

华中科技大学

课程实验报告

课程名称：操作系统原理

院 系：计算机科学与技术

专业班级：CS1409

学 号：U201414808

姓 名：王林

指导教师：

2016 年 12 月 27 日

华中科技大学

目录

实验一 进程控制.....	4
1、 实验目的	4
2、 实验内容	4
3、 实验步骤和结果	4
4、 实验总结和体会	5
实验二 线程同步与通信.....	6
1、 实验目的	6
2、 实验内容	6
3、 实验步骤和结果	6
4、 实验总结和体会	7
实验三 共享内存与进程同步	8
1、 实验目的	8
2、 实验内容	8
3、 实验步骤和结果	8
4、 实验总结和体会	10
实验四 LINUX 文件目录	11
1、 实验目的	11
2、 实验内容	11
3、 实验步骤和结果	11
4、 实验总结和体会	12
附录.....	13

实验一 进程控制

1、 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。

2、 实验内容

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 1 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!

3、 实验步骤和结果

1.了解预备知识:

在正式开始进行实验代码的编写之前先了解进程控制相关的知识,学习管道的创建,中断的处理。在实验一中我们需要了解并熟练运用的函数如下:

- A. `fork()`: 进程创建
- B. `pipe()`: 管道创建
- B. `wait()`、`waitpid()`: 等待进程终止
- C. `signal()`: 预置信号接收后的处理方式
- D. `kill()`: 杀死指定进程

2.编写实验代码

(1)创建管道

首先定义一个大小为二的一维整型数组,然后用 `pipe()` 函数将其创建为管道。

```
int pipefd[2];
```

```
pipe(pipefd); /*创建无名管道*/
```

(2)创建子进程一和子进程二

使用 `fork` 函数, `child1 = fork();child2 = fork();`

(3)等待子进程一、二退出

使用 `wait` 函数;

(4)关闭管道

3.测试实验结果

对源程序编译、链接、生成可执行程序后，运行可执行程序，观察运行结果是否和预想一致。

实验环境：Ubuntu 16.04 gcc gdb vim terminal

```

strawberrylin@strawberrylin-X550JK: ~/Gitrepository/OperationSystem_lab/first_lab - [2016-12-28 01:02:48]
[0] <git:(master f89d2f0*) > ls
main.c test2
[0] <git:(master f89d2f0*) > ./test2
I send you 1 times.
I send you 2 times.
I send you 3 times.
I send you 4 times.
I send you 5 times.
I send you 6 times.
I send you 7 times.
I send you 8 times.
I send you 9 times.
^C
Child Process 1 is killed by Parent
Child Process 2 is killed by Parent
Parent process is killed
[0] <git:(master f89d2f0*) >
  
```

图 1-1 进程控制实验截图

如图，进程 1 每次想管道传递信息并进行+1 次数计数，子进程依次从管道接受信息，并打印到屏幕上。当屏幕按下“ctrl + c”后，父进程接受中断信息，并且传递中断信号给子进程，子进程接受中断信号，终止进程。屏幕依次打印出进程一、二终止信息以及父进程结束信息，屏幕结果与预期相同，测试通过。

4、 实验总结和体会

在本次实验中，学习了进程间的通信，本次实验室通过管道的相关操作实现，还有进程的控制，包括进程的创建、修改、终止进程等操作，还有进程间的软中断通信操作。通过上述内容的学习，增加了对 Linux 进程的了解。

在实现一个父进程创建多个子进程时，采用的逻辑是：先在父进程创建一个子进程，然后如果不在子进程，再创建一个子进程，依照这种方法，实现一个父进程创建多个子进程。创建进程需要通过进程 id 号来控制进程创建函数的调用，否则会在子进程同时创建子进程，造成进程过多的现象。

创建子进程时，子进程会从父进程继承，但是只是相当于从父进程拷贝一个副本，子进程的值修改并不能对父进程产生影响。

本次实验中，采用的是无管道，一个进程读管道信息，一个进程写管道信息，读写过程存在阻塞。读写之前需要对对应管道进行关闭操作。

在本次实验中，因为管道的操作存在 PV 操作，所以在接受到中断信号后，应该立即退出进程，不应该再进行管道读写，否则会出现挂起现象。

实验二 线程同步与通信

1、 实验目的

- 1、掌握 Linux 下线程的概念；
- 2、了解 Linux 线程同步与通信的主要机制；
- 3、通过信号灯操作实现线程间的同步与互斥

2、 实验内容

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)

3、 实验步骤和结果

1.了解预备知识:

1、Linux 下的信号灯及其 P、V 操作

在 Linux 中信号灯是一个数据集合，可以单独使用这一集合的每个元素。

有关的系统调用命令：

- 1) semget: 返回一个被内核指定的整型的信号灯索引。
- 2) semop: 执行对信号灯集的操作
- 3) semctl: 执行对信号灯集的控制操作

2、线程

1) 线程创建 pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

2) pthread_join(pthread_t th, void **thread_return);

作用：挂起当前线程直到由参数 th 指定的线程被终止为止

2.编写实验代码

(1)创建信号灯，信号灯赋初值

使用系统调用semget()创建一个新的信号量集，通过semctl函数进行信号灯的初始化。

```
semid = semget(0,2,IPC_CREAT|0666);
sem_args.setval = 1;
ret3 = semctl(semid,0,SETVAL,sem_args);
sem_args.setval =0;
ret3 = semctl(semid,1,SETVAL,sem_args);
```

(2)创建两个线程

调用函数 pthread_create，创建两个线程。

```
ret1 = pthread_create(&p1, NULL, subp1, NULL);
ret2 = pthread_create(&p2, NULL, subp2, NULL);
```

(3)等待两个线程运行结束

调用 pthread_join 函数，挂起当前线程，直到 P1 和 P2 运行结束。

```
pthread_join(p1, NULL);
```

```
pthread_join(p2, NULL);
```

(4)删除信号灯

调用semctl函数， semctl(semid, 0, IPC_RMID, sem_args);

3.测试实验结果

对源程序编译、链接、生成可执行程序后，运行可执行程序，观察运行结果是否和预想一致。

实验环境：Ubuntu 16.04 gcc gdb vim terminal

```
strawberrylin@strawberrylin-X550JK: ~/Gitrepository/OperationSystem_lab/second_lab 79x24
└─[strawberrylin@strawberrylin-X550JK] - [~/Gitrepository/OperationSystem_lab/s
econd_lab] - [2016-12-28 04:21:51]
└─[0] <git:(master f89d2f0x*) > ./lab2_out
1
3      6      10      15      21      28      36      45      55      66
78     91     105     120     136     153     171     190     210     231
253    276    300    325    351    378    406    435    465    496
528    561    595    630    666    703    741    780    820    861
903    946    990    1035   1081   1128   1176   1225   1275   1326
1378   1431   1485   1540   1596   1653   1711   1770   1830   1891
1953   2016   2080   2145   2211   2278   2346   2415   2485   2556
2628   2701   2775   2850   2926   3003   3081   3160   3240   3321
3403   3486   3570   3655   3741   3828   3916   4005   4095   4186
4278   4371   4465   4560   4656   4753   4851   4950   5050
```

图 2-1 线程通信与同步实验截图

如图，线程一完成了计算，实现 1~100 的累加，线程二将每次的中间结果打印到屏幕上，顺序以及结果没有出现差错，证明测试成功。

4、实验总结和体会

本次实验中，学习了信号量的创建以及 PV 操作、线程控制等。

创建信号量时，需要创建两个信号量，分别赋值为 1 和 0，这样才能实现正确的线程执行顺序，两个线程互斥，交替执行

本次实验重点是实现 P 操作和 V 操作，从而实现对线程的控制，实现线程互斥。PV 操作是对信号量的操作，P 操作实现-1 操作，V 操作实现+1 操作。两个信号的 PV 操作在两个线程成对出现，实现线程执行顺序的控制。信号量为 1 时执行，为 0 时挂起等待。

本次实验中碰到的问题在于开始只设置了一个信号量，则不能正确控制线程执行顺序，解决方法时增加一个信号量。

实验三 共享内存与进程同步

1、 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。

2、 实验内容

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。

3、 实验步骤和结果

1. 了解预备知识：

1、共享内存

使用共享内存是运行在同一计算机上的进程进行进程间通信的最快的方法。

`shmget` 创建共享内存；`shmat` 将内存映射到程序的缓冲区；`shmctl`对其状态信息进行读取和修改。

2、进程控制

`fork` 与 `execv` 系统调用

3、Linux 下的信号灯及其 P、V 操作

在 Linux 中信号灯是一个数据集合，可以单独使用这一集合的每个元素。

有关的系统调用命令：

- 1) `semget`: 返回一个被内核指定的整型的信号灯索引。
- 2) `semop`: 执行对信号灯集的操作
- 3) `semctl`: 执行对信号灯集的控制操作

4、环形缓冲

缓冲的目的是为了匹配 CPU 与设备的速度差异和负荷的不均衡，从而提高处理机与外设的并行程度。缓冲技术可以用硬件缓冲器来实现，通常容量较小软件缓冲是应用较广泛的一种缓冲技术，由缓冲区和对缓冲区的管理两部分组成

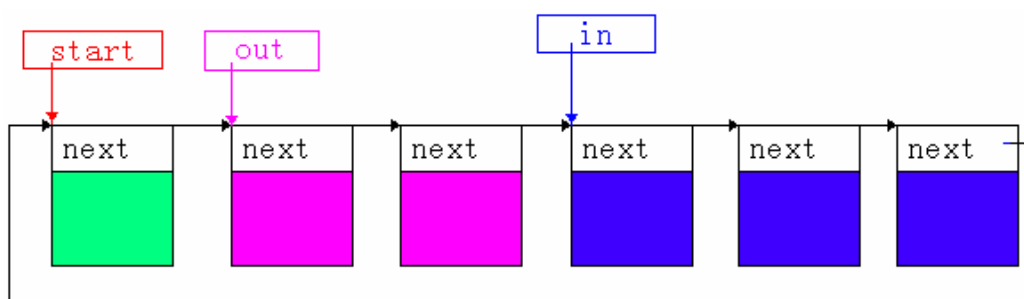


图 3-1 环形缓冲示意图

在主存中分配一组大小相等的存储区作为缓冲区，并将这些缓冲区链接起来。可循环使用这些缓冲区：输入指针 `in`、输出指针 `out`

2. 编写实验代码

(1)创建信号灯，信号灯赋初值

使用系统调用 `semget()` 创建一个新的信号量集，通过 `semctl` 函数进行信号灯的初始化。

```
semid = semget(0,3,IPC_CREAT|0666);
sem_args.setval = 0;
ret1 = semctl(semid,0,SETVAL,sem_args);
sem_args.setval = 5;
ret2 = semctl(semid,1,SETVAL,sem_args);
sem_args.setval = 1;
ret1 = semctl(semid,2,SETVAL,sem_args);
```

第一个信号量作为可读缓冲区的信号灯，第二个信号量作为可写缓冲区的信号灯，第三个作为互斥锁信号灯，控制进程执行。

(2)创建共享内存组

调用 `shmget` 函数和 `shmctl` 函数，创建 5 块共享内存，通过数组实现共享环形内存组。

```
shmids[0] = shmget(0, SIZE, IPC_CREAT|0600);
shmctl(shmids[0], IPC_STAT, &buf);
shmids[1] = shmget(0, SIZE, IPC_CREAT|0600);
shmctl(shmids[1], IPC_STAT, &buf);
shmids[2] = shmget(0, SIZE, IPC_CREAT|0600);
shmctl(shmids[2], IPC_STAT, &buf);
shmids[3] = shmget(0, SIZE, IPC_CREAT|0600);
shmctl(shmids[3], IPC_STAT, &buf);
shmids[4] = shmget(0, SIZE, IPC_CREAT|0600);
shmctl(shmids[4], IPC_STAT, &buf);
```

(3)创建两个进程

调用 `fork` 函数，一个父进程创建两个子进程，采用实验一种进程创建的逻辑，创建两个进程。

(4)等待两个进程执行结束

调用 `wait` 函数

(5)删除信号灯，删除共享内存组

调用 `shmctl` 函数和 `semctl` 函数

```
semctl(semid, 0, IPC_RMID, sem_args);
shmctl(shmids[0], IPC_RMID, NULL);
shmctl(shmids[1], IPC_RMID, NULL);
shmctl(shmids[2], IPC_RMID, NULL);
shmctl(shmids[3], IPC_RMID, NULL);
shmctl(shmids[4], IPC_RMID, NULL);
```

3.测试实验结果

对源程序编译、链接、生成可执行程序后，运行可执行程序，观察运行结果是否和预想一致。

实验环境：Ubuntu 16.04 gcc gdb vim terminal

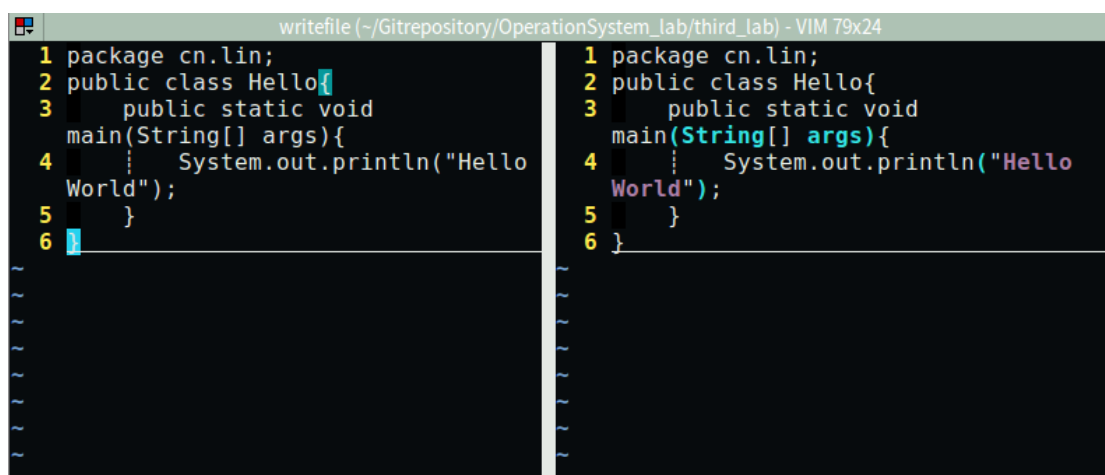


图 3-2 共享内存与进程同步实验截图

如图，上图右部是源文件，左部是拷贝的文件，对比可以发现，源文件与拷贝生成的文件一样，拷贝成功，测试成功。

4、 实验总结和体会

本次实验中，学习了共享内存组的使用，结合了前面实验中信号量的操作以及进程创建个控制实验。

本次试验中，进程控制采用生产者消费者模式，进程一负责从从文件度内容，并放入共享内存，进程二负责从共享内存度内容，并写入拷贝的文件，从而实现文件的拷贝。

两个进程通过共享内存实现进程之间的通信。

设置共享内存组来实现对硬件延迟的控制，确保文件内容被正确拷贝。共享内存组通过数组小标取余来实现环形缓冲。

本次试验中碰到的问题是开始没有采用生产者-消费者模式，所以每次仍然是读一次写一次，没有利用环形缓冲，后来改为生产者-消费者模式。

实验四 Linux 文件目录

1、 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

2、 实验内容

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。

3、 实验步骤和结果

1. 了解预备知识:

1、Linux 文件属性接口

`int fstat(int fd, struct stat *buf);` 返回文件描述符相关的文件的状态信息

`int stat(const char *path, struct stat *buf);` 通过文件名获取文件信息, 并保存在 `buf` 所指的结构体 `stat` 中

`int lstat(const char *path, struct stat *buf);` 如读取到了符号连接, `lstat` 读取符号连接本身的状态信息, 而 `stat` 读取的是符号连接指向文件的信息。

2、Linux 目录结构接口

`DIR *opendir(const char *name);`

通过路径打开一个目录, 返回一个 `DIR` 结构体指针(目录流), 失败返回 `NULL`;

`struct dirent *readdir(DIR *)`

读取目录中的下一个目录项, 没有目录项可以读取时, 返回为 `NULL`;

`int chdir(const char *path);`

改变目录, 与用户通过 `cd` 命令改变目录一样, 程序也可以通过 `chdir` 来改变目录, 这样使得 `fopen()`, `opendir()`, 这里需要路径的系统调用, 可以使用相对于当前目录的相对路径打开文件(目录)。

`int closedir(DIR*)`

关闭目录流

2. 编写实验代码

(1) 打开一个目录

调用函数 `opendir`, 要判断是否打开成功。

```
if((dp = opendir(dir))==NULL){
    printf("Error\n");
    exit(0);
}
```

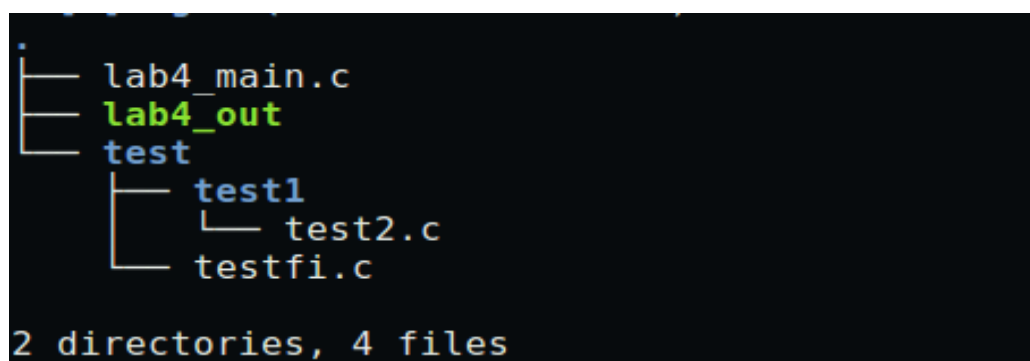
(2)遍历所有目录项，如果是文件夹，打印文件夹信息，然后递归调用函数，如果是文件，打印相关信息。

调用 `readdir` 函数遍历目录项，调用 `lstat` 函数获取目录项信息。对信息解码输出

3.测试实验结果

对源程序编译、链接、生成可执行程序后，运行可执行程序，观察运行结果是否和预想一致。

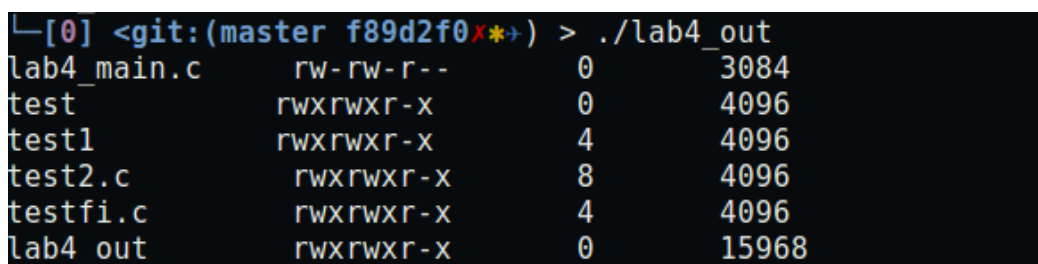
实验环境：Ubuntu 16.04 gcc gdb vim terminal



```

.
├── lab4_main.c
├── lab4_out
└── test
    ├── test1
    │   ├── test2.c
    │   └── testfi.c
    └──
2 directories, 4 files
    
```

图 4-1 用于测试文件结构树形图



```

└─[0] <git:(master f89d2f0) > ./lab4_out
lab4_main.c      rw-rw-r--  0      3084
test            rwxrwxr-x  0      4096
test1           rwxrwxr-x  4      4096
test2.c         rwxrwxr-x  8      4096
testfi.c        rwxrwxr-x  4      4096
lab4_out        rwxrwxr-x  0     15968
    
```

图 4-2 测试文件信息输出截图

通过上图，根据结果分析，可得程序正确输出文件夹和文件的名字，权限，路径深度，文件大小等正确信息，经分析，测试通过。

4、 实验总结和体会

本次试验中，学习了有关文件 Linux 系统下文件借口的相关操作，通过对一些函数的调用，实现一个类似系统函数“ls”的程序，能够正确打印文件的信息。

本次实验的问题在于得到文件的信息后，如何解码，解码后才能得到直观可阅读的文件信息，并且用于输出。

附录

附录为四个实验的源程序，包括：

实验一 进程控制实验 【lab1_main.c】

实验二 线程同步与通信实验 【lab2_main.c】 【lab2.h】

实验三 共享内存与进程同步实验 【lab3_main.c】 【lab3.h】

实验四 Linux 文件目录实验 【lab4_main.c】

【lab1_main.c】

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

```
#include<stdlib.h>
```

```
#include<string.h>
```

```
#include<unistd.h>
```

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

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

```
#include<signal.h>
```

```
typedef void (*stop_handler)(int);
```

```
int pipefd[2];
```

```
int wait1_mark,wait2_mark;
```

```
char string[] = "I send you x times.";
```

```
char readbuffer[80];
```

```
pid_t childpid1,childpid2;
```

```
void waiting1(){
```

```
    int times = 1;
```

```
    while(wait1_mark){
```

```
        if(childpid1 == 0){
```

```
            close(pipefd[0]);
```

```
            string[11] = times%10 + '0';
```

```
            write(pipefd[1],string,(strlen(string)+1));
```

```
            times ++;
```

```
            sleep(1);
```

```
        }
```

```
    }
```

```
}
```

```
void waiting2(){
```

```
    while(wait2_mark){
```

```
        if(childpid2 == 0){
```

```
            close(pipefd[1]);
```

```
            read(pipefd[0],readbuffer,sizeof(readbuffer));
```

```
            printf("%s\n",readbuffer);
```

```
        }
```

```
    }
```

```
}
```

```
void stop1(int sig_no){
    if (sig_no == SIGUSR1) {
        wait1_mark = 0;
        printf("\nChild Process 1 is killed by Parent\n");
        exit(0);
    }
}
```

```
void stop2(int sig_no){
    if (sig_no == SIGUSR2) {
        wait2_mark = 0;
        printf("\nChild Process 2 is killed by Parent\n");
        exit(0);
    }
}
```

```
void stop3(int sig_no) {
    if (sig_no == SIGINT) {
        kill(childpid1,SIGUSR1);
        kill(childpid2,SIGUSR2);
    }
}
```

```
int main(void){
    pipe(pipefd);          //Create the pipe
    stop_handler stop1_t = stop1;
    stop_handler stop2_t = stop2;
    stop_handler stop3_t = stop3;

    signal(SIGINT,stop3_t);

    while((childpid1 = fork())== -1);
    if(childpid1 == 0){
        wait1_mark = 1;
        signal(SIGINT,SIG_IGN);
        signal(SIGUSR1,stop1_t);
        waiting1();
        exit(0);
    }
    else{
        while((childpid2 = fork())== -1);
        if(childpid2 == 0){
```

```

        wait2_mark = 1;
        signal(SIGINT,SIG_IGN);
        signal(SIGUSR2,stop2_t);
        waiting2();
        exit(0);
    }
    else{
        wait(0);
        wait(0);
        printf("\nParent process is killed\n");
        exit(0);
    }
}
return 0;
}

```

【lab2. h】

```

#ifndef _LAB2_H
#define _LAB2_H
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<sys/types.h>
#include<sys/sem.h>
#include<unistd.h>

```

```

union semun{
    int setval;
    struct semid_ds *buf;
    unsigned short *array;
};

```

```

//functional definition:P() V()
void P(int semid,unsigned short index);
void V(int semid,unsigned short index);

```

```

//message light and the handler
int semid;
pthread_t p1,p2;

```

```

//function :
void *subp1();
void *subp2();
#endif

```

【lab2_main. c】

```
#include"lab2.h"
int sum = 0;

int main(){
    int ret1,ret2,ret3;
    union semun sem_args;

    semid = semget(0,2,IPC_CREAT|0666);
    sem_args.setval = 1;
    ret3 = semctl(semid,0,SETVAL,sem_args);
    sem_args.setval =0;
    ret3 = semctl(semid,1,SETVAL,sem_args);
    ret1 = pthread_create(&p1,NULL,subp1,NULL);
    ret2 = pthread_create(&p2,NULL,subp2,NULL);
    if(ret1||ret2){
        printf("Create pthread_child failed\n");
        exit(0);
    }
    pthread_join(p1,NULL);
    pthread_join(p2,NULL);
    semctl(semid,0,IPC_RMID,sem_args);
    return 0;
}
```

```
void P(int semid,unsigned short index){
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = -1;
    sem.sem_flg = 0;
    semop(semid,&sem,1);
}
```

```
void V(int semid,unsigned short index){
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = 1;
    sem.sem_flg = 0;
    semop(semid,&sem,1);
}
```

```
void *subp1(){
```



```

    int i;

    for(i = 1;i <= 100;i ++){
        P(semid,0);
        sum = sum + i;
        V(semid,1);
    }
}

void *subp2(){
    int i;

    for(i = 0;i < 100;i ++){
        P(semid,1);
        printf("%d\t",sum);
        if(i%10==0)printf("\n");
        V(semid,0);
    }
}

【lab3. h】
#ifndef _DATA_H
#define _DATA_H
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
#include<signal.h>
#include<sys/sem.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<string.h>
#include<sys/wait.h>

#define SIZE 10
pid_t p1,p2,p3;
int semid;
int shmid[5];

union semun{
    int setval;
    struct semid_ds *buf;
    unsigned short *array;
};

```

```
//P V function
void P(int semid,unsigned short index);
void V(int semid,unsigned short index);
#endif
【lab3_main.c】
#include"data.h"

int main(){
    int ret1,ret2;
    union semun sem_args;
    struct shmid_ds buf;
    FILE *fp1,*fp2;
    //信号灯
    semid = semget(0,3,IPC_CREAT|0666);
    sem_args.setval = 0;
    ret1 = semctl(semid,0,SETVAL,sem_args);
    sem_args.setval = 5;
    ret2 = semctl(semid,1,SETVAL,sem_args);
    sem_args.setval = 1;
    ret1 = semctl(semid,2,SETVAL,sem_args);
    //创建共享内存组
    shmid[0] = shmget(0,SIZE,IPC_CREAT|0600);
    shmctl(shmid[0],IPC_STAT,&buf);
    shmid[1] = shmget(0,SIZE,IPC_CREAT|0600);
    shmctl(shmid[1],IPC_STAT,&buf);
    shmid[2] = shmget(0,SIZE,IPC_CREAT|0600);
    shmctl(shmid[2],IPC_STAT,&buf);
    shmid[3] = shmget(0,SIZE,IPC_CREAT|0600);
    shmctl(shmid[3],IPC_STAT,&buf);
    shmid[4] = shmget(0,SIZE,IPC_CREAT|0600);
    shmctl(shmid[4],IPC_STAT,&buf);
    //创建进程
    while((p1 = fork()) == -1) printf("error1");

    if(p1 == 0){//进程 1
        char *shmaddr1[5] ;
        int num1 = 0;

        if((fp1=fopen("/home/strawberrylin/Gitrepository/OperationSystem_lab/third_lab/readfile","r"))
        ==NULL) {
            printf("file open error\n");
            exit(-1);
        }
        shmaddr1[0] = (char*)shmat(shmid[0],NULL,0);
```

```

shmaddr1[1] = (char*)shmat(shmid[1],NULL,0);
shmaddr1[2] = (char*)shmat(shmid[2],NULL,0);
shmaddr1[3] = (char*)shmat(shmid[3],NULL,0);
shmaddr1[4] = (char*)shmat(shmid[4],NULL,0);
do{
    P(semid,1);
    P(semid,2);
    num1 = (num1 + 1)%5;
    *shmaddr1[num1] = fgetc(fp1);
    V(semid,2);
    V(semid,0);
}while(*shmaddr1[num1]!=EOF);
shmdt(shmaddr1[0]);
shmdt(shmaddr1[1]);
shmdt(shmaddr1[2]);
shmdt(shmaddr1[3]);
shmdt(shmaddr1[4]);
fclose(fp1);
}else {
    while((p2 = fork()) == -1);
    if(p2 == 0){//进程 2
        char *shmaddr2[5] ;
        int num2 = 0;

if((fp2=fopen("/home/strawberrylin/Gitrepository/OperationSystem_lab/third_lab/writefile","w+"
""))==NULL) {

        printf("file open error\n");
        exit(-1);
    }
    shmaddr2[0] = (char*)shmat(shmid[0],NULL,0);
    shmaddr2[1] = (char*)shmat(shmid[1],NULL,0);
    shmaddr2[2] = (char*)shmat(shmid[2],NULL,0);
    shmaddr2[3] = (char*)shmat(shmid[3],NULL,0);
    shmaddr2[4] = (char*)shmat(shmid[4],NULL,0);
    do{
        P(semid,0);
        P(semid,2);
        num2 = (num2 + 1)%5;
        if(*shmaddr2[num2]!=EOF)
            fputc(*shmaddr2[num2],fp2);
        V(semid,2);
        V(semid,1);
    }while(*shmaddr2[num2]!=EOF);
    shmdt(shmaddr2[0]);

```

```

        shmdt(shmaddr2[1]);
        shmdt(shmaddr2[2]);
        shmdt(shmaddr2[3]);
        shmdt(shmaddr2[4]);
        fclose(fp2);
    }
    else{//主进程
        wait(&p1);
        wait(&p2);
    }
}

semctl(semid,0,IPC_RMID,sem_args);
shmctl(shmid[0],IPC_RMID,NULL);
shmctl(shmid[1],IPC_RMID,NULL);
shmctl(shmid[2],IPC_RMID,NULL);
shmctl(shmid[3],IPC_RMID,NULL);
shmctl(shmid[4],IPC_RMID,NULL);
return 0;
}

```

```

void P(int semid,unsigned short index){
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = -1;
    sem.sem_flg = 0;
    semop(semid,&sem,1);
}

```

```

void V(int semid,unsigned short index){
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = 1;
    sem.sem_flg = 0;
    semop(semid,&sem,1);
}

```

【lab4_main.c】

```

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<dirent.h>

```

```

void printDir(char *dir,int depth){
    DIR *dp;
    struct dirent *entry;
    struct stat statbuf;
    if((dp = opendir(dir))==NULL){
        printf("Error\n");
        exit(0);
    }
    //strcpy(dir,"") I//
    while((entry = readdir(dp))!=NULL){
        lstat(entry->d_name,&statbuf);
        if(entry->d_type == 4){
            if((strcmp(entry->d_name,".")&&(strcmp(entry->d_name,".."))){
                printf("%-15s",entry->d_name);
                if(statbuf.st_mode&S_IRUSR)printf("r");
                else printf("-");
                if(statbuf.st_mode&S_IWUSR)printf("w");
                else printf("-");
                if(statbuf.st_mode&S_IXUSR)printf("x");
                else printf("-");
                if(statbuf.st_mode&S_IRGRP)printf("r");
                else printf("-");
                if(statbuf.st_mode&S_IWGRP)printf("w");
                else printf("-");
                if(statbuf.st_mode&S_IXGRP)printf("x");
                else printf("-");
                if(statbuf.st_mode&S_IROTH)printf("r");
                else printf("-");
                if(statbuf.st_mode&S_IWOTH)printf("w");
                else printf("-");
                if(statbuf.st_mode&S_IXOTH)printf("x");
                else printf("-");
                char *temp = (char *)malloc(sizeof(char) * 100);
                printf("\t%d\t%d\n",depth,statbuf.st_size);
                strcat(temp,dir);
                strcat(temp,entry->d_name);
                strcat(temp,"/");
                printDir(temp,depth + 4);
            }
        }
        else{
            printf("%-15s\t",entry->d_name);
            if(statbuf.st_mode&S_IRUSR)printf("r");
            else printf("-");

```

```

        if(statbuf.st_mode&S_IWUSR)printf("w");
        else printf("-");
        if(statbuf.st_mode&S_IXUSR)printf("x");
        else printf("-");
        if(statbuf.st_mode&S_IRGRP)printf("r");
        else printf("-");
        if(statbuf.st_mode&S_IWGRP)printf("w");
        else printf("-");
        if(statbuf.st_mode&S_IXGRP)printf("x");
        else printf("-");
        if(statbuf.st_mode&S_IROTH)printf("r");
        else printf("-");
        if(statbuf.st_mode&S_IWOTH)printf("w");
        else printf("-");
        if(statbuf.st_mode&S_IXOTH)printf("x");
        else printf("-");
        printf("\t%d\t%d\n",depth,statbuf.st_size);
    }
}
closedir(dp);
}

int main(){
    printDir("/home/strawberrylin/Gitrepository/OperationSystem_lab/forth_lab/",0);
    return 0;
}

```