LINUX 输入输出与文件

1. 基于文件指针的文件操作(缓冲)

linux 的文件和文件描述符

linux 中对目录和设备的操作都是文件操作，文件分为普通文件，目录文件，链接文件和设备文件。

普通文件：也称磁盘文件，并且能够进行随机的数据存储(能够自由 seek 定位到某一个位置)；

管道：是一个从一端发送数据，另一端接收数据的数据通道；

目录：也称为目录文件，它包含了保存在目录中文件列表的简单文件。

设备：该类型的文件提供了大多数物理设备的接口。它又分为两种类型：字符型设备和块设备。字符型设备一次只能读出和写入一个字节的数据，包括调制解调器、终端、打印机、声卡以及鼠标；块设备必须以一定大小的块来读出或者写入数据，块设备包括 CD-ROM、RAM 驱动器和磁盘驱动器等，一般而言，字符设备用于传输数据，块设备用于存储数据。

链接：类似于 Windows 的快捷方式和 Linux 里的别名，指包含到达另一个文件路径的文件。

套接字：在 Linux 中,套接字也可以当作文件来进行处理。

基于文件指针的文件操作函数是 ANSI 标准函数库的一部分。

1.1. 文件的创建，打开与关闭

原型为：

```
#include <stdio.h>  //头文件包含
FILE *fopen(const char *path,const char *mode);  //文件名  模式
int fclose(FILE *stream);
```

fopen 以 mode 的方式打开或创建文件，如果成功，将返回一个文件指针，失败则返回 NULL.

fopen 创建的文件的访问权限将以 0666 与当前的 umask 结合来确定。

mode 的可选模式列表

模式	读	写	位置	截断原内容	创建
rb	Y	N	文件头	N	N
rb+	Y	Y	文件头	N	N
wb	N	Y	文件头	Y	Y
wb+	Y	Y	文件头	Y	Y
ab	N	Y	文件尾	N	Y
ab+	Y	Y	文件尾	N	Y

在 Linux 系统中,mode 里面的'b'(二进制)可以去掉，但是为了保持与其他系

统的兼容性，建议不要去掉。ab 和 a+b 为追加模式，在此两种模式下，无论文件读写点定位到何处，在写数据时都将是在文件末尾添加，所以比较适合于多进程写同一个文件的情况下保证数据的完整性。

1.2. 读写文件

基于文件指针的数据读写函数较多，可分为如下几组：

数据块读写：

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

fread 从文件流 stream 中读取 nmemb 个元素，写到 ptr 指向的内存中，每个元素的大小为 size 个字节。

fwrite 从 ptr 指向的内存中读取 nmemb 个元素，写到文件流 stream 中，每个元素 size 个字节。

所有的文件读写函数都从文件的当前读写点开始读写，读写完以后，当前读写点自动往后移动 size*nmemb 个字节。

整块 copy，速度较快，但是是二进制操作

格式化读写：

```
#include <stdio.h>
int printf(const char *format, ...);                    //相当于 fprintf(stdout,format,...);
int scanf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);    重点
int fscanf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);        // eg: sprintf(buf,"the string is;%s",str);  重点
int sscanf(char *str, const char *format, ...);  重点
```

fprintf 将格式化后的字符串写入到文件流 stream 中

sprintf 将格式化后的字符串写入到字符串 str 中

单个字符读写：

使用下列函数可以一次读写一个字符

```
#include <stdio.h>
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
int getc(FILE *stream);          ✦ 等同于 fgetc(FILE* stream)
int putc(int c, FILE *stream);     ✦ 等同于 fputc(int c, FILE* stream)
int getchar(void);               ✦ 等同于 fgetc(stdin);
int putchar(int c);              ✦ 等同于 fputc(int c, stdout);
```

getchar 和 putchar 从标准输入输出流中读写数据，其他函数从文件流 stream 中读写数据。

字符串读写：

char *fgets(char *s, int size, FILE *stream);
int fputs(const char *s, FILE *stream);
int puts(const char *s); ✦ 等同于 fputs(const char *s,stdout);
char *gets(char *s); ✦ 等同于 fgets(const char *s, int size, stdin);

fgets 和 fputs 从文件流 stream 中读写一行数据;
puts 和 gets 从标准输入输出流中读写一行数据。
fgets 可以指定目标缓冲区的大小，所以相对于 gets 安全，但是 fgets 调用时，如果文件中当前行的字符个数大于 size，则下一次 fgets 调用时，将继续读取该行剩下的字符,fgets 读取一行字符时，保留行尾的换行符。
fputs 不会在行尾自动添加换行符，但是 puts 会在标准输出流中自动添加一换行符。

文件定位：

文件定位指读取或设置文件当前读写点，所有的通过文件指针读写数据的函数，都是从文件的当前读写点读写数据的。
常用的函数有：

#include <stdio.h>
int feof(FILE * stream); //通常的用法为 while(!feof(fp))
int fseek(FILE *stream, long offset, int whence);//设置当前读写点到偏移 whence 长度为 offset 处
long ftell(FILE *stream); //用来获得文件流当前的读写位置
void rewind(FILE *stream); //把文件流的读写位置移至文件开头
✦fseek(fp, 0, SEEK_SET);

feof 判断是否到达文件末尾的下一个（注意到达文件末尾之后还会做一次）
fseek 设置当前读写点到偏移 whence 长度为 offset 处，whence 可以是：
SEEK_SET (文件开头 ＊0)
SEEK_CUR (文件当前位置 ＊1)
SEEK_END (文件末尾 ＊2)
ftell 获取当前的读写点
rewind 将文件当前读写点移动到文件头

注：基于文件指针的文件操作请参考《C 语言文件操作常用函数详解.doc》

1.3. 目录操作

改变目录或文件的访问权限

#include <sys/stat.h>
int chmod(const char* path, mode_t mode);//mode 形如：0777
path 参数指定的文件被修改为具有 mode 参数给出的访问权限。

获取、改变当前目录：

原型为：

```
#include <unistd.h>   //头文件
char *getcwd(char *buf, size_t size);   //获取当前目录，相当于 pwd 命令
int chdir(const char *path);                    //修改当前目录，即切换目录，相当于 cd 命令
```

其中 getcwd()函数：将当前的工作目录绝对路径复制到参数 buf 所指的内存空间，参数 size 为 buf 的空间大小. 在调用此函数时，buf 所指的内存空间要足够大，若工作目录绝对路径的字符串长度超过参数 size 大小，则回值 NULL，errno 的值则为 ERANGE。倘若参数 buf 为 NULL，getcwd()会依参数 size 的大小自动配置内存(使用 malloc())，如果参数 size 也为 0，则 getcwd()会**依工作目录绝对路径的字符串程度来决定所配置的内存大小**，进程可以在使用完此字符串后自动利用 free()来释放此空间。所以常用的形式：getcwd(NULL, 0);

chdir()函数：用来将当前的工作目录改变成以参数 path 所指的目录

Example：

```
#include<unistd.h>
main()
{
    chdir(“/tmp”);
    printf(“current working directory: %s\n”,getcwd(NULL,0));
}
```

创建和删除目录：

原型为：

```
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int mkdir(const char *pathname, mode_t mode);   //创建目录,mode 是目录权限
int rmdir(const char *pathname);                    //删除目录
```

获取目录信息：

原型为：

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);      //打开一个目录
struct dirent *readdir(DIR *dir);   //读取目录的一项信息，并返回该项信息的结构体指针
void rewinddir(DIR *dir);           //重新定位到目录文件的头部
void seekdir(DIR *dir,off_t offset);//用来设置目录流目前的读取位置
off_t telldir(DIR *dir);                //返回目录流当前的读取位置
int closedir(DIR *dir);             //关闭目录文件
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *pathname, struct stat *buf);  获取文件状态
```

读取目录信息的步骤为：

③ 用 opendir 函数打开目录；

③ 使用 readdir 函数迭代读取目录的内容，如果已经读取到目录末尾，又想重新开始读，则可以使用 rewinddir 函数将文件指针重新定位到目录文件的起始位置；

③ 用 closedir 函数关闭目录

opendir()用来打开参数 name 指定的目录，并返回 DIR*形态的目录流，和文件操作函数 open()类似，接下来对目录的读取和搜索都要使用此返回值。函数失败则返回 NULL；

readdir()函数用来读取目录的信息，并返回一个结构体指针，该指针保存了目录的相关信息。有错误发生或者读取到目录文件尾则返回 NULL；dirent 结构体如下：

```
struct dirent
{
    ino_t  d_ino;              /* inode number（此目录进入点的 inode） */
    off_t  d_off;              /* offset to the next dirent（目录开头到进入点的位移 */
    unsigned short d_reclen;   /* length of this record（目录名的长度） */
    unsigned char d_type;      /* type of file（所指的文件类型） */
    char   d_name[256];        /* filename（文件名） */
};
```

seekdir()函数用来设置目录流目前的读取位置，再调用 readdir()函数时，便可以从此新位置开始读取。参数 offset 代表距离目录文件开头的偏移量。

telldir()函数用来返回目录流当前的读取位置。

结构体 stat 的定义为：

```
struct stat {
        dev_t         st_dev;      /*如果是设备，返回设备表述符，否则为 0*/
        ino_t         st_ino;      /* i 节点号 */
        mode_t        st_mode;     /* 文件类型 */
        nlink_t       st_nlink;    /* 链接数 */
        uid_t         st_uid;      /* 属主 ID */
        gid_t         st_gid;      /* 组 ID */
        dev_t         st_rdev;     /* 设备类型*/
        off_t         st_size;     /* 文件大小，字节表示 */
        blksize_t     st_blksize;  /* 块大小*/
        blkcnt_t      st_blocks;   /* 块数 */
        time_t        st_atime;    /* 最后访问时间*/
        time_t        st_mtime;    /* 最后修改时间*/
        time_t        st_ctime;    /* 最后权限修改时间 */
};
```

man 2 stat 看 time_t

time_t 的结构，在头文件 /usr/include/linux/time.h

st_mtim

```
struct timespec {
    __kernel_time_t tv_sec;         /* seconds */
    long        tv_nsec;        /* nanoseconds */
};
```

__kernel_time_t 是 long 类型的

示例：

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

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

int main(int argc,char *argv[])
{
    struct dirent *pDirInfo;
    DIR *pDir;
    if(argc < 2)
        pDir = opendir(".");
    else
        pDir = opendir(argv[1]);
    if(NULL == pDir)
    {
        perror("open dir fail!");
        return -1;
    }
    while( (pDirInfo = readdir(pDir)) != NULL )
        printf("%s\n",pDirInfo->d_name);
    closedir(pDir);
    return 0;

}
```

Example：以树形结构的形式输出指定目录下面的所有文件

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <stdlib.h>
void printdir(char *dir, int depth)
{
    DIR *dp = opendir(dir);
    if(NULL == dp)
    {
        fprintf(stderr,"cannot open directory: %s\n", dir);
        return;
    }
    chdir(dir);
    struct dirent *entry;
    struct stat statbuf;
    while((entry = readdir(dp)) != NULL)
    {
        stat(entry->d_name,&statbuf);
        if(S_ISDIR(statbuf.st_mode))
        {
```

```
            if(strcmp(".",entry->d_name) == 0 || strcmp("..",entry->d_name) == 0)
                continue;
            printf("%*s%s/\n",depth,"",entry->d_name);
            printdir(entry->d_name,depth+4);
        }
        else
            printf("%*s%s\n",depth,"",entry->d_name);
            //printf(“%*s”,4,”*”); 该函数表示输出“___*”,前面输出 3 个空格。
            //如果是 printf(“%*s”,4,“**”);则表示输出“__**”， 前面输出 2 个空格。
    }
    chdir("..");
    closedir(dp);
}

int main(int argc, char* argv[])
{
    char *topdir, pwd[2]=".";
    if (argc < 2)
        topdir=pwd;
    else
        topdir=argv[1];
    printf("Directory scan of %s\n",topdir);
    printdir(topdir,0);
    printf("done.\n");
    exit(0);
}
```

1.4. 标准输入/输出流

在进程一开始运行，就自动打开了三个对应设备的文件，它们是标准输入、输出、错误流，分别用全局文件指针 stdin、stdout、stderr 表示，stdin 具有可读属性，缺省情况下是指从键盘的读取输入，stdout 和 stderr 具有可写属性，缺省情况下是指向屏幕输出数据。
示例：

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    char szBuf[32];
    printf("Input string:");              //向屏幕输出一字符串
    fgets(szBuf,sizeof(szBuf),stdin);//从键盘读入一行字符串
    fprintf(stdout,"The string is:%s",szBuf);//向屏幕输出一行字符串
    return 0;
}
```

2. 基于文件描述符的文件操作(非缓冲)

2.1. 文件描述符

内核为每个进程维护一个已打开文件的记录表，文件描述符是一个较小的正整数（0—1023），它代表记录表的一项，通过文件描述符和一组基于文件描述符的文件操作函数，就可以实现对文件的读、写、创建、删除等操作。常用基于文件描述符的函数有 **open（打开）**、creat（创建）、**close（关闭）**、**read（读取）**、**write（写入）**、ftruncate（改变文件大小）、lseek（定位）、fsync（同步）、fstat（获取文件状态）、fchmod（权限）、flock（加锁）、fcntl（控制文件属性）、dup（复制）、dup2、**select** 和 ioctl。基于文件描述符的文件操作并非 ANSI C 的函数。
如果不清楚某个函数的具体实现形式，可以通过下面的方式查询
man 函数名　查看该函数的帮助。

2.2. 打开、创建和关闭文件

open 和 creat 都能打开和创建函数，原型为

```
#include <sys/types.h>    //头文件
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);        //文件名  打开方式
int open(const char *pathname, int flags, mode_t mode);//文件名  打开方式  权限
int creat(const char *pathname, mode_t mode);    //文件名  权限    //现在已经不常用了
creat 函数等价于✦open(pathname,O_CREAT|O_TRUNC|O_WRONLY,mode);
```

open()函数出错时返回-1，相关参数如下：
flags 和 mode 都是一组掩码的合成值，flags 表示打开或创建的方式，mode 表示文件的访问权限。
flags 的可选项有

掩码	含义
O_RDONLY	以只读的方式打开
O_WRONLY	以只写的方式打开
O_RDWR	以读写的方式打开
O_CREAT	如果文件不存在，则创建文件
O_EXCL	仅与 O_CREAT 连用，如果文件已存在，则强制 open 失败
O_TRUNC	如果文件存在，将文件的长度截至 0
O_APPEND	已追加的方式打开文件，每次调用 write 时，文件指针自动先移到文件尾，用于多进程写同一个文件的情况。
O_NONBLOCK	非阻塞方式打开，无论有无数据读取或等待，都会立即返回进程之中。
O_NODELAY	非阻塞方式打开
O_SYNC	同步打开文件，只有在数据被真正写入物理设备设备后才返回

mode 的可选项有：
S_IRWXU 00700 权限，代表该文件所有者具有可读、可写及可执行的权限。

S_IRUSR 或 S_IREAD，00400 权限，代表该文件所有者具有可读取的权限。
S_IWUSR 或 S_IWRITE，00200 权限，代表该文件所有者具有可写入的权限。
S_IXUSR 或 S_IEXEC，00100 权限，代表该文件所有者具有可执行的权限。

S_IRWXG 00070 权限，代表该文件用户组具有可读、可写及可执行的权限。
S_IRGRP 00040 权限，代表该文件用户组具有可读的权限。
S_IWGRP 00020 权限，代表该文件用户组具有可写入的权限。
S_IXGRP 00010 权限，代表该文件用户组具有可执行的权限。

S_IRWXO 00007 权限，代表其他用户具有可读、可写及可执行的权限。
S_IROTH 00004 权限，代表其他用户具有可读的权限
S_IWOTH 00002 权限，代表其他用户具有可写入的权限。
S_IXOTH 00001 权限，代表其他用户具有可执行的权限。
但是通常采用直接赋数值的形式,如：

```
int fd = open(“1.txt”,O_WRONLY | O_CREAT,0755);  //表示给 755 的权限
if(-1 == fd)
{
    perror("open failed!\n");
    exit(-1);
}
```

注意：LINUX 中基于文件描述符的 open 函数，对于一个不存在的文件，不能通过 O_WRONLY 的方式打开，必须加上 O_CREAT 选项。

close 用于文件的关闭：
int close(int fd);//fd 表示文件描述词,是先前由 open 或 creat 创建文件时的返回值。
文件使用完毕后，应该调用 close 关闭它，一旦调用 close，则该进程对文件所加的锁全都被释放，并且使文件的打开引用计数减 1，只有文件的打开引用计数变为 0 以后，文件才会被真正的关闭。

2.3. 读写文件

读写文件的函数原型为：

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);//文件描述词  缓冲区  长度
ssize_t write(int fd, const void *buf, size_t count);
```

对于 read 和 write 函数，出错返回-1，读取完了之后，返回 0， 其他情况返回读写的个数。
Example：将 aaa.txt 中的内容复制到 bbb.txt 中，其中 bbb.txt 起初不存在。

```
#include <stdio.h>
#include <stdlib.h>              //包含 exit
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>               //用 perror 输出错误
#include <unistd.h>
#define FILENAME1 "./aaa.txt"        //用宏定义文件的路径，可以实现一改都改
#define FILENAME2 "./bbb.txt"
main()
```

```
{
    char buf[512] = {0};
    int fo1 = open(FILENAME1, O_RDONLY);//fo1,fo2 都是文件描述词
    int fo2 = creat(FILENAME2, 0755);  //创建文件
    //int fo2 = open(FILENAME2, O_WRONLY | O_CREAT);
    if( (-1 == fo1) || (-1 == fo2) )
    {
        perror("open failed!\n");
        //用于输出错误信息.类似于：fputs(”open failed\n”,stderr);
        exit(-1);
    }
    int fr = 0;
    while( (fr = read(fo1, buf, sizeof(buf))) > 0 )
    //如果 read 读取成功，返回的是长度，否则，返回-1
    {
        int fw = write(fo2, buf, fr);
        if( -1 == fw )
        {
            perror("write failed!");
                exit(-1);
        }
    }
    close(fo1);
    close(fo2);
}
```

2.4. 改变文件大小

函数原型：

```
#include <unistd.h>
int ftruncate(int fd, off_t length);
```

函数 ftruncate 会将参数 fd 指定的文件大小改为参数 length 指定的大小。参数 fd 为已打开的文件描述词，而且必须是以写入模式打开的文件。如果原来的文件大小比参数 length 大，则超过的部分会被删去。

返回值 执行成功则返回 0，失败返回-1。

实例：

```
int main()
{
    int fd = open("a.txt", O_WRONLY);
    ftruncate(fd, 1000);
    close(fd);
    return 0;
}
```

2.5. 文件定位

函数 lseek 将文件指针设定到相对于 whence，偏移值为 offset 的位置

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);//fd 文件描述词
```

whence 可以是下面三个常量的一个

SEEK_SET 从文件头开始计算

SEEK_CUR 从当前指针开始计算

SEEK_END 从文件尾开始计算

利用该函数可以实现文件空洞（对一个新建的空文件，可以定位到偏移文件开头 1024 个字节的地方，在写入一个字符，则相当于给该文件分配了 1025 个字节的空间，形成文件空洞）通常用于多进程间通信的时候的共享内存。

```
int main()
{
    int fd = open("c.txt", O_WRONLY | O_CREAT);
    lseek(fd, 1024, SEEK_SET);
    write(fd, "a", 1);
    close(fd);
    return 0;
}
```

2.6. 获取文件信息

可以通过 fstat 和 stat 函数获取文件信息，调用完毕后，文件信息被填充到结构体 struct stat 变量中，函数原型为：

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

int stat(const char *file_name, struct stat *buf);      //文件名  stat 结构体指针
int fstat(int fd, struct stat *buf);            //文件描述词   stat 结构体指针
```

结构体 stat 的定义为：

```
    struct stat {
            dev_t         st_dev;      /*如果是设备，返回设备表述符，否则为 0*/
            ino_t         st_ino;      /* i 节点号 */
            mode_t        st_mode;     /* 文件类型 */  无符号短整型
            nlink_t       st_nlink;    /* 链接数 */
            uid_t         st_uid;      /* 属主 ID */
            gid_t         st_gid;      /* 组 ID */
            dev_t         st_rdev;     /* 设备类型*/
            off_t         st_size;     /* 文件大小，字节表示 */
            blksize_t     st_blksize;  /* 块大小*/
            blkcnt_t      st_blocks;   /* 块数 */
```

```
    time_t      st_atime;    /* 最后访问时间*/
    time_t      st_mtime;    /* 最后修改时间*/
    time_t      st_ctime;    /* 最后权限修改时间 */
};
```

对于结构体的成员 st_mode，有一组宏可以进行文件类型的判断

宏	描述
S_ISLNK(mode)	判断是否是符号链接
S_ISREG(mode)	判断是否是普通文件
S_ISDIR(mode)	判断是否是目录
S_ISCHR(mode)	判断是否是字符型设备
S_ISBLK(mode)	判断是否是块设备
S_ISFIFO(mode)	判断是否是命名管道
S_ISSOCK(mode)	判断是否是套接字

通常用于判断：if(S_ISDIR(st.st_mode)){}

Example:获得文件的大小

```
#include<sys/stat.h>
#include<unistd.h>
main()
{
    struct stat buf;
    stat ("/etc/passwd",&buf);
    printf("/etc/passwd file size = %d \n",buf.st_size);//st_size 可以得到文件大小
}
```

如果用 fstat 函数实现，如下：

```
int fd = open ("/etc/passwd",O_RDONLY);  //先获得文件描述词
fstat(fd, &buf);
```

实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>

int main()
{
    int fd = open("/home/wangxiao/0926/a.txt", O_RDONLY);
    if(fd == -1)
    {
        perror("open error");
        exit(-1);
```

```
    }

    struct stat buf;
    int iRet = fstat(fd, &buf);
    if(iRet == -1)
    {
        perror("fstat error");
        exit(-1);
    }

    if(S_ISREG(buf.st_mode))
    {
        printf("regular file!\n");
    }
    if(S_ISDIR(buf.st_mode))
    {
        printf("directory!\n");
    }
    if(S_ISLNK(buf.st_mode))
    {
        printf("link file!\n");
    }

    printf("the size of file is : %d\n", buf.st_size);

    time_t tt = buf.st_atime;
    struct tm *pT = gmtime(&tt);
    printf("%4d-%02d-%02d      %02d:%02d:%02d\n",  (1900+pT->tm_year),  (1+pT->tm_mon),
pT->tm_mday, (8+pT->tm_hour), pT->tm_min, pT->tm_sec);
    //  printf("the last access time is : %d\n", buf.st_atime);

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

2.7. 文件描述符的复制

系统调用函数 dup 和 dup2 可以实现文件描述符的复制，经常用来重定向进程的 stdin(0),stdout(1),stderr(2)。

dup 返回新的文件描述符（没有使用的文件描述符的最小编号）。这个新的描述符是旧文件描述符的拷贝。这意味着两个描述符共享同一个数据结构。

dup2 允许调用者用一个有效描述符(oldfd)和目标描述符(newfd)，函数成功返回时，目标描述符将变成旧描述符的复制品，此时两个文件描述符现在都指向同一个文件，并且是函数第一个参数（也就是 oldfd）指向的文件。

原型为：

```
#include <unistd.h>   //头文件包含
```

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

文件描述符的复制是指用另外一个文件描述符指向同一个打开的文件，它完全不同于直接给文件描述符变量赋值，例如：

描述符变量的直接赋值：

```
char szBuf[32];
int fd=open(“./a.txt”,O_RDONLY);
int fd2=fd;   //类似于 C 语言的指针赋值，当释放掉一个得时候，另一个已经不能操作了
close(fd);              //导致文件立即关闭
printf(“read:%d\n”,read(fd2,szBuf,sizeof(szBuf)-1));  //读取失败
close(fd2);             //无意义
```

在此情况下，两个文件描述符变量的值相同，指向同一个打开的文件，但是内核的文件打开引用计数还是为 1，所以 close(fd)或者 close(fd2)都会导致文件立即关闭掉。

描述符的复制：

```
char szBuf[32];
int fd=open(“./a.txt”,O_RDONLY);
int fd2=dup(fd);            //内核的文件打开引用计算+1，变成 2 了
close(fd);              //当前还不会导致文件被关闭，此时通过 fd2 照样可以访问文件
printf(“read:%d\n”,read(fd2,szBuf,sizeof(szBuf)-1));
close(fd2);             //内核的引用计数变为 0，文件正式关闭
```

Example：

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

int main(int argc,char *argv[])
{
    char szBuf[32]={0};
    int fda=open("./a.txt",O_RDWR); //假设 a.txt 的内容为：hello world
    int fdaa=dup(fda);
    read(fda,szBuf,4);
    puts(szBuf);                    //关闭之前先输入原来的内容
    close(fda);

//  lseek(fdaa, 0, SEEK_SET);
    read(fdaa,szBuf,sizeof(szBuf));
    puts(szBuf);                    //输出现在的内容
    close(fdaa);
}
```

解析：假设 a.txt 中的内容为：hello world.上面的例子会发现第一次输出的结果是 hell。关闭 close(fda)的时候，文件实际上还没有真正的关闭，此时文件指针已经向后移动了。执行第二次 read

命令将 o world 读出来，最后关闭 fdaa。

dup 有时会用在一些特定的场合，如下：

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdlib.h>
int main()
{
        int fd = open("a.txt", O_WRONLY | O_CREAT);
        if(fd == -1)
        {
                perror("open error");
                exit(-1);
        }
        printf("\n");    /* 必不可少 */
        close(1);
        int fd2 = dup(fd);
        close(fd);
        printf("hello world\n");
        close(fd2);
        return 0;
}
```

该程序首先打开了一个文件，返回一个文件描述符，因为默认的就打开了 0,1,2 表示标准输入，标准输出，标准错误输出。而用 close(1);则表示关闭标准输出，此时这个文件描述符就空着了。后面又用 dup，此时 dup(fd);则会复制一个文件描述符到当前未打开的最小描述符，此时这个描述符为 1.后面关闭 fd 自身，然后在用标准输出的时候，发现标准输出重定向到你指定的文件了。那么 printf 所输出的内容也就直接输出到文件了。

dup2(int fdold,int fdnew)也是进行描述符的复制，只不过采用此种复制，新的描述符由用户用参数 fdnew 显示指定，而不是象 dup 一样由内核帮你选定（内核选定的是随机的）。对于 dup2，如果 fdnew 已经指向一个已经打开的文件，内核会首先关闭掉 fdnew 所指向的原来的文件。此时再针对于 fdnew 文件描述符操作的文件，则采用的是 fdold 的文件描述符。如果成功 dup2 的返回值于 fdnew 相同,否则为-1.

思考下面程序的结果：

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

int main(int argc,char *argv[])
{
```

```
    char szBuf[32]={0};
    int fda=open("./a.txt",O_RDONLY);
    int fdb=open("./b.txt",O_RDONLY);
    int fdbb=dup(fdb);
    int fda2=dup2(fda,fdb);  //可以设定为：int fda2 = dup2(fda,5);即自己设为5
    printf("fda:%d fdb:%d fdbb:%d fda2:%d",fda,fdb,fdbb,fda2);
    read(fdb,szBuf,sizeof(szBuf)-1);          //此时fdb已经不再定位b.txt而是a.txt
    printf("result:%s\n",szBuf);
    close(fda);
    close(fdb);
    close(fdbb);
    close(fda2);
}
```

2.8. 标准输入输出文件描述符

与标准的输入输出流对应，在更底层的实现是用标准输入、标准输出、标准错误文件描述符表示的。它们分别用 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO 三个宏表示，值分别是 0、1、2 三个整型数字。

标准输入文件描述符 ✦ STDIN_FILENO ✦ 0

标准输出文件描述符 ✦ STDOUT_FILENO ✦ 1

标准错误输出文件描述符 ✦ STDERR_FILENO ✦ 2

示例：

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main()
{
    char szBuf[32],szBuf2[50];
    printf("Input string:");
    fflush(stdout); //要刷新标准输出流，才可以立即在屏幕上显示”Input  string”
    //fflush用于linux中的时候，只对fflush(stdout)有效。
    int iRet=read(STDIN_FILENO,szBuf,sizeof(szBuf));
    szBuf[iRet]=0;  //read是以无类型指针方式读的数据，不会自动在缓冲区后加0结束标记。
    sprintf(szBuf2,"The string is:%s",szBuf);
    write(STDOUT_FILENO,szBuf2,strlen(szBuf2));
    return 0;
}
```

fsync(int fd) 同步缓冲区到磁盘

2.9. I/O 多路转接模型

I/O 多路转接模型：在这种模型下，如果请求的 I/O 操作阻塞，且它不是真正阻塞 I/O，而是让其中的一个函数等待，在这期间，I/O 还能进行其他操作。如本节要介绍的 select()函数，

就是属于这种模型。

Select 函数：

#include <sys/select.h>

#include <sys/time.h>

int select(int maxfd, fd_set *readset,fd_set *writeset, fd_set *exceptionset, const struct timeval * timeout);

返回:就绪描述字的正数目，0——超时，-1——出错

参数解释：

maxfd： 最大的文件描述符（其值应该为最大的文件描述符字 + 1）

readset： 内核读操作的描述符字集合

writeset：内核写操作的描述符字集合

exceptionset：内核异常操作的描述符字集合

timeout：等待描述符就绪需要多少时间。NULL 代表永远等下去，一个固定值代表等待固定时间，0 代表根本不等待，检查描述字之后立即返回。

其中 readset、writeset、exceptionset 都是 fd_set 集合。该集合的相关操作如下：

```
void FD_ZERO(fd_set *fdset);        /* 将所有 fd 清零 */
void FD_SET(int fd, fd_set *fdset);  /* 增加一个 fd */
void FD_CLR(int fd, fd_set *fdset);  /* 删除一个 fd */
int FD_ISSET(int fd, fd_set *fdset);  /* 判断一个 fd 是否有设置 */
```

一般来说，在使用 select 函数之前，首先要使用 FD_ZERO 和 FD_SET 来初始化文件描述符集，在使用 select 函数时，可循环使用 FD_ISSET 测试描述符集，在执行完对相关文件描述符之后，使用 FD_CLR 来清除描述符集。

另外，select 函数中的 timeout 是一个 struct timeval 类型的指针，该结构体如下：

```
struct timeval
{
    long tv_sec;        /* second */        //秒
    long tv_usec;   /* microsecond */   //微秒
};
```

Example:多路转接模型 select

```
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#define FILENAME1 "a.txt"
#define FILENAME2 "dir.c"
int main()
{
        char buf[10] = {0};
```

```
    int fd1 = open(FILENAME1,O_RDWR);
int fd2 = open(FILENAME2,O_RDWR);
    int fd3 = open(FILENAME1,O_RDWR);
    int fd4 = open(FILENAME2,O_RDWR);

    if( (-1 == fd1) || (-1 == fd2) || (-1 == fd3) || (-1 == fd4) )
    {
        perror("open");
        exit(-1);
    }

    fd_set fdrd,fdwr;        //绑定读写集合

    FD_ZERO(&fdrd);          //清除以前读的绑定
    FD_ZERO(&fdwr);          //清除以前写的绑定

    FD_SET(fd1,&fdrd);       //将 fd1 与读绑定
    FD_SET(fd2,&fdrd);
    FD_SET(fd3,&fdwr);       //将 fd3 与写绑定
    FD_SET(fd4,&fdwr);

    int max1 = fd1 > fd2 ? fd1 : fd2;        //获取读绑定中的文件描述词最大值
    int max2 = fd3 > fd4 ? fd3 : fd4;        //获取写绑定中的文件描述词最大值
    int max = max1 > max2 ? max1 : max2;         //获得读写文件描述词最大值

    struct timeval tv;       //用于记录时间，表示过这么长时间不响应就退出
    tv.tv_sec = 2;           //秒
    tv.tv_usec = 0;          //微妙

    while(1)
    {
        if( select(max+1, &fdrd, &fdwr, NULL, &tv) == -1 )  //从 1—max+1 查找
        {
            perror("select");
            break;
        }

        if( FD_ISSET(fd1,&fdrd) )   //如果 fd1 设置的是读绑定
        {
            read(fd1,buf,sizeof(buf)-1);
            puts(buf);
            sleep(1);
        }

        if( FD_ISSET(fd2,&fdrd) )
```

```
            {
                read(fd2,buf,sizeof(buf)-1);
                puts(buf);
                sleep(1);
            }

            if( FD_ISSET(fd3,&fdwr) )    //如果 fd3 设置的是写绑定
            {
                write(fd3,buf,sizeof(buf));
                sleep(2);
            }

            if( FD_ISSET(fd4,&fdwr) )
            {
                write(fd4,buf,sizeof(buf));
                sleep(2);
            }
        }
        close(fd1);
        close(fd2);
        close(fd3);
        close(fd4);
}
```

样例存在问题

补充：

1. 程序参数：

```
int main(int argc,char ** argv)
{
  int i = 0;
  for( ; i< argc; i++)
    printf("%s",argv[i]);
}
```

参数 argc 表示命令行传入的参数个数，并且一次保存到 argv 数组中。例如程序叫 main.c，最后执行的时候，如果直接是./main 则 argc=1,此时 argv[0]就是./main;如果是./main aaa bbb，则 argc=3，argv[0]:./main, argv[1]:aaa argv[2]:bbb.

2. 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、警告或者与系统状态有关的一般信息。通常是在/var/log 目录下的 messages 中包含了系统信息。通过 syslog 可以向系统的日志发送日志信息。

函数原型如下：

```
#include <syslog.h>
void syslog(int priority, const char* message, arguments…);
```

对于 priority 有如下几个常见的：

LOG_EMERG	紧急情况
LOG_ALERT	高优先级故障（如：数据库崩溃）
LOG_CRIT	严重错误（如：硬件错误）

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(写不到 messages 里面)

```
#include <syslog.h>
main()
{
   //openlog("log",LOG_PID|LOG_CONS|LOG_NOWAIT,LOG_USER);
   syslog(LOG_ALERT,"this is alert\n");
   syslog(LOG_INFO,"this is info\n");
   syslog(LOG_DEBUG,"this is debug%d\t %s",10,"aaaa");
   syslog(LOG_ERR,"err");
   syslog(LOG_CRIT,"crit");
   //closelog();
}
```

利用 tail -10 /var/log/messages 可以查看。

还可以通过函数 openlog 函数来改变日志信息的表达方式。openlog 的原型如下：

```
#include <syslog.h>
void openlog(const char* ident, int logopt, int facility);
void closelog(void);
```

它可以设置一个字符串 ident，该字符串会添加在日志信息的前面。你可以通过它来指明是哪个应用程序创建了这条信息。facility 值为 LOG_USER。logopt 参数对后续 syslog 调用的行为进行配置。如下：

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

3. 日期

```
#include <time.h>
main()
{
  char *wday[] = { "sunday", "monday", "tuesday", "wednesday", "thursday", "friday","saterday" };
    time_t timep;
    struct tm *pTm;
    time(&timep);
    pTm = gmtime(&timep);
    printf("%04d-%02d-%02d\t",(1900 + pTm->tm_year), (1 + pTm->tm_mon), (pTm->tm_mday));
    printf("%02d:%02d:%02d\t",(8 + pTm->tm_hour), (pTm->tm_min), (pTm->tm_sec));  //似乎小时总是相差 8，所以加上 8 试试
    printf("%s\n",wday[pTm->tm_wday]);
}
```

2.10. MMAP 文件映射

http://baike.baidu.com/link?url=SoyVq2nGkiUnhXhXvPsDtuZPrxv_JnEv36oH35wrDI8JXUa-4pHFdDBseyDX3EiLov7K8MDcANFsZJZ8hJnvAK

向文件写内容，必须是文件有大小的。