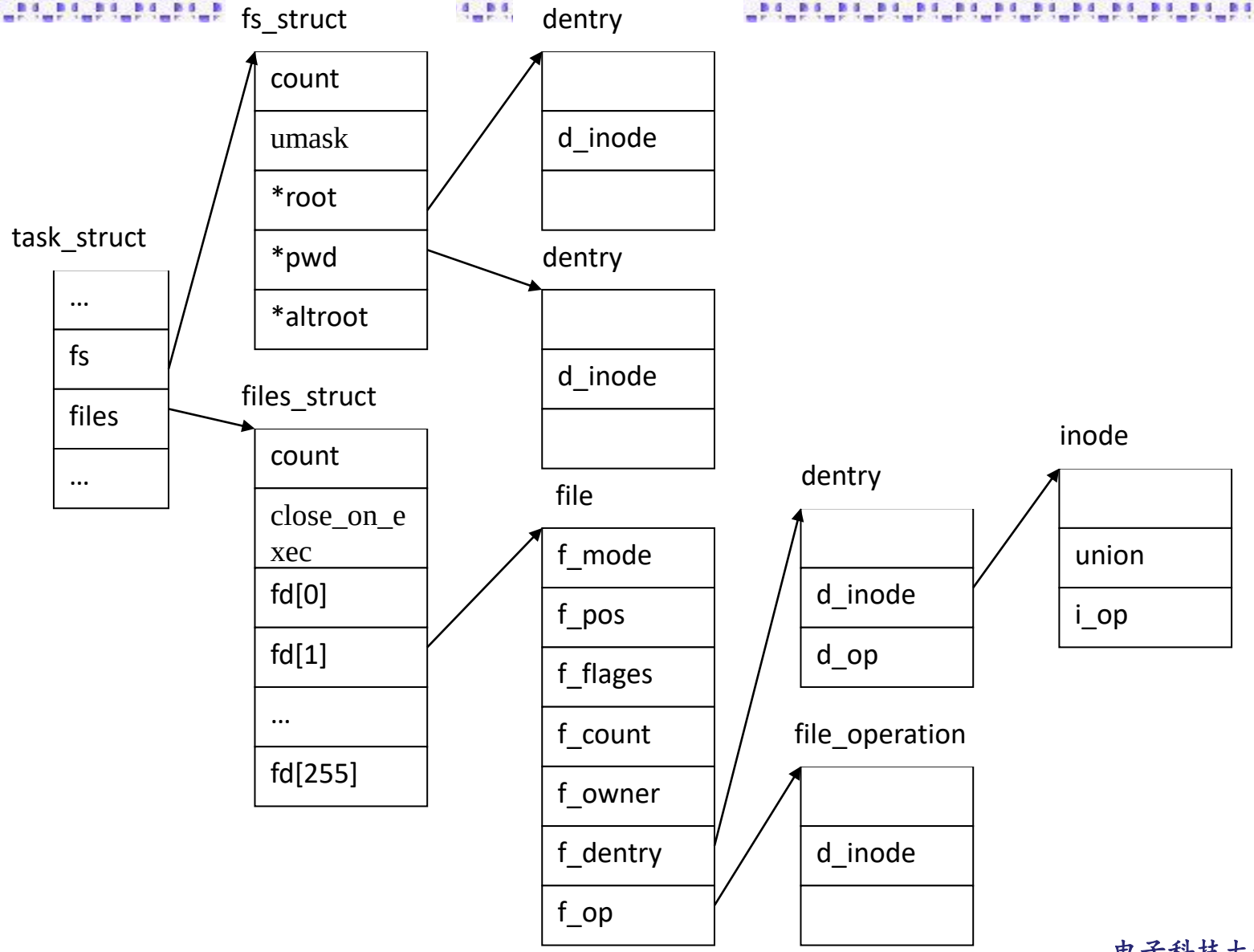
文件在内核中的表现形式

- 内核使用的三种表来表示进程使用的文件
  - 每个进程在PCB中有一个**文件描述符表**，每个描述符表项指向一个文件表
  - 内核为每一个被该进程使用（打开）的文件维护一张**文件表**
  - 每个文件（或设备）都有一个**索引节点**，它包含了文件类型属性及文件数据

文件在内核中的表现形式



Linux内核文件表示



文件描述符表：files_struct结构

- 每个进程用一个files_struct结构来记录其文件使用情况
- 主要结构成员：
  - atomic_t count; 共享该表的进程数
  - int max_fds; 当前文件对象的最大数
  - int max_fdset; 当前文件描述符的最大数
  - int next_fd; 已分配的文件描述符加1
  - struct file ** fd; 指向文件表（文件对象）指针数组
  - fd_set *close_on_exec; 执行exec时需要关闭的文件描述符
  - struct file * fd_array[32]; 文件表（文件对象）指针的初始化数组

文件表：file结构

- 文件表（文件对象）是已打开的文件在内核中的表示，主要用于建立进程和磁盘上的文件的对应关系
- 文件表和物理文件的关系类似进程和程序的关系，一个物理文件可能存在多个对应的文件表（打开多次）
- 一个物理文件对应的索引节点是唯一的

文件表：file结构

- 主要结构成员：
  - struct list_head f_list; 所有打开的文件形成链表
  - struct dentry *f_dentry; 指向对应目录项的指针
  - struct file_operations *f_op; 指向文件操作表的指针
  - mode_t f_mode; 文件的状态标志/打开模式
  - loff_t f_pos; 文件的当前位置
  - unsigned short f_flags; 打开文件时所指定的标志
  - unsigned short f_count; 使用该结构的进程数
  - unsigned int f_uid, f_gid; 用户的UID和GID

文件描述符

- 对于内核而言，所有打开文件都用文件描述符标识
- 文件描述符是一个非负整数。当打开一个现存文件或创建一个新文件时，内核向进程返回一个文件描述符（打开文件的计数）
- 通常情况下，文件描述符0、1、2特指标准输入、标准输出、标准错误（用户程序可直接使用而不需要打开）。它们也可以由常数代替（在头文件中unistd.h定义）：
  - STDIN_FILENO
  - STDOUT_FILENO
  - STDERR_FILENO

目录项对象：dentry结构

- Linux中引入目录项对象的概念主要目的是方便查找文件的索引节点
- 一个路径的各个组成部分，不管是目录还是普通的文件，都是一个目录项对象

目录项对象：dentry结构

- 主要结构成员
  - Atomic_t d_count; 目录项dentry引用计数
  - unsigned int d_flags; dentry状态标志
  - struct inode * d_inode; 与文件关联的索引节点
  - struct dentry * d_parent; 父目录的dentry结构
  - int d_mounted; 目录项的安装点
  - struct qstr d_name; 文件名
  - unsigned long d_time; 重新生效时间
  - struct dentry_operations *d_op; 操作目录项的函数
  - struct super_block *d_sb; 目录项树的根
  - unsigned char d_iname [DNAME_INLINE_LEN] ;文件名前16个字符

索引节点表：inode结构

- 文件系统中的每个物理文件由一个索引节点表（索引节点对象）描述，且只能由一个索引节点对象描述。
- 索引节点指向物理文件的具体存储位置
- 系统通过索引节点来定位每一个文件（文件名可以随时更改，但是索引节点对于物理文件是唯一的，并且随物理文件的存在而存在）
- 索引节点包含了文件的长度、创建及修改时间、权限、所属关系、磁盘中的位置等信息。

索引节点表：inode结构

- **主要结构成员**
  - unsigned long i_ino;　　inode号
  - kdevt idev;　　常规文件所在设备号
  - umode_t i_mode;　　文件类型以及存取权限
  - nlink_t i_nlink;　　连接到该inode的硬连接数
  - uid_t i_uid;　　文件属主的用户ID
  - gid_t i_gid;　　文件属主所在组的ID
  - kdev_t i_rdev;　　特殊文件所在设备号
  - loff_t i_size;　　文件大小（以字节为单位）

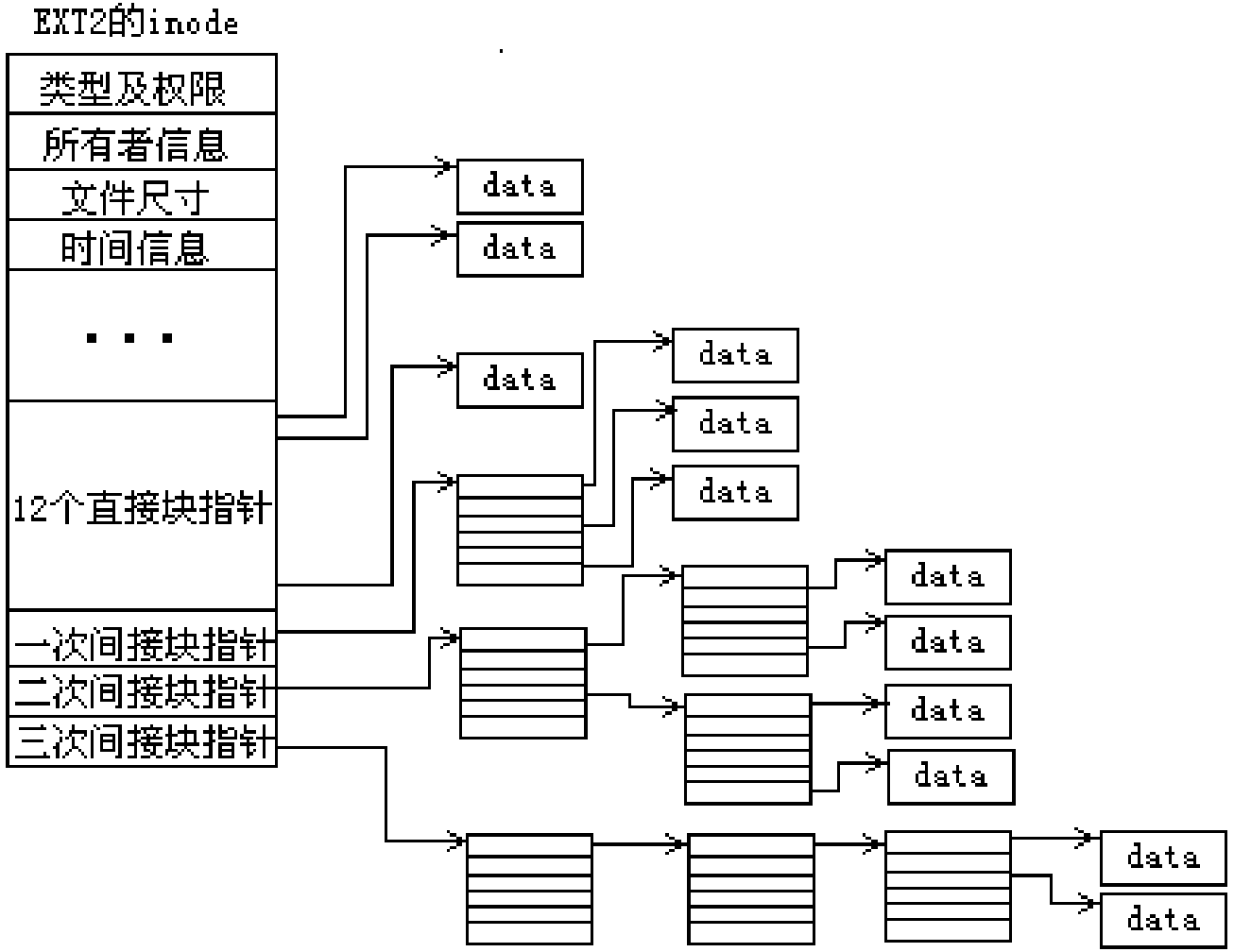
索引节点表：inode结构

- 主要结构成员（续前页）
  - struct inode_operations *i_op; 指向inode进行操作的函数指针
  - struct super_block *i_sb; 指向该文件系统超级块的指针
  - atomic_t i_count; 当前使用该inode的引用计数
  - union

```
{
    struct minix_inode_info minix_i;
    struct Ext2_inode_info Ext2_i;
    struct hpfs_inode_info hpfs_i;
   …
} u;
```

联合体成员指向具体文件系统的inode结构

EXT2文件系统inode结构



EXT2的inode中物理块指针示意图

主要内容

- UNIX/Linux文件系统的基本结构
- 索引节点的功能
- 文件的分类与权限
- 文件I/O
- 目录操作与文件属性

文件类型与文件访问权限

文件类型

标识	文件类型
-	普通文件
d	目录文件
c	字符设备文件
b	块设备文件
p	管道或FIFO
l	符号链接
s	套接字

文件访问权限

标识	文件访问权限
r	读权限
w	写权限
x	执行权限

文件权限

- **读取权限：**
  - 浏览文件/目录中内容的权限；
- **写入权限：**
  - 对文件而言是修改文件内容的权限
  - 对目录而言是删除、添加和重命名目录内文件的权限；
- **执行权限：**
  - 对可执行文件而言是允许执行的权限
  - 对目录而言是进入目录的权限。

目录权限的特殊性

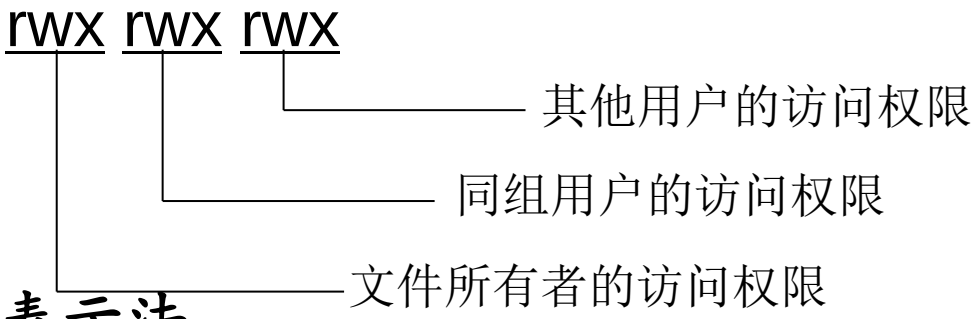
- 当打开一个任意类型的文件时，对该文件路径名中包含的每一个目录都应具有执行权限
- 为了在一个目录中创建一个新文件，必须对该目录具有写权限和执行权限
- 为了删除一个文件，必须对包含该文件的目录具有写权限和执行权限，对该文件本身则不需要有读、写权限

基于用户的文件权限管理

- **文件用户分类**
  - **文件所有者**：建立文件和目录的用户；
  - **文件所有者所在组用户**：文件所有者所属用户组中的其他用户；
  - **其他用户**：既不是文件所有者，又不是文件所有组所在组的其他所有用户。
  - **超级用户**：负责整个系统的管理和维护，拥有系统中所有文件的全部访问权限。

基于用户的文件权限管理

- 字母表示法



- 数字表示法

---	--x	-w-	-wx	r--	r-x	rw-	rwx
000	001	010	011	100	101	110	111

```
root@ubuntu: /usr/bin
root@ubuntu:/usr/bin# ls -l passwd
-rwsr-xr-x 1 root root 45420 Feb 16  2014 passwd
root@ubuntu:/usr/bin#
```

修改文件权限的chmod命令

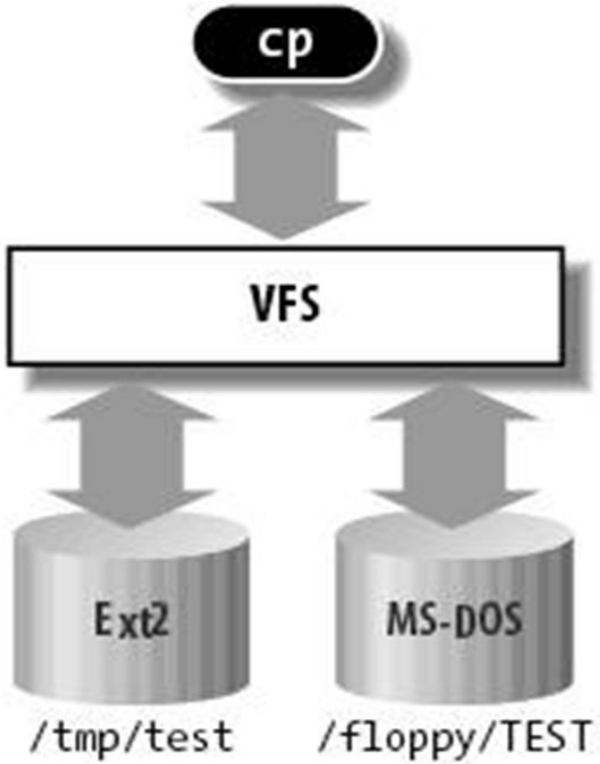
- **功能：修改文件的访问权限**
- **格式：chmod <模式> <文件>**
- **模式：**
  - 对象：u 文件所有者 、g 同组用户、o 其他用户
  - 操作符：+增加、-删除、=赋予
  - 权限：r 读、w 写、x 执行、s设置用户ID
- **举例：**
  - 取消同组用户对file文件的写入权限
    - chmod g-w file
  - 将pict目录的访问权限设置为775
    - chmod 755 pict
  - 设置file文件的设置用户ID位
    - chmod u+s file

主要内容

- UNIX/Linux文件系统的基本结构
- 索引节点的功能
- 文件的分类与权限
- 文件I/O
- 目录操作与文件属性

文件I/O操作

cp /floppy/TEST /tmp/test



(a)

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

(b)

打开文件

Open函数可打开任何类型的文件，包括设备文件和管道。

头文件：fcntl.h

```
int open( const char *pathname, int oflag, ...);
```

- **该函数打开或创建一个文件。其中第二个参数oflag说明打开文件的选项，第三个参数是变参，仅当创建新文件时才使用。**
  - O_RDONLY: 只读打开；
  - O_WRONLY: 只写打开；
  - O_RDWR: 读、写打开；
  - O_APPEND: 每次写都加到文件尾；
  - O_CREAT: 若此文件不存在则创建它，此时需要第三个参数mode，该参数约定了所创建文件的权限，计算方法为mode&~umask
  - O_EXCL: 如同时指定了O_CREAT，此指令会检查文件是否存在，若不存在则建立此文件；若文件存在，此时将出错。
  - O_TRUNC: 如果此文件存在，并以读写或只写打开，则删除原有内容
- **open返回文件描述符（当前进程中最小未使用的描述符数值）**

open函数参数

```
//come from /usr/include/bit/fcntl.h
/* open/fcntl - O_SYNC is only implemented on blocks devices and on files located on an ext2 file system */
#define O_ACCMODE       0003        //主要访问权限位为低两位，用来测试权限用
#define O_RDONLY        00          //只读
#define O_WRONLY        01          //只写
```

```
#define O_RDWR          02          //读写方式
#define O_CREAT         0100        //如果没有，创建
#define O_EXCL          0200        //如果存在，返回错误
#define O_NOCTTY        0400        //终端控制信息
#define O_TRUNC         01000       //截短
#define O_APPEND        02000       //追加
```

创建文件

头文件：fcntl.h

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

- 该函数用于创建一个新文件，其等效于open函数的如下调用：

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

- creat函数的一个不足之处是它以只写方式打开所创建的文件
- 早期UNIX版本中open的第二个参数不包含O_CREAT，无法打开一个尚未存在的文件。如果要创建一个临时文件，并要先写后读该文件。则必须先依次调用creat，close，open。

关闭打开的文件

头文件unistd.h

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

- 该函数关闭打开的一个文件。内核对文件描述符表、对应的文件表项和索引节点表项进行相应的处理，来完成关闭文件的操作。
- 进程关闭文件后，就不能再通过该文件描述符操作该文件
- 当一个进程终止时，它所有的打开文件将由内核自动关闭

读文件

头文件unistd.h

```
ssize_t  read( int filedes,  void *buf, size_t  nbytes);
```

- read函数从打开的文件中读数据。如成功，则返回实际读到的字节数，如已到达文件的末尾或无数据可读，则返回0；有多种情况可使实际读到的字节数少于要求读的字节数：
  - 读普通文件，在读到要求字节数之前就到达文件尾；
  - 当从终端设备读，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返回值小于所要求读的字节数；
  - 某些面向记录的设备，如磁带，一次最多返回一个记录。
- 读操作完成后，文件的当前位置将从读之前的位置加上实际读的字节数。

写文件

头文件unistd.h

```
ssize_t  write( int filedes,  const void *buf, size_t  nbytes);
```

- 该函数返回实际写的字节数，通常与参数nbytes的值相同，否则表示出错。
- 如果出错，则返回 - 1。write出错的原因可能是磁盘满、没有访问权限、或写超过文件长度限制等等。
- 对于普通文件，写操作从文件当前位置开始写（除非打开文件时指定了O_APPEND选项）。
- 写操作完成后，文件的当前位置将从写之前的位置加上实际写的字节数。

设置查询文件当前位置

头文件unistd.h

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

- 进程中每打开一个文件都有一个与其相关联的“文件当前位置”（读写位置）
- 打开文件时，如果指定了O_APPEND选项则文件当前位置为文件尾（文件长度），其他情况下文件当前位置默认为文件头（0）
- lseek函数用于设置或查询文件当前位置

设置查询文件当前位置

- **对参数的解释与参数whence的值有关：**
  - 若whence是SEEK_SET，则将该文件当前位置设为文件头+offset（以字节为单位）
  - 若whence是SEEK_CUR，则将该文件当前位置设为文件当前位置+offset （以字节为单位）
  - 若whence是SEEK_END，则将该文件当前位置设为文件尾+offset个字节（以字节为单位）
- **offset可正可负**

```
//come from /usr/include/unistd.h
/* Values for the WHENCE argument to lseek.  */
#ifndef  _STDIO_H        /* <stdio.h> has the same definitions.*/
# define SEEK_SET   0    /* Seek from beginning of file.  */      //文件起始位置
# define SEEK_CUR   1    /* Seek from current position.  */       //当前位置
# define SEEK_END   2    /* Seek from end of file.  */            //文件结束位置
```

设置查询文件当前位置

- 若lseek成功执行，则返回新的文件当前位置。因此可用lseek查询文件文件当前位置:

  currpos = lseek(fd, 0, SEEK_CUR)

- lseek仅将文件当前位置记录在内核file结构中，它**并不引起任何I/O操作**，然后用于影响下一次读、写操作。
- 文件当前位置**可以大于文件的当前长度**，但并不改变索引节点中文件长度信息（下一次写将延长该文件，并在文件中构成一个空洞，但文件大小并不是文件当前位置指示的值。对空洞位置的读操作将返回0）

主要内容

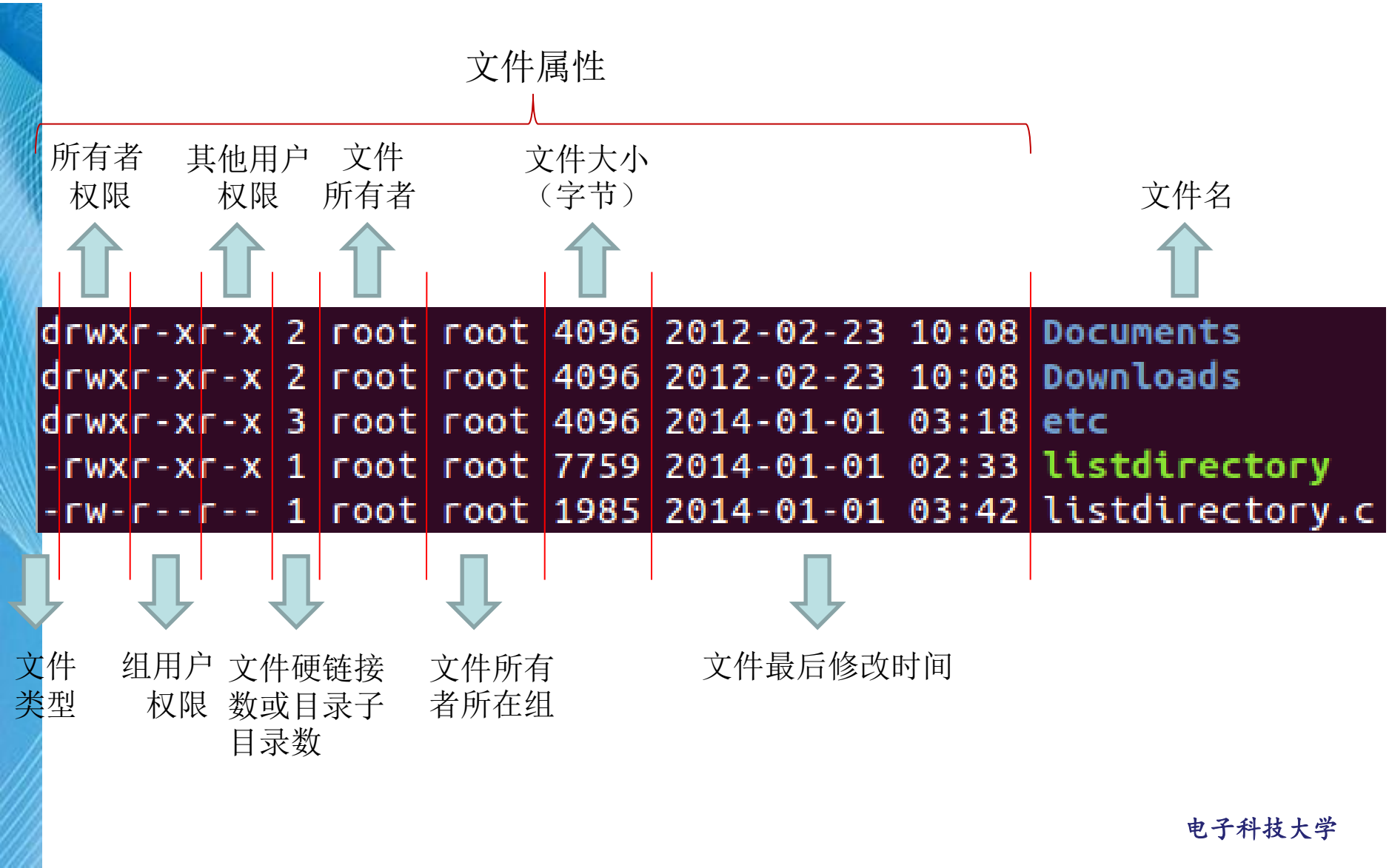
- UNIX/Linux文件系统的基本结构
- 索引节点的功能
- 文件的分类与权限
- 文件I/O
- 目录操作与文件属性

ls –l功能分析

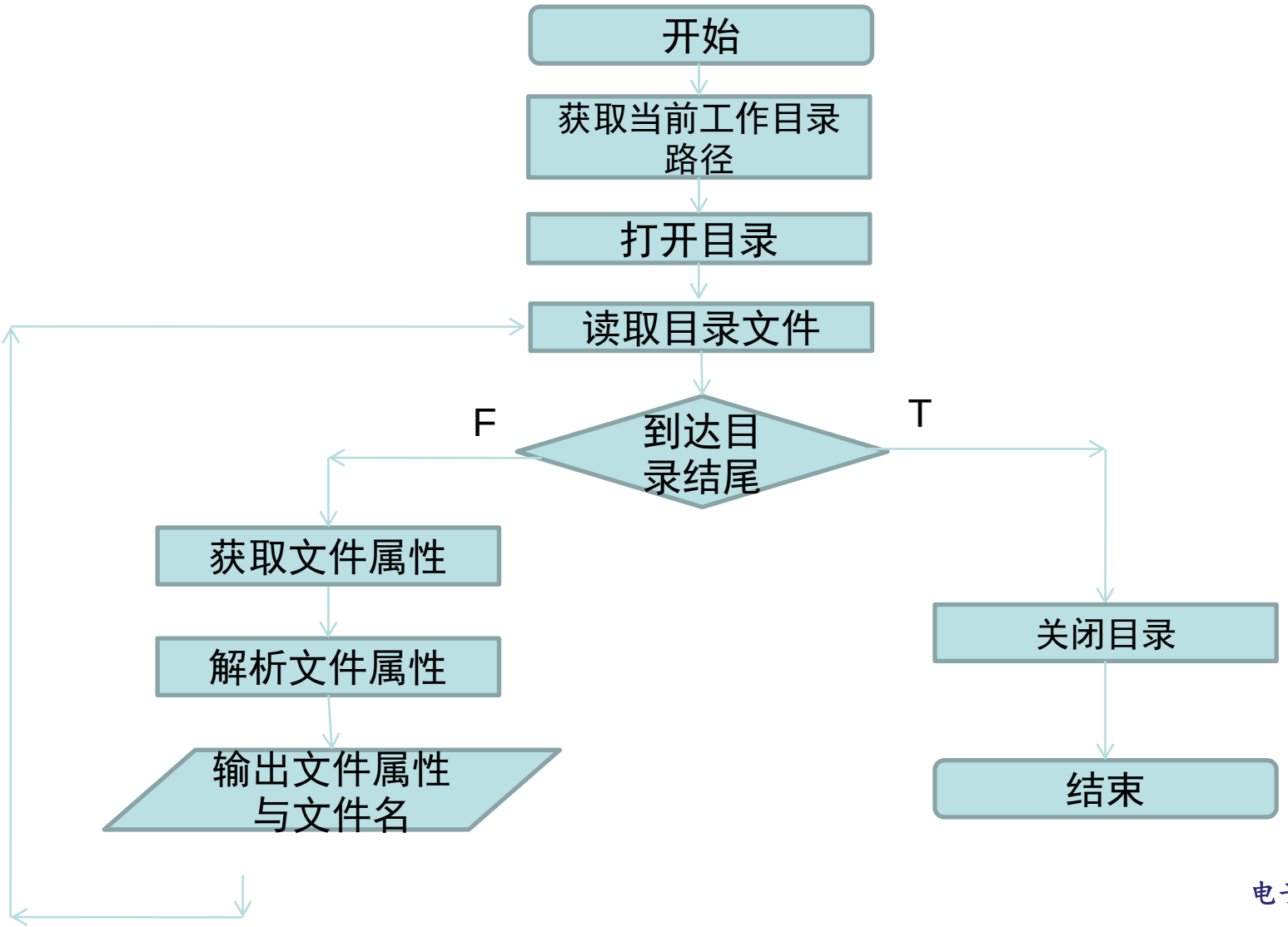
```
root@xrxy-virtual-machine: ~
root@xrxy-virtual-machine:~# cd /root/
root@xrxy-virtual-machine:~# ls -l
total 64
-rwxr-xr-x 1 root root 7255 2013-12-31 22:43 a.out
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Desktop
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Documents
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Downloads
drwxr-xr-x 3 root root 4096 2014-01-01 03:18 etc
-rwxr-xr-x 1 root root 7759 2014-01-01 02:33 listdirectory
-rw-r--r-- 1 root root 1985 2014-01-01 03:42 listdirectory.c
-rw-r--r-- 1 root root 1985 2014-01-01 02:41 listdirectory.c~
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Music
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Pictures
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Public
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Templates
drwxr-xr-x 6 root root 4096 2014-01-01 03:18 usr
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Videos
root@xrxy-virtual-machine:~#
```

列出当前目录的所有文件（包括普通文件、目录文件、字符特殊文件、套接字等），并显示文件类型、访问权限、文件大小等重要属性

ls –l功能分析


文件属性
所有者权限
其他用户权限
文件所有者
文件大小（字节）
文件名
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Documents
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Downloads
drwxr-xr-x 3 root root 4096 2014-01-01 03:18 etc
-rwxr-xr-x 1 root root 7759 2014-01-01 02:33 listdirectory
-rw-r--r-- 1 root root 1985 2014-01-01 03:42 listdirectory.c
文件类型
组用户权限
文件硬链接数或目录子目录数
文件所有者所在组
文件最后修改时间

ls –l 程序设计


开始
获取当前工作目录路径
打开目录
读取目录文件
到达目录结尾
F
T
获取文件属性
解析文件属性
输出文件属性与文件名
关闭目录
结束

目录操作-获取当前工作路径

- 常用函数：getcwd，get_current_dir_name
- 头文件：unistd.h
- 函数定义：
  - char *getcwd(char *buf, size_t size)
    - 将当前的工作目录绝对路径字符串复制到参数buf 所指的缓冲区，参数size 为缓冲区大小
    - 若参数buf 为NULL，参数size 为0，则函数根据路径字符串的长度自动分配缓冲区，并将分配的路径字符串缓冲区指针作为函数返回值（该内存区需要手动释放）
    - 失败返回NULL
  - char *get_current_dir_name(void)
    - 成功返回路径字符串缓冲区指针（该内存区需要手动释放），失败返回NULL

目录操作-打开关闭目录

- 常用函数：opendir，closedir
- 头文件：dirent.h
- 函数定义：
  - DIR * opendir(const char * name);
    - 打开参数name指定的目录，并使一个目录流与它关联
    - 目录流类似于C库函数中的文件流
    - 失败返回NULL
  - int closedir(DIR *dir);
    - 关闭指定目录流，释放相关数据结构
    - 成功返回0；失败返回-1

目录操作-读取目录文件

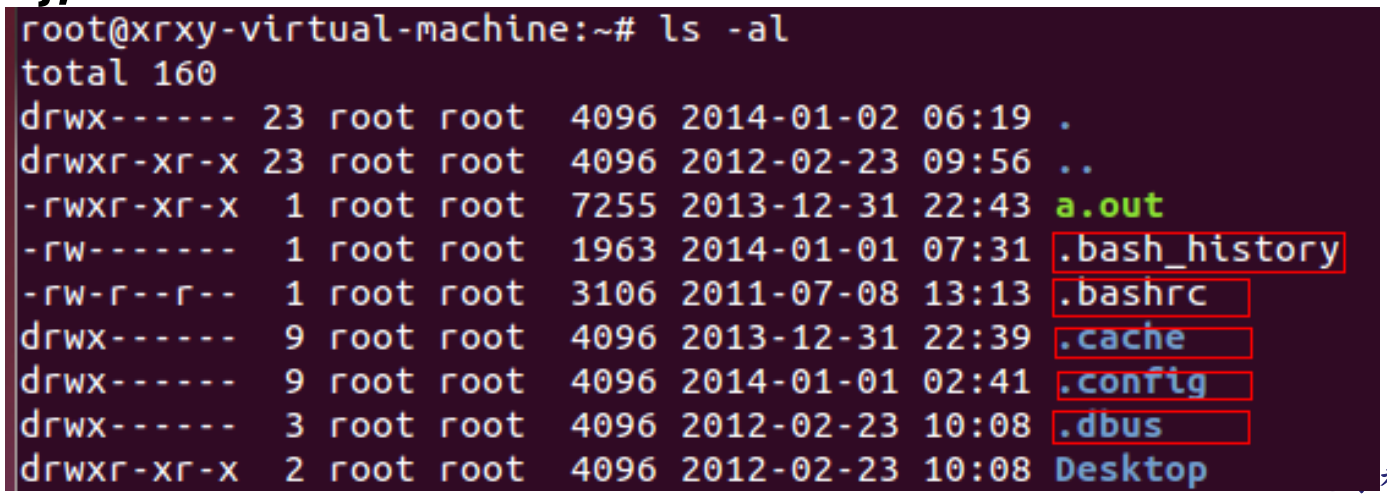
- 常用函数：readdir
- 头文件：sys/types.h；dirent.h
- 函数定义：
  - struct dirent * readdir(DIR * dir);
    - 读取目录流标识的目录文件

```
1  if((currentdir = opendir(buf)) == NULL)
2   {
3     printf("open directory fail\n");
4     return 0;
5   }
6   else
7   {
8     printf("file in directory include:\n")
9     while((currentdp = readdir(currentdir))!= NULL)
10        printf("%s ", currentdp->d_name);
11  }
```

目录操作-读取目录文件

- 重要数据结构

```
struct dirent
  {
    ino_t d_ino;  i节点号
    off_t d_off;   在目录文件中的偏移
    usigned short d_reclen; 文件名长度
    unsigned char d_type;   文件类型
    char d_name[256];文件名
  };
```



文件属性管理-读取文件属性

- 常用函数：stat，lstat，fstat
- 头文件： sys/stat.h
- 函数定义：
  - int stat(const char *path, struct stat *buf);
  - int lstat(const char *path, struct stat *buf);
    - 两个函数参数相同，功能类似
    - 读取path参数所指定文件的文件属性并将其填充到buf参数所指向的结构体中
    - 对于符号链接文件，lstat返回符号链接的文件属性，stat返回符号链接引用文件的文件属性
  - int fstat(int filedes, struct stat *buf);
    - 与前两个函数功能类似，指定文件的方式改为通过文件描述符

文件属性管理-文件属性解析

重要数据结构

```
struct stat {
  mode_t    st_mode;
  ino_t     st_ino;
  dev_t     st_dev;
  dev_t     st_rdev;
  nlink_t   st_nlink;
  uid_t     st_uid;
  gid_t     st_gid;
  off_t     st_size;
  time_t    st_atime;
  time_t    st_mtime;
  time_t    st_ctime;
  blksize_t st_blksize;
  blkcnt_t  st_blocks;
};
```

文件类型与访问权限

i节点号

文件使用的设备号

设备文件的设备号

文件的硬链接数

文件所有者用户ID

文件所有者组ID

文件大小（以字节为单位）

最后一次访问该文件的时间

最后一次修改该文件的时间

最后一次改变该文件状态的时间

包含该文件的磁盘块的大小

该文件所占的磁盘块数

文件属性管理-文件属性解析

重要数据结构

- mode_t st_mode;
  - 无符号整数，其低16位定义如下

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					U	G	T	R	W	X	R	W	X	R	W	X

文件类型域　文件特殊属性域　所有者权限域　组权限域　其他用户权限域

文件属性管理-判定文件类型

- 是否为普通文件: S_ISREG(st_mode)

```
#define S_IFMT 0170000
#define S_IFREG 0100000
#define S_ISREG(m) (((m) & S_IFMT) == S_IFREG)
```

- 是否为目录文件 S_ISDIR(st_mode)
- 是否为字符设备 S_ISCHR(st_mode)
- 是否为块设备 S_ISBLK(st_mode)
- 是否为FIFO S_ISFIFO(st_mode)
- 是否为套接字 S_ISSOCK(st_mode)
- 是否为符号连接 S_ISLINK(st_mode)

判断文件类型代码示例

```
int main(int argc, char *argv[])
{
        int                 i;
        struct stat     buf;
        char            *ptr;
        for (i = 1; i < argc; i++) {
                printf("%s: ", argv[i]);
                if (lstat(argv[i], &buf) < 0) {
                        err_ret("lstat error");
                        continue;
                }
```

判断文件类型代码示例

```
        if(S_ISREG(buf.st_mode))  ptr = "regular";
        else if (S_ISDIR(buf.st_mode))  ptr = "directory";
        else if (S_ISCHR(buf.st_mode))  ptr = "character
special";
        else if (S_ISBLK(buf.st_mode))  ptr = "block special";
        else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))  ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode)) ptr = "socket";
        else ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

文件属性管理:根据用户ID获取用户属性

- 常用函数：getpwuid
- 头文件： sys/types.h，pwd.h
- 函数定义：
  - struct passwd *getpwuid(uid_t uid);
    - 输入用户ID，返回用户属性信息（passwd结构）

```
struct passwd{
    char *pw_name;      /* 用户名*/
    char *pw_passwd;    /* 密码.*/
    __uid_t pw_uid;     /* 用户ID.*/
    __gid_t pw_gid;     /*组ID.*/
    char *pw_gecos;     /*真实名*/
    char *pw_dir;       /* 主目录.*/
    char *pw_shell;     /*使用的shell*/};
```

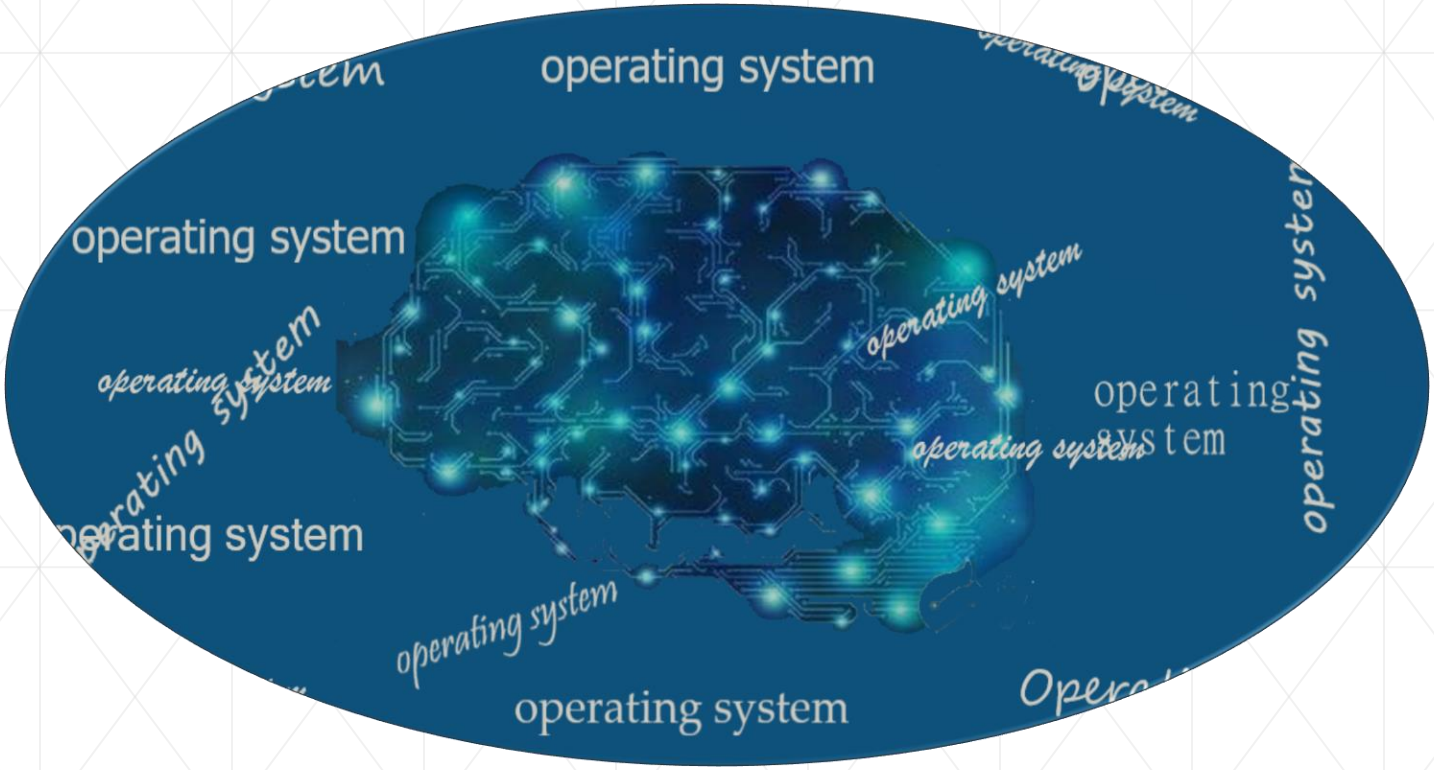
文件属性管理：根据组ID获取组属性

- 常用函数：getgrgid
- 头文件： sys/types.h，grp.h
- 函数定义：
  - struct group *getgrgid(gid_t gid);
    - 输入用户组ID，返回用户组属性信息（group结构）

```
struct group{
  char *gr_name; /*组名称*/
  char *gr_passwd; /* 组密码*/
  gid_t gr_gid; /*组ID*/
  char **gr_mem; /*组成员账号*/ }
```

ls –l实现关键代码

```
56
57    if(getcwd(buf,100) != NULL)
58      printf("%s\n",buf);
59    if((currentdir = opendir(buf)) == NULL)
60    {
61      printf("open directory fail\n");
62      return 0;
63    }
64    while((currentdp = readdir(currentdir)) != NULL)
65    {
66      if(currentdp->d_name[0] != '.')
67      {
68          if(lstat(currentdp->d_name,&currentstat) == -1)
69          {
70              printf("get stat error\n");
71              continue;
72          }
73          print_type(currentstat.st_mode);
74          print_perm(currentstat.st_mode);
75          print_link(currentstat.st_nlink);
76          print_usrname(currentstat.st_uid);
77          print_grname(currentstat.st_gid);
78          print_time(currentstat.st_mtime);
79          print_filename(currentdp);
80      }
81    }
82    closedir(currentdir);
83    return 0;
```



感谢观看！

授课教师：

电子邮箱：