

目 录

1	实验一 进程控制	1
1.1	实验目的	1
1.2	实验内容	1
1.3	实验设计	1
1.3.1	开发环境	1
1.3.2	实验设计	1
1.4	实验调试	4
1.4.1	实验步骤	4
1.4.2	实验调试及心得	4
	附录 实验代码	5
2	实验二 线程同步与通信	8
2.1	实验目的	8
2.2	实验内容	8
2.3	实验设计	8
2.3.1	开发环境	8
2.3.2	实验设计	8
2.4	实验调试	9
2.4.1	实验步骤	9
2.4.2	实验调试及心得	9
	附录 实验代码	11
3	实验三 共享内存与进程同步	17
3.1	实验目的	17
3.2	实验内容	17
3.3	实验设计	17
3.3.1	开发环境	17
3.3.2	实验设计	17
3.4	实验调试	20
3.4.1	实验步骤	20
3.4.2	实验调试及心得	20
	附录 实验代码	22
4	实验四 Linux 文件目录	30
4.1	实验目的	30
4.2	实验内容	30
4.3	实验设计	30
4.3.1	开发环境	30
4.3.2	实验设计	30
4.4	实验调试	32
4.4.1	实验步骤	32
4.4.2	实验调试及心得	32
	附录 实验代码	34

1 实验一 进程控制

1.1 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。

1.2 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!

1.3 实验设计

1.3.1 开发环境

ubuntu_1804

1.3.2 实验设计

根据实验指导,在 `main` 函数里要完成创建无名管道、设置软中断信号

SIGINT、创建子进程 1、2、等待子进程 1、2 退出、关闭管道的工作。同时，父进程信号处理函数的工作为向子进程 1、2 发送信号 SIGUSR1 和 SIGUSR2，告诉子进程结束自己。子进程 1 的工作为设置忽略信号 SIGINT、设置信号 SIGUSR1、发送数据至管道。子进程 2 的工作为设置忽略信号 SIGINT、设置信号 SIGUSR2、接收管道数据，显示数据。子进程 1 和 2 接收到 SIGUSR1 和 SIGUSR2 信号后，需要关闭管道，显示退出信息、结束自己。

下面为父进程与子进程的流程图。

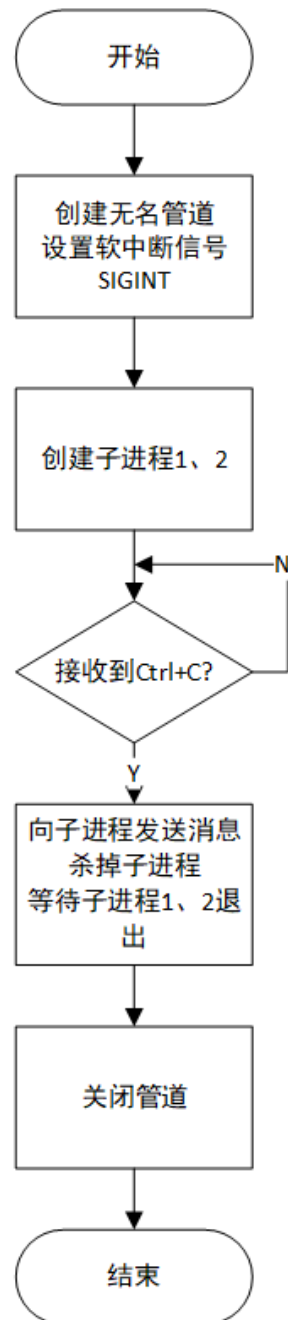


图 1-1 父进程流程图

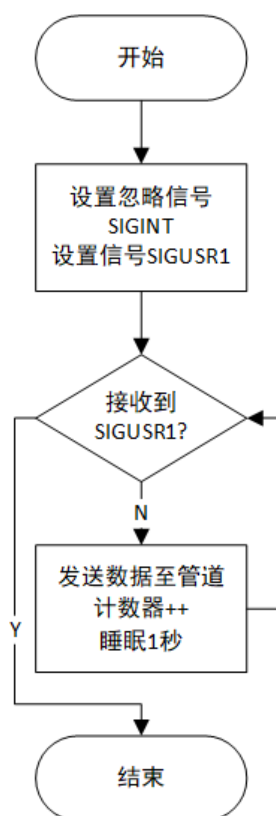


图 1-2 子进程 1 流程图

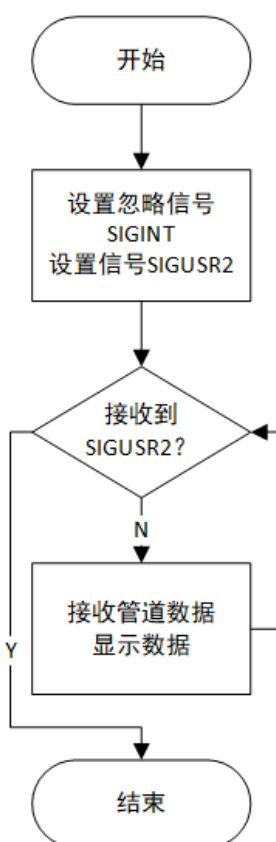


图 1-3 子进程 2 流程图

1.4 实验调试

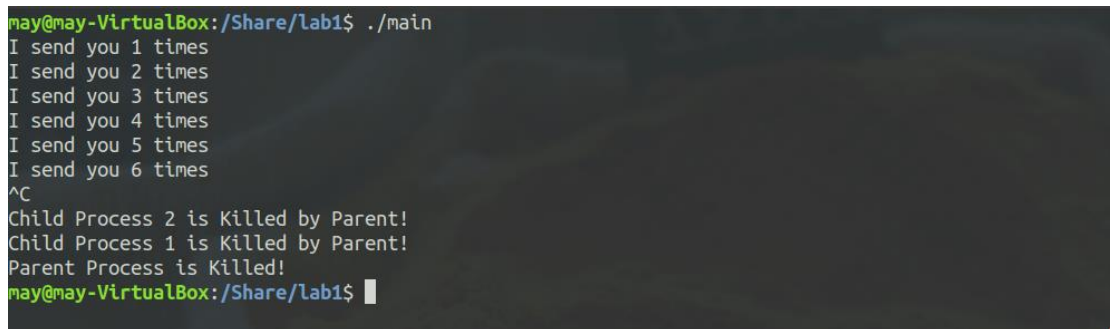
1.4.1 实验步骤

- 1、仔细阅读实验指导 PPT，熟悉相关 API 函数。
- 2、编写 main 函数，并且编写子进程 1、2 的代码。
- 3、编写信号处理函数。
- 4、编译，运行代码，并且进行测试与调试。

1.4.2 实验调试及心得

- 1、运行结果

运行结果如下图：



```
may@may-VirtualBox:/Share/Lab1$ ./main
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
^C
Child Process 2 is Killed by Parent!
Child Process 1 is Killed by Parent!
Parent Process is Killed!
may@may-VirtualBox:/Share/Lab1$
```

图 1-4 测试图

如上图，当接收到来自键盘的 Ctrl+C 信号后，程序结束，测试通过。

- 2、实验调试

实验编写没有太大问题，没有碰到明显 bug。

- 3、实验心得

本次实验加深了我对进程的理解，对进程争用资源的现象有了实质的认识，通过学习解决了进程争用资源的问题。通过编写一个简单的程序，掌握了 linux 进程的进本控制。同时也掌握了 linux 系统中的软中断和管道通信。其中新颖一点的知识是使用无名管道实现子进程之间的通信，同时了解到管道是半双工的，数据只能单向流动，一方通信时，必须建立两个管道，因此这就是 main 函数结束后还要关闭一次管道释放资源的原因。

附录 实验代码

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

pid_t p1, p2;          //子进程
int pipe_fd[2];        //管道

void my_func(int sig_no) {
    if(sig_no==SIGINT){//向 p1、 p2 发消息
        kill(p1, SIGUSR1);
        kill(p2, SIGUSR2);
    }
    if(sig_no==SIGUSR1){
        close(pipe_fd[1]);//关闭写
        printf("\nChild Process 1 is Killed by Parent!");
        exit(0);
    }
    if(sig_no==SIGUSR2){
        close(pipe_fd[0]);//关闭读
        printf("\nChild Process 2 is Killed by Parent!");
        exit(0);
    }
}

int main(void) {
    char writebuf[100];    //写缓冲区
    char readbuf[100];     //读缓冲区
    int status;            //子进程状态信息，用于等待结束

    if (pipe(pipe_fd) < 0) {
```

```

    printf("Creat pipe fail.\n");
    exit(-1);
}

signal(SIGINT, my_func); //设置软中断信号

while ((p1 = fork()) == -1) ;
if (p1 == 0) {
    //子进程 1
    int count = 1;
    signal(SIGINT, SIG_IGN);      //忽略 SIGINT
    signal(SIGUSR1, my_func);     //处理子进程 1
    //写数据
    close(pipe_fd[0]);
    while (1) {
        sprintf(writebuf, "I send you %d times\n", count++);
        if (write(pipe_fd[1], writebuf, 100) < 0) {
            printf("Fail to write.\n");
            exit(-1);
        }
        sleep(1);
    }
}
else {
    //父进程执行
    while ((p2 = fork()) == -1) ;
    if (p2 == 0) {
        //子进程 2
        signal(SIGINT, SIG_IGN);    //忽略 SIGINT
        signal(SIGUSR2, my_func);   //处理子进程 2
        //读数据
        close(pipe_fd[1]);
        while (1) {
            if (read(pipe_fd[0], readbuf, 100) < 0) {
                printf("Fail to read.\n");
            }
        }
    }
}
}

```

```

        exit(-1);
    }
    printf("%s",readbuf);
}
}
else {
    //父进程执行
    for (int i = 0; i < 2;i++) {
        waitpid(-1, &status, 0);//等待两个子进程
    }
    //关闭管道
    close(pipe_fd[0]);
    close(pipe_fd[1]);
    printf("\nParent Process is Killed!\n");
}
}

return 0;
}

```


2 实验二 线程同步与通信

2.1 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。

2.2 实验内容

1、线程同步

设计并实现一个计算线程与一个 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);
- 主进程等待子线程退出。

2、线程互斥 (选做)

编程模拟实现飞机售票：

- 创建多个售票线程；
- 已售票使用公用全局变量；
- 创建互斥信号灯；
- 对售票线程临界区施加 P、V 操作；
- 主进程等待子线程退出，各线程在票卖完时退出。

2.3 实验设计

2.3.1 开发环境

ubuntu_1804

2.3.2 实验设计

对于线程同步实验，其实是一个生产者消费者问题。根据实验指导 PPT, main

函数要

完成创建信号灯、信号，灯赋初值、创建两个线程、等待两个线程运行结束、删除信号灯的操作，这些操作使用到的函数全部都在 PPT 中提及到了。线程 1 负责计算，计算过程中设置一个加数（每循环一次就进行加 1 的操作），然后结果为自身加上这个加数。首先对一个空缓冲区进行 P 操作，进行计算后，对一个满缓冲区进行 V 操作。线程 2 负责打印，对一个满缓冲区进行 P 操作，打印完后，对一个空缓冲区进行 V 操作。

对于线程互斥实验，创建两个售票线程，已售票使用公共全局 sold。为 sold 创建互斥信号灯，这样就能使程序正常运行了。大部分和线程同步实验类似。线程 1、2 都是进行相同的操作，当售的票小于总数时，对信号灯进行 P 操作，sold++，打印信息，然后对信号灯进行 V 操作。为了显示出多线程在工作，让每个线程睡眠 1s，目的是为了保证能看到多线程工作，不然一个线程在一个时间片内就能将所有的票卖完（票数少的情况下）。

2.4 实验调试

2.4.1 实验步骤

- 1、仔细阅读实验指导 PPT，熟悉相关 API 函数。
- 2、编写 main 函数，并且编写子进程 1、2 的代码。
- 3、编写信号处理函数。
- 4、编译，运行代码，并且进行测试与调试。

2.4.2 实验调试及心得

1、运行结果

运行结果如下图：

如下图，程序输出正确，测试通过。

对线程同步实验，正确输出计算结果 5050。

对线程互斥实验，正确卖票。

```
a=3321
a=3403
a=3486
a=3570
a=3655
a=3741
a=3828
a=3916
a=4005
a=4095
a=4186
a=4278
a=4371
a=4465
a=4560
a=4656
a=4753
a=4851
a=4950
a=5050
may@may-VirtualBox: /Share/Lab2$
```

图 2-1 线程同步测试图

```
may@may-VirtualBox: /Share/Lab2$ ./lab2_1
thread 2 sells 1
thread 1 sells 2
thread 2 sells 3
thread 1 sells 4
thread 2 sells 5
thread 1 sells 6
thread 2 sells 7
thread 1 sells 8
thread 2 sells 9
thread 1 sells 10
thread 2 sells 11
thread 1 sells 12
thread 2 sells 13
thread 1 sells 14
thread 2 sells 15
may@may-VirtualBox: /Share/Lab2$
```

图 2-2 线程互斥测试图

2、实验调试

实验编写没有太大问题，没有碰到明显 bug。

在编译的时候，要加上 `-lpthread` 进行编译，因为 linux 默认不加 pthread 库，要在编译的时候进行动态链接。

3、实验心得

本次实验加深了我对线程的理解，线程与进程之间有很大联系又有一定的区别，虽然在实验中没有很深刻地体会到两者的差别，但是在实际的使用过程中还是有一定的限制，线程是轻量级的，不会将数据拷贝到自己的存储空间。同时，在本次实验中最大的收获就是学会了信号灯的设置，以及 P、V 操作的实现。同时，能用 P、V 操作实现线程的同步与互斥。把我们课堂上学到的知识运用到实践中。

附录 实验代码

线程同步:

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

int semid,a=0;
int add_num=1;

union semun {
    int          val;    // value for SETVAL
    struct semid_ds *buf;  // buffer for IPC_STAT, IPC_SET
    unsigned short *array; // array for GETALL, SETALL
    struct seminfo *__buf; // buffer for IPC_INFO (Linux-specific)
};

void P(int semid, int index)
{
    struct sembuf sem;
    sem.sem_num = index; /*要操作的信号灯的编号*/
    sem.sem_op = -1; /*要执行的操作*/
    sem.sem_flg = 0; /*操作标志，一般设置为 0*/
    semop(semid,&sem,1);
    return;
}

void V(int semid,int index)
{
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = 1;
```

```

        sem.sem_flg = 0;
        semop(semid,&sem,1);
        return;
    }

void *thread1(void *arg)
{
    for(int i=0;i<100;i++)
    {
        P(semid,0);
        a+=add_num;
        add_num++;
        V(semid,1);
    }
}

void *thread2(void *arg)
{
    for(int i=0;i<100;i++)
    {
        P(semid,1);
        printf("a=%d\n",a);
        V(semid,0);
    }
}

int main()
{
    pthread_t pid1,pid2;
    int ret;
    union semun arg1;
    union semun arg2;

    //创建信号灯
    semid=semget(IPC_PRIVATE,2,IPC_CREAT|0666);

```

```

    arg1.val=1;
    arg2.val=0;

    //信号灯赋值
    semctl(semid,0,SETVAL,arg1);//empty buffer
    semctl(semid,1,SETVAL,arg2);//full buffer

    //创建两个线程
    while((ret=pthread_create(&pid1,NULL,thread1,NULL))!=0);
    while((ret=pthread_create(&pid2,NULL,thread2,NULL))!=0);

    //等待线程结束
    pthread_join(pid1,NULL);
    pthread_join(pid2,NULL);

    //删除信号灯集
    semctl(semid, 0, IPC_RMID);

}

```

线程互斥：

```

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

```

```
int semid;
```

```
int sold=0;
```

```
const int total=15;
```

```
union semun {
```

```

    int            val;    // value for SETVAL
    struct semid_ds *buf;   // buffer for IPC_STAT, IPC_SET
    unsigned short *array; // array for GETALL, SETALL
    struct seminfo  *__buf; // buffer for IPC_INFO (Linux-specific)
};

```

```

void P(int semid, int index)

```

```

{
    struct sembuf sem;
    sem.sem_num = index; /*要操作的信号灯的编号*/
    sem.sem_op = -1; /*要执行的操作*/
    sem.sem_flg = 0; /*操作标志，一般设置为 0*/
    semop(semid,&sem,1);
    return;
}

```

```

void V(int semid,int index)

```

```

{
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = 1;
    sem.sem_flg = 0;
    semop(semid,&sem,1);
    return;
}

```

```

void *thread1(void *arg)

```

```

{
    while(sold<total)
    {
        P(semid,0);
        sold++;
        printf("thread 1 sells %d\n",sold);
        V(semid,0);
        sleep(1);
    }
}

```

```

    }
}

void *thread2(void *arg)
{
    while(sold<total)
    {
        P(semid,0);
        sold++;
        printf("thread 2 sells %d\n",sold);
        V(semid,0);
        sleep(1);
    }
}

int main()
{
    pthread_t pid1,pid2;
    int ret;
    union semun arg1;

    //创建信号灯
    semid=semget(IPC_PRIVATE,1,IPC_CREAT|0666);
    arg1.val=1;

    //信号灯赋值
    semctl(semid,0,SETVAL,arg1);

    //创建两个线程
    while((ret=pthread_create(&pid1,NULL,thread1,NULL))!=0);
    while((ret=pthread_create(&pid2,NULL,thread2,NULL))!=0);

    //等待线程结束
    pthread_join(pid1,NULL);
    pthread_join(pid2,NULL);
}

```



```
//删除信号灯集  
semctl(semid, 0, IPC_RMID);  
  
}
```

3 实验三 共享内存与进程同步

3.1 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；
- 2、掌握 Linux 下进程同步与通信的主要机制。

3.2 实验内容

1. 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。
2. （选做）对下列参数设置不同的取值，统计程序并发执行的个体和总体执行时间，分析不同设置对缓冲效果和进程并发执行的性能影响，并分析其原因：
 - （1）信号灯的设置；
 - （2）缓冲区的个数；
 - （3）进程执行的相对速度。

3.3 实验设计

3.3.1 开发环境

ubuntu_1804

3.3.2 实验设计

首先定义共享内存区结构体，结构体包含两个成员。分别为存储的数据，与存储的数据长度。同时还要定义存储的数据最大大小、共享内存的数量以及共享内存的起始 key 值（用于子进程获取共享内存区）。

在 writebuf 进程内，首先获取两个信号灯，代表空闲缓冲区以及满缓冲区，然后获取 main 里创建的共享内存区。首先对空闲缓冲区进行一次 P 操作，然后将共享内存区连接到当前进程的地址空间中，然后读取规定大小的数据存放在共享内存区的数据区，返回读取数据的实际长度，然后对满缓冲区进行一次 V 操作。撤离当前共享内存区，指向下一共享内存区。如果实际读取的长度小于规定大小，则结束循环。

同理，在 `writebuf` 进程内，首先获取两个信号灯，代表空闲缓冲区以及满缓冲区，然后获取 `main` 里创建的共享内存区。首先对满缓冲区进行一次 `P` 操作，然后将共享内存区连接到当前进程的地址空间中，然后将共享内存区的数据区的内容写入到目标文件中，返回共享内存区的数据区的内容的实际长度，然后对空闲缓冲区进行一次 `V` 操作。撤离当前共享内存区，指向下一共享内存区。如果共享内存区的数据区的内容的实际长度小于规定大小，则结束循环。

在 `main` 函数中，产生两个信号灯并对它们赋初值。然后产生共享内存区。创建两个子进程分别执行 `writebuf` 和 `readbuf`。等待两个子进程结束后，删除共享内存区和信号灯，结束。

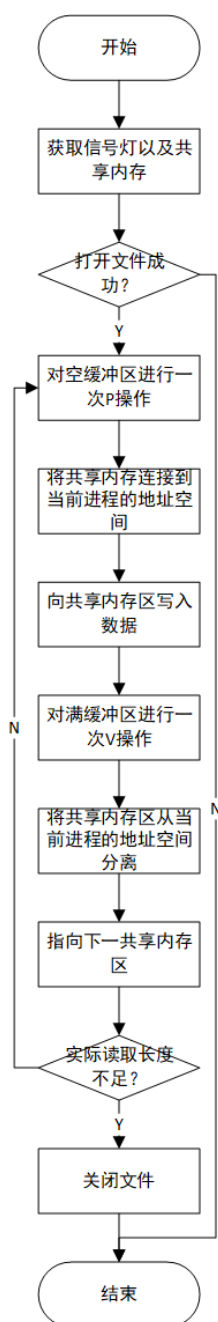


图 3-1 `writebuf` 流程图

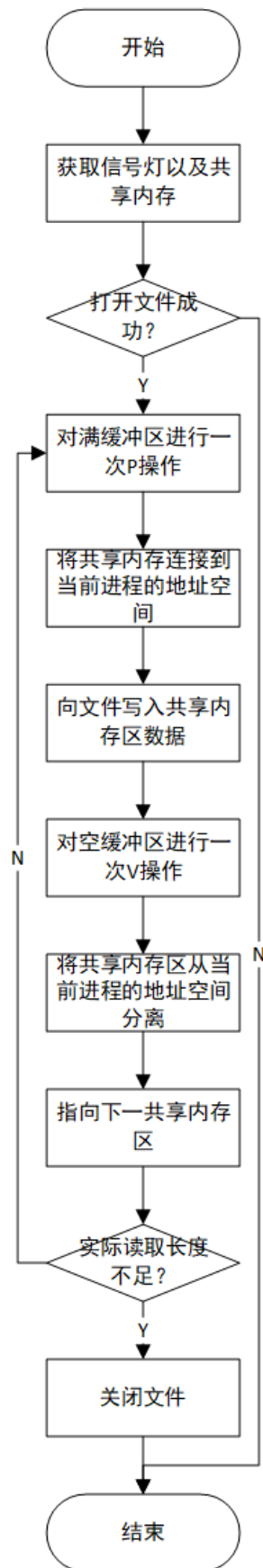


图 3-2 readbuf 流程图

3.4 实验调试

3.4.1 实验步骤

- 1、仔细阅读实验指导 PPT，熟悉相关 API 函数。
- 2、编写 main 函数，并且编写子进程 writebuf、readbuf 的代码。
- 3、编译，运行代码，并且进行测试与调试。

3.4.2 实验调试及心得

1、运行结果

运行结果如下图：

如下图，誊抄后产生 dst 文件。比较两个文件，发现没有差异，测试通过。

```
may@may-VirtualBox:/Share/lab3$ ./main
Copy ...
Done.
may@may-VirtualBox:/Share/lab3$ diff src dst
may@may-VirtualBox:/Share/lab3$
```

图 3-3 测试图

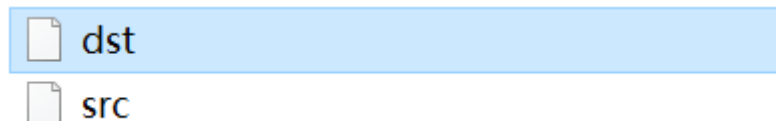


图 3-4 测试图

2、实验调试

实验编写时遇到 `execv` 函数不能使用的情况，再查找资料后使用了 `execlp` 函数进行替代。

在编写 `readbuf` 和 `writebuf` 代码的时候，对进程该何时结束判断模糊，导致最后誊抄不能正常完成，在进行仔细的思考后，发现判断条件出了问题，后来利用读到的实际长度作为进程是否结束的标志，如果实际长度不够，则结束进程。之后发现程序能够正常完成誊抄工作。

3、实验心得

本次实验让我掌握了 `linux` 下共享内存的概念与使用方法。共享内存也是进程间进行通信的一种方法，在之前几次实验中我也学到了其他的方法，这次实验则是提出了共享内存的概念。与此同时，我还掌握了环形缓冲的结构与使用方法，知道了它的工作原理，将课堂上的知识运用到实践中来了。结合前面几次实验，

我对 Linux 进程间的不同的通信方式有了广泛的了解。

在本次实验的选作题中，经过对不同参数的设置，发现缓冲区个数在适中的情况下，程序并发执行的性能更佳。缓冲区个数太少，则不能充分利用 CPU 的高速特性，太多了则会导致在一个时间片内进程不能写完所有的缓冲区，这些都可能導致性能下降。同时进程执行的相对速度也会影响性能，当两个进程的速度相当的时候，程序并发执行的性能更佳，否则会出现一个进程等待另外一个进程的现象，降低性能。在信号灯的设置方面没有太好的实验效果，猜测信号灯越多，性能越低，因为信号灯的存在要求进行 P、V 操作，这些都是很浪费时间的。

附录 实验代码

头文件

```
#ifndef MYSTRUCT_H
#define MYSTRUCT_H

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define DATA_SIZE 100000
#define BUF_SIZE 10 //共享内存数量
#define s_addr 100
```

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

```
typedef struct sh_mem
{
    char data[DATA_SIZE];
    int length;
}sh_mem;
```

```
#endif
```

Writebuf

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>
#include <fcntl.h>
```

```

#include "my_struct.h"

int buf_empty;//空闲缓冲区
int buf_full;//满缓冲区

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

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

int main() {
    int fp;
    int shm_id[BUF_SIZE];
    sh_mem *p;

    //获取信号灯
    buf_empty = semget(1,1,IPC_CREAT|0666);
    buf_full = semget(2,1,IPC_CREAT|0666);

    //获取共享内存区

```



```

for(int i=0;i<BUF_SIZE;++i)
{
    shm_id[i] = shmget(s_addr+i,sizeof(sh_mem),IPC_CREAT|0666);
}

//打开文件
if((fp = open("src",O_RDONLY)) == -1)
{
    printf("Fail to open file.\n");
}

int i=0,temp=0;
while (1) {
    P(buf_empty,0);
    p=(sh_mem *)shmat(shm_id[i],0,0);
    p->length=read(fp, p->data, DATA_SIZE);
    V(buf_full,0);
    temp=p->length;//判断结束条件
    shmdt(p);
    i=(i+1)%BUF_SIZE;//环形缓冲区
    if(temp<DATA_SIZE) break;
}

//关闭文件
close(fp);

return 0;
}

```

Readbuf

```

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

```

```

#include<fcntl.h>
#include "my_struct.h"

int buf_empty;//空闲缓冲区
int buf_full;//满缓冲区

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

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

int main() {
    int fp;
    int shm_id[BUF_SIZE];
    sh_mem *p;

    //获取信号灯
    buf_empty = semget(1,1,IPC_CREAT|0666);
    buf_full = semget(2,1,IPC_CREAT|0666);

```

```

//获取共享内存区
for(int i=0;i<BUF_SIZE;++i)
{
    shm_id[i] = shmget(s_addr+i,sizeof(sh_mem),IPC_CREAT|0666);
}

//打开文件
if((fp = open("dst",O_WRONLY|O_CREAT,0666)) == -1)
{
    printf("Fail to open file.\n");
}

int i=0,temp=0;
while (1) {
    P(buf_full,0);
    p=(sh_mem *)shmat(shm_id[i],0,0);
    write(fp,p->data,p->length);
    V(buf_empty,0);
    temp=p->length;//判断结束条件
    shmdt(p);
    i=(i+1)%BUF_SIZE;//环形缓冲区
    if(temp<DATA_SIZE) break;
}

//关闭文件
close(fp);

return 0;
}

```

Main

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

```

```

#include <sys/shm.h>
#include <sys/wait.h>
#include "my_struct.h"

int buf_empty;//空闲缓冲区
int buf_full;//满缓冲区

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

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

int main() {
    int status;//子进程状态
    pid_t writebuf_pid;
    pid_t readbuf_pid;
    int shm_id[BUF_SIZE];//共享内存
    sh_mem *p;

    //产生信号灯

```

```

buf_empty = semget(1,1,IPC_CREAT|0666);
buf_full  = semget(2,1,IPC_CREAT|0666);

//赋值
union semun arg1;
union semun arg2;
arg1.val=10;
arg2.val=0;
semctl(buf_empty,0,SETVAL,arg1);
semctl(buf_full,0,SETVAL,arg2);

//产生共享内存区
for(int i=0;i<BUF_SIZE;++i)
{
    shm_id[i] = shmget(s_addr+i,sizeof(sh_mem),IPC_CREAT|0666);
}

printf("Copy ... \n");

while ((writebuf_pid = fork()) == -1) ;
if (writebuf_pid == 0) { //子进程
    execlp("./writebuf", "writebuf", NULL);
} else { //main
    while ((readbuf_pid = fork()) == -1);
    if (readbuf_pid == 0) { //子进程
        execlp("./readbuf", "readbuf", NULL);
    } else { // main
        //等待子进程结束
        for (int i = 0; i < 2; i++) {
            waitpid(-1, &status, 0);
        }
        printf("Done.\n");

        //删除共享内存区
        for(int i=0;i<BUF_SIZE;++i)

```

```
    {  
        shmctl(shm_id[i],IPC_RMID,0);  
    }  
  
    //删除信号灯  
    semctl(buf_empty,0,IPC_RMID);  
    semctl(buf_full,0,IPC_RMID);  
  
    return 0;  
}  
}  
}
```

4 实验四 Linux 文件目录

4.1 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。

4.2 实验内容

编程实现目录查询功能：

- 1、功能类似 `ls -lR`；
- 2、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；
- 3、递归显示子目录中的所有文件信息。

4.3 实验设计

4.3.1 开发环境

ubuntu_1804

4.3.2 实验设计

为了完成实验所需功能，设计了两个模块来完成打印文件信息和递归打目录信息。

- 1、打印文件信息

根据 ppt 中给出的 `stat` 结构体的信息，可以方便地获取文件的信息。

首先要获取文件类型信息，通过 `statbuf.st_mode` 与上 `S_IFMT` (`S_IFMT` 是一个掩码，它的值是 `017000` (八进制)，用来过滤出前四位表示的文件类型)，然后只需要判断结果是否与对应文件的标识相同即可。然后获取权限信息，包括个人，同组，其他人的权限，这个也是要用 `statbuf.st_mode` 与上对应的宏定义来判断是否有相应的权限，如读、写、执行。然后获取硬连接的数目，文件所有者的名字，组名字。这里要通过 `getpwuid` 和 `getgrpid` 来获取用户 `uid` 和组 `uid` 信息，然后才能获取名字信息。最后获取文件大小，文件最后修改时间信息，在获取文

件最后修改时间时，不能直接输出 statbuf.st_mtime 信息，要用 ctime 函数转换时间格式输出。

2、递归打印目录信息

此模块流程图如下所示：

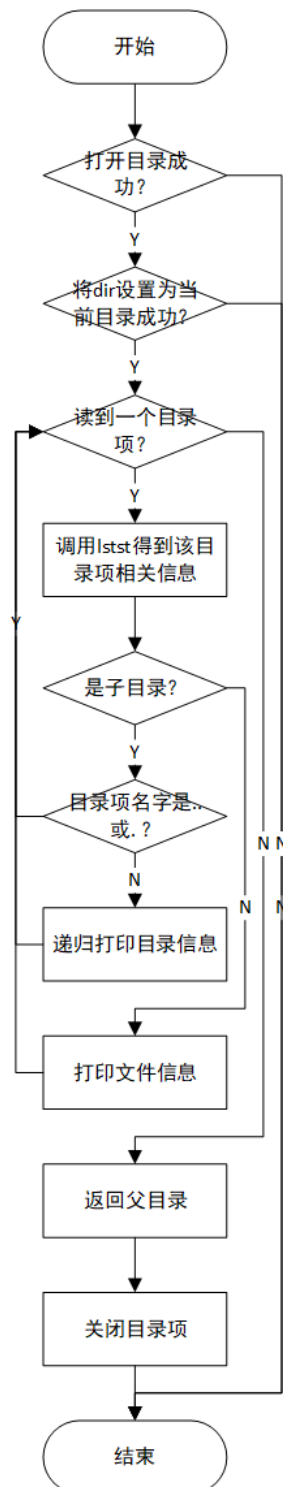


图 4-1 递归打印目录信息函数流程图

4.4 实验调试

4.4.1 实验步骤

- 1、仔细阅读实验 PPT，熟悉所要用到的 API 函数。
- 2、编写打印信息模块，根据 stat 结构体信息获取相应信息。
- 3、编写递归打印目录信息模块，参考 PPT 上所给模板思考清楚程序流程。
- 4、编写 main 函数，进行测试、调试。

4.4.2 实验调试及心得

- 1、运行结果

运行结果如下图所示：

```
may@may-VirtualBox:/Share/lab4$ ./main
0 directory rwxrwxrwx 1 root root 0 Fri Jan  4 14:03:24 2019 1
1 directory rwxrwxrwx 1 root root 0 Fri Jan  4 14:03:28 2019 2
2 directory rwxrwxrwx 1 root root 0 Fri Jan  4 14:03:31 2019 3
3 directory rwxrwxrwx 1 root root 0 Fri Jan  4 14:03:30 2019 4
1 regular file rwxrwxrwx 1 root root 0 Sat Dec 29 17:18:23 2018 test.txt
0 regular file rwxrwxrwx 1 root root 13272 Fri Jan  4 13:53:57 2019 main
0 regular file rwxrwxrwx 1 root root 3807 Fri Jan  4 14:00:37 2019 main.c
may@may-VirtualBox:/Share/lab4$ ls -l
总用量 20
drwxrwxrwx 1 root root    0 1月  4 14:03
-rwxrwxrwx 1 root root 13272 1月  4 13:53 main
-rwxrwxrwx 1 root root  3807 1月  4 14:00 main.c
may@may-VirtualBox:/Share/lab4$
```

图 4-2 测试图

原目录如下：



图 4-3 原目录

由上图看出，程序测试通过。能够递归打印目录信息。

- 2、实验调试

本次实验比较简单，没遇到太大的错误。

开始编写程序的时候，没有考虑到时间格式的问题，直接将 `stat` 结构体里面的时间信息输出了，发现获取的时间格式不正确，后来查阅资料发现可以用 `ctime` 函数转换时间格式，再次编译，发现测试通过。

3、实验心得

本次实验比较简单，实现了具有递归打印目录功能的类似 `ls` 的程序，初步了解了 `linux` 文件系统目录结构，了解到 `linux` 采用 `stat` 结构体几乎保存所有的文件状态信息，同时 `linux` 还提供文件属性接口以及目录结构接口，方便获取文件信息。经过这次实验，掌握了文件和目录的程序设计方法，了解到了递归打印目录的工作流程，能够自己实现一个小小的命令行操作了。

附录 实验代码

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

```
char cur_dir[50];
```

```
void printinfo(struct stat statbuf) {
```

```
    //文件类型
```

//S_IFMT 是一个掩码，它的值是 017000（八进制），用来过滤出前四位表示的文件类型

```
    switch (statbuf.st_mode & S_IFMT) {
```

```
        case S_IFSOCK:
```

```
            printf("socket ");
```

```
            break;
```

```
        case S_IFLNK:
```

```
            printf("symbolic link ");
```

```
            break;
```

```
        case S_IFREG:
```

```
            printf("regular file ");
```

```
            break;
```

```
        case S_IFBLK:
```

```
            printf("block device ");
```

```
            break;
```

```
        case S_IFDIR:
```

```
            printf("directory ");
```

```
            break;
```

```

        case S_IFCHR:
            printf("character device ");
            break;
        case S_IFIFO:
            printf("FIFO<pipe> ");
            break;
        default:
            break;
    }

```

//权限信息

//个人

```

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("-");
}

```

//同组

```

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("-");
    }

    //他人
    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("-");
    }

    //硬连接的数目
    printf(" %ld ",(long) statbuf.st_nlink);

    //文件所有者名字，组名字
    struct passwd *uid = getpwuid(statbuf.st_uid);
    struct group *gid = getgrgid(statbuf.st_gid);
    printf("%s %s ", uid->pw_name, gid->gr_name);

    //文件大小
    printf("%ld ", statbuf.st_size);

    //文件最后修改时间
    char *time = ctime(&statbuf.st_mtime);
    time[strlen(time) - 1]='\0';
    printf("%s ", time);
}

void printdir(char *dir, int depth) {
    DIR *dp;
    struct dirent *entry;
    struct stat statbuf;

    //打开目录
    if ((dp = opendir(dir)) == NULL) {
        perror("Fail to open dir.\n");
        return;
    }

```

```

//将 dir 设置为当前目录
if (chdir(dir) != 0) {
    perror("Fail to change dir.\n");
    return;
}

while ((entry = readdir(dp)) != NULL) {
    lstat(entry->d_name, &statbuf);
    if ((statbuf.st_mode & S_IFMT) == S_IFDIR)
    {
        if(strcmp(entry->d_name, "..") == 0 || strcmp(entry->d_name, ".") ==
0)
        {
            continue;
        }
        else
        {
            printf("%d ", depth);
            printinfo(statbuf);
            printf("%s\n", entry->d_name);
            printdir(entry->d_name, depth + 1);
        }
    }
    else
    {
        printf("%d ", depth);
        printinfo(statbuf);
        printf("%s\n", entry->d_name);
    }
}

//返回父目录
if (chdir("..") < 0)
{

```

```

        perror("Fail to change dir.\n");
        return;
    }

    //关闭目录项
    closedir(dp);
}

int main() {
    if (getcwd(cur_dir, 50) == NULL)
    {
        perror("Fail to get current work directory.\n");
        exit(-1);
    }

    printdir(cur_dir, 0);

    return 0;
}

```