实验 3-补充　文件 I/O 编程

一、　Linux 中文件及文件描述符概述

在Linux 中对目录和设备的操作都等同于文件的操作，因此，大大简化了系统对不同设备的处理，提高了效率。**Linux 中的文件主要分为6种：普通文件、目录文件、链接文件、管道文件、套接字文件和设备文件**。

那么，**内核如何区分和引用特定的文件**呢？这里用到了一个重要的概念——**文件描述符**。对于Linux 而言，所有对设备和文件的操作都是使用文件描述符来进行的。文件描述符是一个非负的整数，它是一个**索引值**，并指向在内核中**每个进程打开文件的记录表**。当打开一个现存文件或创建一个新文件时，内核就向进程返回一个文件描述符；当需要读写文件时，也需要把文件描述符作为参数传递给相应的函数。

通常，一个进程启动时，都会打开3 个文件：标准输入、标准输出和标准出错处理。这3 个文件分别对应文件描述符为0、1 和2（也就是宏替换STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO）。

基于文件描述符的I/O操作是Linux 中最常用的操作之一，基于文件描述符的I/O操作虽然不能移植到类Linux 以外的系统上去（如Windows），但它往往是实现某些I/O操作的惟一途径，如Linux中低级文件操作函数、多路I/O、TCP/IP套接字编程接口（socket接口）等。同时，它们也很好地兼容POSIX标准，因此，可以很方便地移植到任何POSIX平台上。

二、　文件 I/O 和标准 I/O 的区别

文件I/O又称为低级磁盘I/O，遵循POSIX 相关标准，任何兼容POSIX 标准的操作系统上都支持文件I/O。标准I/O被称为高级磁盘I/O，遵循ANSI C相关标准。只要开发环境中有标准C库，标准I/O就可以使用。（Linux中使用的是GLIB C，它是标准C库的超集，不仅包含ANSI C中定义的函数，还包括POSIX标准中定义的函数，因此，Linux 下既可以使用标准I/O，也可以使用文件I/O）。

通过文件I/O 读写文件时，每次操作都会执行相关**系统调用**。这样处理的好处是直接读写实际文件，坏处是频繁的系统调用会增加系统开销。标准I/O可以看成是在文件I/O 的基础上封装了缓冲机制。先读写缓冲区，必要时再访问实际文件，从而减少了系统调用的次数。

文件I/O中用文件描述符表示一个打开的文件，可以访问不同类型的文件，如普通文件、设备文件和管道文件等。而标准I/O中用FILE（流）表示一个打开的文件，通常只用来访问普通文件。

文件I/O不带缓冲，每次读写操作都要进行一次系统调用，直接对文件进行读写，所以能实时反映文件的内容。而标准I/O在打开文件时会自动缓冲，每次读写时仅对缓冲区进行操作，不对文件操作，退出时才更新文件。

三、　文件 I/O 操作

文件I/O操作的系统调用，主要用到5个函数：open()、read()、write()、lseek()和close()。这些函数的特点是不带缓存，直接对文件（包括设备）进行读写操作。这些函数虽然不是ANSI C的组成部分，但是是POSIX的组成部分。

1.　基本文件操作

1) 函数说明。

- open()函数用于打开或创建文件，在打开或创建文件时可以指定文件的属性及用户的权限等各种参数。
- close()函数用于关闭一个被打开的文件。当一个进程终止时，所有被它打开的文件都由内核自动关闭，很多程序都使用这一功能而不显示地关闭一个文件。
- read()函数用于将从指定的文件描述符中读出的数据放到缓存区中，并返回实际读入的字节数。若返回0，则表示没有数据可读，即已达到文件尾。读操作从文件的当前指针位置开始。当从终端设备文件中读出数据时，通常一次最多读一行。
- write()函数用于向打开的文件写数据，写操作从文件的当前指针位置开始。对磁盘文件进行写操作，若磁盘已满或超出该文件的长度，则write()函数返回失败。
- lseek()函数用于在指定的文件描述符中将文件指针定位到相应的位置。它只能用在可定位（可随机访问）文件操作中。管道、套接字和大部分字符设备文件是不可定位的，所以在这些文件的操作中无法使用lseek()调用。

2) 函数格式。

① open()函数的语法格式如表1所示：

表1 open()函数的语法格式

所需头文件	#include <sys/types.h> /*提供类型pid_t的定义*/ #include <sys/stat.h> /*获取文件属性*/ #include <fcntl.h> /*定义open等函数原型*/	
函数原型	int open(const char *pathname, int flags, int perms)	
函数传入值	pathname	被打开的文件名（可包括路径名）
	flag：文件打开的方式	O_RDONLY：以只读方式打开文件
		O_WRONLY：以只写方式打开文件
		O_RDWR：以读写方式打开文件
		O_CREAT：如果该文件不存在，就创建一个新的文件，并用第三个参数为其设置权限
		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。此时open是原子操作，防止多个进程同时创建同一个文件
		O_TRUNC：若文件已经存在，那么会删除文件中的全部原有数据，并且设置文件大小为0。
		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，即将写入的数据添加到文件的末尾
	perms	创建的新文件的存取权限 可以用一组宏定义：S_I(R/W/X)(USR/GRP/OTH) 其中R/W/X 分别表示读/写/执行权限 USR/GRP/OTH 分别表示文件所有者/文件所属组/其他用户 例如，S_IRUSR | S_IWUSR 表示设置文件所有者的可读可写属性。八进制表示法中600也表示同样的权限
函数返回值	成功：返回文件描述符 失败：-1	

在open()函数中，flags参数可通过“|”组合构成，但前3个标志常量（O_RDONLY、O_WRONLY以及O_RDWR）不能相互组合。perms是文件的存取权限，既可以用宏定义表示法，也可以用八进制表示法。

② close()函数的语法格式表2所示。

表2 close()函数的语法格式	
所需头文件	#include <unistd.h>
函数原型	int close(int fd)
函数输入值	fd：文件描述符
函数返回值	0：成功 -1：出错

③ read()函数的语法格式如表3所示。

表3 read()函数的语法格式	
所需头文件	#include <unistd.h>
函数原型	ssize_t read(int fd, void *buf, size_t count)
函数传入值	fd：文件描述符
	buf：指定存储器读出数据的缓冲区
	count：指定读出的字节数
函数返回值	成功：读到的字节数 0：已到达文件尾 -1：出错

在读普通文件时，若读到要求的字节数之前已到达文件的尾部，则返回的字节数会小于希望读出的字节数。

④ write()函数的语法格式如表4所示。

表4 write()函数的语法格式	
所需头文件	#include <unistd.h>
函数原型	ssize_t write(int fd, void *buf, size_t count)
函数传入值	fd：文件描述符
	buf：指定存储器写入数据的缓冲区
	count：指定读出的字节数
函数返回值	成功：已写的字节数 -1：出错

在写普通文件时，写操作从文件的当前指针位置开始。

⑤ lseek()函数的语法格式如表5所示。

表5 lseek()函数的语法格式		
所需头文件	#include <unistd.h> #include <sys/types.h>	
函数原型	off_t lseek(int fd, off_t offset, int whence)	
函数传入值	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，可正可负（向前移，向后移）	
	whence：当前位置的基点	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

函数返回值	成功：文件的当前位移 -1：出错

3) 函数使用实例。

下面列出文件基本操作的实例，基本功能是从一个文件（源文件）中读取最后5B 数据并拷贝到另一个文件（目标文件）。在实例中源文件是以只读方式打开，目标文件是以只写方式打开（可以是读写方式）。若目标文件不存在，可以创建并设置权限的初始值为0644，即文件所有者可读可写，文件所属组和其他用户只能读。

```
/* copy_file.c */
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#define BUFFER_SIZE 1 /* 宏定义，每次读写缓存大小，单位为字节，影响运行效率*/
#define SRC_FILE_NAME "src_file" /* 源文件名*/
#define DEST_FILE_NAME "dest_file" /* 目标文件名文件名*/
#define OFFSET 5 /* 复制的数据大小，单位为字节，即5个字节*/
int main()
{
int src_file, dest_file;
unsigned char buff[BUFFER_SIZE];
int real_read_len;
/* 以只读方式打开源文件，若文件不存在则出错*/
src_file = open(SRC_FILE_NAME, O_RDONLY);
/* 以只写方式打开目标文件，若此文件不存在则创建该文件, 访问权限值为0644（0644也可以写为S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH） */
dest_file = open(DEST_FILE_NAME,O_WRONLY|O_CREAT, 0644);
if (src_file< 0 || dest_file< 0)
{
printf("Open file error\n");
exit(1);
}
/* 将源文件的读写指针移到最后5B的起始位置*/
lseek(src_file, -OFFSET, SEEK_END);
/* 读取源文件的最后5B数据并写到目标文件中（指针向前移动OFFSET），每次读写1B */
while ((real_read_len = read(src_file, buff, sizeof(buff))) > 0)
{
write(dest_file, buff, real_read_len);
}
close(dest_file);
close(src_file);
return 0;
}
```

```
# gcc copy_file.c  -o  copy_file
# vim  src_file                    //创建源文件，内容为“123456789”
# ./copy_file                      //执行程序
# cat  dest_file                   //查看目标文件，内容应为“6789”，即源文件最后5个字节的内容，因为每个数字占用1个字节位置，再加上结束符，共5个字节。
```

① 用open()函数打开文件时，如果没有带O_CREAT选项，则如果该文件不存在时，会提示错误，open()函数的返回值为-1，可以通过errno查看具体的错误信息。

【注意】

errno用来记录系统调用时的错误信息，使用时需要添加errno.h头文件。errno的返回值为数字，可以通过《errno代码（中文）.txt》查看数字所对应的具体错误。

a) 添加头文件#include <errno.h>

b) 在src_file = open(SRC_FILE_NAME, O_RDONLY)后面添加如下代码：

printf("errno = %d\n",errno);

c) 删掉src_file，再次测试程序，可得到errno的信息，如下图所示。

② 修改OFFSET的值，可分别获取源文件最后不同OFFSET大小的内容。

a) 将OFFSET修改为2，其他不变；

b) # gcc copy_file.c -o copy_file //重新编译

c) # rm dest_file //删除以前的目标文件，避免影响测试结果，可在打开dest_file时添加O_TRUNC，就不需要每次测试时删除dest_file

d) # ./copy_file //执行程序，重新生成目标文件

e) # cat dest_file //查看目标文件，内容应为“9”

③ 若要把源文件第N个字符到结束的所有内容送到目标文件中，需要作如下修改（注意，此时OFFSET值为2）：

a) 将lseek(src_file, -OFFSET, SEEK_END)改为lseek(src_file, **OFFSET, SEEK_SET**)

b) # gcc copy_file.c -o copy_file //重新编译

c) # rm dest_file //删除以前的目标文件

d) # ./copy_file //执行程序，重新生成目标文件

e) # cat dest_file //查看目标文件，内容应为“3456789”，因为此时OFFSET值为2