实验4 IPC通信机制的应用编程

# 实验目的

IPC通信机制是linux编程的核心内容，IPC通信机制的应用编程练习是理解IPC通信机制的重要手段，也是熟练运行IPC通信机制解决进程间通信问题的必要方式。本实验要求用C语言编写和调试应用IPC通信机制的实现程序。以达到理解和运用IPC通信机制的目的。

# 实验内容

IPC共有三种机制：消息队列、信号量以及共享内存，他们都能实现任意两个进程间对于特定资源的公共访问。学习这三种机制并进行一定的编程学习，了解其运行的方法。

# 实验步骤

* 分别学习消息队列、信号量、共享内存三种使用方法以及使用场合，进行编程确认。

# 详细设计

### 消息队列

消息队列：创建一个消息队列，进程A往队列里面写，那么进程B通过读队列中的容来获取进程A传送的信息。

通过如下系统提供的函数可以实现消息队列的创建、写入、读取、释放等等操作。

int msgget(key\_t key,int msgflg);  
int msgctl(int magid,int cmd,struct msgid\_ds \*buf);  
int msgsnd(int msgid,void \*msg\_ptr,size\_t msg\_sz,int msgflag);  
int msgrcv(int msgid,void \*msg\_ptr,size\_t msg\_sz,long int msg\_type,int msgflag);

* 设计一个发送端程序，通过 msgget 来创建一个消息队列，创建一个带有标志的缓冲区作为共享的空间，当写入前，设置标志为1，然后写入内容，通过 msgsnd 来向消息队列写入数据
* 设计一个接收端程序，通过 msgget 来获取一个消息队列，这里的队列的编号要相同，通过 msgrcv 来接受消息队列中的数据即可。

### 信号量

信号量主要是通过操作系统中的PV操作来实现，在操作系统中可以了解PV操作的详细含义，主要是实现对某一共享资源的竞争的确定，通过加减锁的方式来实现对共享资源的限定操作。在Linux中主要通过如下函数实现:

int semget(key\_t key,int num\_sems,int sem\_flgs);  
int semctl(int sem\_id,int sem\_num,int command...);  
int semop(int sem\_id,struct sembuf \*sem\_ops,size\_t num\_sem\_ops);

通过使用上述函数来实现对一个共享资源的pv操作，即加减锁操作，观察其运行中临界资源的情况。大致流程为创建信号量、初始化信号量、p操作、v操作、释放信号量资源等等。

### 共享内存

共享内存是最简单好用的进程间通信的方式，是对于共享空间的竞争等操作需要自己处理，它允许两个不相关的进程访问同一个逻辑地址，这样为程序的编写带来方便。相关的函数有：

1、int shmget(key\_t key,size\_t size,int shmflag);  
2、void \*shmat(int shm\_id,const void \*shm\_addr,int shm\_flag);  
3、int shmctl(int shm\_id,int cmd,struct shmid\_ds \*buf);  
4、int shmdt(const void \*shm\_addr);  
常用值：于共享内存的创建有关  
1、shmflag:0666|IPC\_CREAT  
2、shmaddr=0 shm\_flag=0  
3、cmd=IPC\_SET cmd=IPC\_RMID

如同消息队列一样，首先要创建一个带有标志的区域用来对数据的存放和标记，通过 shmget 来为指定键值处建立一个共享内存空间，通过 shmat 来获取这个空间的指针，最后就是同消息队列一样根据约定的标记来进行读写操作就行了。

## 调试分析

## 测试结果

### 消息队列

发送端：

x31415@iZ2zea1nnstgr3h5ccfovoZ:~/linux/ipc/msg$ ./msgsend   
msgsend input: alk;djfa;jlkdsjfkldsajf  
msgsend input: klsdfjkl;ads  
msgsend input: asjdklfjklsda;f  
msgsend input: asdfjlkasjfkladsjflkdjasflkjdaslkfja;sdf  
msgsend input: djaklsfjldaks;fjlkdasjf  
msgsend input: ajdklsfjjkldsaf  
msgsend input: jadlksflaksf  
msgsend input: jkadlsjfda;sldfjadslkfjlkadsjflkasdjf  
msgsend input: endlkdasjf

接收端：

x31415@iZ2zea1nnstgr3h5ccfovoZ:~/linux/ipc/msg$ ./msgrecive  
msgrcv: alk;djfa;jlkdsjfkldsajf  
  
msgrcv: klsdfjkl;ads  
  
msgrcv: asjdklfjklsda;f  
  
msgrcv: asdfjlkasjfkladsjflkdjasflkjdaslkfja;sdf  
  
msgrcv: djaklsfjldaks;fjlkdasjf  
  
msgrcv: ajdklsfjjkldsaf  
  
msgrcv: jadlksflaksf  
  
msgrcv: jkadlsjfda;sldfjadslkfjlkadsjflkasdjf  
  
msgrcv: endlkdasjf

### 信号量

编译运行即可 ./main a 看出程序对于临界资源的操作过程。其中 a 表示的就是进入临界区， b 表示释放离开临界区。

x31415@iZ2zea1nnstgr3h5ccfovoZ:~/linux/ipc/Semaphore$ ./main a  
abababababababababab  
12486 - finished...

### 共享内存

两个终端下编译运行即可，同消息队列方式一致。

发送端：

x31415@iZ2zea1nnstgr3h5ccfovoZ:~/linux/ipc/sharecache$ ./shmsend  
memery attached at 0x7fd44f1f1000  
input text: alkdsfjlkdsafjldas  
waiting.....  
waiting.....  
input text: sdajlkfjdlksajfkladsjfladskfj  
waiting.....  
waiting.....  
input text: kjaldsfjklasdjflkajdfkljalkdjf  
jklasdfjklajdfwaiting.....  
input text: lkjasdlkfjkladsjflkjasdlkfjdas  
waiting.....  
ewaiting.....  
input text:end

接收端：

x31415@iZ2zea1nnstgr3h5ccfovoZ:~/linux/ipc/sharecache$ ./shmrecive   
memery attached at 0x7f7662287000  
waiting....  
waiting....  
waiting....  
waiting....  
waiting....  
get shared cache: alkdsfjlkdsafjldas  
  
waiting....  
waiting....  
waiting....  
waiting....  
waiting....  
get shared cache: sdajlkfjdlksajfkladsjfladskfj  
  
waiting....  
waiting....  
waiting....  
waiting....  
waiting....  
waiting....  
get shared cache: kjaldsfjklasdjflkajdfkljalkdjf  
  
waiting....  
waiting....  
waiting....  
get shared cache: jklasdfjklajdflkjasdlkfjkladsjflkjasdlkfjdas  
  
waiting....  
waiting....  
waiting....  
waiting....  
waiting....  
get shared cache: end

可以看出类似消息队列的模式，当发送方没有写入数据时，也就是发送方获取对共享空间的操作权限时，接收方只能等待，同样接收方接受时，发送方不能写入。

# 实验总结

IPC三种通信机制是非情缘关系的进程间通信的主要方式，消息队列和共享内存的操作方式虽不同，但是其效果类似，信号量是更接近操作系统的对临界资源的操作，为资源加锁等操作提供了底层实现。 这里只是大致的了解了相关函数的使用方法已经使用场景，没有进行更深入的研究，所以这是这一部分学习的不足。

# 代码

## 消息队列

发送端：

#include<stdio.h>  
#include<stdlib.h>  
#include<string.h>  
#include<unistd.h>  
#include<sys/msg.h>  
#include<errno.h>  
  
struct msg\_st{  
 long int msg\_type;  
 char text[BUFSIZ];  
}msg;  
void error(char \*e){  
 fprintf(stderr, "%s: %d\n", e, errno);  
 exit(EXIT\_FAILURE);  
}  
int main(){  
 int flag = 1;  
 char buf[BUFSIZ];  
 int msgid = msgget((key\_t)2333, 0666|IPC\_CREAT);  
 if(!~msgid)error("msgget error");  
 while(flag){  
 // 输入内容  
 printf("msgsend input: ");  
 fgets(buf, BUFSIZ, stdin);  
  
 // 向消息队列发送  
 msg.msg\_type = 1;  
 strcpy(msg.text, buf);  
  
 if(msgsnd(msgid, (void\*)&msg, BUFSIZ, 0) == -1)error("msgsnd error");   
  
 if(!strncmp(buf, "end", 3))flag = 0;  
 }  
 exit(EXIT\_SUCCESS);  
 return 0;  
}

接收端：

#include<stdio.h>  
#include<stdlib.h>  
#include<string.h>  
#include<unistd.h>  
#include<sys/msg.h>  
#include<errno.h>  
  
struct msg\_st{  
 long int msg\_type;  
 char text[BUFSIZ];  
}msg;  
void error(char \*e){  
 fprintf(stderr, "%s: %d\n", e, errno);  
 exit(EXIT\_FAILURE);  
}  
int main(){  
 int flag = 1;  
 long int msg\_type = 0;  
 int msgid = msgget((key\_t)2333, 0666|IPC\_CREAT);  
 if(!~msgid)error("msgget error");  
 while(flag){  
 if(!~msgrcv(msgid, (void\*)&msg, BUFSIZ, msg\_type, 0))error("msgrcv error");  
 printf("msgrcv: %s\n", msg.text);  
 if(!strncmp(msg.text, "end", 3))flag = 0;  
 }  
 // 删除消息队列  
 if(!~msgctl(msgid, IPC\_RMID, 0))error("msgctl error");  
 exit(EXIT\_SUCCESS);  
 return 0;  
}

## 信号量

#include<stdio.h>  
#include<stdlib.h>  
#include<string.h>  
#include<unistd.h>  
#include<sys/types.h>  
#include<sys/stat.h>  
#include<fcntl.h>  
#include<sys/sem.h>  
#include<sys/ipc.h>  
#include<errno.h>  
  
union sem{  
 int v;  
 struct semid\_ds \*buf;  
 unsigned short \*a;  
};  
void error(char \*e){  
 fprintf(stderr, "%s: %d\n", e, errno);  
 // exit(EXIT\_FAILURE);  
}  
  
static int semid = 0;  
// 初始化信号量  
static int setsemval(){  
 union sem s;  
 s.v = 1;  
 if(!~semctl(semid, 0, SETVAL, s))return 0;  
 return 1;  
}  
// 删除信号量  
static void delsemval(){  
 union sem s;  
 if(!~semctl(semid, 0, IPC\_RMID, s))error("fail to delete semval\n");  
}  
// 对信号量做减一操作，即等待p(sv)  
static int sem\_p(){  
 struct sembuf semb;  
 semb.sem\_num = 0;  
 semb.sem\_op = -1;  
 semb.sem\_flg = SEM\_UNDO;  
 if(!~semop(semid, &semb, 1)){  
 error("sem\_p failed\n");  
 return 0;  
 }  
 return 1;  
}  
// 对信号量做增一操作，即v  
static int sem\_v(){  
 struct sembuf semb;  
 semb.sem\_num = 0;  
 semb.sem\_op = 1;  
 semb.sem\_flg = SEM\_UNDO;  
 if(!~semop(semid, &semb, 1)){  
 error("sem\_v failed\n");  
 return 0;  
 }  
 return 1;  
}  
  
int main(int argc, char \*\*argv){  
 char msg = 'X';  
   
 // 创建信号量  
 semid = semget((key\_t)2333, 1, 0666|IPC\_CREAT);  
  
 if(argc > 0){  
 // 第一次执行，初始化信号量  
 if(!setsemval()){  
 error("fail to init sem\n");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 设置要在屏幕中输出的信息，  
 msg = argv[1][0];  
 sleep(2);  
 }  
  
 for(int i = 0; i < 10; ++i){  
 // 进入临界区  
 if(!sem\_p())exit(EXIT\_FAILURE);  
  
 // 输出  
 printf("%c", msg);  
 // 清楚缓冲区，立即显示  
 fflush(stdout);  
  
 sleep(rand() % 3);  
 // 离开临界区再次输出  
 printf("%c", msg + 1);  
 fflush(stdout);  
 if(!sem\_v()){  
 exit(EXIT\_FAILURE);  
 }  
 sleep(rand() % 2);  
 }  
 sleep(10);  
 printf("\n%d - finished...\n", getpid());  
 if(argc > 1){  
 // 第一次调用，在推出前删除信号量  
 sleep(3);  
 delsemval();  
 }  
 exit(EXIT\_SUCCESS);  
 return 0;  
}

## 共享内存