

学生实验报告

开课学院及实验室：计算机科学与工程实验室电子楼 412

2023 年 04 月 20 日

学院	计算机科学与网络工程学院	年级/专业/班		姓名		学号	
实验课程名称	计算机系统结构与操作系统					成绩	
实验项目名称	实验一 进程管理与进程通信					指导老师	

一、实验目的

- 1、熟悉 LINUX 的常用基本命令，学会编译、调试 C 程序。
- 2、认识进程的创建、控制、互斥及守护进程的建立。
- 3、了解和熟悉 LINUX 支持的信号量机制、管道机制、消息通信机制及共享存储区机制。

二、实验设备和资料

- 1、微型计算机：Linux 操作系统。
- 2、《操作系统实验指导书》的实验一的 8 个分实验。

三、实验内容与运行结果

1. 进程的创建

- 1) fork() 创建两个子进程，并输出字符。父进程显示'a'，子进程分别显示'b'和'c'。

★ 程序源码(解决编译警告、添加注释)

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

void main()
{
    int p1, p2;
    printf("FWM ");
    fflush(stdout);
    // 创建子进程 p1, 创建失败会一直尝试
    while ((p1 = fork()) == -1);
    // 子进程 p1 输出
    if (p1 == 0)
        putchar('b');
    else
    {
        // 创建子进程 p2, 创建失败会一直尝试
        while ((p2 = fork()) == -1);
        // 子进程 p2 输出
```

```

if (p2 == 0)
    putchar('c');
// 父进程输出
else
    putchar('a');
}
exit(0);
}

```

多次运行 shell 脚本

```

#!/bin/bash

echo "该脚本将运行程序 $1_$2 $3 次!"
gcc "$1_$2.c" -o "$1_$2"
for ((i=1;i<=$3;i++))
do
    echo "$($1_$2)"
    echo "-----"
done

```

运行结果截图

```

fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ ./1_0
FWM 32116160100 bacfwm-0100@ubuntu-18:~/OS/实验1$ ./1_0
FWM 32116160100 abcfwm-0100@ubuntu-18:~/OS/实验1$ ./1_0
FWM 32116160100 abcfwm-0100@ubuntu-18:~/OS/实验1$ c./1_0
FWM 32116160100 abcfwm-0100@ubuntu-18:~/OS/实验1$ ./1_0
FWM 32116160100 bacfwm-0100@ubuntu-18:~/OS/实验1$

```

```
fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ ./BulidRun.sh 1 0 10
该脚本将运行程序 1_0 10 次！
FWM 32116160100 abc
-----
FWM 32116160100 acb
-----
FWM 32116160100 abc
-----
FWM 32116160100 acb
-----
FWM 32116160100 abc
-----
FWM 32116160100 abc
-----
FWM 32116160100 abc
-----
FWM 32116160100 abc
-----
FWM 32116160100 abc
-----
FWM 32116160100 abc
-----
FWM 32116160100 abc
-----
fwm-0100@ubuntu-18:~/OS/实验1$
```

- 2) 修改程序，使每个进程循环显示。子进程显示“daughter...”和“son...”，父进程显示“parent...” 观察并分析结果。

★ 程序源码(解决编译警告、添加注释)

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

void main()
{
    int p1, p2, i;
    printf("FWM \n\n");
    // 创建子进程 p1 直到成功
    while ((p1 = fork()) == -1);
    // 子进程 p1 输出
    if (p1 == 0)
        for (i = 0; i < 10; i++)
            printf("daughter %d\n", i);
    else
    {
        // 创建子进程 p2 直到成功
        while ((p2 = fork()) == -1);
        // 子进程 p2 输出
        if (p2 == 0)
            for (i = 0; i < 10; i++)
```

```

        printf("son  %d\n", i);
// 父进程输出
else
    for (i = 0; i < 10; i++)
        printf("parent  %d\n", i);
    }
}

```

运行结果截图

```

fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ ./1_1
FWM 32116160100

daughter 0
parent 0
daughter 1
parent 1
daughter 2
parent 2
daughter 3
parent 3
daughter 4
parent 4
daughter 5
parent 5
daughter 6
parent 6
daughter 7
parent 7
daughter 8
parent 8
daughter 9
parent 9
son 0
son 1
son 2
son 3
son 4
son 5
son 6
son 7
son 8
son 9
fwm-0100@ubuntu-18:~/OS/实验1$

```

2. 进程的控制

`fork()` 创建一个进程，再调用 `exec()` 用新的程序替换该子进程的内容。然后利用 `wait()` 来控制进程执行顺序

程序源码(解决编译警告、添加注释)

```

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

```

```

#include <sys/types.h>
#include <sys/wait.h>

void main()
{
    int pid;
    // 创建子进程
    pid = fork();
    switch (pid)
    {
        // 创建失败
        case -1:
            printf("fork fail!\n");
            exit(1);
            // 子进程
        case 0:
            execl("/bin/ls", "ls", "-l", "-color", NULL);
            printf("exec fail!\n");
            exit(1);
            // 父进程
        default:
            // 同步
            wait(NULL);
            printf("ls completed !\n");
            printf("\nFWM ██████████\n\n");
            exit(0);
    }
}

```

★ 运行结果截图


```
fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ ./2_0
total 220
-rwxrwxrwx 1 fwm-0100 41 Apr 22 12:36 Bulid.sh
-rwxrwxrwx 1 fwm-0100 191 Apr 22 14:06 BulidRun.sh
-rwxrwxrwx 1 fwm-0100 394 Apr 22 12:38 BulidALL.sh
-rw-rw-rw- 1 fwm-0100 1358 Apr 22 12:38 8_0.c
-rwxrwxr-x 1 fwm-0100 8744 Apr 22 13:56 8_0
-rw-rw-rw- 1 fwm-0100 734 Apr 22 12:38 7_0_server.c
-rwxrwxr-x 1 fwm-0100 8568 Apr 22 13:56 7_0_server
-rw-rw-rw- 1 fwm-0100 533 Apr 22 12:38 7_0_client.c
-rwxrwxr-x 1 fwm-0100 8528 Apr 22 13:56 7_0_client
-rw-rw-rw- 1 fwm-0100 1561 Apr 22 12:38 6_0.c
-rwxrwxr-x 1 fwm-0100 8696 Apr 22 13:56 6_0
-rw-rw-rw- 1 fwm-0100 1923 Apr 22 12:38 5_改正.c
-rwxrwxr-x 1 fwm-0100 8656 Apr 22 13:56 5_改正
-rw-rw-rw- 1 fwm-0100 1795 Apr 22 12:38 5_原程序.c
-rwxrwxr-x 1 fwm-0100 8656 Apr 22 13:56 5_原程序
-rw-rw-rw- 1 fwm-0100 1124 Apr 22 13:56 4_0.c
-rwxrwxr-x 1 fwm-0100 8680 Apr 22 13:56 4_0
-rw-rw-rw- 1 fwm-0100 973 Apr 22 12:38 3_0.c
-rwxrwxr-x 1 fwm-0100 8384 Apr 22 13:56 3_0
-rw-rw-rw- 1 fwm-0100 126 Apr 22 12:38 2_hello.c
-rwxrwxr-x 1 fwm-0100 8304 Apr 22 13:56 2_hello
-rw-rw-rw- 1 fwm-0100 541 Apr 22 12:38 2_1.c
-rwxrwxr-x 1 fwm-0100 8464 Apr 22 13:56 2_1
-rw-rw-rw- 1 fwm-0100 592 Apr 22 12:38 2_0.c
-rwxrwxr-x 1 fwm-0100 8464 Apr 22 13:56 2_0
-rw-rw-rw- 1 fwm-0100 685 Apr 22 12:38 1_1.c
-rwxrwxr-x 1 fwm-0100 8384 Apr 22 14:06 1_1
-rw-rw-rw- 1 fwm-0100 591 Apr 22 12:38 1_0.c
-rwxrwxr-x 1 fwm-0100 8520 Apr 22 14:09 1_0
ls completed !

FWM 32116160100

fwm-0100@ubuntu-18:~/OS/实验1$
```

3. 进程互斥

修改实验（一）中的程序 2，用 `lockf()` 来给每一个进程加锁，以实现进程之间的互斥。观察并分析出现的现象

★ 程序源码(解决编译警告、添加注释)

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

void main()
{
    int p1, p2, i;
    /*创建子进程 p1*/
    while ((p1 = fork()) == -1);
    if (p1 == 0)
```

```

{
    /*加锁,
    这里第一个参数为 stdout（标准输出设备的描述符）*/
    lockf(1, 1, 0);
    for (i = 0; i < 10; i++)
        printf("daughter %d\n", i);
    /*解锁*/
    lockf(1, 0, 0);
}
else
{
    /*创建子进程 p2*/
    while ((p2 = fork()) == -1);
    if (p2 == 0)
    {
        /*加锁*/
        lockf(1, 1, 0);
        for (i = 0; i < 10; i++)
            printf("son %d\n", i);
        /*解锁*/
        lockf(1, 0, 0);
    }
    else
    {
        /*加锁*/
        lockf(1, 1, 0);
        for (i = 0; i < 10; i++)
            printf(" parent %d\n", i);
        /*解锁*/
        lockf(1, 0, 0);
    }
}
}
}

```

运行结果截图

```
fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ ./3_0
daughter 0
daughter 1
daughter 2
daughter 3
daughter 4
daughter 5
daughter 6
daughter 7
daughter 8
daughter 9
parent 0
parent 1
parent 2
parent 3
parent 4
parent 5
parent 6
parent 7
parent 8
parent 9
son 0
son 1
son 2
son 3
son 4
son 5
son 6
son 7
son 8
son 9
fwm-0100@ubuntu-18:~/OS/实验1$
```

4. 守护进程

写一个使用守护进程(daemon)的程序，实现：

- 1、创建日志文件/var/log/Mydaemon.log
- 2、每分钟写入一个时间戳(time_t 格式)

★ 程序源码(解决编译警告、添加注释)

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

void main()
{
    time_t t;
    FILE *fp;
    fp = fopen("/var/log/Mydaemon.log", "a");
    setlinebuf(fp);
    // 建立文件
    // 打开文件
    // 将文件输出流设置为行缓冲
```



```

pid_t pid;                                // 守护神
pid = fork();                             // 创建子进程
if (pid > 0)
{
    printf("Daemon on duty!\n");
    exit(0);
}
else if (pid < 0)
{
    printf("Can't fork!\n");
    exit(-1);
}
// 父进程退出，保留子进程作为守护进程运行
while (1)
{
    if (fp >= 0)
    {
        sleep(1);
        printf("Daemon on duty!\n");
        t = time(0);                      // 记录当前时间戳
        fprintf(fp, "Now On the FWM's Computer current time is %s\n",
asctime(localtime(&t)));                  // 将时间戳写入日志文件
    }
    fclose(fp); // 关闭文件指针
}

```

运行结果截图

```
fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ sudo ./4_0
Daemon on duty!
fwm-0100@ubuntu-18:~/OS/实验1$ Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
Daemon on duty!
fwm-0100@ubuntu-18: ~
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~$ tail -f /var/log/Mydaemon.log
Now On the FWM's Computer current time is Sat Apr 22 13:59:26 2023
Now On the FWM's Computer current time is Sat Apr 22 13:59:56 2023
Now On the FWM's Computer current time is Sat Apr 22 14:00:26 2023
Now On the FWM's Computer current time is Sat Apr 22 14:00:56 2023
Now On the FWM's Computer current time is Sat Apr 22 14:01:26 2023
Now On the FWM's Computer current time is Sat Apr 22 14:01:56 2023
Now On the FWM's Computer current time is Sat Apr 22 14:02:26 2023
Now On the FWM's Computer current time is Sat Apr 22 14:02:56 2023
Now On the FWM's Computer current time is Sat Apr 22 14:03:26 2023
Now On the FWM's Computer current time is Sat Apr 22 14:03:56 2023
Now On the FWM's Computer current time is Sat Apr 22 14:04:26 2023
```

5. 信号机制

1、编写程序:

创建两个子进程，用 `signal()` 让父进程中断信号（按 `^c` 键）；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 process1 is killed by parent!

Child process2 is killed by parent!

父进程等待两个子进程终止后，输出如下的信息后终止：

Parent process is killed!

★ 程序源码(解决编译警告、添加注释)

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

void waiting(), stop();
int wait_mark;

void main()
{
    int p1, p2, stdout;
    // 创建子进程 p1
    while ((p1 = fork()) == -1);
    // 父进程
    if (p1 > 0)
    {
        // 创建子进程 p2
        while ((p2 = fork()) == -1);
        // 父进程
        if (p2 > 0)
        {
            wait_mark = 1;
            signal(SIGINT, stop); /*接收到^c 信号, 转 stop*/
            waiting(); /*等待^c 信号*/
            kill(p1, 16); /*向 p1 发软中断信号 16*/
            kill(p2, 17); /*向 p2 发软中断信号 17*/
            wait(0); /*同步*/
            wait(0); /*同步*/
            printf("Parent process is killed!\n");
            printf("\nFWM ██████████\n");
            exit(0);
        }
        // 子进程 p2
        else
        {
            wait_mark = 1;
            signal(17, stop); /*接收到软中断信号 17, 转 stop*/
            waiting();
        }
    }
}
```

```

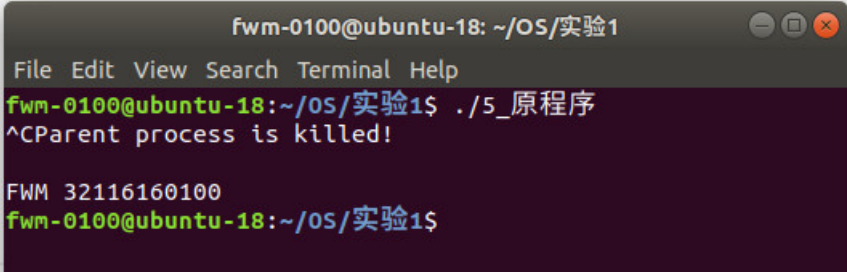
        lockf(stdout, 1, 0);          /*对标准输出进行上锁*/
        printf("Child process 2 is killed by parent!\n");
        lockf(stdout, 0, 0);          /*对标准输出进行解锁*/
        exit(0);
    }
}
// 子进程 p1
else
{
    wait_mark = 1;
    signal(16, stop);                 /*接收到软中断信号 16, 转 stop*/
    waiting();
    lockf(stdout, 1, 0);              /*对标准输出进行上锁*/
    printf("Child process 1 is killed by parent!\n");
    lockf(stdout, 0, 0);              /*对标准输出进行解锁*/
    exit(0);
}
}

void waiting()
{
    while (wait_mark != 0);
}

void stop()
{
    wait_mark = 0;
}

```

★ 运行结果截图



```

fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ ./5_原程序
^CParent process is killed!
FWM 32116160100
fwm-0100@ubuntu-18:~/OS/实验1$

```

2、分析利用软中断通信实现进程同步的机理

软中断主要涉及到 signal 和 kill 两个函数，signal 用于注册收到某个中断信号时的处理方式，kill 则是用于发送某个中断信号。

利用软中断通信时，首先需要设计信号处理函数，如上面程序就设计了处理函数 stop()，然后进程需要在信号到来之前执行 signal() 函数注册，通过处理函数改变循环的条件 wait_mark 来实现进程的同步。

6. 进程管道通信

编写程序实现进程的管道通信。用系统调用 `pipe()` 建立一管道，二个进程 P1 和 P2 分别向管道各写一句话：

Child 1 is sending a message!

Child 2 is sending a message!

父进程从管道中读出二个来自子进程的信息并显示（要求先接收 P1，后 P2）。

★ 程序源码(解决编译警告、添加注释)

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

int pid1, pid2;

void main()
{
    int fd[2];
    char outpipe[100], inpipe[100];
    // 创建一个管道
    pipe(fd);
    // 创建子进程 1
    while ((pid1 = fork()) == -1);
    // 子进程 1
    if (pid1 == 0)
    {
        // 对管道加锁，实现管道操作互斥
        lockf(fd[1], 1, 0);
        // 把串放入数组 outpipe 中
        sprintf(outpipe, "child 1 process is sending message!");
        // 向管道写长为 50 字节的串
        write(fd[1], outpipe, 50);
        // 自我阻塞 5 秒
        sleep(5);
        // 解锁
        lockf(fd[1], 0, 0);
        exit(0);
    }
    // 父进程
    else
    {
```

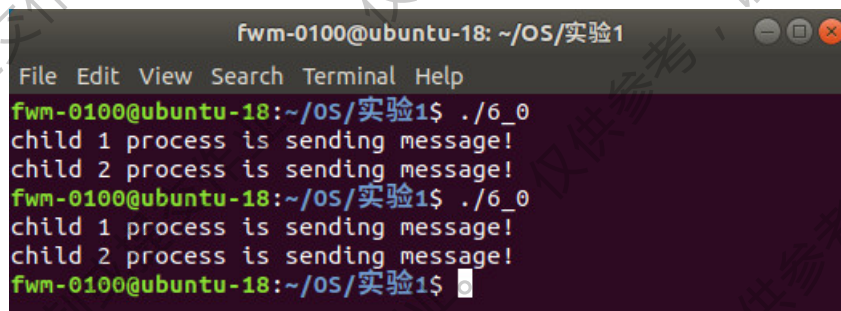


```

// 创建子进程 2
while ((pid2 = fork()) == -1);
// 子进程 2
if (pid2 == 0)
{
    // 子进程 2 操作同子 1
    lockf(fd[1], 1, 0);
    sprintf(outpipe, "child 2 process is sending message!");
    write(fd[1], outpipe, 50);
    sleep(5);
    lockf(fd[1], 0, 0);
    exit(0);
}
// 父进程
else
{
    // 等待子 1
    wait(0);
    // 从管道中读长为 50 字节的串
    read(fd[0], inpipe, 50);
    // 输出
    printf("%s\n", inpipe);
    // 等待子 2
    wait(0);
    read(fd[0], inpipe, 50);
    printf("%s\n", inpipe);
    exit(0);
}
}
}

```

★ 运行结果截图



```

fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ ./6_0
child 1 process is sending message!
child 2 process is sending message!
fwm-0100@ubuntu-18:~/OS/实验1$ ./6_0
child 1 process is sending message!
child 2 process is sending message!
fwm-0100@ubuntu-18:~/OS/实验1$

```

7. 消息发送与接收

消息的创建、发送和接收。使用系统调用 `msgget()`, `msgsnd()`, `msgrcv()`, 及 `msgctl()` 编制一长度为 1 k 的消息发送和接收的程序。

★ 程序源码(解决编译警告、添加注释)

```
// Server.c
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define MSGKEY 75 // 信息队列描述符

// 消息结构
struct msgform
{
    long mtype; // 消息类型, 必须为 long 类型
    char mtext[1000]; // 消息正文
} msg;

int msgqid;

void server()
{
    // 创建 75#消息队列, 权限 0777
    msgqid = msgget(MSGKEY, 0777 | IPC_CREAT);
    do
    {
        // 接收消息
        msgrcv(msgqid, &msg, 1030, 0, 0);
        // 输出
        printf("(server)received\n");
    } while (msg.mtype != 1);

    // 删除消息队列, 归还资源
    msgctl(msgqid, IPC_RMID, 0);
    exit(0);
}

void main()
{
    server();
}

// Client.c
#include <sys/types.h>
```

```

#include <sys/msg.h>
#include <sys/ipc.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define MSGKEY 75

struct msgform
{
    long mtype;
    char mtext[1000];
} msg;

int msgqid;

void client()
{
    int i;
    // 打开 75#消息队列
    msgqid = msgget(MSGKEY, 0777);
    for (i = 10; i >= 1; i--)
    {
        msg.mtype = i;
        printf("(client)sent\n");
        // 发送消息
        msgsnd(msgqid, &msg, 1024, 0);
    }
    exit(0);
}

void main()
{
    client();
}

```

运行结果截图

```
fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ ./7_0_server&
[1] 8035
fwm-0100@ubuntu-18:~/OS/实验1$ ./7_0_client
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
[1]+  Done                  ./7_0_server
fwm-0100@ubuntu-18:~/OS/实验1$
```

8. 共享存储区通信

编制一长度为 1k 的共享存储区发送和接收的程序。

- ★ 程序源码(解决编译警告、添加注释)

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

#define SHMKEY 75

int shmid, i;
int *addr;

void client()
{
    int i;
    shmid = shmget(SHMKEY, 1024, 0777); /*打开共享存储区*/
    addr = shmat(shmid, 0, 0);           /*获得共享存储区首地址*/
    for (i = 9; i >= 0; i--)
    {
```

```

// 同步机制
// -1 代表获得共享存储区首地址失败 或者 server 正在写入
while (*addr != -1);
printf("(client) sent\n");
// 往首地址里写入一个 int 型数据
*addr = i;
}
exit(0);
}

void server()
{
    shmid = shmget(SHMKEY, 1024, 0777 | IPC_CREAT); /*创建共享存储区*/
    addr = shmat(shmid, 0, 0);                      /*获取首地址*/
    do
    {
        // 同步机制
        // 当 client 更改首地址数据时代表写入完毕可读
        *addr = -1;
        while (*addr == -1);
        printf("(server) received\n");
    } while (*addr);
    shmctl(shmid, IPC_RMID, 0); /*撤消共享存储区, 归还资源*/
    exit(0);
}

void main()
{
    printf("\nFWM [REDACTED] \n\n");
    while ((i = fork()) == -1);
    if (!i)
        server();
    system("ipcs -m");
    while ((i = fork()) == -1);
    if (!i)
        client();
    wait(0);
    wait(0);
}

```

运行结果截图


```
fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$
fwm-0100@ubuntu-18:~/OS/实验1$ ./8_0
FWM 32116160100

----- Shared Memory Segments -----
key          shmid      owner      perms      bytes      nattch     status
0x00000000    6         fwm-0100   600        67108864   2          dest
0x00000000    9         fwm-0100   600        524288     2          dest
0x00000000   10         fwm-0100   600        16777216   2          dest
0x00000000   14         fwm-0100   600        524288     2          dest
0x00000000   17         fwm-0100   600        524288     2          dest
0x00000000   18         fwm-0100   600        524288     2          dest
0x00000000   21         fwm-0100   600        6094848    2          dest
0x00000000   22         fwm-0100   600        6094848    2          dest
0x00000000   49         fwm-0100   600        524288     2          dest
0x00000000   62         fwm-0100   600        524288     2          dest
0x0000004b   63         fwm-0100   777        1024       1          dest
(client) sent
(server) received
(client) sent
(server) received
(client) sent
(server) received
(client) sent
(server) received
(client) sent
(server) received
(client) sent
(server) received
(client) sent
(server) received
(client) sent
(server) received
(client) sent
(server) received
(client) sent
(server) received
(fwm-0100@ubuntu-18:~/OS/实验1$
```

四、实验分析与思考

(一) 进程的创建

1. 系统是怎样创建进程的？

1) 申请空白 PCB

系统会先从 PCB 集合给新进程申请空白的 PCB，并为其分配唯一数字进程标识符 PID。

2) 分配资源

系统为新进程分配各种物理和逻辑资源，如：

- 内存空间：进程堆栈、数据段、代码段等
- CPU 时间片
- 文件描述符等

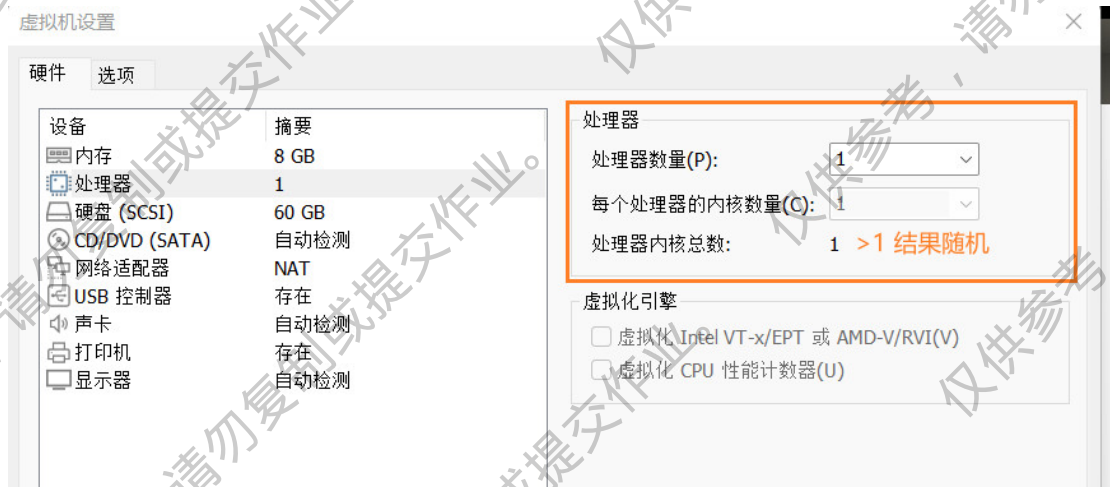
3) 初始化 PCB

- 初始化**标识信息**，将系统分配的标识符和父进程标识符填入新 PCB 中
- 初始化**处理机信息**，使程序计数器指向程序入口地址，使栈指针指向栈顶

初始化处理器控制信息，将进程的状态设置为就绪态或静止就绪态

4) 将进程 PCB 加入就绪队列（若进程就绪队列未满）

2. 结果没有随机



3. 当首次调用新创建进程时，其入口在哪里？

使用 `fork()` 函数创建新进程时，`fork()` 函数会复制父进程的地址空间给子进程，子进程与父进程拥有完全相同的代码、数据与堆栈。操作系统仅在子进程需要修改相关共享内存空间时才会复制相关的页面到新的地址空间（写时复制技术）。

故可知新进程的入口为 `fork()` 函数返回的位置。一般返回位置为 `fork()` 函数的下一行，但在下图小实验（一）代码中，返回位置应处于 `while()` 内。

```
void main()
{
    int p1, p2;
    printf("FWM 32116160100 ");
    fflush(stdout);
    // 创建子进程p1，创建失败会一直尝试
    → while ((p1 = fork()) == -1);
    // 子进程p1输出
    if (p1 == 0)
        ...
}
```

(二) 进程的控制

1. 编写一 hello 程序，实现输出“hello, 姓名”。将 hello 替代上述的 `exec1` 中运行的 `ls` 命令。

★ 源码

// 2_hello.c

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

```
int main(int argc, char* argv[])
{
    printf("hello, \n\n");
    return 0;
}
```

// 2_1.c

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

void main()
{
    int pid;
    // 创建子进程
    pid = fork();
    switch (pid)
    {
        // 创建失败
        case -1:
            printf("fork fail!\n");
            exit(1);
        // 子进程
        case 0:
            execl("./hello", "hello", NULL);
            printf("exec fail!\n");
            exit(1);
        // 父进程
        default:
            // 同步
            wait(NULL);
            printf("hello completed !\n");
            exit(0);
    }
}
```

运行结果截图

```
fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ ./2_1
hello,冯韦铭

hello completed !
fwm-0100@ubuntu-18:~/OS/实验1$
```

2. 什么是进程同步？wait() 是如何实现进程同步的？

什么是进程同步？

进程同步是用于在多进程并发合作场景中为避免资源的争抢的一系列措施，其主要用于协调资源共享关系与相互合作关系，避免因竞争条件、临界区问题、死锁等而导致的程序错误和异常情况的发生。

wait() 是如何实现进程同步的？

wait() 可以用于在有亲缘关系的进程之间实现进程同步。

父进程在调用 wait() 函数时，会进入阻塞状态，直到子进程退出才结束阻塞状态。子进程在执行完毕退出时会向父进程发送一个 SIGCHLD 信号，当父进程接收到该信号时就知道子进程已经结束了，可以开始后续的操作。

在子进程返回前，父进程会一直处于阻塞状态，由此来确保子进程一定先于父进程结束，达到进程同步效果。

此外，在查询资料时我还了解到 wait 还具有回收子进程的占用资源的重要功能。因为在 Linux 中，进程退出后资源并不会被立刻释放，而是通过父进程的 wait 或 waitpid 等函数回收其资源，并且 wait 或 waitpid 成功返回时，内核会将子进程的退出状态放到 status 指向的内存单元，可以通过宏 WEXITSTATUS(status) 解析子进程退出原因。

当父进程没使用 wait 子进程退出时，会导致子进程变成僵尸进程(Zombie Process)。因为在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存等。但是仍然为其保留一定的信息（包括进程号、退出状态、运行时间等），这些信息直到父进程调用 wait/waitpid 才会被释放。

如果父进程直到结束也没有释放僵尸进程，那么在父进程结束后，系统 init 进程将会接管僵尸进程，为其“收尸”。可以通过 ps aux | grep Z 命令查看系统内存在的僵尸进程。

(三) 进程互斥

该题无思考题

(四) 守护进程

1. 以下语句“setlinebuf(fp); //设置行缓冲”起到什么作用？如果没有该语句，

程序的执行结果会怎样？

✖️ setlinebuf(fp)作用

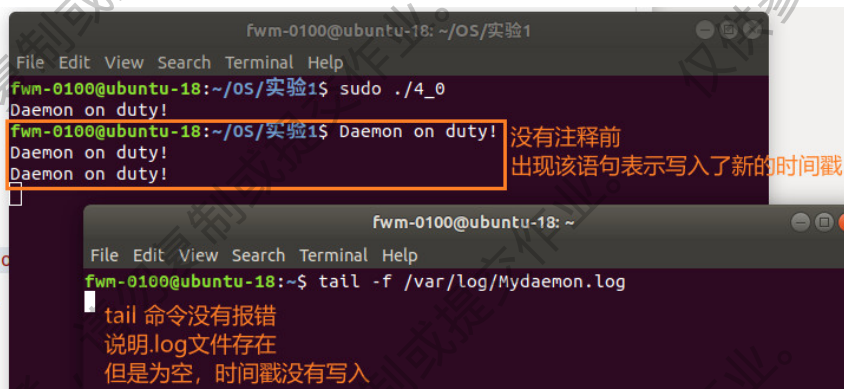
setlinebuf(fp)语句的作用是设置文件流 fp 为行缓冲刷新方式，当遇到'\n'时就会刷新缓冲区。如果没有该语句文件流默认的刷新方式为全刷新方式，即文件缓存区满才会刷新将内容写入文件。该题将 setlinebuf(fp)注释掉是全刷新方式，所以在缓冲区满之前不会将内容写入文件，故与下图结果对应，.log 文件为空。

✖️ 没有该语句的执行结果

如下图所示，.log 文件存在但是文件内容为空，执行结果为程序可以运行（成功创建文件），但是无法达到预取效果（没有写入内容，.log 文件为空）

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

void main()
{
    time_t t;
    FILE *fp;
    fp = fopen("/var/log/Mydaemon.log", "a");
    //setlinebuf(fp);
    pid_t pid;
    pid = fork();
    if (pid > 0)
    {
```



(五) 信号机制

1. 程序的预期结果为显示：

Child process1 is killed by parent!

Child process2 is killed by parent!

Parent process is killed!

预期的结果可以正确显示吗？如果不可以，程序该如何修改才能得到正确结果？

✖️ 预期的结果不可以正确显示

✖️ 修改程序

在原程序中，按下键盘上的 CTRL + C 后父进程会分别对两个子进程发送 16、17 号信号。按下 CTRL + C 会发出 SIGINT 信号，进程接收到该信号时默认会终止当前正在运行的程序。父进程在接收信号之前使用了 signal() 函数更改了接收到 SIGINT 信号的处理方式，但是按下 CTRL + C 不仅仅会对父进程发送 SIGINT 信号，也会对两个子进程发出，两个子进程因为没有提前设置接收到 SIGINT 信号的处理方式，默认被终止了。

故修改的方法为：在两个子进程设置忽略 SIGINT 信号的处理方式。分别添加：

```
signal(SIGINT, SIG_IGN); /*忽略 CTRL+C 信号*/
```

✖️ 修改后程序源码

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



```

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

void waiting(), stop();
int wait_mark;

void main()
{
    int p1, p2, stdout;
    // 创建子进程 p1
    while ((p1 = fork()) == -1);
    // 父进程
    if (p1 > 0)
    {
        //创建子进程 p2
        while ((p2 = fork()) == -1);
        // 父进程
        if (p2 > 0)
        {
            wait_mark = 1;
            signal(SIGINT, stop);    /*接收到^c 信号, 转 stop*/
            waiting();                /*等待^c 信号*/
            kill(p1, 16);             /*向 p1 发软中断信号 16*/
            kill(p2, 17);             /*向 p2 发软中断信号 17*/
            wait(0);                  /*同步*/
            wait(0);                  /*同步*/
            printf("Parent process is killed!\n");
            printf("\nFWM ██████████\n");
            exit(0);
        }
        // 子进程 p2
        else
        {
            wait_mark = 1;
            signal(SIGINT, SIG_IGN); /*忽略 CTRL+C 信号*/
            signal(17, stop);        /*接收到软中断信号 17, 转 stop*/
            waiting();
            lockf(stdout, 1, 0);      /*对标准输出进行上锁*/
            printf("Child process 2 is killed by parent!\n");
            lockf(stdout, 0, 0);      /*对标准输出进行解锁*/
            exit(0);
        }
    }
}

```

```

    }
    // 子进程 p1
    else
    {
        wait_mark = 1;
        signal(SIGINT, SIG_IGN); /* 忽略 CTRL+C 信号*/
        signal(16, stop); /*接收到软中断信号 16, 转 stop*/
        waiting();
        lockf(stdout, 1, 0); /*对标准输出进行上锁*/
        printf("Child process 1 is killed by parent!\n");
        lockf(stdout, 0, 0); /*对标准输出进行解锁*/
        exit(0);
    }
}

void waiting()
{
    while (wait_mark != 0);
}

void stop()
{
    wait_mark = 0;
}

```

```

fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ ./5_改正
^CChild process 2 is killed by parent!
Child process 1 is killed by parent!
Parent process is killed!

FWM 32116160100
fwm-0100@ubuntu-18:~/OS/实验1$

```

2. 该程序段前面部分用了两个 `wait(0)`，它们起什么作用？

两个 `wait(0)` 的作用是进程同步，`wait(0)` 可以保证父进程在子进程结束前保持阻塞状态等待子进程结束，同时可以完全释放子进程占用的系统资源，两个 `wait(0)` 保证了父进程在两个子进程都结束之前保持阻塞等待，以达到父进程是输出一定位于两个子进程输出后的效果。

(六) 进程管道通信

1. 程序中的 sleep(5)起什么作用？

进程中的 sleep(5)使子进程写入管道后休眠了 5 秒，实际上作用是让系统有足够的时间将子进程写入管道的数据从内核的缓冲区转移到管道内，避免因为缓冲区数据尚未写入就被父进程所读取。

2. 怎样保证先 child1 进程，再 child2 进程？

原程序中是通过代码的先后顺序来保证的。在原程序中，父进程先创建子进程 1，然后 wait 等待子进程 1 写入完成程序结束后读取管道，再创建子进程 2 然后同样 wait 子进程 2 写入管道，最后读取管道数据。通过程序执行的先后保证了先 child1 进程，再 child2 进程。

3. 子进程 1 和 2 为什么也能对管道进行操作？

因为父进程在创建子进程前调用 pipe 函数获取了系统为管道创建的两个文件描述符（对应读、写端）。子进程 1 和 2 在创建时会复制父进程的文件描述表，故两个子进程也获得了该管道两个文件描述符，可以通过管道文件描述符对管道进行操作。

4. 思考

指导书给的代码中是通过代码的先后顺序来保证一定先子进程 1 写入后子进程 2 写入，我尝试使用软中断写出一个程序，在同一时间内有两个子进程存在，并且他们可以有序地写入管道。

在编写程序的过程中，我踩了以下几个坑。

a) signal user define 1

在一开始的程序中我没有添加子进程就绪机制，在父进程创建完两个子进程后立刻向子进程 1 发送信号，理想状态下子进程 1 在收到信号后调用 ChildHandler() 写入管道，但实际上程序会报错 signal user define 1，原因是子进程 1 还尚未调用 signal() 注册信号处理函数完毕就收到了中断信号，导致子进程中断退出。解决方法是添加了一个子进程就绪机制，当两个子进程都向父进程发送信号报告就绪后，父进程才可以向子进程发送信号。

b) 死循环

接上文添加子进程就绪机制后，程序仍然无法运行。原因在于两个子进程可能（基本上是一定）会同时向父进程发送就绪信号，在这种情况下父进程只会响应一个就绪信号，导致父进程无法得知两个子进程都已经就绪。解决方法是在两个子进程 kill() 以及父进程信号处理函数 ParentHandler() 前后添加锁，使他们互斥，避免同时发送信号以及在处理信号时被打断；并且两个子进程应该分别发送不同信号（子 1: SIGUSR1 子 2: SIGUSR2）。

总结：

1. 需要保证在发送信号前接收信号方已经完成 signal 处理函数的注册。
2. 一个进程同时接收到两个相同信号，可能会忽略一个信号。一定避免同时发送信号，特别是相同的信号！
3. 正在调用信号处理函数时又接收到另一个信号，分为两种情况：相同信号、不同信号，若是相同信号，则会继续当前函数执行，该信号会自动被储存而不会中断信号处理函数的执行，直到信号处理函数执行完毕再重新调用相应的处理函数；若是不同信号则会直接中断当前执行信号处理函数。

** 源码

```
// FWM
// 
// 2023.4.23 ON Ubuntu LTS 18

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

#define BUFSIZE 100

int RDY = 0;
int filedes[2];
char outpipe[BUFSIZE], inpipe[BUFSIZE];

void ParentHandler()
{
    // 互斥锁，避免在处理信号时候被打断
    lockf(1, 1, 0);
    // 子进程 READY 信号
    RDY++;
    lockf(1, 0, 0);
}

void Child1Handler()
{
    sprintf(inpipe, "Child 1 is sending a message!\n");
    write(filedes[1], inpipe, sizeof(inpipe));
}

void Child2Handler()
{

```

```

    sprintf(inpipe, "Child 2 is sending a message!\n");
    write(filedes[1], inpipe, sizeof(inpipe));
}

void main()
{
    int pid_1, pid_2;

    // 创建管道
    if (pipe(filedes))
    {
        printf("Failed to create pipe!\n");
        exit(1);
    }

    // 设置文件描述符为非阻塞模式
    // read 就不会在读取完数据后阻塞
    int flags = fcntl(filedes[0], F_GETFL, 0);
    fcntl(filedes[0], F_SETFL, flags | O_NONBLOCK);

    /*
    父进程注册信号处理函数
    SIGUSR1 子进程 1 准备完成
    SIGUSR2 子进程 2 准备完成

    使用两个不同的信号
    避免在连续接受相同信号时忽略处理
    */
    signal(SIGUSR1, ParentHandler);
    signal(SIGUSR2, ParentHandler);

    // 创建子进程
    while ((pid_1 = fork()) == -1);
    if (pid_1 != 0)
        while ((pid_2 = fork()) == -1);

    switch (pid_1)
    {
    case 0:
        // ----- 子进程 1 -----
        signal(SIGUSR2, Child1Handler);
        // 互斥锁，避免两个子进程同时发生信号
        lockf(1, 1, 0);
        // 告诉父亲自己已准备完毕

```



```

kill(getppid(), SIGUSR1);
lockf(1, 0, 0);
// 等待父亲信号
pause();
exit(0);

default:
// ----- 父进程 -----
if (pid_2 != 0)
{
    // 等待子进程就绪
    while (RDY != 2);
    // 子进程就绪, 发写信号
    kill(pid_1, SIGUSR2);
    // 等待子进程 1 完成信号
    wait(0);
    // 向子进程 2 发写信号
    kill(pid_2, SIGUSR2);
    // 等待子进程 2 完成信号
    wait(0);
    // 读取管道
    close(filedes[1]); // 关闭管道写端
    while (read(filedes[0], outpipe, sizeof(inpip)) > 0)
        printf("%s", outpipe);
    close(filedes[0]);
    exit(0);
}

// ----- 子进程 2 -----
signal(SIGUSR2, Child2Handler);
// 互斥锁, 避免两个子进程同时发生信号
lockf(1, 1, 0);
kill(getppid(), SIGUSR2);
lockf(1, 0, 0);
pause();
exit(0);
}
}

```



```

// Server.c
// FWM
// 
// 2023.4.23 ON Ubuntu LTS 18

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <signal.h>

#define MSGKEY 75 // 信息队列描述符

// 消息结构
struct msgform
{
    long mtype; // 消息类型，必须为 long 类型
    pid_t PID; // client 的 PID，同步需要
    char mtext[1000]; // 消息正文
} msg;

int msgqid;
pid_t cli_pid;
bool flag = 1;

void Rcv() { flag = 0; }

void server()
{
    // 创建 75#消息队列，权限 0777
    msgqid = msgget(MSGKEY, 0777 | IPC_CREAT);

    // 接收 client 的 PID 消息
    msgrcv(msgqid, &msg, 1024, 9999, 0);
    cli_pid = msg.PID;
    printf("client_pid 为: %d\n", cli_pid);

    // 发送 PID
    msg.mtype = 9999;
    msg.PID = getpid();
    msgsnd(msgqid, &msg, 1024, 0);
}

```

```

do
{
    while (flag);
    flag = 1;
    // 接收消息
    msgrcv(msgqid, &msg, 1024, 0, 0);
    // 输出
    printf("(server)received\n");
    // 告知 client 可以发送
    kill(cli_pid, SIGUSR1);
} while (msg.mtype != 1);

// 删除消息队列, 归还资源
msgctl(msgqid, IPC_RMID, 0);
exit(0);
}

void main()
{
    signal(SIGUSR1, Rcv);
    server();
}

// Client.c
// FWM
// 
// 2023.4.23 ON Ubuntu LTS 18

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <signal.h>

#define MSGKEY 75

struct msgform
{
    long mtype;

```

```

    pid_t PID;
    char mtext[1000];
} msg;

int msgqid;
bool flag = 1;
pid_t ser_pid;

void Snd() { flag = 0; }

void client()
{
    // 打开 75#消息队列
    msgqid = msgget(MSGKEY, 0777);

    // 发送自己的 PID
    msg.mtype = 9999;
    msg.PID = getpid();
    msgsnd(msgqid, &msg, 1024, 0);

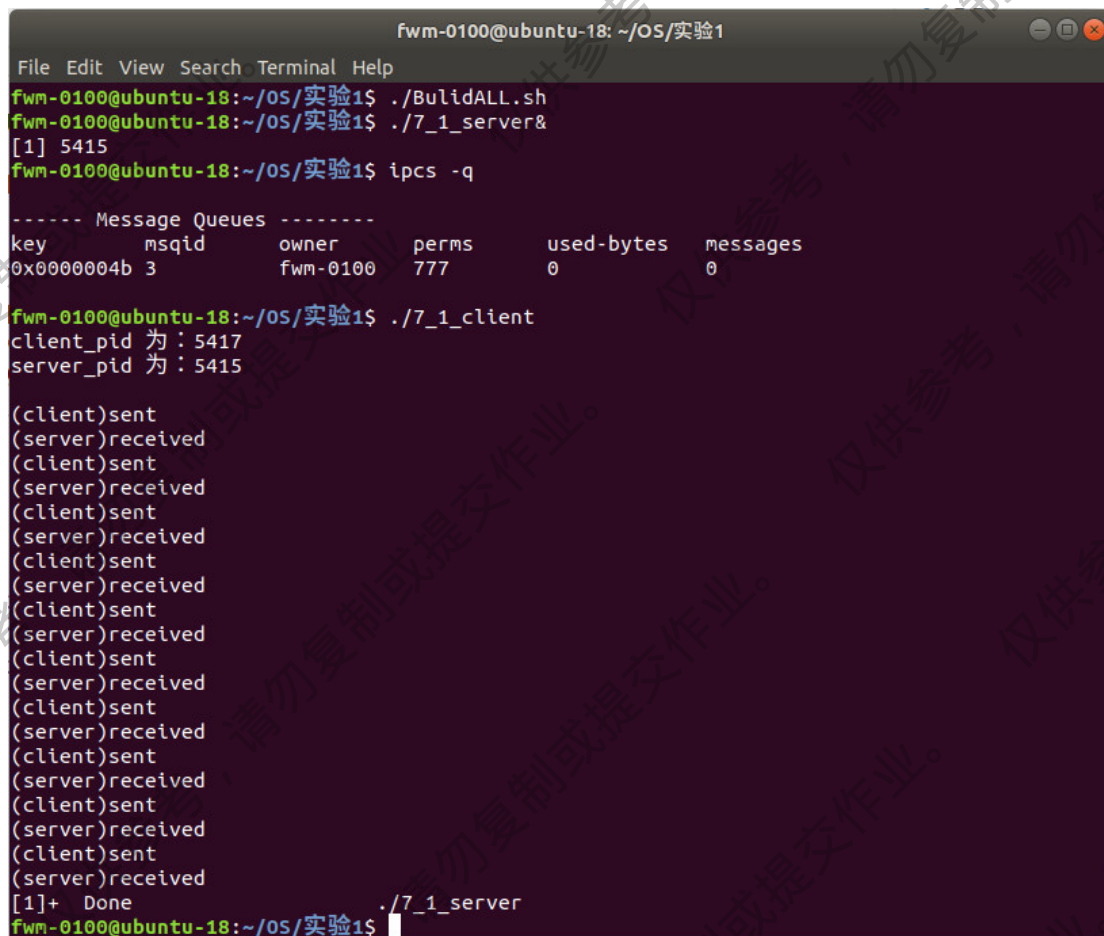
    // 接收
    msgrcv(msgqid, &msg, 1024, 9999, 0);
    ser_pid = msg.PID;
    printf("server_pid 为: %d\n\n", ser_pid);

    for (int i = 10; i >= 1; i--)
    {
        msg.mtype = i;
        printf("(client)sent\n");
        // 发送消息
        msgsnd(msgqid, &msg, 1024, 0);
        kill(ser_pid, SIGUSR1);
        // 等待 server 读取完毕
        while (flag);
        // 复位 flag
        flag = 1;
    }
    exit(0);
}

void main()
{
    signal(SIGUSR1, Snd);
    client();
}

```


运行结果截图



```
fwm-0100@ubuntu-18: ~/OS/实验1
File Edit View Search Terminal Help
fwm-0100@ubuntu-18:~/OS/实验1$ ./BulidALL.sh
fwm-0100@ubuntu-18:~/OS/实验1$ ./7_1_server&
[1] 5415
fwm-0100@ubuntu-18:~/OS/实验1$ ipcs -q

----- Message Queues -----
key          msqid        owner        perms        used-bytes   messages
0x0000004b  3            fwm-0100    777          0            0

fwm-0100@ubuntu-18:~/OS/实验1$ ./7_1_client
client_pid 为 : 5417
server_pid 为 : 5415

(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
(client)sent
(server)received
[1]+  Done                  ./7_1_server
fwm-0100@ubuntu-18:~/OS/实验1$
```

(八) 共享存储区通信

1. 比较两种消息通信机制中数据传输的时间

两种通信机制各具优点,各自拥有不同的应用场景,故难以直接对比数据传输时间,下对两种机制的性能进行对比。

1) 建立阶段

消息队列机制的建立仅仅是软件上的设定,但是共享存储区需要硬件操作,比较复杂。相比之下,消息队列的建立消耗资源更少。

2) 数据传输阶段

共享存储区的数据传输只是将数据写入内存,拥有硬件的支持、速度更快、开销更小。但是消息队列需要通过系统软件控制才可以实现发送、接收消息,需要消耗 CPU 资源。相比之下,共享存储区的数据传输拥有开销小、速度快的优势。

3) 同步机制

消息队列自带了同步控制的机制,当消息队列已满,发送方可以进入休眠状态避免浪费 CPU 资源;同理当消息队列为空,接收方也会休眠避免浪费资源。

共享存储区则截然不同，共享存储区需要程序员自行设置读写同步、访问写入互斥的操作，如果没有设计良好的同步机制将大量浪费 CPU 在询问是否可以写入、是否有新数据到来。

综上，两种通信机制各有千秋，在小数据量、频繁操作的场景采用消息队列，在大数据量、频繁访问的场景采用共享存储区。