

操作系统课程设计实验报告

实验题目：Linux 进程管理

姓 名：张艺中 王翔宇

学 号：21272134 22200215

专 业：网络工程 网络空间安全

班 级：21272411 22270311

老师姓名：张祯

日 期：2024 年 4 月 29 日

目 录

一 题目介绍.....	1
二 实验思路.....	1
三 遇到问题及解决方法.....	5
四 核心代码及实验结果展示	6
4.1 实验小组分工	6
4.2 核心代码及实验结果	6
五 个人实验改进与总结.....	19
5.1 个人实验改进	19
5.2 个人实验总结	19
六 参考文献.....	19

(大家注意，目录是自动生成的，页码从正文部分开始，当同学们把正文写完后，只需要右击目录，选择更新域，目录会自动更新)

一 题目介绍

本实验通过编写模拟 Shell，管道通信程序，消息通信程序，共享内存机制，了解 Linux 进程的创建，进程通信的方法。

具体实验目的：

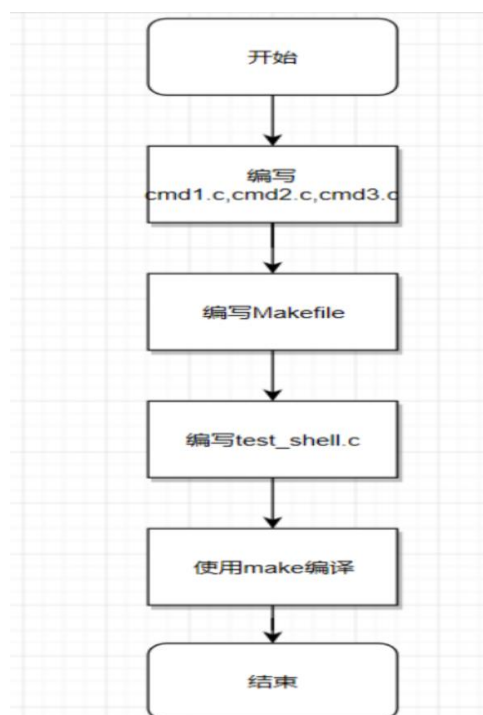
通过对 Linux 进程控制的相关系统调用的编程应用，进一步加深对进程概念的理解，明确进程和程序的联系和区别，理解进程并发执行的具体含义。

通过 Linux 管道通信机制、消息队列通信机制、共享内存通信机制的应用，加深 对不同类型的进程通信方式的理解。

通过对 Linux 的 Posix 信号量及 IPC 信号量的应用，加深对信号量同步机制的理解。

二 实验思路

实现一个模拟的 shell



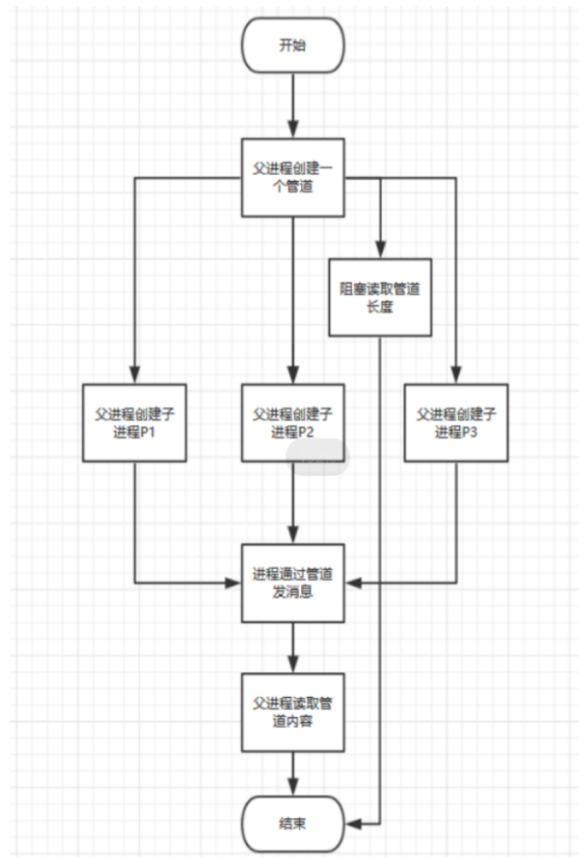
`pid = fork()` 会返回多个值，只需在 `fork()` 后使用多个判断语句即可。

`pid < 0` 表示错误，我打印 `error` 之后退出

`pid = 0` 表示子进程运行，使用 `execl` 替换进程，替换为我们想要的进程，如 `cmd.o`。

`pid > 0` 表示父进程运行，使用 `wait(NULL)` 函数等待所有子进程的退出。

实现一个管道通信程序



- 子进程一先将 64k 的数据写入管道，父进程在第一时间将数据全部读取出来
 - 父进程将子进程一的数据读取之后，子进程二、子进程三才能写入数据
 - 子进程二、子进程三数据写入后，父进程随后才能读取第二批数据
 - 子进程写入数据 1 和父进程读取数据 1 利用 `wait(0)` 限制了先后关系，父进程必须
 - 接收到子进程结束之后返回的 0，才能继续运行，否则阻塞。
 - `write_mutex` 限制了父进程先读取数据，然后子进程二、三写入数据，
 - `read_mutex1` 和 `read_mutex2` 分别限制了子进程二、三写入数据
- 2,3 和父进程读取数
- 据 2,3 先后关系，只有子进程二、三均完成后，父进程才允许读取管道。

利用 Linux 的消息队列通信机制实现两个线程间的通信实验思路：

实验要求为实现两个发送进程和一个接收进程之间通过消息队列实现进程间的通信。

具体实验要求概括如下：

2. 消息队列的创建与共享：

- `sender1()` 创建一个消息队列。

-
- sender2() 共享 sender1() 创建的消息队列。
3. 消息的发送与接收:
 - sender1() 和 sender2() 等待用户通过终端输入字符, 并将这些字符通过消息队列发送给 receiver。可以多次发送消息, 直到用户输入“exit”。
 - receiver() 从消息队列中接收来自 sender1 和 sender2 的消息, 并将收到的消息显示在终端上。
 4. 特殊消息的处理:
 - sender1() 在用户输入“exit”后, 发送消息“end1”给 receiver, 并等待 receiver 的应答消息“over1”, 然后将应答消息显示在终端上。
 - sender2() 在用户输入“exit”后, 发送消息“end2”给 receiver, 并等待 receiver 的应答消息“over2”, 然后将应答消息显示在终端上。
 - receiver() 在收到“end1”消息后, 向 sender1 发送应答消息“over1”; 在收到“end2”消息后, 向 sender2 发送应答消息“over2”。
 5. 同步与互斥:
 - 使用信号量机制实现三个线程/进程之间的同步与互斥, 确保消息队列在并发访问时的正确性和一致性。
 6. 终止条件与清理:
 - receiver() 在接收到“end1”和“end2”消息并发送相应的应答消息后, 删除消息队列并结束运行。

实验设计思路:

1. 采用 System V 消息队列
 - 创建或打开消息队列
`int msgget(key_t key, int msgflg);`其中 key 用于标识消息队列, msgflg 是标志位, 用于指定消息队列的创建权限和行为, 如 IPC_CREAT 表示如果消息队列不存在则创建。
 - 向消息队列发送消息
`int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);`
msqid: 消息队列标识符, 由 msgget 返回。
msgp: 指向消息的指针。
msgsz: 消息正文的大小 (不包括消息类型)。
msgflg: 控制操作的标志位 (如 IPC_NOWAIT 表示如果队列已满则不等待)。
 - 从消息队列接收消息
`ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);`

msgsz: 要接收的消息正文的最大大小。

msgtyp: 指定消息类型。如果为 0，则接收队列中的第一条消息；如果大于 0，则接收类型为 msgtyp 的第一条消息。

msgflg: 控制操作的标志位（如 IPC_NOWAIT 表示如果没有符合条件的消息则不等待，而是立即返回 -1，并设置 errno 为 ENOMSG）。

- 控制消息队列（删除、设置信息等）

int msgctl(int msqid, int cmd, struct msqid_ds *buf);

cmd: 控制命令，如 IPC_RMID（删除消息队列）、IPC_STAT（获取消息队列的当前状态）、IPC_SET（设置消息队列的状态）。

buf: 指向 msqid_ds 结构的指针，用于获取或设置消息队列的信息。

2. 采用 System V 信号量接口

该实验采用了一个互斥信号量，用于互斥两个发送进程之间对消息队列的读写，由于消息队列本身存在机制（当消息队列为空时，接收进程会自动阻塞直到有消息，当消息队列为满时，发送进程会自动阻塞直至消息队列有空间）。

3. 进程结束后的操作

当进程最终运行结束后接收进程删除消息队列以及信号量的关闭和删除。

利用 Linux 的共享内存通信机制实现两个进程间的通信：

实验要求为一个发送进程和一个接收进程之间利用共享内存进行进程通信。

具体实验要求概括如下：

1. 创建共享内存：sender 程序创建共享内存，用于传递信息。
2. 生成计算表达式：sender 随机生成 100 以内的计算表达式（如 12+34），并通过共享内存发送给 receiver。
3. 计算表达式：receiver 接收表达式并显示在终端上，计算结果后，通过共享内存发送回 sender。
4. 显示结果：sender 显示接收到的计算结果。
5. 循环操作：上述过程重复 10 次。
6. 结束通信：第 10 次后，sender 向 receiver 发送 "end"，receiver 回复 "over" 后，sender 删除共享内存并结束程序运行。
7. 互斥与同步：使用信号量机制确保两个进程对共享内存的互斥和同步使用。

实验设计思路：

1. 创建共享内存

共享内存是一种进程间通信(IPC)机制,允许多个进程共享一块物理内存,从而实现高速的数据传递。共享内存的类型是通过内存标识符 `shmid` 表示的,这个标识符是由操作系统内核分配和管理的。

1) 创建或获取共享内存

```
int shmget(key_t key, size_t size, int shmflg);
```

`key`: 唯一标识共享内存段的键值,通常使用 `ftok` 函数生成。

`size`: 共享内存段的大小(以字节为单位)。

`shmflg`: 标志位,指定访问权限和创建选项,例如 `IPC_CREAT` 表示如果共享内存段不存在则创建, `0666` 表示读写权限。

2) 共享进程映射到进程

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

`shmid`: 共享内存段的标识符,由 `shmget` 返回。

`shmaddr`: 附加地址,一般设为 `NULL`,由系统决定附加位置。

`shmflg`: 标志位,例如 `SHM_RDONLY` 表示只读附加, `0` 表示读写附加。

3) 解除映射

```
int shmdt(const void *shmaddr);
```

`shmaddr`: 指向共享内存段的指针,之前由 `shmat` 返回。

2. 使用 POSIX 信号量

该实验采用了三个信号量

```
sem_t* sem_share = sem_open(SEM_SHARE, O_CREAT, PERMISSIONS, 1);
```

```
sem_t* sem_send = sem_open(SEM_SEND, O_CREAT, PERMISSIONS, 0);
```

```
sem_t* sem_recv = sem_open(SEM_RECV, O_CREAT, PERMISSIONS, 0);
```

`sem_share` 用于发送进程和接收进程之间对共享内存的互斥操作,`sem_send` 和 `sem_recv` 用于发送进程和接收进程之间的同步,当发送进程向共享内存写入消息后激活 `sem_send`,接收进程收到 `sem_send` 后从阻塞态转为就绪态,读出消息,处理后将反馈写入内存并激活 `sem_recv`,接收进程收到 `sem_recv` 后读出消息。之后进行下一轮的操作。

3. 信号量的删除

当进程最终运行结束后发送进程解除内存映射、删除共享内存以及信号量的关闭和删除。

三 遇到问题及解决方法、

问题:在进行利用消息队列进行进程间通信时不明白到底要使用多少个信号量

解决方法:在网络上查阅了资料,了解了消息队列本身的特性:

阻塞操作：

- 当发送进程尝试发送消息到已满的消息队列时，发送操作可以被阻塞，直到有足够的空间存储消息。
- 当接收进程尝试从空的消息队列读取消息时，接收操作可以被阻塞，直到有消息可供读取。

因此只采用一个互斥信号量来实现写进程之间对消息队列的互斥操作

四 核心代码及实验结果展示

4.1 实验小组分工

王翔宇：负责 3，4

张艺中：负责 1，2

4.2 核心代码及实验结果

实现一个模拟的 shell

Shelltest.c

```
#include<stdio.h>
#include<unistd.h>
#include<wait.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
char *argv[8];//可执行文件的文件名
int argc = 0;
void do_parse(char *buf)//解析命
{ int i;
  int status = 0;//命令可执行的状态
  for(i = 0; buf[i] != 0; i++)
  {
    if(!isspace(buf[i]) && status == 0)//从字符到空格
    {
      argv[argc++] = buf+ i;
      status = i;
    }
  }
```

```
    else
    {
        if(!isspace(buf[i]))//从空格到字符
        {
            status = 0;
            buf[i] = 0;

        }
    }
}
argv[argc] = NULL;
}
```

```
void do_execute()//执行命令
{
    pid_t pid = fork();
    int wpid;
    if(pid > 0)//父进程
    {
        int st;
        while(wait(&st) != pid);
    }
    else if(pid == 0)
    {
        execvp(argv[0], argv);
        perror("execvp");
        exit(EXIT_FAILURE);
    }else
    {
        perror("fork!");
        exit(EXIT_FAILURE);
    }
}
```

```
int main()
```

```
{
char buf[1024] = {};
while(1)
{
printf("myshell>");
fflush(stdout);//刷新缓冲区
fgets(buf, sizeof(buf), stdin);
if(!strcmp(buf, "exit\n"))//如果输入 exit，则退出程序
exit(0);
else
{
do_parse(buf);//解析命令行
do_execute();//执行命令
}
}
return 0;
}
```

Shelltest2.c

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

```
int main()
{
pid_t pid, wpid;
pid = fork();
if (pid == 0)//子进程
{
sleep(10);//休眠 10s
exit(1);//子进程退出
}
else if (pid > 0)//父进程
{
```

```

do
{ //轮询方式探测是否是子进程 pid, 是否可以被回收, status 暂不考虑, 我们使用
NULL

    wpid = waitpid(pid,NULL,WNOHANG);
    if (wpid == 0)//子进程不可回收
    {
        printf("no child exit!\n");
        sleep(1);
    }
    if (wpid == pid)//子进程被回收
    {
        printf("success wait child.\n");
    }
} while (wpid == 0);
printf("parent exit!\n");
}
return 0;
}

```

此代码主要功能是先对子进程进行休眠, 让其执行父进程, 父进程的工作主要是判断子进程是否可以回收, 当执行 10 次之后, 子进程结束, 父进程可以回收, 回收结束后父进程也结束。

编写 cmd 代码

```

heisenberg@ubuntu:~$ cd exp3
heisenberg@ubuntu:~/exp3$ ls
exp3_1  exp3_2
heisenberg@ubuntu:~/exp3$ mkdir exp3_3
heisenberg@ubuntu:~/exp3$ cd exp3_3
heisenberg@ubuntu:~/exp3/exp3_3$ vim cmd1.c
heisenberg@ubuntu:~/exp3/exp3_3$ vim cmd2.c
heisenberg@ubuntu:~/exp3/exp3_3$ vim cmd3.c
heisenberg@ubuntu:~/exp3/exp3_3$ 

```

编译

```

heisenberg@ubuntu:~/exp3/exp3_3$ gcc cmd1.c -o cmd1
heisenberg@ubuntu:~/exp3/exp3_3$ gcc cmd2.c -o cmd2
heisenberg@ubuntu:~/exp3/exp3_3$ gcc cmd3.c -o cmd3
heisenberg@ubuntu:~/exp3/exp3_3$ 

```

利用管道通信

```

pipe_communication.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#define READ 0 //filedes[0]用于读
#define WRITE 1 //filedes[1]用于写
int main() {
/*
函数原型: pipe(int filedes[2])
参数含义: filedes[0]对应管道读端, filedes[1]对应管道写端
功能: pipe 在内存缓冲区中创建一个管道, 并将读写该管道的一对文件描述符保存在 filedes 所指数组中
返回值: 成功返回 0, 失败返回-1
*/
int filedes[2];
pid_t pid1, pid2, pid3; //pid_t 本质就是 int
char buf[256]; //用作 read 的缓冲区, 保存读取的字符
pipe(filedes); //创建无名管道
if((pid1 = fork()) == -1) { //创建子进程
    printf("fork error(pid1)!\n");
    exit(1);
}
if(pid1 == 0) {
    sleep(1); //挂起一秒
    printf("正在产生子进程 pid1:%d\n", getpid());
    //子进程向父进程写数据, 关闭管道的读端
    close(filedes[READ]);
    write(filedes[WRITE], "pid111111\n", strlen("pid111111\n"));
    exit(0);
}
if ((pid2 = fork()) == -1) {
    printf("fork error(pid2)\n");
    exit(1);
}

```

```

    }
    if (pid2 == 0) {
        sleep(1);
        printf("正在产生子进程 pid2:%d\n", getpid());
        close(filedes[READ]);
        write(filedes[WRITE], "pid222222\n", strlen("pid222222\n"));
        exit(0);
    }
    if ((pid3 = fork()) == -1) {
        printf("fork error(pid3)\n");
        exit(1);
    }
    if (pid3 == 0) {
        sleep(1);
        printf("正在产生子进程 pid3:%d\n", getpid());
        close(filedes[READ]);
        write(filedes[WRITE], "pid333333\n", strlen("pid333333\n"));
        exit(0);
    }
    else {
        //waitpid()会暂时停止目前进程的执行，直到有信号来或者子进程结束
        pid1 = waitpid(pid1, NULL, WUNTRACED);
        pid2 = waitpid(pid2, NULL, WUNTRACED);
        pid3 = waitpid(pid3, NULL, WUNTRACED);
        printf("main pid: %d\n", getpid());
        printf("wait pid: %d %d %d 返回信息\n", pid1, pid2, pid3);
        /*父进程从管道读取子进程写的的数据，关闭管道的写端*/
        close(filedes[WRITE]);
        //read(): 读取的数据保存在缓冲区 buf
        read(filedes[READ], buf, sizeof(buf));
        printf("3 个子进程传输的数据为: \n%s\n", buf);
    }
    return 0;
}

```

```

heisenberg@ubuntu:~/exp3/exp3_4$ vim pipe_communication.c
heisenberg@ubuntu:~/exp3/exp3_4$ ./pipe_communication
正在产生子进程pid1:5626
正在产生子进程pid2:5627
正在产生子进程pid3:5628
main pid: 5625
wait pid: 5626 5627 5628 返回信息
3 个子进程传输的数据为：
pid111111
pid333333
pid222222

```

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#define READ 0 //filedes[0]用于读
#define WRITE 1 //filedes[1]用于写
int main() {
    /*
    函数原型：pipe(int filedes[2])
    参数含义：filedes[0]对应管道读端，filedes[1]对应管道写端
    功能：pipe 在内存缓冲区中创建一个管道，并将读写该管道的一对文件描述符保存在filedes所指数组中
    返回值：成功返回0，失败返回-1
    */
    int filedes[2];
    pid_t pid1,pid2,pid3;//pid_t 本质就是 int
    char buf[256]; //用作 read 的缓冲区，保存读取的字符
    pipe(filedes); //创建无名管道
    if((pid1 = fork()) == -1) { //创建子进程
        printf("fork error(pid1)!\n");
        exit(1);
    }
    if(pid1 == 0) {
        sleep(1); //挂起一秒
        printf("正在产生子进程pid1:%d\n",getpid());
        //子进程向父进程写数据，关闭管道的读端
        close(filedes[READ]);
        write(filedes[WRITE], "pid111111\n", strlen("pid111111\n"));
        exit(0);
    }
    if ((pid2 = fork()) == -1) {
        printf("fork error(pid2)\n");
        exit(1);
    }
    if (pid2 == 0) {
        sleep(1);
        printf("正在产生子进程pid2:%d\n",getpid());
        close(filedes[READ]);
        write(filedes[WRITE], "pid222222\n", strlen("pid222222\n"));
        exit(0);
    }
    if ((pid3 = fork()) == -1) {
        printf("fork error(pid3)\n");
        exit(1);
    }
    if (pid3 == 0) {
        sleep(1);
        printf("正在产生子进程pid3:%d\n",getpid());
        close(filedes[READ]);
        write(filedes[WRITE], "pid333333\n", strlen("pid333333\n"));
        exit(0);
    }
    else {
        //waitpid()会暂时停止目前进程的执行，直到有信号来或者子进程结束
        pid1 = waitpid(pid1, NULL, WUNTRACED);
        pid2 = waitpid(pid2, NULL, WUNTRACED);
        pid3 = waitpid(pid3, NULL, WUNTRACED);
        printf("main pid: %d\n",getpid());
        printf("wait pid: %d %d %d 返回信息\n",pid1,pid2,pid3);
        /*父进程从管道读取子进程写的的数据，关闭管道的写端*/
        close(filedes[WRITE]);
    }
}

```

```
haha@haha-virtual-machine:~/three$ ./sender1
Enter a message for sender1 (or 'exit' to quit): hello
Sender1: Sent message 'hello'
Enter a message for sender1 (or 'exit' to quit): world
Sender1: Sent message 'world'
Enter a message for sender1 (or 'exit' to quit): exit
Sender1: Sent 'end1' message
Sender1: Received response from receiver: over1
haha@haha-virtual-machine:~/three$

haha@haha-virtual-machine:~/three$ ./sender2
Enter a message for sender2 (or 'exit' to quit): !
Sender2: Sent message '!'
Enter a message for sender2 (or 'exit' to quit): exit
Sender2: Sent 'end2' message
Sender2: Received response from receiver: over2
haha@haha-virtual-machine:~/three$

haha@haha-virtual-machine:~/three$ ./receiver
Receiver: Received message from sender1: hello
Receiver: Received message from sender1: world
Receiver: Received message from sender1: end1
Receiver: Sending response 'over1' to sender1
Receiver: Received message from sender2: !
Receiver: Received message from sender2: end2
Receiver: Sending response 'over2' to sender2
haha@haha-virtual-machine:~/three$
```

利用 Linux 的消息队列通信机制实现两个线程间的通信

sender1:

```
1 // 消息结构体
2 struct message {
3     long int mtype;
4     char mtext[MSG_SIZE];
5 };
6
7 int main() {
8     // 创建或获取消息队列
9     key_t key = ftok(".", 'A');
10    int msg_id = msgget(key, IPC_CREAT | 0666);
11    if (msg_id == -1) {
12        perror("msgget");
13        exit(EXIT_FAILURE);
14    }
15
16    // 创建信号量
17    int sem_id = semget(key, 1, IPC_CREAT | 0666);
18    if (sem_id == -1) {
19        perror("semget");
20        exit(EXIT_FAILURE);
21    }
22    semctl(sem_id, 0, SETVAL, 1);
23
24    struct message msg;
25    wait_semaphore(sem_id);
26
27    while (1) {
28        // 用户输入消息
29        char input[MSG_SIZE];
30        printf("Enter a message for sender1 (or 'exit' to quit): ");
31        fgets(input, MSG_SIZE, stdin);
32        input[strcspn(input, "\n")] = '\0'; // 移除换行符
33
34        // 用户输入'exit', 结束发送消息
35        if (strcmp(input, "exit") == 0) {
36            strcpy(msg.mtext, "end1");
37            msg.mtype = 1; // 标记为sender1的消息类型
38            msgsnd(msg_id, &msg, sizeof(msg.mtext), 0);
39            printf("Sender1: Sent 'end1' message\n");
40            signal_semaphore(sem_id);
41            break;
42        }
43    }
```

```

73
74 // 发送消息给receiver
75 strcpy(msg.mtext, input);
76 msg.mtype = 1; // 标记为sender1的消息类型
77 msgsnd(msg_id, &msg, sizeof(msg.mtext), 0);
78 printf("Sender1: Sent message '%s'\n", input);
79
80
81
82
83 }
84
85 // 等待接收者的应答消息
86 msgrcv(msg_id, &msg, sizeof(msg.mtext), 3, 0);
87 printf("Sender1: Received response from receiver: %s\n", msg.mtext);
88
89 return 0;
90 }
91

```

sender2:

```

4
5 int main() {
6 // 获取已存在的消息队列
7 key_t key = ftok(".", 'A');
8 int msg_id = msgget(key, 0666);
9 if (msg_id == -1) {
10     perror("msgget");
11     exit(EXIT_FAILURE);
12 }
13 // 获取已存在的信号量
14 int sem_id = semget(key, 1, 0666);
15 if (sem_id == -1) {
16     perror("semget");
17     exit(EXIT_FAILURE);
18 }
19

```

receiver.c


```

struct message msg;
int end_count = 0; // 用于计数接收到的结束消息

while (1) {

    if (msgrcv(msg_id, &msg, sizeof(msg.mtext), 0, 0) == -1) {
        perror("msgrcv");
        exit(EXIT_FAILURE);
    }
    printf("Receiver: Received message from sender%d: %s\n", msg.mtype, msg.mtext);

    // 当接收到end1消息时, 向sender1发送应答消息
    if (strcmp(msg.mtext, "end1") == 0) {
        printf("Receiver: Sending response 'over1' to sender1\n");
        strcpy(msg.mtext, "over1");
        msg.mtype = 3;
        msgsnd(msg_id, &msg, sizeof(msg.mtext), 0);
        end_count++;
    }

    // 当接收到end2消息时, 向sender2发送应答消息
    if (strcmp(msg.mtext, "end2") == 0) {
        printf("Receiver: Sending response 'over2' to sender2\n");
        strcpy(msg.mtext, "over2");
        msg.mtype = 3;
        msgsnd(msg_id, &msg, sizeof(msg.mtext), 0);
        end_count++;
    }

    // 如果接收到两个结束消息, 退出循环
    if (end_count >= 2) {
        break;
    }
}

```

```

    // 如果接收到两个结束消息, 退出循环
    if (end_count >= 2) {
        break;
    }
}

// 删除消息队列和信号量
msgctl(msg_id, IPC_RMID, NULL);
if (semctl(sem_id, 0, IPC_RMID) == -1) {
    perror("semctl - IPC_RMID");
    exit(EXIT_FAILURE);
}

return 0;

```

```
haha@haha-virtual-machine: ~/桌面/three$ ./sender1
Enter a message for sender1 (or 'exit' to quit): hello
Sender1: Sent message 'hello'
Enter a message for sender1 (or 'exit' to quit): world
Sender1: Sent message 'world'
Enter a message for sender1 (or 'exit' to quit): exit
Sender1: Sent 'end!' message
Sender1: Received response from receiver: over1
haha@haha-virtual-machine: ~/桌面/three$

haha@haha-virtual-machine: ~/桌面/three$ ./sender2
Enter a message for sender2 (or 'exit' to quit): !
Sender2: Sent message '!'
Enter a message for sender2 (or 'exit' to quit): exit
Sender2: Sent 'end!' message
Sender2: Received response from receiver: over2
haha@haha-virtual-machine: ~/桌面/three$

haha@haha-virtual-machine: ~/桌面/three$ ./receiver
Receiver: Received message from sender1: hello
Receiver: Received message from sender1: world
Receiver: Received message from sender1: end!
Receiver: Sending response 'over1' to sender1
Receiver: Received message from sender2: !
Receiver: Received message from sender2: end2
Receiver: Sending response 'over2' to sender2
haha@haha-virtual-machine: ~/桌面/three$
```

利用 Linux 的共享内存通信机制实现两个进程间的通信

Sender.c:

```
#define SHM_KEY 8084//共享内存标识符
#define SHM_SIZE 256
#define PERMISSIONS 0666//创建和读写权限
#define SEM_SHARE "shm"
#define SEM_SEND "s_send"
#define SEM_RECV "r_send"//信号量标识符

void input(char* msg);

int main() {
    char msg[SHM_SIZE];//消息最大长度

    int shmid = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | PERMISSIONS);//
    if (shmid < 0) {
        perror("shmget failed");
        exit(EXIT_FAILURE);
    }

    char* shmptr = shmat(shmid, NULL, 0);//共享内存映射到该进程
    sem_t* sem_share = sem_open(SEM_SHARE, O_CREAT, PERMISSIONS, 1);
    sem_t* sem_send = sem_open(SEM_SEND, O_CREAT, PERMISSIONS, 0);
    sem_t* sem_recv = sem_open(SEM_RECV, O_CREAT, PERMISSIONS, 0);

    for (int i = 1; i <= 11; i++) {
        if (i < 11)
            input(msg);
        else
            strcpy(msg, "end\n");

        sem_wait(sem_share);
        strcpy(shmptr, msg);
        sem_post(sem_share);
        sem_post(sem_send);
        printf("\nsender to receiver: %s", msg);
    }
}
```

```

        sem_wait(sem_recv);
        sem_wait(sem_share);
        strcpy(msg, shmptr);
        sem_post(sem_share);
        printf("sender has received: %s", msg);
        sleep(1);
    }
    shmdt(shmptr); //解除共享内存映射
    shmctl(shmid, IPC_RMID, NULL); //删除共享内存
    sem_close(sem_share); //关闭信号量
    sem_close(sem_send);
    sem_close(sem_recv);
    sem_unlink(SEM_SHARE); //删除信号量名称
    sem_unlink(SEM_SEND);
    sem_unlink(SEM_RECV);
    S
    return 0;
}

void input(char* msg) {
    int a = rand() % 100;
    int b = rand() % 100;
    sprintf(msg, "%d+%d\n", a, b);
}

```

Receiver.c:

```

0 #define SHM_KEY 8084
1 #define SHM_SIZE 256
2 #define PERMISSIONS 0666
3 #define SEM_SHARE "shm"
4 #define SEM_SEND "s_send"
5 #define SEM_RECV "r_send"
6
7 int main() {
8     char msg[SHM_SIZE];
9
10    int shmid = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | PERMISSIONS);
11    if (shmid < 0) {
12        perror("shmget failed");
13        exit(EXIT_FAILURE);
14    }
15
16    char* shmptr = shmat(shmid, NULL, 0);
17    sem_t* sem_share = sem_open(SEM_SHARE, O_CREAT, PERMISSIONS, 1);
18    sem_t* sem_send = sem_open(SEM_SEND, O_CREAT, PERMISSIONS, 0);
19    sem_t* sem_recv = sem_open(SEM_RECV, O_CREAT, PERMISSIONS, 0);
20
21    for (int i = 1; i <= 11; i++) {
22        sem_wait(sem_send);
23        sem_wait(sem_share);
24        strcpy(msg, shmptr);
25        sem_post(sem_share);
26        printf("\nreceiver has received: %s", msg);
27
28        if (i < 11) {
29            int a, b;
30            sscanf(msg, "%d+%d", &a, &b);
31            sprintf(msg, "%d\n", a + b);
32        } else {
33            strcpy(msg, "over\n");
34        }
35
36        sem_wait(sem_share);
37        strcpy(shmptr, msg);
38        sem_post(sem_share);
39        sem_post(sem_recv);
40        printf("receiver to sender: %s", msg);

```

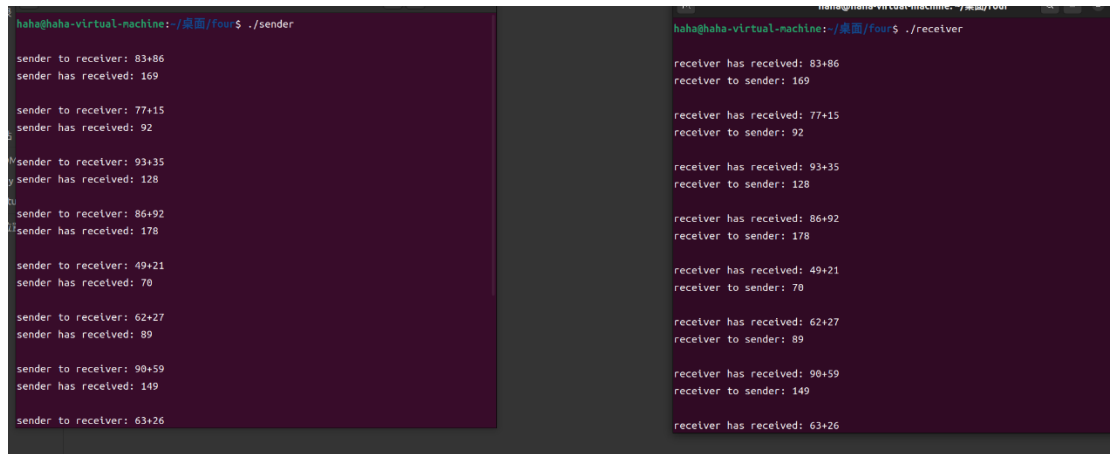
```

    if (i < 11) {
        int a, b;
        sscanf(msg, "%d+%d", &a, &b);
        sprintf(msg, "%d\n", a + b);
    } else {
        strcpy(msg, "over\n");
    }

    sem_wait(sem_share);
    strcpy(shmptr, msg);
    sem_post(sem_share);
    sem_post(sem_recv);
    printf("receiver to sender: %s", msg);
    sleep(1);
}

shmdt(shmptr);
return 0;

```



```
haha@haha-virtual-machine: ~/桌面/404$ ./sender
sender to receiver: 83+86
sender has received: 169

sender to receiver: 77+15
sender has received: 92

sender to receiver: 93+35
sender has received: 128

sender to receiver: 86+92
sender has received: 178

sender to receiver: 49+21
sender has received: 70

sender to receiver: 62+27
sender has received: 89

sender to receiver: 90+59
sender has received: 149

sender to receiver: 63+26

haha@haha-virtual-machine: ~/桌面/404$ ./receiver
receiver has received: 83+86
receiver to sender: 169

receiver has received: 77+15
receiver to sender: 92

receiver has received: 93+35
receiver to sender: 128

receiver has received: 86+92
receiver to sender: 178

receiver has received: 49+21
receiver to sender: 70

receiver has received: 62+27
receiver to sender: 89

receiver has received: 90+59
receiver to sender: 149

receiver has received: 63+26
```

五 个人实验改进与总结

5.1 个人实验改进

网上的代码中写的非常的冗余，乍一看代码量非常多但是很多都是重复的代码，于是我参考网上的代码之后，规范了代码，提高代码质量，把能合并的都合并在一起。

在进行通过消息队列进程通信的实验时，我借鉴网络上的源码，发现信号量的使用有点多，但是消息队列本身存在着一定的同步机制，于是我减少了同步信号量的使用，只使用一个互斥信号量来实现两个写进程之间对消息队列的互斥操作。

5.2 个人实验总结

通过此次实验，我们深入了解了进程间通信的两种主要机制：System V 消息队列和 POSIX 共享内存。我们成功地实现了两个进程之间的消息传递与同步操作。在消息队列部分，我们创建并操作了消息队列，实现了进程间的消息发送和接收，并处理了阻塞与非阻塞模式。在共享内存部分，我们实现了进程间共享数据的存取，并通过信号量确保了共享内存的互斥访问，避免了竞争条件。此次实验不仅巩固了我们对进程间通信的理论知识，也提升了我们的实际编程能力，为后续的系统编程打下了坚实的基础。

六 参考文献

1. https://blog.csdn.net/qg_21989927/article/details/109353402
2. https://blog.csdn.net/weixin_45097312/article/details/102815657
3. https://blog.csdn.net/qg_48458789/article/details/116592086