

**课 程 实 验 报 告**

**课程名称： 操 作 系 统 原 理**

**专业班级：计算机科学与技术CS1408班**

**学 号： U201414775**

**姓 名： 余文梦**

**指导教师： 谢夏**

**报告日期： 2016年12月30号**

**计算机科学与技术学院**

**目录**

[实验一：线程的同步 1](#_Toc730335966)

[一、实验目的 1](#_Toc1585744793)

[二、实验内容 1](#_Toc737832378)

[三、 实验心得 3](#_Toc6477106)

[实验二：进程控制 4](#_Toc2134722426)

[一、实验目的 4](#_Toc8890548)

[二、实验内容 4](#_Toc1370798704)

[三、 实验心得 7](#_Toc1749423761)

[实验三：共享内存与进程同步 8](#_Toc240285218)

[一、实验目的 8](#_Toc1169718261)

[二、实验内容 8](#_Toc300222964)

[三、实验心得 12](#_Toc1935620652)

[实验四：Linux文件目录 13](#_Toc1460096652)

[一、实验目的 13](#_Toc746028640)

[二、 实验内容 13](#_Toc356515594)

[三、 实验心得 19](#_Toc1894101483)

# 实验一：线程的同步

## 一、实验目的

1、掌握Linux系统用户界面中键盘命令的使用。

2、学会一种Linux下的编程环境。

3、掌握Linux线程的概念。

4、通过信号灯操作实现线程间的同步与互斥。

## 二、实验内容

**1、设计要求**

通过Linux多线程与信号灯机制，设计并实现计算机线程与I/O线程共享缓冲区的同步与通信。

程序要求:两个线程,共享公共变量a

线程1负责计算(1到100的累加，每次加一个数)

线程2负责打印（输出累加的中间结果)

**2、设计思路**

由于线程共享所有的内存空间,因此可以直接设置一个全局变量share作为操作数。调用pthread\_create函数可以创建两个线程。第一个线程调用计算函数,对share进行累加操作。第二个线程负责将share值打印输出到标准输出。

此次实验,需要利用PV信号灯实现线程之间的同步与互斥。信号量最重要的两个属性就是本身代表的意义和初始值。此实验中,设置两个信号量,一个代表是否可计算,初始值为1;另一个代表是否可打印,初始值为0。于是,在对应的计算和打印函数中正确调用PV原子操作就可以实现计算和打印两个线程的互斥。

另外,在主函数中,需要调用pthread\_join函数等待线程的结束,当然,由于共享全部内部内存,也可以设置全局变量,在线程结束前对全局变量进行相应赋值操作,然后主函数进行判断,这样也可以实现等待的功能。线程结束后,调用semctl函数删除信号量。至此,整个实验要求就基本实现了。

**3、运行环境**

操作系统: Ubuntu 16.04.1 LTS

编辑器: Vim 7.4

编译器: gcc 5.4.0

调试器: GNU gdb 7.11.1

硬件: 64位机,数据采取小端方式储存。

**4、源程序**

pv.h

#include<stdio.h>

#include<pthread.h>

#include<sys/types.h>

#include<sys/sem.h>

/\*P,V操作函数\*/

void P(int semid,int index);

void V(int semid,int index);

int semid; /\*信号灯句柄\*/

pthread\_t p1,p2; /\*线程句柄\*/

void\* subp1(); /\*计算\*/

void\* subp2(); /\*打印\*/

int share = 0; /\*共享变量\*/

pv.c

int main(){

int mykey = 1234;/\*信号量标识码\*/

/\*创建两个信号量

\* 0--代表计算

\* 1--代表打印\*/

semid=semget((key\_t)mykey,2,IPC\_CREAT|0666);

/\*对信号量赋初始值\*/

semctl(semid,0,SETVAL,1);

semctl(semid,1,SETVAL,0);

/\*创建两个线程,线程1执行计算函数,线程2执行打印输出函数\*/

if((pthread\_create(&p1,NULL,(void\*)subp1,NULL))!=0)

return 1;

if((pthread\_create(&p2,NULL,(void\*)subp2,NULL))!=0)

return 1;

/\*等待两个线程运行结束\*/

pthread\_join(p1,NULL);

pthread\_join(p2,NULL);

/\*删除两个信号灯\*/

semctl(semid,0,IPC\_RMID);

semctl(semid,1,IPC\_RMID);

return 0;

}

void\* subp1(){//执行计算功能

int i = 0;

for(i = 1; i <= 100 ;++i){

P(semid,0);/\*检测是否可以计算\*/

share += i;

V(semid,1);

}

pthread\_exit(0);/\*0代表正常退出\*/

}

void\* subp2(){//执行打印功能

int i = 0,j = 0;

for(i = 0;i < 100;++i){

P(semid,1);/\*检测是否可以进行打印\*/

printf("%4d ",share);

j++;

V(semid,0);

}

pthread\_exit(0);/\*0代表正常退出\*/

}

**5、实验结果**

程序的打印输出结果如图1-1所示.经分析,实验结果与预期一致。

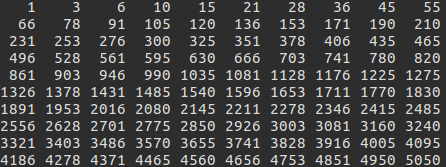


图1-1 打印结果

## 实验心得

虽然在实验开始前自己对linux已经有不少的了解了,但在线程和进程编程方面还一知半解。在上理论课时,一直不明白为什么信号量的初始值不能为负数,以及信号量实质上代表什么。通过本次实验,自己对这些也有了更深地理解。

刚开始的时候,对于信号量如何赋初始值,即semctl函数的调用,犯了不少的错误。缘于没有理解semtcl函数参数的意义。特别是第三个参数,开始一直处于忽略状态,后来,才知道SETALL与SETVAL的不同之处。前者可以同时对多个信号量进行赋初始值,后者只能对单个信号量赋初始值。因为这个导致自己的程序运行无法终止.这也提醒自己,用一个函数之前,一定要先明白其作用,以及其接口的实际意义,才能避免未知错误。

不过,总体来说,此次实验还是比较简单的,包括线程之间共享全部内存的概念。也是经过此次实验,敲开了自己线程和进程编程的大门,收获不少!

# 实验二：进程控制

## 一、实验目的

1、加深对进程的理解,进一步认识并发执行的实质；

2、分析进程争用资源现象,学习解决进程互斥的方法；

3、掌握Linux进程基本控制；

4、掌握Linux系统中的软中断和管道通信。

## 二、实验内容

**1、设计要求**

编写程序，演示多进程并发执行和进程软中断、管道通信。

父进程使用系统调用pipe( )建立一个管道,然后使用系统调用fork()创建两个子进程，子进程1和子进程2；

子进程1每隔1秒通过管道向子进程2发送数据:

I send you x times. (x初值为1，每次发送后做加一操作）

子进程2从管道读出信息，并显示在屏幕上。

父进程用系统调用signal()捕捉来自键盘的中断信号（即按Ctrl+C键）；当捕捉到中断信号后，父进程用系统调用Kill()向两个子进程发出信号，子进程捕捉到信号后分别输出下列信息后终止：

Child Process l is Killed by Parent!

Child Process 2 is Killed by Parent!

父进程等待两个子进程终止后，释放管道并输出如下的信息后终止

Parent Process is Killed!

**2、设计思路**

首先应该知道进程的概念,与上一个实验的线程不同的是,子进程虽然是父进程的副本,但是进程之间各自占有不同的内存。这一点特别重要!在linux中创建子进程也特别简单,只需要调用fork()函数即可。fork函数会返回两个值,一个是零,代表子进程;另一个非零,代表父进程。因此可以用条件判断语句将父子进程区分开。

此实验要求利用管道,实现进程之间的交互。管道可以理解成一根排水管,由于各进程的程序完全一致,所以,有几个进程,就有几个读和写的描述符指向管道。因此,实验中可以关闭不需要进行读写操作的描述符。比如父进程的读写描述符。

此实验的难点在于进程的软中断通信。通过软中断,接收终端或者其他进程中发出的信号。主要涉及signal函数的使用,此函数第一个参数代表信号的类型,本实验主要用到SIGINT,SIGUSR1和SIGUSR2。第二个参数是一个函数指针,用于指向接收到信号后需要调用的处理函数。特别地,第二参数有几个特殊值,本次实验只用到了SIG\_IGN,其作用是设置忽略信号。而kill函数可以向进程发出信号,并且此信号不能被忽略。

基于上面这些,合理组织程序,便能很快地完成实验要求。

**3、运行环境**

操作系统: Ubuntu 16.04.1 LTS

编辑器: Vim 7.4

编译器: gcc 5.4.0

调试器: GNU gdb 7.11.1

硬件: 64位机,数据采取小端方式储存。

**4、源程序**

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<signal.h>

#include<sys/wait.h>

#include<sys/types.h>

void myfunc1();

void myfunc2();

void handleParent();//父进程信号处理函数

void handle1();//子进程1信号处理函数

void handle2();//子进程2信号处理函数

int pipefd[2];//管道读写

pid\_t p1,p2;//子进程标识符

int status = -1;//设置进程退出码

int main()

{

pipe(pipefd);//pipefd[0]用于读,pipefd[1]用于写

signal(SIGINT,handleParent);

p1 = fork();//创建子进程1

if(p1 == 0){

close(pipefd[0]);//关闭子进程1的读的通道

myfunc1();

}

else{

p2 = fork();//创建子进程2

if(p2 == 0){

close(pipefd[1]);//关闭子进程2的写通道

myfunc2();

}

else{

wait(&status);//等待子进程1退出

wait(&status);//等待子进程2退出

}

}

/\*关闭父进程的读写描述符\*/

close(pipefd[0]);

close(pipefd[1]);

printf("\nParent Process is Killed.");

return 0;

}

void myfunc1()

{

int i = 1;//计数器

signal(SIGINT,SIG\_IGN);//忽略信号SIGINT

signal(SIGUSR1,handle1);//设置接收信号SIGUSR1

while(1){

write(pipefd[1],&i,sizeof(int));//往管道中写内容

i++;//计数器加1

sleep(1);//休眠1s

}

}

void myfunc2()

{

int m;//用于接收管道中的值

signal(SIGINT,SIG\_IGN);//忽略信号SIGINT

signal(SIGUSR2,handle2);//设置接收信号SIGUSR1

while(1){

read(pipefd[0],&m,sizeof(int));//往管道中读内容

printf("I send you %d times\n",m);

}

}

void handleParent()

{

kill(p1,SIGUSR1);//发送信号SIGUSR1给进程p1

kill(p2,SIGUSR2);//发送信号SIGUSR2给进程p2

}

void handle1()

{

close(pipefd[1]);

printf("\nChild Process 1 is Killed by Parent!");

exit(0);//0代表正常退出

}

void handle2()

{

close(pipefd[0]);

printf("\nChild Process 2 is Killed by Parent!");

exit(0);

}

1. **实验结果**

程序运行几秒后的输出结果如图2-1所示.当终端发出终止父进程的命令时,输出结果如图2-2所示.经测试,输出结果与预期一致。

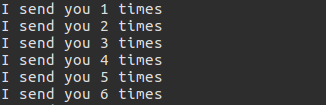


图2-1 程序中间结果

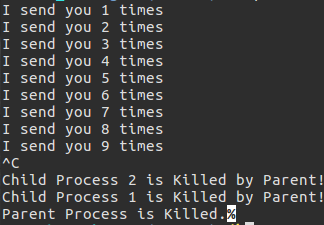


图2-2 程序最终输出结果

## 实验心得

在做实验的过程中，碰到最大的难点就是软中断信号的处理。开始，也没有理解发送与接送信号的方式和特点。经过不断地试错，了解到信号只是一个标记，当某进程设置了该信号标记后，一旦有其他进程向它发送该信号，其将立刻调用编程者指定的信号处理函数。所以，没有明白这一点，导致自己碰壁不少。

另外，还有一个错误，就是最后杀死进程的语句重复输出，这一点也让自己百思不得其解，没想到是自己忽略了进程独立运行的特点。没有将对应进程的语句放到其应该在的位置。

结合上一次的线程实验，可以帮助自己更好地理解进程和线程之间的差异，也能加深对课堂理论知识的记忆和理解。所以，很多知识，如果不动手实践的话，是很难真正地理解。当然，自己现在懂得也许还是错的，但学习就是不断更新，不断纠错的过程吧！

# 实验三：共享内存与进程同步

## 一、实验目的

1、掌握Linux下共享内存的概念与使用方法；

2、掌握环形缓冲的结构与使用方法；

3、掌握Linux下进程同步与通信的主要机制。

## 二、实验内容

**1、设计要求**

利用多个共享内存（有限空间）构成的环形缓冲，将源文件复制到目标文件，实现两个进程的誊抄。

1. **设计思路**

首先，需要理解共享内存的概念。因为每个进程是有自己独立的内存块的，所以需要一个共享机制，可以更方便地实现进程之间的交互。与上一个实验中管道不同，共享内存没有读写描述符，其可以任意读写。不过，各进程之间若想对同一块内存进行读写，首先需要用shmget创建一块共享内存，与普通内存不同的是，该内存多了一个标识符，这样，不同的进程便可以通过该标识符访问同一块内存。从而达到共享的目的。

本实验可以在上一个进程控制的基础上，进一步将fork()与exec函数族结合使用，达到真正的进程同步。即先通过fork()函数创建一个父进程的副本，然后通过exec函数族加载一个新的程序。

这时候，可能容易碰到一个问题，就是子进程程序怎么获取父进程所创建的共享内存，也即怎么获取那个标识符。这里就需要深度理解在创建进程和共享内存的时候，那个看起来有点像随机值的参数的意义了。比如，shmgt(123456,10,IPC\_CREAT|0666)中的123456。这个值其实可以理解成标识符的来源，可以想象成系统将这个值映射成一个标识符，返回给进程。这样，当不同进程调用shmgt时，传入同一个随机值，系统返回的标识符也是一样的，那么不同地进程就可以通过此机制控制同一块内存，达到真正的共享的目的。

另外，本实验还需要实现环形缓冲区。为了简便，可以使用数组构建一个环形缓冲区，只需要在数组下标计算时注意取模即可。那么，怎样通过PV信号灯达到缓冲区互斥和同步也成了本实验另一个难点。其实，实现起来也很简单，只需要明白信号灯的初始值的意义，给信号量赋上正确的初始值。

这样，便能完成本实验设计的要求了。

**3、运行环境**

操作系统: Ubuntu 16.04.1 LTS

编辑器: Vim 7.4

编译器: gcc 5.4.0

调试器: GNU gdb 7.11.1

硬件: 64位机,数据采取小端方式储存。

**4、源程序**

父进程的源程序：

#include<stdio.h>

#include<sys/sem.h>

#include<sys/shm.h>

#include<sys/wait.h>

#include<unistd.h>

#include<stdlib.h>

#define M 10

#define \_SPID 123456

#define \_PVID 1234567

int main(int argc,char\*\* argv)

{

if(argc < 3) return 1;

pid\_t readbuf,writebuf;

int shmid = 0;//共享内存的标识符

int semid = 0;//信号灯标识符

char\* S = NULL;//共享内存的首地址

/\*创建共享内存组--数组形式\*/

shmid = shmget(\_SPID,M,IPC\_CREAT|0666);

S = (char\*)shmat(shmid,NULL,SHM\_R|SHM\_W);

/\*创建两个信号灯

\* 信号灯0--代表缓冲区为空

\* 信号灯1--代表缓冲区为满\*/

semid = semget(1234567,2,0666|IPC\_CREAT);

/\*信号灯赋初值\*/

semctl(semid,0,SETVAL,M);//0号信号灯赋初值M

semctl(semid,1,SETVAL,0);//1号信号灯赋初值0

writebuf = fork(); /\*创建写缓冲的进程\*/

if(writebuf == 0){

char \*const arg[] = {"write",argv[1],NULL};

execv("./write",arg);//调用read

exit(0);

}

else{

readbuf = fork();/\*创建读缓冲的进程\*/

if(readbuf == 0){

char \*const arg[] = {"read",argv[2],NULL};

execv("./read",arg);//调用write

exit(0);

}

else{

waitpid(writebuf,NULL,0);

waitpid(readbuf,NULL,0);

/\*删除信号灯\*/

semctl(semid,0,IPC\_RMID);

semctl(semid,1,IPC\_RMID);

shmctl(shmid,IPC\_RMID,0);/\*删除共享内存组\*/

}

}

}

读文件进程的源程序：

#include<stdio.h>

#include<sys/shm.h>

#include<unistd.h>

#include"pv.c"

#define \_SPID 123456

#define \_PVID 1234567

#define M 10//缓冲区的个数

int main(int argc,char\*\* argv){

int shmid = shmget(\_SPID,M,IPC\_CREAT|0666);

char\* S = (char\*)shmat(shmid,NULL,SHM\_R|SHM\_W);

int semid = semget(\_PVID,2,0666|IPC\_CREAT);

int out = 0;//读缓冲区的下标

FILE \*fd = fopen(argv[1],"w");

while(S[out] != EOF){

P(semid,1);

fprintf(fd,"%c",S[out]);

fprintf(stdout,"%c",S[out]);

out = (out + 1) % M; //取模构成环形缓冲区

V(semid,0);

}

fclose(fd);

semctl(semid,0,IPC\_RMID);

semctl(semid,1,IPC\_RMID);

shmctl(shmid,IPC\_RMID,0);/\*删除共享内存组\*/

return 0;

}

写文件进程的源程序：

#include<stdio.h>

#include<sys/shm.h>

#include<unistd.h>

#include"pv.c"

#define \_SPID 123456

#define \_PVID 1234567

#define M 10//缓冲区个数

int main(int argc,char\*\* argv){

int shmid = shmget(\_SPID,M,IPC\_CREAT|0666);

char\* S = (char\*)shmat(shmid,NULL,SHM\_R|SHM\_W);

int semid = semget(\_PVID,2,0666|IPC\_CREAT);

int in = 0;//写缓冲的下标

FILE \*fd = fopen(argv[1],"r");

while(!feof(fd)){

P(semid,0);

S[in] = fgetc(fd);

in = (in + 1) % M;//取模构成环形缓冲区

V(semid,1);

}

fclose(fd);

semctl(semid,0,IPC\_RMID);

semctl(semid,1,IPC\_RMID);

shmctl(shmid,IPC\_RMID,0);/\*删除共享内存组\*/

return 0;

}

**5、实验结果**

为了测试文件复制功能，准备了一个测试文件test.c，其内容如图3-1所示。

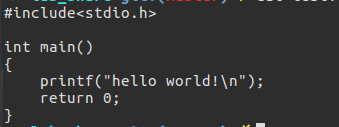


图3-1 test.c

接着，用本实验生成的程序mycp复制test.c，得到文件a.c。文件a.c里面的内容如图3-2所示。

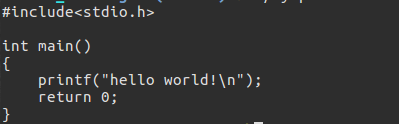


图3-2 a.c

## 三、实验心得

在前两个实验中，对于进程或线程的标识符没有充分的理解导致本次实验受挫不少。当前，如果此次实验自己没有尝试去使用exec()函数加载一段新的程序的话，自己也不会去明白上面那些。所以，在学习过程中，应该要多尝试新事物，只有尝新，才能加深对旧知识的理解。

虽然，已经能够达到实验的要求，但其实自己还有些不甚理解的地方。比如，为什么父子进程都需要对共享内存进行释放，虽然父子进程中都调用了shmgt函数，但他们创建的是同一块内存。现在自己的解释是，shmgt创建共享内存的方法只是赋予本进程对该内存的使用权，并且该内存不能通过其他方式使用。这样理解的话，进程之间的交互也就不局限于父子进程之间了。这样理解，似乎更符合实际的原理。

当然，上面只是目前的理解，很可能是错的，也意味着学习确实是一个永无止境的过程。

# 实验四：Linux文件目录

## 一、实验目的

1、了解Linux文件系统与目录操作；

2、了解Linux文件系统目录结构；

3、掌握文件和目录的程序设计方法。

## 实验内容

**1、设计要求**

编程实现目录查询功能，功能类似ls -lR，查询指定目录下的文件及子目录信息，显示文件的类型、大小、时间等信息递归显示子目录中的所有文件信息。

**2、设计思路**

本实验应该是比较简单的，它只需要把系统提供的目录项结构体stat里的数据成员都理清楚。下面，是自己在学习过程中，对该结构体内的部分数据成员的理解。Stat结构体如下：

struct stat {

unsigned short st\_mode; // 文件的权限信息和类型信息:   
 unsigned short st\_nlink; //硬连接的数目  
 unsigned short st\_uid; // 文件所有者的ID   
 unsigned short st\_gid; //文件所有者的组ID   
 unsigned long st\_size; //文件大小  
 unsigned long st\_mtime; // 最后修改内容的时间   
 ...//剩下的数据成员本实验用不到

};

数据成员st\_mode在应该是特别重要的一个了，它里面包含了权限及文件类型等信息。通过man查找相关手册了解到，若以八进制表示，高位表示类型信息，而低位表示权限信息，如下所示。

S\_IFMT 0170000 bit mask for the file type bit field

S\_IFSOCK 0140000 socket

S\_IFLNK 0120000 symbolic link

S\_IFREG 0100000 regular file

S\_IFBLK 0060000 block device

S\_IFDIR 0040000 directory

S\_IFCHR 0020000 character device

S\_IFIFO 0010000 FIFO

以上是表示类型信息的高位部分。

S\_ISUID 04000 set-user-ID bit

S\_ISGID 02000 set-group-ID bit (see below)

S\_ISVTX 01000 sticky bit (see below)

S\_IRWXU 00700 owner has read, write, and execute permission

S\_IRUSR 00400 owner has read permission

S\_IWUSR 00200 owner has write permission

S\_IXUSR 00100 owner has execute permission

S\_IRWXG 00070 group has read, write, and execute permission

S\_IRGRP 00040 group has read permission

S\_IWGRP 00020 group has write permission

S\_IXGRP 00010 group has execute permission

S\_IRWXO 00007 others(not in group) have read, write, and

execute permission

S\_IROTH 00004 others have read permission

S\_IWOTH 00002 others have write permission

S\_IXOTH 00001 others have execute permission

以上是表示权限信息的低位部分。

举个例子，若st\_mode等于0100777,高位为010,表示regular file,即普通文件，而低位为0777,其中7的二进制表示为111,代表rwx，那么，0777即代表owner，group和others对该文件均具有读写和执行权限。整个st\_mode可以输出“-rwxrwxrwx”。怎么输出这种信息就是编程问题了。

数据成员st\_nlink代表该文件被硬链接的次数。

数据成员st\_uid和st\_gid文件所有者的ID和组ID。可以借助getpwuid函数将ID值转换为用户名和组名输出。

数据成员st\_size代表文件的大小，以字节B为单位输出。

数据成员st\_mtime代表文件最后一次被修改的时间，可以借助ctime函数将其转换成常见的时间格式输出，需要注意的是，ctime返回的字符串末尾有一个换行符，所以，为了避免后面输出文件名的乱序问题，可以借助strncpy函数将末尾的换行符去掉。综合上面的数据成员，可以打印如下示例：

-rwxrwxr-x 1 iyuge2 iyuge2 13672 Wed Dec 28 16:32:23 2016 myls

依次代表文件类型及权限、硬链接数、用户名、组名、文件大小、时间日期和文件名。

除了stat结构体外，本实验还涉及其它一些系统函数调用，比如opendir()用于打开文件，chdir()用于更改目录，lstat()函数用于获取文件信息，即得到上面的stat结构，结构体dirent内含有文件名等信息。

**3、运行环境**

操作系统: Ubuntu 16.04.1 LTS

编辑器: Vim 7.4

编译器: gcc 5.4.0

调试器: GNU gdb 7.11.1

硬件: 64位机,数据采取小端方式储存。

**4、源程序**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/stat.h>

#include<dirent.h>

#include<time.h>

#include<pwd.h>

/\*判断非目录文件类型\*/

char filetype(struct stat\* s)

{

switch(s->st\_mode & S\_IFMT){

case S\_IFBLK: return 'b';//块设备文件

case S\_IFCHR: return 'c';//一次性读写文件

case S\_IFDIR: return 'd';//目录

case S\_IFIFO: return 'p';//输入输出流文件

case S\_IFLNK: return 'l';//链接文件

case S\_IFREG: return '-';//普通文件

case S\_IFSOCK: return 's';

default:

return '@';//出错标志，自己设的

}

}

/\*返回权限类型\*/

char\* filerwx(unsigned short n)

{

switch(n){

case 0: return "---";

case 1: return "--x";

case 2: return "-w-";

case 3: return "-wx";

case 4: return "r--";

case 5: return "r-x";

case 6: return "rw-";

case 7: return "rwx";

default: fprintf(stderr,"RWXerror");

}

}

/\*

\* op->0 ls

\* op->1 ls -l

\* op->2 ls -R

\* op->3 ls -lR

\* \*/

void printdir(char \*dir,int depth,int op)

{

char \*t;

unsigned long totalSize = 0;

DIR \*dp;

struct passwd \*gname,\*uname;

struct dirent \*entry;

struct stat statbuf;

if((dp = opendir(dir)) == NULL){

fprintf(stderr,"open %s dir error!\n",dir);

exit(0);

}

chdir(dir);//将dir设置为当前目录

while((entry = readdir(dp)) != NULL){

lstat(entry->d\_name,&statbuf);

totalSize += statbuf.st\_size;

if((op >= 2) && S\_ISDIR(statbuf.st\_mode)){//此文件为目录且有-R

if(strcmp(entry->d\_name,".") == 0 || strcmp(entry->d\_name,"..") == 0){

continue;

}

printf("<dir>%s.:\n",entry->d\_name);

printdir(entry->d\_name,depth+4,op);//递归打印目录

}

else{//非目录或无-R

if(!(op % 2)){//ls

printf("%s\n",entry->d\_name);

}

else{//ls -l

t = ctime(&statbuf.st\_ctime);

strncpy(t,t,strlen(t)-1);

t[strlen(t)-1] = '\0';

/\*以上三行为了去掉ctime默认加的\n\*/

gname = getpwuid(statbuf.st\_gid);

uname = getpwuid(statbuf.st\_uid);

printf("%c%s%s%s %4ld %s %s %8ld %s %s\n",filetype(&statbuf),filerwx((statbuf.st\_mode & 0700) >> 6),filerwx((statbuf.st\_mode & 070) >> 3),filerwx(statbuf.st\_mode & 07),(long)statbuf.st\_nlink,uname->pw\_name,gname->pw\_name,statbuf.st\_size,t,entry->d\_name);

/\*st\_mode为8进制数\*/

}

}

}

printf("Total size: %ld\n\n",totalSize);

chdir("..");

closedir(dp);

}

int main(int argc,char\*\*argv)

{

char \*topdir = ".";//初始化为当前目录

int i = 1;

while(i < argc){//判断目录

if(argv[i][0] != '-'){

topdir = argv[i];

break;

}

++i;

}

if(argc >= 2){//判断参数

for(i = 1;i < argc;++i){

if(argv[i][0] == '-'){

if(!strcmp(argv[i],"-l"))

printdir(topdir,0,1);

else if(!strcmp(argv[i],"-R"))

printdir(topdir,0,2);

else if(!strcmp(argv[i],"-lR"))

printdir(topdir,0,3);

return 0;

}

}

}

printdir(topdir,0,0);

return 0;

}

**5、实验结果**

myls ..的输出上层目录中的文件名，结果如图4-1所示。

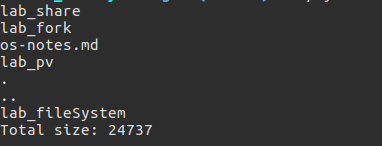


图4-1 myls ..的输出结果

myls -l ..输出上层目录中文件的详细信息，结果如图4-2所示。

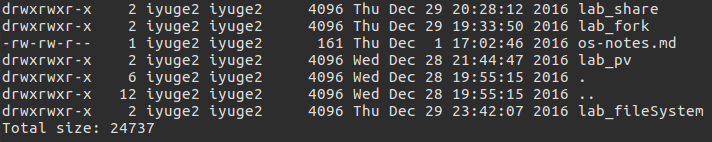


图4-2 myls -l ..的输出结果

Myls -lR ..递归输出文件目录和文件名信息，结果如图4-3所示。

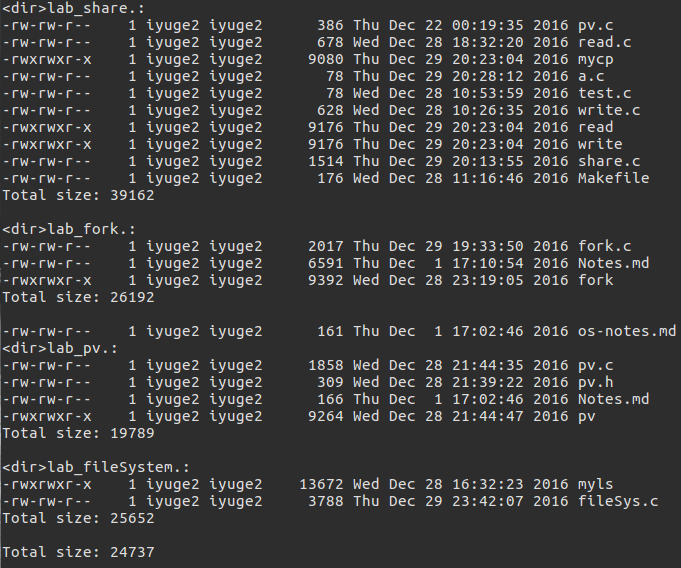


图4-3 myls -lR ..的输出结果

## 实验心得

可能因为自己本身一直用的就是linux操作系统，所以对文件目录等操作比较熟练。不过，以前自己也只是用而已，没有花精力去思考他们都是怎么实现的。所以，这次能够自己动手实现一个ls，还是蛮开心的。虽然只实现了其中-l和-R两个参数的功能，但触类旁通，其他的原理应该都是类似的。

本次实验中，主要的时间都耗费在怎样得到自己想要输出的信息。尤其是对st\_mode的解读上。不过，通过此次学习，自己也强化了一下对man手册的解读能力，应该说，man手册中的内容是最为正式的说明，理应认真对待。所以，这次实验，通过man的查找也为自己省了不少的时间。

总体上来说，此次实验比较简单，但想做到完美，还是挺难的。做完这次实验，看到自己输出的结果与系统ls功能输出的一致，也是蛮兴奋的。通过这个过程，也能帮助自己以后更好地去理解linux。