华中科技大学

操作系统原理实验报告

姓　　名：　　刘逸帆

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　校交 1601 班

学　　号：　　U201610504

指导教师：　　邵志远

分数	
教师签名	

2019 年　1 月　4 日

目　录

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 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!

1.3 实验设计

1.3.1 开发环境

操作系统：Ubuntu 16.04 LTS

机器内存：8GB

1.3.2 实验设计

分别利用代码设计 main 主进程、用于发送数据的 Process1 子进程、用于接收数据的 Process2 子进程。

main 主进程中需要实现以下操作：创建管道、为键盘中断信号 SIGINT 绑定中断函数、创建两个子进程 sub1 和 sub2、等待两个子进程结束后关闭管道并输出提示信息。主进程流程图如图 1.1 所示。

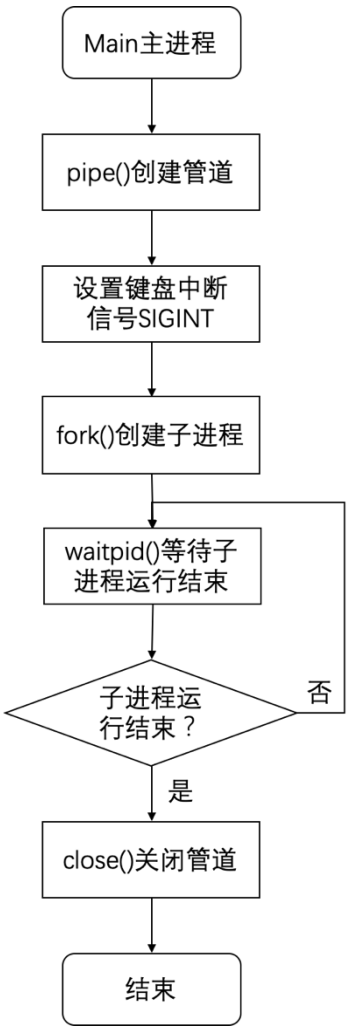

图 1.1 主进程流程图

在实现过程中，调用系统函数 pipe()创建管道；调用系统函数 signal()设置中断信号 SIGINT 与中断处理函数 my_func 绑定，在捕获键盘中断时通过中断处理函数分别向两个子进程发送 SIGUSR1 和 SIGUSR2 用户自定义中断；调用系统函数 fork()创建子进程 sub1 和 sub2；调用系统函数 waitpid()等待进程结束；调用系统函数 close()关闭管道。

Process1 子进程中需要实现以下操作：解除与键盘中断 SIGINT 的绑定、绑定用户自定中断信号 SIGUSR1、每隔一秒通过管道向子进程 2 发送数据。子进

程 1 流程图如图 1.2 所示。

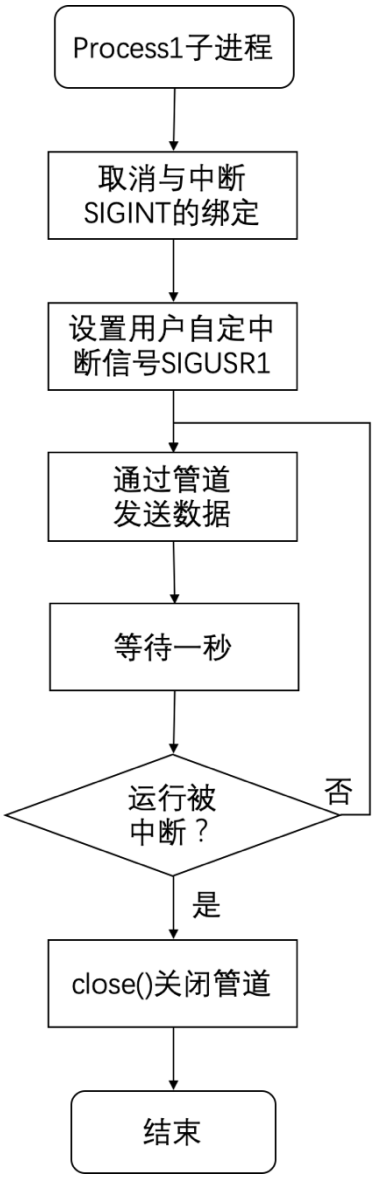

图 1.2 Process1 子进程流程图

在实现过程中，通过 signal(SIGINT,SIG_IGN)取消与键盘中断的绑定，并通过调用系统函数 signal()设置中断信号 SIGUSR1 与中断处理函数 my_sig1_func 绑定，在捕获到用户自定中断 SIGUSR1 时关闭管道、输出提示信息并通过 exit(0)结束子进程 1；通过 while(1)循环与系统函数 write()、sleep(1)实现每隔一秒通过管道向子进程 2 发送数据。

Process2 子进程中需要实现以下操作：解除与键盘中断 SIGINT 的绑定、绑定用户自定中断信号 SIGUSR2、从管道中读取子进程 1 发送的数据并输出显示数据内容。子进程 2 流程图如图 1.3 所示。

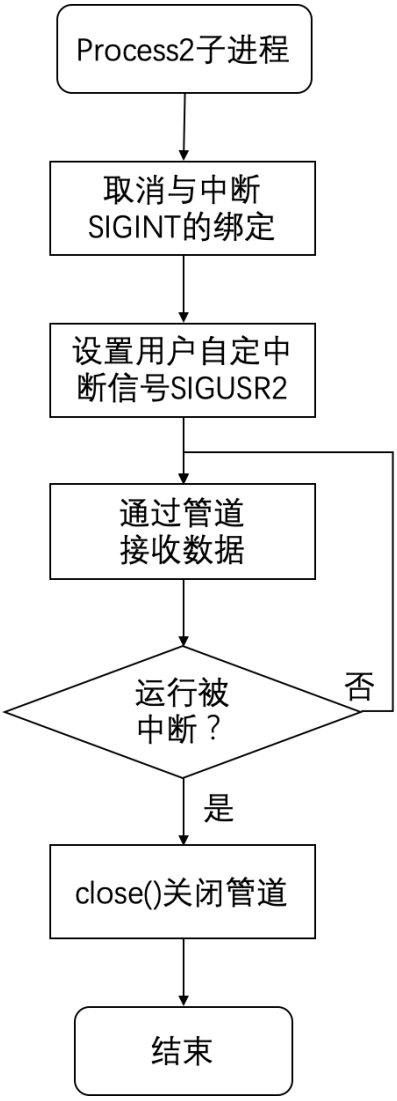

图 1.3 Process2 子进程流程图

在实现过程中，通过 signal(SIGINT,SIG_IGN)取消与键盘中断的绑定，并通过调用系统函数 signal()设置中断信号 SIGUSR2 与中断处理函数 my_sig2_func 绑定，在捕获到用户自定中断 SIGUSR2 时关闭管道、输出提示信息并通过 exit(0)结束子进程 2；通过 while(1)循环与系统函数 read()实现持续通过管道读取子进程 1 发送的数据。

1.4 实验调试

1.4.1 实验步骤

1. 通过 gcc 编译程序。
2. 执行程序，观察两个子进程之间数据通信的结果。
3. 产生键盘中断，观察提示信息以及程序是否能正常退出。

1.4.2 实验调试及心得

运行编译后的程序，两个子进程之间数据通信的结果如图 1.4 所示。

```
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ gcc -o 1 ./1.c
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ ./1
I send you 1 times
I receive 1 times
I send you 2 times
I receive 2 times
I send you 3 times
I receive 3 times
I send you 4 times
I receive 4 times
I send you 5 times
I receive 5 times
```

图 1.4 子进程间通信效果

产生键盘中断，程序的反应如图 1.5 所示。

```
I send you 28 times
I receive 28 times
I send you 29 times
I receive 29 times
I send you 30 times
I receive 30 times
I send you 31 times
I receive 31 times
^C
Child Process 2 is Killed by Parent!
Child Process 1 is Killed by Parent!
Parent Process is Killed!
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$
```

图 1.5 键盘中断被处理

验证了设计的程序达到了实验要求，

心得体会：

在本次实验中主要利用管道实现了进程间通信，通过代码实践，对 LINUX 系统下进程的创建与进程之间数据的传递有了比较真切的体会，同时对于系统处理中断的过程也有了更清晰的理解。在实现实验要求中的功能时需要调用一些系统 API，编写代码的过程中主要使用 ubuntu 系统终端下的 man 指令查阅手册来获得各个 API 的调用方法，这样的方式锻炼了我搜索信息的能力。

附录 实验代码

源代码 1.c 的内容如下：

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

```
int pipefd[2];

pid_t sub1;
pid_t sub2;

void my_func(int sig_no){
	if(sig_no == SIGINT){
//break in main
		//send sign to sub1 and sub2
		kill(sub1,SIGUSR1);
		kill(sub2,SIGUSR2);
	}
}

void my_sig1_func(int sig_no){
	if(sig_no == SIGUSR1){
		//close pipe
		close(pipefd[0]);
		close(pipefd[1]);
		//printf sign
		printf("\nChild Process %d is Killed by Parent!",1);
		exit(0);
	}
}

void my_sig2_func(int sig_no){
	if(sig_no == SIGUSR2){
		//close pipe
		close(pipefd[0]);
		close(pipefd[1]);
		//printf sign
		printf("\nChild Process %d is Killed by Parent!",2);
		exit(0);
	}
}

int main(void)
{
	pipe(pipefd);//create unnamed pipe
			//pipefd[0] is only used for read
			//pipefd[1] is only used for write

	//bind SIGINT with my_func
```

```
    signal(SIGINT, my_func);

    //create sub1
    sub1 = fork();
    if(sub1 == 0){//sub1
        signal(SIGINT,SIG_IGN);//ignore SIGINT
        signal(SIGUSR1,my_sig1_func);//bind SIGUSR1 with my_sig1_func
        int count=1;
        while(1){
            write(pipefd[1],&count,sizeof(int));
            printf("I send you %d times\n", count++);
            sleep(1);
        }
    }
    else{//main or sub2
        //create sub2
        sub2 = fork();
        if(sub2 == 0){//sub2
            signal(SIGINT,SIG_IGN);//ignore SIGINT
            signal(SIGUSR2,my_sig2_func);//bind SIGUSR2 with my_sig2_func
            while(1){
                int buf=0;
                read(pipefd[0],&buf,sizeof(int));
                printf("I receive %d times\n",buf);
            }
        }
        else{//main
            //waiting sub function unitl they exit
            int status;
            waitpid(sub1,&status,0);
            waitpid(sub2,&status,0);
            //close pipe
            close(pipefd[0]);
            close(pipefd[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

操作系统：Ubuntu 16.04 LTS

机器内存：8GB

2.3.2 实验设计

利用代码在主进程中创建信号灯，创建两个线程并通过信号灯实现对共享变量 a 的互斥访问。

在 main 主进程中需要实现以下操作：创建信号灯集、为信号灯赋初值、创建两个线程、等待两个线程运行结束并删除信号灯集。主进程流程图如图 2.1 所示。

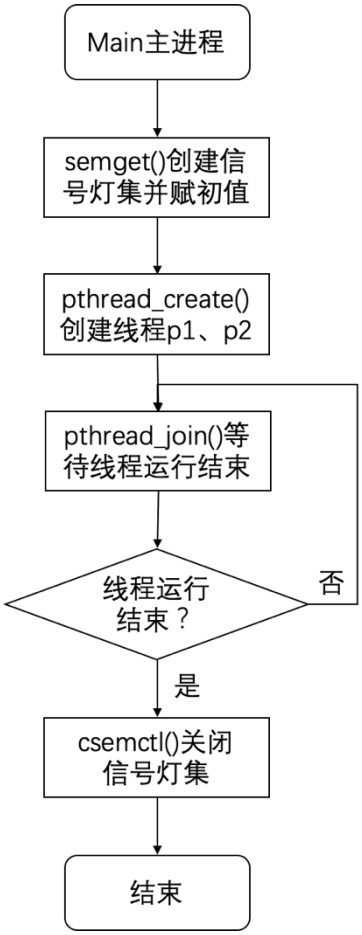

图 2.1 主进程流程图

在实现过程中，通过调用系统函数 semget()创建包含两个信号灯的信号灯集；通过调用系统函数 semctl()为两个信号灯分别赋初值 0 和 1；通过调用系统函数 pthread_create()创建子线程 p1 和 p2，分别用于执行函数 subp1()和函数 subp2()内的代码；通过调用系统函数 pthread_join()等待两个线程运行结束；通过调用系统函数 semctl()关闭信号灯集。

在 p1 线程中需要实现以下操作：通过 while 循环对共享变量 a 累加 1～100 这 100 个数字，每次累加前进行 P 操作抢夺 0 号信号灯，每次累加后进行 V 操作释放 1 号信号灯；累加完成后结束线程。P1 线程流程图如图 2.2 所示。

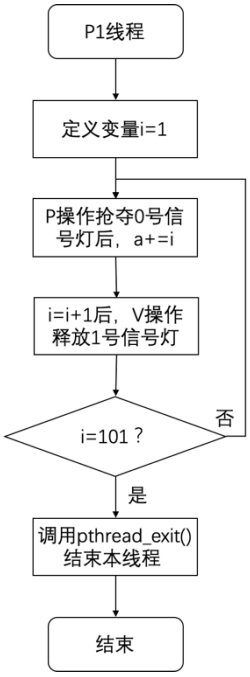

图 2.2 线程 p1 流程图

在实现过程中，PV 操作被分别封装为 P、V 两个函数，P、V 函数内通过调用系统函数 semop()实现对信号灯的操作；在累加 100 次后通过调用系统函数 pthread_exit(0)退出线程。

在 p2 线程中需要实现以下操作：通过 while 循环输出共享变量 a，每次输出前进行 P 操作抢夺 1 号信号灯，每次累输出后进行 V 操作释放 0 号信号灯；输出 100 次后结束线程。P2 线程流程图如图 2.3 所示。

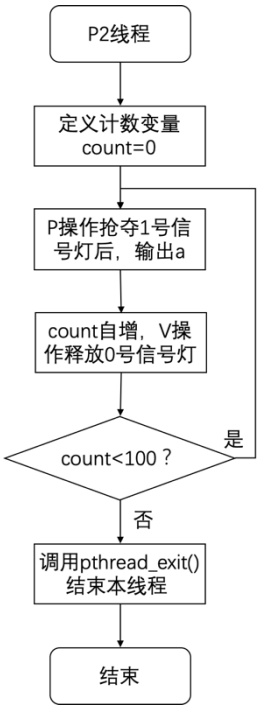

图 2.3 线程 p2 流程图

在实现过程中，PV 操作被分别封装为 P、V 两个函数，P、V 函数内通过调用系统函数 semop()实现对信号灯的操作；在累计输出 100 次后通过调用系统函数 pthread_exit(0)退出线程。

选做内容：

通过 PV 操作模拟生产者消费者问题：飞机售票，设总票数为 100，设置 4 个子线程代表 4 个售票窗口，同时进行售票。

在 main 主进程中实现以下操作：调用系统函数 semget()创建信号灯集、调用系统函数 semctl()为 0 号信号灯集赋初值 1、调用系统函数 pthread_create()创建四个子线程、调用系统函数 pthread_join()等待四个子线程运行结束、调用系统函数 semctl()删除信号灯。

四个售票线程中实现以下操作：P 操作抢夺 0 号信号灯，若有票则进行购买后 V 操作释放 0 号信号灯；否则 V 操作释放 0 号信号灯并调用系统函数 pthread_exit()结束线程。

2.4 实验调试

2.4.1 实验步骤

1. 通过 gcc 编译程序 2.c。
2. 执行程序，观察两个线程的工作结果。
3. 等待累加结束，观察线程和程序的退出情况。

选做内容：

1. 通过 gcc 编译程序 2_extend.c。
2. 执行程序，观察四个线程的售票结果。
3. 等待售票结束，观察线程和程序的退出情况。

2.4.2 实验调试及心得

使用 gcc 编译程序时，由于 pthread 不属于 linux 系统下的默认库，需要通过在命令行中添加参数-pthread 的方式对代码进行编译。

运行编译后的程序，观察累加和结果输出过程如图 2.4 所示。

```
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ gcc -o 2 ./2.c -pthread
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ ./2
a: 1
a: 3
a: 6
a: 10
a: 15
a: 21
a: 28
a: 36
a: 45
a: 55
a: 66
a: 78
a: 91
a: 105
a: 120
a: 136
```

图 2.4 共享变量 a 的累加过程

等待程序累加结束，程序退出时的提示信息如图 2.5 所示。

```
a: 4656
a: 4753
a: 4851
a: 4950
a: 5050
p1 terminated
p2 terminated
sem has been closed
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$
```

图 2.5 共享变量 a 累加 100 次结束

可知累加结果正确且线程均正常退出，验证了实验设计的正确性。

测试选做内容：

运行编译后的程序，观察售票过程如图 2.6 所示。

```
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ gcc -o 2_extend ./2_extend.c -pthread
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ ./2_extend
ticket 1 has been sold to p2
ticket 2 has been sold to p1
ticket 3 has been sold to p2
ticket 4 has been sold to p1
ticket 5 has been sold to p3
ticket 6 has been sold to p4
ticket 7 has been sold to p2
ticket 8 has been sold to p1
ticket 9 has been sold to p3
ticket 10 has been sold to p4
ticket 11 has been sold to p2
ticket 12 has been sold to p1
ticket 13 has been sold to p3
ticket 14 has been sold to p4
ticket 15 has been sold to p2
ticket 16 has been sold to p1
```

图 2.6 四个售票线程的工作过程

等待售票结束后，程序退出时的提示信息如图 2.7 所示。

```
ticket 94 has been sold to p4
ticket 95 has been sold to p2
ticket 96 has been sold to p1
ticket 97 has been sold to p3
ticket 98 has been sold to p4
ticket 99 has been sold to p2
ticket 100 has been sold to p1
p1 terminated
p2 terminated
p3 terminated
p4 terminated
sem has been closed
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$
```

图 2.7 售票完毕

可知四个售票线程工作正常且均能正常退出。验证了实验设计的正确性。

心得体会：

本次实验主要通过信号灯与公有变量实现了线程之间的同步与通信，对比实验一中的进程实验有很多的不同之处。线程相较于进程，从主进程处继承的内容会有很大不同：子进程只对父进程的数据域进行拷贝，在子进程中修改共有变量的值不会影响到父进程或其他子进程中的变量；对于线程而言，多线程共享同一个共有变量，在多个线程同时对这个变量进行读写操作时容易产生不可预料的后果，因此需要通过对信号灯的 PV 操作来限制临界区的读写；对于信号灯而言，子进程之间将其共享，从而能够控制多个进程同步读写共享缓冲区。

附录 实验代码

源代码 2.c 的内容：

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

// function of P、V opreations
void P(int semid,int index);
void V(int semid,int index);

// define semaphore and threads
int semid;
pthread_t p1,p2;

// define sub process
void *subp1();
void *subp2();

// define shared sum
int a=0;

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;
	sem.sem_op = -1;
	sem.sem_flg = SEM_UNDO;
	if(semop(semid,&sem,1)==-1)printf("P error\n");
	return;
}

void V(int semid,int index)
{
	struct sembuf sem;
	sem.sem_num = index;
	sem.sem_op =    1;
	sem.sem_flg = 0;
	if(semop(semid,&sem,1)==-1)printf("V error\n");
	return;
}

// subp1: caculate the sum from 1 to 100
void *subp1()
{
	int i;
	for(i=1;i<=100;i++){
		P(semid,0);
		a+=i;
		V(semid,1);
	}
	pthread_exit(0);
}

// subp2: printf the sum
void *subp2()
{
	int count=0;
	while(count<100){
		P(semid,1);
		printf("a: %d\n", a);
		count+=1;
		// sleep(1);
		V(semid,0);
```

```
    }
    pthread_exit(0);
}

int main(void)
{
    // 创建信号灯  semget(key_t key, int nsems, int semflg);
    semid = semget(IPC_PRIVATE, 2, IPC_CREAT|0600);
    if(semid==-1){
        printf("semget error\n");
        return -1;
    }

    // 信号灯赋初值；
    union semun arg_0;
    union semun arg_1;
    arg_0.val=0;
    arg_1.val=1;
    semctl(semid,0,SETVAL,arg_1);
    semctl(semid,1,SETVAL,arg_0);

    // 创建两个线程  pthread_create(pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine)(void *), void *arg);
    pthread_create(&p1,NULL,subp1,NULL);
    pthread_create(&p2,NULL,subp2,NULL);

    // 等待两个线程运行结束  pthread_join(pthread_t thread, void **value_ptr);
    if(pthread_join(p1,NULL)==0)printf("p1 terminated\n");
    if(pthread_join(p2,NULL)==0)printf("p2 terminated\n");

    // 删除信号灯；
    if(semctl(semid,0,IPC_RMID)!=-1)printf("sem has been closed\n");

    return 0;
}
```

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；
3、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

操作系统：Ubuntu 16.04 LTS
机器内存：8GB

3.3.2 实验设计

利用代码在主进程中创建共享内存单循环链表，构造环形缓冲。创建两个进程 readbuf、writebuf 分别对缓冲区进行读和写，从而实现文件的复制。

在主进程中，需要实现以下操作：为缓行缓冲链表定义 buffer_node 节点，包含大小为 BUFFERSIZE 的数据缓冲区与一个 next 指针；调用系统函数 shmget() 从系统内存申请 BLOCKSIZE 个大小为 sizeof(buffer_node)的共享内存组，在每块内存申请后调用系统函数 shmat()将内存区域映射为 buffer_node 节点，将

BLOCKSIZE 个节点的 next 指针相连构成环形缓冲区；调用系统函数 semget()创建包含 0 号、1 号两个信号灯信号灯集；因为信号灯所需要的初值即共享缓冲区的个数，调用系统函数 semctl()分别对 0 号信号灯赋值 BUFFERSIZE、对 1 号信号灯赋值 0；调用系统函数 fork()创建 subp_readbuf、subp_writebuf 两个子进程；调用系统函数 waitpid()等待两个子进程结束；调用系统函数 semctl()和 shmctl()分别删除信号灯集和共享内存组；调用系统函数 semdt()解除共享内存的映射。主进程的流程图如图 3.1 所示。

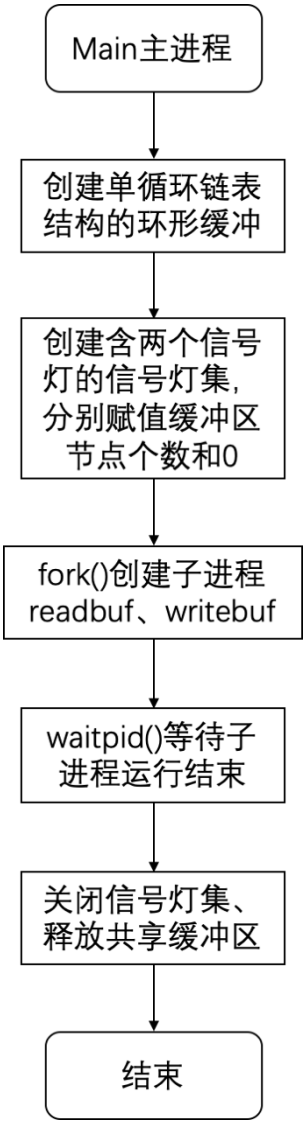

图 3.1 主进程流程图

在用于写缓冲区的子进程 writebuf 中，需要实现以下操作：调用系统函数 open()打开待复制的文件；调用实验二中的 P 函数对 0 号信号灯进行 P 操作，抢夺到信号灯后调用系统函数 read 从文件中读出 BUFFERSIZE 大小的数据并写入缓冲区节点中，返回成功写入的数据字节数，若写入字节与最大可写入字节不等，判断为写入了最后一段数据，为这个节点置结尾标志为最后写入的字节数并结束进程，调用实验二中的 V 函数对 1 号信号灯进行 V 操作。子进程 writebuf 流程

图如图 3.2 所示。

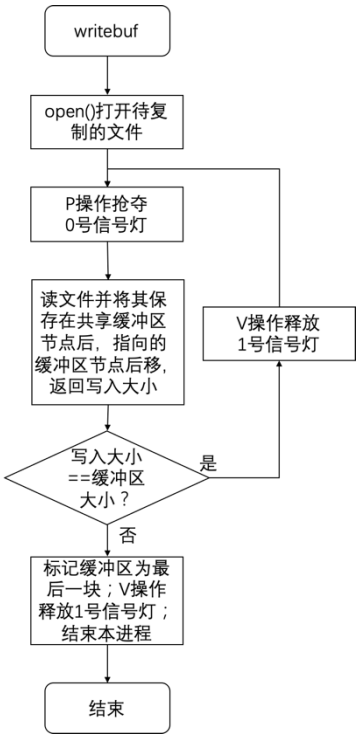

图 3.2 子进程 writebuf 流程图

在用语读缓冲区的子进程 readbuf 中，需要实现以下操作：调用系统函数 open()新建复制后将得到的文件；调用实验二中的 P 函数对 1 号信号灯进行 P 操作，抢夺到信号灯后判断是否读到最后一个缓冲区节点，若是则调用系统函数 write 从缓冲区中读出对应大小的数据并写入新生成的文件中，退出子进程，否则若不是最后一个缓冲节点则调用系统函数 write 从缓冲区中读出 BUFFERSIZE 大小的数据并写入新生成的文件中。子进程 readbuf 的流程图如图 3.3 所示。

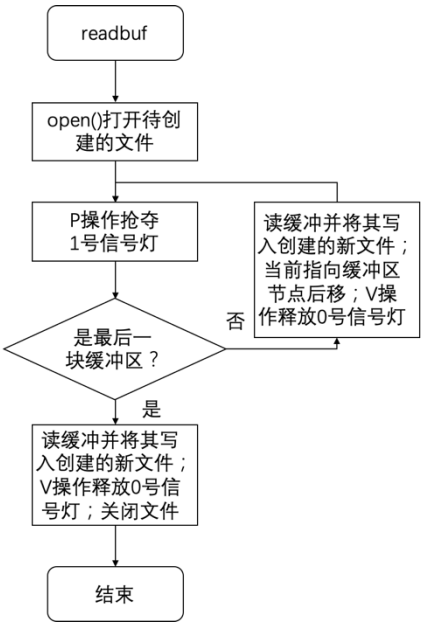

图 3.3 子进程 readbuf 流程图

选做内容：

保持缓冲区个数与大小不变，信号灯即可用缓冲区的个数减半，记录程序执行时间。

保持缓冲区大小不变，将缓冲区的个数减半，记录程序执行时间。

为 readbuf 子进程添加 sleep 函数降低进程的运行速度，记录程序执行时间。

对比进行修改后的三个程序执行时间与原来的执行时间，推导结论。

3.4 实验调试

3.4.1 实验步骤

1. 通过 gcc 编译程序 3.c。
2. 执行程序，观察两个子进程分别对缓冲区进行读写的结果。
3. 等待文件拷贝完成，观察提示信息以及程序是否能正常退出。
4. 使用 diff 指令测试拷贝后的文件与原文件是否相同。

选做内容：

1. 通过 gcc 编译程序 3_less_signal.c。
2. 通过 gcc 编译程序 3_less_block.c。
3. 通过 gcc 编译程序 3_slower_process.c。
4. 命令行中使用 time 指令执行程序，观察记录每隔程序的执行时间。
5. 比较各程序的执行时间并归纳结论。

3.4.2 实验调试及心得

运行编译后的程序，观察两个子进程读写缓冲区的过程，如图 3.4 所示。

```
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ ./3
open file: /home/parallels/test.mkv
the 1 time(write into shm) size: 1024
the 2 time(write into shm) size: 1024
the 3 time(write into shm) size: 1024
open file: /home/parallels/new.mkv
the 4 time(write into shm) size: 1024
the 5 time(write into shm) size: 1024
the 6 time(write into shm) size: 1024
the 7 time(write into shm) size: 1024
the 8 time(write into shm) size: 1024
the 9 time(write into shm) size: 1024
the 10 time(write into shm) size: 1024
the 1 time(read from shm): 1024
the 2 time(read from shm): 1024
the 11 time(write into shm) size: 1024
the 12 time(write into shm) size: 1024
the 3 time(read from shm): 1024
the 4 time(read from shm): 1024
```

图 3.4 子进程读写环形缓冲过程

等待程序执行结束后，文件拷贝完成，通过 diff 指令判断新文件与旧文件内容是否相同，比较结果如图 3.5 所示。

```
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ diff /home/parallels/test.mkv /home/parallels/new.mkv
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$
```

图 3.5 diff 指令判断文件拷贝是否成功

比较后两个文件相同，文件拷贝成功，验证了程序设计的正确性。

选做内容：

源程序的运行时间如图 3.6 所示。

```
the 665296 time(read from shm): 1024
the 665297 time(read from shm): 1024
the 665298 time(read from shm): 1024
the last time(read from shm): 313
sem has been closed
shm has been closed

real    0m7.395s
user    0m1.104s
sys     0m8.896s
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$
```

图 3.6 源程序 3.c 运行时间

信号灯数量减半后程序的运行时间如图 3.7 所示。

```
the last time(write into shm) size: 313
the 665296 time(read from shm): 1024
the 665297 time(read from shm): 1024
the 665298 time(read from shm): 1024
the last time(read from shm): 313
sem has been closed
shm has been closed

real    0m7.813s
user    0m1.060s
sys     0m8.832s
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$
```

图 3.7 信号灯数量减半后的程序运行时间

共享内存块的个数减半后与信号灯减半效果相同，不做赘述。

共享内存块的大小减半后程序的运行时间如图 3.8 所示。

```
the last time(write into shm) size: 313
the 1330589 time(read from shm): 512
the 1330590 time(read from shm): 512
the 1330591 time(read from shm): 512
the 1330592 time(read from shm): 512
the 1330593 time(read from shm): 512
the 1330594 time(read from shm): 512
the 1330595 time(read from shm): 512
the 1330596 time(read from shm): 512
the last time(read from shm): 313
sem has been closed
shm has been closed

real    0m15.875s
user    0m2.444s
sys     0m18.448s
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$
```

图 3.8 共享内存块大小减半后的程序运行时间

某一个子进程添加 sleep 降低执行速度后程序的运行时间如图 3.9 所示。

```
the 665296 time(read from shm): 1024
the 665297 time(write into shm) size: 1024
the 665297 time(read from shm): 1024
the 665298 time(write into shm) size: 1024
the 665298 time(read from shm): 1024
the last time(write into shm) size: 313
the last time(read from shm): 313
sem has been closed
shm has been closed

real    1m16.747s
user    0m4.144s
sys     0m20.848s
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$
```

图 3.9 减慢一个子进程执行速度后的程序运行时间

比较四个程序的执行时间可以得出以下结论：

1. 在一定范围内，保证两个子进程的执行速度相近的前提下，给予数量更多的缓冲区能够提升拷贝程序的执行速度；

2. 在一定范围内，保证两个子进程的执行速度相近的前提下，给予更大体积的缓冲区能够提升拷贝程序的执行速度；

3. 在保证缓冲区数量体积相同的前提下，两个子进程的执行速度相近时，拷贝程序的执行速度更高；

4. 显然地，提升缓冲区数量达到某一临界值后，拷贝程序的执行速度不会再提升。此时缓冲区大小已经足够，磁盘的存取速度成为瓶颈；

5. 显然地，提升缓冲区大小达到某一临界值后，拷贝程序的执行速度不会再提升反而可能下降。此时缓冲区为匹配读写速度不一致的目的几乎无法实现。

心得体会：

在本次实验中需要利用两个子进程读写环形缓冲实现文件的抄写。在实验的过程中需要注意进程对比于线程，数据域将被拷贝而不是直接使用，为实现进程之间的通信只能通过信号灯的方式。在新建文件时需要注意调用系统函数 open 时权限参数的选择。另外在实现文件的拷贝时，本次设计的程序对最后一块缓冲的判断仍然存在缺陷：程序通过判断最后写入缓冲的数据大小与之前写入的大小不一致来认为写入了最后一段文件数据，若程序大小恰为缓冲区大小的整数倍则可能出现问题。但在实际测试中并没有成功取到这样大小的程序，应对相应情形的代码也没有得到验证。

在选做内容中使用了指令 time 来记录不同程序的运行时间并加以比较，通过分析比较结果对需要共享内存的进程之间的同步关系有了比较深刻的理解。

附录 实验代码

源程序 3.c 的内容：

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

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <fcntl.h>
// sem
#include <sys/sem.h>
// shm
#include <sys/shm.h>
// fork
#include <unistd.h>
// waitpid
#include <sys/wait.h>
// errno
#include <errno.h>

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) */
};

// define shared memory id
#define BLOCKSIZE 10
int shmid[BLOCKSIZE];
// define buffer
#define BUFFERSIZE 1024
struct buffer_node{
        char buffer[BUFFERSIZE];
        int my_eof;
        struct buffer_node *next;
};
struct buffer_node *head=NULL,*tail=NULL,*temp=NULL;
// define semaphore
int semid;
// define process 'readbuf' and 'writebuf'
pid_t readbuf;
pid_t writebuf;
// define function of P、V opreations
void P(int semid,int index);
void V(int semid,int index);
// define sub process
void subfunc_readbuf();
```

```
void subfunc_writebuf();

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;
}

// read from file and write into shared memory
void subp_writebuf()
{
	int fd;
	int read_size;
	int times=0;
	const char* filename="/home/parallels/test.mkv";
	struct buffer_node *temp=head;
	if ((fd=open(filename,O_RDONLY))==-1){
		printf("open file fail\n");
		return;
	}
	else
		printf("open file: %s\n",filename);
	while(1){
		P(semid,0);
		read_size=read(fd, temp->buffer, BUFFERSIZE);
		if(read_size!=BUFFERSIZE && read_size!=0){// make eof
			printf("the last time(write into shm) size: %d\n",read_size);
			temp->my_eof=read_size;
			close(fd);
			V(semid,1);
```

```
                return;
            }
            printf("the %d time(write into shm) size: %d\n",++times,read_size);
            temp=temp->next;
            V(semid,1);
        }
        return;
}

// read from shared memory and write into new file
void subp_readbuf()
{
        int fd;
        int write_size;
        int times=0;
        const char* filename="/home/parallels/new.mkv";
        struct buffer_node *temp=head;
        if
((fd=open(filename,O_WRONLY|O_CREAT,S_IRWXU|S_IXGRP|S_IROTH|S_IXOTH))==-1){
            printf("open file fail\n");
            return;
        }
        else
            printf("open file: %s\n",filename);
        while(1){
            P(semid,1);
            if(temp->my_eof!=0){// read eof
                write_size=write(fd, temp->buffer, temp->my_eof);
                printf("the last time(read from shm): %d\n",write_size);
                close(fd);
                V(semid,0);
                return ;
            }
            write_size=write(fd, temp->buffer, BUFFERSIZE);
            printf("the %d time(read from shm): %d\n",++times,write_size);
            temp=temp->next;
            V(semid,0);
        }
        return;
}


int main(void)
{
```

```
// 创建共享内存组  int shmget(key_t key, size_t size, int shmflg);
int i;
for(i=0;i<BLOCKSIZE;i++){
	// get shm
	shmid[i] = shmget(IPC_PRIVATE, sizeof(struct buffer_node), IPC_CREAT|0600);
	if(shmid[i]==-1){
		printf("shmget error: %s\n",strerror(errno));
		return -1;
	}

	// make shm list
	if(i==0){// head node
		// link shm
		head=shmat(shmid[i],NULL,SHM_RND);
		if(head==(void*)-1){
			printf("shmat error\n");
			return -1;
		}
		head->next=head;
		head->my_eof=0;
		tail=head;
	}
	else{// else nodes
		// link shm
		temp=shmat(shmid[i],NULL,SHM_RND);
		if(temp==(void*)-1){
			printf("shmat error\n");
			return -1;
		}
		temp->next=head;
		temp->my_eof=0;
		tail->next=temp;
		tail=temp;
	}
}

// 创建信号灯  semget(key_t key, int nsems, int semflg);
semid = semget(IPC_PRIVATE, 2, IPC_CREAT|0600);
if(semid==-1){
	printf("semget error\n");
	return -1;
}

// 信号灯赋初值；
```

```
    union semun arg_0,arg_1;
    arg_1.val=BLOCKSIZE;
    arg_0.val=0;
    semctl(semid,0,SETVAL,arg_1);
    semctl(semid,1,SETVAL,arg_0);

    // 创建两个进程 readbuf、writebuf; Readbuf 负责读、writebuf 负责写，如何定义？
    readbuf=fork();
    if(readbuf==0){//readbuf
        subp_readbuf();
    }
    else{//main or writebuf
        writebuf=fork();
        if(writebuf==0){//writebuf
            subp_writebuf();
        }
        else{//main
            // 等待两个进程运行结束；
            int status;
            waitpid(readbuf,&status,0);
            waitpid(writebuf,&status,0);
            // 删除信号灯；
            if(semctl(semid,0,IPC_RMID)!=-1)printf("sem has been closed\n");
            // 删除共享内存组；
            for(i=0;i<BLOCKSIZE;i++){
                if(shmctl(shmid[i],0,IPC_RMID)==-1)printf("shm %d closed error\n",i);
            }
            printf("shm has been closed\n");
            // unmap the shared memory
            while(head->next!=tail){
                temp=head;
                head=head->next;
                shmdt((void*)temp);
            }
            shmdt((void*)head);// now head->next==tail
            shmdt((void*)tail);
        }
    }

    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：

- 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。

4.3 实验设计

4.3.1 开发环境

操作系统：Ubuntu 16.04 LTS
机器内存：8GB

4.3.2 实验设计

本次实验通过对输入程序的目录名称进行解析，参考 ls -lR 递归输出该目录下的所有目录及文件的详细信息。

设计 printDir 递归函数，传入的参数为目录名和目录层次（深度），在 main 函数中对其进行调用。在 printDir 函数中需要实现以下功能：调用系统函数 opendir()打开传入参数中的目录名，若打开目录失败则对错误进行提示并返回-1；打开文件后调用系统函数 chdir()更改当前目录为新目录；通过 while 循环对当前目录进行处理：调用系统函数 readdir()依次读取目录项，若读取失败则对错误进行提示并返回-1，调用该系统函数 lstat()读取目录项的详细信息，通过 lstat 读取

到的详细信息判断后续进行何种操作，若该目录项对应一个目录则跳过’.’、’..’目录并对该目录输出类型、权限、层次、名称、大小、最后修改日期、用户组，随后递归调用 printDir()函数；若该目录项对应一个文件则直接对其输出文件类型、权限、层次、名称、大小、最后修改日期、用户组。while 循环执行结束后说明对当前目录的信息输出完毕，调用系统函数 closedir()关闭当前目录并通过 chdir()回到上层目录，printDir()的程序流程图如图 4.1 所示。

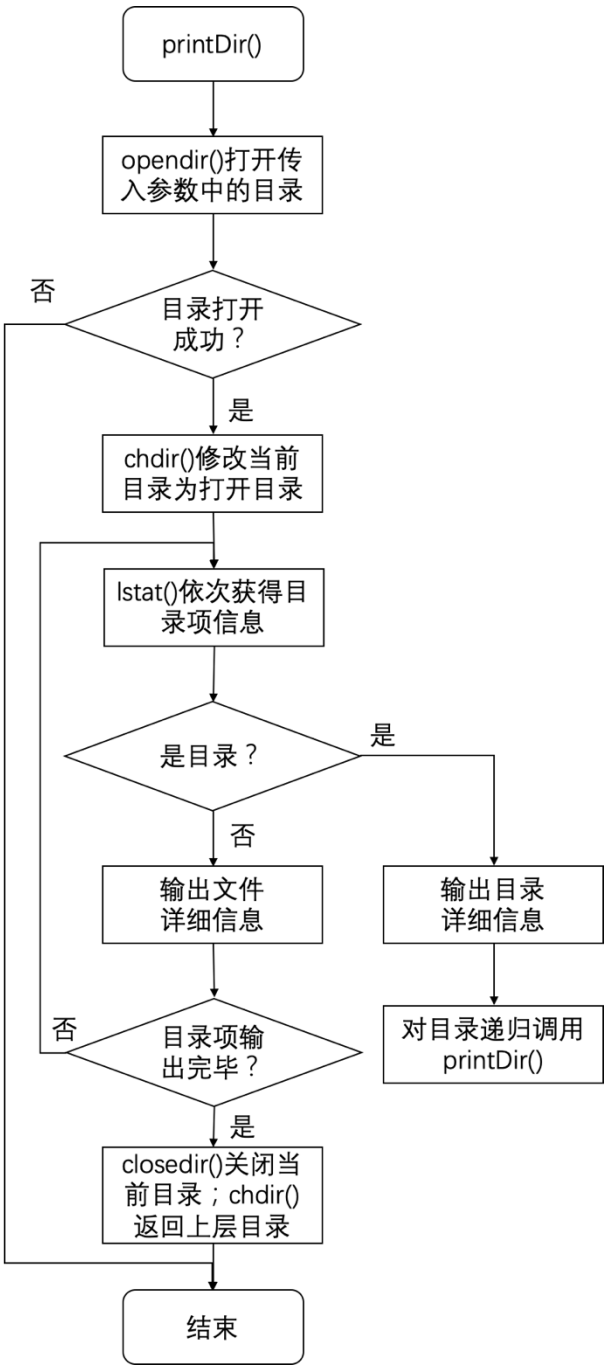

图 4.1 printDir()的流程图

其中在对目录和文件的详细信息进行输出时，对于文件类型和权限，通过 statbuf 中的 st_mode 成员进行判断；对于名称，直接输出目录项名称；对于文件

大小、最后修改时间、用户 ID、组 ID，分别通过 statbuf 中的 st_size、st_mtime、st_uid、st_gid 进行判断。

4.4 实验调试

4.4.1 实验步骤

1. 通过 gcc 编译程序 4.c。
2. 执行程序，输出当前目录详情，对比 ls -l 观察详情信息是否正确。
3. 执行程序，输出一个较复杂的工程目录详情，观察目录结构存在多层的情形下，程序在层次之间的跳转是否正确。

4.4.2 实验调试及心得

执行编译后的程序，输入当前目录，程序将递归输出当前目录下的各个目录项详细信息；再通过 ls -l 指令输出当前目录详细信息，对比两次输出。命令行测试界面如图 4.2 所示。

```
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ gcc -o 4 ./4.c
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ ./4
input directory name please:
.
type            layer name                      size(bytes)  last modification              userID          groupID
-rwxrwxr-x      1     1                         9416         Thu Jan  3 14:43:10 2019      parallels       parallels
-rw-r--r--      1     1.c                       2156         Thu Jan  3 14:43:08 2019      parallels       parallels
-rwxrwxr-x      1     2                         9256         Fri Jan  4 14:39:34 2019      parallels       parallels
-rw-r--r--      1     2.c                       2307         Fri Jan  4 14:39:23 2019      parallels       parallels
-rwxrwxr-x      1     2_extend                  13496        Fri Jan  4 14:43:09 2019      parallels       parallels
-rw-r--r--      1     2_extend.c                3188         Fri Jan  4 14:42:54 2019      parallels       parallels
-rwxrwxr-x      1     3                         13912        Mon Dec 24 18:20:23 2018      parallels       parallels
-rw-r--r--      1     3.c                       4968         Mon Dec 24 18:23:42 2018      parallels       parallels
-rwxrwxr-x      1     3_less_block              13920        Fri Dec 28 14:05:06 2018      parallels       parallels
-rw-r--r--      1     3_less_block.c            4970         Fri Dec 28 14:04:33 2018      parallels       parallels
-rwxrwxr-x      1     3_less_signal             13920        Fri Dec 28 14:05:49 2018      parallels       parallels
-rw-r--r--      1     3_less_signal.c           4973         Fri Dec 28 14:04:40 2018      parallels       parallels
-rwxrwxr-x      1     3_slower_process          13976        Fri Jan  4 16:36:34 2019      parallels       parallels
-rw-r--r--      1     3_slower_process.c        4988         Fri Jan  4 16:36:19 2019      parallels       parallels
-rwxrwxr-x      1     4                         13480        Fri Jan  4 16:49:46 2019      parallels       parallels
-rw-r--r--      1     4.c                       4163         Fri Dec 28 17:04:48 2018      parallels       parallels
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ ls -l
total 0
-rwxrwxr-x 1 parallels parallels  9416 Jan  3 14:43 1
-rw-r--r-- 1 parallels parallels  2156 Jan  3 14:43 1.c
-rwxrwxr-x 1 parallels parallels  9256 Jan  4 14:39 2
-rw-r--r-- 1 parallels parallels  2307 Jan  4 14:39 2.c
-rwxrwxr-x 1 parallels parallels 13496 Jan  4 14:43 2_extend
-rw-r--r-- 1 parallels parallels  3188 Jan  4 14:42 2_extend.c
-rwxrwxr-x 1 parallels parallels 13912 Dec 24 18:20 3
-rw-r--r-- 1 parallels parallels  4968 Dec 24 18:23 3.c
-rwxrwxr-x 1 parallels parallels 13920 Dec 28 14:05 3_less_block
-rw-r--r-- 1 parallels parallels  4970 Dec 28 14:04 3_less_block.c
-rwxrwxr-x 1 parallels parallels 13920 Dec 28 14:05 3_less_signal
-rw-r--r-- 1 parallels parallels  4973 Dec 28 14:04 3_less_signal.c
-rwxrwxr-x 1 parallels parallels 13976 Jan  4 16:36 3_slower_process
-rw-r--r-- 1 parallels parallels  4988 Jan  4 16:36 3_slower_process.c
-rwxrwxr-x 1 parallels parallels 13480 Jan  4 16:49 4
-rw-r--r-- 1 parallels parallels  4163 Dec 28 17:04 4.c
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$
```

图 4.2 对比程序的输出结果和 ls -l 输出结果

对比可知除时间格式不同外，两次输出详细信息的内容均相同，验证了程序功能的正确性。

再次运行程序，输入一个结构较复杂的目录，程序输出如图 4.3 所示。

```
parallels@parallels-vm:~/Desktop/Parallels Shared Folders/Home/to-ubunto/oslab_solution$ ./4
input directory name please:
/home/parallels/weibo_pro/

type            layer name                    size(bytes)  last modification            userID          groupID
-rwxr-xr-x      1     AUTHORS.rst             477          Sun Oct 14 19:41:12 2018    parallels       parallels
drwxrwxr-x      1     logs                    4096         Tue Nov 20 13:31:53 2018    parallels       parallels
-rw-rw-r--      2     weibo.log               12316        Tue Nov 20 16:03:07 2018    parallels       parallels
-rwxr-xr-x      1     Dockerfile              1401         Sun Oct 14 19:41:12 2018    parallels       parallels
-rwxr-xr-x      1     .gitignore              150          Sun Oct 14 19:41:12 2018    parallels       parallels
-rwxr-xr-x      1     .gitattributes          93           Sun Oct 14 19:41:12 2018    parallels       parallels
drwxr-xr-x      1     page_get                4096         Tue Nov 20 13:31:53 2018    parallels       parallels
-rwxr-xr-x      2     user.py                 6377         Sun Oct 14 19:41:12 2018    parallels       parallels
-rwxr-xr-x      2     basic.py                4020         Sun Oct 14 19:41:12 2018    parallels       parallels
-rwxr-xr-x      2     __init__.py             183          Sun Oct 14 19:41:12 2018    parallels       parallels
drwxrwxr-x      2     __pycache__             4096         Tue Nov 20 13:31:54 2018    parallels       parallels
-rw-r--r--      3     basic.cpython-35.pyc    3429         Tue Nov 20 13:31:53 2018    parallels       parallels
-rw-r--r--      3     status.cpython-35.pyc   781          Tue Nov 20 13:31:54 2018    parallels       parallels
-rw-r--r--      3     __init__.cpython-35.pyc 375          Tue Nov 20 13:31:53 2018    parallels       parallels
-rw-r--r--      3     user.cpython-35.pyc     5713         Tue Nov 20 13:31:54 2018    parallels       parallels
-rwxr-xr-x      2     status.py               573          Sun Oct 14 19:41:12 2018    parallels       parallels
drwxr-xr-x      1     first_task_execution    4096         Sun Oct 14 19:41:12 2018    parallels       parallels
-rwxr-xr-x      2     user_first.py           144          Sun Oct 14 19:41:12 2018    parallels       parallels
-rwxr-xr-x      2     dialogue_first.py       152          Sun Oct 14 19:41:12 2018    parallels       parallels
```

图 4.3 对多层目录的详情输出

测试结果中，目录项的层次一项由 1->2->1，说明程序能够正确递归输出多层目录的内容，在输出子目录详情后能够回到父目录继续进行输出，验证了程序设计的正确性。

心得体会：

本次实验需要模仿 ls -lR 功能实现对目录详细信息的递归输出，递归实现并不难，主要涉及到对比较多的系统 api 进行调用，需要熟练使用 man 指令或使用搜索引擎进行搜索。如在对目录项的权限进行获取时，需要将比较多的宏定义变量与 st_mode 成员进行按位与之类的操作，在对最后修改时间进行获取时需要使用到 ctime 之类的函数对指定格式的数据进行转换，在获得用户名和组名时需要使用 getpwuid 函数获取字符串形式的用户名和组名等。

总的来说本次实验比较简单，但若需要实现 ls -lR 指令对应的输出，还需要在本次实验代码的基础上添加队列存储机制，从而在递归的过程中划分不同目录。

附录 实验代码

源代码 4.c 的内容：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
// lstat here
#include <sys/stat.h>
// opendir here
#include <sys/types.h>
#include <dirent.h>
```

```
// time here
#include <time.h>
// get uid's name
#include <pwd.h>

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

	// open the dir
	dp=opendir(dir);
	if(dp==NULL){
		printf("opendir error\n");
		return -1;
	}

	// change currently working dir to *dir
	chdir(dir);
	while(1){
		entry=readdir(dp);
		if(entry==NULL)break;
		if(lstat(entry->d_name, &statbuf)==-1){
			printf("lstat error\n");
			return -1;
		}
		if (S_ISDIR(statbuf.st_mode)){// read a dir
			// do nothing if dir's name is '.' or ".."
			if(strcmp(entry->d_name,".")==0 || strcmp(entry->d_name,"..")==0);
			else{
				// type
				printf("d");
				// access
				if (S_IRUSR&statbuf.st_mode) printf("r");
				else printf("-");
				if (S_IWUSR&statbuf.st_mode) printf("w");
				else printf("-");
				if (S_IXUSR&statbuf.st_mode) printf("x");
				else printf("-");
				if (S_IRGRP&statbuf.st_mode) printf("r");
				else printf("-");
				if (S_IWGRP&statbuf.st_mode) printf("w");
				else printf("-");
				if (S_IXGRP&statbuf.st_mode) printf("x");
```

```
                else printf("-");
                if (S_IROTH&statbuf.st_mode) printf("r");
                else printf("-");
                if (S_IWOTH&statbuf.st_mode) printf("w");
                else printf("-");
                if (S_IXOTH&statbuf.st_mode) printf("x");
                else printf("-");

                struct passwd *pwd;
                // depth, name, size(bytes)
                printf("\t%-4d %-25s %-12ld", depth+1, entry->d_name,
statbuf.st_size);
                // time
                printf("\t%s",strtok(ctime(&statbuf.st_mtime),"\n"));
                // uid, gid -> name
                pwd = getpwuid(statbuf.st_uid);
                printf("\t%s",pwd->pw_name);
                pwd = getpwuid(statbuf.st_gid);
                printf("\t%s",pwd->pw_name);

                printf("\n");

                // into next direct
                printDir(entry->d_name, depth+1);
            }
        }
        else{// normal file
            // type
            printf("-");
            // access
            if (S_IRUSR&statbuf.st_mode) printf("r");
            else printf("-");
            if (S_IWUSR&statbuf.st_mode) printf("w");
            else printf("-");
            if (S_IXUSR&statbuf.st_mode) printf("x");
            else printf("-");
            if (S_IRGRP&statbuf.st_mode) printf("r");
            else printf("-");
            if (S_IWGRP&statbuf.st_mode) printf("w");
            else printf("-");
            if (S_IXGRP&statbuf.st_mode) printf("x");
            else printf("-");
            if (S_IROTH&statbuf.st_mode) printf("r");
            else printf("-");
```

```
            if (S_IWOTH&statbuf.st_mode) printf("w");
            else printf("-");
            if (S_IXOTH&statbuf.st_mode) printf("x");
            else printf("-");

            struct passwd *pwd;
            // depth, name, size(bytes)
            printf("\t%-4d    %-25s       %-12ld", depth+1, entry->d_name, statbuf.st_size);
            // time
            printf("\t%s",strtok(ctime(&statbuf.st_mtime),"\n"));
            // uid, gid -> name
            pwd = getpwuid(statbuf.st_uid);
            printf("\t%s",pwd->pw_name);
            pwd = getpwuid(statbuf.st_gid);
            printf("\t%s",pwd->pw_name);

            printf("\n");
        }
    }
    // read dir done
    chdir("..");
    closedir(dp);
    return 0;
}

int main(void){
    char dir_name[100];
    int depth=0;
    printf("input directory name please:\n");
    scanf("%s", dir_name);
    // table name
    printf("\ntype\t\tlayer name\t\t\t      size(bytes)    last modification\t\tuserID \t\tgroupID\n");
    printDir(dir_name, depth);
    return 0;
}
```