实验三 Linux 进程管理

17042127　　　　陶逸群

设计要求

- 实现一个模拟的 shell。
- 实现一个管道通信程序。
- 利用 Linux 消息队列通信机制实现两个线程之间的通信。
- 利用 Linux 的共享内存通信机制实现两个进程间的通信。
- 在模拟shell实验中，增加接收find和grep命令，并给结果显示。
- 在管道通信实验中，增加进程间有名管道通信 。
- 在共享内存实验中，增加共享内存的双向通信。

实验步骤

1. 实现一个模拟的 shell

   在第一个实验中，要实现一个模拟的 shell，共有 5 个命令，cmd1、 cmd2、cmd3、find 以及 grep。

   在具体实现中，前三个命令只打印命令的信息，后两个命令的功能 和 Linux 系统中的命令一致。

   我调用了 Linux 的 execv 系列系统原语实现模拟的 shell，每个命令单 独写一个程序，然后在主函数中通过 execv 系列函数调用相应的命令程 序并通过空格切分得到命令的参数，即可实现命令的相应功能。

   与此同时，我还使用 `getpwuid` 等函数模拟了shell中对用户名主机名的输出。

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <pwd.h>
#define NUM 10
//获取登陆用户名
void GetLoginname(){
    struct passwd* pass;
    pass = getpwuid(getuid());
    printf("[%s@ ",pass->pw_name);
}

//获取主机名
void GetHostname(){
    char name[128];
    gethostname(name,sizeof(name)-1);
    printf("%s ",name);;
}
void ChildProcess(int num,char * const myargv[],char ** environ){
```

```c
    pid_t pid=fork();
    if(pid<0)
        perror("fork error\n");
    else if(pid==0){
        switch(num){
            case 1:{
                execve("cmd1",myargv,environ);
                break;
            }
            case 2:{
                execve("cmd2",myargv,environ);
                break;
            }
            case 3:{
                execve("cmd3",myargv,environ);
                break;
            }
            case 4:{
                execve("/bin/find",myargv,environ);
                break;
            }
            case 5:{
                execve("/bin/grep",myargv,environ);
                break;
            }
            default:{
                printf("process will never go here");
                break;
            }
        }
        exit(1);
    }else{
        int status = 0;
        pid_t ret = waitpid(pid,&status,0);//阻塞父进程
        if(!(ret > 0 && WIFEXITED(status))){
            perror("waitpid");
        }
    }
}
//获取当前文件路径
void GetDir(){
    char pwd[128];
    getcwd(pwd,sizeof(pwd)-1);
    int len = strlen(pwd);
    char *p = pwd+len-1;
    while(*p != '/' && len--){
        p--;
    }
    p++;
    printf("%s] #",p);
}

int main(int argc,char * const argv[],char **environ)
{
    while(1) {
        GetLoginname();
        GetHostname();
        GetDir();
        char cmd[1024];
        char *myargv[NUM];
        fflush(stdout); //清空输出缓冲区
        int s = read(0,cmd,sizeof(cmd));
        cmd[s-1] = '\0';
        int i = 0;
        myargv[0] = strtok(cmd, " ");
```

```c
        while(myargv[i] !=NULL){
            i++;
            myargv[i] = strtok(NULL, " ");
        }
        myargv[i] = NULL;
        if (strcmp(myargv[0],"exit")==0) break;
        else if (strcmp(myargv[0],"cmd1")==0){
            ChildProcess(1,myargv,environ);
        }
        else if (strcmp(myargv[0],"cmd2")==0){
            ChildProcess(2,myargv,environ);
        }
        else if (strcmp(myargv[0],"cmd3")==0){
            ChildProcess(3,myargv,environ);
        }else if (strcmp(myargv[0],"find")==0){
            ChildProcess(4,myargv,environ);
        }else if (strcmp(myargv[0],"grep")==0){
            ChildProcess(5,myargv,environ);
        }
        else printf("Command not found\n");
    }
    return 0;
}

```

2. 实现管道通信
    - 无名管道通信

      首先由父进程通过 int pipe(int fd[2])函数创建一个无名管道，然后再 创建四个子进程：pid0、pid1 、 pid2和pid3，pid0设置成非阻塞写，不断写入字符，测试无名管道大小，父进程通过 `wait(0)` 等待0号进程写入完毕后非阻塞读，清空管道，pid1、pid2、pid3这三个子进程利用管道与父 进程进行通信。父进程通过等待 `read_mutex1,read_mutex2,read_mutex3` 三个信号量，等待三个子进程全部发送完消息（子进程结束）后接受子进程发送的消息， `write_mutex` 信号量控制pid1、pid2、pid3三个进程间写互斥。

```c
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <semaphore.h>

#define STR_MAX_SIZE 1024


int main(int argc, char **argv) {
    int pipefd[2], i = 0;
    int pid0,pid1,pid2,pid3;
    ssize_t n;
    char buf[1];
    char str[STR_MAX_SIZE];
    int count = 0;

    // 创建有名信号量，若不存在则创建，若存在则直接打开，默认值为0
    sem_t *write_mutex;
    sem_t *read_mutex1;
    sem_t *read_mutex2;
```

```c
    sem_t *read_mutex3;
    write_mutex = sem_open("pipe_w", O_CREAT | O_RDWR, 0666, 0);
    read_mutex1 = sem_open("pipe_r_1", O_CREAT | O_RDWR, 0666, 0);
    read_mutex2 = sem_open("pipe_r_2", O_CREAT | O_RDWR, 0666, 0);
    read_mutex3 = sem_open("pipe_r_3", O_CREAT | O_RDWR, 0666, 0);
    memset(buf, 0, 1);
    memset(str, 0, STR_MAX_SIZE);

    // 创建管道并检查操作是否成功
    if(pipe(pipefd) < 0){
        printf("创建无名管道失败");
        exit(-1);
    }


    // 创建0号子进程并检查操作是否成功
    // 利用非阻塞写测试管道大小
    pid0 = fork();
    if(pid0 < 0){
        printf("第一个子进程创建失败");
        exit(-1);
    }

    if (pid0 == 0) {
        count = 0;
        close(pipefd[0]);
        // 管道默认是阻塞写，通过`fcntl`设置成非阻塞写，在管道满无法继续写入时返回-EAGAIN，作为循环终止条件
        int flags = fcntl(pipefd[1], F_GETFL);
        fcntl(pipefd[1], F_SETFL, flags | O_NONBLOCK);
        n = 0;
        // 写入管道
        while (n!=-1) {
            n = write(pipefd[1], buf, 1);
            count++;
        }
        printf("space = %dKB\n", (count * 1) / 1024);
        exit(0);
    }



    // 创建第一个子进程并检查操作是否成功
    pid1 = fork();
    if(pid1 < 0){
        printf("第一个子进程创建失败");
        exit(-1);
    }

    if (pid1 == 0) {
        sem_wait(write_mutex);
        close(pipefd[0]);
        n = write(pipefd[1], "这是一号进程\n", 20);
        printf("进程一写入了 %ld字节\n", n);
        sem_post(write_mutex);
        sem_post(read_mutex1);
        exit(0);
    }

    // 创建第二个子进程并检查操作是否成功
    pid2 = fork();
    if(pid2 < 0){
        printf("第二个子进程创建失败");
        exit(-1);
    }
```

```c

    if (pid2 == 0) {
        sem_wait(write_mutex);
        close(pipefd[0]);
        n = write(pipefd[1],  "这是二号进程\n", 20);
        printf("进程二写入了 %ld字节\n", n);
        sem_post(write_mutex);
        sem_post(read_mutex2);
        exit(0);
    }

    // 创建第三个子进程并检查操作是否成功
    pid3 = fork();
    if(pid3 < 0){
        printf("第三个子进程创建失败");
        exit(-1);
    }

    if (pid3 == 0) {
        sem_wait(write_mutex);
        close(pipefd[0]);
        n = write(pipefd[1], "这是三号进程\n", 20);
        printf("进程三写入了 %ld字节\n", n);
        sem_post(write_mutex);
        sem_post(read_mutex3);
        exit(0);
    }
    // 等待0号子进程运行完成，父进程继续运行
    // 用非阻塞读，清空管道
    wait(0);
    close(pipefd[1]);
    int flags = fcntl(pipefd[0], F_GETFL);
    n = 0 ;
    count = 0;
    // 设置非阻塞性读，作为循环结束标志
    fcntl(pipefd[0], F_SETFL, flags | O_NONBLOCK);
    while (n!=-1) {
        n = read(pipefd[0], buf, 1);
        count++;
    }
    printf("空间大小为 %dKB \n", (count * 1) / 1024);
    sem_post(write_mutex);

    // 等待子进程一、二、三写入完毕
    sem_wait(read_mutex1);
    sem_wait(read_mutex2);
    sem_wait(read_mutex3);
    n = read(pipefd[0], str, STR_MAX_SIZE);
    printf("读取了%ldB \n", n);
    for (i = 0; i < n; i++) {
        printf("%c", str[i]);
    }

    sem_close(write_mutex);
    sem_close(read_mutex1);
    sem_close(read_mutex2);
    sem_close(read_mutex3);
    sem_unlink("pipe_w");
    sem_unlink("pipe_r_1");
    sem_unlink("pipe_r_2");
    sem_unlink("pipe_r_3");
    return 0;
}

```

- 有名管道

  首先由通过mkfifo()函数传入参数管道名以及权限创建管道， 然后通过 open()函数传入参数管道名和读写方式打开管道，即可实现有名管道的读写操作。

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <pthread.h>
void *func(void * fd)
{

    int wri = write(*(int*)fd, "this is Thread_write", 30);
    if(wri < 0)
    {
        printf("wirte fifo failed!\n");
    }
    close(*(int*)fd);
}
int main()
{
     if(mkfifo("fifo", 0666) < 0 ){
        if(errno != EEXIST){        //建立管道失败，并且最后管道不存在则退出
            printf("create FIFO falied!\n");
            return 0;
        }
    }
    int fd = open("fifo", O_WRONLY);
    if(fd < 0)
    {
        printf("open fifo failed!\n");
        return 0;
    }
    pthread_t tid = 1;
    pthread_create(&tid, NULL, func, &fd);
    pthread_join(tid, NULL);

    return 0;
}

```

```c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <pthread.h>
void *func(void *fd)
{

    char readbuf[1024];
    read( *(int*)fd, readbuf, 30);
    printf("this is Thread_read!\n");
    printf("Receive message: %s\n", readbuf);
    close(*(int*)fd);
}
int main()
```

```
{
    int fd;
    char buff[2048];
    if(mkfifo("fifo", 0666) < 0 ){
        if(errno != EEXIST){        //建立管道失败，并且最后管道不存在则退出
            printf("create FIFO falied!\n");
            return 0;
        }
    }
    fd = open("fifo", O_RDONLY);
    if(fd < 0)
    {
        printf("open FIFO falied!\n");
    }
    pthread_t tid = 0;
    pthread_create(&tid, NULL, func, &fd);
    pthread_join(tid, NULL);


    return 0;
}

```

3. 利用 Linux 消息队列通信机制实现两个线程之间的通信

首先创建信号并对信号量初始化，在该实验中我使用了四个信号量， 如下表所示：

信号量	作用	初始值
`sem_send`	sender1和sender2发送消息互斥	1
`sem_receive`	sender1，sender2和receiver接受消息同步	0
`sem_over1`	sender1结束	0
`sem_over2`	sender2结束	0

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <sys/msg.h>
#include <pthread.h>
#include <semaphore.h>

#define snd_1 1
#define snd_2 2
#define rcv_1 3
#define rcv_2 4
#define MAX_SIZE    1024
#define QUEUE_ID    22222

typedef struct msg {
    long message_type;
    char message[MAX_SIZE];
}msg;

sem_t sem_send,sem_receive,sem_over1,sem_over2;
```

```c
void *sender1() {
    msg buf;
    int mq;
    mq = msgget((key_t) QUEUE_ID,  IPC_CREAT|0666 );
    while (1) {
        sem_wait(&sem_send);
         printf("sender:\n");
        buf.message_type = snd_1;
        fflush(stdout);
        fgets(buf.message, BUFSIZ, stdin);
        if(strcmp(buf.message,"exit\n")==0){
            strcpy(buf.message,"end1\n");
            msgsnd(mq, (void *) &buf, MAX_SIZE,  0);
            sem_post(&sem_receive);
            break;
        }else{
            msgsnd(mq, (void *) &buf, MAX_SIZE,  0);
            sem_post(&sem_receive);
        }
    }
    sem_wait(&sem_over1);
    msgrcv(mq, (void *) &buf, MAX_SIZE, rcv_1, 0);
    printf("%s", "over1\n");
    printf("--------------------------------------------\n");
    sem_post(&sem_send);
    pthread_exit(NULL);
}


void *sender2() {
    msg buf;
    int mq;
    mq = msgget((key_t) QUEUE_ID,  IPC_CREAT|0666 );
    while (1) {
        sem_wait(&sem_send);
         printf("sender2:\n");
        buf.message_type = snd_2;
        fflush(stdout);
        fgets(buf.message, BUFSIZ, stdin);
        if(strcmp(buf.message,"exit\n")==0){
            strcpy(buf.message,"end2\n");
            msgsnd(mq, (void *) &buf, MAX_SIZE,  0);
            sem_post(&sem_receive);
            break;
        }else{
            msgsnd(mq, (void *) &buf, MAX_SIZE,  0);
            sem_post(&sem_receive);
        }
    }
    sem_wait(&sem_over2);
    msgrcv(mq, (void *) &buf, MAX_SIZE, rcv_2, 0);
    printf("%s","over2\n");
    printf("--------------------------------------------\n");
    sem_post(&sem_send);
    pthread_exit(NULL);
}


void *receiver() {
    msg buf, over1, over2;
    struct msqid_ds t;
    int stop = 2;
    int mq;
    over1.message_type = rcv_1;
```

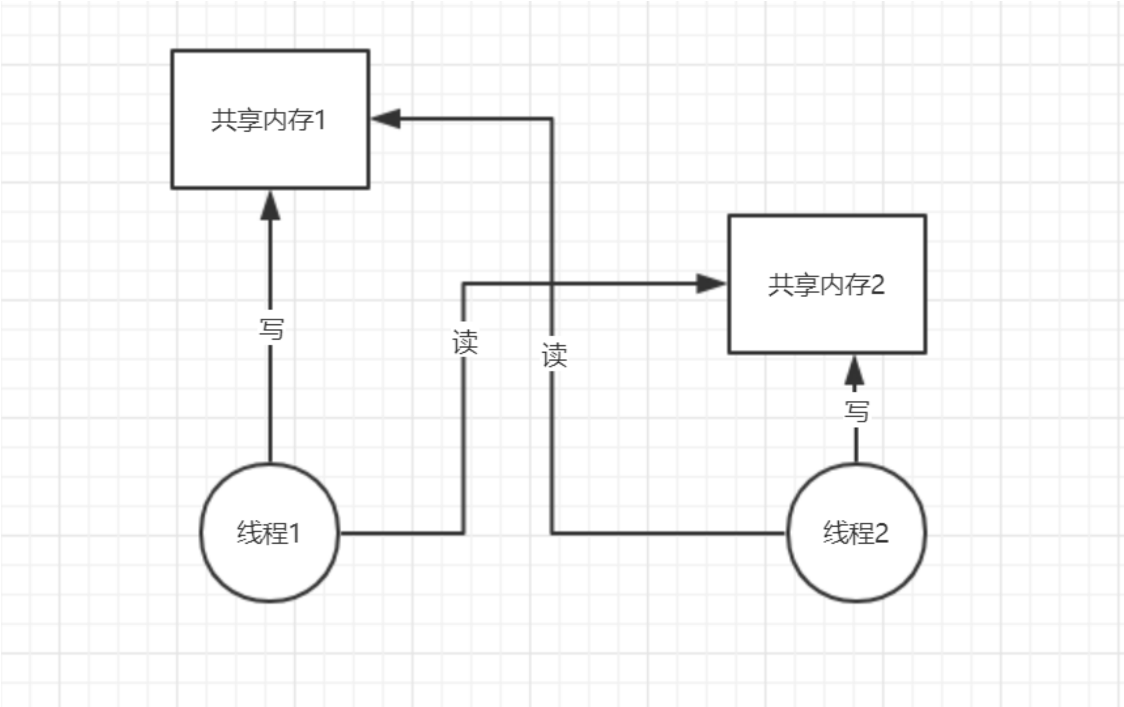
```c
    strcpy(over1.message, "over1\n");
    over2.message_type = rcv_2;
    strcpy(over2.message, "over2\n");


    mq = msgget((key_t) QUEUE_ID,  IPC_CREAT|0666 );

    do {
        sem_wait(&sem_receive);
        msgrcv(mq, (void *) &buf, MAX_SIZE, 0, 0);
        printf("Received%ld: %s", buf.message_type, buf.message);
        printf("--------------------------------------------\n");

        if ((strncmp(buf.message, "end1", strlen("end1"))==0)&&buf.message_type == snd_1) {
           msgsnd(mq, (void *) &over1, MAX_SIZE, 0);
            sem_post(&sem_over1);
            stop--;
        }else if((strncmp(buf.message, "end2", strlen("end2"))==0)&&buf.message_type == snd_2) {
           msgsnd(mq, (void *) &over2, MAX_SIZE, 0);
            sem_post(&sem_over2);
            stop--;
        }else{
                sem_post(&sem_send);
        }
    } while (stop);
    msgctl(mq, IPC_RMID, &t);
    pthread_exit(NULL);
}

int main(int argc, char **argv) {
    pthread_t t1, t2,t3;

    sem_init(&sem_send, 0, 1);
    sem_init(&sem_receive, 0, 0);
    sem_init(&sem_over1, 0, 0);
    sem_init(&sem_over2, 0, 0);

    pthread_create(&t3, NULL, receiver, NULL);
    pthread_create(&t1, NULL, sender1, NULL);
    pthread_create(&t2, NULL, sender2, NULL);

    pthread_join(t3, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

编写程序创建三个线程：sender1 线程、sender2 线程线程二和 receiver 线程。

receiver 线程先等待 `sem_receive` 信号量，然后读出队列中的消息。并释放 `sem_receive` 信号量。如果 sender 线程发送的消息为 exit，则 receiver 线程根据发送线程不同释放 `sem_over1` 或 `sem_over2` 信号量。 sender 线程首先等待 `sem_send` 信号量，然后向消息队列写入消息，并释放 `sem_receive` 信号量。如果 sender 线程发送消息 exit，则退出循环，等待对应的 `sem_over1` 或者 `sem_over2` 信号量。

4. 利用 Linux 的共享内存通信机制实现两个进程间的双向通信

首先创建共享内存，创建信号并对信号量初始化，两个进程分别创建一块共享内存，他们间的关系如下图所示：



这样不需要任何信号量即可实现两个线程间的双向通信。

```c
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

#define PATHNAME "."
#define PROJ_ID_R 0x6638
#define PROJ_ID_W 0x6639


int CreateShm(int size);
int DestroyShm(int shmid);
int GetShm(int size);

static int CommShm(int size,int flags,int app)
{
    key_t key = ftok(PATHNAME,PROJ_ID_W);
    if(app==0){
            key = ftok(PATHNAME,PROJ_ID_R);
    }
    if(key < 0)
    {
        perror("ftok");
        return -1;
    }
    int shmid = 0;
    if((shmid = shmget(key,size,flags)) < 0)
    {
        perror("shmget");
        return -2;
    }
    return shmid;
}
int DestroyShm(int shmid)
{
    if(shmctl(shmid,IPC_RMID,NULL) < 0)
    {
        perror("shmctl");
        return -1;
    }
```

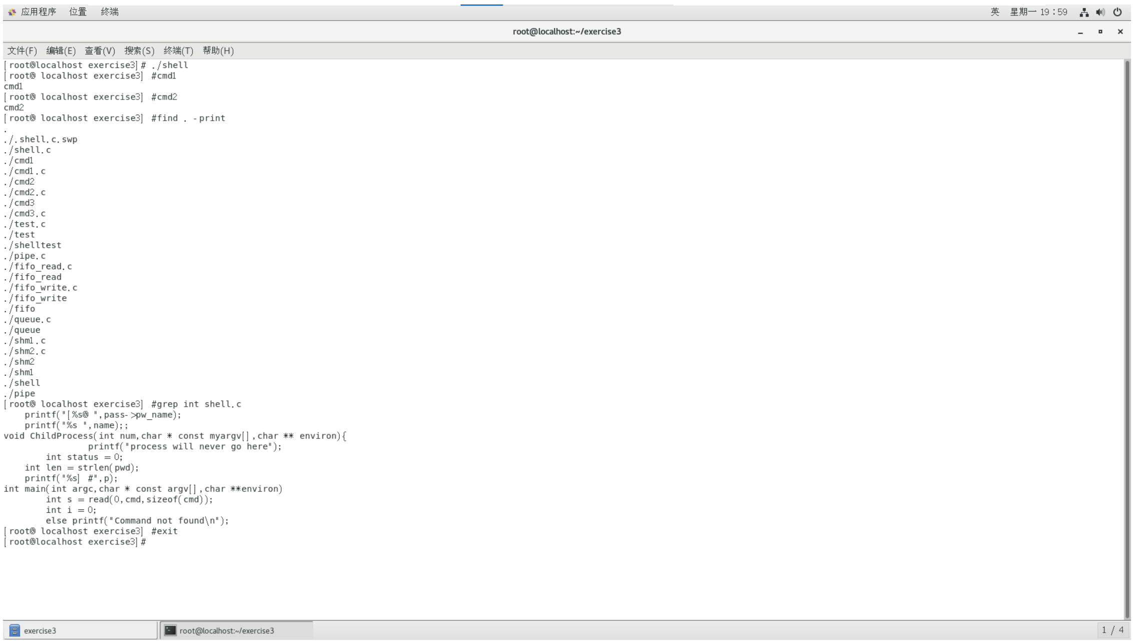
```c
    return 0;
}
int CreateShm(int size)
{
    return CommShm(size,IPC_CREAT,1);
}
int GetShm(int size)
{
    return CommShm(size,IPC_CREAT,0);
}

int main()
{
    int shmid_r = GetShm(4096);
    int shmid_w = CreateShm(4096);
    char *addr_r = shmat(shmid_r,NULL,0);
    char *addr_w = shmat(shmid_w,NULL,0);
    int app;
    while(1)
    {
        printf("1--read  2--write 3--exit");
        scanf("%d",&app);
        if(app == 3){
            break;
        }else if(app == 1){
            printf("%s\n",addr_r);
        }else{
            printf("input:\n");
            scanf("%s",addr_w);
        }
    }
    shmdt(addr_r);
    shmdt(addr_w);
    DestroyShm(shmid_w);
    return 0;
}

```

```c
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

#define PATHNAME "."
#define PROJ_ID_R 0x6639
#define PROJ_ID_W 0x6638


int CreateShm(int size);
int DestroyShm(int shmid);
int GetShm(int size);

static int CommShm(int size,int flags,int app)
{
    key_t key = ftok(PATHNAME,PROJ_ID_W);
    if(app==0){
            key = ftok(PATHNAME,PROJ_ID_R);
    }
    if(key < 0)
    {
        perror("ftok");
        return -1;
    }
    int shmid = 0;
```

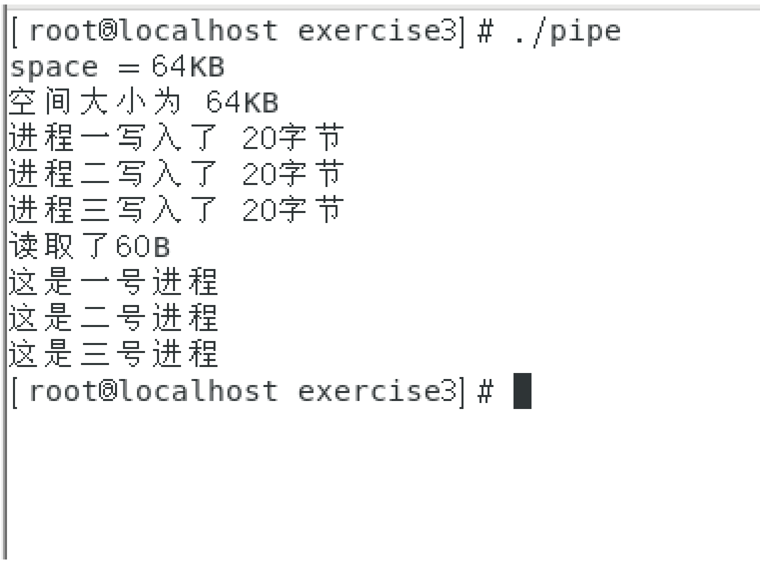
```c
    if((shmid = shmget(key,size,flags)) < 0)
    {
        perror("shmget");
        return -2;
    }
    return shmid;
}
int DestroyShm(int shmid)
{
    if(shmctl(shmid,IPC_RMID,NULL) < 0)
    {
        perror("shmctl");
        return -1;
    }
    return 0;
}
int CreateShm(int size)
{
    return CommShm(size,IPC_CREAT,1);
}
int GetShm(int size)
{
    return CommShm(size,IPC_CREAT,0);
}

int main()
{
    int shmid_r = GetShm(4096);
    int shmid_w = CreateShm(4096);
    char *addr_r = shmat(shmid_r,NULL,0);
    char *addr_w = shmat(shmid_w,NULL,0);
    int app;
    while(1)
    {
        printf("1--read  2--write 3--exit");
        scanf("%d",&app);
        if(app == 3){
            break;
        }else if(app == 1){
            printf("%s\n",addr_r);
        }else{
            printf("input:\n");
            scanf("%s",addr_w);
        }
    }
    shmdt(addr_r);
    shmdt(addr_w);
    DestroyShm(shmid_w);
    return 0;
}

```

结果演示

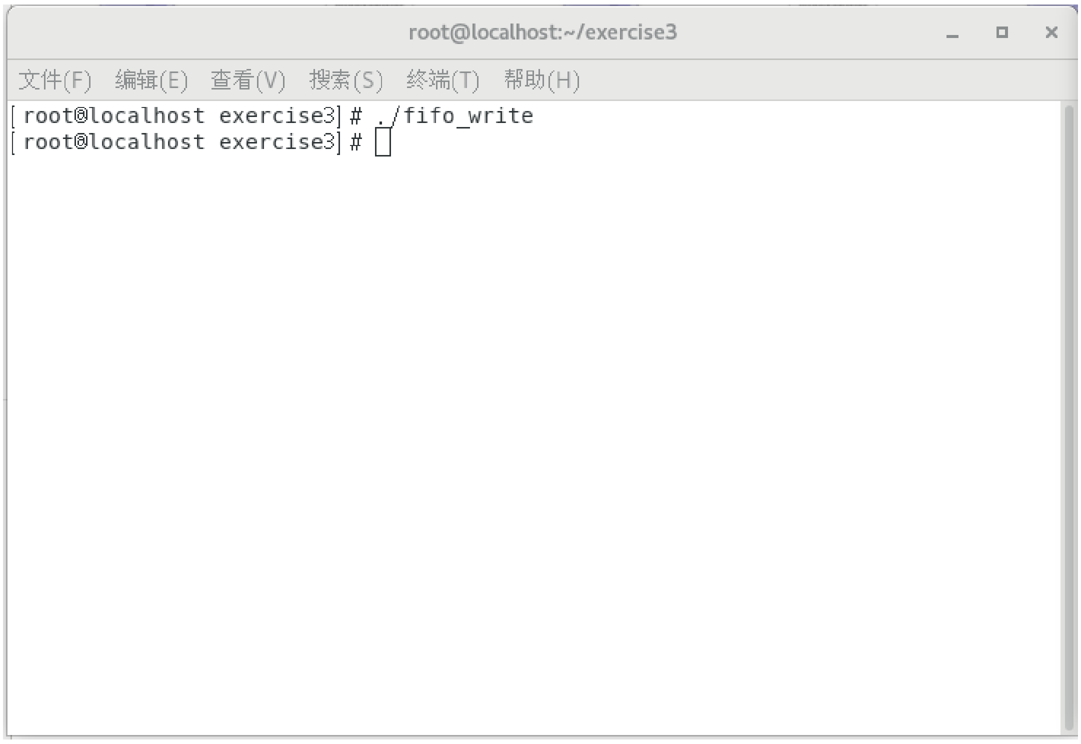
- 模拟 shell



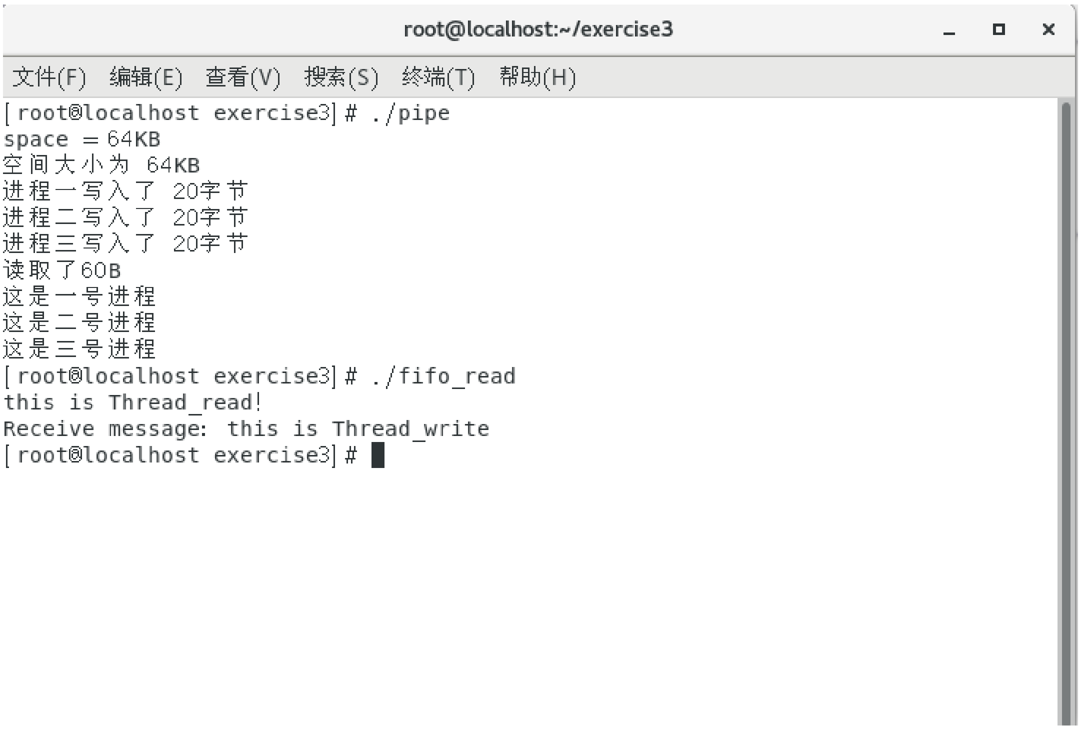
- 管道通信实验
  - 无名管道



  - 有名管道

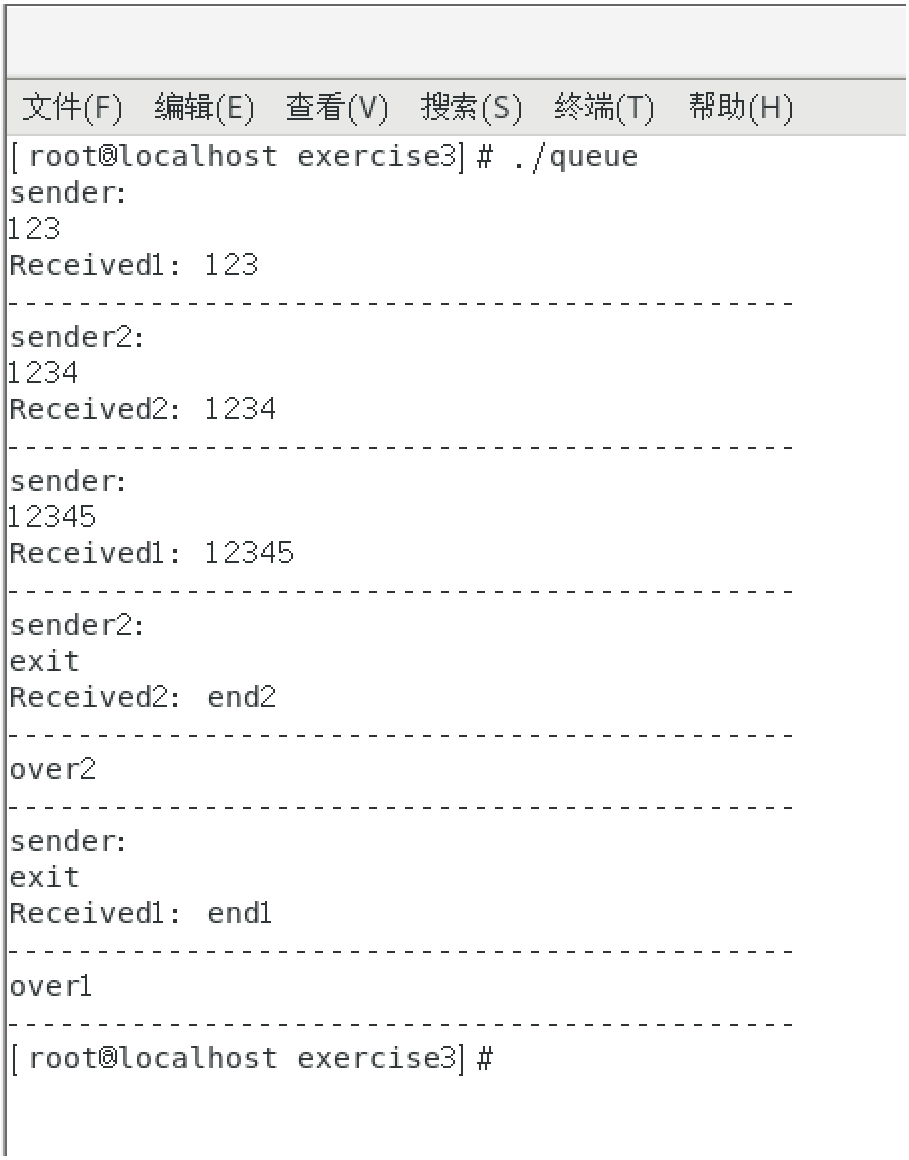


root@localhost:~/exercise3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost exercise3]# ./fifo_write
[root@localhost exercise3]#



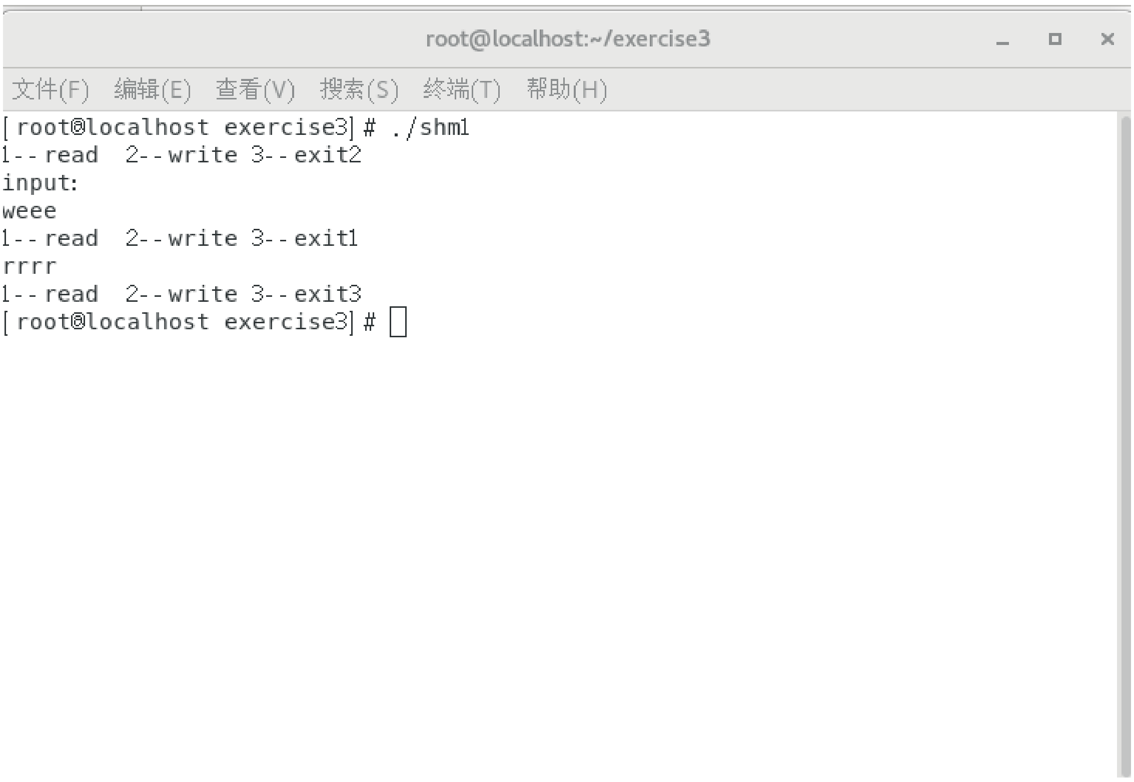
root@localhost:~/exercise3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost exercise3]# ./pipe
space = 64KB
空间大小为 64KB
进程一写入了 20字节
进程二写入了 20字节
进程三写入了 20字节
读取了60B
这是一号进程
这是二号进程
这是三号进程
[root@localhost exercise3]# ./fifo_read
this is Thread_read!
Receive message: this is Thread_write
[root@localhost exercise3]#

- 消息队列

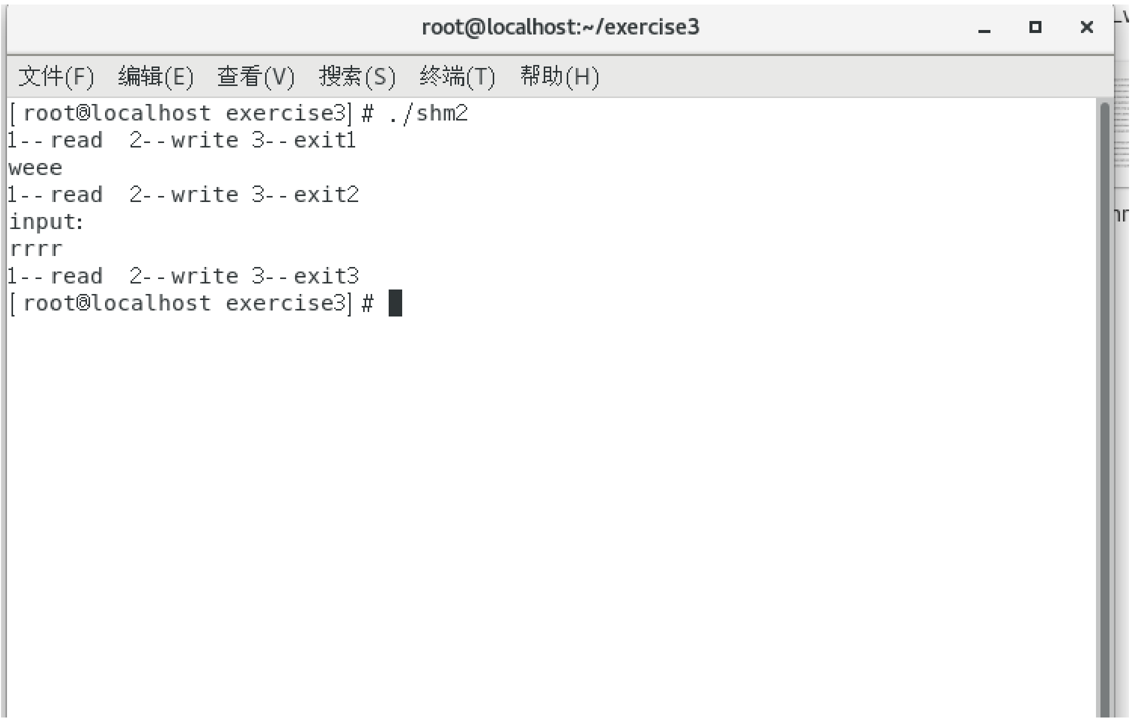


```
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
[root@localhost exercise3]# ./queue
sender:
123
Received1: 123
--------------------------------------------
sender2:
1234
Received2: 1234
--------------------------------------------
sender:
12345
Received1: 12345
--------------------------------------------
sender2:
exit
Received2: end2
--------------------------------------------
over2
--------------------------------------------
sender:
exit
Received1: end1
--------------------------------------------
over1
--------------------------------------------
[root@localhost exercise3]#
```

- 共享内存双向通信



```
root@localhost:~/exercise3
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
[root@localhost exercise3]# ./shm1
1--read  2--write 3--exit2
input:
weee
1--read  2--write 3--exit1
rrrr
1--read  2--write 3--exit3
[root@localhost exercise3]#
```



体会

通过本次实验，我对 Linux 的通信机制有了深入的了解，通过实验我分别通 过有名管道、无名管道、消息队列、共享内存，四种方式实现了进程或线程间 的通信，并通过对比了解了不同机制之间的区别。同时我还对 Linux 信号量实现 进程互斥有了更加深入的体会。