4 Linux文件操作

常用的宏函数：

```
#define ARGS_CHECK(argc,num) {if(argc != num){fprintf(stderr,"args
error!\n");return -1;}}
#define ERROR_CHECK(ret,num,msg) {if(ret == num){perror(msg);return -1;}}
```

4.0 文件和文件系统

在UNIX系统当中，“万物皆文件”是一种重要的设计思想。在传统的定义当中，我们把**存储在磁盘当中的数据集合**称为文件。而在UNIX的设计当中，文件这个概念得到了进一步泛化，所有满足**速度较慢**、**容量较大**和可以**持久化存储**中任意一个特征的数据集合都可以称为**文件**，包括不限于磁盘数据、输入输出设备、用于进程间通信的管道、网络等等都属于文件。

文件系统是操作系统用于管理文件的子模块。在文件系统当中，会根据文件的存在形式分为普通文件、目录文件、链接文件和设备文件等类型：

- 普通文件：也称磁盘文件，并且能够进行随机(能够自由使用lseek或者fseek定位到某一个位置)的数据存储;
- 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：**字符设备**和**块设备**。字符设备一次只能读出和写入一个字节的数据，包括终端、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，指包含到达另一个文件路径的文件。

4.1 基于文件流的文件操作

文件流，又称为（用户态）文件缓冲区，它是由标准C库（ISO C）设计和定义的用于管理文件的数据结构。如果进程想要使用C库函数操作文件数据，就必须提前在内存中先申请创建一个文件流对象。

4.1.1 文件流的创建与关闭

创建文件流使用`fopen`，关闭文件流使用`fclose`。

```
#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`的mode参数是一个字符串，决定了以什么样的方式打开文件。如果打开文件执行成功，将返回一个指向文件流的指针（简称为文件指针）；失败则返回NULL，此时可以使用`perror`来检查报错原因。

下面是mode参数的可选模式列表，如下所示：

模式	读	写	位置	截断原内容	创建
rb	Y	N	文件头	N	N

模式	读	写	位置	截断原内容	创建
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

a和a+为追加模式，在此两种模式下，在一开始的时候读取文件内容是从文件起始处开始读取的，而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（写完以后读写点就移动到文件末尾了），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。

4.1.2 读写文件

文件的读写操作其实就是把数据在外部设备（磁盘、键盘、屏幕...）和内存之间进行拷贝。这样的话，进程就可以直接访问外部设备数据或者是将数据持久化存储到外部设备中。当进程调用读写函数时，数据会在用户内存和文件流之间进行拷贝，随后文件流内部的数据会在操作系统和硬件的支持下最终和外部设备进行交互。

基于文件流的读写函数可以分为如下几组：

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，每个元素的大小为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，读写完成以后，当前读写点自动往后移动size*nmemb个字节。

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, ...);
int fscanf(FILE *stream, const char *format, …);
int sprintf(char *str, const char *format, ...);
//eg:sprintf(buf,"the string is;%s",str);
int sscanf(char *str, const char *format, …);
```

- `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。（`gets`在新版库当中已经弃用）

`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。

4.1.3 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);
```

- `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)、SEEK_CUR （文件当前位置)、SEEK_END （文件末尾)
- `ftell`获取当前的读写点
- `rewind`将文件当前读写点移动到文件头

4.1.4 文本文件和二进制文件

如果文件当中的内容是一串ASCII字符的序列，那么这类文件就是**文本文件**。对于文本类型的文件，如果使用`fread`读取到内存当中，那么就应该分配字符串类型的内存区域以存储数据，假若想要根据文件的内容得到相关类型的数据，就需要使用`fscanf`函数。

```
// 假如文件里面的内容是文本的100000
// echo -n 100000 > file1
int main(int argc, char *argv[])
{
    FILE * fp = fopen("file1","r+");
    // 从文本内容读到字符串数据
    char buf[1024] = {0};
    fread(buf,1,1024,fp);
    printf("buf = %s\n", buf);
    // 从文本内容读到int数据
    fseek(fp,0,SEEK_SET);
    int i;
    fscanf(fp,"%d", &i);
    printf("i = %d\n", i);
    return 0;
}
```

如果文件内容是内存数据块按字节为单位直接进行存储，那么该文件就是**二进制文件**。对于二进制文件，目前只能使用`fread`和`fwrite`进行读写操作。对应二进制文件的读写，需要遵循一个基本原则就是：**写入是什么类型，读取就使用同样的类型。**

4.2 文件相关的系统调用

接下来我们所介绍的函数都是系统调用，也就是说，它们是调用操作系统内核来实现相关的功能。这些系统调用一般都遵循POSIX规范，但是通常无法在Windows环境下调用。

4.2.1 修改文件权限

首先要介绍的系统调用是`chmod`，该系统调用可以修改文件权限。

```
#include <sys/stat.h>
int chmod(const char* path, mode_t mode);
//mode形如：0777 是一个八进制整型
//path参数指定的文件被修改为具有mode参数给出的访问权限。
```

示例：

```
int main(int argc, char *argv[])
{
    // ./chmod 0777 file1
    ARGS_CHECK(argc,3);
    mode_t mode;
    sscanf(argv[1],"%o",&mode);
    int ret = chmod(argv[2],mode);
    ERROR_CHECK(ret,-1,"chmod");
    return 0;
}
```

4.2.2 获取和修改当前工作目录

当前工作目录是**进程的属性**之一。当进程以相对路径的方式访问文件时，当前工作目录是就是该文件在文件系统路径当中的起点。

```
#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
```

`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。因此在调用此函数时，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(其底层使用了`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。用户可以在使用完此字符串后，自行利用`free`来释放此空间。所以常用的形式：

```
getcwd(NULL, 0);
```

`chdir`函数则用于将本进程的当前工作目录属性改成参数path所指的目录

```
int main()
{
    chdir("/tmp");
    printf("current working directory: %s\n",getcwd(NULL,0));
}
```

4.2.3 创建和删除空目录

```
#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
```

如果需要查看库函数`mkdir`而不是shell命令mkdir，使用man命令的时候需要指定系统库函数帮助手册（缩写为mkdir(2)）

```
$ man 2 mkdir
```

如果函数的（指针）参数使用const来进行修饰，这个参数就称为传入参数。这意味在函数的内部是不能够通过指针来修改传入参数的内容

4.2.3 目录的存储原理

我们把文件在磁盘当中的位置、文件的创建/修改时间、文件的权限等信息称为**文件元信息（file metadata）**。文件系统在设计的时候就需要利用专门的**索引结构（inode）**来存储文件的元信息。

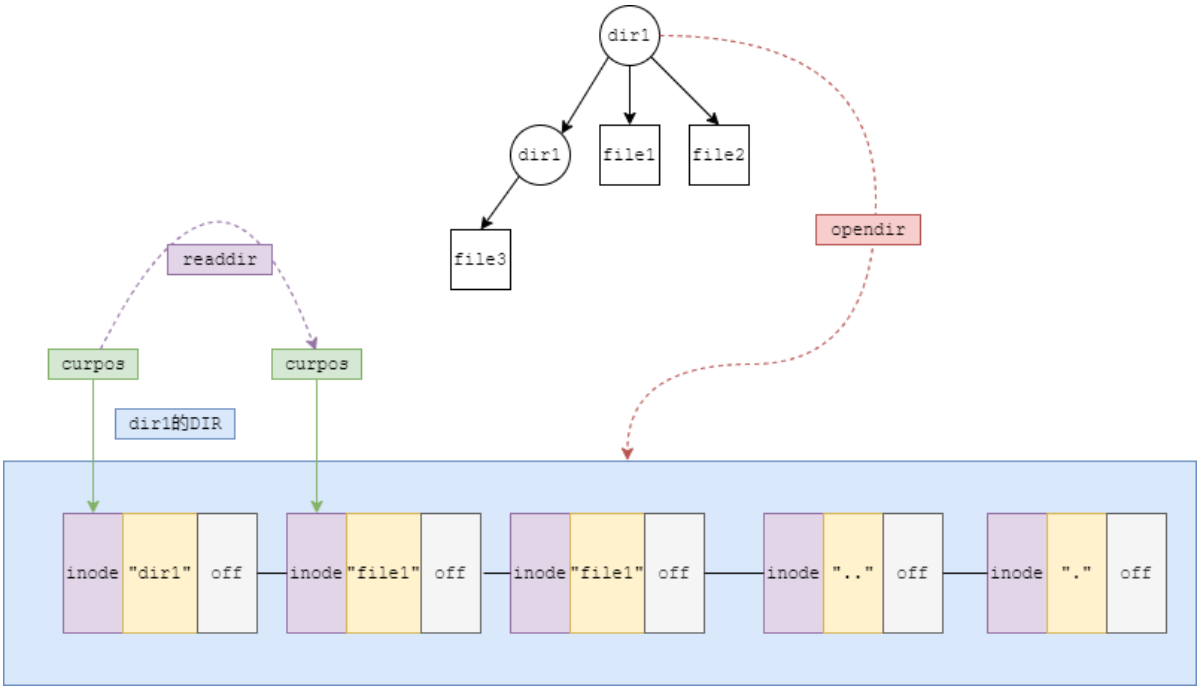
> 类似于地址和内存的关系，inode（索引结点）描述了文件在磁盘上的具体位置信息。在ls命令中添加 -i参数可查看文件的inode信息。那么所谓的硬链接，就是指inode相同的文件。一个inode的结点上的硬链接的个数就称为引用计数

```
$ ls -ial
# 查看所有文件的inode信息
$ ln 当前文件 目标
# 建立名为“目标”的硬链接
```

> - 这里可以检查 . 文件引用计数，发现是2。这是因为一个目录可以通过本目录或者是父目录来访问它本身，若它还有子目录，它的引用数也会增加
> - 删除磁盘上文件的时候，只有引用计数为0时候，才会将磁盘内容移除文件系统（断开和目录的链接）
> - 当然，为了避免引用死锁，一般用户是不能使用ln命令来为目录建立硬链接

在Linux当中，目录是一种特别的文件，它的总体大小相对固定，其文件内容是把很多文件的文件名和索引结点存放在一起组织起来。因为一个目录下面的各个文件的文件名大小不一，也需要支持频繁增加修改操作，所以目录文件的内容就是一个链表，链表中的每个结点称为**目录项(dirent)**。目录项内包含了某个目录下某个子文件的信息，其定义如下，可以看出，要访问下个dirent结点，实际是依赖于本结点中d_off属性。

```
struct dirent{
    ino_t d_ino;        //该文件的inode
    off_t d_off;        //到下一个dirent的偏移
    unsigned short d_reclen;//文件名长度
    unsigned char d_type;  //所指的文件类型
    char  d_name[256];   //文件名
};
```



4.2.4 目录流相关操作

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);   //打开一个目录流
struct dirent *readdir(DIR *dir);  //读取目录流当中的一个目录项，
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
```

- 使用`opendir`系统调用可以在内存中创建**目录流**数据结构用来处理和目录相关的操作。目录流底层是一种链表结构，其中链表的结点就是目录项。
- 此外，目录流当中还存在一个当前读写位置指针，每一次使用`readdir`函数之后，会取得读写位置指针所指向的目录项，并且读写位置指针会自动后移。
- 使用`closedir`函数可以关闭目录流。

> dirent当中采用了类似于变长数组的形式来存放文件名，但是会提供一些冗余的空间，这样当调整文件名的时候，如果新文件名的长度不超过原来分配的空间，则不需要调整分配的空间

下面是一个使用目录流的例子，它实现了类似于tree命令的效果：

```
//使用深度优先遍历访问目录的例子
int printDir(char *path)
{
    DIR* pdir = opendir(path);
    ERROR_CHECK(pdir,NULL,"opendir");
    struct dirent *pdirent;
    char buf[1024];//注意递归时传递的路径是否合理
    while((pdirent = readdir(pdir)))
    {
        if(strcmp(pdirent->d_name,".") == 0 || strcmp(pdirent->d_name,"..") == 0)
        {
            continue;
        }
        printf("%s\n",pdirent->d_name);
        sprintf(buf,"%s%s%s",path,"/",pdirent->d_name);//这里不需要担心斜杠太多的问题
        if(pdirent->d_type == 4)
        {
            printDir(buf);
        }
    }
    closedir(pdir);
    return 0;
}
int tabPrintDir(char *path,int width)//这里实现了类似tree的效果
{
    DIR* pdir = opendir(path);
    ERROR_CHECK(pdir,NULL,"opendir");
    struct dirent *pdirent;
    char buf[1024];
    while((pdirent = readdir(pdir)))
    {
```

```
        if(strcmp(pdirent->d_name,".") == 0 || strcmp(pdirent->d_name,"..") == 0)
        {
            continue;
        }
        printf("%*s%s\n",width,"",pdirent->d_name);
        sprintf(buf,"%s%s%s",path,"/",pdirent->d_name);
        if(pdirent->d_type == 4)
        {
            tabPrintDir(buf,width+4);
        }
    }
    closedir(pdir);
    return 0;
}

int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    puts(argv[1]);
    printDir(argv[1]);
    tabPrintDir(argv[1],0);
    return 0;
}
```

- `seekdir`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。
- 使用`readddir`时，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。
- `telldir`函数用来返回目录流当前的读取位置

```
//下面是一个例子
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    DIR* pdir = opendir(argv[1]);
    ERROR_CHECK(pdir,NULL,"opendir");
    struct dirent *pdirent;
    off_t pos;
    while((pdirent = readdir(pdir)))
    {
        printf("ino = %ld len = %d type = %d filename = %s\n",pdirent-
>d_ino,pdirent->d_reclen,pdirent->d_type,pdirent->d_name);
        if(strcmp(pdirent->d_name,"a.out") == 0)
        {
            pos = telldir(pdir);//只会成功，不会失败
        }
    }
    //seekdir(pdir,pos);
    rewinddir(pdir);
    pdirent = readdir(pdir);
    printf("-------------------------\n");
    printf("ino = %ld len = %d type = %d filename = %s\n",pdirent->d_ino,pdirent-
>d_reclen,pdirent->d_type,pdirent->d_name);
    return 0;
```

```
}
```

使用`stat`系统调用可以获取文件的详细信息，相关的信息存入stat结构体当中。

```
// man 2 stat
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *pathname, struct stat *statbuf);
//结构体stat的定义
  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 */
};
```

下面的例子实现了一个类似于ls -al命令的效果：

```
//示例
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int ret;
    struct stat buf;
    ret = stat(argv[1],&buf);
    ERROR_CHECK(ret,-1,"stat");

    printf("%x %ld %s %s %ld %s\n",buf.st_mode
                                  ,buf.st_nlink
                                  ,getpwuid(buf.st_uid)->pw_name
                                  ,getgrgid(buf.st_gid)->gr_name
                                  ,buf.st_size,ctime(&buf.st_mtime));
    return 0;
}
```

> 注意getpwuid和getgrgid需要包含头文件pwd.h和grp.h

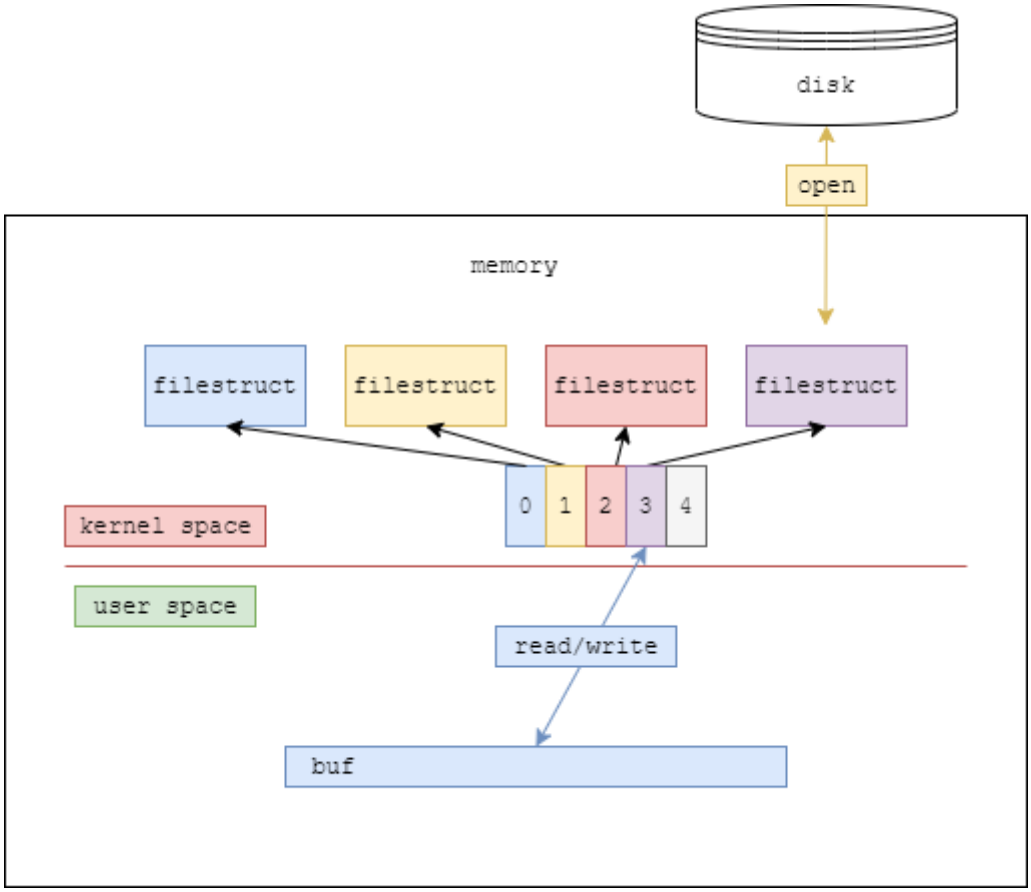
4.3 基于文件描述符的文件操作

4.3.1 文件描述符

之前所讨论的文件操作都是操作文件流，即FILE。我们把所有和FILE类型相关的文件操作（比如fopen，fread等等）称为**带缓冲的IO**，它们是ISO C的组成部分，它们都是库函数，其底层调用了系统调用来使用内核的功能。

POSIX标准支持另一类**无缓冲的IO**，这些操作都是系统调用。值得注意的是，在这里无缓冲是没有分配**用户态文件缓冲区**的意思。在操作文件时，**进程**会在内存地址空间的内核区部分里面**维护一个数据结构**来管理和文件相关的所有操作，这个数据结构称为打开文件或者是**文件对象（file / file struct）**，除此以外，内核区里面还会维护一个索引数组来管理所有的文件对象，该数组的下标就被称为**文件描述符(file descriptor)。**

从类型来说，文件描述符是一个非负整数，它可以传递给用户。用户在获得文件描述符之后可以定位到相关联的文件对象，从而可以执行各种IO操作。



4.3.2 打开、创建和关闭文件

```
#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 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。
```

- `open`可以打开一个已存在的文件或者创建一个新文件，并创建一个文件对象，返回相关的文件描述符。

- 使用完文件以后，要记得使用`close`来关闭文件。一旦调用`close`，会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。

flags的数据类型是int，但是在这里并非当作一个普通整数来使用，用户需要把这个int看成是32个bit的集合，不同的bit用来描述不同的信息。比如最低位的两个bit就用来描述读写权限。

```
#define O_RDONLY         00
#define O_WRONLY         01
#define O_RDWR           02
```

由上可知，如果需要同时组合两个无关权限，那么就可以采用按位或的操作。

常见的flags的可选项如下：

选项	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 O_NDELAY	对管道、设备文件和socket使用，以非阻塞方式打开文件，无论有无数据读取或等待，都会立即返回进程之中

当flags选项当中存在O_CREAT选项时，`open`就应该选择3参数的版本，其中第三个参数mode说明了要创建的文件的权限，通常会直接填入一个八进制的整型字面值，例如：

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

下面是一个简单的打开文件的例子：

```c
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDONLY);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n", fd);
    close(fd);
    return 0;
}
```

在使用`open`系统调用的时候，内核会按照**最小可用**的原则分配一个文件描述符。一般情况下， 进程一经启动就会打开3个文件对象，占用了0、1和2文件描述符，分别关联了标准输入、标准输出和标准错误输出，所以此时再打开的文件占用的文件描述符就是3。

4.3.3 基于文件描述符的读写文件操作

基本的读写操作

```c
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);//文件描述符 缓冲区 缓冲区长度上限
ssize_t write(int fd, const void *buf, size_t count);//文件描述符 缓冲区 内容长度
```

使用`read`和`write`系统调用可以读写文件，它们统称为不带缓冲的IO。

- `read`的原理是将数据从文件对象内部的内核文件缓冲区拷贝出来（这部分的数据最初是在外部设备中，通过硬件的IO操作拷贝到内存之上）到用户态的buf之中。
- 而`write`就相反，它将数据从用户态的buf当中拷贝到内核区的文件对象的内核文件缓冲区，并最终会写入到设备中。
- 类似于FILE当中的读写行为，每次调用`read`或者`write`会影响其下次读写位置，值得注意的是，这个读写位置信息是保存在内核文件对象当中的。

下面是一个简单的使用`read`/`write`的例子：

```c
//读取文件内容
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n",fd);
    char buf[128] = {0};
    int ret = read(fd, buf, sizeof(buf));
    printf("buf = %s, ret = %d\n", buf, ret);
    close(fd);
    return 0;
}

//写入一个int数据
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
```

```
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n",fd);
    int i = 3;
    int ret = write(fd, &i, sizeof(i));
    printf("ret = %d\n",ret);
    close(fd);
    return 0;
}
//如果希望查看文件的2进制信息，在vim中输入命令:%!xxd
//使用od -h 命令也可查看文件的16进制形式
```

读磁盘文件和读设备文件的区别

值得注意的是，使用`read`读取磁盘文件和设备文件的行为是有着明显差异的：

- 当读取磁盘文件/设备文件的时候，如果文件/设备缓冲区剩余内容长度超过count，那么本次`read`会读取count个字节；
- 当读取磁盘文件/设备文件的时候，如果文件/设备缓冲区存在剩余内容，且长度不足count，那么本次`read`会读取文件剩余内容；

```
int main(int argc, char *argv[])
{
    // echo -n hello > file1
    // file1里面有5个字节的内容
    // ./read file1
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    char buf[6] = {0};
    ssize_t sret = read(fd,buf,5);
    ERROR_CHECK(sret,-1,"read");
    printf("buf = %s, sret = %ld\n",buf,sret);
    memset(buf,0,sizeof(buf));//每次read操作之前先清空
    sret = read(fd,buf,5);
    ERROR_CHECK(sret,-1,"read");
    printf("buf = %s, sret = %ld\n",buf,sret);
    close(fd);
    return 0;
}
```

- 当读取磁盘文件的时候，如果文件无剩余内容，那么本次`read`操作会立刻返回，返回值为0；
- 当读取设备文件的时候，如果设备缓冲区无剩余内容，那么本次`read`会阻塞。

```
int main()
{
    char buf[1024] = {0};
    //STDIN_FILENO 就是 0
    ssize_t sret = read(STDIN_FILENO,buf,sizeof(buf));
    ERROR_CHECK(sret,-1,"read");
    printf("sret = %ld, buf = %s\n", sret, buf);
    return 0;
}
```

读写二进制文件

使用`read`和`write`可以读写文本文件，也可以读写二进制文件。

- 如果操作的数据全是字符串，那么就是操作文本数据；
- 如果操作的数据存在非字符串类型，比如int、double等其他类型，那么就是操作二进制数据；
- 操作二进制数据遵循一个原则：**“按什么类型写入，就按什么类型读出！”**

```
//先写入数据到文件当中
int main(int argc, char *argv[])
{
    // ./write_binary file1
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    int data = 100000;
    ssize_t sret = write(fd,&data,sizeof(int));
    ERROR_CHECK(sret,-1,"write");
    printf("sret = %ld\n", sret);
    close(fd);
    return 0;
}
//从二进制文件当中进行读取
int main(int argc, char *argv[])
{
    // ./read_binary file1
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    int data;
    ssize_t sret = read(fd,&data,sizeof(int));
    ERROR_CHECK(sret,-1,"read");
    printf("sret = %ld\n", sret);
    ++data;
    printf("data = %d\n", data);
    close(fd);
    return 0;
}
```

通过读写操作实现拷贝

通过组合`read`和`write`，可以实现类似cp命令的效果：

```
int main(int argc, char *argv[])
{
    // ./cp src dest
    ARGS_CHECK(argc,3);
    int fdr = open(argv[1],O_RDONLY);
    ERROR_CHECK(fdr,-1,"open fdr");
    int fdw = open(argv[2],O_WRONLY|O_TRUNC|O_CREAT,0666);
    ERROR_CHECK(fdw,-1,"open fdw");
    //char buf[4096] = {0};
    char buf[4096000] = {0};
    // buf选择char数组，不是字符串的含义，而是因为char的字节是1
```

```
    while(1){
        memset(buf,0,sizeof(buf));
        ssize_t sret = read(fdr,buf,sizeof(buf));
        ERROR_CHECK(sret,-1,"read");
        // 读取磁盘文件，返回值为0，则读完
        if(sret == 0){
            break;
        }
        // 写入dest
        write(fdw,buf,sret);
    }
    close(fdr);
    close(fdw);
    return 0;
}
```

在上述代码cp命令的实现之中，我们可以调整buf数组的长度来影响的程序执行的效率。经过测试，buf的长度越大，则整个程序的执行效率越高，其原因也很简单，`read`/`write`是系统调用，每次执行都需要一段时间来让硬件的状态在用户态和内核态之间切换，在文件长度固定的情况下，buf长度越大，`read`/`write`的执行次数越少，自然效率就越高。

> 关于buf长度对性能的影响，以及read/write和fread/fwrite的性能对比，可以参考《UNIX环境高级编程》第3.9和第5.8节。

4.3.4 文件偏移

系统调用`lseek`可以（内核中的）文件对象的文件读写偏移量设定到以whence为启动，偏移值为offset的位置。它的返回值是读写点距离文件开始的距离。（所以`lseek`其实可以用来获取文件的大小）

```
#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
```

```
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    int ret = lseek(fd, 5, SEEK_SET);
    printf("pos = %d\n", ret);
    char buf[128] = {0};
    read(fd, buf, sizeof(buf));
    printf("buf = %s\n", buf);
    close(fd);
    return 0;
}
```

4.3.5 截断文件

使用`ftruncate`函数可以截断文件，从而控制文件的大小。

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

`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为文件描述符，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；如果原来的文件大小比参数length小，文件将会扩充到大小为length，多余的数据填二进制0。

示例：

```
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_WRONLY);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n",fd);
    off_t length = 3;
    int ret = ftruncate(fd,length);
    ERROR_CHECK(ret,-1,"ftruncate");
    return 0;
}
```

假如在上述的例子，把length的长度设置得特别大（比如40960或者更大），然后我们使用stat命令来查看文件实际所分配的磁盘空间大小，会发现文件大小居然会大于分配的磁盘空间大小。这就意味着，文件已经占用了文件系统当中的空间，但是底层磁盘还没有为其分配真正的磁盘块，这就是**文件空洞**。

4.3.6 文件映射

文件映射基本使用

使用`mmap`系统调用可以实现**文件映射**功能，也就是将一个磁盘文件直接映射到内存用户态地址空间的一片区域当中，这样的话，内存内容就和磁盘文件内容一一对应，也不再需要使用`read`和`write`系统调用就可以进行IO操作，直接读写内存数据即可。

需要注意的是，`mmap`不能修改文件的大小，所以需要经常配合函数`ftruncate`来使用。

```
#include <sys/mman.h>
void *mmap(void *adr, size_t len, int prot, int flag, int fd, off_t off);
```

- addr参数用于指定映射存储区的起始地址。这里设置为NULL，这样就由系统自动分配（一般是分配在堆空间里面）；
- fd参数是文件描述符，用来指示要建立映射的文件；
- prot参数用来表示权限，其中PROT_READ|PROT_WRITE表示可读可写（open的flag应该填写O_RDWR）；
- flag参数在目前是采用MAP_SHARED，后面讲解进程通信的时候会介绍其他类型。

下面是使用`mmap`的例子：

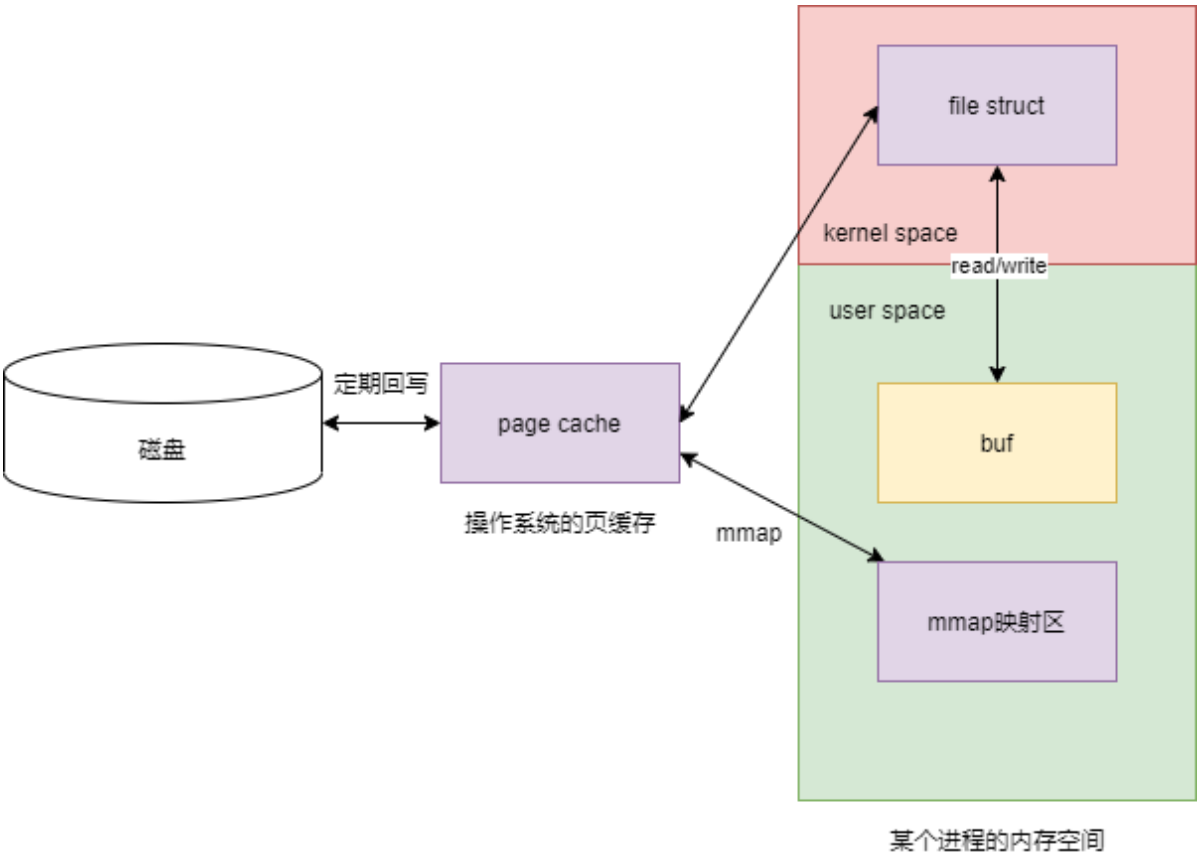
```
//假设文件本身的内容是hello
```

```c
int main(int argc, char *argv[])
{
    // ./mmap file1
    ARGS_CHECK(argc,2);
    // 先 open 文件
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    // 建立内存和磁盘之间的映射
    char *p = (char *)mmap(NULL,5,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
    ERROR_CHECK(p,MAP_FAILED,"mmap");//mmap失败返回不是NULL
    for(int i = 0; i < 5; ++i){
        printf("%c", *(p+i));
    }
    printf("\n");
    *(p+4) = 'O';
    munmap(p,5);
    close(fd);
    return 0;
}
```

文件映射的底层原理

一个经常讨论的问题就是使用`mmap`和`read/write`的效率到底谁更高？这个问题的答案依赖问题发生所在的场景。从原理上来说，`read/write`是让数据在内核态的文件对象和用户态内存之间进行来回拷贝，文件对象会和一片由操作系统管理的内存区域（被称为页缓存）相关联，一般来说，操作系统会选择一个合适策略并使用专门的硬件(比如DMA设备)来同步磁盘和页缓存当中的内容，这样`read/write`操作最终就会影响到磁盘。而`mmap`的处理就更加简单粗暴，它直接把页缓存的一部分映射到用户态内存，这样用户在用户态当中的操作就直接对应页缓存的操作。

这样看上去的话，`mmap`的效率总是会比`read/write`更加高，因为它避免了一次数据在用户态和内核态之间的拷贝。但是考虑到`read/write`的特殊性质——它们总是顺序地而不是随机地访问磁盘文件的内容，所以操作系统可以根据这个特点进行优化，比如文件内容的预读等等，最终经过测试——`read/write`在顺序读写的时候性能更好，而`mmap`在随机访问的时候性能更好。



4.3.7 文件流和文件描述符的关系

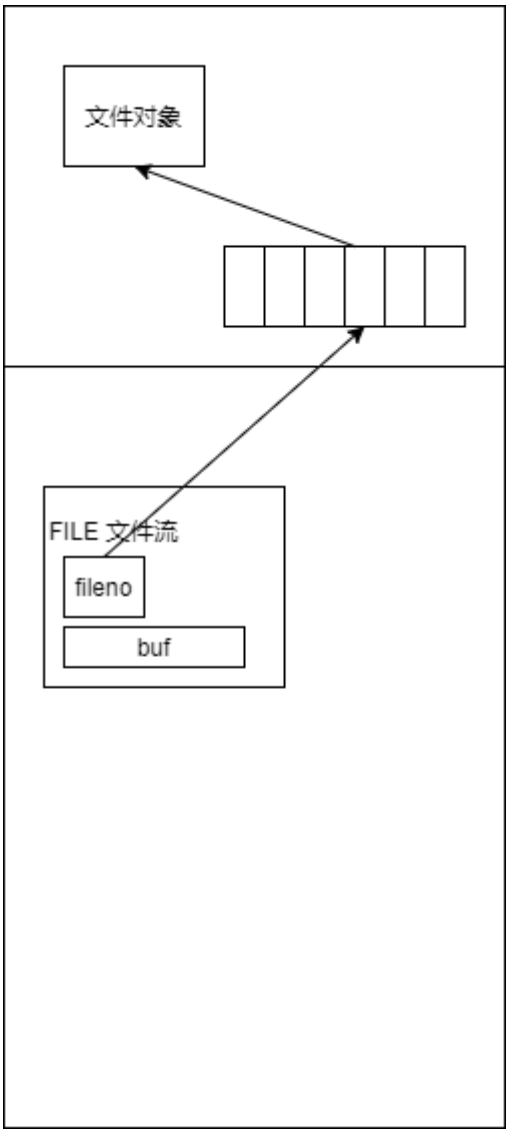
文件流的底层

`fopen`函数实际在运行的过程中也获取了文件的文件描述符。使用`fileno`函数可以得到文件流的文件描述符。在使用`fopen`打开文件流之后，依然是可以使用文件描述符来执行IO的，例如

```
int main(int argc, char *argv[])
{
    // ./fileno file1
    ARGS_CHECK(argc,2);
    FILE * fp = fopen(argv[1],"r+");
    ERROR_CHECK(fp,NULL,"fopen");
    //write(3,"hello",5);
    //write(fp->_fileno,"world",5);
    write(fileno(fp),"hello",5);
    fclose(fp);
    return 0;
}
```

`fopen`的原理：`fopen`函数在执行的时候，会先调用`open`函数，打开文件并且获取文件对象的信息（通过文件描述符可以获取文件对象的具体信息），然后`fopen`函数会在用户态空间申请一片空间作为缓冲区；

`fopen`的优势：因为`read`和`write`是系统调用，需要频繁地切换用户态和内核态，所以比较耗时。借助用户态缓冲区，可以减少`read`和`write`的次数。



从另一方面来说，如果需要高效地使用不带缓冲IO，为了和存储体系匹配，最好是一次读取/写入一个块大小的数据。如果获取了文件指针，就不要通过文件描述符的方式来关闭文件

标准输入输出流

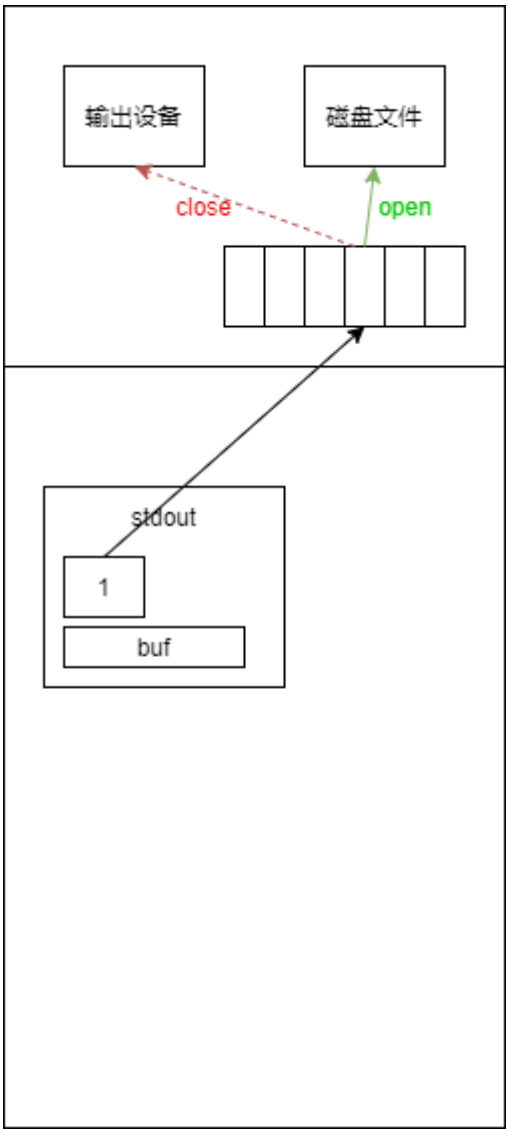
在每个进程启动伊始，都会自动打开三个文件流，就是标准输入流stdin、标准输出流stdout、标准错误输出流stderr。而与标准的输入输出流对应的在更底层的实现是用标准输入、标准输出、标准错误文件描述符表示的。它们分别用STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO三个宏表示，值分别是0、1、2三个整型数字。

```
int main()
{
    printf("stdin fd = %d\n", fileno(stdin));
    printf("stdout fd = %d\n", fileno(stdout));
    printf("stderr fd = %d\n", fileno(stderr));
    return 0;
}
```

通过更改文件描述符所对应的文件对象，可以实现重定向功能：

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

```
    // ./13_redirect file1
    ARGS_CHECK(argc,2);
    // 在关闭1之前请先打印一个换行符
    printf("You can see me!\n");
    //close(1);
    close(STDOUT_FILENO);
    int fd = open(argv[1],O_WRONLY);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n", fd);
    printf("You can't see me!\n");
    return 0;
}
```



4.3.8 文件描述符的复制

dup和dup2

系统调用函数 `dup` 和 `dup2` 可以实现文件描述符的复制。所谓文件描述符的复制并不是简单地拷贝一份文件描述符的整数值，而是使用一个新的文件描述符去引用同一个文件对象。

- `dup` 返回一个新的文件描述符，该文件描述符是自动分配的，数值是没有使用的文件描述符的最小编号；

- `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（执行完成以后，如果newfd已经打开了文件，该文件将会被关闭）

原型为：

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

下面是使用 `dup` 的例子：

```
int main(int argc, char *argv[])
{
    // ./dup file1
    ARGS_CHECK(argc,2);
    int oldfd = open(argv[1],O_RDWR);
    ERROR_CHECK(oldfd,-1,"open");
    printf("oldfd = %d\n",oldfd);
    int newfd = dup(oldfd);
    ERROR_CHECK(newfd,-1,"dup");
    printf("newfd = %d\n", newfd);
    // 新旧文件描述符数值不相同
    // 引用的文件对象相同，共享偏移量
    write(oldfd,"hello",5);
    write(newfd,"world",5);
    close(newfd);
    close(oldfd);
    return 0;
}
```

更加精细的重定向

使用 `dup` 函数可以实现输出重定向。

```
#include <func.h>
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("\n");
    close(STDOUT_FILENO);
    int fd1 = dup(fd);
    printf("fd1 = %d\n", fd1);
    close(fd);
    printf("the out of stdout\n");
    return 0;
}
```

下面是该程序的详细流程：

- 首先打开了一个文件对象，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，用close则表示关闭标准输出；
- 此时文件描述符1就空着，然后调用dup,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；
- 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，printf所输出的内容也就直接输出到文件了。

使用`dup2`可以构建更加复杂的重定向例子。

```
//更加复杂的例子
int main(int argc, char *argv[])
{
    // ./redirect file1
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_WRONLY);
    ERROR_CHECK(fd,-1,"open");
    printf("我过来啦！\n");
    int newfd = 10;
    dup2(STDOUT_FILENO,newfd);//使用newfd备份输出文件对象
    //让1引用磁盘文件
    dup2(fd,STDOUT_FILENO);
    printf("我过去啦！\n");
    //让1引用输出设备
    dup2(newfd,STDOUT_FILENO);
    printf("我又回来啦！\n");
    close(fd);
    return 0;
}
```

4.3.9 有名管道

（有名）管道文件是一种特殊类型的文件，它是进程间通信机制在文件系统当中的映射。管道采用半双工的通信方式，它在ls -l命令中类型显示为p。管道文件不同于普通的磁盘文件，它只能暂存数据，而不能持久存储数据。

传输方式	含义
全双工	双方可以同时向另一方发送数据
半双工	某个时刻只能有一方向另一方发送数据，其他时刻的传输方向可以相反
单工	永远只能由一方向另一方发送数据

```
$ mkfifo [管道名字]
使用cat打开管道可以打开管道的读端
$ cat [管道名字]
打开另一个终端，向管道当中输入内容可以实现写入内容
$ echo “string” > [管道名字]
此时读端也会显示内容
```

当然也可自己写编程程序来分别实现读端和写端。

- `open`可以用来打开管道的一端，O_RDONLY表示打开读端，O_WRONLY表示打开写端；
- 写端打开读端未打开时，或者是读端打开写端未打开时，进程会陷入阻塞状态；
- 当读写都打开之后，可以使用`read/write`进行读写操作；
- `write`暂时是不会触发阻塞的；
- 而`read`管道非常类似于`read`设备文件，如果管道当中没有数据，进程就会陷入阻塞。

```
//读端
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fdr = open(argv[1],O_RDONLY);
    ERROR_CHECK(fdr,-1,"open");
    printf("fdr = %d\n",fdr);
    char buf[128] = {0};
    read(fdr, buf, sizeof(buf));
    printf("buf = %s\n", buf);
    return 0;
}
//写端
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fdw = open(argv[1],O_WRONLY);
    ERROR_CHECK(fdw,-1,"open");
    printf("fdw = %d\n",fdw);
    char buf[128] = "helloworld";
    write(fdw, buf, strlen(buf));
    printf("buf = %s\n", buf);
    return 0;
}
```

> 请不要使用vim来打开管道文件

在管道的两端打开以后，任意一个进程都可以关闭管道，其表现如下：

- 若管道的写端先关闭，则之后管道的读端执行`read`操作时会立刻返回，且返回值为0；
- 若管道的读端先关闭，则之后管道的写端执行`write`操作时会触发SIGPIPE信号，导致进程异常终止。

4.4 I/O多路复用模型

4.4.1 简陋的即时聊天存在的问题

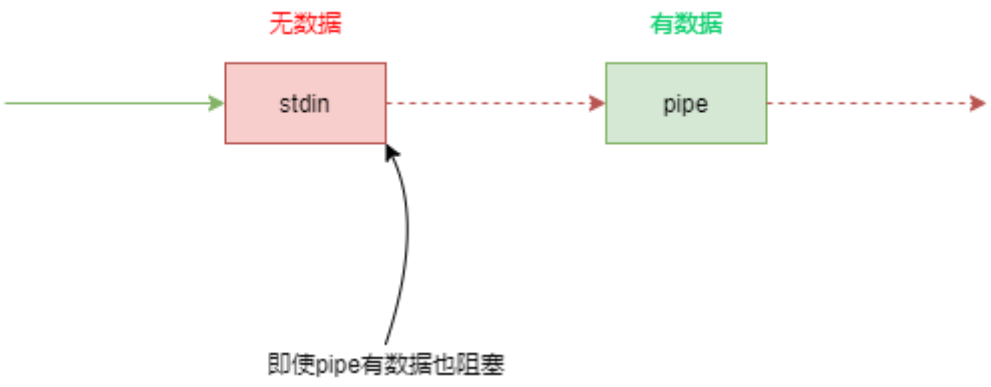
在使用管道文件时，用户通常会把当管道文件作单工通信来使用。这样的情况下，想要实现全双工通信，用户就可以使用2个管道文件。下面是一个利用两个管道实现全双工通信（一个简陋的即时聊天）的例子。

```
//1号
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,3);
    int fdr = open(argv[1],O_RDONLY);//管道打开的时候，必须要先将读写端都打开之后才能继续
```

```
    int fdw = open(argv[2],O_WRONLY);
    printf("I am chat1\n");
    char buf[128] = {0};
    while(1)
    {
        memset(buf,0,sizeof(buf));
        read(STDIN_FILENO, buf, sizeof(buf));
        write(fdw, buf, strlen(buf)-1);
        memset(buf,0,sizeof(buf));
        read(fdr, buf, sizeof(buf));
        printf("buf = %s\n", buf);
    }
    return 0;
}
//2号
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,3);
    int fdw = open(argv[1],O_WRONLY);//管道打开的时候，必须要先将读写端都打开之后才能继续
    int fdr = open(argv[2],O_RDONLY);
    printf("I am chat2\n");
    char buf[128] = {0};
    while(1)
    {
        memset(buf,0,sizeof(buf));
        read(fdr, buf, sizeof(buf));
        printf("buf = %s\n", buf);
        memset(buf,0,sizeof(buf));
        read(STDIN_FILENO, buf, sizeof(buf));
        write(fdw, buf, strlen(buf)-1);
    }
    return 0;
}
```

在上述例子运行的时候，有时候会出现**阻塞**的问题：一般的阻塞是指进程在执行`read`时，假如缓冲区当中没有任何数据，进程的状态就会切换到阻塞状态，不再继续运行。而上述例子当中的阻塞会情况更加复杂一些，即使输入设备缓冲区或者管道读缓冲区当中的某一个存在数据时，整个进程也会处于阻塞状态。

即使有数据也出现阻塞的原因其实并不复杂，要么是进程`read`管道文件时，输入设备缓冲区有数据而管道读缓冲区无数据；要么就是进程`read`输入设备时，输入设备缓冲区无数据而管道读缓冲区有数据——究其根本，就是多个`read`的**等待是串行的而不是并行**的。



4.4.2 I/O多路复用模型和select

想要解决上述多个资源的读取操作引发的阻塞问题，一种自然的思路就是将串行的等待改造成并行的等待。操作系统提供了一种**I/O多路复用机制**来实现这种并行的等待：当进程使用I/O多路复用系统调用时，进程陷入等待状态；而此时操作系统可以同时监听多个文件描述符是否就绪（也就是文件缓冲区当中是否存在数据）；当监听的多个文件描述符中存在至少一个文件描述符就绪的时候，操作系统可以唤醒进程，随后进程就可以遍历就绪的文件描述符，再执行读取操作（这个读取操作就不会引发阻塞了）。

本节要介绍的`select`函数，就是I/O多路复用模型的其中一种实现，相关的函数定义如下：

```
#include <sys/select.h>
#include <sys/time.h>
//readset、writeset、exceptionset都是fd_set集合
//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 */
int select(int maxfd, fd_set *readset,fd_set *writeset, fd_set *exceptionset,
struct timeval * timeout);
```

这里先简要地介绍一下`select`使用流程：

- 首先，需要先为监听集合申请内存；
- 使用`FD_ZERO`初始化监听集合；
- 将所有需要监听的文件描述符使用`FD_ZERO`加入监听集合；
- 调用`select`系统调用使进程陷入阻塞状态；
- 从阻塞当中被唤醒以后，可以遍历所有监听的文件描述符，找到真正就绪的文件描述符；
- 对就绪的文件描述符执行IO操作。

关于`select`的各个参数，其含义如下：

- maxfd：所有监听集合当中数值最大的文件描述符再+1；
- readset：用来监听读事件的集合，当readset当中的文件描述符所对应的文件缓冲区有至少一个不为空时，`select`会返回并保留所有就绪的文件描述符（**这意味着`select`函数会修改readset的内容**）；
- writeset：用来监听写事件的集合，当writeset当中的文件描述符所对应的文件缓冲区有至少一个为空时，`select`会返回并保留所有就绪的文件描述符；
- exceptionset：用来监听异常事件的集合；
- timeout：描述超时时间。NULL代表永远等下去，而一个固定值代表最多等待的时间。

下面我们使用I/O多路复用机制`select`来改造之前的即时聊天系统：首先，通信的两端需要建立两个管道来实现通信，一个管道用于发送数据，另一个管道用于接收数据；对于通信的某一端，有两个文件描述符是可能会引发阻塞的，一个是管道的写端，另一个是标准输入；因此，就需要监听这两个文件描述符。下面是示例：

```
int main(int argc, char *argv[])
{
    // ./chat1 1.pipe 2.pipe
```

```
    ARGS_CHECK(argc,3);
    int fdr = open(argv[1],O_RDONLY);
    ERROR_CHECK(fdr,-1,"open fdr");
    int fdw = open(argv[2],O_WRONLY);
    ERROR_CHECK(fdw,-1,"open fdw");
    printf("chat is established!\n");
    char buf[4096];
    // 创建一个监听集合
    fd_set rdset;
    while(1){ // 注意初始化和增加监听操作每次循环都必须执行!
        // 初始化
        FD_ZERO(&rdset);
        // 把管道读端和stdin加入监听
        FD_SET(STDIN_FILENO,&rdset);
        FD_SET(fdr,&rdset);
        // 调用select函数，使进程阻塞
        select(fdr+1,&rdset,NULL,NULL,NULL);
        // select返回以后，rdset里面是本次的就绪集合
        if(FD_ISSET(STDIN_FILENO,&rdset)){
            //stdin就绪
            memset(buf,0,sizeof(buf));
            ssize_t sret = read(STDIN_FILENO,buf,sizeof(buf));
            if(sret == 0){
                write(fdw,"nishigehaoren",13);
                break;
            }
            write(fdw,buf,strlen(buf));
        }
        if(FD_ISSET(fdr,&rdset)){
            //fdr就绪
            memset(buf,0,sizeof(buf));
            ssize_t sret = read(fdr,buf,sizeof(buf));
            if(sret == 0){
                printf("Hehe!\n");
                break;
            }
            printf("buf = %s\n", buf);
        }

    }
    close(fdw);
    close(fdr);
    return 0;
}

int main(int argc, char *argv[])
{
    // ./chat2 1.pipe 2.pipe
    ARGS_CHECK(argc,3);
    int fdw = open(argv[1],O_WRONLY);
    ERROR_CHECK(fdw,-1,"open fdw");
    int fdr = open(argv[2],O_RDONLY);
    ERROR_CHECK(fdr,-1,"open fdr");
    printf("chat is established!\n");
    char buf[4096];
    // 创建一个监听集合
```

```
    fd_set rdset;
    while(1){
        // 初始化
        FD_ZERO(&rdset);
        // 把管道读端和stdin加入监听
        FD_SET(STDIN_FILENO,&rdset);
        FD_SET(fdr,&rdset);
        select(fdr+1,&rdset,NULL,NULL,NULL);
        // select返回以后，rdset里面是本次的就绪集合
        if(FD_ISSET(STDIN_FILENO,&rdset)){
            //stdin就绪
            memset(buf,0,sizeof(buf));
            read(STDIN_FILENO,buf,sizeof(buf));
            write(fdw,buf,strlen(buf));
        }
        if(FD_ISSET(fdr,&rdset)){
            //fdr就绪
            memset(buf,0,sizeof(buf));
            read(fdr,buf,sizeof(buf));
            printf("buf = %s\n", buf);
        }
    }
    close(fdr);
    close(fdw);
    return 0;
}
```

4.4.3 select的退出机制

当管道写端先关闭的时候，之后对面的管道读端调用`read`会立刻返回——这就意味着，当管道写端关闭以后，管道读端会一直处于就绪状态，这种就绪的状态会影响到`select`函数，导致`select`函数不会阻塞，并最终造成了进入死循环。

我们可以简单地改写程序来兼容对方退出的情况，就是当read的返回值为0的时候，就退出程序：

```
    ...
        if(FD_ISSET(STDIN_FILENO, &rdset))
        {
            memset(buf,0,sizeof(buf));
            read_ret = read(STDIN_FILENO, buf, sizeof(buf));
            if(read_ret == 0)
            {
                printf("chat is broken!\n");
                break;
            }
            write(fdw, buf, strlen(buf)-1);
        }
        if(FD_ISSET(fdr, &rdset))
        {
            memset(buf,0,sizeof(buf));
            read_ret = read(fdr, buf, sizeof(buf));
            if(read_ret == 0)
            {
                printf("chat is broken!\n");
```

```
                break;
            }
            printf("buf = %s\n", buf);
        }
    ...
```

4.4.4 select函数的超时处理

使用timeval结构体可以设置超时时间。传入select函数中的timeout参数是一个timeval结构体指针，timeval结构体的定义如下：

```
struct timeval
{
    long tv_sec;//秒
    long tv_usec;//微秒
};
//用法
    ...
    struct timeval timeout;
    while(1)
    {
        bzero(&timeout, sizeof(timeout));
        timeout.tv_sec = 3;
        ret = select(fdr+1, &rdset, NULL, NULL, &timeout);
        if(ret > 0)
        {
            ...
        }
        else
        {
            printf("time out!\n");
        }
    }
```

使用的超时判断的时候要注意，每次调用`select`之前需要重新为timeout赋值，因为调用`select`会修改timeout里面的内容

4.4.5 写集合的原理

写阻塞和写就绪：当管道的写端向管道中写入数据达到上限以后，后续的写入操作将会导致进程进入阻塞态，称为写阻塞现象，而处于写阻塞状态以后，当管道中的数据被读端读取以后，写端就可以恢复写入操作，称为写就绪。

```
// 写端一直写
int main(int argc, char *argv[])
{
    // ./write 1.pipe
    ARGS_CHECK(argc,2);
    int fdw = open(argv[1],O_WRONLY);
    ERROR_CHECK(fdw,-1,"open");
    char buf[4096] = {0};
    ssize_t total = 0;
    while(1){
```

```
        ssize_t sret = write(fdw,buf,sizeof(buf));
        ERROR_CHECK(sret,-1,"write");
        total += sret;
        printf("sret = %ld, total = %ld\n", sret, total);
    }
    return 0;
}

//读端读得慢
int main(int argc, char *argv[])
{
    // ./read 1.pipe
    ARGS_CHECK(argc,2);
    int fdr = open(argv[1],O_RDONLY);
    ERROR_CHECK(fdr,-1,"open");
    char buf[1024] = {0};
    while(1){
        sleep(3);
        read(fdr,buf,sizeof(buf));
    }
    close(fdr);
    return 0;
}
```

类似于读文件描述符集合，select也可以设置专门的写文件描述符集合，select可以监听处于写阻塞状态下的文件，一旦文件转为写就绪，就可以将进程唤醒

```
int main(int argc, char* argv[])
{
    ARGS_CHECK(argc, 2);
    int fdr = open(argv[1],O_RDWR);//非阻塞方式open管道的一端
    int fdw = open(argv[1],O_RDWR);//可以一次性打开管道的读写端
    fd_set rdset,wrset;
    int ret;
    char buf[128];
    while(1)
    {
        FD_ZERO(&rdset);
        FD_ZERO(&wrset);
        FD_SET(fdr, &rdset);
        FD_SET(fdw, &wrset);
        ret = select(fdw+1, &rdset, &wrset, NULL, NULL);
        if(FD_ISSET(fdr, &rdset))
        {
            bzero(buf, sizeof(buf));
            read(fdr, buf, sizeof(buf));
            puts(buf);
            usleep(250000);

        }
        if(FD_ISSET(fdw, &wrset))
        {
            write(fdw,"helloworld", 10);
```

```
            usleep(500000);
        }
    }
}
```