**操作系统课程设计实验报告**

实验题目：

姓    名：金浅予     项宇琦       廖越强 林文渊

学    号：18271120 18113111 18041618 18011317

专    业： 网络工程   信息安全   网络工程   网络工程

班    级： 18272411 18273611 18272412  18272411

老师姓名：                   张桢

日    期：       2020   年  6     月  11      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....................................................... 1

四 核心代码及实验结果展示.............................................. 1

4.1 实验小组分工........................................................... 1

4.2 核心代码及实验结果............................................. 1

五 个人实验改进与总结....................................................... 1

5.1 个人实验改进........................................................... 1

5.2 个人实验总结........................................................... 2

六 参考文献............................................................................... 2

(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

1. 设计目的

1. 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。
2. 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。
3. 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。
4. 请根据自身情况，进一步阅读分析相关系统调用的内核源码实现。

2. 设计内容

1. 实现一个模拟的 shell 编写三个不同的程序 cmd1.c，cmd2.c，cmd3.c，每个程序的功能自定，分别编译成可执 行文件 cmd1，cmd2，cmd3。然后再编写一个程序，模拟 shell 程序的功能：能根据用户输 入的字符串（表示相应的命令名），为相应的命令创建子进程并让它去执行相应的程序，而 父进程则等待子进程结束，然后再等待接收下一条命令。如果接收到的命令为 exit，则父进 程结束，退出模拟 shell；如果接收到的命令是无效命令，则显示“Command not found”，继 续等待输入下一条命令。
2. 实现一个管道通信程序 由父进程创建一个管道，然后再创建 3 个子进程，并由这三个子进程利用管道与父进程 之间进行通信：子进程发送信息，父进程等三个子进程全部发完消息后再接收信息。通信的 具体内容可根据自己的需要随意设计，要求能试验阻塞型读写过程中的各种情况，测试管道 的默认大小，并且要求利用 Posix 信号量机制实现进程间对管道的互斥访问。运行程序，观 察各种情况下，进程实际读写的字节数以及进程阻塞唤醒的情况。
3. 利用 Linux 的消息队列通信机制实现两个线程间的通信 编写程序创建三个线程：sender1 线程、sender2 线程和 receive 线程，三个线程的功能 描述如下： ①sender1 线程：运行函数 sender1()，它创建一个消息队列，然后等待用户通过终端输 入一串字符，并将这串字符通过消息队列发送给 receiver 线程；可循环发送多个消息，直到 用户输入“exit”为止，表示它不再发送消息，最后向 receiver 线程发送消息“end1”，并且 等待 receiver 的应答，等到应答消息后，将接收到的应答信息显示在终端屏幕上，结束线程 的运行。 ②sender2 线程：运行函数 sender2()，共享 sender1 创建的消息队列，等待用户通过终 端输入一串字符，并将这串字符通过消息队列发送给 receiver 线程；可循环发送多个消息， 直到用户输入“exit”为止，表示它不再发送消息，最后向 receiver 线程发送消息“end2”， 并且等待 receiver 的应答，等到应答消息后，将接收到的应答信息显示在终端屏幕上，结束 线程的运行。 ③Receiver 线程：运行函数 receive()，它通过消息队列接收来自 sender1 和 sender2 两 个线程的消息，将消息显示在终端屏幕上，当收到内容为“end1”的消息时，就向 sender1 发送一个应答消息“over1”； 当收到内容为“end2”的消息时，就向 sender2 发送一个应 答消息“over2”；消息接收完成后删除消息队列，结束线程的运行。选择合适的信号量机制 实现三个线程之间的同步与互斥。
4. 利用 Linux 的共享内存通信机制实现两个进程间的通信 编写程序 sender，它创建一个共享内存，然后等待用户通过终端输入一串字符，并将这 串字符通过共享内存发送给 receiver；最后，它等待 receiver 的应答，收到应答消息后，将 接收到的应答信息显示在终端屏幕上，删除共享内存，结束程序的运行。编写 receiver 程序， 它通过共享内存接收来自 sender 的消息，将消息显示在终端屏幕上，然后再通过该共享内 存向 sender 发送一个应答消息“over”，结束程序的运行。选择合适的信号量机制实现两个 进程对共享内存的互斥及同步使用。

（大家注意，正文部分格式我已设定好，大家不要改，按这个格式书写就可以了）

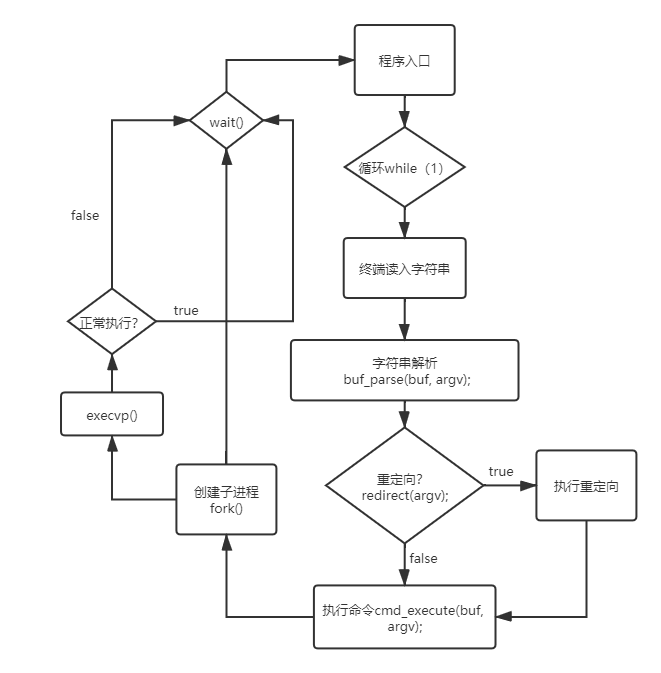
**（正文部分一律用 宋体，五号字，1.25倍间距，首行缩进两格）**

**（根据实验设计指导书，描述实验题目内容，考察的知识点，问题的关键点）**

# 二 实验思路

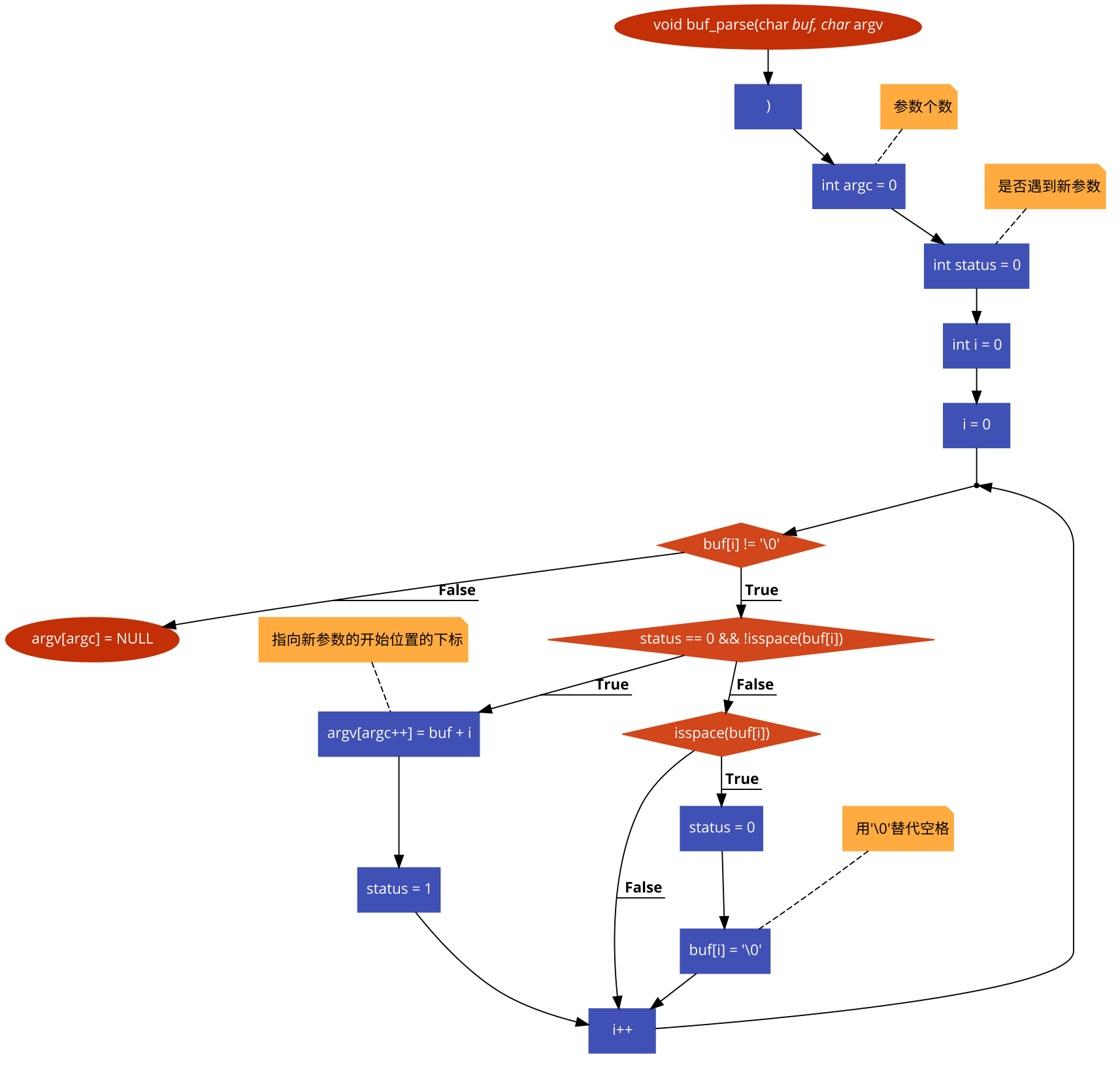
## 实现一个模拟的 shell

### 图1.整体流程图

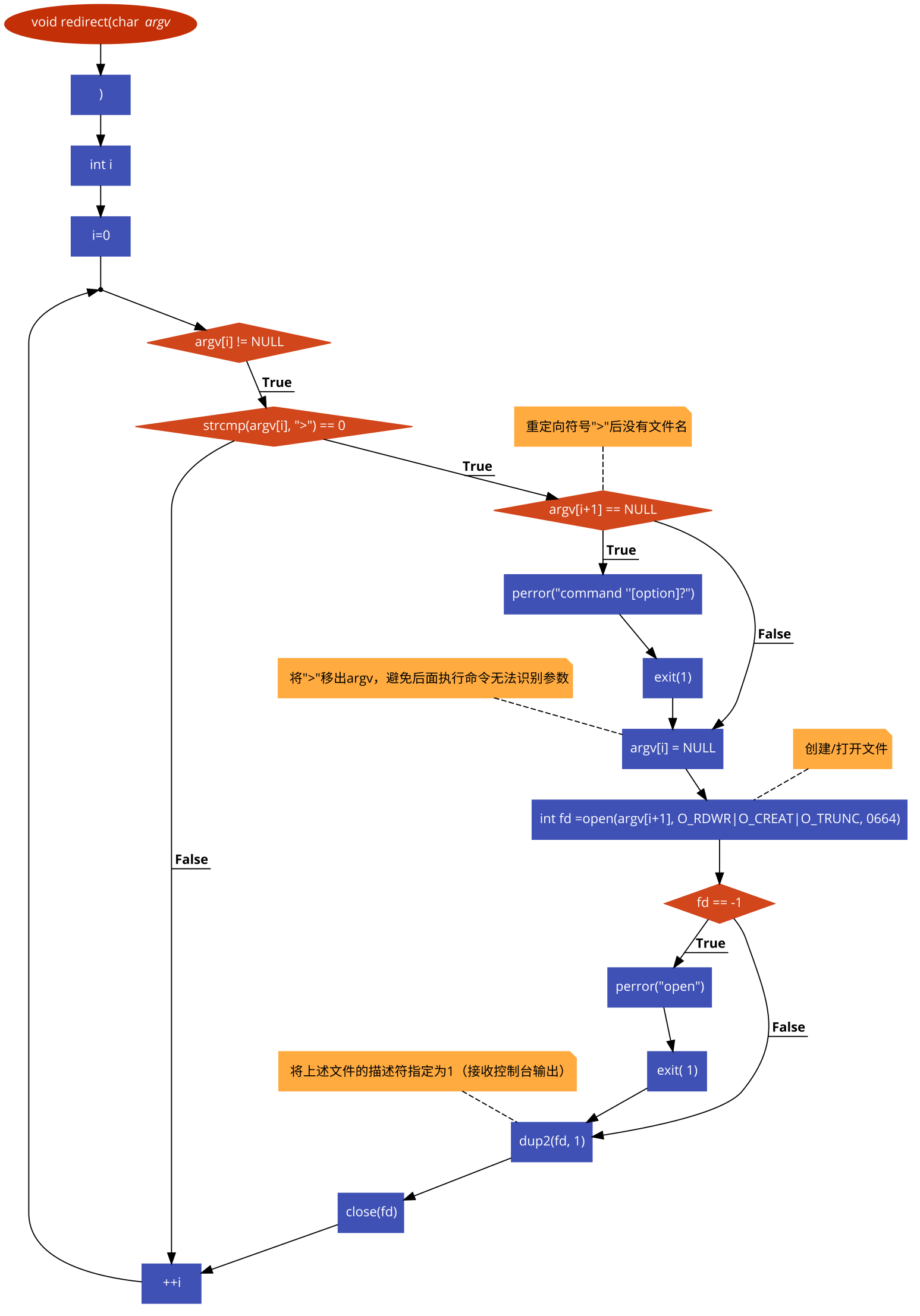


1. 将终端输入存放在一个字符串变量buf中
2. 对字符串变量进行解析，分离出字符串中的程序名和参数并分别存放起来
3. 判断是否包含重定向
4. 创建子进程，再使用exec系列函数对子进程进行替换，来执行对应的程序，父进程处于wait状态，直到子进程执行完后才继续执行

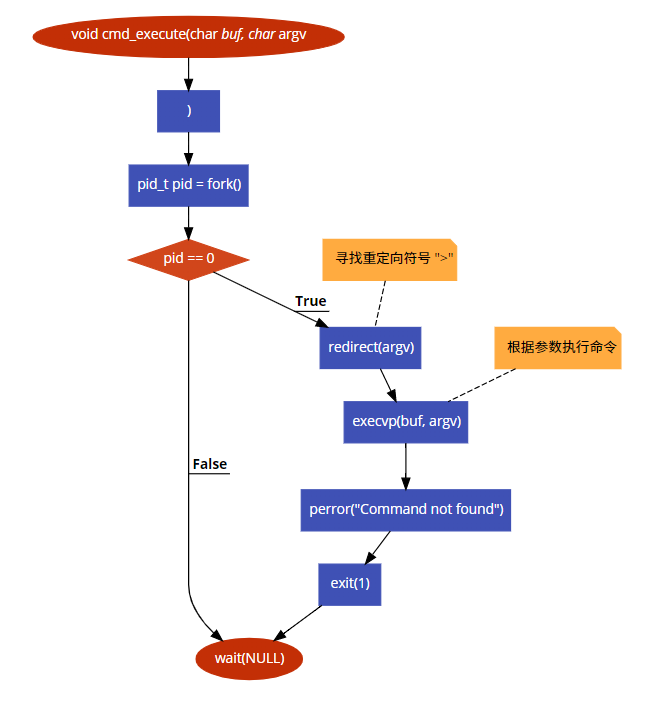
### 图1.参数提取buf\_parse()



### 图2.重定向redirect()

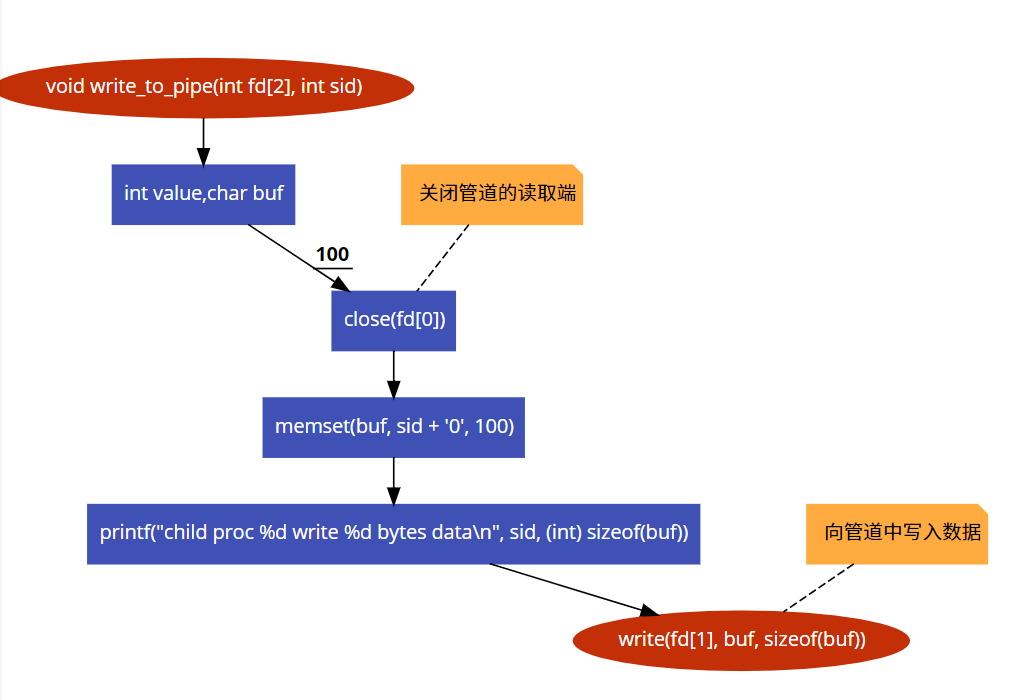


### 图3.执行命令cmd\_execute()



## 实现一个管道通信程序

### 图1.write\_to\_pipe()



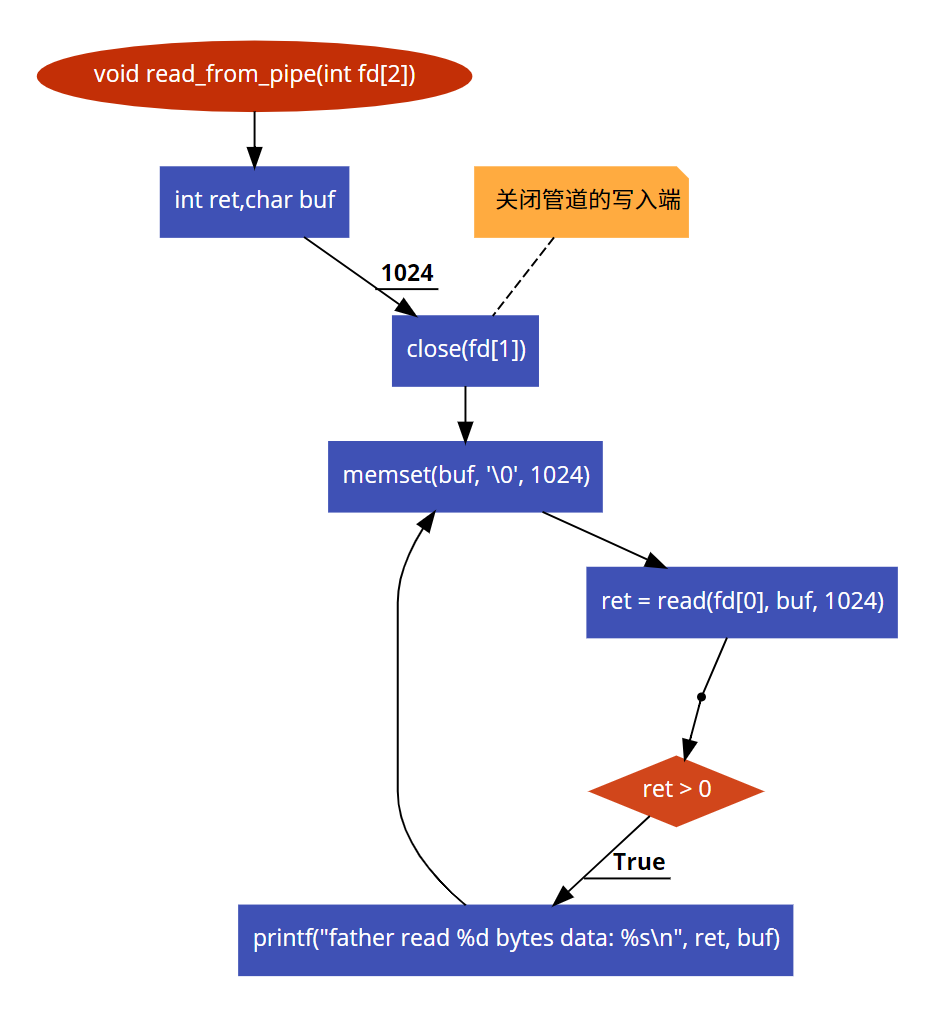
读取端fd[0]，写入端fd[1]

关闭读管道

memset将缓存区buf初始化为sid+'0'，标注为字符输出显示buf最大空间

buf区数据写入写管道

### 图2.read\_from\_pipe()



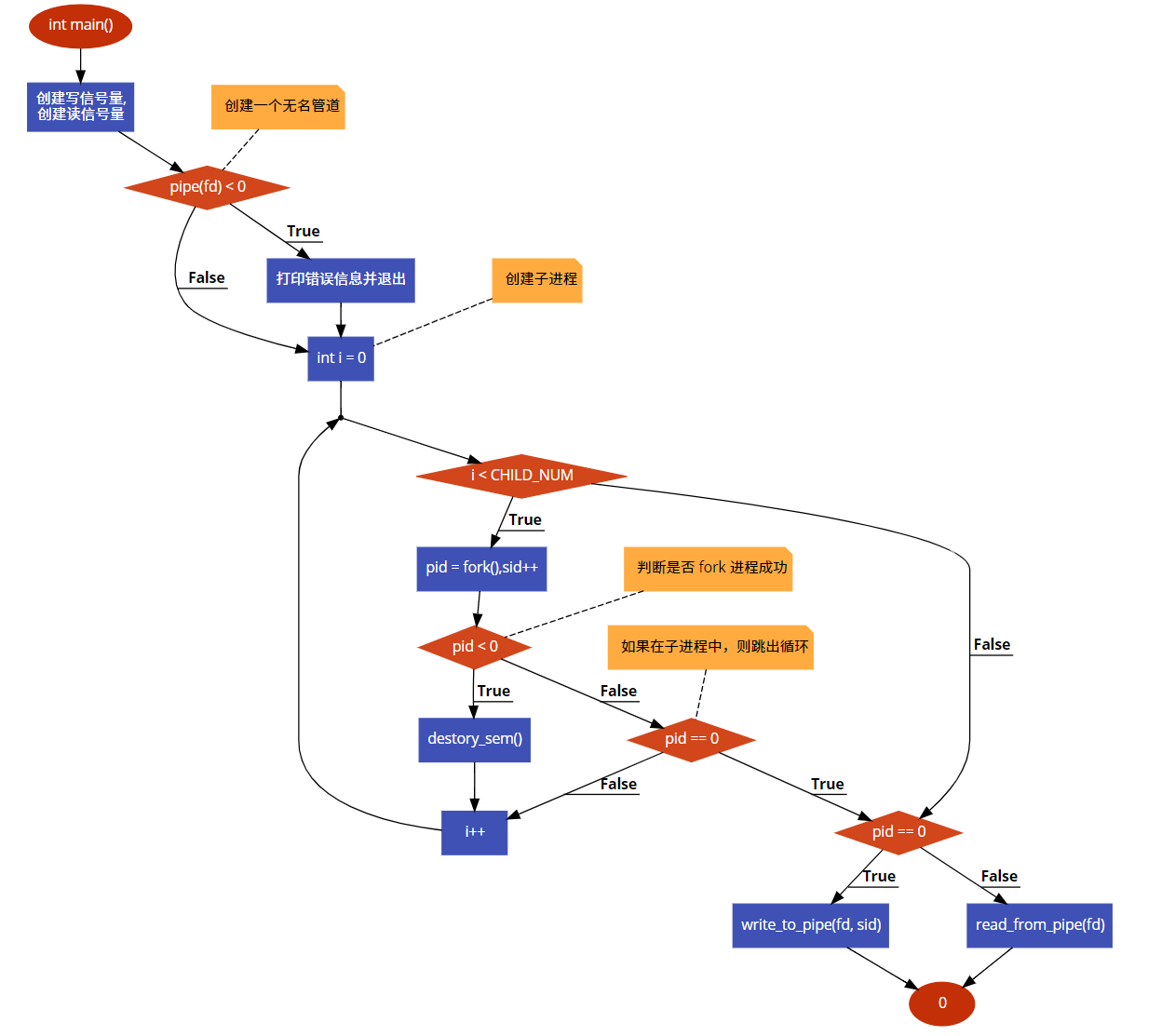
关闭写入端

buf全部填充为'\0'

读入数据

读取成功后打印信息并重置缓冲区直至失败

### 图3.主函数



创建所需信号量，利用fork创建子进程

若在子进程中则执行写函数

转至父进程后执行读函数

## 利用 Linux 的消息队列通信机制实现两个线程间的通信

### 图1.send\_thread()

设定info长度256

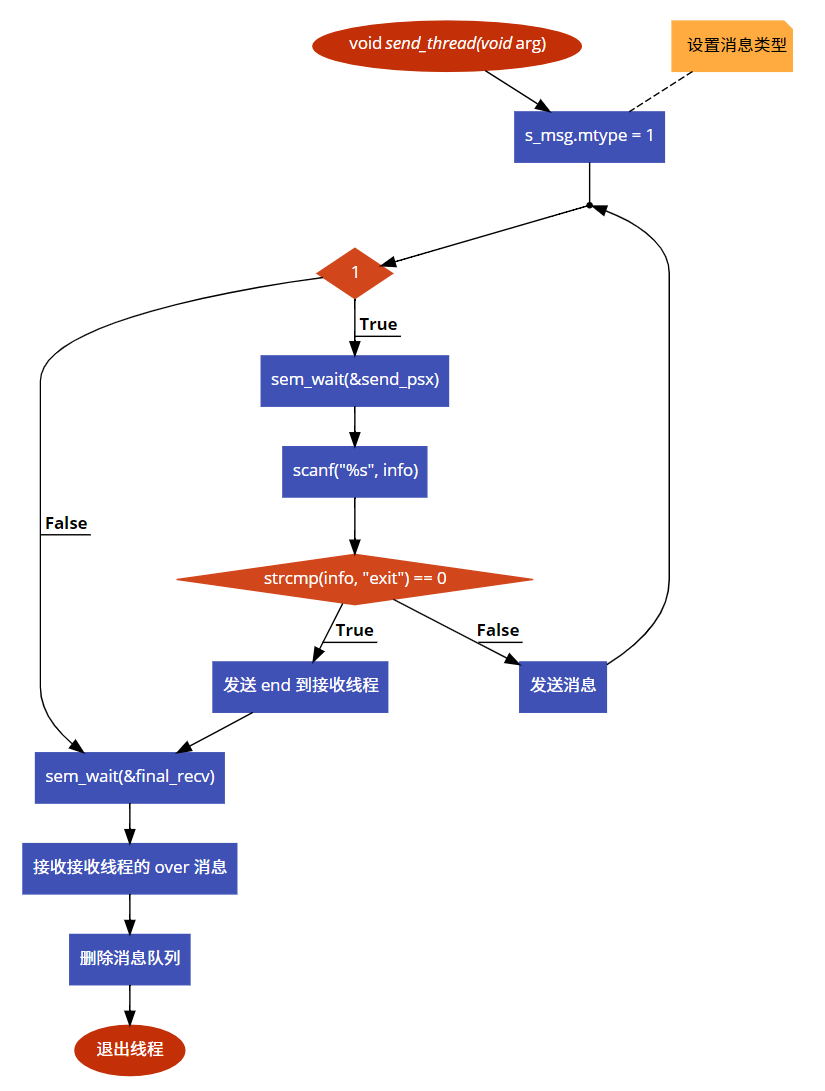
消息类型默认为正常消息

判定是否为结束消息，若不是则消息保存至消息队列并计数

exit退出循环后

final\_recv信号量-1

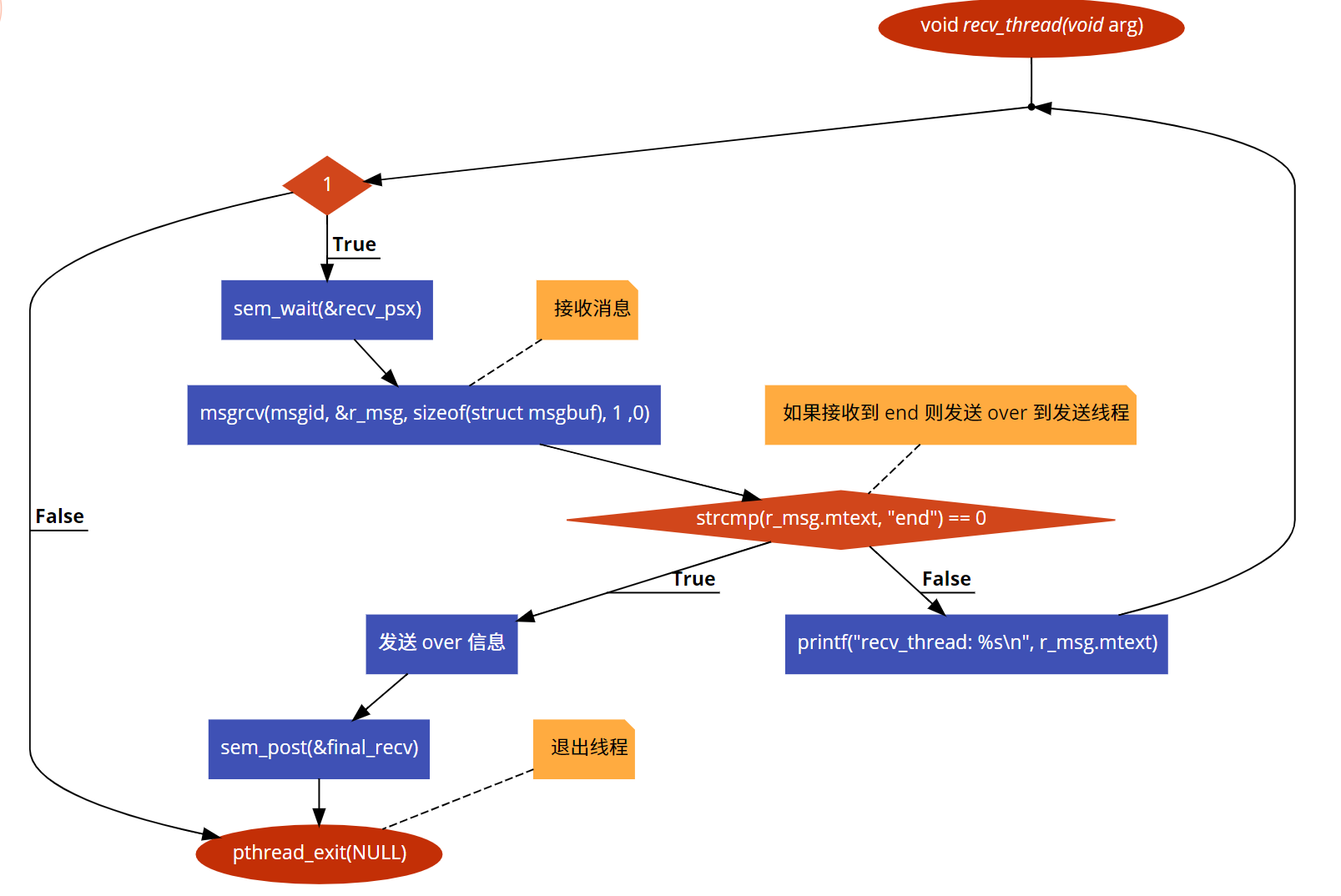
接受over信息，删除消息队列并退出



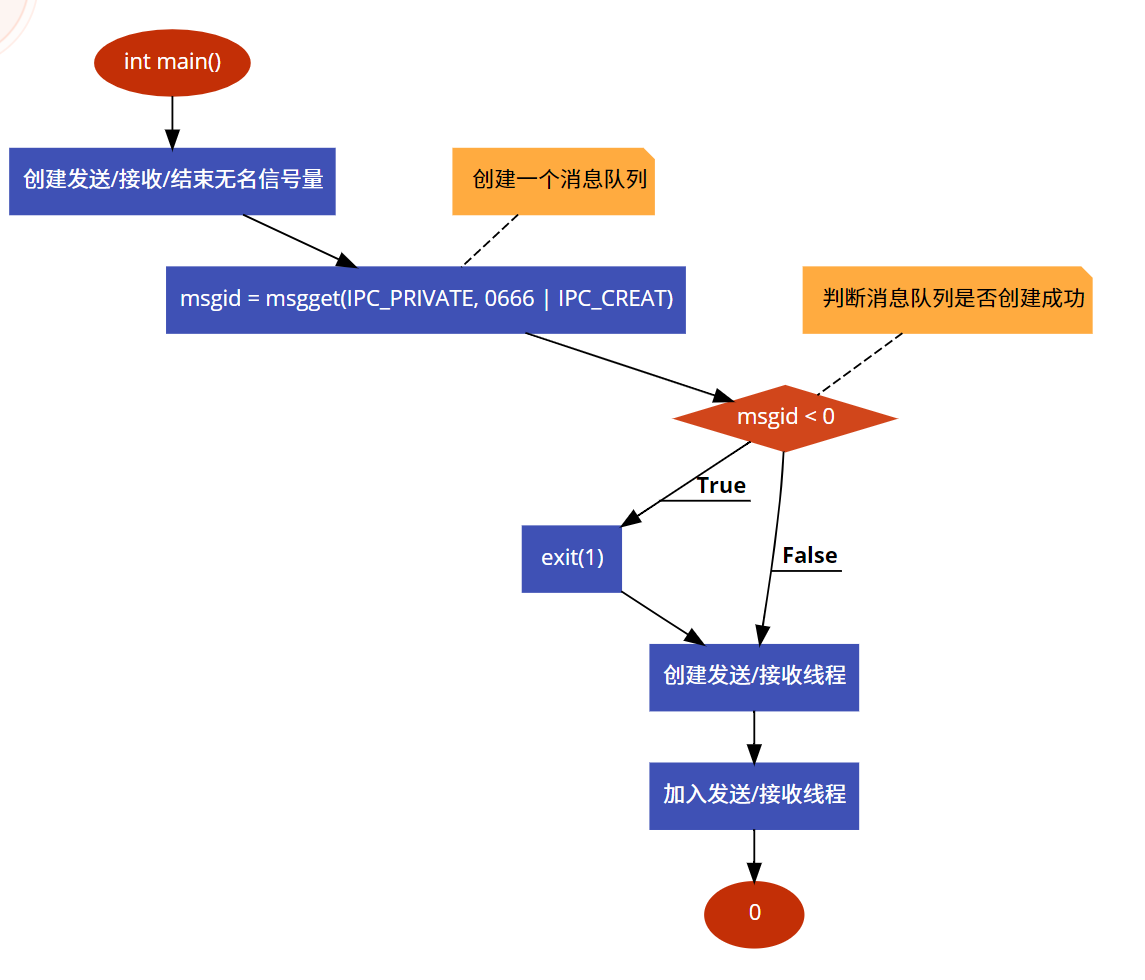
### 图2.recv\_thread()

默认消息类型为正常消息

接受到end，写入消息队列，消息类型改变则退出循环

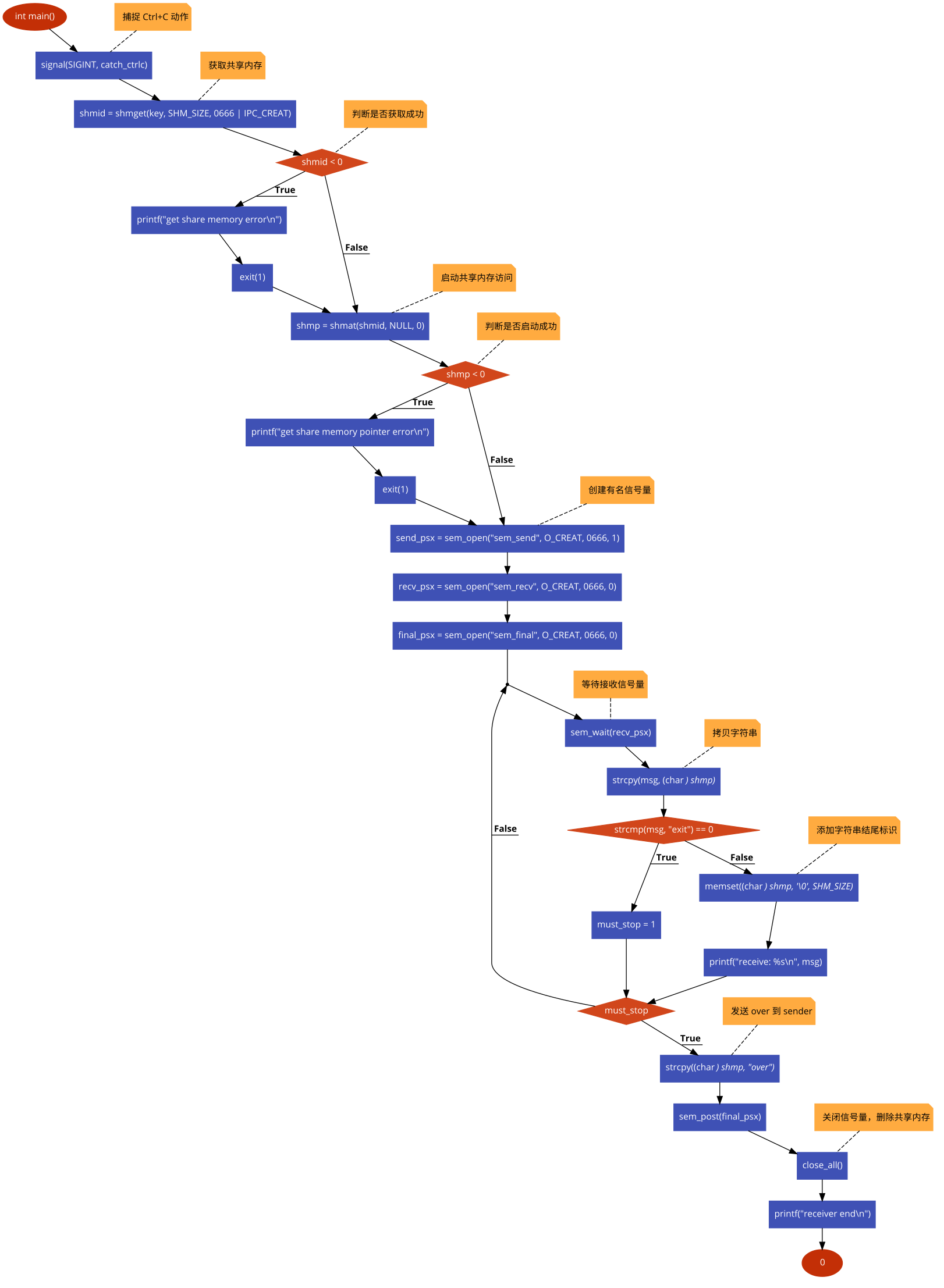


### 图3.主函数

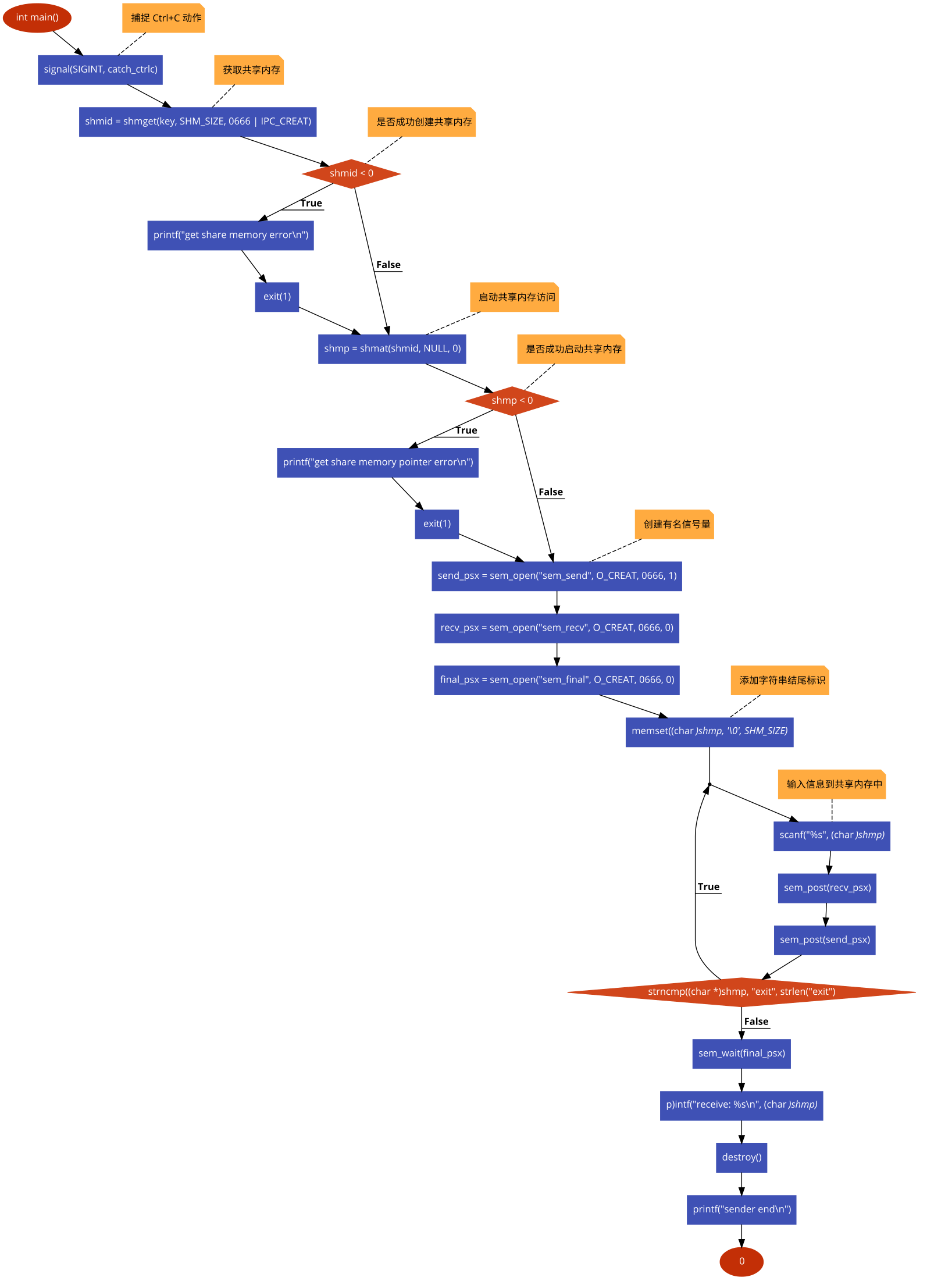


## 利用 Linux 的共享内存通信机制实现两个进程间的通信

### 图1.接收端主函数



### 图2.发送端主函数



**（再次强调，此处需要画人肉眼能清晰的流程图，不清晰的图不要放）**

**（图居中，并且下面的图注格式为： 图1. 名字 ）**

**（对于每个流程图需要配以文字详细阐述其内容，及其关键点，如实验中用到的每个信号量，为什么要用这个信号量，为什么用两个就够了，而不用三个）**

# 三 遇到问题及解决方法

**（此处请同学们务必认真规范书写，对于每个实验，写几条遇到的问题，以条目化形式呈现，如 1. 2. 3.**

**）**

关于fork()函数:

1.fork()不保证子进程和父进程的执行顺序，所以会出现父进程先于子进程结束的情况，此时该进程的退出信息无法被处理，变成了孤儿进程，操作系统会为其找到pid==1的进程接管此进程

2.当出现了fork()的嵌套时，会发生套娃事件(子进程生成子进程生成子进程...)

// 创建子进程

for (int i = 0; i < CHILD\_NUM; i++) {

//fork（）函数通过系统调用创建一个与原来进程几乎完全相同的进程

//在父进程中，fork返回新创建子进程的进程ID；在子进程中，fork返回0；如果出现错误，fork返回一个负值；

//父进程的pid指向子进程的进程id

pid = fork();

sid++;

// 判断是否 fork 进程成功

if (pid < 0) {

printf("fork error\n");

destory\_sem();

exit(0);

// 如果在子进程中，则跳出循环

//子进程没有子进程，所以pid==0

} else if (pid == 0) {

break;

}

}

我们需要使用fork()利用一个父进程来创建多个子进程，也就是 fork()如果发现其返回值是0 或者 -1 直接结束掉循环

# 四 核心代码及实验结果展示

## 4.1 实验小组分工

**（此处书写 实验1，实验2， 实验3和4 中自己完成部分）**

**对于实验3和4，书写小组分工，按如下格式：）**

**张三：负责XXXX部分**

**李四：负责XXXX部分**

金浅予:负责实验三，利用消息队列实现进程间通信

林文渊: 负责第四个实验，实现利用 Linux 的共享内存通信机制实现两个进程间的通信

廖越强: 负责第一个实验，实现一个简单的shell

项宇琦:负责实验二，实现管道通信程序

## 4.2 核心代码及实验结果

编译:gcc -o main main.c -lpthread

执行：./main

### 1.实现一个模拟的shell

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <unistd.h>

#include <sys/wait.h>

#include <fcntl.h>

void buf\_parse(char \*buf, char \*argv[]);

void cmd\_execute(char \*buf, char \*argv[]);

void redirect(char \* argv[]);

int main(){

char buf[1024] = {};

char \* argv[8] = {}; // storage arguments

while(1){

printf("shell# ");

memset(buf, 0x00, sizeof(buf));

/\* receive string \*/

while(scanf("%[^\n]%\*c", buf) == 0){ // return 0 means that user only input '\n', ignore it

printf("shell# ");

setbuf(stdin, NULL); // clear stdin(the '\n' mentioned above)

}

/\* do parse \*/

buf\_parse(buf, argv);

//printf("buf:%s", buf);

//printf("argv[0]:%s argv[1]:%s argv[2]:%s\n", argv[0], argv[1], argv[2]);

if(strcmp(buf,"exit") == 0) break;

/\* command execute \*/

cmd\_execute(buf, argv);

}

return 0;

}

void buf\_parse(char \*buf, char \*argv[]){

int argc = 0; // argument counter

int status = 0; // new argument appear

int i = 0;

for(i = 0; buf[i] != '\0'; i++){

if(status==0 && !isspace(buf[i])){

argv[argc++] = buf + i; //point to start bit of new argument in buf

status = 1;

}

else if(isspace(buf[i])){

status = 0;

buf[i] = '\0'; //replace ' ' with 0

}

}

argv[argc] = NULL;

}

void cmd\_execute(char \*buf, char \*argv[]){

pid\_t pid = fork();

if(pid == 0){

// find ">", redierct

redirect(argv);

execvp(buf, argv);

perror("Command not found");

exit(1);

}

wait(NULL);

}

void redirect(char \* argv[])

{

int i;

for(i=0; argv[i] != NULL; ++i)

{

if(strcmp(argv[i], ">") == 0)

{

printf("find > \n");

if(argv[i+1] == NULL){ // no argument after >

perror("command '>'[option]?");

exit(1);

}

argv[i] = NULL;

// reate/open file

int fd =open(argv[i+1], O\_RDWR|O\_CREAT|O\_TRUNC, 0664);

printf("open file: %s\n", argv[i+1]);

if(fd == -1){

perror("open");

exit( 1);

}

// redirct

dup2(fd, 1);

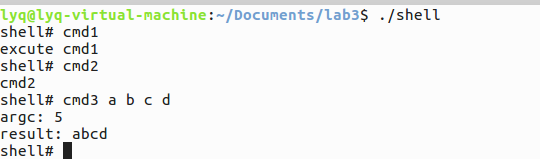
close(fd);

}

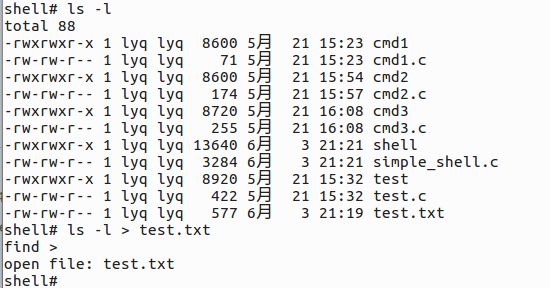
}

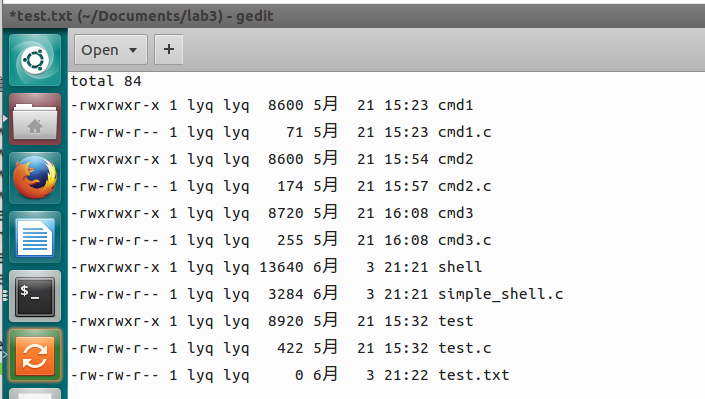
}

执行三个自己编写的简单可执行程序

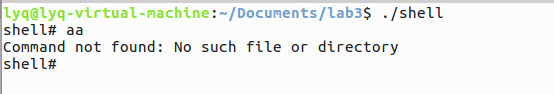


执行其他命令以及重定向





输入无效命令



退出程序



### 2.实现管道通信程序

实验代码

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <string.h>

#define CHILD\_NUM 3

void write\_to\_pipe(int fd[2], int sid);

//fd参数返回两个文件描述符,fd[0]指向管道的读端,fd[1]指向管道的写端。fd[1]的输出是fd[0]的输入。

void read\_from\_pipe(int fd[2]);//fd[2]=>一个写入端，一个读取端

// 销毁信号量

void destory\_sem() {

sem\_unlink("sem\_write");

sem\_unlink("sem\_read");

}

int main() {

int fd[2];

pid\_t pid;

int sid = 0;

sem\_t \*write\_psx, \*read\_psx;

// 创建写信号量

write\_psx = sem\_open("sem\_write", O\_CREAT, 0666, 1);

// 创建读信号量

read\_psx = sem\_open("sem\_read", O\_CREAT, 0666, 0);

// 创建一个无名管道

if (pipe(fd) < 0) {

printf("pipe error\n");

exit(1);

}

// 创建子进程

for (int i = 0; i < CHILD\_NUM; i++) {

//fork（）函数通过系统调用创建一个与原来进程几乎完全相同的进程

//在父进程中，fork返回新创建子进程的进程ID；在子进程中，fork返回0；如果出现错误，fork返回一个负值；

//父进程的pid指向子进程的进程id

pid = fork();

sid++;

// 判断是否 fork 进程成功

if (pid < 0) {

printf("fork error\n");

destory\_sem();

exit(0);

// 如果在子进程中，则跳出循环

//子进程没有子进程，所以pid==0

} else if (pid == 0) {

break;

}

}

if (pid == 0) {

sem\_wait(write\_psx);

write\_to\_pipe(fd, sid);

sem\_post(write\_psx);

sem\_post(read\_psx);

exit(0);

} else {

for(int i=0;i<CHILD\_NUM;i++){

sem\_wait(read\_psx);

}

read\_from\_pipe(fd);

destory\_sem();

}

return 0;

}

// 写数据到管道

void write\_to\_pipe(int fd[2], int sid) {

int value;

char buf[100];

// 关闭管道的读取端

close(fd[0]);

memset(buf, sid + '0', 100);

//memset将缓存区buf初始化为sid+'0'，标注为字符输出显示buf最大空间

printf("child proc %d write %d bytes data\n", sid, (int) sizeof(buf));

// 向管道中写入数据

write(fd[1], buf, sizeof(buf));

}

// 从管道中读取数据

void read\_from\_pipe(int fd[2]) {

int ret;

char buf[1024];

// 关闭管道的写入端

close(fd[1]);

memset(buf, '\0', 1024);

ret = read(fd[0], buf, 1024);

while (ret > 0) {

printf("father read %d bytes data: %s\n", ret, buf);

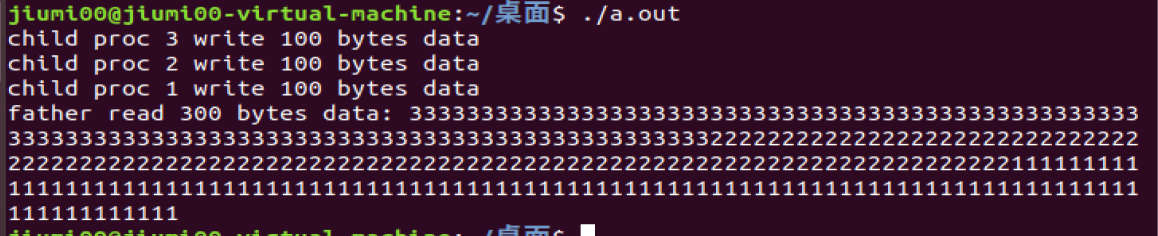
memset(buf, '\0', 1024);

ret = read(fd[0], buf, 1024);

}

}

验证结果



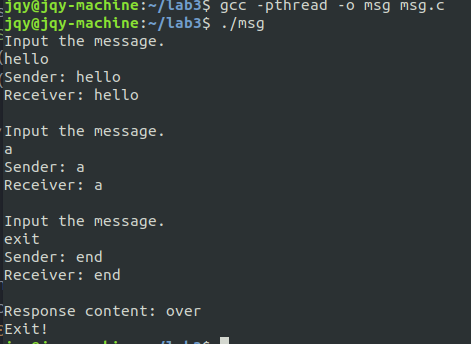
### 3.利用消息队列实现进程间通信

**实验代码**





**实验运行结果**

****

### 4.利用共享内存实现进程间通信

**公共头 common.h**

#ifndef **LAB3\_4\_H**

#define **LAB3\_4\_H**

#include **<stdio.h>**

#include **<stdlib.h>**

#include **<unistd.h>**

#include **<signal.h>**

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

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

#include **<semaphore.h>**

#include **<fcntl.h>**

#include **<string.h>**

#define **SHM\_SIZE** 1024

#define **KEY\_NUM** 2333

#endif

**发送端代码 sender.c**

#include **"common.h"**

**void** \*shmp;

**int** shmid;

**void** destroy();

**void** catch\_ctrlc(**int** sig);

**int** main() {

key\_t key = **KEY\_NUM**;

sem\_t \*send\_psx, \*recv\_psx, \*final\_psx;

*// 捕捉 Ctrl+C 动作*

signal(**SIGINT**, catch\_ctrlc);

*// 获取共享内存*

shmid = shmget(key, **SHM\_SIZE**, 0666 | **IPC\_CREAT**);

*// 是否成功创建共享内存*

**if** (shmid < 0) {

printf(**"get share memory error\n"**);

exit(1);

}

*// 启动共享内存访问*

shmp = shmat(shmid, **NULL**, 0);

**if** (shmp < 0) {

printf(**"get share memory pointer error\n"**);

exit(1);

}

*// 创建有名信号量*

send\_psx = sem\_open(**"sem\_send"**, **O\_CREAT**, 0666, 1);

recv\_psx = sem\_open(**"sem\_recv"**, **O\_CREAT**, 0666, 0);

final\_psx = sem\_open(**"sem\_final"**, **O\_CREAT**, 0666, 0);

*// 添加字符串结尾标识*

memset((**char** \*)shmp, **'\0'**, **SHM\_SIZE**);

**do** {

printf(**"sender> "**);

sem\_wait(send\_psx);

*// 输入信息到共享内存中*

scanf(**"%s"**, (**char** \*)shmp);

sem\_post(recv\_psx);

sem\_post(send\_psx);

} **while** (strncmp((**char** \*)shmp, **"exit"**, strlen(**"exit"**)));

sem\_wait(final\_psx);

printf(**"receive: %s\n"**, (**char** \*)shmp);

destroy();

printf(**"sender end\n"**);

**return** 0;

}

**void** destroy() {

*// 分离共享内存*

shmdt(shmp);

*// 删除共享内存*

shmctl(shmid, **IPC\_RMID**, **NULL**);

sem\_unlink(**"sem\_send"**);

sem\_unlink(**"sem\_recv"**);

sem\_unlink(**"sem\_final"**);

}

**void** catch\_ctrlc(**int** sig)

{

destroy();

exit(0);

}

**接收端代码 recv.c**

#include **"common.h"**

**void** \*shmp;

**int** shmid;

sem\_t \*send\_psx, \*recv\_psx, \*final\_psx;

**void** close\_all();

**void** catch\_ctrlc(**int** sig);

**int** main() {

key\_t key = **KEY\_NUM**;

**char** msg[**SHM\_SIZE**];

**int** must\_stop = 0;

*// 捕捉 Ctrl+C 动作*

signal(**SIGINT**, catch\_ctrlc);

*// 获取共享内存*

shmid = shmget(key, **SHM\_SIZE**, 0666 | **IPC\_CREAT**);

**if** (shmid < 0) {

printf(**"get share memory error\n"**);

exit(1);

}

*// 启动共享内存访问*

shmp = shmat(shmid, **NULL**, 0);

**if** (shmp < 0) {

printf(**"get share memory pointer error\n"**);

exit(1);

}

*// 创建有名信号量*

send\_psx = sem\_open(**"sem\_send"**, **O\_CREAT**, 0666, 1);

recv\_psx = sem\_open(**"sem\_recv"**, **O\_CREAT**, 0666, 0);

final\_psx = sem\_open(**"sem\_final"**, **O\_CREAT**, 0666, 0);

**do** {

sem\_wait(recv\_psx);

*// 拷贝字符串*

strcpy(msg, (**char** \*) shmp);

**if** (strcmp(msg, **"exit"**) == 0) {

must\_stop = 1;

} **else** {

*// 添加字符串结尾标识*

memset((**char** \*) shmp, **'\0'**, **SHM\_SIZE**);

printf(**"receive: %s\n"**, msg);

}

}**while** (!must\_stop);

*// 发送 over 到 sender*

strcpy((**char** \*) shmp, **"over"**);

sem\_post(final\_psx);

close\_all();

printf(**"receiver end\n"**);

**return** 0;

}

**void** close\_all()

{

sem\_close(send\_psx);

sem\_close(recv\_psx);

sem\_close(final\_psx);

shmdt(shmp);

}

**void** catch\_ctrlc(**int** sig)

{

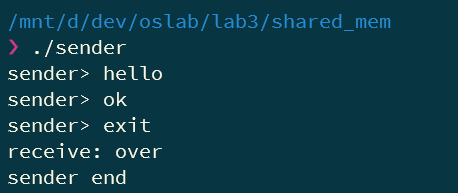
close\_all();

exit(0);

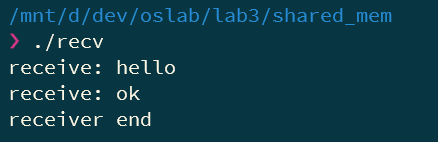
}

验证结果

发送端



接收端



# 五 个人实验改进与总结

## 5.1 个人实验改进

### 模拟shell

对输入的命令解析完后，在执行命令前添加了重定向功能。含重定向的shell命令，需要创建或打开重定向的目标文件，将其文件描述符修改为控制台标准输出的描述符（即1），即可实现重定向

void redirect(char \* argv[])

{

int i;

for(i=0; argv[i] != NULL; ++i)

{

if(strcmp(argv[i], ">") == 0)

{

if(argv[i+1] == NULL){

perror("command '>'[option]?");

exit(1);

}

argv[i] = NULL;

//reate/open file

int fd =open(argv[i+1], O\_RDWR|O\_CREAT|O\_TRUNC, 0664);

if(fd == -1){

perror("open");

exit( 1);

}

// redirct

dup2(fd, 1);

close(fd);

}

}

}

### 管道通信程序

为了实现等待三个子进程信号量写完再开始读的操作，设置循环

for(int i=0;i<CHILD\_NUM;i++){

sem\_wait(read\_psx);

}

根据宏定义CHILD\_NUM的个数来设置sem\_wait()的次数

### 消息队列

int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg)

msgsnd(msgid, &s\_msg, sizeof(struct msgbuf), 0);

msgflg为0时，当消息队列满时，msgsnd将会阻塞，直到消息能写进消息队列

如果msgsnd函数被阻塞，则在下面某个条件满足时解除阻塞。

（1）消息队列中有容纳要写入消息的空间。

（2）消息队列被删除。

（3）进程被信号中断。

### 共享内存

 linux下可以通过信号机制来实现程序的软中断，默认情况下一个程序对ctrl-c发出的信号（SIGINT）的处理方式是退出进程，所以当我们按下ctrl-c的时候就可以终止一个进程的运行。

signal(SIGINT, catch\_ctrlc);

...

void close\_all()

{

sem\_close(send\_psx);

sem\_close(recv\_psx);

sem\_close(final\_psx);

shmdt(shmp);

}

void catch\_ctrlc(int sig)

{

close\_all();

exit(0);

}

**（大部分同学的代码都是参考网上资料，或者借鉴已完成同学的，此处重点突出自己如何改进，为什么这样改进，说出自己改进部分的优缺点，有效文字不得少于200字）**

## 5.2 个人实验总结

### 模拟shell

Linux系统提供给用户的最重要的系统程序是Shell命令语言解释程序。它不属于内核部分，而是在核心之外，以用户态方式运行。其基本功能是解释并执行用户打入的各种命令，实现用户与Linux核心的接口。

它的执行过程基本上按如下步骤：

(1)读取用户由键盘输入的命令行。

(2)分析命令，以命令名作为文件名，并将其它参数改造为系统调用execve( )内部处理所要求的形式。

(3)终端进程调用fork( )建立一个子进程。

(4)。当子进程运行时调用execve( )，子进程根据命令名到目录中查找有关文件，将它调入内存，执行这个程序。

(5)当子进程完成处理后终止，向父进程（终端进程）报告

如果是含重定向的shell命令，则需要创建或打开重定向的目标文件，将其文件描述符修改为控制台标准输出的描述符（即1），即可实现重定向

### 管道通信程序

利用pipe()创建无名管道并判断是否创建成功

在 POSIX 标准中，信号量分两种，一种是无名信号量，一种是有名信号量。无名信号量一般用于线程间同步或互斥，而有名信号量一般用于进程间同步或互斥。

这里用sem\_open()创建一个有名信号量

函数介绍:

sem\_t \*sem\_open(const char \*name, int oflag, mode\_t mode, unsigned int value);

name：信号量文件名

flags：sem\_open() 函数的行为标志。- O\_CREAT:若此文件不存在则创建它

mode：文件权限(可读、可写、可执行)的设置。0666:可读可写

value：信号量初始值。

利用for循环来创建所需的三个子进程，如果发现是子进程就直接从创建子进程的循环中跳出来，保证了每次只有父进程来做循环创建子进程的工作 当pid指向子进程id时,先等待一个写信号量(阻塞直至sem>0，接触阻塞后-1)，再把数据写入管道。

post增加写/读信号量的值(+1)，退出 如果是父进程,等待一个读信号量，读入管道后释放信号量

关于读/写管道端的操作：

分配buf缓冲区，关闭另一个端口，初始化buf，执行读/写操作

非阻塞写和非阻塞读的目的在于，阻塞写时，管道满了之后进程被阻塞，无法设置终止条件从而结束写，读也是一样，管道空了之后进程被阻塞，无法设置终止条件从而结束读。

### 消息队列

利用int msgget(key\_t key, int msgflg)来创建一个消息队列，并根据返回值来判断是否创建成功

关于pthread\_join()：主线程等待子线程的终止。也就是在子线程调用了pthread\_join()方法后面的代码，只有等到子线程结束了才能执行，并且可以对线程资源进行回收

区别正常消息与结束消息，设置结束时识别的关键字

消息队列提供了一种从一个进程向另一个进程发送一个数据块的方法。  每个数据块都被认为含有一个类型，接收进程可以独立地接收含有不同类型的数据结构。我们可以通过发送消息来避免命名管道的同步和阻塞问题。但是消息队列与命名管道一样，每个数据块都有一个最大长度的限制。

   (1)消息队列是消息的链表,具有特定的格式,存放在内存中并由消息队列标识符标识.

    (2)消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取.

### 共享内存

共享内存并未提供同步机制，所以我们通常需要用其他的机制来同步对共享内存的访问，比如信号量

不相关的进程可以通过shmget()函数的返回值访问同一共享内存.程序对所有共享内存的访问都是间接的，程序先通过调用shmget函数并提供一个键，再由系统生成一个相应的共享内存标识符（shmget函数的返回值）.

第一次创建完共享内存时，它还不能被任何进程访问，shmat函数的作用就是用来启动对该共享内存的访问，并把共享内存连接到当前进程的地址空间

要使用一块**共享内存**

* 进程必须首先**分配**它
* 随后需要**访问这个共享内存块的每一个进程都必须将这个共享内存绑定到自己的地址空间中**
* 当完成通信之后，所有进程都将**脱离共享内存**，并且由一个进程**释放该共享内存块（shmdt()）**

**（此处类似于写观后感，同学们辛辛苦苦完成实验，单独编译内核可能就折腾很久，相信会有很多的感触和收获，让你的笔在第一时间记录这美好的瞬间，此处不得少于200字）**

# 六 参考文献

**（此处罗列同学们在做实验过程中 所参考的文献、书籍、网站， 或者向某个高手同学请教都可以写， 大多数同学应该是参考 网站、书籍和问人比较多，以条目化的形式呈现， 如1. 2. 3. ）**

1. <https://www.cnblogs.com/binbinstory/p/7979173.html>
2. <https://blog.csdn.net/kwinway/article/details/79633285>
3. <https://blog.csdn.net/u013476751/article/details/42921931?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase>
4. <https://blog.csdn.net/ljianhui/article/details/10253345>
5. <https://blog.csdn.net/l1902090/article/details/37741317>