实验六 进程间通信

**实验目的**

了解Linux系统的通信机制，管道、消息队列、信号量的创建与使用、共享内存。

**实验内容**

# 进程间通信概述：

进程的地址空间是各自独立的，因此进程之间交互数据必须采用专门的通信机制，特别是在大型的应用系统中，往往需要多个进程相互协作共同完成一个任务，这 就需要使用进程间通信（Internet Process Connection，IPC）编程技术。

Linux下进程间通信的方法基本上是从UNIX平台继承来的。Linux操作系统不但继承了system V IPC通信机制，还集成了基于套接字（socket）的进程间通信机制。前者是贝尔实验室对UNIX早期的进程间通信手段的改进和扩充，其通信的进程局限于单台计算机内；后者则突破了这一局限，通信的进程可以运行在不同的主机上，也就是进行网络通信。

管道（pipe）：管道是一种半双工的通信方式，数据只能单方向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

有名管道（named pipe）：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

信号量（semophore）：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止当某进程正在访问共享资源时，其它进程也访问该资源。因此主要作为进程间以及同一进程内不同线程之间的同步手段。消息队列（message queue）：消息队列是由消息的链表，存放着内核中并由消息队列标识符标识。消息队列克服了信号传递信息量少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

信号（signal）：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已发生。

共享内存（shared memory）：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建但是多个进程都可以访问。共享内存是最快的 IPC方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量配合使用，来实现进程间的同步和通信。

套接字（socket）：套接口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

# 管道

**管道：**管道是一种两个进程间进行进行通信的机制。管道这一特点决定了其使用的局限性。

数据只能由一个进程流向另一个进程；如果要进行全双工通信，需要建立两个管道。管道只能用于父子进程或者兄弟进程间的通信，也就是管道只能用于具有亲缘关系的进程间的通信，无亲缘关系的进程不能使用管道。

1．管道的创建：

Linux下创建管道可以通过函数pipe来完成。该函数如果调用成功返回0，并且数组中将包含两个新的文件描述符；如果有错误发生，返回-1。该函数原型如下： #include<unistd.h>

int pipe(int fd[2])

注意：管道只有通信双方同时存在时，才可以进行通信。例6-1，说明管道的创建、以及对管道的读写是如何进行的。

例6-1 pipe.c #include <stdio.h> #include <stdlib.h> #include <string.h>

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

/\*读管道\*/

void read\_from\_pipe (int fd)

{

char message[100]; read (fd,message,100);

printf("read from pipe:%s",message);

}

/\*写管道\*/

void write\_to\_pipe (int fd)

{

char \*message = "Hello, pipe!\n"; write (fd, message,strlen(message)+1);

}

int main(void)

{

int fd[2];

pid\_t pid;

int stat\_val;

if (pipe (fd))

{

printf ("create pipe failed!\n"); exit (1);

}

pid = fork(); switch (pid)

{

}

return 0;

}

case -1:

case 0:

default:

printf ("fork error!\n"); exit (1);

/\*子进程关闭fd1\*/ close (fd[1]); read\_from\_pipe (fd[0]); exit (0);

/\*父进程关闭fd0\*/ close (fd[0]); write\_to\_pipe (fd[1]); wait (&stat\_val);

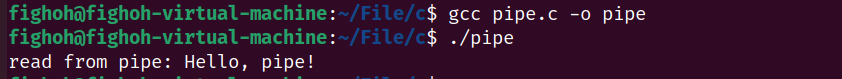
exit (0);

程序说明：在该程序中，父进程向管道中写数据，子进程从管道中获取数据。可以看到，对管道的读写和对一般文件的读写没什么区别。

程序的运行结果如下：

$./pipe

$Hello,pipe!



注意：必须在系统调用fork()之前调用pipe()，否则子进程将不会继承管道的文件描述符。

管道的一种常见法是：在父进程创建子进程后向子进程传递参数。例如，一个应用软件有一个主进程和很多个不同的子进程。主进程创建子进程后，在子进程调用exec函数执行一个新程序前，通过管道给即将执行的程序传递命令行参数，子进程根据传来的参数进行初始化或其他操作。下面通过例6-2和例6-3来演示这种

用法，这个程序包括一个主进程程序和一个子进程中要执行的新程序。例6-2 monitor.c

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <sys/types.h> #include <sys/stat.h>

int main(int arg, char \*argv[],char \*\* environ)

{

int fd[2]; pid\_t pid; int stat\_val;

if (arg < 2)

{

printf("wrong parameters \n"); exit(0);

}

if (pipe(fd))

{

perror("pipe failed"); exit(1);

}

pid = fork(); switch (pid)

{

case -1:

perror ("fork failed!\n"); exit(1);

case 0:

close (0); dup (fd[0]);

execve ("ctrlprocess",(void \*)argv,environ); exit (0);

default:

close(fd[0]);

/\*将命令行第一个参数写进管道\*/ write (fd[1], argv[1], strlen(argv[1])); break;

}

wait (&stat\_val); exit (0);

}

例6-3 ctrlprocess.c #include <stdio.h> #include <unistd.h>

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

{

int n;

char buf[1024];

while (1)

{

if ((n = read (0, buf, 1024)) > 0) // 标准输入stdin的描述符为0

{

buf[n] = '\0';

printf ("ctrlprocess receive: %s\n",buf);

if (!strcmp (buf,"exit")) exit(0);

if (!strcmp (buf, "getpid"))

{

printf ("My pid:%d\n", getpid()); sleep (3);

exit (0);

}

}

}

}

程序说明：主进程向管道中写一个命令行参数，子进程从标准输入里面读出该参数，进行相应的操作，首先编译ctlprocess.c：

$gcc –o ctrlprocess ctrlprocess.c

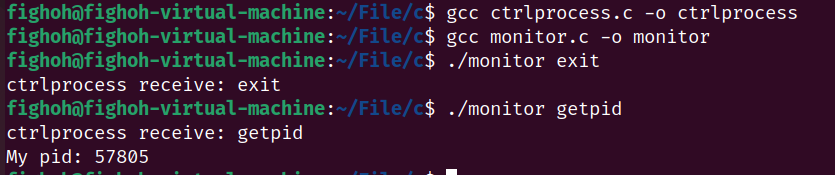
在编译主程序monitor.c:

$gcc –o monitor monitor.c

分别运行./monitor exit 和./monitor getpid,结果分别如下：

$./monitor exit ctrlprocess receive exit

$./monitor getpid ctrlprocess receive getpid My pid:17031



可见，被监控子进程接受监控主进程命令，执行不同的操作。这在实际项目中是经常见到的，监控进程启动加载多个具有不同功能的子进程，并通过管道的形式向子进程传递参数。

# 消息队列

消息队列是一个存放在内核中的消息链表，每个消息队列由消息队列标识符标识。与管道不同的是消息队列存放在内核中，只有在内核重启或者显式地删除一个消 息队列时，该消息队列才会被真正的删除。

操作消息队列时，需要用到一些数据结构，熟悉这些数据结构是掌握消息队列的关键。

1. **消息的创建与读写**创建消息队列

消息队列是随着内核的存在的，每个消息队列在系统范围内对应唯一的键值。要获得一个消息队列的描述符，只需提供该消息队列的键值即可，该键值通常由函数ftok返回。该函数定义在头文件sys/ipc.h中：

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

key\_t ftok(const char \*pathname,int proj\_id);

ftok函数根据pathname和proj\_id这两个参数生成惟一的键值。该函数执行成功会返回一个键值，失败返回-1。例如6-4是一个获取键值的例子：

例6-4 ftok.c #include <stdio.h>

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

int main( void )

{

int i;

for ( i = 1; i <= 5; i++ )

printf( "key[%d] = %ul \n", i, ftok( ".", i) ); exit(0);

}

# 程序运行的结果是什么？请简单分析。

# 

注意：参数pathname在系统中一定要存在且进程有权访问，参数proj\_id的取值范围为1~255。ftok()返回的键值可以提供给函数msgget。msgget()根据这个键值创建一个新的消息队列或者访问一个已存在的消息队列。msgget定义在头文件 sys/msg.h中：

int msgget(key\_t key,int msgflg);

msgget的参数key即为ftok函数的返回值。msgflg是一个标志参数。以下是msgflg

的可能取值。

IPC\_CREATE：如果内核中不存在键值与key相等的消息队列，则新建一个消息队列；如果存在这样的消息队列，返回该消息队列的描述符。 IPC\_EXCL：和IPC\_CREATE一起使用，如果对应键值的消息队列已经存在，则出错，返回-1。

注意：IPC\_EXCL单独使用是没有任何意义的。

该函数如果调用成功返回一个消息队列的描述符，否则返回-1。

# 写消息队列

创建了一个消息队列后，就可以对消息队列进行读写了。函数msgsnd用于向消息队列发送（写）数据。该函数定义在头文件sys/msg.h中：

int msgsnd(int msqid,struct msgbuf \*msgp,size\_t msgsz,int magflg); msgsnd各参数含义如下：

msqid:函数向msgid标识的消息队列发送一个消息。

msgp:msgp指向发送的消息。

msgsz:要发送的消息的大小，不包含消息类型占用4个字节。

magflg:操作标志位。可以设置为0或者IPC\_NOWAIT。如果msgflg为0，则当消息队列msgsnd函数成功返回0，失败返回-1.常见错误有：EAGAIN，说明消息队列已满；EIDRM，说明消息队列已被删除；EACCESS，说明无权访问消息队列。例6-5演示了如何向消息队列发送消息。

例6-5 sendmsg.c #include <stdio.h> #include <stdlib.h> #include <sys/ipc.h> #include <sys/msg.h>

#define BUF\_SIZE 256

#define PROJ\_ID 32

#define PATH\_NAME "." int main(void)

{

/\*用户自定义消息缓冲\*/ struct mymsgbuf {

long msgtype;

char ctrlstring[BUF\_SIZE];

} msgbuffer;

int qid; /\*消息队列标识符\*/ int msglen;

key\_t msgkey;

/\*获取键值\*/

if((msgkey = ftok (PATH\_NAME, PROJ\_ID)) == -1)

{

perror ("ftok error!\n"); exit (1);

}

/\*创建消息队列\*/

if((qid = msgget (msgkey, IPC\_CREAT|0660)) == -1)

{

perror ("msgget error!\n"); exit (1);

}

/\*填充消息结构，发送到消息队列\*/ msgbuffer. msgtype = 3;

strcpy (msgbuffer.ctrlstring , "Hello,message queue"); msglen = sizeof(msgbuffer) - 4;

if(msgsnd (qid, &msgbuffer, msglen, 0) == -1)

{

perror ("msgget error!\n"); exit (1);

}

exit(0);

}

# 编译运行该程序，从输出的结果可以观察到哪些信息？？

# 

# 看出成功创建消息队列

1. **读消息队列**

消息队列中放入数据后，其他进程就可以读取其中的消息了。读取消息的系统调

用为msgrcv()，该函数定义在头文件sys/msg.h中，其原型如下：

int msgrcv(int msqid,struct msgbuf \*msgp,size\_t msgsz,long int msgtyp,int msgflg);

该函数有5个参数，含义如下：

msqid：消息队列描述符。

msgp：读取的消息存储到msgp指向的消息结构中。 msgsz：消息缓冲区的大小。 msgtyp：为请求读取的消息类型。

msgflg：操作标志位。Msgflg可以为IPC\_NOWAIT,IPC\_EXCEPT,IPC\_NOERROR此时错误码为ENOMSG；IPC\_EXCEPT，与msgtyp配合使用，返回队列中第一个类型不为msgtyp的消息；IPC\_NOERROR，如果队列中满足条件的消息内容大于所请求的msgsz字节，则把该消息截断，截断部分将被丢弃。

调用msgrcv函数的时候，成功会返回读出消息的实际字节数，否则返回-1，常见错误码有：E2BIG，标识消息的长度大于msgsz；EIDRM，标识消息队列已被删除；EINVAL，说明msqid无效或msgsz小于0.

下面通过例6-5来说明如何从消息队列读消息，该例读取例6-4写入消息队列的数据。

例：6-5 rcvmsg.c #include <stdio.h> #include <stdlib.h> #include <sys/ipc.h> #include <sys/msg.h>

#define BUF\_SIZE 256

#define PROJ\_ID 32

#define PATH\_NAME "."

int main(void)

{

/\*用户自定义消息缓冲区\*/ struct mymsgbuf{

long msgtype;

char ctrlstring[BUF\_SIZE];

} msgbuffer;

|  |  |  |
| --- | --- | --- |
| int | qid; | /\*消息队列标识符\*/ |
| int | msglen; |  |
| key\_t | msgkey; |  |

/\*获取键值\*/

if((msgkey = ftok(PATH\_NAME, PROJ\_ID)) == -1)

{

perror("ftok error!\n"); exit(1);

}

/\*获取消息队列标识符\*/

if((qid = msgget(msgkey, IPC\_CREAT|0660)) == -1)

{

perror ("msgget error!\n"); exit (1);

}

msglen = sizeof(struct mymsgbuf) - 4;

if (msgrcv(qid, &msgbuffer, msglen, 3, 0) == -1) /\*读取数据\*/

{

perror ("msgrcv error!\n"); exit (1);

}

printf("Get message %s\n", msgbuffer.ctrlstring);

exit(0);

}

# 编译运行，并分析程序结果。

# 

* 1. **信号量**

信号量是一个计数器，常用于处理进程或线程的同步问题，特别是对临界资源的访问的同步。临界资源可以简单地理解为在某一时刻只能由一个进程或线程进行操作的资源，这里的资源可以是一段代码、一个变量或某种硬件资源。信号量的值大于或等于0时表示可供并发进程使用的资源实体数；小于0是代表正在等待使用临界资源的进程数。

# 1.信号量的创建与使用

1. 信号量集的创建或打开

Linux 下使用系统函数semget 创建或打开信号集。这个函数定义在头文件

sys/sem.h中，函数原型如下：

int semget(key\_t key, int nsems, int semflg);

该函数执行成功则返回一个信号集的标识符，失败返回-1。函数的第一个参数是有ftok得到的键值；第二个参数nsems指明要创建的信号集包含的信号个数，如果只是打开信号集，把nsems设置为0即可；第三个参数semflg为操作标志，可以

取如下值.

IPC\_CREATE:调用semget()时，它会将此值与系统中其他信号集的key进行对比，如过存在相同的key，说明信号集已存在，此时返回该信号集的标识符，否则新建一个信号集并返回其标识符。

IPC\_EXCL：该宏须和IPC\_CREATE一起使用，否则而没有意义。当semflg取 ipc\_create|IPC\_EXCL时，表示如果发现信号集已经存在，则返回错误，错误码为 EEXIST。

1. 信号量的操作

信号量的值与相应资源的使用情况有关，当它的值大于0时，表示当前可用资源的数量，当它的值小于0时，其绝对值表示等待使用资源的进程个数。信号量的值只能由PV操作来改变。在Linux下，PV操作通过调用函数semop实现。该函数定义在头文件sys/sem.h，原型如下：

int semop(int semid,struct sembuf \*sops,size\_t nsops);

函数的参数semid为信号集的标识符；参数sops指向进行操作的结构体数组首地址：参数nsops指出将要进行操作的信号的个数。semop函数调用成功返回0，否则返回-1。

semop的第二个参数sops指向的结构体数组中，每个sembuf结构体对应一个特定信号的操作。因此对信号进行操作必须熟悉该数据结构，该结构体定义在 linux/sem.h，如下所示：

struct sembuf{

ushort sem\_num; /\*信号在信号集中的索引\*/ short sem\_op; /\*操作类型\*/

short sem\_flg; /\*操作标志\*/

}

sem\_op的取值及意义

|  |  |
| --- | --- |
| **取值范围** | **操作意义** |
| sem\_op>0 | 信号加上sem\_op的值，表示进程释放控制的资源 |
| sem\_op=0 | 如果没有设置IPC\_NOWAIT则调用进程进入睡眠状态，直到信号值为0；否则进程不会睡眠，直接返回EAGAIN |
| sem\_op<0 | 信号加上sem\_op的值，若没有设置IPC\_NOWAIT，则调用进程  阻塞，直到资源可用；否则进程直接返回EAGAIN |

信号量的应用实例：

信号量一般用于处理访问临界资源的同步问题。下面通过例6-6和例6-7中的 server.c和client.c程序来演示信号量如何控制对资源的访问。Server创建一个信号集，并对信号集循环减1，相当于分配资源。client执行时检查信号量，如果其值大于0代表有资源可用，继续执行，如果小于等于0代表资源已经分配完毕，进程 client退出。

例6-6 server.c

#include <sys/types.h> #include <linux/sem.h>

#define MAX\_RESOURCE 5 int main(void)

{

key\_t key; int semid;

struct sembuf sbuf = {0, -1, IPC\_NOWAIT}; union semunsemopts;

if ((key = ftok(".", 's')) == -1)

{

perror ("ftok error!\n"); exit (1);

}

if ((semid = semget (key,1,IPC\_CREAT|0666)) == -1)

{

perror ("semget error!\n"); exit (1);

}

semopts.val = MAX\_RESOURCE;

if (semctl (semid,0,SETVAL, semopts) == -1)

{

perror ("semctl error!\n"); exit (1);

}

while (1)

{

if(semop(semid, &sbuf, 1) == -1)

{

perror ("semop error!\n"); exit (1);

}

sleep (3);

}

exit (0);

}

例6-7 client.c

#include <sys/types.h> #include <linux/sem.h>

int main(void)

{

key\_t key;

int semid, semval; union semunsemopts;

if((key = ftok (".",'s')) == -1)

{

perror ("ftok error!\n"); exit (1);

}

if((semid = semget (key, 1, IPC\_CREAT | 0666)) == -1)

{

perror ("semget error!\n"); exit (1);

}

while(1)

{

if ((semval = semctl(semid, 0, GETVAL, 0)) == -1)

{

perror ("semctl error!\n"); exit (1);

}

if (semval > 0)

{

printf ("Still %d resources can be used\n", semval);

}

else

{

}

printf ("No more resources can be used!\n"); break;

sleep (3);

}

exit (0);

}

**client的运行结果。分析程序的执行过程。**

**首先在终端上编译并运行server.c再在另外一个终端上编译运行client.c，观察**

# 共享内存

# 

# 

共享内存就是分配一块能被其他进程访问的内存。每个共享内存段在内核中维护着一个内部结构shmid\_ds（和消息队列、信号量一样），该结构定义在头文件 linux/shm.h中。

# 共享内存的创建和操作

LINUX下使用函数shmget来创建一个共享内存区，或者访问一个已存在的共享内存区。该函数定义在头文件linux/shm.h中，原型如下：

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

函数中：参数key是由ftok()得到的键值；参数size以字节为单位指定内存的大小；

shmflg为操作标志位，它的值为一些宏，如下所示：

IPC\_CREATE:调用shmget时，系统将辞职与其他所有共享内存区的key进行比较，如果存在相同的key，说明共享内存区已存在，此时返回该共享内存区的标识符，否则新建一个共享内存区并返回其标识符。

IPC\_EXCL: 该宏必须和IPC\_CREATE一起使用， 否则没有意义。当shmflg取 IPC\_CREATE|IPC\_EXCL时，表示如果发现信号集已经存在，则返回-1，错误码为EEXIST。

注意：当创建一个新的共享内存区是，size值必须大于0；如果是访问一个已存在的共享内存区，置size为0.

# 共享内存区的操作

在使用共享内存区前，必须通过shmat函数将其附加到进程的地址空间。进程与共享内存就建立了连接。shmat调用成功后就会返回一个指向共享内存区的指针，使用该指针就可以访问共享内存区了，如果失败返回-1。该函数声明在linux/shm.h文件中，具体结构代码原型如下：

void \* shmat(int shmid,const void \*shmaddr,int shmflg);

参数shmid为shmget的返回值；参数shmflg为存取权限标志；参数shmaddr为共享

内存的附加点。参数shmaddr不同取值情况的含义说明如下：

如果为空，则由内核选择而一个空闲的内存区；如果非空，返回地址取决于调用 者是否给shmflg参数指定了SHM\_RND值，如果没有指定，则共享内存区附加到 有shmaddr向下舍入一个共享内存低端边界地址后的地址（SHMLBA，一个常址）。通常将参数shmaddr设置为NULL。

当进程结束使用共享内存区时，要通过函数shmdt断开与共享内存的链接。该函数声明在sys/shm.h文件中，具体结构代码原型如下：

int shmdt(const void\* shmaddr);

参数shmaddr为shmat函数的返回值。该函数调用成功后，返回0，否则返回-1。进程脱离共享内存区后，数据结构shmid\_ds中shm\_nattch就会减1。但是共享内存段依然存在，只有shm\_attch为0后，即没有任何进程再使用该共享内存区，共享内存区才能在内核中被删除。一般来说，当一个进程终止时，它所附加的共享内存区都会自动脱离。

# 共享内存区的控制

Linux对共享内存区的控制是通过调用函数shmctl来完成的，该函数定义在头文件 sys/shm.h中，原型代码如下所示：

int shmctl(int shmid,int cmd, struct shmid\_ds \*buf);

函数中：参数shmid为共享内存区的标识符；buf为指向shmid\_ds结构体的指针；

cmd为操作标志位，支持一下3种控制操作。 IPC\_RMID:从系统中删除有shmid标识的共享内存区。 IPC\_SET：设置共享内存区的shmid\_ds结构。

IPC\_STAT：读取共享内存区的shimid\_ds结构，并将其存储到buf指向的地址中。例6-8和例6-9是writer和reader程序，两程序在进入共享内存区之前，首先都检查信号集中的信号的值是否为1（相当于是否能进入共享内存区），如果不为1，调用sleep()进入睡眠状态直到信号的值变为1。进入共享内存区之后，将信号的值减1（相当于加锁），这样就实现了互斥的访问共享资源。在退出共享内存时，将信号值加1（相当于解锁）。

例6-8 writer.c

#include "sharemem.h" #define SHM\_SIZE 1024

int main()

{

int semid, shmid; char \*shmaddr;

char write\_str[SHM\_SIZE];

if ((shmid = createshm (".", 'm', SHM\_SIZE)) == -1)

{

exit(1);

}

if ((shmaddr = shmat (shmid, (char \*)0, 0)) ==(char \*)-1)

{

perror ("attach shared memory error!\n"); exit (1);

}

if ((semid = createsem (".", 's', 1, 1)) == -1)

{

exit (1);

}

while (1)

{

wait\_sem (semid, 0);

sem\_p (semid, 0); /\*P操作\*/

printf ("writer: ");

fgets (write\_str, 1024, stdin); int len = strlen (write\_str) - 1; write\_str[len] = '\0';

strcpy (shmaddr, write\_str);

sleep (10); /\*使reader处于阻塞状态\*/

sem\_v (semid, 0); /\*V操作\*/

sleep (10); /\*等待reader进行读操作\*/

}

}

例6-9 reader.c #include "sharemem.h"

int main()

{

int semid, shmid;

char\*shmaddr;

if ((shmid = createshm(".", 'm', SHM\_SIZE)) == -1)

{

exit (1);

}

if((shmaddr = shmat (shmid, (char \*)0, 0)) == (char \*)-1)

{

perror ("attach shared memory error!\n"); exit (1);

}

if((semid = opensem("." ,'s')) == -1)

{

exit (1);

}

while(1)

{

printf("reader: ");

wait\_sem(semid,0); /\* 等待信号值为1 \*/ sem\_p(semid,0); /\* P操作 \*/

printf("%s\n", shmaddr);

sleep(10); /\* 使writer处于阻塞状态 \*/

sem\_v(semid,0); /\* V操作 \*/

sleep(10); /\* 等待writer进行写操作 \*/

}

}

编译两个程序，同时在两个终端运行writer和reader，在writer端输入字符串“hello， reader”，等待一会，看到reader端输出后，在writer.c端的提示符后面输入字符串 “new information”。

# 运行的结果是什么？请分析程序的执行过程。

# 

# 附：sharememory.h

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h> #include <sys/shm.h> #include <errno.h>

#define SHM\_SIZE 1024 union semun{

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

/\*创建信号量函数\*/

int createsem (const char \* pathname, int proj\_id, int members, int init\_val)

{

key\_t msgkey; int index, sid; union semunsemopts;

if ((msgkey = ftok(pathname, proj\_id)) == -1)

{

perror ("ftok error!\n"); return -1;

}

if ((sid = semget (msgkey, members, IPC\_CREAT | 0666)) == -1)

{

perror ("semget call failed.\n"); return -1;

}

/\*初始化操作\*/ semopts.val = init\_val;

for (index = 0; index < members; index++)

{

semctl (sid, index, SETVAL, semopts);

}

return (sid);

}

/\*打开信号量函数\*/

int opensem(const char \* pathname, int proj\_id)

{

key\_t msgkey; int sid;

if ((msgkey = ftok(pathname, proj\_id)) == -1)

{

perror ("ftok error!\n"); return -1;

}

if ((sid = semget(msgkey, 0, IPC\_CREAT | 0666)) == -1)

{

perror("semget call failed.\n"); return -1;

}

return (sid);

}

/\*P操作函数\*/

int sem\_p(int semid, int index)

{

struct sembuf buf = {0,-1,IPC\_NOWAIT};

if (index < 0)

{

perror("index of array cannot equals a minus value!"); return -1;

}

buf.sem\_num = index;

if (semop (semid ,& buf,1) == -1)

{

perror ("a wrong operation to semaphore occurred!"); return -1;

}

return 0;

}

/\*v操作函数\*/

int sem\_v (int semid, int index)

{

struct sembuf buf = {0, +1, IPC\_NOWAIT};

if (index < 0)

{

perror("index of array cannot equals a minus value!"); return -1;

}

buf.sem\_num = index;

if (semop (semid,& buf,1) == -1)

{

perror ("a wrong operation to semaphore occurred!"); return -1;

}

return 0;

}

/\*删除信号集函数\*/

int sem\_delete (int semid)

{

return (semctl(semid, 0, IPC\_RMID));

}

/\*等待信号为1\*/

int wait\_sem( int semid, int index)

{

while (semctl (semid, index, GETVAL, 0) == 0)

{

sleep (1);

}

return 1 ;

}

/\*创建共享内存函数\*/

int createshm( char \* pathname, int proj\_id, size\_t size)

{

key\_t shmkey; int sid;

/\*获取键值\*/

if ((shmkey = ftok(pathname, proj\_id)) == -1)

{

perror("ftok error!\n"); return -1;

}

if ((sid = shmget(shmkey, size, IPC\_CREAT | 0666)) == -1)

{

perror ("shmget call failed.\n"); return -1;

}

return (sid);

}