实验七 进程间通信（二）

一、实验目的

1、 了解共享内存、消息队列、信号量的概念；

2、 掌握利用共享内存、消息队列、信号量实现进程间通信的编程方法；

二、实验准备

拷贝 lab7 中文件（课堂上提供）到~/raspberry/prj_os/lab7，

执行命令

make

编译；

三、实验内容

共享内存(shared memory)、消息队列(message queue)、信号量(semaphore)是存在于 Linux 内核之中的三种 IPC 对象；

多个进程可以操作同一个 IPC 对象，因为 IPC 对象在内核中，而不是在进程的 4GB 虚拟内存之中，所以 IPC 对象可以作为进程间相互通信的方式；

Shell 下运行命令

ipcs –a

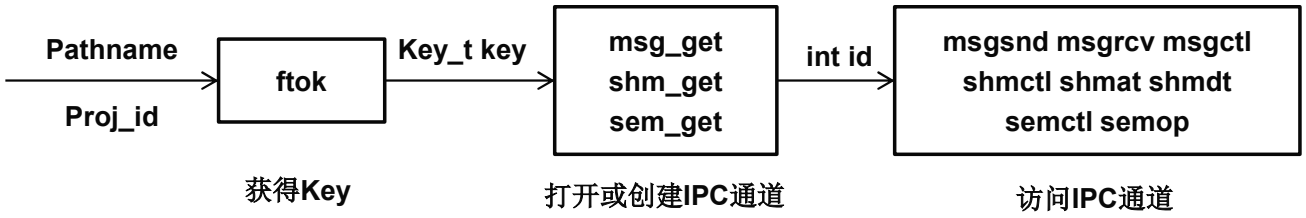
可以查看上述三种对象；

```
yangyb@ubuntu:~$ ipcs -a

                                    共享内存
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 1114112    yangyb     600        524288     2          dest
0x00000000 393217     yangyb     600        524288     2          dest
0x00000000 425986     yangyb     600        524288     2          dest
0x00000000 622595     yangyb     600        524288     2          dest
0x00000000 1015812    yangyb     600        524288     2          dest
0x00000000 819205     yangyb     600        524288     2          dest
0x00000000 851974     yangyb     600        16777216   2
0x00000000 884743     yangyb     600        33554432   2          dest
0x00000000 1146888    yangyb     600        1048576    2          dest
0x00000000 1310729    yangyb     600        524288     2          dest
0x00000000 1572875    yangyb     600        524288     2          dest

                          信号量
------ Semaphore Arrays --------
key        semid      owner      perms      nsems

                          消息队列
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

每个 IPC 对象都有一个唯一的编号，该编号是由系统分配的。不同的进程如何知道这个编号，进而通过它进行通信呢？以共享内存为例：

假设，进程 p1 创建了共享内存。可以在创建时，调用 ftok 函数，得到一个 key 值，调用 shmget()函数，该函数会返回所创建共享内存的编号，并将 key 和编号关联起来。若进程 p2 想利用这个共享内存和 p1 进程

通信，也可以调用 ftok 函数，得到同样的 key，再根据 key 值，调用 shmget 函数，就可以获得该共享内存的编号。该过程可以通过下面的图来表达。



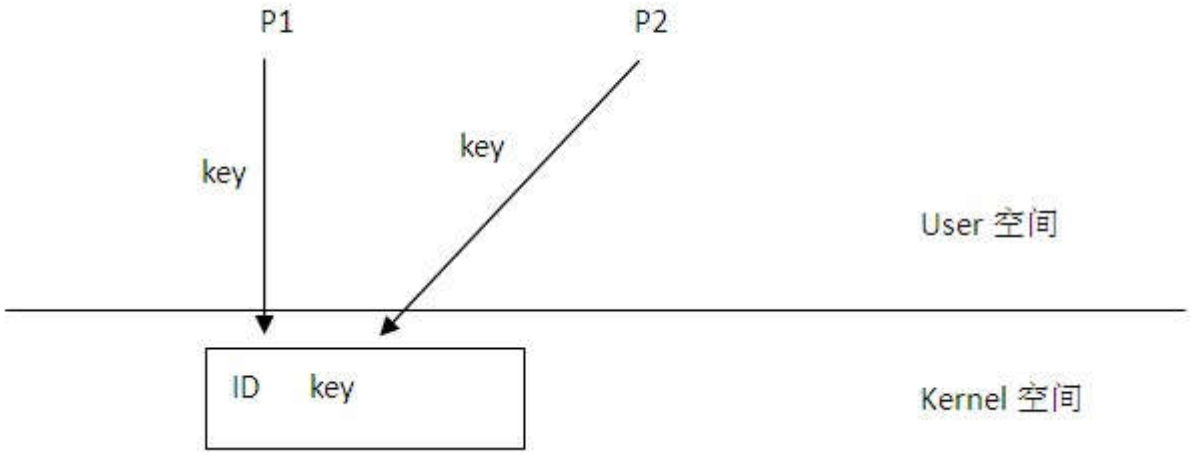
- ftok()函数

所需头文件	#include <sys/types.h> #include <sys/ipc.h>
函数原型	key_t ftok(const char *pathname, int proj_id);
函数传入值	pathname：是一个存在的文件或目录名；
	proj_id：是非 0 整数；
函数返回值	成功：返回一个键值 Key
	出错：-1

函数说明：

The ftok() function uses the identity of the file named by the given pathname (which must refer to an existing, accessible file) and the least significant 8 bits of proj_id (which must be nonzero) to generate a key_t type System V IPC key, suitable for use with msgget(2), semget(2), or shmget(2).

该函数会返回一个 key 值，先运行的进程根据 key 来创建 IPC 对象，后运行的进程根据 key 来打开 IPC 对象。示意图如下：

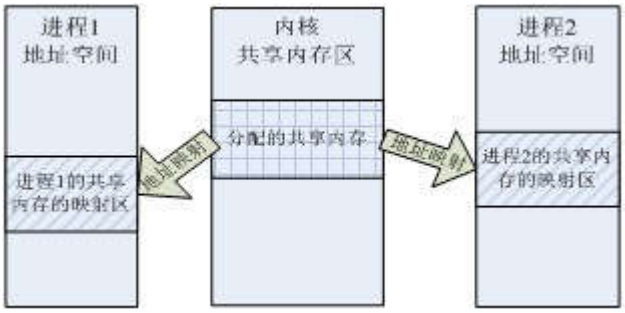


使用 ftok 创建共享内存，毫无关系的进程，可以通过得到同样的 key，来操作同一个共享内存，对共享内存进行读写时，需要利用信号量进行同步或互斥。

说明：调用 ftok()函数获得的 key 对共享内存、消息队列、信号量都适用；

（一） 共享内存

共享内存是一种最为高效的进程间通信方式，因为进程可以直接读写内存，不需要任何数据的复制。为了在多个进程间交换信息，内核专门留出了一块内存区，这段内存区可以由需要访问的进程将其映射到自己的私有地址空间。因此，进程就可以直接读写这一内存区而不需要进行数据的复制，从而大大提高了效率。当然，由于多个进程共享一段内存，因此也需要依靠某种同步机制，如互斥锁和信号量等。其原理示意图如下图所示。



共享内存的实现分为两个步骤：第一步是创建共享内存，这里用到的函数是 shmget()，也就是从内存中获得一段共享内存区域；第二步是映射共享内存，也就是把这段创建的共享内存映射到具体的进程空间中，这里使用的函数是 shmat()。到这里，就可以使用这段共享内存了，也就是可以使用不带缓冲的 I/O 读写命令对其进行操作。除此之外，还有撤销映射的操作，其函数为 shmdt()。

- 获得/创建共享内存函数 shmget()

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmget(key_t key, size_t size, int shmflg)
函数传入值	key：共享内存的键值，多个进程可以通过它访问同一个共享内存，其中有个特殊值 IPC_PRIVATE，用于创建当前进程的私有共享内存
	size：共享内存区大小
	shmflg：同 open()函数的权限位，也可以用八进制表示法
函数返回值	成功：共享内存段标识符
	出错：-1

函数说明：

shmget() returns the identifier of the System V shared memory segment associated with the value of the argument key. A new shared memory segment, with size equal to the value of size rounded up to a multiple of PAGE_SIZE, is created if key has the value IPC_PRIVATE or key isn't IPC_PRIVATE, no shared memory segment corresponding to key exists, and IPC_CREAT is specified in shmflg.

调用 shmget()函数获得与 key 关联的共享内存 ID；若 shmflag 中指定了 IPC_CREAT，则当不存在与 Key 关联的共享内存时，会新建一个共享内存；

- 共享内存映射/解除映射函数 shmat()/shmdt()

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	char *shmat(int shmid, const void *shmaddr, int shmflg)	
函数传入值	shmid：要映射的共享内存区标识符	
	shmaddr：将共享内存映射到指定地址（若为 NULL 则表示系统自动分配地址并把该段共享内存映射到调用进程的地址空间）	
	shmflg	SHM_RDONLY：共享内存只读
		默认 0：共享内存可读写
函数返回值	成功：被映射的段地址	
	出错：-1	

函数说明：

shmat() attaches the System V shared memory segment identified by shmid to the address space of the calling process. The attaching address is specified by shmaddr with one of the following criteria:

If shmaddr is NULL, the system chooses a suitable (unused) address at which to attach the segment.

On success shmat() returns the address of the attached shared memory segment; on error (void *) -1 is returned, and errno is set to indicate the cause of the error.

shmat()函数将共享内存映射到进程 4GB 虚拟空间的某个地址，该地址作为函数的返回值返回；然后该进程就可以采用所映射的地址来访问共享内存；

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmdt(const void *shmaddr)
函数传入值	shmaddr：被映射的共享内存段地址
函数返回值	成功：0
	出错：-1

函数说明：

shmdt() detaches the shared memory segment located at the address specified by shmaddr from the address space of the calling process. The to-be-detached segment must be currently attached with shmaddr equal to the value returned by the attaching shmat() call.

On success shmdt() returns 0; on error -1 is returned, and errno is set to indicate the cause of the error.

shmdt()实现 shmat()相反的功能，也就是解除 shmat()所建立的共享内存和进程某个虚拟地址的映射关系；

- 共享内存管理函数 shmctl()

所需头文件	#include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int shmctl(int shmid, int cmd, struct shmid_ds *buf)	
函数传入值	msqid：共享内存区标识符 ID	
	cmd：命令参数	IPC_STAT：读取消息队列的数据结构 shmid_ds，并将其存储在 buf 指定的地址中
		IPC_SET：设置消息队列的数据结构 shmid_ds 中的 ipc_perm 域（IPC 操作权限描述结构）值，这个值取自 buf 参数
		IPC_RMID：从系统内核中删除消息队列
	buf：描述消息队列的 shmid_ds 结构类型变量	
函数返回值	成功：0	
	出错：-1	

函数说明：

shmctl() performs the control operation specified by cmd on the System V shared memory segment whose identifier is given in shmid.

shmctl()用于对共享内存进行管理和控制，比如删除共享内存，此时 cmd 参数填 IPC_RMID，如下：

shmctl(shm_id,IPC_RMID,NULL)

- 示例

如下示例中，write()进程创建共享内存，向共享内存写数据；read()进程从共享内存读出 write()进程写入的数据；

同时运行 write()和 read()进程，可以看到 write()进程写入的数据被 read()进程正确读出显示；

另外还可以看到，共享内存里面的数据被 read 进程反复读出；如果我们希望 write 进程写一个数据，read 进程相应读一个数据，则需要对程序做出改进或采用别的方法；

```
yangyb@ubuntu:~/raspberry/prj_os/lab7/shm$ ./write
hello
world
quit
yangyb@ubuntu:~/raspberry/prj_os/lab7/shm$
```

```
yangyb@ubuntu:~/raspberry/prj_os/lab7/shm$ ./read
hello
hello
hello
hello
hello
hello
world
world
world
world
yangyb@ubuntu:~/raspberry/prj_os/lab7/shm$
```

write.c 源代码及说明如下：

```c
int main(void)
{
        int shm_id;
        key_t key;
        char *p;
        //获取key
        if((key = ftok("./",0xa)) < 0){
          perror("ftok");
          exit(1);
        }
        //获得共享内存
        if((shm_id = shmget(key,SHM_SIZE,IPC_CREAT|0666)) < 0){
          perror("shmget");
          exit(1);
        }

        //将共享内存映射到当前进程的虚拟空间中
         if( (p   = shmat(shm_id,NULL,0)) == NULL){
          perror("shmat");
          exit(1);
         }
         while(1){
           fgets(p,SHM_SIZE,stdin);
           if(strncmp(p,"quit",4) == 0)
           break;
         }

        if(shmdt(p) < 0){
          perror("shmdt");
          exit(1);
        }
        return 0;
}
```

这里的P就是共享内存映射到write进程4GB虚拟内存空间的地址，在write进程中访问P就是访问共享内存

这里写P就是写共享内存

read.c 源代码如下：

```c
int main(void)
{
        int shm_id;
        key_t key;
        char *p;

        //获取key
        if((key = ftok("./",0xa)) < 0){
          perror("ftok");
          exit(1);
        }

        //获得共享内存
        if((shm_id = shmget(key,SHM_SIZE,IPC_CREAT|0666)) < 0){
          perror("shmget");
          exit(1);
        }

        //将共享内存映射到当前进程的虚拟空间中
         if( (p   = shmat(shm_id,NULL,0)) == NULL){
          perror("shmat");
          exit(1);
         }
         while(1){
          if(strncmp(p,"quit",4) == 0)
          break;
          printf("%s",p);
          sleep(1);
         }

        if(shmdt(p) < 0){
          perror("shmdt");
          exit(1);
        }
        return 0;
}
```

这里的P就是共享内存映射到read进程4GB虚拟内存空间的地址，在read进程中访问地址P就是访问对应的共享内存

这里读P就是读共享内存

读写进程结束后，可以通过 shmctl()进程删除共享内存；
shmctl()程序源代码如下，里面调用了 shmctl()函数；

```
int main(int argc,char **argv)
{
        int shm_id;

        if(argc != 2){
          fprintf(stderr,"Usage:%s <shmid>\n",argv[0]);
          exit(1);
        }
        shm_id = atoi(argv[1]);
        //删除共享内存
        if(shmctl(shm_id,IPC_RMID,NULL) < 0){
          perror("shmget");
          exit(1);
        }
        return 0;
}
```

cmd=IPC_RMID
表示删除IPC对象

首先用命令 ipcs -m 查看一次共享内存，其中 shmid 为 2588682 的共享内存就是在上述读写测试中创建的共享内存；然后运行./shmctl 2588682，再次查看，发现这时 shmid 为 2588682 的共享内存已被删除；

```
yangyb@ubuntu:~/raspberry/prj_os/lab7/shm$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 2326528    yangyb     600        524288     2          dest
0x00000000 1605633    yangyb     600        524288     2          dest
0x00000000 1638402    yangyb     600        524288     2          dest
0x00000000 1835011    yangyb     600        524288     2          dest
0x00000000 2129924    yangyb     600        67108864   2          dest
0x00000000 2031622    yangyb     600        524288     2          dest
0x00000000 2064391    yangyb     600        16777216   2
0x00000000 2228232    yangyb     600        524288     2          dest
0x00000000 2359305    yangyb     600        1048576    2          dest
0x0a010039 2588682    yangyb     666        1024       0
0x00000000 2555915    yangyb     600        524288     2          dest

yangyb@ubuntu:~/raspberry/prj_os/lab7/shm$ ./shmctl 2588682
yangyb@ubuntu:~/raspberry/prj_os/lab7/shm$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 2326528    yangyb     600        524288     2          dest
0x00000000 1605633    yangyb     600        524288     2          dest
0x00000000 1638402    yangyb     600        524288     2          dest
0x00000000 1835011    yangyb     600        524288     2          dest
0x00000000 2129924    yangyb     600        67108864   2          dest
0x00000000 2031622    yangyb     600        524288     2          dest
0x00000000 2064391    yangyb     600        16777216   2
0x00000000 2228232    yangyb     600        524288     2          dest
0x00000000 2359305    yangyb     600        1048576    2          dest
0x00000000 2555915    yangyb     600        524288     2          dest

yangyb@ubuntu:~/raspberry/prj_os/lab7/shm$
```

（二） 消息队列

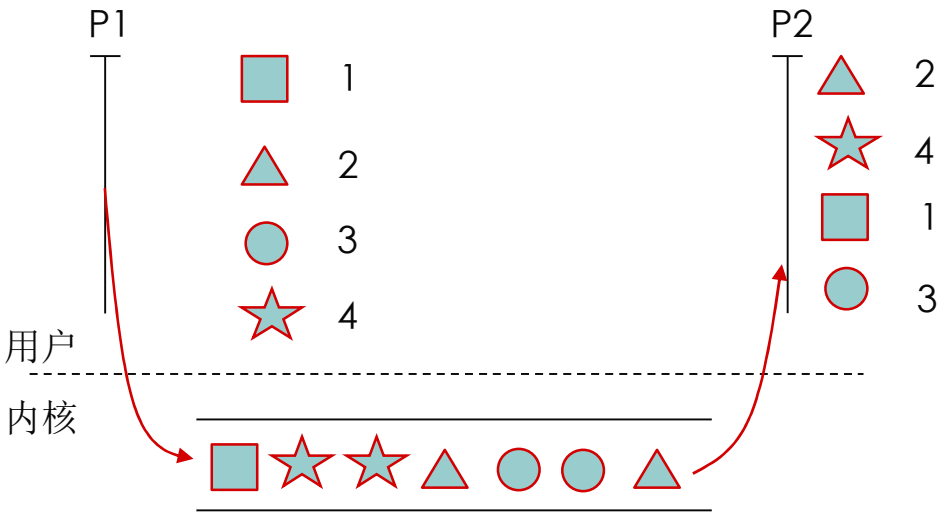
- 消息队列是 IPC 对象的一种；
- 消息队列由消息队列 ID 来唯一标识；
- 消息队列是消息的列表；用户可以在消息队列中添加消息、读取消息等；
- 消息队列可以按照类型来发送/接收消息；

消息队列中的消息是个结构体，包含消息类型 mtype 和 消息正文 mtext 2 个元素；

```
struct msgbuf {
        long mtype;       /* message type, must be > 0 */
        char mtext[1];    /* message data */
};
```

进程间使用消息队列进行通信的步骤：

1） 获得 Key
2） 获得/创建消息队列，msgget()函数
3） 发送/接收消息，msgsnd()和 msgrcv()函数
4） 解除映射；
5） 删除消息队列，msgctl()函数；



如上图，接收进程在接收消息时，要指定所接收消息的类型；如果同一个类型对应有多个消息，遵循先进先出的原则；

- 获得/创建消息队列函数 msgget()

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int msgget(key_t key, int msgflg)
函数传入值	key：消息队列的键值，多个进程可以通过它访问同一个消息队列，其中有个特殊值 IPC_PRIVATE，用于创建当前进程的私有消息队列
	msgflg：权限标志位

函数返回值	成功：消息队列 ID
	出错：-1

调用 msgget()函数获得与 key 关联的消息队列 ID；若 msgflag 中指定了 IPC_CREAT，则当不存在与 Key 关联的消息队列时，会新建一个消息队列；

- 发送/接收消息函数 msgsnd() / msgrcv()

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)	
函数传入值	msqid：消息队列的队列 ID	

	msgp：指向消息结构的指针，该消息结构 msgbuf 通常如下。
	struct msgbuf
	{
	long mtype; /* 消息类型，该结构必须从这个域开始 */
	char mtext[1]; /* 消息正文 */
	}	

	msgsz：消息正文的字节数（不包括消息类型指针变量）	
	msgflg	IPC_NOWAIT：若消息无法立即发送（如当前消息队列已满），函数会立即返回
		0：msgsnd 调用阻塞直到发送成功为止
函数返回值	成功：0	
	出错：-1	

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg)	
函数传入值	msqid：消息队列的队列 ID	
	msgp：消息缓冲区，同 msgsnd()函数的 msgp	
	msgsz：消息正文的字节数（不包括消息类型指针变量）	
	msgtyp	0：接收消息队列中第一个消息
		大于 0：接收消息队列中第一个类型为 msgtyp 的消息
		小于 0：接收消息队列中第一个类型值不大于 msgtyp 绝对值且类型值最小的消息
函数传入值	msgflg	MSG_NOERROR：若返回的消息比 msgsz 字节多，则消息就会截短到 msgsz 字节，且不通知消息发送进程
		IPC_NOWAIT：若在消息队列中并没有相应类型的消息可以接收，则函数立即返回
		0：msgsnd()调用阻塞直到接收一条相应类型的消息为止
函数返回值	成功：0	

	出错：-1

The msgsnd() and msgrcv() system calls are used, respectively, to send messages to, and receive messages from, a System V message queue. The calling process must have write permission on the message queue in order to send a message, and read permission to receive a message.

The msgsnd() system call appends a copy of the message pointed to by msgp to the message queue whose identifier is specified by msqid.
The msgrcv() system call removes a message from the queue specified by msqid and places it in the buffer pointed to by msgp.

消息队列独立于进程存在，进程结束，消息队列依然存在；

- 控制消息队列函数 msgctl()

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgctl (int msgqid, int cmd, struct msqid_ds *buf)	
函数传入值	msqid：消息队列的队列 ID	
	cmd：命令参数	IPC_STAT：读取消息队列的数据结构 msqid_ds，并将其存储在 buf 指定的地址中
		IPC_SET：设置消息队列的数据结构 msqid_ds 中的 ipc_perm 域（IPC 操作权限描述结构）值，这个值取自 buf 参数
		IPC_RMID：从系统内核中删除消息队列
	buf：描述消息队列的 msqid_ds 结构类型变量	
函数返回值	成功：0	
	出错：-1	

msgctl() performs the control operation specified by cmd on the System V message queue with identifier msqid.
当 cmd 取值 IPC_RMID 时，msgctl()的作用是删除 ID 号为 msqid 的消息队列；

➢ 示例

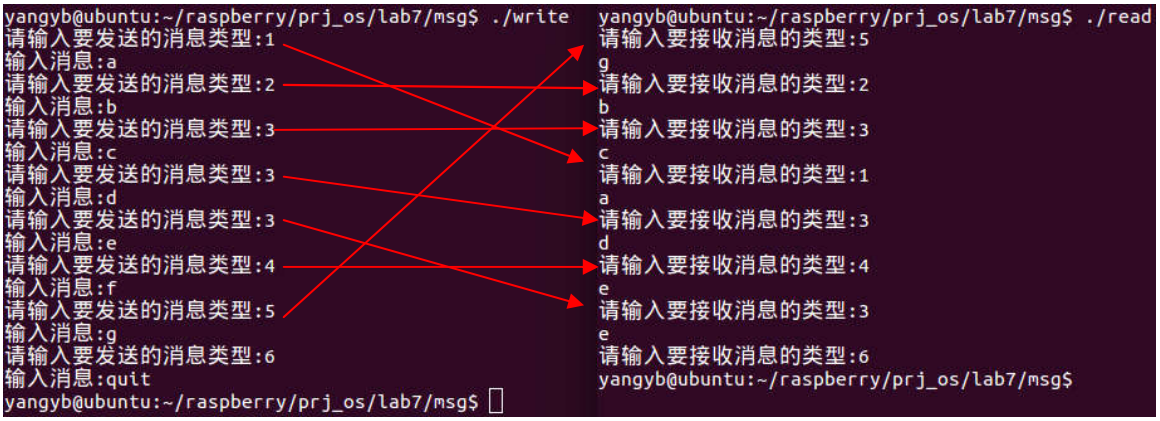
write.c 源代码：

```c
int main(void)
{
        int msg_id;
        key_t key;
        //获取key
        if((key = ftok("./",0xa)) < 0){
          perror("ftok");
          exit(1);
        }

        //获得消息队列
        if((msg_id = msgget(key,IPC_CREAT|0666)) < 0){
          perror("msgget");
          exit(1);
        }

        //发送消息
        struct msgbuf m;
         while(1){
           bzero(&m,sizeof(m));
           printf("请输入要发送的消息类型:");
           scanf("%ld",&m.mtype);
           while(getchar() != '\n');
           printf("输入消息:");
           fgets(m.mtext,MSG_SIZE,stdin);
           if(msgsnd(msg_id,&m,strlen(m.mtext),0) < 0){
             perror("msgsnd");
             exit(1);
           }
           if(strncmp(m.mtext,"quit",4) == 0)
             break;
         }

        return 0;
}
```

read.c 源代码：

```c
int main(void)
{
        int msg_id;
        key_t key;
        //get key
        if((key = ftok("./",0xa)) < 0){
          perror("ftok");
          exit(1);
        }

        //get message queue
        if((msg_id = msgget(key,IPC_CREAT|0666)) < 0){
          perror("msgget");
          exit(1);
        }

        //receive message
        struct msgbuf m;
        while(1){
          bzero(&m,sizeof(m));
          printf("请输入要接收消息的类型:");
          scanf("%ld",&m.mtype);
          if(msgrcv(msg_id,&m,MSG_SIZE,m.mtype,0) < 0){
            perror("msgget");
            exit(1);
          }
          if(strncmp(m.mtext,"quit",4) == 0)
          break;
          printf("%s",m.mtext);
        }

        return 0;
}
```

分别在 2 个 terminal 窗口运行 wirte() 和 read()程序，可以看到，read 进程是根据消息类型来接收消息的，当同一个消息类型有多个消息时，按先后顺序依次接收消息；



读写进程结束后，可以通过 msgctl()进程删除共享内存；
msgctl.c 源代码如下，大家可自行测试一下；

```c
int main(int argc,char **argv)
{
        int msg_id;

        if(argc != 2){
          fprintf(stderr,"Usage:%s <msgid>\n",argv[0]);
          exit(1);
        }
        msg_id = atoi(argv[1]);
        //删除消息队列
        if(msgctl(msg_id,IPC_RMID,NULL) < 0){
          perror("msgget");
          exit(1);
        }
        return 0;
}
```

（三） 信号量（信号量）

在多任务操作系统环境下，多个进程会同时运行，并且一些进程间可能存在一定的关联。多个进程可能为了完成同一个任务相互协作，这就形成了进程间的**同步关系**。而且在不同进程间，为了争夺有限的系统资源（硬件或软件资源）会进入竞争状态，这就是进程间的**互斥关系**。

进程间的互斥关系与同步关系存在的根源在于**临界资源**。临界资源是在同一个时刻只允许有限个（通常只有一个）进程可以访问（读）或修改（写）的资源，通常包括硬件资源（处理器、内存、存储器及其他外围设备等）和软件资源（共享代码段、共享结构和变量等）。访问临界资源的代码叫做临界区，临界区本身也会成为临界资源。

信号量是用来解决进程间的同步与互斥问题的一种进程间通信机制，包括一个称为信号量的变量和在该信号量下等待资源的进程等待队列，以及对信号量进行的两个原子操作（**PV 操作**）。其中信号量对应于某一种资源，取一个非负的整型值。信号量值指的是当前可用的该资源的数量，若等于 0 则意味着目前没有可用的资源。

PV 原子操作的具体定义如下:

● P 操作（等待操作）：如果有可用的资源（信号量值>0），则占用一个资源（给信号量值减 1，进入临界区代码）；如果没有可用的资源（信号量值=0），则被阻塞直到系统将资源分配给该进程（进入等待队列，一直等到资源轮到该进程）。

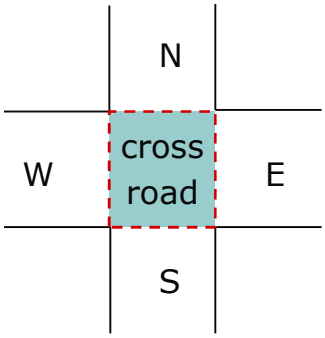
● V 操作（释放操作）：如果在该信号量的等待队列中有进程在等待资源，则唤醒一个阻塞进程；如果没有进程等待它，则释放一个资源（给信号量值加 1）。

常见的使用信号量访问临界区的伪代码如下：

```
{
    /* 设 R 为某种资源，S 为资源 R 的信号量 */
    INIT_VAL(S); /* 对信号量 S 进行初始化 */
    非临界区;
    P(S); /* 进行 P 操作 */
    临界区（使用资源 R）; /* 只有有限个（通常只有一个）进程被允许进入该区 */
```

```
    V(S); /* 进行 V 操作 */
    非临界区;
}
```

如下的十字路口就是临界资源的一个典型例子，东西向和南北向的车流不能同时经过十字路口；为避免争抢十字路口带来的冲突，需要在十字路口设置红绿灯，使得同一时刻只有一个方向的车流；



这里十字路口就是临界资源，红绿灯就是一个二值（红禁止、绿通行）信号量；
信号量名称：cross-road
信号量值：
0， 红灯，禁止通行
1， 绿灯，允许通行

信号量包括 posix 有名信号量、 posix 基于内存的信号量(无名信号量)和 System V 信号量(IPC 对象)；System V 的信号量是一个或者多个信号量的一个集合，其中的每一个都是单独的计数信号量；而 Posix 信号量指的是单个计数信号量；

二值信号量：值为 0 或 1，表示资源是否可用，用于实现同步；
计数信号量：可以取任意非负值，表示资源的数量；

- 函数 semget()

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semget(key_t key, int nsems, int semflg)
函数传入值	key：信号量的键值，多个进程可以通过它访问同一个信号量，其中有个特殊值 IPC_PRIVATE，用于创建当前进程的私有信号量
	nsems：需要创建的信号量数目，通常取值为 1
	semflg：同 open()函数的权限位，也可以用八进制表示法，其中使用 IPC_CREAT 标志创建新的信号量，即使该信号量已经存在（具有同一个键值的信号量已在系统中存在），也不会出错。
函数返回值	成功：信号量标识符，在信号量的其他函数中都会使用该值
	出错：-1

➢ 函数 semctl()

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semctl(int semid, int semnum, int cmd, union semun arg)
函数传入值	semid：semget()函数返回的信号量集标识符
	semnum：信号量编号，当使用信号量集时才会被用到。通常取值为 0，就是使用单个信号量（也是第一个信号量）
	cmd：指定对信号量的各种操作，当使用单个信号量（而不是信号量集）时，常用的操作有以下几种。 ● IPC_STAT：获得该信号量（或者信号量集）的 semid_ds 结构，并存放在由第 4 个参数 arg 结构变量的 buf 域指向的 semid_ds 结构中。semid_ds 是在系统中描述信号量的数据结构 ● IPC_SETVAL：将信号量值设置为 arg 的 val 值 ● IPC_GETVAL：返回信号量的当前值 ● IPC_RMID：从系统中删除信号量（或者信号量集）

	arg：是 union semnn 结构，可能在某些系统中不给出该结构的定义，此时必须由程序员自己定义
	union semun {
	int val; /* Value for SETVAL */
	struct semid_ds *buf; /* Buffer for IPC_STAT, IPC_SET */
	unsigned short *array; /* Array for GETALL, SETALL */
	struct seminfo *__buf; /* Buffer for IPC_INFO (Linux-specific) */
	};

函数返回值	成功：根据 cmd 值的不同而返回不同的值 IPC_STAT、IPC_SETVAL、IPC_RMID：返回 0 IPC_GETVAL：返回信号量的当前值
	出错：-1

➢ 函数 semop()

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semop(int semid, struct sembuf *sops, unsigned nsops)
函数传入值	semid：semget()函数返回的信号量标识符
	sops：指向信号量操作数组，一个数组包括以下成员。 struct sembuf { short sem_num; /* 信号量编号，0 对应第一个信号量 */ short sem_op; /* 信号量操作：取值为-1 则表示 P 操作，取值为+1 则表示 V 操作 */ short sem_flg; /* 0, IPC_NOWAIT,SEM_UNDO;通常设置为 SEM_UNDO。这样在进

	程没释放信号量而退出时，系统自动释放该进程中未释放的信号量 */ }
	nsops：操作数组 sops 中的操作个数（元素数目），通常取值为 1（一个操作）
函数返回值	成功：信号量标识符，在信号量的其他函数中都会使用该值
	出错：-1

四、 作业

采用共享内存和信号量实现读、写进程之间的通信；

write 进程

1） write 进程创建共享内存和信号量 2 个 IPC 对象；

2） write 进程向共享内存写入数据，然后执行 v 操作，以信号量方式通知给 read 进程；

read 进程

3） read 进程获得共享内存和信号量 ID；

4） read 进程执行 p 操作，阻塞自己，等待 write 进程的信号量；

5） read 进程收到 write 进程发送的信号量，解除阻塞，从共享内存读出 write 进程写入的数据；

6） 最后 write 进程向共享内存写入字符串 “quit”，然后解除对共享内存的映射；同时以信号量方式通知 read 进程退出；

7） read 进程看到 write 进程的 “quit” 指令后，解除对共享内存的映射，删除共享内存和信号量，退出；

测试要求：按以下步骤测试，截屏；

1） write 进程首先依次输入如下内容；在 read 进程窗口读出并显示 write 进程写入的这些内容；截屏；

a

b

c

d

2） 新开一个 terminal 窗口运行 ipcs –a 命令查看 IPC 对象的情况，截屏；

3） write 进程向共享内存写入字符串 “quit”；

4） 再次运行 ipcs –a 命令查看 IPC 对象的情况，截屏；比较前后 2 次 ipcs –a 命令查看 IPC 对象的不同；