Linux 文件与I/O操作

Andrew Huang <bluedrum@163.com>

课程内容

- 系统调用
- 底层库函数
- 标准库函数
- 目录与文件维护

系统调用

- Linux 大部分的系统功能是通过系统调用(System Call)来实现的.如open,send之类.
- 这些函数在C程序调用起来跟标准C库函数(printf…)非常类似.但是实现机制完全不同.
- 库函数仍然是运行在Linux 用户空间程序.很多时候内部会调用系统调用.
- 但系统调用是内核实现的.在C库封装成函数.但通过系统软中断进行调用.
  - 用time命令测试时间,系统时间实际就是系统调用时间累积
    - **time ./demo1**
  - 用strace 可以跟踪一种程序系统调用使用情况
    - **strace ./demo1** #不需要调试信息

两者关系

- 可以参考C库函数malloc与系统调用sbrk的关系

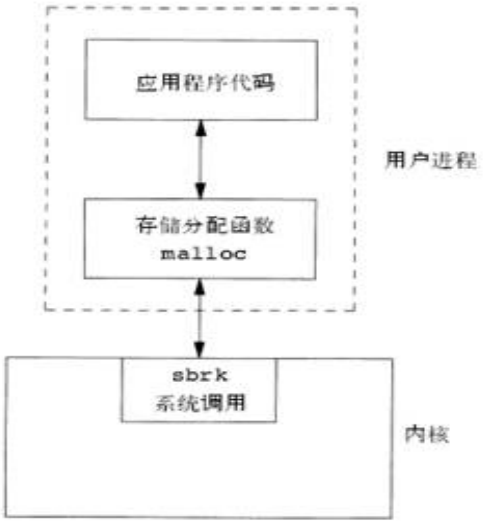

图1-2 malloc函数和sbrk系统调用

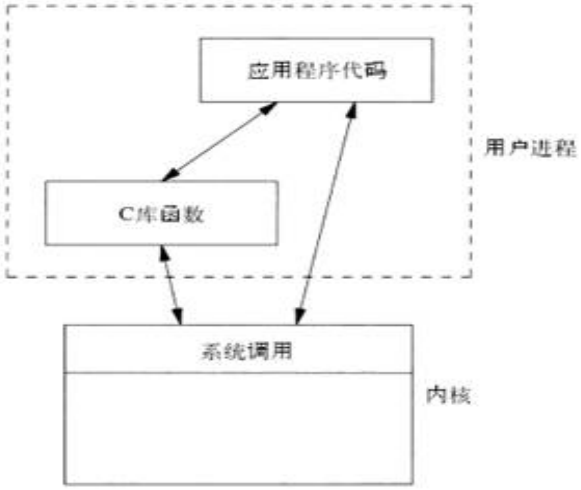

图1-3 C库函数和系统调用之间的差别

常见C标准库函数

- printf,getch,scanf
- strcpy,strcmp,strlen
- memcpy,memcmp,memset
- fopen,fwrite,fread

常见系统调用函数

- 进程控制:fork(),waitpid()
- 文件控制open(),write()
- 网络收发函数 socket(),bind(),send(),write()
- 权限控制 **access()**
- 标准C的函数,应该在MSDN和Linux下的man都能同时查找联机帮助
  - 并且声明定义在stdlib.h当中
- 而Linux系统调用只能用man查找相应帮助
  - 大部分声明定义在unistd.h当中

文件控制

1. Linux 文件结构

- Linux环境中的文件具有特别重要的意义，因为它们为操作系统服务和设备提供了一个简单而统一的接口.在Linux中,一切（或几乎一切）都是文件。
- 通常程序完全可以像使用文件那样使用磁盘文件、串行口、打印机和其他设备。
- 大多数情况下，你只需要使用五个基本的函数——open、close、read、write和ioctl
- Linux中的任何事物都可以用一个文件代表，或者可以通过特殊的文件进行操作。
- 一些特殊文件
  - 目录
  - 设备文件
  - /dev/console
  - /dev/tty
  - /dev/null

2. 底层库函数

- Linux 在底层实现一整套处理文件函数.
  - 这一些函数能处理普通文件,网络socket文件,设备文件等
  - 全部是系统调用实现的函数
- 文件处理函数
  - open – 打开或创建一个文件
  - creat – 建立一个空文件
  - close – 关闭一个文件
  - read – 从文件读入数据
  - write – 向文件写入一个数据
  - lseek – 在文件中移动读写位置

- unlink – 删除一个文件
- remove – 删除一个文件本身
- fcntl – 控制一个文件属性

文件描述符

- 值为一个非负整数
- 用于表示一个打开文件
- 在内核空间被引用,并且由系统调用(open)所创建
- read,write使用文件描述符
- 内核缺省打开三个文件描述符
  - 1-标准输出
  - 2-错误输出
  - 0-标准输入

1)open –-- 打开或创建一个文件

	open（打开文件）
相关函数	read，write，fcntl，close，link，stat，umask，unlink，fopen
包含文件	#include<sys/types.h> #include<sys/stat.h> #include<fcntl.h>
定义函数	int open(const char * pathname, int flags); int open(const char * pathname,int flags, mode_t mode);
函数说明	参数pathname 指向欲打开的文件路径字符串。 flags 标志位,参见下一页
返回值	若所有欲核查的权限都通过了检查则返回0 值，表示成功，只要有一个权限被禁止则返回-1。

范例	#include<unistd.h>
	#include<sys/types.h>
	#include<sys/stat.h>
	#include<fcntl.h>
	main()
	{
	int fd,size;
	char s [ ]=”Linux Programmer!\n”,buffer[80];
	fd=open(“/tmp/temp”,O_WRONLY|O_CREAT);
	write(fd,s,sizeof(s));
	close(fd);
	fd=open(“/tmp/temp”,O_RDONLY);
	size=read(fd,buffer,sizeof(buffer));

	close(fd);
	printf("%s",buffer);
	}

open 的标志位

- flags可以去下面的一个值或者是几个值的组合.
  - O_WRONLY 以只写方式打开文件
  - O_RDONLY 以只读方式打开文件
  - O_RDWR 以可读写方式打开文件。上述三种旗标是互斥的，也就是不可同时使用，但可与下列的旗标利用OR(|)运算符组合。
  - O_CREAT 若欲打开的文件不存在则自动建立该文件。
  - O_EXCL 如果O_CREAT 也被设置，此指令会去检查文件是否存在。文件若不存在则建立该文件，否则将导致打开文件错误。此外，若O_CREAT与O_EXCL同时设置，并且欲打开的文件为符号连接，则会打开文件失败。
  - O_TRUNC 若文件存在并且以可写的方式打开时，此旗标会令文件长度清为0，而原来存于该文件的资料也会消失。
  - O_APPEND 当读写文件时会从文件尾开始移动，也就是所写入的数据会以附加的方式加入到文件后面。
  - O_NONBLOCK 以不可阻断的方式打开文件，也就是无论有无数据读取或等待，都会立即返回进程之中。

多进程open同一个文件

- open的打开的文件描述符,只在同一进程内是唯一的.换句话说,不同程序打开同名文件将会产生不同的描述符.
- 不同进程写同一个文件会产生互相覆盖情况,大部分情况是无法预知.这是相当危险的情况,一般要加入互锁和进程间通讯来防止这种情况发生.

open mode标志位情况

- 如果使用了O_CREATE标志,那么我们要使用open的第二种形式.,mode用来表示文件的访问权限. (sys/stat.h.中定义)
  - S_IRUSR 用户可以读 S_IWUSR 用户可以写
  - S_IXUSR 用户可以执行 S_IRWXU 用户可以读写执行
  - S_IRGRP 组可以读 S_IWGRP 组可以写
  - S_IXGRP 组可以执行 S_IRWXG 组可以读写执行

- S_IROTH 其他人可以读 S_IWOTH 其他人可以写
- S_IXOTH 其他人可以执行 S_IRWXO 其他人可以读写执行
- S_ISUID 设置用户执行ID S_ISGID 设置组的执行ID

- 我们也可以用数字来代表各个位的标志.Linux总共用5个数字来表示文件的各种权限. 00000.第一位表示设置用户ID.第二位表示设置组ID,第三位表示用户自己的权限位,第四位表示组的权限,最后一位表示其他人的权限.

open返回值

- 错误代码
  - 成功打开文件返回文件描述符,否则返回一个负数
  - EEXIST 参数pathname 所指的文件已存在，却使用了O_CREAT和O_EXCL标志。
  - EACCESS 参数pathname所指的文件不符合所要求测试的权限。
  - EROFS 欲测试写入权限的文件存在于只读文件系统内。
  - EFAULT 参数pathname指针超出可存取内存空间。
  - EINVAL 参数mode 不正确。
  - ENAMETOOLONG 参数pathname太长。
  - ENOTDIR 参数pathname不是目录。
  - ENOMEM 核心内存不足。
  - ELOOP 参数pathname有过多符号连接问题。
  - EIO I/O 存取错误。

2)close – 关闭一个文件

	close（关闭文件）
相关函数	open，fcntl，shutdown，unlink，fclose
表头文件	#include<unistd.h>
定义函数	int close(int fd);
函数说明	当使用完文件后若已不再需要则可使用 close()关闭该文件，二 close()会让数据写回磁盘，并释放该文件所占用的资源。参数 fd 为先前由 open()或 creat()所返回的文件描述词。
返回值	若文件顺利关闭则返回 0，发生错误时返回-1。
错误代码	EBADF 参数 fd 非有效的文件描述词或该文件已关闭。
附加说明	虽然在进程结束时，系统会自动关闭已打开的文件，但仍建议自行关闭文件，并确实检查返回值。

范例	参考 open()

close系统调用说明

- close调用终止一个文件描述符fildes与其对应文件之间的关联。文件描述符被释放并能够重新使用。close调用成功就返回0，出错就返回-1。
- 有时检查close调用的返回结果十分重要。有的文件系统，特别是网络文件系统，可能不会在关闭文件之前报告文件写操作中出现的错误，因为执行写操作时，数据可能未被确认写入。
- 运行中的程序能够一次打开的文件数目是有限制的。这个限制由头文件limits.h中的OPEN_MAX常数定义，它会随着系统的不同而不同，但POSIX规范要求它至少要为16。这个限制本身还会受到本地系统全局性限制的影响。

3)write – 向文件写入一个数据

	write（将数据写入已打开的文件内）
相关函数	open，read，fcntl，close，lseek，sync，fsync，fwrite
表头文件	#include<unistd.h>
定义函数	ssize_t write (int fd,const void * buf,size_t count);
函数说明	write()会把参数 buf 所指的内存写入 count 个字节到参数 fd 所指的文件内。当然，文件读写位置也会随之移动。
返回值	如果顺利 write()会返回实际写入的字节数。当有错误发生时则返回-1，错误代码存入 errno 中。
错误代码	EINTR 此调用被信号所中断。 EAGAIN 当使用不可阻断 I/O 时（O_NONBLOCK），若无数据可读取则返回此值。 EADF 参数 fd 非有效的文件描述词，或该文件已关闭。
范例	请参考 open（）。

write系统调用 说明

- 系统调用**write**的作用是，把缓冲区**buf**的前**nbytes**个字节写入与文件描述符**fildes**关联的文件中。它返回实际写入的字节数。如果文件描述符有错或者底层的设备驱动程序对数据块长度比较敏感，该返回值可能会小于**nbytes**。如果这个函数的返回值是**0** ，就表示未写出任何数据，如果是**-1**，就表示在**write**调用中出现了错误，对应的错误代码保存在全局变量**errno**里面。
  - 参见simple_write.c
  - write可能会报告说它写入的字节比你要求的少。这并不一定是个错误。在程序中，你需要检查errno以发现错误，然后再次调用write写入剩余的数据。
- 当write写入成功后,当前写入指针会自动移到所写入最后一个字节下一个位置,相当于自动调用一次lseek(fd,count, SEEK_CUR);**这样设计主要是为了方便了连续写入方便**.

read 从文件读出数据

	read（由已打开的文件读取数据）
相关函数	readdir，write，fcntl，close，lseek，readlink，fread
表头文件	#include<unistd.h>
定义函数	ssize_t read(int fd,void * buf ,size_t count);
函数说明	read()会把参数 fd 所指的文件传送 count 个字节到 buf 指针所指的内存中。若参数 count 为 0，则 read()不会有作用并返回 0。返回值为实际读取到的字节数，如果返回 0，表示已到达文件尾或是无可读取的数据，此外文件读写位置会随读取到的字节移动。
附加说明	如果顺利 read()会返回实际读到的字节数，最好能将返回值与参数 count 作比较，若返回的字节数比要求读取的字节数少，则有可能读到了文件尾、从管道(pipe)或终端机读取，或者是 read()被信号中断了读取动作。当有错 误发生时则返回-1，错误代码存入 errno 中，而文件读写位置则无法预期。
错误代码	EINTR 此调用被信号所中断。 EAGAIN 当使用不可阻断 I/O 时（O_NONBLOCK），若无数据可读取则返回此值。 EBADF 参数 fd 非有效的文件描述词，或该文件已关闭。
范例	参考 open（）。

read系统调用说明

- 系统调用read的作用是从与文件描述符fildes相关联的文件里读入nbytes个字节的数据，并把它们放到数据区buf中。它返回实际读入的字节数，它可能会小于请求的字节数。如果read调用返回0，就表示未读入任何数据，已到达了文件尾。同样，如果是-1，就表示read调用出现了错误。
- 参见simple_read.c
- 当read读取成功后,当前写入指针会自动移到所写入最后一个字节下一个位置,相当于自动调用一次lseek(fd,count, SEEK_CUR);**这样设计主要是为了方便了连续读入方便.**

```
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
main()
{
int fd,size;
char s [ ]="Linux Programmer!\n",buffer[80];
fd=open("/tmp/temp",O_WRONLY|O_CREAT);
write(fd,s,sizeof(s));
close(fd);
fd=open("/tmp/temp",O_RDONLY);
size=read(fd,buffer,sizeof(buffer));
close(fd);
printf("%s",buffer);
}
```

4)lseek 移动文件指针

	lseek（移动文件的读写位置）
相关函数	dup，open，fseek
表头文件	#include<sys/types.h> #include<unistd.h>
定义函数	off_t lseek(int fildes,off_t offset ,int whence);
函数说明	每一个已打开的文件都有一个读写位置，当打开文件时通常其读写位置是指向文件开头，若是以附加的方式打开文件(如 O_APPEND)，则读写位置会指向文 件尾。当 read()或 write()时，读写位置会随之增加，lseek()便是用来控制该文件的读写位置。参数 fildes 为已打开的文件描述词，参数 offset 为根据参数 whence 来移动读写位置的位移数。
参数	whence 为下列其中一种: SEEK_SET 参数 offset 即为新的读写位置。 SEEK_CUR 以目前的读写位置往后增加 offset 个位移量。 SEEK_END 将读写位置指向文件尾后再增加 offset 个位移量。 当 whence 值为 SEEK_CUR 或 SEEK_END 时，参数 offet 允许负值的出现。 下列是教特别的使用方式: 1) 欲将读写位置移到文件开头时:lseek（int fildes,0,SEEK_SET）; 2) 欲将读写位置移到文件尾时:lseek（int fildes，0,SEEK_END）; 3) 想要取得目前文件位置时:lseek（int fildes，0,SEEK_CUR）;
返回值	当调用成功时则返回目前的读写位置，也就是距离文件开头多少个字节。若有错误则返回-1，errno 会存放错误代码。

附加说明	Linux 系统不允许 lseek（）对 tty 装置作用，此项动作会令 lseek（）返回 ESPIPE。

lseek特殊用法

- 快速判断文件当前位置
  - off_t curpos;
  - currpos = lseek(fd, 0, SEEK_CUR);
- lseek可以在普通文件中移动读写指针以外,可以在输入流,输出流中移动.但是不是能在管道和先进先出文件中移动
- 因此可以快速判断当前文件是否为管道
  - currpos = lseek(fd, 0, SEEK_CUR);
  - 如果(currpos = –1) &&(errno = EPIPE)那fd对应是一个管道文件

lseek实例

- 在键盘输入中移动.
  - **a.out < /etc/motd** #普通文件应该是成功
  - **cat /etc/motd | a.out** #管道文件失败
  - **a.out < /var/spool/cron/FIFO** #FIFO文件失败

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

int main(void)
{
   if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
            printf("cannot seek\n");
   else
            printf("seek OK\n");
   exit(0);
}
```

lseek位置参数

- 对于一个普通文件,lseek的位置参数必须是一个非负数
- 对于特殊文件,lseek的位置参数可以为负数
- lseek只是把当前读写位置记录在内核之中,并没有产生I/O操作.
- 当lseek的位置值超过文件本身尺寸,会产生什么后果?
  - 这样运行是可以的,并且不会出错.
  - 这样会在文件尾部新增pos-size大小的内容,每个位的值为0,一般称为文件空洞(file hole)
  - 这是一个最简单扩大文件到指定尺寸的方法.

lseek,创建文件空洞

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

char	buf1[] = "abcdefghij";
char	buf2[] = "ABCDEFGHIJ";

int main(void)
{
	int	fd;

	if ( (fd = creat("file.hole", FILE_MODE)) < 0)
		err_sys("creat error");
		  if (write(fd, buf1, 10) != 10)
		err_sys("buf1 write error");
	/* offset now = 10 */

	if (lseek(fd, 40, SEEK_SET) == -1)
		err_sys("lseek error");
	/* offset now = 40 */

	if (write(fd, buf2, 10) != 10)
		err_sys("buf2 write error");
	/* offset now = 50 */

		  close(fd);
	exit(0);
}
```

5)底层I/O函数:一个文件拷贝程序

- copy_system.c，
- 改进后的块拷贝 copy_block. c
- 用time测试带缓冲与不带缓冲性能差别,

```
[root@TecherHost linux_c]# gcc copy_block.c -o copy_block
[root@TecherHost linux_c]# ./copy_block
[root@TecherHost linux_c]# time ./copy_system

real    0m9.283s
user    0m2.560s
sys     0m6.630s
[root@TecherHost linux_c]# time ./copy_block

real    0m0.014s
user    0m0.000s
sys     0m0.020s
[root@TecherHost linux_c]#
```

O_EXCL的用法

- 利用O_EXCL进行双重打开

```
fd = open(filename,O_RDWR| O_CREAT |O_EXCL);
  if(fd == -1)
  {
    if(errno == EEXIST)
      {
          printf("EEXIST\n");
           fd = open(filename,O_RDWR);
       }

    if(fd == -1)
     {
        printf("failure!\n");
        exit(-1);
    }
  }
```

3. Linux 一些特殊文件

- 除了普通文件(regular file)以外,Linux 还存在一些特殊文件.如
  - 目录文件(directory file)。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。对一个目录文件具有读许可权的任一进程都可以读该目录的内容，但只有内核可以写目录文件。但可以通一些特殊调用来处理.
  - 字符特殊文件(character special file)。块特殊文件(block special file)。
  - FIFO。这种文件用于进程间的通信，有时也将其称为命名管道
  - 套接口(socket)。这种文件用于进程间的网络通信。套接口也可用于在一台宿主机上的进程之间的非网络通信
  - 符号连接(symbolic link)。这种文件指向另一个文件

标准文件读写函数

- 在输入输出操作中，直接使用底层系统调用的问题是它们的效率非常低
  - 系统调用会影响系统的性能。与函数调用相比，系统调用的开销要大些，因为在执行系统调用时，Linux必须从用户代码切换到内核代码运行，然后再返回用户代码。减少这种开销的一个好方法是，在程序中尽量减少系统调用的次数，并且让每次系统调用完成尽可能多的工作。例如每次读写大量的数据而不是每次仅读写一个字符。
  - 硬件会对底层系统调用一次所能读写的数据块做出一定的限制。例如，磁带机通常的写操作数据块长度是10k，所以如果所写的数据量不是10k的整数倍，磁带机还是会以10k为单位卷绕磁带，这就在磁带上留下了空隙。
  - 为了给设备和磁盘文件提供更高层的接口，与UNIX一样，Linux发行版提供了一系列的标准函数库。它们是一些由函数构成的集合，你可以把它们包括在自己的程序中去处理那些与设备和文件有关的问题。提供输出缓冲功能的标准I/O库就是一个这样的例子。你可以高效地写任意长度的数据块，库函数则在数据满足数据块长度要求时安排执行底层系统调用。这就极大降低了系统调用的负面影响。

1.标准I/O库

- 标准I/O库及其头文件stdio.h为底层I/O系统调用提供了一个通用的接口 .常见的函数有fopen,fclose等。
- 这个库现在已经成为ANSI标准C的一部分，而我们前面见到的系统调用(open,close)却还不是。
- 标准I/O库提供了许多复杂的函数，用于格式化输出和扫描输入。它还负责满足设备的缓冲需求。
- 在很多方面，使用标准I/O库和使用底层文件描述符类似。你需要先打开一个文件以建立一个访问路径。这个操作的返回值将作为其他I/O库函数的参数。在标准I/O库中，与底层文件描述符对应的对等物叫流（stream），它被实现为指向结构FILE的指针。
- 在启动程序时，有三个文件流是自动打开的。它们是stdin、stdout和stderr。它们都是在stdio.h头文件里定义的，分别代表着标准输入、标准输出和标准错误输出，与底层文件描述符0、1和2相对应。
- 需要学习标准I/O库中的库函数
  - fopen、fclose
  - fread、fwrite
  - fflush
  - fseek
  - fgetc、getc、getchar
  - fputc、putc、putchar
  - fgets、gets
  - printf、fprintf和sprintf
  - scanf、fscanf和sscanf

1)标准I/O库: fopen

- fopen函数

- fopen库函数类似于底层的open系统调用。它主要用于文件和终端的输入输出。如果需要对设备的行为进行明确的控制，那最好使用底层系统调用，因为这可以避免用库函数带来的一些非预期的潜在副作用，如输入/输出缓冲。
  - **FILE * fopen(const char * filename,const char * mode);**
  - fopen打开由filename参数指定的文件，并把它与一个文件流关联起来。mode参数指定文件的打开方式
  - “r”或“rb”：以只读方式打开
  - “w”或“wb”：以写方式打开，并把文件长度截短为零。
  - “a”或“ab”：以写方式打开，新内容追加在文件尾。
  - “r+”或“rb+”或“r+b”：以修改方式打开（读和写）。
  - “w+”或“wb+”或“w+b”：以修改方式打开，并把文件长度截短为零。
  - a+”或“ab+”或“a+b”：以修改方式打开，新内容追加在文件尾。
  - 字母b表示文件是一个二进制文件而不是文本文件。
- fopen在成功时返回一个非空的FILE *指针。失败时返回NULL值，NULL值的定义在头文件stdio.h里。

2)标准I/O库: fread

- fread库函数的作用是从一个文件流里读取数据。数据从文件流stream读到由ptr指定的数据缓冲区里。fread和fwrite都是对数据记录进行操作的，size参数指定每个数据记录的长度，计数器nitems给出要传输的记录个数。它的返回值是成功地读到数据缓冲区里的记录个数（而不是字节数）。当到达文件尾时，它的返回值可能会小于nitems，甚至可以是零。
  - **size_t fread(void * ptr,size_t size,size_t nitems,FILE * stream);**
- 对所有向缓冲区里写数据的标准I/O函数来说，为数据分配空间和检查错误的工作是程序员的责任

3)标准I/O库: fwrite

- fwrite库调用与fread有相似的接口。它从指定的数据缓冲区里取出数据记录，并把它们写到输出流中。它的返回值是成功写入的记录个数。
  - **size_t fwirte (void * ptr,size_t size,size_t nitems,FILE * stream);**

4)标准I/O库: fclose

- fclose库函数关闭指定的文件流stream，使所有尚未写出的数据都写出。因为stdio库会对数据进行缓冲，所以使用fclose是很重要的。如果程序需要确保数据已经全部写出，就应该调用fclose函数。虽然当程序正常结束时，会自动对所有还打开的文件流调用fclose函数，但这样做就没有机会检查由fclose报告的错误了。与文件描述符一样，可用文件流的数目也是有限制的。这个限制由头文件stdio.h中的FOPEN_MAX常量定义，最小为8。
  - **int fclose(FILE * stream);**

5)标准I/O库: fflush

- fflush库函数的作用是把文件流里的所有未写出数据立刻写出。例如，你可以用这个函数来确保在试图读入一个用户响应之前，先向终端送出一个交互提示符。使用这个函数还可以确保在程序继续执行之前重要的数据都已经被写到磁盘上。有时在调试程序时，还可以用它来确定程序是正在写数据而不是被挂起了。注意，调用fclose函数隐含执行了一

次flush操作，所以不必在fclose之前调用fflush。

- **int fflush(FILE * stream);**

6)标准I/O库: fseek

- fseek函数是与lseek系统调用等价的文件流函数。它在文件流里为下一次读写操作指定位置。offset和whence参数的含义和取值与前面的lseek系统调用完全一样。但lseek返回的是一个off_t数值，而fseek返回的是一个整数：0表示成功，-1表示失败并设置errno指出错误。
  - **int fseek(FILE * stream,long int offset,int whence);**

7)标准I/O库: fgets和gets函数

- fgets函数从输入文件流stream里读取一个字符串。它把读到的字符写到s指向的字符串里，直到出现下面这几种情况之一：遇到换行符，已经传输了n-l个字符，或者到达文件尾。它会把遇到的换行符也传递到接收字符串里去，再加上一个表示结尾的空字节\0。一次调用最多只能传输n-1个字符，因为它必须把空字节加上以结束字符串。
  - **char * fgets(char * s,int n,FILE * stream);**
  - **char * gets(char * s);**
  - 当它成功完成时，它返回一个指向字符串s的指针。如果文件流已经到达文件尾，fgets会设置这个文件流的EOF标识并返回一个空指针。如果出现读错误，fgets返回一个空指针并设置errno给出错误的类型。
  - gets函数类似于fgets，只不过它从标准输入读取数据并丢弃遇到的换行符。它在接收字符串的尾部加上一个null字节。
  - gets对传输字符的个数并没有限制，所以它可能会溢出自己的传输缓冲区。因此应该避免使用它并用fgets来代替。因特网上的许多安全问题都可以追溯到在程序中使用了可能造成各种缓冲区溢出的函数，gets就是一个这样的函数，所以千万要小心！

2 格式化输入和输出

- 包括向一个文件流输出数据的printf系列函数和从一个文件流读取数据的scanf系列函数。
- printf、fprintf和sprintf函数
  - printf系列函数能够对各种不同类型的参数进行格式编排和输出。每个参数在输出流中的表示形式是由格式参数format控制的，它是一个包含普通的可打印字符和称为“转换控制符”代码的字符串，转换控制符规定了其余的参数应该以何种方式被输出到何种地方。
  - printf函数把自己的输出送到标准输出。fprintf函数把自己的输出送到一个指定的文件流。sprintf函数把自己的输出和一个结尾空字符写到作为参数传递过来的字符串s里。这个字符串必须足够大以容纳所有的输出数据。
- 常用的转换控制符
  - %d：%i：以十进制格式输出一个整数。
  - %o：%x：以八进制或十六进制格式输出一个整数。
  - %c：输出一个字符。
  - %s：输出一个字符串。
  - %f：输出一个（单精度）浮点数。
  - %e：以科学计数法格式输出一个双精度浮点数。
  - **%g**：以一般格式输出一个双精度浮点数。

- 你可以利用字段限定符对数据的输出格式做进一步的控制。它是对转换控制符的扩展，能够对输出数据的间隔进行控制。一个常见用法是设置浮点数的小数位数或设置字符串两端的空格数
- 字段限定符是转换控制符里紧跟在%字符后面的数字。
- 表中的所有示例都输出到一个10个字符宽的区域里。注意：负值的字段宽度表示数据在该字段里以左对齐的格式输出。可变字段宽度用一个星号（*）来表示。在这种情况下，下一个参数用来表示字段宽。%字符后面以0开头表示数据前面要用数字0填充。根据POSIX规范的要求，printf不对数据字段进行截断，而是扩充数据字段以适应数据的宽度。因此，如果想打印一个比字段宽长的字符串，数据字段会加宽。

格　式	参　数	输出
%10s	"Hello"	Hello
%-10s	"Hello"	Hello
%10d	1234	1234
%-10d	1234	1234
%010d	1234	0000001234
%10.4f	12.34	12.3400
%*s	10," Hello"	Hello

scanf、fscanf和sscanf函数

- scanf系列函数的工作方式与printf系列函数很相似，只是前者的作用是从一个文件流里读取数据，并把数据值放到传递过来的指针参数指向的地址处的变量中。它们也使用一个格式字符串来控制输入数据的转换，其工作原理和许多转换控制符都与printf系列函数的情况一致。
- scanf函数读入的值将保存到对应的变量里去，这些变量的类型必须正确，并且它们必须精确匹配格式字符串。否则，内存就可能会发生冲突，从而使程序崩溃。编译器是不会对此做出错误提示的.
- scanf系列函数的format格式字符串里同时包含着普通字符和转换控制符，就像printf函数中一样。但那些普通字符是用来指定在输入数据里必须出现的字符。
- scanf转换控制符
  - %d：读取一个十进制整数。
  - %o、%x：读取一个八进制或十六进制整数。
  - %f、%e、%g：读取一个浮点数。
  - %c：读取一个字符（不会忽略空格）。
  - %s：读取一个字符串。
  - %[]：读取一个字符集合（见下面的说明）。
  - %%：读取一个%字符。
- 类似于printf，scanf的转换控制符里也可以加上对输入数据字段宽度的限制。长度限定符（h对应于短整数，l对应于长整数）指明接收参数的长度是否比默认情况更短或更长。

也就是说，%hd表示要读入一个短整数，%ld表示要读入一个长整数，而%lg表示要读入一个双精度浮点数。

- 以星号（*）打头的控制符表示对应位置上的输入数据将被忽略，也就是说，这个数据不会被保存，因此不需要使用一个变量来接收它。
- 我们使用%c控制符从输入中读取一个字符。它不会跳过起始的空白字符。
- 我们使用%s控制符来扫描字符串，但使用时必须小心。它会跳过起始的空白字符，但会在字符串里出现的第一个空白字符处停下来，所以，我们最好还是用它来读取单词而不是一般意义上的字符串。此外，如果没有使用字段宽限定符，它能够读取的字符串的长度是没有限制的，所以接收字符串必须有足够的空间来容纳输入流中可能的最长字符串。较好的选择是使用一个字段限制符，或者结合使用fgets和sscanf，从输入中读入一行数据，再对它进行扫描。这样可以避免可能被恶意用户利用而造成缓冲区溢出的情况。
- 我们使用%[]控制符读取一个由一个字符集合中的字符构成的字符串。格式字符串%[A-Z]将读取一个由大写字母构成的字符串。如果字符集中的第一个字符是^，就表示将读取一个由不属于该字符集合中的字符构成的字符串。因此读取一个其中带空格的字符串，并且在遇到第一个逗号时停止，可以用%[^,]。
- scanf函数的返回值是它成功读取的数据项个数，如果在读第一个数据项时失败了，返回值就将是零。如果在匹配第一个数据项之前就已经到达了输入的结尾，就会返回EOF。如果文件流发生读错误，流错误标志就会被设置并且错误变量errno将被设置以指明错误类型。
- 对scanf系列函数的评价并不高
  - 从历史来看，它们的具体实现都有安全漏洞
  - 它们的使用不够灵活。
  - 使用它们编写的代码不容易看出究竟要读取什么。
- 尽量使用其他函数，如fread或fgets来读取输入行，再用字符串函数把输入分割成需要的数据项。
  - 或者放弃键盘输入,采用命令行或管道进行输入

3. 文件流错误

- 为了表明错误，许多stdio库函数会返回一个超出范围的值，比如空指针或EOF常数。此时，错误由外部变量errno指出
- 也可以通过检查文件流的状态来确定是否发生了错误，或者是否到达了文件尾。
  - ferror函数测试一个文件流的错误标识，如果该标识被设置就返回一个非零值，否则返回零。
  - feof函数测试一个文件流的文件尾标识，如果该标识被设置就返回非零值，否则返回零。

4. 文件流和文件描述符

- 每个文件流都和一个底层文件描述符相关联。你可以把底层的输入输出操作与高层的文件流操作混在一起使用，但一般来说这并不是一个明智的做法，因为数据缓冲的后果难以预料。
  - 以通过调用fileno函数来确定文件流使用的是哪个底层文件描述符。
    - **int fileno(FILE * stream);**
  - 我们可以通过调用fdopen函数在一个已打开的文件描述符上创建一个新的文件流。实质上，这个函数的作用是为一个已经打开的文件描述符提供stdio缓冲区

- **FILE * fdopen(int fildes,char * mode);**
- fdopen函数的操作方式与fopen函数是一样的，只是前者的参数不是一个文件名，而是一个底层的文件描述符。当我们已经通过open系统调用创建了一个文件（可能是出于为了更好地控制其访问权限的目的），但又想通过文件流来对它进行写操作时，这个函数就很有用了。

5. 标准I/O测试样例

- Copy_stdio.c 用标准I/O实现的拷贝文件函数
  - 注意**fread**本身内部自动有缓冲区.所以调整外部缓冲区对读写的速度影响不大.

文件特殊处理

- stat,fstat,lstat取得文件详细情况
- access文件权限判断

1. stat(), fstat(), lstat()

- #include <sys/types.h>
- #include <sys/stat.h>
- **int stat(char *pathname, struct stat *buf);**
  - 返回名字为pathname文件的详细情况
- **int fstat(int filedes, struct stat *buf);**
  - 返回已经打开文件的详细情况
- **int lstat(char *pathname, struct stat *buf);**
  - 可以检查普通文件和符号链接对应文件pathname的详细情况

fstate

	fstat（由文件描述符取得文件状态）
相关函数	stat，lstat，chmod，chown，readlink，utime
表头文件	#include<sys/stat.h> #include<unistd.h>
定义函数	int fstat(int fildes,struct stat *buf);
函数说明	fstat()用来将参数 fildes 所指的文件状态，复制到参数 buf 所指的结构中(struct stat)。Fstat()与 stat()作用完全相同，不同处在于传入的参数为已打开的文件描述词。详细内容请参考 stat()。
返回值	执行成功则返回 0，失败返回-1，错误代码存于 errno。

范例	#include<sys/stat.h>
	#include<unistd.h>
	#include<fcntk.h>
	main()
	{
	struct stat buf;
	int fd;
	fd = open (“/etc/passwd”,O_RDONLY);
	fstat(fd,&buf);
	printf(“/etc/passwd file size +%d\n “,buf.st_size);
	}

执行	/etc/passwd file size = 705

stat结构

```
struct stat
{
mode_t st_mode; /* file type & mode */
ino_t     st_ino; /* i-node # */
dev_t     st_dev;     /* device # */
dev_t     st_rdev;    /* device # for special file */
nlink_t   st_nlink;  /* # of links */
uid_t     st_uid;     /* owner UID (user ID) */
gid_t     st_gid;     /* owner GID (group ID) */
off_t     st_size;    /* size in bytes, for regular files */
time_t st_atime;   /* time of last access */
time_t st_mtime;  /* time of last modification */
time_t   st_ctime;    /* time of last file status change */
long      st_blksize; /* best I/O block size */
long      st_blocks; /* # 512-byte blocks allocated */
};
```

文件类型

l Stat. st_mode包含了文件类型信息,因此可以用下列宏来取得文件类型

Macro	**Type of file**
S_ISREG()	Regular file
S_ISDIR()	Directory file
S_ISCHR()	Character special file
S_ISBLK()	Block special file
S_ISFIFO()	Pipe or FIFO
S_ISLNK()	Symbolic link (not in POSIX.1 or SVR4)
S_ISSOCK()	Socket (not in POSIX.1 or SVR4)

判断文件类型

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

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;

     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";
#ifdef      S_ISLNK
        else if (S_ISLNK(buf.st_mode))
         ptr = "symbolic link";
#endif
#ifdef      S_ISSOCK
        else if (S_ISSOCK(buf.st_mode))    ptr = "socket";
#endif
        else    ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

文件尺寸

- stat. st_size记录是文件尺寸
  - 如果为0,表示是空文件
- 如果文件名是目录名,st_size则是16或512的除数
- fstate使用链接文件st_size是被链接文件的尺寸
- lstate的链接文件st_size是链接文件本身

文件处理时间

Field	Description	Example	ls option
st_atime	文件访问最后时间	read	-u
st_mtime	文件最后修改的时间	write	default
st_ctime	i-node 结点最后修改时间	chmod, chown	-c

Linux文件权限

- 9个权限位
  - rwxr-xr-- (read, write, execute)
- 三个组别
  - User (or owner)
  - Group
  - Other (or world)
- “chmod” 改变权限

文件权限判断

- 文件权限位和类型也保留在st_mode当中.可以用位与的方法来取得权限位

```
if((buf.st_mode & S_IRUSR) == S_IRUSR)
                        printf("r");
                else
                        printf("-");

if((buf.st_mode & S_IWUSR) == S_IWUSR)
                        printf("w");
                else
                        printf("-");

if((buf.st_mode & S_IXUSR) == S_IXUSR)
                        printf("x");
                else
                        printf("-");
```

用系统调用设置权限

l 在**open(), creat(), and umask()**权限设置

st_mode mask	**Meaning**	**对应值**
S_IRUSR	user-read	00400
S_IWUSR	user-write	00200
S_IXUSR	user-execute	00100
S_IRGRP	group-read	00040
S_IWGRP	group-write	00020
S_IXGRP	group-execute	00010
S_IROTH	other-read	00004
S_IWOTH	other-write	00002
S_IXOTH	other-execute	00001

2. access 当前进程是否有某个权限

	access（判断是否具有存取文件的权限）
相关函数	stat，open，chmod，chown，setuid，setgid
表头文件	#include<unistd.h>
定义函数	int access(const char * pathname,int mode);
函数说明	access()会检查是否可以读/写某一已存在的文件。参数 mode 有几种情况组合，R_OK，W_OK，X_OK 和 F_OK。R_OK，W_OK 与 X_OK 用来检查文件是否具有读取、写入和执行的权限。F_OK 则是用来判断该文件是否存在。由于 access()只作 权限的核查，并不理会文件形态或文件内容，因此，如果一目录表示为“可写入”，表示可以在该目录中建立新文件等操作，而非意味此目录可以被当做文件处理。 例如，你会发现 DOS 的文件都具有“可执行”权限，但用 execve()执行时则会失败。
返回值	若所有欲查核的权限都通过了检查则返回 0 值，表示成功，只要有一权限被禁止则返回-1。
错误代码	EACCESS 参数 pathname 所指定的文件不符合所要求测试的权限。 EROFS 欲测试写入权限的文件存在于只读文件系统内。 EFAULT 参数 pathname 指针超出可存取内存空间。 EINVAL 参数 mode 不正确。 ENAMETOOLONG 参数 pathname 太长。 ENOTDIR 参数 pathname 为一目录。 ENOMEM 核心内存不足 ELOOP 参数 pathname 有过多符号连接问题。 EIO I/O 存取错误。
附加说明	使用 access()作用户认证方面的判断要特别小心，例如在 access()后再做 open()的空文件可能会造成系统安全上的问题。

access mode参数取值

mode	**Description**
R_OK	有读权限
W_OK	写权限
X_OK	执行权限
F_OK	测试文件是否存在

access实例

```
int main(int argc, char *argv[])
{
        if (argc != 2)
                err_quit("usage: a.out <pathname>");

        if (access(argv[1], R_OK) < 0)
                err_ret("access error for %s", argv[1]);
        else
                printf("read access OK\n");

        if (open(argv[1], O_RDONLY) < 0)
                err_ret("open error for %s", argv[1]);
        else
                printf("open for reading OK\n");

        exit(0);
}
```

3.chmod改变文件权限

	chmod（改变文件的权限）
相关函数	fchmod，stat，open，chown
表头文件	#include<sys/types.h> #include<sys/stat.h>
定义函数	int chmod(const char * path,mode_t mode);
函数说明	chmod()会依参数 mode 权限来更改参数 path 指定文件的权限。
参数	mode 有下列数种组合 只有该文件的所有者或有效用户识别码为 0，才可以修改该文件权限。基于系统安全，如果欲将数据写入一执行文件，而该执行文件具有 S_ISUID 或 S_ISGID 权限，则这两个位会被清除。如果一目

	录具有 S_ISUID 位权限，表示在此目录下只有该文件的所有者或 root 可以删除该文件。
返回值	权限改变成功返回 0，失败返回-1，错误原因存于 errno。
错误代码	EPERM 进程的有效用户识别码与欲修改权限的文件拥有者不同，而且也不具 root 权限。 EACCESS 参数 path 所指定的文件无法存取。 EROFS 欲写入权限的文件存在于只读文件系统内。 EFAULT 参数 path 指针超出可存取内存空间。 EINVAL 参数 mode 不正确 ENAMETOOLONG 参数 path 太长 ENOENT 指定的文件不存在 ENOTDIR 参数 path 路径并非一目录 ENOMEM 核心内存不足 ELOOP 参数 path 有过多符号连接问题。 EIO I/O 存取错误

范例	/* 将 /etc/passwd 文件权限设成 S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH */
	#include<sys/types.h>
	#include<sys/stat.h>
	main()
	{
	chmod("/etc/passwd",S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
	}

4. 链接文件

- unlink、link和symlink系统调用
  - 我们可以用unlink系统调用来删除一个文件。
  - unlink系统调用删除一个文件的目录项并减少它的链接数。它在成功时返回0，失败时返回-1。如果你想通过调用这个函数来成功删除文件，就必须拥有该文件所属目录的写和执行权限。
  - 先用open创建一个文件，然后对其调用unlink是某些程序员用来创建临时文件的技巧。这些文件只有在被打开的时候才能被程序使用，当程序退出文件关闭的时候它们就会被自动地删除掉。
  - link系统调用将创建一个指向已有文件path1的硬链接。新目录数据项由path2给出。我们可以通过symlink系统调用以类似的方式创建符号链接。注意：一个文件的符号链接不会像正常（硬）链接那样，它不能防止该文件被删除。
    - **int link (const char * oldpath,const char * newpath);**
    - **int symlink(const char * oldpath,const char * newpath);**

5. 文件和目录的维护

- chdir系统调用和getcwd函数
  - 程序可以像用户在文件系统里漫游那样来浏览目录。就像我们在shell里使用cd命令来切换目录一样，在程序里则可以使用chdir系统调用。
    - **int chdir(const char * path);**
  - 程序可以通过调用getcwd函数来确定自己的当前工作目录。
    - **char * getcwd(char * buf,size_t size);**
    - getcwd函数把当前目录的名字写到给定的缓冲区buf里。如果目录的名字超出了参数size给出的缓冲区长度（一个ERANGE错误），它就返回NULL。如果成功，它返回指针buf。
- remove（删除文件）
  - remove()会删除参数pathname指定的文件。如果参数pathname为一文件，则调用unlink()处理，若参数pathname为一目录，则调用rmdir()来处理。请参考unlink()与rmdir()。
  - **int remove(const char * pathname)**;

目录处理

1. 文件和目录的维护

- mkdir和rmdir系统调用
  - 我们可以使用mkdir和rmdir系统调用来建立和删除目录。
  - mkdir系统调用的作用是创建目录，它相当于mkdir程序。mkdir调用将把参数path作为新建目录的名字。目录的权限由参数mode设定，mode的含义将按open系统调用的O_CREAT选项中的有关定义设置，当然，它还要服从umask的设置情况。
    - **int mkdir(const char * path,mode_t mode);**
  - rmdir系统调用的作用是删除目录，但只有在目录为空时才行。rmdir程序就是用这个系统调用来完成工作的。
    - **int rmdir(const char *path);**

2. 扫描目录

- Linux系统上一个常见问题就是对目录进行扫描，也就是确定一个特定目录下存放的文件。
- 但不同的文件系统结构及其实现方法已经使这种办法没什么可移植性了。现在，一整套标准的库函数已经被开发出来.
- 与目录操作有关的函数在dirent.h头文件中声明。它们把一个名为DIR的结构作为目录操作的基础。被称为“目录流”的指向这个结构的指针（DIR *）被用来完成各种目录操作，其使用方法与文件流（FILE *）非常相似。目录数据项本身在dirent结构中返回，该结构也是在dirent.h头文件里声明的，用户绝不要直接改动DIR结构中的数据字段。

- opendir函数
  - opendir函数的作用是打开一个目录并建立一个目录流。如果成功，它返回一个指向DIR结构的指针，该指针用于读取目录数据项。
  - **DIR * opendir(const char * name);**
  - opendir在失败时会返回一个空指针。注意，目录流用一个底层文件描述符来访问目录本身，所以如果打开的文件过多，opendir可能会失败。
- readdir函数
  - readdir函数将返回一个指针，指针指向的结构里保存着目录流drip中下一个目录项的有关资料。后续的readdir调用将返回后续的目录项。如果发生错误或者到达目录尾，readdir将返回NULL。POSIX兼容的系统在到达目录尾时会返回NULL，但并不改变errno的值，在发生错误时才会设置errno。
  - **struct dirent * readdir(DIR * dir);**
  - 注意，如果在readdir函数扫描目录的同时还有其他进程在该目录里创建或删除文件，readdir将不保证能够列出该目录里的所有文件（和子目录）。
- telldir函数
  - telldir函数的返回值记录着一个目录流里的当前位置。你可以在随后的seekdir调用中利用这个值来重置目录扫描到当前位置。
  - **off_t telldir(DIR *dir);**
- seekdir函数
  - seekdir函数的作用是设置目录流dirp的目录项指针。loc的值用来设置指针位置，它应该通过前一个telldir调用获得。
- closedir函数
  - closedir函数关闭一个目录流并释放与之关联的资源。它在执行成功时返同0，发生错误时返回-1。
  - **Int closedir(DIR *dir);**
- 完整的扫描目录的例子
  - printdir.c

样例:打印目录树

- Printdir.c
  - 绝大部分操作都是在printdir函数里完成的，所以我们重点对它进行说明。在用opendir函数检查完指定目录是否存在后，printdir调用chdir进入指定目录。如果readdir函数返回的数据项不为空，程序就检查该数据项是否是一个目录。如果不是，程序就根据depth的值缩进打印文件数据项的内容。
  - 如果该数据项是一个目录，我们就需要对它进行递归遍历。在跳过.和..数据项（它们分别代表当前目录和上一级目录）后，printdir函数调用自己并再次进入一个同样的处理过程。那它又是如何退出这些循环的呢？一旦while循环完成，chdir(“. . ”)调用把它带回到目录树的上一级，从而可以继续列出上级目录中的清单。调用closedir(dp)关闭目录以确保打开的目录流数目不超出其需要。

3. 用户信息

- Linux运行的每个程序实际上都是被某个用户运行的，因此都有一个关联的UID。
- UID有它自己的类型——uid_t，它定义在头文件sys/types.h中。它通常是一个小整数。
  - getuid函数返回程序关联的UID，它通常是启动程序的用户的UID。

- getlogin函数返回与当前用户关联的登录名

- 密码数据库结构passwd定义在头文件pwd.h中
  - getpwuid（从密码文件中取得指定uid 的数据）
  - getpwnam（从密码文件中取得指定账号的数据） 。

- 样例参见user.c

passwd成员	说　明
char *pw_name	用户登录名
uid_t pw_uid	UID编号
gid_t pw_gid	GID编号
char *pw_dir	用户主目录
char *pw_gecos	用户全名
char *pw_shell	用户默认shell

4. 日志

- 许多应用程序需要记录它们的活动。系统程序经常需要向控制台或日志文件写消息。这些消息可能指示错误、警告或是与系统状态有关的一般信息
- 对一个典型的Linux安装来说，文件/var/log/messages包含所有系统信息，/var/log/mail包含来自邮件系统的其他日志信息，/var/log/debug可能包含调试信息。
- UNIX规范为所有程序提供了一个接口，通过syslog函数来产生日志信息：
- syslog函数向系统的日志工具发送一条日志信息。每条信息都有一个priority参数，该参数是一个严重级别与一个设施值的按位或。严重级别控制日志信息的处理，设施值记录日志信息的来源。
- 我们可以通过setlogmask函数来设置一个日志掩码，并通过它来控制日志信息的优先级。优先级未在日志掩码中置位的后续syslog调用都将被丢弃。例如，你可以通过这个方法关闭LOG_DEBUG消息而不用改变程序主体。
- 严重级别按优先级递减排列

优　先　级	说　明
LOG_EMERG	紧急情况
LOG_ALERT	高优先级故障，例如数据库崩溃
LOG_CRIT	严重错误，例如硬件故障
LOG_ERR	错误
LOG_WARNING	警告
LOG_NOTICE	需要注意的特殊情况
LOG_INFO	一般信息
LOG_DEBUG	调试信息

logopt参数	说　明
LOG_PID	在日志信息中包含进程标识符，这是系统分配给每个进程的一个唯一值
LOG_CONS	如果信息不能被记录到日志文件中，就把它们发送到控制台
LOG_ODELAY	在第一次调用syslog时才打开日志功能
LOG_NDELAY	立即打开日志功能，而不是等到第一次记录日志时

5. 出错处理

- 许多系统调用和函数都会因为各种各样的原因而失败。失败时，它们会设置外部变量errno的值来指明失败的原因
- 许多不同的函数库都把这个变量用做报告错误的标准方法。程序必须在函数报告出错之后立刻检查errno变量，因为它可能被下一个函数调用所覆盖，即使下一个函数自身并没有出错，也可能会覆盖这个变量。
- 错误代码的取值和含义都列在头文件errno.h里 ,常用错误代码有
  - EPERM：操作不允许。
  - ENOENT：文件或目录不存在。
  - EINTR：系统调用被中断。
  - EIO：I/O错误。
  - EBUSY：设备或资源忙。
  - EEX IST：文件存在。
  - EINVA L：无效参数。
  - EMFILE：打开的文件过多。
  - ENODEV：设备不存在。
  - EISDIR：是一个目录。
  - ENOTDIR：不是一个目录。

- 有两个非常有用的函数可以用来报告出现的错误，它们是strerror和perror。
- strerror函数把错误编码映射为一个字符串，该字符串对发生的错误类型进行说明。这在记录错误条件时十分有用。
  - **char * strerror(int errnum);**
- perror函数也把error变量中报告的当前错误映射到一个字符串，并把它输出到标准错误输出流。该字符串的前面先加上参数s（如果s不为空）给出的信息，再加上一个冒号和一个空格。
  - **void perror(const char *s);**

思考题

- open系统调用返回文件描述符和标准C库的fopen打开的FILE两者的相同点和区别是什么?

课堂练习

- 写出两种计算文件尺寸的方法
  - fstat
  - fseek+ftell
- 写一个ln命令的翻版
- 请写一个类似于ls –l [目标] 功能的命令行工具.
  - 根据用户输出目标名来显示文件信息的详细情况,如果目标名是一个目录,则显示目录下所有文件详细信息.如果是一个文件,则显示文件本身的详细信息.如果为空,表示显示当前目录的详情
  - 显示每一个文件详情按下列格式显示
    - **[**类型**:1**个字符**]:[**权限**:9**个字符**] [**拥有者**][**拥有组**] [**文件尺寸**][**最后修改时间**][**文件名**]**
- 写一个测试文件读入速度的小工具
  - 测试文件要求是10M以上大文件,文件名要求用命令行参数传入进来.结果显示在屏幕上
  - 用不同尺寸的buffer(4K,128K,512K,1M,5M) ,分别去读入测试文件
  - 把文件总尺寸除以读入时间,即是测试速度值,每一种buffer的速度值分别显示
  - 用**open,close,read,write**实现

扩展练习

- 请分别用read/fread分别写一个文本文件显示程序,类似于cat readme.txt
- 请编写一个Unix文本文件转Windows文本文件互相转换的工具
  - 文件名由命令行参数输入
  - 注意Unix文本文件最后一行要为空行,回车符为\n,而Windows文本回车符为\r\n
  - 扩展练习:把指定的带通配符的文件名进行转换,如w2u /home/hxy/*.txt 把/home/hxy目录下所有以.txt后缀名全部由windows转成Unix文本
  - 用fgetch把文本文件当成二进制文件来处理.
- 设计一种文本文件格式,用每一行表示一个文件的描述，可以是如下格式:
  - 其中一行示例 dst.out=/home/hxy/in.out 表示把/home/hxy/in.out 文件拷贝到 dst.out 这个文件来
  - 要求文本文件有5行以上，即可以拷5个文件
- 设计一个程序,要求:
  - 设计一个命令行选项 –f file,读取这个配置文件
  - 能从头到尾读取这个配置文件。
  - 按文件的要求拷贝到指定位置
- 扩展设计,如果发现目标文件是可执行程序自己,要求能覆盖自己,并在拷贝后重新启动,
  - 参考方案之一:可用Shell 脚本实现,在程序退出后,用返回值来告诉脚本,是否进行自升级.
- 可参考遍历文本文件演示代码
  - show_text.c