華中科技大學

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：U201414797

姓　　名：张丹朱

指导教师：张杰

2016 年 12 月 28 日

目　录

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 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!

3. 实验步骤和结果

1. main 函数

父进程创建无名管道并创建两子进程，发送信号给两子进程以结束子进程，等待两子进程结束后再结束。其流程图如下图 1.1 所示：

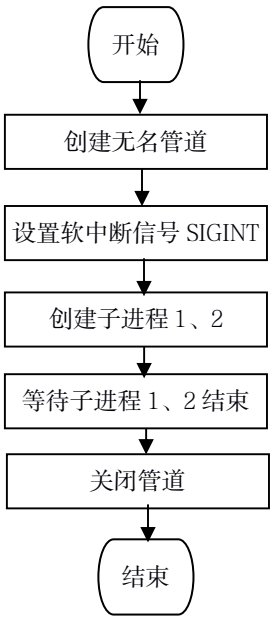

图 1.1　main 函数流程图

2. 父进程信号处理

父进程收到 SIGINT 后分别发送信号 SIGUSR1 和 SIGUSR2 给子进程 1 和 2 以结束这两个子进程。

3. 子进程 1 和 2

子进程首先设置忽略信号 SIGINT，再分别设置信号 SIGUSR1 和 SIGUSR2。

子进程 1 用于发送数据至管道，子进程 2 用于接收管道数据并显示数据，其流程皆循环执行直到收到结束的信号 SIGUSR1（或 SIGUSR2）。其流程图分别如下图 1.2 和 1.3 所示：

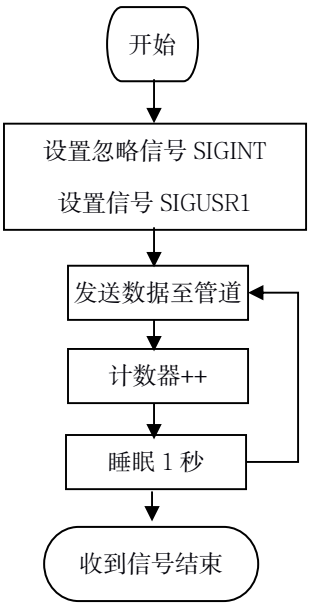

图 1.2　子进程 1 流程图

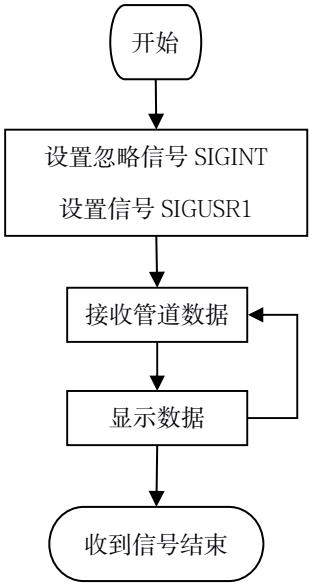

图 1.3　子进程 2 流程图

4. SIGUSR 信号处理

子进程收到父进程发出的信号后关闭管道，显示退出信息然后退出

5. 实验结果

持续打印信息，按下 ctrl+c 后子进程 1 和 2 先结束，最后父进程结束，如下图 1.4 所示：

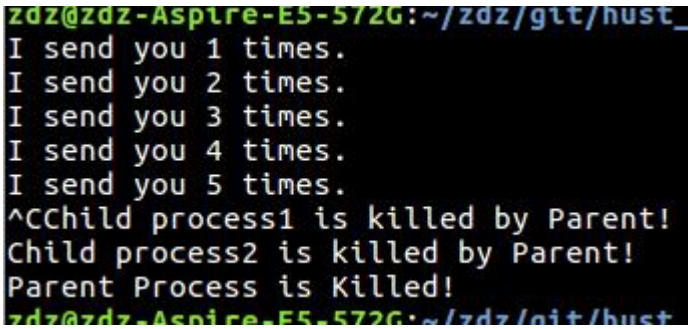

图 1.4　实验 1 结果图

完成实验预期。

4. 实验总结和体会

通过实验，对于无名管道的创建和使用以及进程控制和软中断有了更深入

的了解。实验中利用管道使两个具有亲缘关系的进程进行通信，同时熟悉了信号机制，掌握了通过信号在进程之间传递消息的方法，收获颇多。

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. main 函数

main 函数首先创建两个信号灯，一个表示可算，赋初值为 1，另一个表示可打印，赋初值为 0，使得两个线程可以交替执行且先进行计算。然后创建两个线程，等待两个线程运行结束后再删除信号灯结束程序。

2. 线程 1 和 2

线程 1 负责计算，从 1 到 100 进行累加，每次先对第一个信号灯（即初值为 1 的那个信号灯）进行一次 P 操作，然后计算，最后对另一个信号灯进行一

次 V 操作。

线程 2 负责打印，P、V 的顺序与线程 1 相反。

流程图分别如下图 2.1 和 2.2 所示:

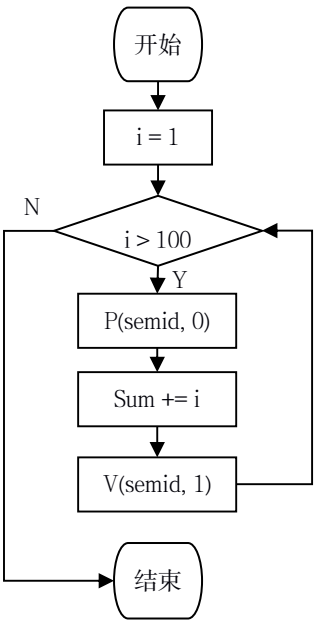

图 2.1　线程 1 流程图

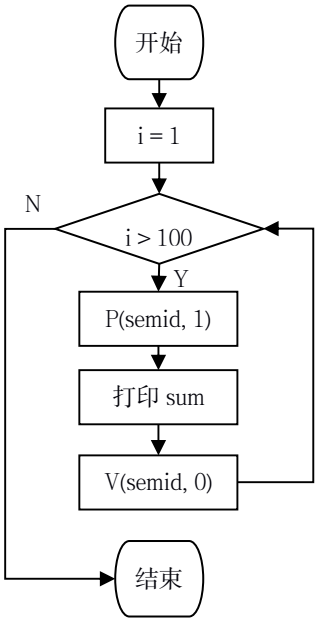

图 2.2　线程 2 流程图

3. 实验结果

如下图 2.3 和图 2.4 所示，依次打印中间结果：

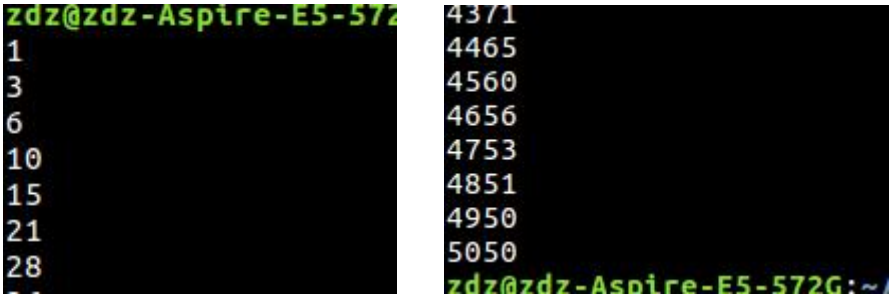

图 2.2 和图 2.3　实验二结果截图（开始与结尾）

符合实验预期。

4. 实验总结和体会

使用信号灯进行两个进程之间的通信时，最主要的是弄清逻辑关系，赋对初值，注意 P、V 操作成对出现。

对于信号量的访问与操作使用 P、V 操作来实现，或是通过 semctrl 进行赋初值与删除等操作，绝不能通过其他方式访问甚至改变信号量的值。

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. 头文件

宏定义信号量、共享内存的键值，以及缓冲区数量和一次最大读取的字符数。另外定义缓冲区的数据结构(包含一个 int 型的数 num 用于记录读入到缓冲区中的字符数量，另用一个字符数组存放从文件中读取到的内容)

2. main 部分

创建信号灯与共享内存，信号灯 0 用于表示空缓冲区，初值为缓冲区个数，信号灯 1 用于表示已写入的缓冲区，初值为 0。接着创建两个子进程，等待子进程结束后删除信号灯与共享内存。

3. writeBuf 部分

执行流程为：首先获取信号量及映射共享内存的地址，接着打开文件，若

打开失败则将第 0 号缓冲的 num 设为 0,同时对 1 号信号灯进行一次 V 操作，再退出。(防止 readBuf 死锁)。

当文件打开成功后，进入循环体，对 0 号信号灯进行一次 P 操作，接着从文件中读入最大为 TEXT_SIZE 的内容到缓冲区，若读入的内容不为空则移到下一个缓冲区，对表示已写入缓冲区资源的 1 号信号灯进行一次 V 操作。否则进行一次 V 操作后退出，表示已经读完了。具体流程图如下图 3.1 所示。

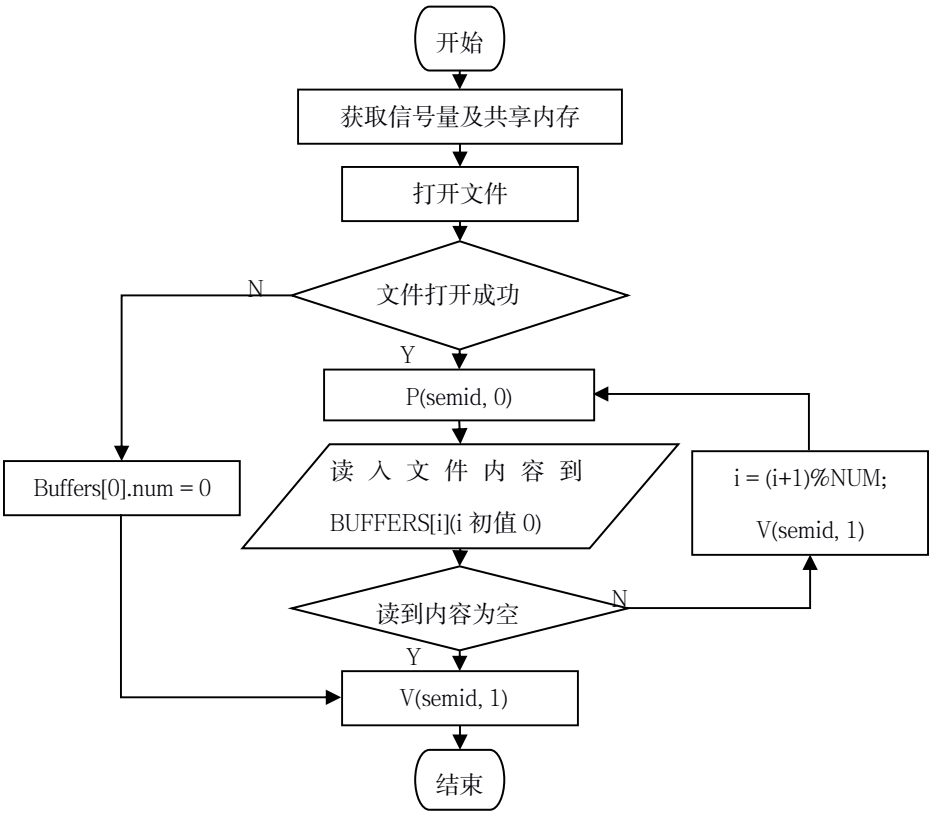

图 3.1　writeBuf 流程图

4. readBuf 部分

本部分与 writeBuf 的主要区别在于循环体部分，是只有在有已写入的缓冲区时才判断是否将该缓冲区内容写入目标文件，若该缓冲区内容不为空则将内容写入目标文件同时移到下一个缓冲区，再对 0 号信号灯进行一次 V 操作。循环体内具体流程见下图 3.2。

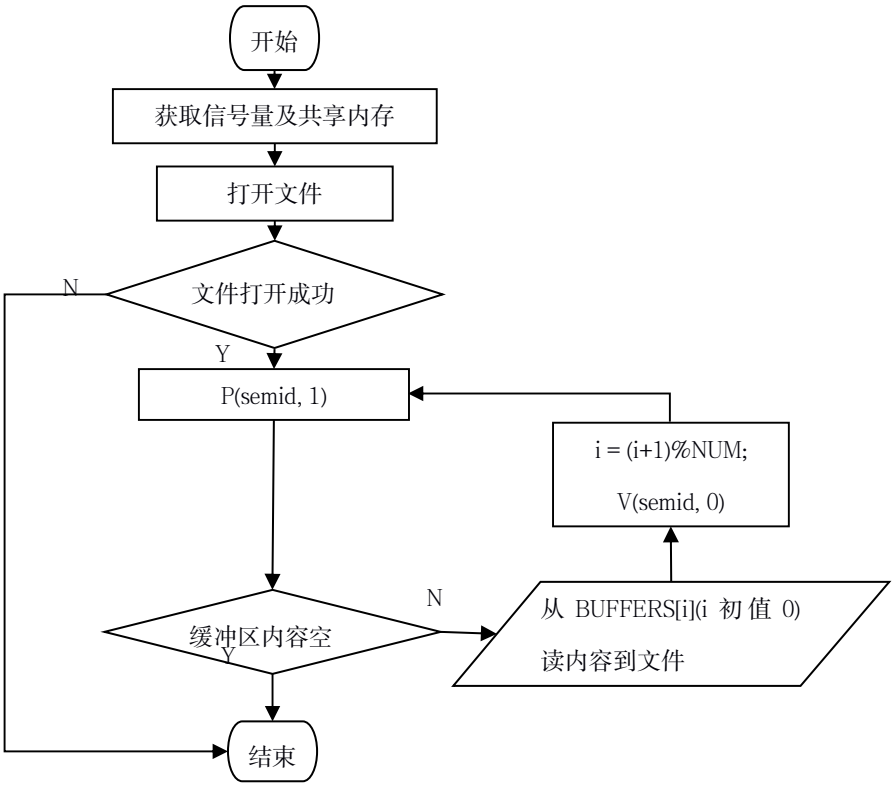

图 3.2　readBuf 流程图

5. 实验结果

实验结果如下图 3.3 所示。

```
$ cat testt.c
cat: testt.c: 没有那个文件或目录
zdz@zdz-Aspire-E5-572G:~/zdz/git/hust_l
$ cat tests.c
#include "stdio.h"

int main()
{
        int sdsadas;
        float kkkkk;
        kkkkk=0;
}
zdz@zdz-Aspire-E5-572G:~/zdz/git/hust_l
$ ./lab3 tests.c testt.c
finish copying!
zdz@zdz-Aspire-E5-572G:~/zdz/git/hust_l
$ cat testt.c
#include "stdio.h"

int main()
{
        int sdsadas;
        float kkkkk;
        kkkkk=0;
}
```

图 3.3　实验三结果截图

首先 testt.c 是不存在的，执行./lab3 tests.c testt.c 后可以看到 testt.c 的内容变成与 tests.c 相同的内容了。

与实验预期相符。

4. 实验总结和体会

本次实验是四次实验中最为复杂的一次，综合进程相关操作、信号量的使用与共享内存的使用，通过实验我对整个进程同步与互斥、共享资源的使用等方面的内容有了更深刻的体会。

本次实验的重点在于理清读、写进程的逻辑关系，设对信号灯的初值。采用数组方式实现循环缓冲，每次写入时同时将读入字符数存入缓冲区中并将此作为 readbuf 的结束标记从而实现誊抄的功能。

在实验过程中曾因搞反了 readBuf 与 WriteBuf 的作用而导致从目标文件读数据写到源文件，产生了重大失误。需总结这次教训，以后在生产者消费者问题上要分清作用，理清逻辑。

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

首先从读入要打印的目录路径，将当前路径置为此。每当读到一个目录项时，获取其相关信息。判断其是否是目录，若是目录，跳过“..”或“.”，打印相关信息后递归调用打印目录信息。否则若是文件直接打印相关信息。读完当前目录中所有目录项后返回父目录。

其中，深度的信息采用缩进的方式来表示，每一层在上一层的基础上行首缩进 4 个空格。

其流程图如下图 4.1 所示：

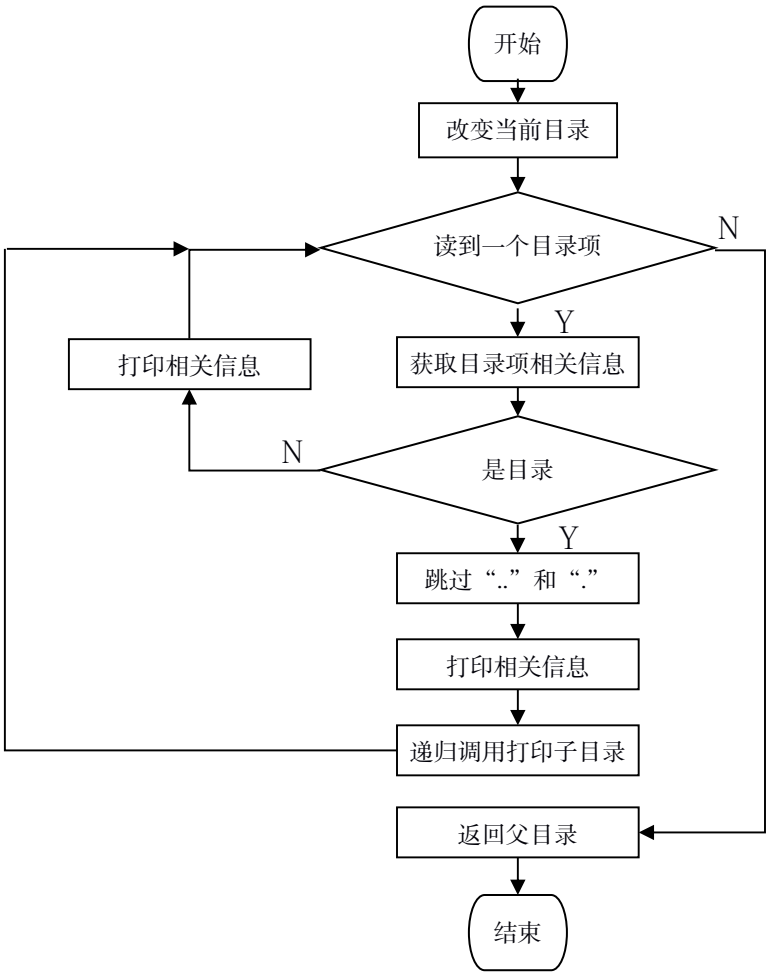

图 4.1　文件目录信息打印流程图

2. 实验结果

实验结果如下图 4.2 所示。上部为 ls –l 所得结果，下部为第四次实验所得结果。

与实验预期相符。

```
zdz@zdz-Aspire-E5-572G:~/zdz/git/hust_labs_2/oslab/lab4$ ls -l
总用量 32
-rw-rw-r-- 1 zdz zdz  2858 12月 27 21:22 lab4.c
-rw-rw-r-- 1 zdz zdz    40 12月 18 11:02 makefile
-rwxrwxr-x 1 zdz zdz 16480 12月 22 11:23 printdir
drwxrwxr-x 2 zdz zdz  4096 12月 24 11:18 testtest
zdz@zdz-Aspire-E5-572G:~/zdz/git/hust_labs_2/oslab/lab4$ ./printdir .
rwxrwxr-x       zdz     zdz     16480   Thu Dec 22 11:23:55 2016        printdir
rwxrwxr-x       zdz     zdz     4096    Sat Dec 24 11:18:49 2016        testtest
    rw-rw-r--   zdz     zdz     9       Sat Dec 24 11:18:49 2016        test.c
rw-rw-r--       zdz     zdz     2858    Tue Dec 27 21:22:22 2016        lab4.c
rw-rw-r--       zdz     zdz     40      Sun Dec 18 11:02:47 2016        makefile
zdz@zdz-Aspire-E5-572G:~/zdz/git/hust_labs_2/oslab/lab4$
```

图 3.3　实验四结果截图

4. 实验总结和体会

本次实验相对来说较为简单，主要是在输出相关信息的时候需要做一些转换使得其能够以正确的形式显示，这一步需要查阅资料才能明白如何进行。

通过本次实验，对于 linux 下的文件系统有了更实际的了解，对于目录项的构成、目录和文件的区别与联系等方面有了深刻的了解。

附录. 源代码

1. 实验一

```
#include "stdio.h"
#include "stdlib.h"
#include "signal.h"
#include "unistd.h"
#include <sys/types.h>
#include <sys/wait.h>

void fsig_deal(int sig);        // 父进程信号处理
void p1();                      // 子进程 1
void p2();                      // 子进程 2
void SIGURS_deal();             // SIGURS1 信号处理
int pid1 = 0, pid2 = 0;
int pipefd[2];

int main(void)
{
    // 创建无名管道
    if(pipe(pipefd) < 0)
    {
        printf("Fail to pipe!\n");
        exit(EXIT_FAILURE);
    }
    // 设置软中断信号 SIGINT
    signal(SIGINT, fsig_deal);
    // 创建子进程 1、2
    pid1 = fork();
    if (pid1 < 0)
    {
        printf("fork error!\n");
    }
    else if (pid1 == 0)   // 若是子进程 1 则执行 p1
    {
        p1();
    }
    else
    {
        pid2 = fork(); // 若是父进程则创建子进程 2
        if (pid2 < 0)
        {
            printf("fork error!\n");
        }
        else if (pid2 == 0) // 若是子进程 2 则执行 p2
        {
            p2();
```

```
            }
        }
        // 等待子进程 1、2 退出
        waitpid(pid1, NULL, 0);
        waitpid(pid2, NULL, 0);
        // 关闭管道
        close(pipefd[0]);
        close(pipefd[1]);
        printf("Parent Process is Killed!\n");
        return 0;
}

// 父进程信号处理
void fsig_deal(int sig)
{
        // 发 SIGUSR1 给子进程 1
        kill(pid1, SIGUSR1);
        // 发 SIGUSR2 给子进程 2
        kill(pid2, SIGUSR2);
}

// 子进程 1
void p1()
{
        int count = 1;
        char buf[50];
        // 设置忽略信号 SIGINT
        signal(SIGINT, SIG_IGN);
        // 设置信号 SIGUSR1
        signal(SIGUSR1, SIGURS_deal);
        while (1)
        {
                // 发送数据至管道数据
                close(pipefd[0]);
                sprintf(buf, "I send you %d times.\n", count);
                write(pipefd[1], buf, sizeof(buf));
                // 计数器++
                count++;
                // 睡眠 1 秒
                sleep(1);
        }
}

// 子进程 2
void p2()
{
        char buf[50];
        // 设置忽略信号 SIGINT
        signal(SIGINT, SIG_IGN);
        // 设置信号 SIGUSR2
        signal(SIGUSR2, SIGURS_deal);
        while (1)
```

```
        {
            // 接收管道数据
            close(pipefd[1]);
            read(pipefd[0], buf, sizeof(buf));
            // 显示数据
            printf("%s", buf);
        }
}

// SIGURS 信号处理
void SIGURS_deal(int sig)
{
        // 关闭管道
        close(pipefd[0]);
        close(pipefd[1]);
        // 显示退出信息
        if (sig == SIGUSR1)
        {
            printf("Child process1 is killed by Parent!\n");
        }
        else if (sig == SIGUSR2)
        {
            printf("Child process2 is killed by Parent!\n");
        }
        // 退出
        exit(EXIT_SUCCESS);
}
```

2. 实验二

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/sem.h>

#define MYKEY 1111

// P、V 操作的函数
void P(int semid, int index);
void V(int semid, int index);
// 信号量、累加中间结果、线程句柄
int semid;
int sum = 0;
pthread_t p1, p2;

// 线程执行函数
void *subp1();
void *subp2();

union semun {
        int     val;
```

```
        struct      semid_ds    *buf;
        unsigned short      *array;
};

// 线程 1 负责计算（1 到 100 的累加，每次加一个数）
void *subp1()
{
        int i;
        for (i = 1; i <= 100; i++)
        {
                P(semid, 0);
                sum += i;
                V(semid, 1);
        }
        return NULL;
}

// 线程 2 负责打印（输出累加的中间结果）
void *subp2()
{
        int j;
        // 从 0 开始输出
        for (j = 1; j <= 100; j++)
        {
                P(semid, 1);
                printf("%d\n", sum);
                V(semid, 0);
        }
        return NULL;
}

void P(int semid, int index)
{
        struct sembuf sem;
        sem.sem_num = index;
        sem.sem_op = -1; // 申请一个共享资源
        sem.sem_flg = 0; // 操作标记：0 或 IPC_NOWAIT 等
        semop(semid, &sem, 1); // 1:表示执行命令的个数
        return;
}

void V(int semid, int index)
{
        struct sembuf sem;
        sem.sem_num = index;
        sem.sem_op = 1; // 释放一个共享资源
        sem.sem_flg = 0;
        semop(semid, &sem, 1);
        return;
}
```

```
int main(void)
{
        union semun    sem_args;
        unsigned short array[2]={1,0};
        sem_args.array = array;
        // 创建信号灯，两个信号灯用于两线程交替执行
        semid = semget(MYKEY,2,IPC_CREAT|0666);
        // 信号灯赋初值
        semctl(semid, 1, SETALL, sem_args);
        // 创建两个线程 subp1、subp2
        pthread_create(&p1, NULL, &subp1, NULL);
        pthread_create(&p2, NULL, &subp2, NULL);
        // 等待两个线程运行结束
        pthread_join(p1, NULL);
        pthread_join(p2, NULL);
        // 删除信号灯
        semctl(semid, 2, IPC_RMID);
        return 0;
}
```

3. 实验三

3.1 头文件 lab3.h

```
#ifndef       LAB3_H
#define       LAB3_H

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/shm.h>

#define SEMKEY 3333
#define       SHMKEY 4444
#define TEXT_SIZE 20
#define       NUM 10                    // 缓冲区数量

// P、V 操作的函数
void P(int semid, int index);
void V(int semid, int index);
// 信号量
int semid;
// 共享内存组
struct shm_st {
        int num;              // 标记是否结束
        char text[TEXT_SIZE]; //记录写入的文本
};
```

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

/* struct shmid_ds {
        uid_t shm_perm.uid;
        uid_t shm_perm.gid;
        mode_t shm_perm.mode;
}; */

void P(int semid, int index)
{
        struct sembuf sem;
        sem.sem_num = index;
        sem.sem_op = -1; // 申请一个共享资源
        sem.sem_flg = 0; // 操作标记：0 或 IPC_NOWAIT 等
        semop(semid, &sem, 1); // 1:表示执行命令的个数
        return;
}

void V(int semid, int index)
{
        struct sembuf sem;
        sem.sem_num = index;
        sem.sem_op = 1; // 释放一个共享资源
        sem.sem_flg = 0;
        semop(semid, &sem, 1);
        return;
}
#endif
```

3.2 主进程 lab3_main.c

```
#include "lab3.h"
int main(int argc, char *argv[])
{
        int pid1, pid2;
        union semun    sem_args;
        unsigned short array[2]={NUM,0};
        sem_args.array = array;

        // 创建共享内存
        int i;
        int shmid;
        shmid = shmget(SHMKEY, sizeof(struct shm_st) * NUM, IPC_CREAT|0666);

        // 创建信号灯
        semid = semget(SEMKEY, 2, IPC_CREAT|0666);
        // 信号灯赋初值
        semctl(semid, 1, SETALL, sem_args);
```

```
    // 创建两进程
    pid1 = fork();
    if (pid1 < 0)
    {
        printf("fork error!\n");
    }
    else if (pid1 == 0)   // 若是子进程 1 则执行 p1
    {
        execv("./lab3_readBuf", argv);
    }
    else
    {
        pid2 = fork(); // 若是父进程则创建子进程 2
        if (pid2 < 0)
        {
            printf("fork error!\n");
        }
        else if (pid2 == 0) // 若是子进程 2 则执行 p2
        {
            execv("./lab3_writeBuf", argv);
        }
    }
    // 等待两个进程运行结束
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
    // 删除信号灯
    semctl(semid, 2, IPC_RMID);
    // 删除共享内存组
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

3.3 lab3_readBuf.c

```
#include "lab3.h"
// 子进程 readBuf 负责从缓冲区读数据到文件
int main(int argc, char *argv[])
{

    int i;
    struct shm_st *buffers;
    int shmid;
    semid = semget(SEMKEY, 2, IPC_CREAT|0666);

    // 获取共享内存组
    shmid = shmget(SHMKEY, sizeof(struct shm_st) * NUM, IPC_CREAT|0666);
    buffers = shmat(shmid, NULL, 0);

    // 打开文件
    FILE *tfp = fopen(argv[2], "w+");
    if (NULL == tfp)
    {
```

```
        printf("Fail to open target file!\n");
        exit(EXIT_FAILURE);
    }

    i = 0;
    while (1)
    {
        P(semid, 1); // 有满的缓冲区才将其写入目标文件
        if (!buffers[i].num)
        {// 结束
            printf("finish copying!\n");
            return 0;
        }
        // 未结束则写入文件同时移到下一个
        fwrite(buffers[i].text, sizeof(char), buffers[i].num, tfp);
        i = (i+1) % NUM;
        V(semid, 0);
    }

    fclose(tfp);
    tfp = NULL;
    return 0;
}
```

3.4 lab3_writeBuf.c

```
#include "lab3.h"
// 子进程 writeBuf 负责写入缓冲区
int main(int argc, char *argv[])
{

    int i;
    struct shm_st *buffers;
    int shmid;
    semid = semget(SEMKEY, 2, IPC_CREAT|0666);

    // 获取共享内存组
    shmid = shmget(SHMKEY, sizeof(struct shm_st) * NUM, IPC_CREAT|0666);
    buffers = shmat(shmid, NULL, 0);

    // 打开文件
    FILE *sfp = fopen(argv[1], "r");
    if (NULL == sfp)
    {
        printf("Fail to open source file!\n");
        buffers[0].num = 0;
        V(semid, 1);
        exit(EXIT_FAILURE);
    }

    i = 0;
    while (1)
```

```
        {
            P(semid, 0); // 有空的缓冲区才读入
            if ((buffers[i].num = fread(buffers[i].text, sizeof(char), TEXT_SIZE, sfp)) != 0)        // 未读完
            {
                i = (i + 1) % NUM;
                V(semid, 1);
            }
            else
            {
                V(semid, 1);
                break ;
            }
        }

        fclose(sfp);
        sfp = NULL;
        return 0;
}
```

4. 实验四

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <dirent.h>
#include <time.h>
#include <pwd.h>
#include <grp.h>

void printright(mode_t mode)
{
        // 用户权限
        mode_t temp;
        if (((temp = mode) & S_IRUSR) == S_IRUSR)
        {
            printf("r");
        }
        else
        {
            printf("-");
        }
        if (((temp = mode) & S_IWUSR) == S_IWUSR)
        {
            printf("w");
        }
        else
        {
            printf("-");
```

```
}
if (((temp = mode) & S_IXUSR) == S_IXUSR)
{
        printf("x");
}
else
{
        printf("-");
}
// 组权限
if (((temp = mode) & S_IRGRP) == S_IRGRP)
{
        printf("r");
}
else
{
        printf("-");
}
if (((temp = mode) & S_IWGRP) == S_IWGRP)
{
        printf("w");
}
else
{
        printf("-");
}
if (((temp = mode) & S_IXGRP) == S_IXGRP)
{
        printf("x");
}
else
{
        printf("-");
}
// 其他
if (((temp = mode) & S_IROTH) == S_IROTH)
{
        printf("r");
}
else
{
        printf("-");
}
if (((temp = mode) & S_IWOTH) == S_IWOTH)
{
        printf("w");
}
else
{
        printf("-");
}
if (((temp = mode) & S_IXOTH) == S_IXOTH)
```

```
        {
            printf("x");
        }
        else
        {
            printf("-");
        }

        printf("\t");
    }

    void printdir(char *dir, int depth)
    {
        DIR *dp;
        struct dirent *entry;
        struct stat statbuf;
        int i;
        struct passwd *u_info;
        struct group *g_info;
        char *timebuf;
        if ((dp = opendir(dir)) == NULL)     // 打开目录失败
        {
            printf("Fail to open dir!");
            return ;
        }
        chdir(dir);

        while ((entry = readdir(dp)) != NULL)      // 读到一个目录项
        {
            lstat(entry->d_name, &statbuf);     // 得到该目录项相关信息
            u_info = getpwuid(statbuf.st_uid);
            g_info = getgrgid(statbuf.st_gid);
            if (S_ISDIR(statbuf.st_mode))// 是目录
            {
                if (!strcmp(entry->d_name, "..") || !strcmp(entry->d_name, "."))
                {       // 跳过目录项名".."或"."的
                    continue ;
                }
                for (i = 0; i < depth; i++) // 打印层数
                {
                    printf(" ");
                }
                // 打印目录项信息
                printright(statbuf.st_mode);   // 打印权限
                timebuf = ctime(&statbuf.st_ctime);// 获取创建时间
                timebuf[strlen(timebuf)-1] = '\0';    // 去掉‘\n’
                printf("%s\t%s\t%ld\t%s\t%s\n",    u_info->pw_name,    g_info->gr_name,    statbuf.st_size,
timebuf, entry->d_name);
                // 递归调用 printdir，打印子目录信息
                printdir(entry->d_name, depth+4);
            }
            else  // 打印文件相关信息
```

```
                {
                        for (i = 0; i < depth; i++) // 打印层数
                        {
                                printf(" ");
                        }
                        printright(statbuf.st_mode);   // 打印权限
                        timebuf = ctime(&statbuf.st_ctime);
                        timebuf[strlen(timebuf)-1] = '\0';   // 去掉‘\n’
                        printf("%s\t%s\t%ld\t%s\t%s\n",   u_info->pw_name,   g_info->gr_name,   statbuf.st_size,
timebuf, entry->d_name);
                }
        }
        // 返回父目录
        chdir("..");
        // 关闭目录项
        closedir(dp);
}

int main(int argc, char *argv[])
{
        if (argc != 2)
        {
                printf("Usage：\n./printdir DIR_NAME\n");
                return 0;
        }
        printdir(argv[1], 0);
        return 0;
}
```