

## 第三章

### 系统I/O

讲师：任继梅

QQ: 59189174

# 课前提问

1. 标准文件I/O和文件I/O有什么区别？
2. 文件I/O常见函数有哪些？
3. 文件读写方式有哪些？
4. 常见的目录操作函数有哪些？

# 本章内容

- ✓ 1.1 Unix文件基础
- ✓ 1.2 I/O相关的操作函数
- ✓ 1.3 文件属性操作函数
- ✓ 1.4 目录操作相关函数
- ✓ 1.5 其他文件相关操作



# 本章目标

- ✓ 了解Unix文件基础 ★★
- ✓ 理解相关的操作函数 ★★★
- ✓ 文件属性操作函数 ★★★★★
- ✓ 目录操作相关函数 ★★★
- ✓ 其他文件相关操作 ★★★

## 第一节

# Unix文件基础

# Unix文件基础-进程和程序

## ● 程序(静态)

- 存放在磁盘文件中的可执行文件
- 通过exec函数族调用运行

## ● 进程(动态)

- 程序的执行实例
- 进程的标识(pid,ppid,...)

## ● 进程控制

- fork()
- exec函数族
- wait()/waitpid()

嵌入式

# Unix文件基础-出错处理

## 全局错误码errno

- 在errno.h中定义，全局可见
- 错误值定义为“EXXX”形式，如EACCESS

## 处理规则

- 如果没有出错，则errno值不会被一个例程清除，即只有出错时，才需要检查errno值
- 任何函数都不会将errno值设置为0，errno.h中定义了所有常数都不为0

## 错误信息输出

- strerror() - 映射errno对应的错误信息
- perror() - 输出用户信息及errno对应的错误信息

# Unix文件基础-用户标识

## ● 用户ID(uid)

- 标识不同的用户，用户登录时通过/etc/passwd文件配置
- 通过getuid()可以获取uid
- 每个文件/目录都有相应的owner权限(-rwx-----)

## ● 组ID(gid)及添加组ID

- 通过组将多个有用户集中起来进行管理
- 用户登录时通过/etc/passwd文件配置
- 组ID与组名通过/etc/group文件配置
- 每个文件/目录都有相应的group权限(----rwx---)

1. 实际用户ID和有效用户ID (进程控制部分讲解)

2. 实际组ID和有效组ID (进程控制部分讲解)



# Unix文件基础-系统调用和库函数

## ● 系统调用

- 用户空间进程访问内核的接口
- 把用户从底层的硬件编程中解放出来
- 极大的提高了系统的安全性
- 使用户程序具有可移植性

## ● 库函数

- 库函数为了实现某个功能而封装起来的API集合
- 提供统一的编程接口，更加便于应用程序的移植

## ● 系统调用与库函数的比较

- 所有的操作系统都提供多种服务的入口点，通过这些入口点，程序向内核请求服务
- 从执行者的角度来看，系统调用和库函数之间有重大的区别，但从用户的角度来看，其区别并不非常的重要。
- 应用程序可以调用系统调用或者库函数，库函数则会调用系统调用来完成其功能。
- 系统调用通常提供一个访问系统的最小界面，而库函数通常提供比较复杂的功能。

## 第二节 文件I/O函数

# 文件I/O












## ● 文件描述符

- Linux内核为每个进程维护了一张表，这个表记录了这个进程正在打开的哪些文件，表里每条记录都有一个编号，这个编号就是文件描述符
- 一个非负整数
- 取值范围0~OPEN\_MAX
- OPEN\_MAX通常为64
- 普通文件的文件描述符一般 $\geq 3$
- 所有文件操作函数都围绕文件描述符展开
- 标准输入、标准输出和标准出错
- 由shell默认打开，分别为0/1/2



# 文件I/O

## 相关系统调用函数列表

-  open 打开文件
-  create 创建新文件
-  close 关闭文件
-  read 读文件
-  write 写文件
-  lseek 设置当前文件偏移量
-  access 测试文件访问权限
-  ioctl io操作杂物箱
-  fcntl 改变已打开文件的性质
-  fstat、stat、lstat 获取文件的元信息
-  dup、dup2 复制文件描述符

# 文件I/O-open和close

## 📄 open()/creat()函数可以打开或者创建一个文件

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

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

- ③ open()和creat()调用成功返回文件描述符，失败返回-1，并设置errno。
- ③ open()/creat()调用返回的文件描述符一定是最小的未用描述符数字。
- ③ creat()等价于open(pathname, O\_CREAT|O\_WRONLY|O\_TRUNC, mode)
- ③ open()可以打开设备文件，但是不能创建设备文件，设备文件必须使用mknod()创建。

# 文件I/O-open和close

原型	int open(const char *pathname, int flags, mode_t mode);		
参数	pathna me	被打开的文件名（可包括路径名）。	
	flags	O_RDONLY: 只读方式打开文件。	这三个参数互斥
		O_WRONLY: 可写方式打开文件。	
		O_RDWR: 读写方式打开文件。	
		O_CREAT: 如果该文件不存在，就创建一个新的文件，并用第三的参数为其设置权限。	
		O_EXCL: 如果使用O_CREAT时文件存在，则可返回错误消息。这一参数可测试文件是否存在。	
		O_NOCTTY: 使用本参数时，如文件为终端，那么终端不可以作为调用open()系统调用的那个进程的控制终端。	
		O_TRUNC: 如文件已经存在，那么打开文件时先删除文件中原有数据。	
		O_APPEND: 以添加方式打开文件，所以对文件的写操作都在文件的末尾进行。	
	mode	被打开文件的存取权限，为8进制表示法。	

# 文件I/O-open和close

## 🔴 close()函数可以关闭一个打开的文件

```
#include <unistd.h>  
int close(int filders);
```

- 🟢 调用成功返回0，出错返回-1，并设置errno。
- 🟢 当一个进程终止时，该进程打开的所有文件都由内核自动关闭。
- 🟢 关闭一个文件的同时，也释放该进程加在该文件上的所有记录锁。



# 文件I/O-read和write

## 📄 read()函数可以从一个已打开的可读文件中读取数据

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

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

- read()调用成功返回读取的字节数，
- 如果返回0，表示到达文件末尾，
- 如果返回-1，表示出错，通过errno设置错误码。
- 读操作从文件的当前位移量处开始，在成功返回之前，该位移量增加实际读取的字节数。

# 文件I/O-read和write

## write()函数可以向一个已打开的可写文件中写入数

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

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

- write()调用成功返回已写的字节数，失败返回-1，并设置errno。
- write()的返回值通常与count不同，因此需要循环将全部待写的数据全部写入文件。
- write()出错的常见原因：
  - 磁盘已满或者超过了一个给定进程的文件长度限制。
- 对于普通文件，写操作从文件的当前位移量处开始，如果在打开文件时，指定了O\_APPEND参数，则每次写操作前，将文件位移量设置在文件的结尾处，在一次成功的写操作后，该文件的位移量增加实际写的字节数。

# 文件I/O-read和write

## write()函数反复的写入：

```
while(1)
{
    pos = 0;
    len = read(fd, buf, sizeof(buf));
    if(len == 0)
        break;
    else if(len < 0)
    {
        perror("read()");
        return -1;
    }

    while(len > 0)
    {
        ret = write(fd2, buf + pos, len);
        if(ret < 0)
        {
            perror("write()");
            return -1;
        }
        pos = pos + ret;
        len = len - ret;
    }
}
```

# 文件I/O-Lseek

调用 **lseek()**函数可以显示的定位一个已打开的文件。

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

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

原型	off_t lseek(int fd,off_t offset,int whence);		
参数	fd: 文件描述符。		
	offset: 偏移量，每一读写操作所需要移动的距离，单位是字节的数量，可正可负（向前移，向后移）		
	whence (当前位置 基点):	SEEK_SET: 当前位置为文件的开头，新位置为偏移量的大小。	
		SEEK_CUR: 当前位置为文件指针的位置，新位置为当前位置加上偏移量。	
		SEEK_END: 当前位置为文件的结尾，新位置为文件的大小加上偏移量的大小。	
返回值	成功: 文件的当前位移		
	-1: 出错		



# 文件I/O-Lseek

- 每个打开的文件都有一个与其相关的“当前文件位移量”，它是一个非负整数，用以度量从文件开始处计算的字节数。
- 通常，读/写操作都从当前文件位移量处开始，在读/写调用成功后，使位移量增加所读或者所写的字节数。
- lseek()调用成功为新的文件位移量，失败返回-1，并设置errno。
- lseek()只对常规文件有效，对socket、管道、FIFO等进行lseek()操作失败。
- lseek()仅将当前文件的位移量记录在内核中，它并不引起任何I/O操作
- 文件位移量可以大于文件的当前长度，在这种情况下，对该文件的写操作会延长文件，并形成空洞。

# 文件I/O-Lseek

## example: 创建一个有空洞的文件：

```
//FIXME :loop back to write rest data
If(write(fd,buf1,strlen(buf1))<strlen(buf1))
{
    fprintf(stderr,"buf1 write error\n");
    return -1;
}
If(lseek(fd,40,seek_set)==-1)
{
    fprintf(stderr,"lseek() error\n");
    return -1;
}
//FIXME:Loop back to write reset data
If(write(fd,buf2,strlen(buf2))<strlen(buf2))
{
    fprintf(stderr,"buf1 write error\n");
    return -1;
}
```



问题：创建有空洞的文件有何用途？

# 文件I/O-作业

## 任务描述

- 使用非缓冲I/O方式，实现一个copy程序，该程序的第一个命令行参数为源文件，第二个命令行参数为目标文件，程序实现将源文件中的内容复制到目标文件。
- 命令行参数可以使用argv[1]访问第一个参数，argv[2]访问第二个参数
- 使用diff工具检查目标文件与源文件是否一致

## 实现思路

- 打开源文件
- 打开目标文件
- 循环读取源文件并写入目标文件
- 关闭源文件
- 关闭目标文件

# 文件I/O-作业

## ● 任务描述

- 使用非缓冲I/O方式，实现一个copy程序，该程序的第一个命令行参数为源文件，第二个命令行参数为目标文件，程序实现将源文件中的内容复制到目标文件。
- 命令行参数可以使用argv[1]访问第一个参数，argv[2]访问第二个参数
- 使用diff工具检查目标文件与源文件是否一致
- 关闭目标文件



# 文件I/O-作业



## 实现思路

- 打开源文件
- 打开目标文件
- 循环读取源文件并写入目标文件
- 关闭源文件

嵌入式

# 嵌

## 第三节

### 文件属性操作

# 文件属性操作-stat、fstat、lstat

以下三个函数可以获取文件/目录的属性信息：

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

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

三个函数的返回：若成功则为0，若出错则为-1,并且设置errno.

给定一个pathname的情况下：

- stat函数返回一个与此命名文件有关的信息结构
- fstat函数获得已在描述符filedes上打开的文件的有关信息
- lstat函数类似于stat，但是当命名的文件是一个符号连接时，lstat返回该符号连接的有关信息，而不是由该符号连接引用的文件的信息。

# 文件属性操作-stat、fstat、lstat

## struct stat定义：

```
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 file system I/O */  
    blkcnt_t   st_blocks;   /* number of 512B 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 */  
};
```

## 文件属性操作-取得文件类型

- 可以用以下的宏确定文件类型。这些宏的参数都是struct stat结构中的st\_mode成员。

S\_ISREG(m) is it a regular file?

S\_ISDIR(m) directory

S\_ISCHR(m) character device

S\_ISBLK(m) block device

S\_ISFIFO(m) fifo

S\_ISLNK(m) symbolic link(Not in POSIX 1-1996.)

S\_ISSOCK(m) socket (Not in POSIX 1-1996)



# 文件属性操作-access

当用**open**函数打开一个文件时，内核以进程的有效用户**ID**和有效组**ID**为基础执行其存取许可权测试。

有时，进程也希望按其实际用户**ID**和实际组**ID**来测试其存取能力。

```
#include<unistd.h>
int access(const char *path,int amode)
```

例如当一个进程使用设置-用户-**ID**，或设置-组-**ID**特征作为另一个用户(或组)运行时，这就可能需要测试其存取能力，即使一个进程可能已经设置-用户-**ID**为根，它仍可能想验证实际用户能否存取一个给定的文件。**access()**函数是按实际用户**ID**和实际组**ID**进行存取许可权测试的。

```
#include<unistd.h>
int result;
Const char *filename="/tmp/myfile";
Result=access(filename,F_OK);
```

# 上 嵌

## 第四节 目录文件操作

# 文件目录操作



## 常见函数列表

- mkdir 创建目录
- rmdir 删除目录
- chdir 改变工作目录
- getcwd 获取当前工作目录
- opendir 打开目录流
- readdir 读目录的每一项
- telldir 返回操作目录的位置
- seekdir 设置目录项的位置
- closedir 关闭目录流

# 目录操作

## 目录操作函数

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

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

int rmdir(const char *pathname);

int chdir(const char *path);

char *getcwd(char *buf, size_t size);
```

# 目录操作

## 目录读写函数

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

DIR *opendir(const char *name);

struct dirent *readdir(DIR *dirp);

long telldir(DIR *dirp);

void seekdir(DIR *dirp, long offset);

int closedir(DIR *dirp);
```



## 目录操作-例子：

```
#include <stdio.h>
#include <dirent.h>

int main()
{
    struct dirent *d;
    DIR *dir = opendir("."); /*读取当前文件夹*/
    if(dir == NULL)
    {
        perror("opendir()");
        return -1;
    }

    /*readdir每一次都返回文件夹中的一项，直到返回NULL，文件夹就为空*/
    while( (d = readdir(dir)) != NULL)
    {
        printf("%s \n", d->d_name);
    }

    closedir(dir);

    return 0;
}
```

## **第四节**

### **其他文件相关操作**

# 知识点-其他文件相关操作



## 常见函数列表

≡	unlink	删除链接
≡	link	创建硬链接
≡	symlink	创建符号链接
≡	chmod	改变文件或目录的访问权限
≡	chown	改变文件或目录的属主和组
≡	mmap	内存映射内存内容同步
≡	munmap	解除

# 其他文件相关操作-链接函数

## 链接相关函数

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

int link(const char *oldpath, const char *newpath);

int unlink(const char *pathname);

int symlink(const char *oldpath, const char *newpath);

int chmod(const char *path, mode_t mode);

int chown(const char *path, uid_t owner, gid_t group);
```



# 其他文件相关操作-ioctl

## 设备控制函数ioctl

```
#include <sys/ioctl.h>
```

```
int ioctl(int d, int request, ...);
```

- 功能：主要是获取设备信息
- 返回值：成功返回0，失败为NULL
- 参数：
  - d：文件描述符
  - request：请求命令
  - ...：可变参数列表



# ioctl 例子:

```
#include <stdio.h>
#include <sys/ioctl.h>
#include <linux/fb.h>

//主要是获取硬件设备信息
//int ioctl(int d, int request, ...);
int main()
{
    struct fb_var_screeninfo var_info;
    struct fb_fix_screeninfo fix_info;

    int fd;
    fd = open("/dev/fb0", "O_RDONLY");
    if(fd < 0)
    {
        perror("open()");
        return -1;
    }

    ioctl(fd, FBIOGET_VSCREENINFO, &var_info);
    ioctl(fd, FBIOGET_FSCREENINFO, &fix_info);

    printf("xres = %d, yres = %d, bit = %d\n",
           var_info.xres, var_info.yres, var_info.bits_per_pixel);
}
```

## ioctl 例子:

```
printf("screen width = %d\n", fix_info.line_length);  
close(fd);  
return 0;  
}
```

# 其他文件相关操作函数

## 内存映射函数mmap

```
#include <sys/mman.h>
```

```
void *mmap(void *addr, size_t length, int prot, int flags,  
           int fd, off_t offset);
```

```
int munmap(void *addr, size_t length);
```

- 功能：创建一个指向一段内存的指针，这段内存与文件内容相关联
- 返回值：成功返回被映射内存空间的首地址，失败为NULL
- 参数：
  - addr：请求使用某个特定的内存地址，为NULL时自动分配
  - len：被映射内存空间的大小
  - prot：内存段访问权限（见下一页）
  - flags：控制程序对内存段的改变所造成的影响（见下一页）
  - fildes：打开的文件描述符



# 其他文件相关操作函数

## 📖 Mmap参数说明

### ☰ Port

PROT_READ	允许读内存段
PROT_WRITE	允许写内存段
PROT_EXEC	允许执行内存段
PROT_NONE	不能访问内存段

### ☰ Flags

MAP_PRIVATE	对内存段的存储操作导致创建该映射文件的一个私有副本。修改只对该副本，而不是原始原件
MAP_SHARED	把对该内存段的修改保存到指定文件中
MAP_FIXED	该内存段必须位于addr指定的地址处

## 实例：写入FB实例

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main()
{
    unsigned char *addr = NULL;

    int fd = open("/dev/fb0", O_RDWR);
    if (fd < 0)
    {
        perror("open()");
        return -1;
    }

    /*屏幕分辨率*/
    addr = mmap(NULL, 1366*768*4, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
    {
        perror("mmap()");
        return -1;
    }
}
```

## 实例：写入FB实例

```
int i = 0;
int j = 0;
for(i = 0; i < 768; i++)
{
    for(j = 0; j < 1366; j++)
    {
        addr[(i*1366 + j)*4 + 0] = 0 ;           //透明度
        addr[(i*1366 + j)*4 + 1] = 123;         //R
        addr[(i*1366 + j)*4 + 2] = 123;         //G
        addr[(i*1366 + j)*4 + 3] = 255;         //B
    }
}

close(fd);

munmap(addr, 1366*768*4);

return 0;
}
```



# 嵌入式

## 第五节 设备文件编程

# Framebuffer控制编程





`/dev/fb0`



Linux下显示设备文件

# 键盘控制编程

 /dev/input/enevtn

 Linux下键盘设备文件

# 串口控制编程



`/dev/ttyUSB0`



Linux下串口设备文件



# 课程总结



## 本节课程内容

- Unix文件基本概念
- 文件I/O操作
- 文件属性操作
- 目录操作
- 设备文件操作



## 下节课程

- 进程管理
- 进程环境
- 进程控制
- 进程关系
- 守护进程

# 联系方式

---

QQ: 59189174

E-mail: [yumeifly@sohu.com](mailto:yumeifly@sohu.com)

