|  |
| --- |
| 杭州电子科技大学 |
| Linux进程管理 |
| 操作系统实验报告 |

|  |
| --- |
| xxxxxx  2019-6-6 |

目录

[操作系统实验—Linux进程管理 3](#_Toc11284136)

[一、 设计目的 3](#_Toc11284137)

[二、 设计内容 3](#_Toc11284138)

[1. 实现一个模拟的shell 3](#_Toc11284139)

[2. 实现一个管道通信程序 3](#_Toc11284140)

[3. 利用linux的消息队列通信机制实现两个进程之间的通信 4](#_Toc11284141)

[4. 利用Linux的共享内存通信机制实现两个进程之间的通信 5](#_Toc11284142)

[三、 实现内容 5](#_Toc11284143)

[1. 实现一个模拟的shell 5](#_Toc11284144)

[1) 实验原理 5](#_Toc11284145)

[2) 实验实现 6](#_Toc11284146)

[2. 实现管道通信 9](#_Toc11284147)

[1) 实验原理 9](#_Toc11284148)

[2) 实验实现 10](#_Toc11284149)

[3. 实现消息队列通信 14](#_Toc11284150)

[1) 实验原理 14](#_Toc11284151)

[2) 实验实现 15](#_Toc11284152)

[4. 实现共享内存通信 20](#_Toc11284153)

[1) 实验原理 20](#_Toc11284154)

[2) 实验实现 21](#_Toc11284155)

操作系统实验—Linux进程管理

# 设计目的

1. 通过对Linux进程控制的相关系统调用编程应用，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. 根据自身的情况，进一步阅读分析系统调用的内核源码实现

# 实验要求

## 实现一个模拟的shell

要求能够根据用户输入的字符串为相应的命令创建相应的子进程并且执行相关的应用程序输入exit则退出，输入未知命令则输出command not found。

## 实现一个管道通信程序

由父进程创建三个子进程并且三个子进程利用管道和父进程通信，等待三个子进程全部发送完消息之后父进程接受消息要求能够实验阻塞性读写过程中的各种情况，测试管道默认大小，并利用Posix信号量机制实现进程之间对管道的互斥访问

## 利用linux的消息队列通信机制实现两个进程之间的通信

创建三个线程sender1 sender2 receive 三个线程功能描述如下：

Sender1:

创建一个消息队列，等待用户通过终端输入一串字符串，并将这串字符串通过消息队列发送给receiver线程并可循环发送多个消息，直到用户输入exit 表示不再发送消息最后向receiver发送end1并且等待receiver的应答等到应答消息后将接受到的消息显示在终端上，结束线程的运行。

Sender2：

共享sender1的消息队列，等待用户通过终端输入一串字符串，并将这串字符串通过消息队列发送给receiver线程并可循环发送多个消息，直到用户输入exit 表示不再发送消息最后向receiver发送end2并且等待receiver的应答等到应答消息后将接受到的消息显示在终端上，结束线程的运行。

Recever：

它通过消息队列接收来自sender1和sender2两个线程的消息，将消息显示在屏幕上,当收到end1的时候向sender1发送over1, 当收到end2的时候向sender2发送over2。 接收完成后结束线程运行，选择合适的信号量完成三个线程的同步和互斥。

## 利用Linux的共享内存通信机制实现两个进程之间的通信

编写程序sender它会创建一个共享内存然后会等待用户通过终端，输入一串字符，并将这段字符通过共享内存发送给receiver 等待receiver的应答，收到消息后，将接收到的应答消息显示在终端屏幕上删除共享内存结束程序运行，编写receiver，它通过共性内存接受sender的消息，将消息显示在终端屏幕上，然后再通过共享内存向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，进程的互斥和同步。

# 开发平台

Ubuntu，gcc，vim，vscode，gdb等。

# 设计和实现内容

## 实现一个模拟的shell

### 实验原理

该模拟程序用于模拟linux中shell的功能最重要的功能是为程序命令创建子进程,并且在子进程中使用exec程序替换函数替换子程序的内容为cmdx中的内容,从而实现shell的基本功能,

Exec函数:

用函数fork创建子进程后，如果希望在当前子进程中运行新的程序，可以调用exec函数执行另一个程序.当进程调用exec函数时，该进程用户空间资源（正文、数据、堆和栈）完全由新程序替代，新程序则从main函数开始执行.因为调用exec函数并没有创建新的进程，所以前后的进程ID并没有改变，也即内核信息基本不做修改.

基本设计框图



### 实验实现

实验代码

主程序

主程序动作

接收用户命令

判断命令

启动子进程

使用程序替换函数替换子程序内容

退出则返回

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <string.h>  #include <sys/types.h>  #include <sys/wait.h>  **char** \*sys\_cmd [4] = {"exit", "cmd1", "cmd2", "cmd3"};  *//获取命令代号*  **int** get\_cmd\_id(**char** \*cmd)  {  **int** i;  for (i = 0; i < 4; i++) {  if (strcmp(cmd, sys\_cmd[i]) == 0) {  return i;  }  }  *//没有命令则返回-1*  return -1;  }  *//创建子进程*  **void** shell\_fork(**int** cmd\_id)  {  *//获取pid*  **pid\_t** pid = fork();  *//创建进程失败*  if (pid < 0) {  printf("fork error\n");  exit(0);  *//创建子进程成功*  } else if (pid == 0) {  *//在子进程中执行相应的命令*  switch (cmd\_id) {  case 1:  execl("./cmd1", "", NULL);  break;  case 2:  execl("./cmd2", "", NULL);  break;  case 3:  execl("./cmd3", "", NULL);  break;  }  *//执行出错*  printf("execl error\n");  exit(0);  } else {  return;  }  }  **int** main(**void**)  {  **char** cmd[50];  **int** cmd\_id = -1;  while (1) {  printf("shell >> ");  scanf("%s", cmd);  cmd\_id = get\_cmd\_id(cmd);  if (cmd\_id == -1) {  printf("command not found\n");  } else if (cmd\_id == 0) {  exit(0);  } else {  shell\_fork(cmd\_id);  }  wait(NULL);  }  }  */\**  *相关函数*  *fork()创建子进程*  *int execl(const char \* path, const char \* arg, ...);*  *程序替换函数*  *execl()用来执行参数path 字符串所代表的文件路径, 接下来的参数代表执行该文件时传递过去的argv(0), argv[1], …, 最后一个参数必须用空指针(NULL)作结束.*  *\*/* |

Cmd1

|  |
| --- |
| #include <stdio.h>  **int** main(**void**)  {  printf("cmd1 run\n");  return 0;  } |

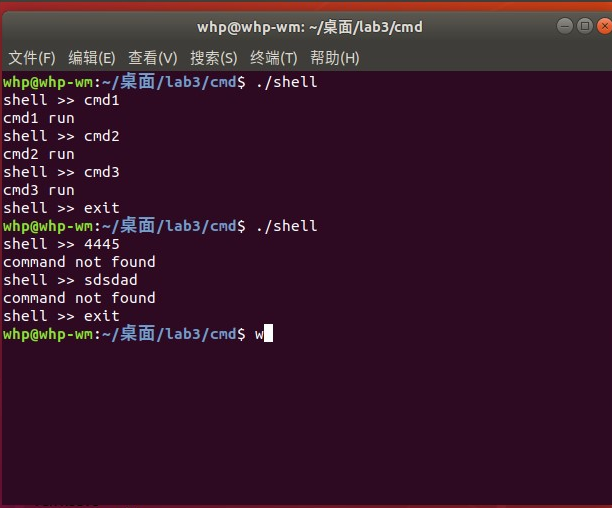
Cmd2

|  |
| --- |
| #include <stdio.h>  **int** main(**void**)  {  printf("cmd2 run\n");  return 0;  } |

Cmd3

|  |
| --- |
| #include <stdio.h>  **int** main(**void**)  {  printf("cmd3 run\n");  return 0;  } |

### 测试截图



## 实现管道通信

### 实验原理

匿名管道通讯

创建匿名管道

int pipe(int pipefd[2]);

pipefd[0] : 表示读管道

pipefd[1] : 表示写管道

返回 0表示成功，非零表示创建失败

管道作用于有血缘关系的进程之间,通过fork来传递

调用pipe函数时在内核中开辟一块缓冲区（称为管道）用于通信，它有一个读端一个

写端，然后通过filedes参数传出给用户程序两个文件描述符，filedes[0]指向管道的读

端，filedes[1]指向管道的写端（很好记，就像0是标准输入1是标准输出一样）。所以管道

在用户程序看起来就像一个打开的文件，通过read(filedes[0]);或者write(filedes[1]);

管道是半双工的，即同一时间同一进程只能读取或者写入

管道的

阻塞：在读写数据的时候如果不能读或写的时候程序将一直等待在这里，直到读或些被执行或者设置的超时时间到达才会返回。

非阻塞：如果不能立即读写程序立即返回。

POSIX 命名信号量 之所以称为命名信号量，是因为它有一个名字、一个用户ID、一个组ID和权限，这些是提供给不共享内存的那些进程使用命名信号量的接口。命名信号量的名字是一个遵守路径名构造规则的字符串。

sem\_open函数用于创建或打开一个命名信号量：

sem\_close函数用于关闭命名信号量

sem\_unlink函数用于在所有进程关闭了命名信号量之后，将信号量从系统中删除



### 实验实现

动作

子进程1用于测试管道大小

子进程2用于发送

子进程3用于发送

主进程读取进程一的输入内容

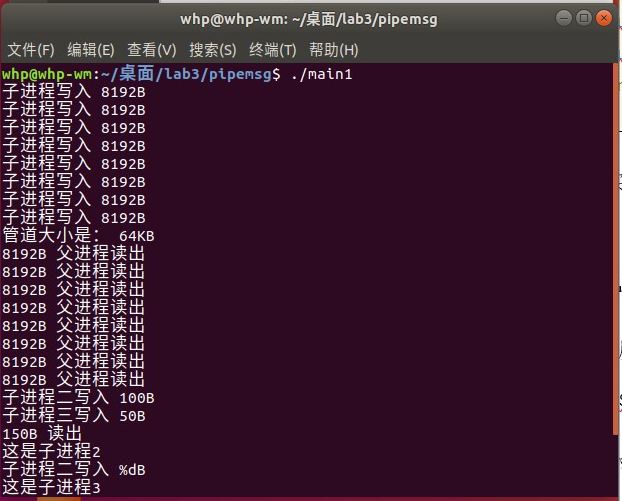
等待进程二和三写入完成后

然后读出管道中的信息

实验代码

|  |
| --- |
| *//进程管道通信实验*  #include <errno.h>  #include <fcntl.h>  #include <semaphore.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>  *//定义每次最大写入量*  #define BUF\_MAX\_SIZE 8192  *//检查是否存在*  #define CHECK(x) \  do { \  if (!(x)) { \  fprintf(stderr, "%s:%d: ", \_\_func\_\_, \_\_LINE\_\_); \  perror(#x); \  exit(-1); \  } \  } while (0)  *//主要函数*  **int** main(**int** argc, **char** \*\*argv) {  **int** pipefd[2], pid, i = 0;  **int** flag = 0;  **ssize\_t** n;  **char** buf[BUF\_MAX\_SIZE];  **char** str[BUF\_MAX\_SIZE];  *//定义三个有名信号量*  sem\_t \*write\_mutex;  sem\_t \*read\_mutex1;  sem\_t \*read\_mutex2;  *//创建信号量*  *//可读可写*  write\_mutex = sem\_open("pipe\_test\_wm", O\_CREAT | O\_RDWR, 0666, 0);  read\_mutex1 = sem\_open("pipe\_test\_rm\_1", O\_CREAT | O\_RDWR, 0666, 0);  read\_mutex2 = sem\_open("pipe\_test\_rm\_2", O\_CREAT | O\_RDWR, 0666, 0);  *//全0写入*  memset(buf, 0, BUF\_MAX\_SIZE);  memset(str, 0, BUF\_MAX\_SIZE);  *//创建管道并检查管道创建情况*  CHECK(pipe(pipefd) >= 0);  *//检查进程创建*  CHECK((pid = fork()) >= 0);  *//创建第一个儿子进程*  if (pid == 0) {  **int** count = 0;  *//关闭读取描述符*  close(pipefd[0]);  **int** flags = fcntl(pipefd[1], F\_GETFL);  *//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*  fcntl(pipefd[1], F\_SETFL, flags | O\_NONBLOCK);  *//写入*  while (!flag) {  n = write(pipefd[1], buf, BUF\_MAX\_SIZE);  *//如果返回-1那么就写满了*  if (n == -1) {  flag = 1;  } else {  count++;  printf("子进程写入 %dB\n", n);  }  }  printf("管道大小是： %dKB\n", (count \* BUF\_MAX\_SIZE) / 1024);  exit(0);  }  CHECK((pid = fork()) >= 0);  if (pid == 0) {  sem\_wait(write\_mutex);  close(pipefd[0]);  n = write(pipefd[1], "这是子进程2\n", 100);  printf("子进程二写入 %dB\n", n);  sem\_post(write\_mutex);  sem\_post(read\_mutex1);  exit(0);  }  CHECK((pid = fork()) >= 0);  if (pid == 0) {  sem\_wait(write\_mutex);  close(pipefd[0]);  n = write(pipefd[1], "这是子进程3\n", 50);  printf("子进程三写入 %dB\n", n);  sem\_post(write\_mutex);  sem\_post(read\_mutex2);  exit(0);  }  *//父进程的操作*  wait(0);  close(pipefd[1]);  **int** flags = fcntl(pipefd[0], F\_GETFL);  fcntl(pipefd[0], F\_SETFL, flags | O\_NONBLOCK);  while (!flag) {  n = read(pipefd[0], str, BUF\_MAX\_SIZE);  if (n == -1) {  flag = 1;  } else {  printf("%dB 父进程读出\n", n);  }  }  sem\_post(write\_mutex);  sem\_wait(read\_mutex1);  sem\_wait(read\_mutex2);  n = read(pipefd[0], str, BUF\_MAX\_SIZE);  printf("%dB 读出\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\_unlink("pipe\_test\_wm");  sem\_unlink("pipe\_test\_rm\_1");  sem\_unlink("pipe\_test\_rm\_2");  return 0;  } |

### 实验截图



## 实现消息队列通信

### 实验原理

通过posix机制实现进程间的互斥和同步,用来实现消息队列通信的效果,同时在通信的过程中使用多个信号量来实现程序的功能.

int msgget(key\_t key, int msgflag)功能： 用于创建一个新的或打开一个已经存在的消息队列，此消息队列与key相对应。

msgsnd()用来把消息添加到消息队列中。它的原型为：

int msgsend(int msgid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg);

msgrcv()函数

该函数用来从一个消息队列获取消息，它的原型为

int msgrcv(int msgid, void \*msg\_ptr, size\_t msg\_st, long int msgtype, int msgflg);

pthread\_create是类Unix操作系统（Unix、Linux、Mac OS X等）的创建线程的函数。它的功能是创建线程（实际上就是确定调用该线程函数的入口点），在线程创建以后，就开始运行相关的线程函数。int pthread\_create(

pthread\_t \*restrict tidp, //新创建的线程ID指向的内存单元。

const pthread\_attr\_t \*restrict attr, //线程属性，默认为NULL

void \*(\*start\_rtn)(void \*), //新创建的线程从start\_rtn函数的地址开始运行

void \*restrict arg //默认为NULL。若上述函数需要参数，将参数放入结构中并将地址作为arg传入。);

pthread\_join()函数，以阻塞的方式等待thread指定的线程结束。当函数返回时，被等待线程的资源被收回。如果线程已经结束，那么该函数会立即返回。并且thread指定的线程必须是joinable的



### 实验实现

|  |
| --- |
| #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 QUEUE\_ID 10086  #define MAX\_SIZE 1024  #define MSG\_STOP "exit"  #define snd\_to\_rcv1 1  #define snd\_to\_rcv2 2  #define rcv\_to\_snd1 3  #define rcv\_to\_snd2 4  #define CHECK(x) \  do { \  if (!(x)) { \  fprintf(stderr, "%s:%d: ", \_\_func\_\_, \_\_LINE\_\_); \  perror(#x); \  exit(-1); \  } \  } while (0) \  #define P(x) sem\_wait(&x)  #define V(x) sem\_post(&x)  **struct** msg\_st {  **long** **int** message\_type;  **char** buffer[MAX\_SIZE + 1];  };  **void** \*sender1();  **void** \*sender2();  **void** \*receiver();  */\* 信号量六个 \*/*  *// 两个发送之间互斥，消息队列大小限制，*  sem\_t w\_mutex, empty, full, over, rcv\_dp, snd\_dp;  **void** \*sender1() {  **int** mq;  **struct** msg\_st buf;  **ssize\_t** bytes\_read;  */\* 打开消息队列\*/*  mq = msgget((**key\_t**) QUEUE\_ID, 0666 | IPC\_CREAT);  CHECK((**key\_t**) -1 != mq);  do {  P(w\_mutex);  P(snd\_dp);  printf("sender1> ");  V(rcv\_dp);  fflush(stdout);  fgets(buf.buffer, BUFSIZ, stdin);  buf.message\_type = snd\_to\_rcv1;  */\* send the message \*/*  P(empty);  CHECK(0 <= msgsnd(mq, (**void**\*)&buf, MAX\_SIZE, 0));  V(full);  V(w\_mutex);  } while (strncmp(buf.buffer, MSG\_STOP, strlen(MSG\_STOP)));  */\* 等待回应 \*/*  P(over);  bytes\_read = msgrcv(mq, (**void** \*) &buf, MAX\_SIZE, rcv\_to\_snd1, 0);  CHECK(bytes\_read >= 0);  printf("%s", buf.buffer);  printf("--------------------------------------------\n");  V(snd\_dp);  pthread\_exit(NULL);  }  **void** \*sender2() {  **int** mq;  **struct** msg\_st buf;  **ssize\_t** bytes\_read;  mq = msgget((**key\_t**) QUEUE\_ID, 0666 | IPC\_CREAT);  CHECK((**key\_t**) -1 != mq);  do {  P(w\_mutex);  P(snd\_dp);  printf("sender2> ");  V(rcv\_dp);  fflush(stdout);  fgets(buf.buffer, BUFSIZ, stdin);  buf.message\_type = snd\_to\_rcv2;  P(empty);  CHECK(0 <= msgsnd(mq, (**void** \*) &buf, MAX\_SIZE, 0));  V(full);  V(w\_mutex);  } while (strncmp(buf.buffer, MSG\_STOP, strlen(MSG\_STOP)));  P(over);  bytes\_read = msgrcv(mq, (**void** \*) &buf, MAX\_SIZE, rcv\_to\_snd2, 0);  CHECK(bytes\_read >= 0);  printf("%s", buf.buffer);  printf("--------------------------------------------\n");  V(snd\_dp);  pthread\_exit(NULL);  }  *//接收函数*  **void** \*receiver() {  **struct** msg\_st buf, over1, over2;  **int** mq, must\_stop = 2;  **struct** msqid\_ds t;  over1.message\_type = 3;  strcpy(over1.buffer, "over1\n");  over2.message\_type = 4;  strcpy(over2.buffer, "over2\n");  */\* 接收 \*/*  mq = msgget((**key\_t**) QUEUE\_ID, 0666 | IPC\_CREAT);  CHECK((**key\_t**) -1 != mq);  do {  **ssize\_t** bytes\_read, bytes\_write;  */\* receive the message \*/*  P(full);  bytes\_read = msgrcv(mq, (**void** \*) &buf, MAX\_SIZE, 0, 0);  V(empty);  CHECK(bytes\_read >= 0);  if (!strncmp(buf.buffer, MSG\_STOP, strlen(MSG\_STOP))) {  if (buf.message\_type == 1) {  bytes\_write = msgsnd(mq, (**void** \*) &over1, MAX\_SIZE, 0);  CHECK(bytes\_write >= 0);  V(over);  must\_stop--;  } else if (buf.message\_type == 2) {  bytes\_write = msgsnd(mq, (**void** \*) &over2, MAX\_SIZE, 0);  CHECK(bytes\_write >= 0);  V(over);  must\_stop--;  }  } else {  P(rcv\_dp);  printf("Received%d: %s", buf.message\_type, buf.buffer);  printf("--------------------------------------------\n");  V(snd\_dp);  }  } while (must\_stop);  */\* 清除信号量 \*/*  P(snd\_dp);  CHECK(!msgctl(mq, IPC\_RMID, &t));  pthread\_exit(NULL);  }  **int** main(**int** argc, **char** \*\*argv) {  **pthread\_t** t1, t2, t3;  **int** state;  *//初始化信号量*  sem\_init(&snd\_dp, 1, 1);  sem\_init(&rcv\_dp, 1, 0);  sem\_init(&empty, 1, 10);  sem\_init(&full, 1, 0);  sem\_init(&w\_mutex, 1, 1);  sem\_init(&over, 1, 0);  *//创建线程*  state = pthread\_create(&t1, NULL, receiver, NULL);  CHECK(state == 0);  state = pthread\_create(&t3, NULL, sender1, NULL);  CHECK(state == 0);  state = pthread\_create(&t2, NULL, sender2, NULL);  CHECK(state == 0);  *//等待线程结束回收资源*  pthread\_join(t3, NULL);  pthread\_join(t2, NULL);  pthread\_join(t1, NULL);  return 0;  } |

### 实验截图



## 实现共享内存通信

### 实验原理

shmget()函数

该函数用来创建共享内存，它的原型为：

int shmget(key\_t key, size\_t size, int shmflg);

shmat()函数 -- at：attach

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。它的原型如下：

void \*shmat(int shm\_id, const void \*shm\_addr, int shmflg);



### 实验实现

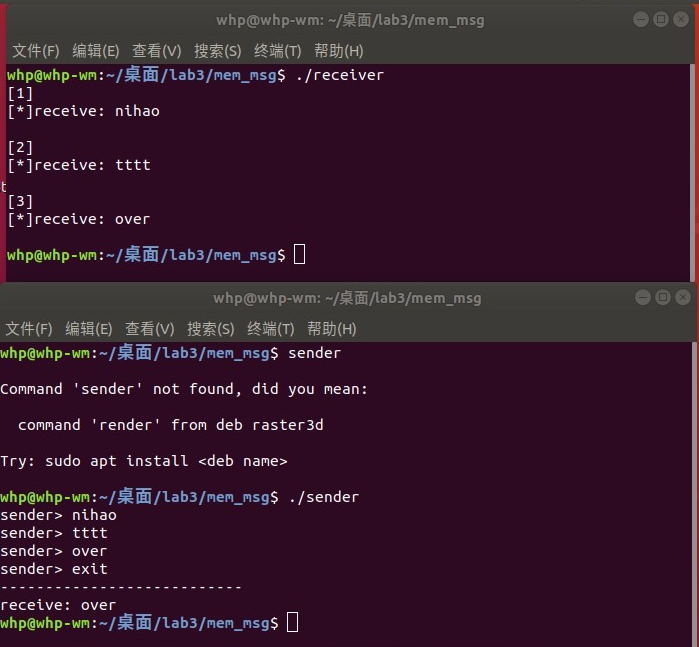
Sender

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/shm.h>  #include <sys/ipc.h>  #include <semaphore.h>  #include <fcntl.h>  #include <string.h>  #define FULL\_PATH "shm\_sgn\_full"  #define EMPTY\_PATH "shm\_sgn\_empty"  #define RTS\_PATH "shm\_sgn\_rts"  #define MSG\_STOP "exit"  #define MAX\_SIZE 1024  #define CHECK(x) \  do { \  if (!(x)) { \  fprintf(stderr, "%s:%d: ", \_\_func\_\_, \_\_LINE\_\_); \  perror(#x); \  exit(-1); \  } \  } while (0) \  #define P(x) sem\_wait(x)  #define V(x) sem\_post(x)  **struct** sm\_st {  **char** data[MAX\_SIZE];  };  *//三个信号量*  sem\_t \*full, \*empty, \*rcv\_to\_snd;  **int** main() {  full = sem\_open(FULL\_PATH, O\_CREAT | O\_RDWR, 0666, 0);  empty = sem\_open(EMPTY\_PATH, O\_CREAT | O\_RDWR, 0666, 1);  rcv\_to\_snd = sem\_open(RTS\_PATH, O\_CREAT | O\_RDWR, 0666, 0);  **void** \*shm = NULL;  **struct** sm\_st \*shared;  **int** shmid, count = 0;  **key\_t** sm\_id = ftok("tok", 1);  shmid = shmget(sm\_id, sizeof(**struct** sm\_st), 0666 | IPC\_CREAT);  CHECK(shmid >= 0);  shm = shmat(shmid, 0, 0);  CHECK(shm >= 0);  shared = (**struct** sm\_st \*) shm;  do {  P(empty);  printf("sender> ");  fflush(stdout);  memset(shared->data, 0, MAX\_SIZE);  fgets(shared->data, MAX\_SIZE, stdin);  V(full);  } while (strncmp(shared->data, MSG\_STOP, strlen(MSG\_STOP)));  P(rcv\_to\_snd);  P(full);  printf("---------------------------\n");  printf("receive: %s\n", shared->data);  V(empty);  sleep(1);  CHECK(shmdt(shared) == 0);  CHECK(shmctl(shmid, IPC\_RMID, 0) == 0);  CHECK(sem\_close(full) == 0);  CHECK(sem\_close(empty) == 0);  CHECK(sem\_close(rcv\_to\_snd) == 0);  CHECK(sem\_unlink(EMPTY\_PATH) == 0);  CHECK(sem\_unlink(FULL\_PATH) == 0);  CHECK(sem\_unlink(RTS\_PATH) == 0);  return 0;  } |

Receiver

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/shm.h>  #include <sys/ipc.h>  #include <semaphore.h>  #include <fcntl.h>  #include <string.h>  #define FULL\_PATH "shm\_sgn\_full"  #define EMPTY\_PATH "shm\_sgn\_empty"  #define RTS\_PATH "shm\_sgn\_rts"  #define MSG\_STOP "exit"  #define MAX\_SIZE 1024  #define CHECK(x) \  do { \  if (!(x)) { \  fprintf(stderr, "%s:%d: ", \_\_func\_\_, \_\_LINE\_\_); \  perror(#x); \  exit(-1); \  } \  } while (0) \  #define P(x) sem\_wait(x)  #define V(x) sem\_post(x)  **struct** sm\_st {  **char** data[MAX\_SIZE];  };  *//三个信号量*  sem\_t \*full, \*empty, \*rcv\_to\_snd;  **int** main(**int** argc, **char** \*\*argv) {  full = sem\_open("shm\_sgn\_full", O\_CREAT | O\_RDWR, 0666, 0);  empty = sem\_open("shm\_sgn\_empty", O\_CREAT | O\_RDWR, 0666, 1);  rcv\_to\_snd = sem\_open("shm\_sgn\_rts", O\_CREAT | O\_RDWR, 0666, 0);  **void** \*shm = NULL;  **struct** sm\_st \*shared;  **int** shmid, count = 0, must\_stop = 0;  **key\_t** sm\_id = ftok("tok", 1);  shmid = shmget(sm\_id, sizeof(**struct** sm\_st), 0666 | IPC\_CREAT);  CHECK(shmid >= 0);  shm = shmat(shmid, 0, 0);  CHECK(shm >= 0);  shared = (**struct** sm\_st \*) shm;  do {  P(full);  if (!strncmp(shared->data, MSG\_STOP, strlen(MSG\_STOP))) {  must\_stop = 1;  } else {  count++;  printf("[%d]\n", count);  printf("[\*]receive: %s\n", shared->data);  }  V(empty);  } while (!must\_stop);  P(empty);  strcpy(shared->data, "over");  V(rcv\_to\_snd);  V(full);  CHECK(shmdt(shared) == 0);  CHECK(sem\_close(full) == 0);  CHECK(sem\_close(empty) == 0);  CHECK(sem\_close(rcv\_to\_snd) == 0);  return 0;  } |

### 实验截图



# 实验遇到的问题及解决方式

做实验中不知道进程和线程如何创建，经过在网上一些查询，知道了fork的使用方式，c语言中多线程函数的使用，终于了解了如何设计和实现本实验的创建进程部分。

使用管道通信的时候会发现管道会有阻塞现象发生，后来了解到管道通信的默认模式是阻塞的，需要调整管道为非阻塞模式才能连续的操作。

有关使用信号量的程序直接使用gcc编译并不会成功，需要使用gcc的参数来编译，这是因为用gcc编译使用了POSIX thread的程序时通常需要加额外的选项，以便使用thread-safe的库及头文件

# 实验结果讨论

首先本次实验结果基本符合预期，完成了实验要求的任务。这些实验内容也让我对一些应用的多端同步和互斥有了了解，我认为这些内容对于软件开发大有益处。

本实验中我们使用信号量机制，来在linux进程通信的过程中实现相应的同步和互斥，很好的巩固了有关进程调度和进程通信的知识。同时其中一些例子也为多线程起到了很好的指导作用，希望本次实验学到的知识能够应用到实战中去。