文件IO

武汉众嵌

主要内容

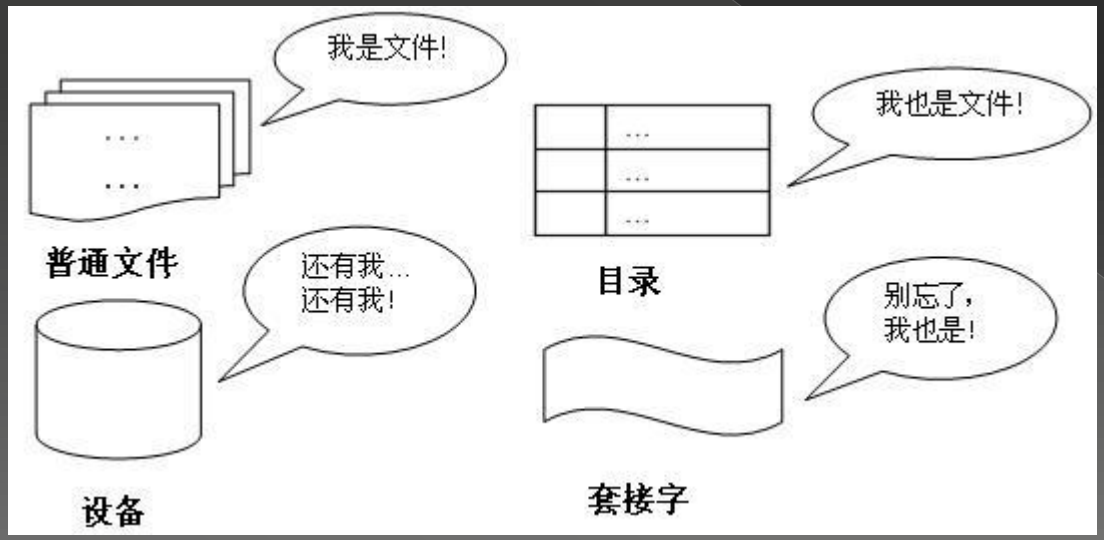
- 文件IO
- 文件目录

文件IO

- 文件的概念
- 文件描述符
- 创建/打开/关闭文件
- 文件的读写
- 文件的偏移
- 文件属性的设置与更改

文件的概念

- 文件：一组在逻辑上具有完整意义的信息项的系列。在Linux中，除了普通文件，其他诸如目录、设备、套接字等也以文件被对待。总之，“一切皆文件”。



文件描述符

- 内核利用文件描述符来访问文件。对于内核而言，所有打开文件都由文件描述符引用。
- 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。
- 在posix.1应用程序中，整数0、1、2应被代换成符号常数STDIN_FILENO、STDOUT_FILENO、STDERR_FILENO。这些常数都定义在头文件<unistd.h>中

创建/打开/关闭文件

- 打开或者创建一个文件：
- int open(const char* pname,int flags);
- int open(const char* pname,int flags,mode_t mode);
- int creat(const char* pname,mode_t);
- open()和creat()调用成功返回文件描述符，失败返回-1，并设置errno。
- 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。
- creat()等价于open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode)
- 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函数(创建特殊函数)创建。

创建/打开/关闭文件

原型	int open(const char *pathname, int flags, mode_t mode);		
参数	pathname	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，并且以只读或只写成功打开，那么先全部删除文件中原有数据。	
		O_APPEND：以添加方式打开文件，在打开文件的同时，文件指针指向文件的末尾。	
	mode	被打开文件的存取权限，为8进制表示法。	

实例：

```
int main()
{
    int fd1,fd2;
    if( (fd1 = open("1.txt",O_CREAT|O_RDWR,0666)) < 0)
    {
        perror("open fd1 failed");
        exit(0);
    }
    if( (fd2 = open("2.txt",O_CREAT|O_RDWR,0666)) <0)
    {
        perror("open fd2 failed");
        exit(0);
    }
    fprintf(stdout,"fd1: %d ,fd2: %d ",fd1,fd2);
}
```

创建/打开/关闭文件

- 关闭一个打开的文件：
- int close(int fd);
- 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。

范例：

```
if(close(fd)<0)
{
        perror(NULL);
}
```

练习：

- example1:使用文件I/O测试系统能最多打开的文件数目：
- example2:以所有者只写方式(参数O_WRONLY)打开文件(mode=644)，如果文件不存在则创建(参数O_CREAT)，如果文件存在则截短(参数O_TRUNC)。注：本例中，mode可以直接赋值为0644。

文件的读写

- 从一个已打开的可读文件中读取数据。
- 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。
- buf参数需要有调用者来分配内存，并在使用后，由调用者释放分配的内存。

实例：

- 从终端读取数据,并输出
- int main()
- {
- char buf[1024];
- if(read(STDIN_FILENO,buf,sizeof(buf)) <0)
- {
- perror("read failed");
- exit(0);
- }
- fprintf(stdout,"%s",buf);
- }

文件的读写

- 向一个已打开的可写文件中写入数据：
- ssize_t write(int fd,const void* buf,size_t count);

- 调用成功返回已写的字节数，失败返回-1，并设置errno。

文件的读写

- 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当前结尾处，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。

例题：

- 从指定文件中读取数据，写入到终端。

文件的偏移

- 偏移函数：
- off_t lseek(int fd,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	

```
int main(){
        int fd;
        char buf[1024];
        if( (fd = open("1.txt",O_CREAT|O_RDWR,0666)) < 0){
                perror("open fd1 failed");
                exit(0);
        }
        if(lseek(fd,3,SEEK_SET) < 0){
                perror("lseek failed");
                exit(0);
        }
        if( read(fd,buf,sizeof(buf)) <0){
                perror("read failed");
                exit(0);
        }
        if(write(STDOUT_FILENO,buf,strlen(buf)) < 0){
                perror("write failed");
                exit(0);
        }
        if(close(fd)<0){
                perror(NULL);
        }
}
```

1.Txt文件内容

Hello world!

文件的偏移

- 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。

文件属性的设置与更改

- int fcntl(int fd,int cmd,...../*int arg*/);
- 返回值：若出错返回-1
- 此函数中，一般情况下，第三个参数总是一个整数，与上面所示函数原型中的注释部分对应。
- fcntl函数有5中功能：
  - 复制一个现有的描述符(cmd = F_DUPFD);
  - 获得/设置描述符标记(cmd= F_GETFD/SETFD);
  - 获得/设置文件状态标志(cmd=F_GETFL/SETFL);
  - (通过信号)获得/设置异步I/O所有权(cmd=F_GETOWN/SETOWN);
  - 获得/设置记录锁(cmd=F_GETLK/SETLK/SETLKW);

```
int main()
{
        int fd;
        char buf[1024];
        if( (fd = open("1.txt",O_CREAT|O_RDWR,0666)) < 0){
                perror("open fd1 failed");
                exit(0);
        }
        int flag = fcntl(fd,F_GETFL);
        fcntl(fd,F_SETFL,flag|O_APPEND);      // 添加append 属性
        if( read(STDIN_FILENO,buf,sizeof(buf)) <0){
                perror("read failed");
                exit(0);
        if(write(fd,buf,strlen(buf)) < 0){
                perror("write failed");
                exit(0);
        }
        if(close(fd)<0){
                perror(NULL);
        }
}
```

文件I/O和标准I/O的转换

- 每个标准I/O流都有一个与其关联的文件描述符，可以对一个流调用fileno函数以获得其描述符：
- int fileno(FILE* fp);
- 函数中的参数是一个文件流，返回值为一个文件描述符。

```
int main(){
        int fd;
        FILE* fp;
        char buf[1024];
        fp = fopen("1.txt","r+");
        if(fp == NULL){
                perror("fopen failed");
                exit(0);
        }
        fd = fileno(fp);
        if(read(fd,buf,sizeof(buf)) < 0){
                perror("read failed");
                exit(0);
        }
        printf("%s",buf);
        if(close(fd)<0){
                perror(NULL);
        }
}
```

文件目录

- stat,fstat函数
- 文件类型介绍
- 用户标识和组标识
- 文件访问权限
- access，umask，chmod函数
- 链接
- 符号链接
- mkdir和rmdir函数
- 读目录

stat,fstat函数

- int stat(const char* path,struct stat* buf);
- int fstat(int fd,struct stat* buf);
- 返回值：成功返回0，失败返回-1；
- 一旦调用stat函数，stat函数就返回与该文件有关的信息结构。fstat函数获取已打开的文件描述符，
- 第二个参数buf是指针，它指向一个我们必须提供的结构。这些函数填写由buf指向的结构

```
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 */
};
```

```
int main(int argc,char** argv)
{
        struct stat buf;
        if(argc != 2 )
        {
                perror("format err.");
                return -1;
        }
        if(stat(argv[1],&buf) < 0)
        {
                perror("stat failed.");
                return -1;
        }
        printf("%-10s",argv[1]);
        printf("%-5ld",buf.st_uid);
        printf("%-5ld",buf.st_gid);
        printf("\t%-5ld",buf.st_size);
        printf("\t%s",ctime(&buf.st_mtime));

}
```

文件类型介绍

- 至今我们已介绍了两种不同的文件类型——普通文件和目录。Unix系统的大多数文件是普通文件或目录，但是也有另外一些文件类型。文件类型包括如下几种：
  - 普通文件(regular). 这是最常用的文件类型，这种文件包含了某种形式的数据。
  - 目录文件(directory).这种文件包含了其他文件的名字以及指向与这些文件有关信息的指针。
  - 符号链接(symbolic).这种文件类型指向另一个文件。
  - FIFO.这种类型文件用于进程间通信，也将称为命名管道
  - 套接字(socket).这种文件类型用于进程间的网络通信。
  - 块特殊文件(block special).这种文件类型提供对设备带缓冲的访问，每次访问以固定长度为单位进行。
  - 字符特殊文件(character special).这种文件类型提供对设备不带缓冲的访问，每次访问长度可变。

文件类型介绍

- 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 (named pipe)?

S_ISLNK(m)  symbolic link? (Not in POSIX.1-1996.)

S_ISSOCK(m) socket? (Not in POSIX.1-1996.)
```

```
int main(int argc,char** argv)
{
        struct stat buf;
        if(argc != 2 ){
                perror("format err.");
                return -1;
        }
        if(stat(argv[1],&buf) < 0){
                perror("stat failed.");
                return -1;
        }
        printf("%s\t",argv[1]);
        if (S_ISREG(buf.st_mode)) printf("regular file.");
        if (S_ISLNK(buf.st_mode)) printf("symbolic link file.");
        if (S_ISDIR(buf.st_mode)) printf("directory file.");
        if (S_ISBLK(buf.st_mode)) printf("block device.");
        if (S_ISCHR(buf.st_mode)) printf("character device.");
        if (S_ISFIFO(buf.st_mode))printf("FIFO file.");
        if (S_ISSOCK(buf.st_mode))printf("socket file.");
        printf("\n");
}
```

用户标识和组标识

- 在Linux系统中，与一个进程相关联的ID有6个或更多。
  - 实际用户ID和实际组ID标识我们究竟是谁。这两个字段在登录时取自口令文件中的登录项。
  - 有效用户ID，有效组ID以及附加组ID决定了我们的文件访问权限。
  - 保存的设置用户ID和保存的设置组ID在执行一个程序时包含了有效用户ID和有效组ID的副本
- 通常，有效用户ID等于实际用户ID，有效组ID等于实际组ID。

实际用户ID	我们实际是谁
实际组ID	
有效用户ID	作用：用于文件访问权限检查
有效组ID	
附加组ID	
保存的设置用户ID	由exec函数保存
保存的设置组ID	

文件访问权限

- 文件的访问权限根据文件访问权限位来决定，我们在获取文件相关信息时，可以从stat结构体中找到st_mode(访问权限)。对于任意文件，都有对应的访问权限。
- 每个文件有9个访问权限位，可分为三组，如左所示：

st_mode	意义
S_IRUSR	用户-读
S_IWUSR	用户-写
S_IXUSR	用户-执行
S_IRGRP	组-读
S_IWGRP	组-写
S_IXGRP	组-执行
S_IROTH	其他-读
S_IWOTH	其他-写
S_IXOTH	其他-执行

练习题：

- 编写一个程序，给出一个文件名，打印对应文件的相关信息(文件类型，访问权限，文件大小，修改日期，用户ID，群组ID，文件名等信息)。

access,umask,chmod函数

- int access(const char* pathname,int mode);
- 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为基础执行其访问权限测试，使用access，就能直接按照实际用户ID和组ID来测试其访问能力。
- pathname测试的文件名，mod测试功能，其功能如下：
- **R_OK** 测试读权限
- **W_OK** 测试写权限
- **X_OK** 测试执行权限
- **F_OK** 测试文件是否存在

access,umask,chmod函数

- mode_t umask(mode_t cmake);
- 其中，参数cmake是由对应的9个权限位中的若干个位按位“或”运算构成。
- 在进程创建一个新文件或新目录时，就一定会使用文件模式创建屏蔽字，对于任何模式下创建屏蔽字中为1的位，在文件mode中相对应位则一定被关闭。
- 注意：umask既可作为函数使用，也可以作为命令使用，使用的功能完全一样

屏蔽位	意义
0400	用户读
0200	用户写
0100	用户执行
0040	组读
0020	组写
0010	组执行
0004	其他读
0002	其他写
0001	其他执行

access,umask,chmod函数

- int chmod(const char* pathname,mode_t mode);
- int fchmod(int fd,mode_t mode);
- chmod函数在指定的文件上进行操作，而fchmod函数则对已打开的文件进行操作。
- 为了改变一个文件的权限位，进程的有效用户ID必须等于文件所有者ID，或者该进程必须具有超级用户权限。

Mode ·	说明
S_ISUID	执行时设置用户ID
S_ISGID	执行时设置组ID
S_ISVTX	保存正文
S_IRWXU	用户读，写，执行
S_IRUSR	用户读
S_IWUSR	用户写
S_IXUSR	用户执行
S_IRWXG	组读，写，执行
S_IRGRP	组读
S_IWGRP	组写
S_IXGRP	组执行
S_IRWXO	其他读，写，执行
S_IROTH	其他读
S_IWOTH	其他写
S_IXOTH	其他执行

链接

- 任何一个文件可以有多个目录项指向其i节点。在Unix系统下，创建一个执行现有文件的链接方法是使用link函数。
- int link(const char* oldpath,const char* newpath);
- 返回值:若成功返回0，出错返回-1；
- 此函数创建一个新目录项newpath，它引用现有的文件oldpath。若newpath已经存在，则返回出错。

链接

- 为了删除一个现有的链接，我们可以调用unlink函数。
- int unlink(const char* pathname);
- 此函数删除一个链接，并将由pathname所引用文件的链接计数减1。
- 只有当链接计数达到0时，该文件的内容才可被删除。
- 关闭一个文件时，内核首先检查打开该文件的进程数。如果该数达到0，然后检查其链接数，如果这个数也是0，那么就删除这个文件。

链接

- 对于解除链接，我们还可以使用remove函数，对于文件，remove的功能与unlink相同，对于目录，remove的功能与rmdir相同。
- int remove(const char* path);/删除/
- int rename(const char* oldname,const char* newname);//重命名函数
- 返回值：若成功返回0，否则返回-1；

链接

- 思考1：能否在不同的文件系统中创建一个链接？
- 思考2：能否对目录进行链接？

符号链接

- 符号链接是指向一个文件的间接指针，它与前面的链接（硬链接）有所不同，硬链接直接指向文件的I节点。引入符号链接的原因是为了避开硬链接的限制：
  - 硬链接通常要求链接和文件位于同一文件系统。
  - 不能对目录文件进行硬链接
- 对于符号链接以及它指向何种对象并无文件系统限制，任何用户都可创建指向目录的符号链接。符号链接一般用于一个文件或整个目录结构系统的另一个位置。

符号链接

- 创建/读一个符号链接：
- int symlink(char* path,char* sympath);
- 该函数创建一个指向path的符号链接文件sympath，在创建符号链接时，path文件存在与否函数并不关心，而且sympath和path也不需要在同一个文件系统中。

符号链接

- ssize_t readlink(char* path,char* buf,size_t bufsize);
- 该函数可以读取符号链接文件的内容。
- 返回值：若出错返回-1，否则返回读到的字节数。

mkdir和rmdir函数

- 在Linux中，使用mkdir和rmdir函数
- int mkdir(const char* pathname,mode_t mode);
- 返回值：成功返回0，否则返回1.
- 此函数创建一个新的空目录。其中.和..目录项是自动创建的。所指定的文件访问权限mode由进程的文件模式创建屏蔽字修改。

mkdir和rmdir函数

- 用rmdir函数可以删除一个空目录。空目录是只包含.和..的目录。
- int rmdir(const char* pathname);
- 如果调用此函数是目录的链接计数成为0，并且没有其他的进程打开此目录，则释放由该目录占用的空间。如果在链接计数为0时，有若干个进程打开了此目录，则在此函数返回前删除最后一个链接及.和..项。另外，此目录不能在创建新的文件。但是在最后一个进程关闭前不释放此目录。

读目录

- 在Linux系统中，对某个目录具有访问权限的任一用户可读该目录，但是，为了防止文件系统产生混乱，只有内核才能写目录。
- DIR* opendir(const char* pathname);
- //打开目录　失败返回NULL
- struct dirent * readdir(DIR* dir);//读取目录第一项
- void rewinddir(DIR* dp);//重设目录位移
- void closedir(DIR* dp);//关闭目录
- long telldir(DIR* dp);//获取目录当前位置
- void seekdir(DIR* dp,long loc);//将目录移到指定位置

读目录

- DIR结构是一个内部结构，上述6个函数用这个内部结构保持当前正被读的目录的有关信息。由opendir返回的指向DIR结构的指针由另外5个函数使用。opendir执行初始化操作，使第一个readdir读目录中的第一个目录项。目录中各目录的顺序与实现有关。(注意：它们通常并不按字母顺序排序)。

- example：编写一个程序，要求读出指定目录的文件。