**实验三 进程间通信**

# 姓 名 吴梓弘 学 号 21013054 成绩

实验时间 11.9 指导教师(签名)

**（诚信声明：本实验报告内容，均由本人亲自上机完成。 签名：\_\_\_\_\_\_\_\_\_\_\_\_\_）**

一．实验目的

1.掌握利用管道机制实现进程间的通信的方法

2.掌握利用消息缓冲队列机制实现进程间的通信的方法

3.掌握利用共享存储区机制实现进程间的通信的方法

4.加深对上述三种通信机制的理解。

二．实验工具与设备

已安装 Linux 操作系统的计算机。

三．实验内容

1. **掌握实现进程间通信的系统调用的功能和方法**
2. **编写一段程序，实现进程间的管道通信。 其中，父进程通过管道向子进程发送一个字符串（子进程的进程号），子进程将它显示出来。**

**代码：**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

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

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

**int main() {**

**int pipe\_fd[2];**

**pid\_t child\_pid;**

**char buffer[32];**

**if (pipe(pipe\_fd) == -1) {**

**perror("Pipe creation failed");**

**return 1;**

**}**

**child\_pid = fork();**

**if (child\_pid == -1) {**

**perror("Fork failed");**

**return 1;**

**}**

**if (child\_pid == 0) {**

**// 子进程**

**close(pipe\_fd[1]); // 关闭子进程的写入端**

**// 从管道中读取数据并显示**

**read(pipe\_fd[0], buffer, sizeof(buffer));**

**printf("子进程PID: %s\n", buffer);**

**close(pipe\_fd[0]);**

**} else {**

**// 父进程**

**close(pipe\_fd[0]); // 关闭父进程的读取端**

**char child\_pid\_str[10];**

**sprintf(child\_pid\_str, "%d", child\_pid);**

**// 向管道中写入子进程的进程号**

**write(pipe\_fd[1], child\_pid\_str, sizeof(child\_pid\_str));**

**close(pipe\_fd[1]);**

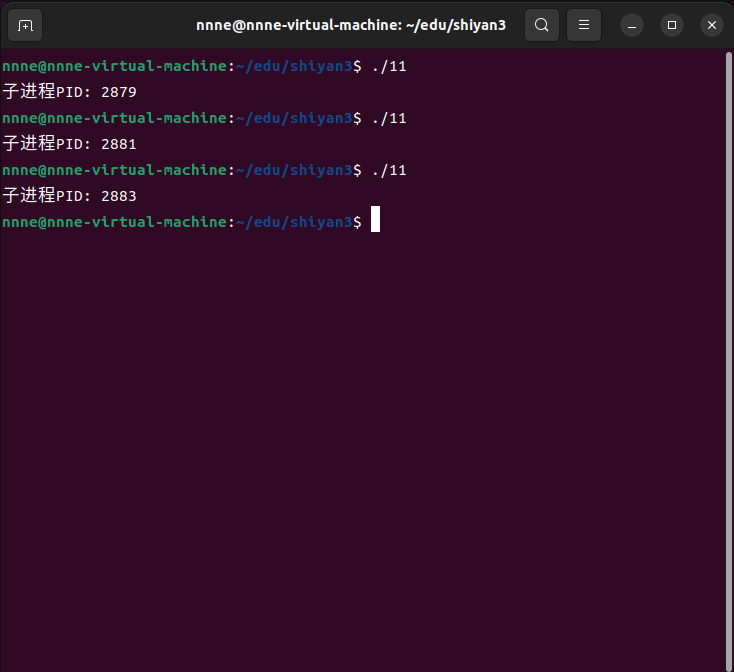
**// 等待子进程完成**

**wait(NULL);**

**}**

**return 0;**

**}运行结果：**

**。、**

1. **编写一段程序，使用消息缓冲队列来实现 client 进程和 server 进程之间的通信。**

**实现代码：**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/wait.h>

#define MSGKEY 75

// 消息结构

struct message {

long mtype; // 消息类型

char mtext[100]; // 消息内容

};

void server\_process() {

int msgid;

struct message msg;

// 创建消息队列

msgid = msgget(MSGKEY, IPC\_CREAT | 0666);

if (msgid == -1) {

perror("Failed to create message queue");

exit(1);

}

while (1) {

msgrcv(msgid, &msg, sizeof(msg.mtext), 0, 0);

if (msg.mtype == 1) {

printf("Server has received an end signal from Client!\n");

// 删除消息队列

msgctl(msgid, IPC\_RMID, NULL);

break;

} else {

printf("Server has received message from Client: %s\n", msg.mtext);

}

}

}

void client\_process() {

int msgid;

struct message msg;

// 获取消息队列

msgid = msgget(MSGKEY, 0);

if (msgid == -1) {

perror("Failed to get message queue");

exit(1);

}

for (int i = 10; i >= 1; i--) {

snprintf(msg.mtext, sizeof(msg.mtext), "Message %d", i);

msg.mtype = i;

msgsnd(msgid, &msg, sizeof(msg.mtext), 0);

printf("Client has sent a message to Server: %s\n", msg.mtext);

}

// 发送结束信号

snprintf(msg.mtext, sizeof(msg.mtext), "End signal from Client");

msg.mtype = 1;

msgsnd(msgid, &msg, sizeof(msg.mtext), 0);

printf("Client has sent an end signal to Server!\n");

}

int main() {

pid\_t server\_pid, client\_pid;

// 创建 server 进程

server\_pid = fork();

if (server\_pid == 0) {

server\_process();

exit(0);

} else if (server\_pid < 0) {

perror("Fork server process failed");

exit(1);

}

// 创建 client 进程

client\_pid = fork();

if (client\_pid == 0) {

client\_process();

exit(0);

} else if (client\_pid < 0) {

perror("Fork client process failed");

exit(1);

}

// 等待子进程结束

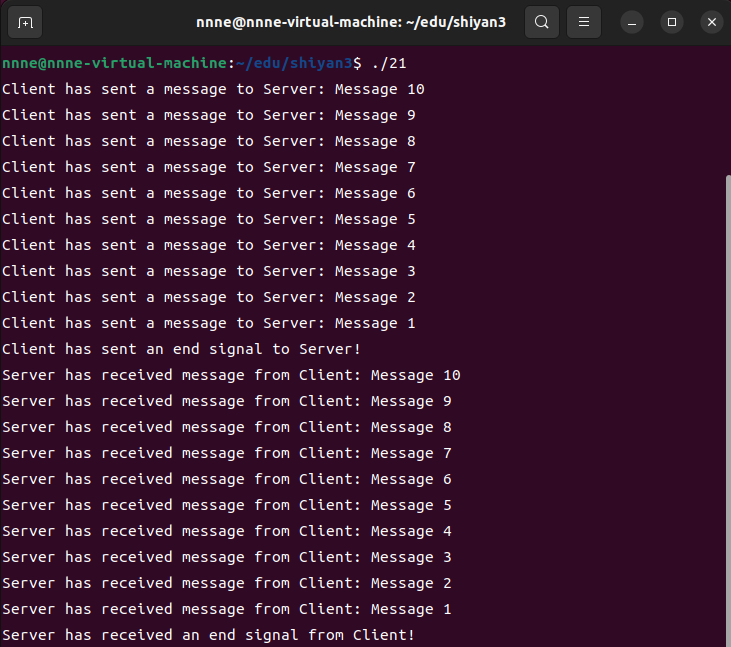
waitpid(server\_pid, NULL, 0);

waitpid(client\_pid, NULL, 0);

// 所有子进程结束后，父进程结束

return 0;

}**运行结果：**



1. **编写一个与上述功能相同的程序，使其用共享存储区来实现两个进程之间的通信。**

**实现代码：**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/shm.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <time.h>

#define SHMKEY 75

int \*shm;

void serverProcess() {

int shmid;

// 创建共享存储区

shmid = shmget(SHMKEY, sizeof(int), IPC\_CREAT | 0666);

if (shmid == -1) {

perror("shmget");

exit(1);

}

// 附加共享存储区

shm = (int \*)shmat(shmid, NULL, 0);

if (shm == (int \*)-1) {

perror("shmat");

exit(1);

}

// 初始化共享存储区的第一个字节为-1，表示服务器空闲

\*shm = -1;

while (1) {

// 等待客户端发送请求

while (\*shm == -1);

int message = \*shm;

//如果受到的值为0,作为结束信号退出

if(message == 0){

printf("Server has quited!\n");

\*shm = -1;

break;

}

printf("Server has received message from Client: %d\n", message);

// 处理完消息后，将共享存储区的值重新置为-1

\*shm = -1;

}

// 分离共享存储区

shmdt(shm);

// 删除共享存储区

shmctl(shmid, IPC\_RMID, NULL);

exit(0);

}

void clientProcess() {

int shmid;

int \*server\_shm;

// 获取共享存储区

shmid = shmget(SHMKEY, sizeof(int), 0666);

if (shmid == -1) {

perror("shmget");

exit(1);

}

// 附加共享存储区

server\_shm = (int \*)shmat(shmid, NULL, 0);

if (server\_shm == (int \*)-1) {

perror("shmat");

exit(1);

}

while(1){

while(\*server\_shm != -1);

sleep(1);

srand(time(NULL));

int message = rand() % 10;

\*server\_shm = message;

printf("Client has set message to Server: %d\n",message);

if(message == 0)break;

}

// 分离共享存储区

shmdt(server\_shm);

exit(0);

}

int main() {

pid\_t server\_pid, client\_pid;

server\_pid = fork();

if (server\_pid < 0) {

perror("Server fork failed");

exit(1);

} else if (server\_pid == 0) {

// 子进程为服务器进程

serverProcess();

} else {

client\_pid = fork();

if (client\_pid < 0) {

perror("Client fork failed");

exit(1);

} else if (client\_pid == 0) {

// 子进程为客户端进程

clientProcess();

} else {

// 等待子进程结束

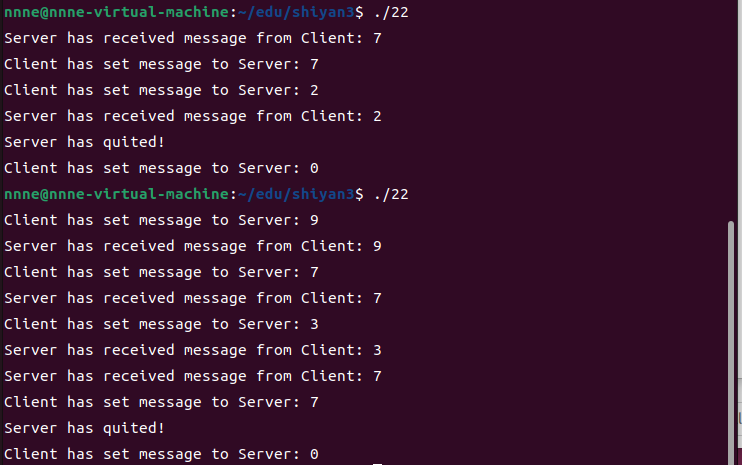
wait(NULL);

}

}

return 0;

}运行结果：



# 四．思考题

1. **上述哪些通信机制提供了发送进程和接收进程之间的同步功能？这些同步是如何进行的？**

在上述的程序中，消息队列提供了发送进程和接收进程之间的同步功能。消息队列的发送和接收操作都是阻塞的，这导致发送进程在发送消息时会等待接收进程接收，而接收进程在没有可接收的消息时会等待发送进程发送消息。这种阻塞特性实现了进程间的同步。

以下是消息队列提供同步的方式：

1、发送阻塞： 当消息队列满了（达到了系统限定的最大消息数量）时，发送进程将被阻塞，直到有足够的空间来存储消息。这确保了发送进程不会一直发送消息，而没有接收进程处理。

2、接收阻塞： 当消息队列为空时，接收进程将被阻塞，直到有消息被发送到队列中。这确保了接收进程只在有可接收的消息时才会进行接收操作。

通过这种方式，消息队列实现了发送和接收进程之间的同步，确保了消息的可靠传递。这对于协调进程之间的操作很重要，以确保在消息发送和接收之间的正确顺序。

1. **上述通信机制各有什么特点，它们分别适合于何种场合？**

在上述的程序中，涉及了两种不同的进程间通信机制：消息队列和共享内存。以下是它们的特点以及适用场合：

1. 消息队列

特点：

异步通信： 消息队列是一种异步通信机制，发送和接收进程不需要同时执行。

数据结构： 可以发送不同结构的消息，消息的类型可以是整数或其他数据类型，适用于复杂的通信需求。

消息缓冲： 具有消息缓冲的特性，可以缓存一定数量的消息，允许发送者发送消息而不需要等待接收者立即读取。

适用场合：

松散耦合： 适用于松散耦合的进程，因为发送和接收是异步的，进程之间不需要等待对方完成操作。

复杂数据结构： 适用于需要传递复杂数据结构的情况，如消息中包含结构体或其他数据类型。

二、缓冲需求

特点：

同步通信： 共享内存是一种同步通信机制，需要使用额外的同步手段（如信号量）来协调进程的访问。

直接访问： 进程可以直接访问共享内存区域，速度相对较快，适用于频繁的数据共享。

简单数据结构：通常用于共享简单的数据结构，如数组或结构体。

适用场合：

高频率数据交换： 适用于需要高频率进行数据交换的进程，因为共享内存的直接访问速度相对较快。

同步需求： 适用于需要同步操作的场合，需要使用额外的同步手段确保多个进程之间的数据一致性。

简单数据结构： 适用于共享简单的数据结构，如需要在多个进程之间传递数组或结构体等。

总的来说，选择消息队列还是共享内存取决于具体的应用场景和通信需求。如果需要异步通信、传递复杂数据结构或者具有缓冲需求，消息队列是一个不错的选择。而如果需要同步通信、高频率数据交换，且可以通过直接访问共享内存提高速度，那么共享内存是更合适的选择。