# Linux平台进程与线程通信

实验名称：Linux平台进程与线程通信

学时安排：2课时 指导老师：李赞

实验类别：验证型、设计型 实验要求：1人1组

学号： 姓名： 班级：

## 一、实验目的和任务

1. 本实验要求理解Linux系统的多种通信机制。

2. 本实验要求理解Linux系统通信不同应用场合的实现。

3.本实验要求学生掌握使用信号量实现对进程运行同步控制。

## 二、实验设备介绍

1.软件需求： win10操作系统，VMware workstation，ubuntu18。

2.硬件需求: PC内存大于1G，硬盘空间大于20G。

## 三、注意事项和要求

1.理解Linux平台多种通信机制的特点和区别。

2.分析程序是否满足任务需求。

3.程序运行结果截屏后放入实验报告中。

## 四、实验内容和步骤

### 4.1Linux中创建新进程

创建进程可使用的方法有fork、system、exec函数调用。

#### 4.1.1调用fork函数创建新进程

fork函数复制正在运行的进程，新进程与原进程都执行fork函数后的语句，fork之前的程序是只执行一次的，fork函数后面的代码则分为父子进程，都会执行。fork函数会在新旧进程中返回不同的值。

1）在父进程中，fork返回新创建子进程的进程ID；

2）在子进程中，fork返回0；

3）如果出现错误，fork返回一个负值；

//045.c fork创建进程1

#include<stdio.h>

#include<unistd.h>

int gval=10;

int main(int argc,char \*argv[])

{

pid\_t pid;

int lval=20;

gval++,lval+=5;

pid=fork();

if(pid==0)

{

gval+=2,lval+=2;

printf("This is Child Proc:[%d,%d]\n",gval,lval);

printf("Child进程ID号为:[%d]\n",getpid());

printf("Child进程的父进程ID号为:[%d]\n",getppid());

}else

{

gval-=2,lval-=2;

printf("This is Parent Proc:[%d,%d]\n",gval,lval);

printf("Parent进程ID号为:[%d]\n",getpid());

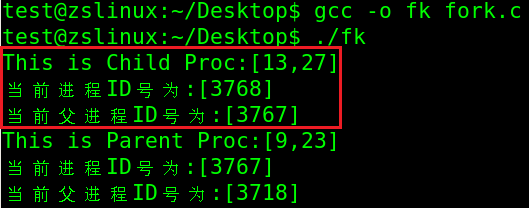
printf("Parent进程的父进程ID号为:[%d]\n",getppid());

}

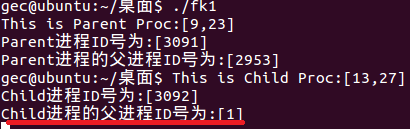
return 0;

}

该程序运行后执行fork方法后，系统将复制程序代码创建第二个相同的进程，两个进程对fork函数返回值是不同的，父进程fork方法得到的是子进程的pid，子进程fork方法返回值是0。所以输出的结果并非是if两个分支都执行了，而是父进程输出的是pid大于零的情部分，子进程输出的是pid为零的情况。



但是在ubuntu中运行得到结果却是有点意外，子进程获得的父进程ID是1，因为父进程执行完毕后就结束了，该进程就不存在了，init进程接管了子进程，这时子进程使用getppid得到的值是1。如果在父进程中添加sleep函数，子进程得到父进程ID。



//046.c fork创建进程2

#include<stdio.h>

#include<unistd.h>

int gval=10;

int main(int argc,char \*argv[])

{

pid\_t pid;

int lval=20;

gval++,lval+=5;

pid=fork();

if(pid==0)

{//子进程执行

gval+=2,lval+=2;

printf("This is Child Proc:[%d,%d]\n",gval,lval);

printf("Child进程ID号为:[%d]\n",getpid());

printf("Child进程的父进程ID号为:[%d]\n",getppid());

}else

{//父进程执行

//先休息一会

sleep(1);

gval-=2,lval-=2;

printf("This is Parent Proc:[%d,%d]\n",gval,lval);

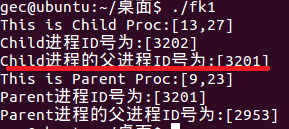
printf("Parent进程ID号为:[%d]\n",getpid());

printf("Parent进程的父进程ID号为:[%d]\n",getppid());

}

return 0;

}



//047.c system创建进程

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

int main()

{

int ret;

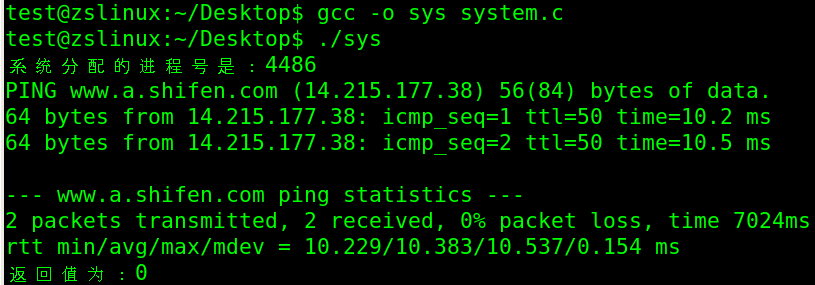
printf("系统分配的进程号是：%d\n",getpid());

ret = system("ping www.baidu.com -c 2");

printf("返回值为：%d\n",ret);

return 0;

}



//048.c execve创建进程

#include<stdio.h>

#include<unistd.h>

int main(void)

{

char \*args[]={"/bin/ls",NULL};

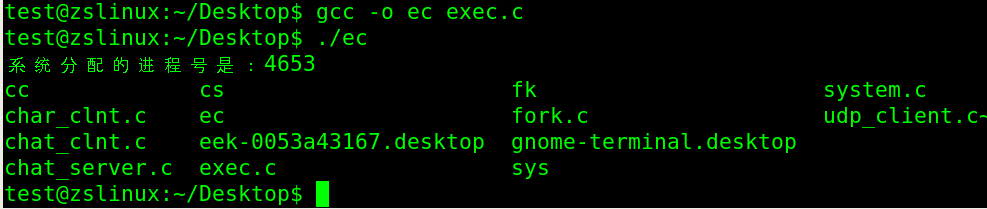
printf("系统分配的进程号是：%d\n",getpid());

if(("/bin/ls",args,NULL)<0)

printf("创建进程出错！\n");

return 0;

}



请分析下面的程序，指出输出结果的特点。

//049.c fork创建多个进程

#include<stdio.h>

#include<unistd.h>

int main(int argc,char \*argv[])

{

pid\_t pid;

pid=fork();

pid=fork();

pid=fork();

printf("当前进程ID号为:[%d]\n",getpid());

return 0;

}

//050.c fork创建进程生成的进程树，尝试根据进程ID绘制出进程创建树

#include<stdio.h>

#include<unistd.h>

int main(int argc,char \*argv[])

{

pid\_t pid;

pid=fork();

if(pid>0)

{

printf("[%d]进程生成了[%d]子进程\n",getpid(),pid);

}

pid=fork();

if(pid>0)

{

printf("[%d]进程生成了[%d]子进程\n",getpid(),pid);

}

pid=fork();

if(pid>0)

{

printf("[%d]进程生成了[%d]子进程\n",getpid(),pid);

}

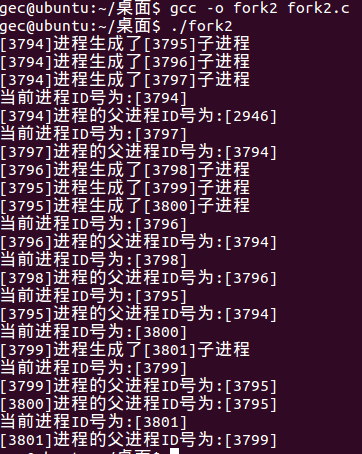
printf("当前进程ID号为:[%d]\n",getpid());

printf("[%d]进程的父进程ID号为:[%d]\n",getpid(),getppid());

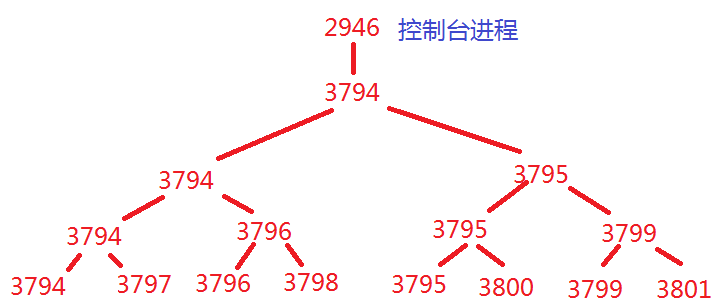
sleep(5);

return 0;

}



根据父子进程关系，得到下面的进程创建树。



#### 4.1.2调用exec函数创建新进程

Linux系统中一个进程是一个运行着的程序段，一个进程主要包括在内存中申请的空间，代码（加载的程序，包括代码段，数据段，BSS），堆，栈，以及内核提供的内核进程信息结构体task\_struct （位置在 /usr/include/linux/sched.h）、打开的文件、上下文（指进程执行活动全过程的静态描述）信息以及挂起的信号等。

exec函数用来执行一个新的进程，系统保留已经分配的进程资源，但是会用新程序的代码覆盖现有代码，有一种借壳运行的特点，系统为旧进程分配了进程ID、文件和内存资源，运行到exec函数时，该资源基本不变的情况下，将新进程代码加载后覆盖旧进程代码，但是进程ID不变。请看下面的代码：

//051.c execve创建进程1

#include<stdio.h>

#include<unistd.h>

int main(int argc,char \*argv[])

{

printf("我是被用来测试的进程，系统分配的进程号是：%d\n",getpid());

return 0;

}

该程序经过编译后可执行文件为：/home/test/Desktop/e1 调用该程序的代码如下：

//052.c execve创建进程2

#include<stdio.h>

#include<unistd.h>

int main(void)

{

char \*args[]={"/home/test/Desktop/e1",NULL};

printf("系统分配的进程号是：%d\n",getpid());

if(execve("/home/test/Desktop/e1",args,NULL)<0)

printf("创建进程出错！\n");

printf("你不可能看到这行输出的。");

return 0;

}

下面的运行截图，可以看到e1程序的代码完全取代了e2进程中从调用execve位置开始之后的代码。



即使exec调用是处在循环体内，循环也会不再继续因为循环结构已经不存在了。PCB标识没有变，但是代码重新加载了，exec容易产生一个副作用就是调用进程的工作会被半途而废。

### 4.2父进程管理子进程创建与终结

父进程应当等待子进程结束后，才可以自我结束，如果父进程早于子进程结束，子进程将成为孤儿进程，容易浪费系统资源。

#### 4.2.1 子进程成为僵尸进程

打开两个终端窗体AB，在A终端中编译并运行下面的程序，在B终端中输入ps[空格]-aux命令查看进程状态，会查看到一个僵尸状态的进程，僵尸进程其STAT为Z+。

// 053.c 僵尸进程的产生

#include<stdio.h>

#include<unistd.h>

int main(int argc,char \*argv[])

{

pid\_t pid=fork();

if(pid==0)

puts("Hi,child process.");

else

{

printf("Child process ID:%d \n",pid);

sleep(30);//父进程延时结束

}

if(pid==0)

puts("End child process.\n");

else

puts("End Parent process.\n");

return 0;

}

#### 4.2.2 父进程响应子进程的退出信号

子进程由父进程的fork方法创建，子进程在退出时必须将退出值通知父进程，否则子进程会成为僵尸进程。（也就是说如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中，这种进程称之为僵尸进程。）如果父进程不调用wait() / waitpid()的话，那么僵尸进程保留的信息就不会释放，其进程号就会一直被占用，而系统所能使用的进程号是有限的，如果大量的产生僵死进程，将因为没有可用的进程号而导致系统不能产生新的进程，此即为僵尸进程的危害。Linux通过sigaction方法来注册一个信号函数来实现(类似windows平台的回调函数)。

// 054.c sigaction 信号响应函数

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <sys/wait.h>

void read\_childproc(int sig)

{

int status;

//等待子进程退出，并获得子进程的返回值

pid\_t id=waitpid(-1,&status,WNOHANG);

if(WIFEXITED(status)) //正常退出为真

{

printf("Removed proc id: %d\n",id);

printf("Child send:%d \n",WEXITSTATUS(status));

}

}

int main(int argc,char \* argv[])

{

pid\_t pid;

struct sigaction act;

act.sa\_handler=read\_childproc;

sigemptyset(&act.sa\_mask); //初始化并清空

act.sa\_flags=0;

sigaction(SIGCHLD,&act,0); //修改SIGCHLD信号的处理动作，子进程终止时调用。

pid=fork();

if(pid==0)

{

puts("Hi,I am the First Child Process.");

sleep(10);

return 12;

}else

{

printf("First Child Proc ID: %d\n",pid);

pid=fork();

if(pid==0)

{

puts("Hi,I am the Second Child Process.");

sleep(10);

exit(24);

}else

{

int i;

printf("Second Child Proc ID: %d\n",pid);

for(i=0;i<5;i++)

{

puts("wait...");

sleep(5);

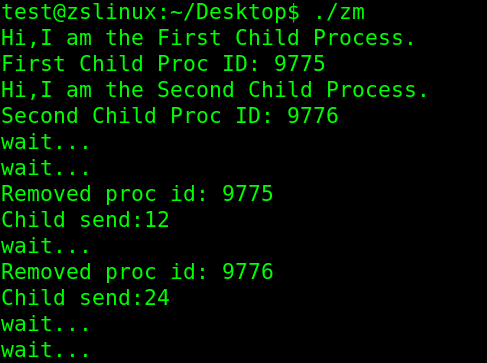
}

}

}

return 0;

}



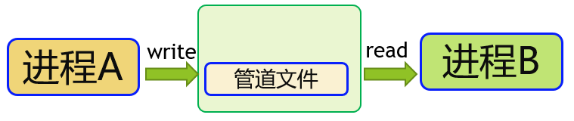
进程间通信(Inter-Process Communication，IPC)是指在进程间实现信息的交换或传递，实现任务进度控制或信息传递等功能，Linux平台支持的进程间(包括线程)通信方式有管道通信、共享内存、信号量通信、信号通信、消息队列、Socket通信。进程通信(IPC)主要应用场合包含下面情况：

|  |  |
| --- | --- |
| 应用场景 | 说明 |
| 数据传递 | 一个进程需要将它的数据发送给另外一个进程，发送的数据量在一个字节到几M字节之间 |
| 共享数据 | 多个进程想要操作相同的的数据变量。 |
| 通知事件 | 一个进程需要向另一个或一组进程发送消息，通知它(它们)发生了某种事件(如进程终止时要通知父进程) |
| 资源共享 | 多个进程之间共享资源。需要内核提供锁和同步机制实现互斥或安全访问。 |
| 进程控制 | 进程希望完全控制另一个进程的执行(如debug进程)，此时控制进程希望能够拦截另一个进程的所有步骤和异常，并能够及时知道它的状态改变。 |

POSIX表示可移植操作系统接口，它定义了操作系统应该为应用程序提供的接口。这一标准的好处就是符合POSIX标准的应用程序支持在POSIX兼容的操作系统中直接编译后运行，使得源代码容易在不同的系统中迁移。Linux平台的API参考了POSIX标准，所以linux代码的通用性比较强。

### 4.3pipe实现的管道通信

Linux中进程通信常使用管道机制，管道在Linux中本质是一个可以读写的文件，使用pipe函数生成一个管道的文件描述符，管道资源是由操作系统管理的。数据从写端流入从读端流出，管道是把若干个内存块用环形队列组织起来，采用管道进行数据读写对数据量无上限。 下图给出了管道在进程间通信的示意。



#### 4.3.1无名管道

在Linux平台无名管道pipe方法的头文件和声明如下：

#include <unistd.h>

int pipe(int filedes[2]);

//成功返回0，失败返回-1； filedes[0]用于接收数据，filedes[1]用于发送数据。

pipe函数创建的管道只能在两个有特定关联的进程间进行通信，例如父子进程之间，或者兄弟进程之间，它们通过fork函数产生联系，这种方式较接近于windows平台的匿名管道。下面示例程序通过fork函数产生两个进程，进程间使用管道进行通信。

//068.c父子进程间使用管道进行通信

#include <stdio.h>

#include <unistd.h>

#include <memory.h>

#define BUF\_SIZE 30

int main(int argc,char \* argv[])

{

int fds[2];

char str[BUF\_SIZE];

char buf[BUF\_SIZE];

pid\_t pid;

pipe(fds);//创建管道文件，并且父子进程变量值都相同

//由此点创建子进程

pid=fork();

if(pid==0)

{

memset(str,0,BUF\_SIZE);

sprintf(str,"[%d]:Hello from child.",getpid());

write(fds[1],str,sizeof(str));

}else

{

memset(buf,0,BUF\_SIZE);

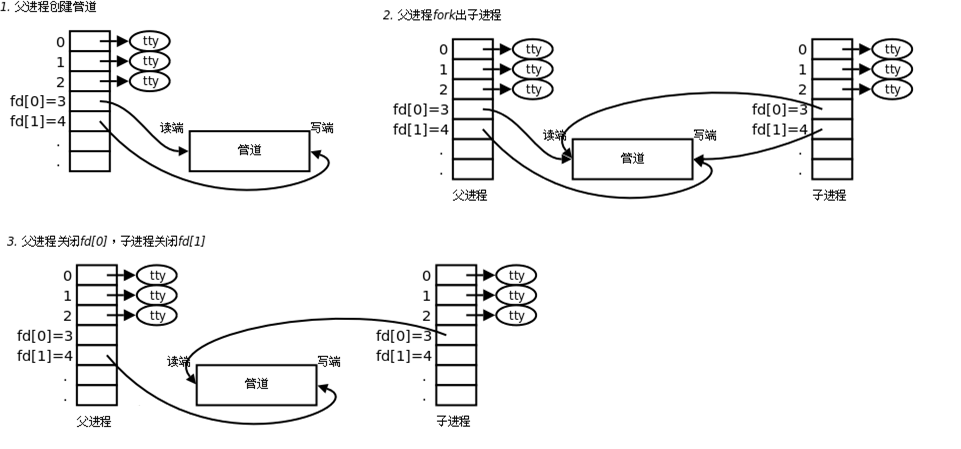
read(fds[0],buf,BUF\_SIZE);

printf("[%d]:Data from child:%s\n",getpid(),buf);

}

}

fork方式创建管道，父子进程间进行通信。



#### 4.3.2命名管道通信

Linux平台创建设备文件来实现命名管道，可通过指定的设备名称来访问管道文件，创建命名管道一种通过函数创建，另一种是在命令终端中使用命令创建。在Shell中mknod和mkfifo可以创建命名管道----《Linux C从入门到精通》page105，在程序中使用mkfifo函数创建命命名管道，头文件和函数原型为：

#include<sys/type.h>

#include<sys/stat.h>

int mkfifo(const char\* pathname,mode\_t mode);

访问命名管道与访问普通磁盘文件流程相同，都是打开文件，对文件进行读写。

//069.c《Linux C从入门到精通》page 106

//命名管道实现进程间通信

#include<stdio.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<stdlib.h>

#define FIFO "fifo4"

int main(void)

{

int fd;

int pid;

char r\_msg[BUFSIZ];

if((pid=mkfifo(FIFO,0777))==-1) //创建命名管道

{

perror("create fifo channel failed!");

return 1;

}

else

printf("create success!\n");

fd=open(FIFO,O\_RDWR); //打开命名管道

if(fd==-1)

{

perror("cannot open the FIFO");

return 1;

}

if(write(fd,"hello world",12)==-1) //写入消息

{

perror("write data error!");

return 1;

}

else

printf("write data success!\n");

if(read(fd,r\_msg,BUFSIZ)==-1) //读取消息

{

perror("read error!");

return 1;

}

else

printf("the receive data is %s!\n",r\_msg);

close(fd);

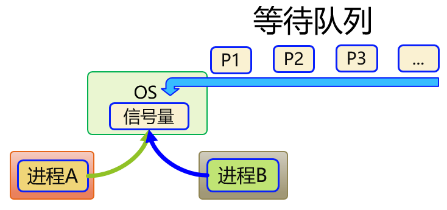
return 0;

}

作业：请编写AB两个程序，A进程向命名管道写入信息，B进程从命名管道读出信息。

### 4.4 Linux中信号量通信

多线程对象操作不受保护的普通的变量，将会导致不确定的操作（有多种可能的结果）。信号量是用来调协线程对共享资源的访问的。信号量是一个特殊类型的变量可以被增加或减少，OS控制访问信号量的过程是原子操作（不会被[线程调度](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B%E8%B0%83%E5%BA%A6/10226112?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E5%8E%9F%E5%AD%90%E6%93%8D%E4%BD%9C/_blank)机制打断），多个线程对信号量并发操作都转换为依次进行。但是信号量的值大于0表示可用的资源数，小于0表示阻塞的线程数。通过使用信号量可以很好的完成线程同步。信号量可看作多个进程互斥访问的一个整型变量。



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，保存在内存中一般用于父子进程或同一进程内的线程间。而有名信号量一般用于进程间同步或互斥，由操作系统用唯一的字符串或者整数标识，操作系统用特殊的文件来记录，用于进程间同步控制。信号量要引入下面的头文件：

#include <fcntl.h>

#include <sys/stat.h>

#include <semaphore.h>

#### 4.4.1无名信号量

linux中使用无名(匿名)信号量的方法如下：

|  |  |
| --- | --- |
| 方法名称 | 方法说明 |
| sem\_init | 创建信号量变量，value是信号量的初值 |
| sem\_post | 信号量值增加1，OS激活其它等待该信号量的线程 |
| sem\_wait | 为0时将调用该方法的线程被OS阻塞，大于0信号量值减少1，线程继续 |
| sem\_destroy | 销毁信号量 |

#include <semaphore.h>

int sem\_init(sem\_t \* sem, int pshared, unsigned int value);//创建信号量变量，value是信号量的初值

int sem\_destroy(sem\_t \*sem);//销毁信号量

int sem\_post(sem\_t \* sem); //信号量值增加1，并激活处于等待状态的线程

int sem\_wait(sem\_t \* sem); //信号量值减少1 为0时将调用该方法的线程被OS阻塞

信号量的value大于0，其他线程就可以sem\_wait成功，成功后信号量的value减1。若value值不大于0，则sem\_wait使得线程阻塞，直到sem\_post释放后value值加1,但是sem\_wait返回之前还是会将此value值减1。下面是使用无名信号量的例子程序：

//070.c semaphore.c 一个线程获取输入值，一个线程对值累加，不使用信号量时结果会是0，

//注意：因为pthread并非Linux系统的默认库，编译时注意加上-lpthread参数。

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

void \* read(void \* arg);

void \* accu(void \* arg);

static sem\_t sem\_one;

static sem\_t sem\_two;

static int num;

int main(int argc,char \* argv[])

{

pthread\_t id\_t1,id\_t2;

sem\_init(&sem\_one,0,0); //创建信号量变量，初值为0

sem\_init(&sem\_two,0,1); //创建信号量变量，初值为1

pthread\_create(&id\_t1,NULL,read,NULL); //创建线程，执行read

pthread\_create(&id\_t2,NULL,accu,NULL); //创建线程，执行accu

pthread\_join(id\_t1,NULL); //阻塞等待线程退出，获取线程退出状态

pthread\_join(id\_t2,NULL);

sem\_destroy(&sem\_one);

sem\_destroy(&sem\_two);

return 0;

}

void \* read(void \* arg)

{

int i;

for(i=0;i<5;i++)

{

fputs("Input num:",stdout);

sem\_wait(&sem\_two);

scanf("%d",&num);

sem\_post(&sem\_one);

}

return NULL;

}

void \* accu(void \*arg)

{

int sum=0,i;

for(i=0;i<5;i++)

{

sem\_wait(&sem\_two);

sum+=num;

sem\_post(&sem\_one);

}

printf("Result: %d \n",sum);

return NULL;

}

#### 4.4.2有名信号量

有名信号量在使用时会使用信号量文件，它的运用场景有：在亲缘(父子或兄弟)进程间同步或互斥功能，在不同的进程间实现同步功能，实现进程间共享内存读写数据的同步控制。在终端执行ipcs -m 命令查看共享内存的资源使用情况，使用 ls -al /dev/shm/ 查看有名信号量的使用情况。

|  |  |
| --- | --- |
| 方法名称 | 方法说明 |
| sem\_open | 创建信号量变量，value是信号量的初值 |
| sem\_post | 信号量值增加1，OS激活其它等待该信号量的线程 |
| sem\_wait | 为0时将调用该方法的线程被OS阻塞，大于0信号量值减少1，线程继续 |
| sem\_timedwait | 指定一个阻塞的时间上限。超时则返回失败 |
| sem\_trywait | 如果递减不能立即执行，调用将返回错误(errno设置为EAGAIN)而不阻塞 |
| sem\_unlink | 在系统中删除信号量 |
| sem\_close | 进程内部关闭一个信号量(系统中仍存在该文件) |
| sem\_destroy | 销毁信号量 |

下面用三个文件演示了有名信号量在进程间实现的同步控制，semcreate.c程序创建两个有名信号量，sempost.c程序对信号量增加值，semwait.c程序等待信号量状态的到达。编译命令为：

gcc -o sc semcreate.c -lpthread

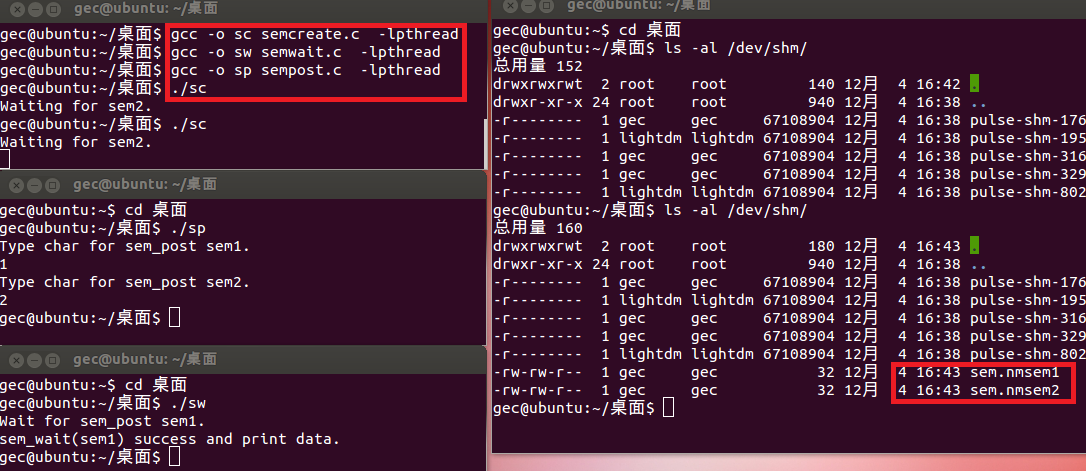
gcc -o sw semwait.c -lpthread

gcc -o sp sempost.c -lpthread

分别在三个命令窗口中运行三个程序，这时有三个独立的进程，再启动第四个命令窗口，查看有名信号量文件状态，在该窗口中输入命令：

ls -al /dev/shm/

semcreate进程创建信号量sem1和sem2并且设置初值为0，并等待sem2状态到达，semwait进程等待sem2信号量状态到达，sempost进程由用户输入来控制对sem1和sem2增加值，当sem1增值后，semwait进程继续并执行完毕。这时在第四个命令窗口再次输入命令可查看信号量文件的状态。sem2增值后，信号量文件被删除。通过有名信号量，sempost进程控制了semcreate和semwait进程的进度。运行结果如下图。



//071.c semcreate.c 创建有名信号量

#include<stdio.h>

#include<semaphore.h>

#include<fcntl.h>

#include<unistd.h>

#include<sys/stat.h>

#include<sys/types.h>

#include <sys/wait.h>

char semName1[]={"nmsem1"};

char semName2[]={"nmsem2"};

int main(int argc, char \*argv[])

{

sem\_t \*sem1 = NULL;//控制print进程运行的信号量

sem\_t \*sem2 = NULL;//控制创建进程结束的信号量

sem1 = sem\_open(semName1, O\_CREAT|O\_RDWR, 0666, 0); //信号量初值为0

if(sem1 == SEM\_FAILED){ //有名信号量创建失败

perror("sem\_open nmsem1");

return -1;

}

sem2 = sem\_open(semName2, O\_CREAT|O\_RDWR, 0666, 0); //信号量值为 0

if(sem2 == SEM\_FAILED){ //有名信号量创建失败

perror("sem\_open nmsem2");

return -1;

}

printf("Waiting for sem2.\n");

sem\_wait(sem2);//等待结束信号量

sem\_close(sem1); //关闭有名信号量

sem\_close(sem2); //关闭有名信号量

sem\_unlink(semName1);//删除有名信号量

sem\_unlink(semName2);//删除有名信号量

}

//072.c semwait.c 等待有名信号量

#include<stdio.h>

#include<semaphore.h>

#include<fcntl.h>

#include<unistd.h>

#include<sys/stat.h>

#include<sys/types.h>

#include <sys/wait.h>

char semName1[]={"nmsem1"};

int main(int argc, char \*argv[])

{

int i=0;

sem\_t \*sem1 = NULL;//控制print进程运行的信号量

sem1 = sem\_open(semName1, 0); //打开一个已有的信号量

if(sem1 == SEM\_FAILED){ //有名信号量创建失败

perror("sem\_open nmsem1");

return -1;

}

printf("Wait for sem\_post sem1.\n");

sem\_wait(sem1);

printf("sem\_wait(sem1) success and print data.\n");

sem\_close(sem1); //关闭有名信号量

}

//073.c sempost.c 给有名信号量增加值

#include<stdio.h>

#include<semaphore.h>

#include<fcntl.h>

#include<unistd.h>

#include<sys/stat.h>

#include<sys/types.h>

#include <sys/wait.h>

char semName1[]={"nmsem1"};

char semName2[]={"nmsem2"};

int main(int argc, char \*argv[])

{

int i=0;

sem\_t \*sem1 = NULL;//控制print进程运行的信号量

sem\_t \*sem2 = NULL;//控制创建进程结束的信号量

sem1 = sem\_open(semName1, 0); //打开一个已有的信号量

if(sem1 == SEM\_FAILED){ //有名信号量创建失败

perror("sem\_open nmsem1");

return -1;

}

sem2 = sem\_open(semName2, 0); //打开一个已有的信号量

if(sem2 == SEM\_FAILED){ //有名信号量创建失败

perror("sem\_open nmsem2");

return -1;

}

printf("Type char for sem\_post sem1.\n");

scanf("%d",&i);

sem\_post(sem1);

printf("Type char for sem\_post sem2.\n");

scanf("%d",&i);

sem\_post(sem2);

sem\_close(sem1); //关闭有名信号量

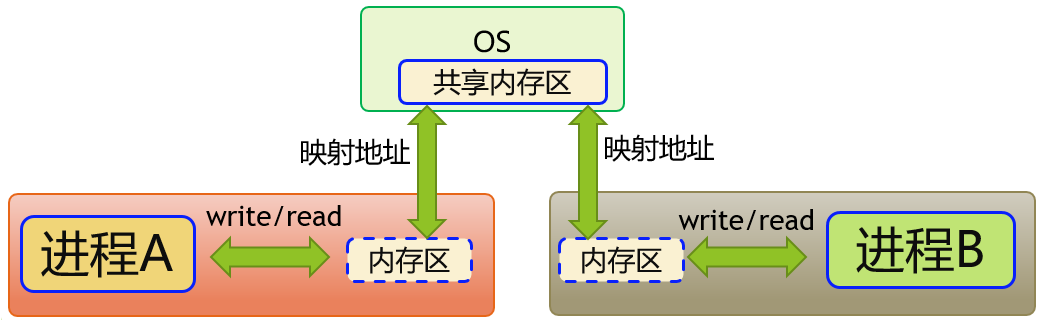
sem\_close(sem2); //关闭有名信号量

}

### 4.5共享内存通信

注：本小节内容参考链接：

https://blog.csdn.net/weixin\_30852367/article/details/97201723



共享内存是多个进程使用相同的内存区域，读写快速，效率高。不同进程之间共享的内存通常安排为同一段物理内存。进程可以将同一段共享内存连接到它们自己的地址空间中，所有进程都可以访问共享内存中的地址，就好像它们是由用C语言函数malloc()分配的内存一样。而如果某个进程向共享内存写入数据，所做的改动将立即影响到可以访问同一段共享内存的任何其他进程。共享内存虽然与管道有一定相似之处，但是也是有较大区别的，主要管道数据是有序读写的，且不能定位，理论上通过管道传输的数据量大小不受限制；共享内存在使用时则有容量的限制，一旦创建，其容量固定，但是这种方式可随意定位访问位置，具有内存访问的特性；管道传递数据要经过写操作和读操作降低了访问效率，多个进程间共享同一个内存实体，进程间数据传递效率是最高的。

共享内存并无安全保护机制，使用共享内存需结合信号量等同步机制保证访问安全。共享内存使用的主要方法如下表：

|  |  |
| --- | --- |
| 方法名称 | 方法说明 |
| shmget | 创建共享内存 |
| shmat | 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 |
| shmdt | 将共享内存从当前进程中分离 |
| shmctl | 控制共享内存 |

shmget方法声明：

int **[shmget](http://hi.baidu.com/zengzhaonong/blog/item/58d110b3c85007a5d9335a93.html)**(key\_t key, size\_t size, int shmflg);

size是要建立共享内存的长度。所有的内存分配操作都是以页(4K)为单位的。进程只申请一块只有一个字节的内存，内存也会分配整一页。

以两个不相关的进程来说明进程间如何通过共享内存来进行通信。其中一个文件shmread.c创建共享内存，并读取其中的信息，另一个文件shmwrite.c向共享内存中写入数据。为了方便操作和数据结构的统一，为这两个文件定义了相同的数据结构，定义在文件shmdata.c中。结构shared\_use\_st中的written作为一个可读或可写的标志，非0：表示可读，0：表示可写，text则是内存中的文件。

#### 4.5.1无访问互斥控制的共享内存读写

下面的代码实现的是无访问互斥控制的共享内存读写。

//shmdata.h

#ifndef \_SHMDATA\_H\_HEADER

#define \_SHMDATA\_H\_HEADER

#define TEXT\_SZ 2048

struct shared\_use\_st

{

int written;// 作为一个标志，非0：表示可读，0：表示可写

char text[TEXT\_SZ];// 记录写入 和 读取 的文本

};

#endif

//shmread.c

#include <stddef.h>

#include <sys/shm.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include "shmdata.h"

int main(int argc, char \*\*argv)

{

void \*shm = NULL;

struct shared\_use\_st \*shared; // 指向shm

int shmid; // 共享内存标识符

// 创建共享内存

shmid = shmget((key\_t)1234, sizeof(struct shared\_use\_st), 0666|IPC\_CREAT);

if (shmid == -1)

{

fprintf(stderr, "shmat failed\n");

exit(EXIT\_FAILURE);

}

// 将共享内存连接到当前进程的地址空间

shm = shmat(shmid, 0, 0);

if (shm == (void \*)-1)

{

fprintf(stderr, "shmat failed\n");

exit(EXIT\_FAILURE);

}

printf("\nMemory attached at %X\n", (int)shm);

// 设置共享内存

shared = (struct shared\_use\_st\*)shm;

// 注意：shm有点类似通过 malloc() 获取到的内存，所以这里需要做个 类型强制转换

shared->written = 0;

while (1) // 读取共享内存中的数据

{

// 没有进程向内存写数据，有数据可读取

if (shared->written == 1)

{

printf("You wrote: %s", shared->text);

sleep(1);

// 读取完数据，设置written使共享内存段可写

shared->written = 0;

// 输入了 end，退出循环（程序）

if (strncmp(shared->text, "end", 3) == 0)

{

break;

}

}

else // 有其他进程在写数据，不能读取数据

{

sleep(1);

}

}

// 把共享内存从当前进程中分离

if (shmdt(shm) == -1)

{

fprintf(stderr, "shmdt failed\n");

exit(EXIT\_FAILURE);

}

// 删除共享内存

if (shmctl(shmid, IPC\_RMID, 0) == -1)

{

fprintf(stderr, "shmctl(IPC\_RMID) failed\n");

exit(EXIT\_FAILURE);

}

exit(EXIT\_SUCCESS);

}

//shmwrite.c

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <sys/shm.h>

#include "shmdata.h"

int main(int argc, char \*\*argv)

{

void \*shm = NULL;

struct shared\_use\_st \*shared = NULL;

char buffer[BUFSIZ + 1]; // 用于保存输入的文本

int shmid;

// 创建共享内存

shmid = shmget((key\_t)1234, sizeof(struct shared\_use\_st), 0666|IPC\_CREAT);

if (shmid == -1)

{

fprintf(stderr, "shmget failed\n");

exit(EXIT\_FAILURE);

}

// 将共享内存连接到当前的进程地址空间

shm = shmat(shmid, (void \*)0, 0);

if (shm == (void \*)-1)

{

fprintf(stderr, "shmat failed\n");

exit(EXIT\_FAILURE);

}

printf("Memory attched at %X\n", (int)shm);

// 设置共享内存

shared = (struct shared\_use\_st \*)shm;

while (1) // 向共享内存中写数据

{

// 数据还没有被读取，则等待数据被读取，不能向共享内存中写入文本

while (shared->written == 1)

{

sleep(1);

printf("Waiting...\n");

}

// 向共享内存中写入数据

printf("Enter some text: ");

fgets(buffer, BUFSIZ, stdin);

strncpy(shared->text, buffer, TEXT\_SZ);

// 写完数据，设置written使共享内存段可读

shared->written = 1;

// 输入了end，退出循环（程序）

if (strncmp(buffer, "end", 3) == 0)

{

break;

}

}

// 把共享内存从当前进程中分离

if (shmdt(shm) == -1)

{

fprintf(stderr, "shmdt failed\n");

exit(EXIT\_FAILURE);

}

sleep(2);

exit(EXIT\_SUCCESS);

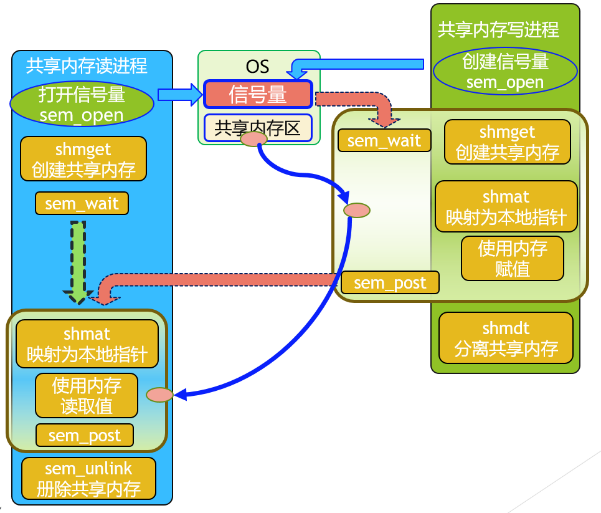
}

1、程序shmread创建共享内存，然后将它连接到自己的地址空间。在共享内存的开始处使用了一个结构struct\_use\_st。该结构中有个标志written，当共享内存中有其他进程向它写入数据时，共享内存中的written被设置为0，程序等待。当它不为0时，表示没有进程对共享内存写入数据，程序就从共享内存中读取数据并输出，然后重置设置共享内存中的written为0，即让其可被shmwrite进程写入数据。

2、程序shmwrite取得共享内存并连接到自己的地址空间中。检查共享内存中的written，是否为0，若不是，表示共享内存中的数据还没有被读完，则等待其他进程读取完成，并提示用户等待。若共享内存的written为0，表示没有其他进程对共享内存进行读取，则提示用户输入文本，并再次设置共享内存中的written为1，表示写完成，其他进程可对共享内存进行读操作。

#### 4.5.2应用有名信号量来实现共享内存读写数据同步

多个进程同时对共享内存区写数据，没有使用信号量来进行同步控制，可能会造成不可预测的错误，可使用有名信号量来实现共享内存读写数据的同步。下面有两个例子程序sharemm-nm-read.c 和sharemm-nm-write.c 两个程序使用约定的键值创建共享内存，sharemm-nm-write.c创建有名信号量并先往共享内存写数据，sharemm-nm-read.c进程会等待直到有数据写入共享内存，获得信号后读取共享内存的数据，从而达到同步的效果。读完数据之后，释放资源，关闭有名信号量，删除有名信号量，把共享内存从当前进程分离，再删除共享内存。执行ipcs -m 命令查看共享内存的资源使用情况，使用 ls -al /dev/shm/ 查看有名信号量的使用情况。所有的资源都被释放了。注意：该项目测试时要求sharemm-nm-write进程先执行，sharemm-nm-read.c进程后执行，否则因为没有先创建信息量变量造成程序无法运行。



//074.c sharemm-nm-read.c 有名信号量控制共享内存的使用2

#include <unistd.h>

#include <stdio.h>

#include<stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <pthread.h>

#include <semaphore.h>

#include <sys/shm.h>

#include <string.h>

#include <stddef.h>

#define BUFF 128

char semName1[]={"write\_nmsem1"};

typedef struct{

char name[100];

int age;

} PersonInfo;

int main ()

{

int count=0;

char write\_buf[] = "2";

int i = 0;

sem\_t \*sem\_1 = NULL;

sem\_1 = sem\_open(semName1, 0); //打开一个已有的信号量

if((sem\_1 == SEM\_FAILED))

{

perror("sharemm read sem\_open");

exit(-1);

}

int shmid;

PersonInfo \*p\_map;

// 使用约定的键值创建共享内存[推荐]，也可以使用ftok函数生成一个key

if((shmid=shmget((key\_t) 1234,BUFF, 0666|IPC\_CREAT))<0)

{

perror("share mm read shmget().");

exit(-1);

}

else

printf("Create shared memory，id = %d\n",shmid);

printf("Wait semaphore to go on ....\n");

/\*须获取信号量后才可对内存进行读操作\*/

sem\_wait(sem\_1);

/\*映射共享内存到本进程地址空间\*/

p\_map=(PersonInfo\*)shmat(shmid,NULL,0);

if(p\_map<(PersonInfo \*)0)

{

perror("shmat");

exit(-1);

}

else

printf("process 2 shmat shared memory success\n");

//读取共享内存的数据

printf("share memory p\_map->name: %s\n",p\_map->name);

printf("share memory p\_map->age : %d\n",p\_map->age);

sem\_post(sem\_1);

sem\_close(sem\_1); //关闭有名信号量 sem\_1

//删除有名信号量

if(sem\_unlink(semName1)< 0)

{

perror("sem\_unlink");

}

printf("remove semName1\n");//删除有名信号量 sem\_1

printf("detach shmaddr\n");

//把共享内存从当前进程中分离

if(shmdt(p\_map) == -1)

{

perror("shmdt");

exit(-1);

}

//删除共享内存

if(shmctl(shmid, IPC\_RMID, 0) == -1)

{

perror("shmctl");

exit(-1);

}

printf("read 进程删除了共享内存.\n");

printf("it is end!\n");

return 0;

}

//075.c sharemm-nm-write.c 有名信号量控制共享内存的使用

#include <unistd.h>

#include <stdio.h>

#include<stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <pthread.h>

#include <semaphore.h>

#include <sys/shm.h>

#include <string.h>

#define BUFF 128

char semName1[]={"write\_nmsem1"};

typedef struct{

char name[100];

int age;

} PersonInfo;

int main ()

{

int count=0;

char write\_buf[] = "helloworld";

int i = 0;

sem\_t \*sem\_1 = NULL;

sem\_1 = sem\_open(semName1, O\_CREAT|O\_RDWR, 0666, 1); //创建有名信号量初值为1

if(sem\_1 == SEM\_FAILED)

{

perror("sem\_open");

exit(-1);

}

/\*信号量减一，P 操作\*/

sem\_wait(sem\_1);

int shmid;

char \*shmaddr;//共享内存地址

PersonInfo \*p\_map;

// 使用约定的键值创建共享内存[推荐]，也可以使用ftok函数生成一个key

if((shmid=shmget((key\_t) 1234,BUFF, 0666|IPC\_CREAT))<0)

{

perror("share mm write shmget");

exit(-1);

}

else

printf("Create shared memory，id = %d\n",shmid);

/\*映射共享内存\*/

p\_map=(PersonInfo\*)shmat(shmid,NULL,0);

if(p\_map<(PersonInfo \*)0)

{

perror("shmat");

exit(-1);

}

else

printf("process 1 shmat shared memory success\n");

printf("Press one string,it will be written into shared memory.\n");

char str[200];

memset(str,0,200);

//输入包含空格的字符串，回车符作为结束

scanf("%[^\n]%\*c",str);

sprintf(p\_map->name,"%s",str);

p\_map->age=100;

/\*信号量加一，V 操作\*/

sem\_post(sem\_1);

sem\_close(sem\_1); //关闭有名信号量 sem\_1

//将共享内存从当前进程中分离

if(shmdt(p\_map) == -1)

{

perror(" detach error ");

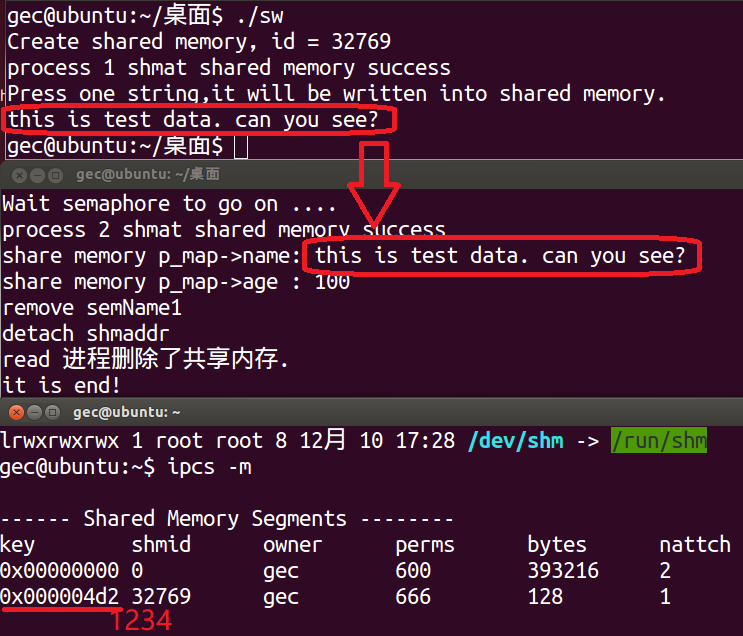
exit(-1);

}

return 0;

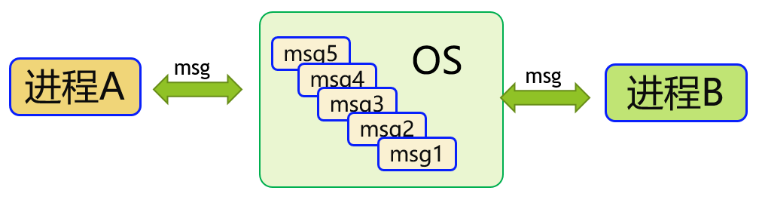
}

程序运行示例如下，进程sw的数据通过共享内存进入到进程sr中，0x4d2则是共享内存的标识号。



### 4.6 Linux中消息队列实现进程间通信

消息队列是通过链表结构组的一组消息，多个进程通过内核管理的消息队列实现进程间通信，消息队列中消息是有序存取。



1 消息的创建

unsigned int threadType;

key\_t key = 1050 + threadType;

msgget(key, IPC\_CREAT|0666));

2 发送消息

msgsnd(unsigned int msgid, void \* msg, unsigned int msgsize,IPC\_NOWAIT);

3 接收消息

msgrcv(msgid, &;msg, sizeof(msg)-sizeof(long), 0,MSG\_NOERROR);

从标识符为msgid的消息队列里接收一个指定类型的消息并存储于msgp中 读取后把消息从消息队列中删除。

下面是线程间通信用的消息队列，有一定的修改。

//079.c mq-send.c 《Linux C从入门到精通》page122

//消息队列发送端。

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/msg.h>

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main(void)

{

key\_t key;

int proj\_id=1;

int msqid;

char message1[]={"hello mrsoft!"};

char message2[]={"goodbye!"};

struct msgbuf

{

long msgtype;

char msgtext[1024];

}snd;

key=ftok("/home/gec",'a');

if(key==-1)

{

perror("create key error!");

return 1;

}

//创建或打开一个消息队列

if((msqid=msgget(key,IPC\_CREAT|0666))==-1)

{

printf("msgget error!\n");

exit(1);

}

//向消息队列发送一个消息message1

snd.msgtype=1;

sprintf(snd.msgtext,"%s",message1);

if(msgsnd(msqid,&snd,sizeof(snd),0)==-1)

{

printf("msgsnd error!\n");

}

//向消息队列发送一个消息message2

snd.msgtype=2;

sprintf(snd.msgtext,"%s",message2);

if(msgsnd(msqid,&snd,sizeof(snd),0)==-1)

{

printf("msgsnd error!\n");

}

//msgctl(msgid,IPC\_RMID,0);

system("ipcs -q");

exit(0);

}

//080.c 消息队列接收端。

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/msg.h>

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main(void)

{

key\_t key;

int proj\_id=1;

int msqid;

char message1[]={"hello mrsoft!"};

char message2[]={"goodbye!"};

struct msgbuf

{

long msgtype;

char msgtext[1024];

}rcv;

key=ftok("/home/gec",'a');

if(key==-1)

{

perror("create key error!");

return 1;

}

//创建或打开一个消息队列

if((msqid=msgget(key,IPC\_CREAT|0666))==-1)

{

printf("msgget error!\n");

exit(1);

}

printf("msqid=%d,key=%x\n",msqid,key);

if(msgrcv(msqid,&rcv,sizeof(rcv),0,IPC\_NOWAIT)==-1)

{

printf("msgrcv error!\n");

exit(1);

}

printf("the received message:%s.\n",rcv.msgtext);

//msgctl(msgid,IPC\_RMID,0);

system("ipcs -q");

exit(0);

}

## 五、程序调试中遇到的问题和解决过程及运行结果

编译和运行实验程序，排查程序出错的原因，对实验过程中的问题截图进行说明和总结。