实验四、进程同步实验

4.1 实验目的

加深对并发协作进程同步与互斥概念的理解，观察和体验并发进程同步与互斥操作的效果，分析与研究经典进程同步与互斥问题的实际解决方案。了解 Linux 系统中 IPC 进程同步工具的用法，练习并发协作进程的同步与互斥操作的编程与调试技术。

4.2 实验说明

在 linux 系统中可以利用进程间通信（interprocess communication ）IPC 中的 3 个对象：共享内存、信号灯数组、消息队列，来解决协作并发进程间的同步与互斥的问题。

1）共享内存是 OS 内核为并发进程间交换数据而提供的一块内存区（段）。如果段的权限设置恰当，每个要访问该段内存的进程都可以把它映射到自己私有的地址空间中。如果一进程更新了段中数据，那么其他进程立即会看到这一更新。进程创建的段也可由另一进程读写。

linux 中可用命令 **ipcs -m** 观察共享内存情况。

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 327682     student    600        393216     2          dest
0x00000000 360451     student    600        196608     2          dest
0x00000000 393220     student    600        196608     2          dest
```

key 共享内存关键值
shmid 共享内存标识
owner 共享内存所由者（本例为 student）
perm 共享内存使用权限(本例为 student 可读可写）
byte 共享内存字节数
nattch 共享内存使用计数
status 共享内存状态
上例说明系统当前已由 student 建立了一些共享内存,每个都有两个进程在共享。

2)信号灯数组是 **OS** 内核控制并发进程间共享资源的一种进程同步与互斥机制。

linux 中可用命令 **ipcs -s** 观察信号灯数组的情况。

```
$ ipcs -s
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0000000 163844     apache     600        1
0x4d00f259 294920     beagleind 600        8
0x00000159 425995     student    644        1
```

semid 信号灯的标识号

nsems 信号灯的个数
其他字段意义同以上共享内存所述。
上例说明当前系统中已经建立多个信号灯。其中最后一个标号为 425996 是由 student 建立的,它的使用权限为 644，信号灯数组中信号灯个数为 1 个。

3)消息队列是 **OS** 内核控制并发进程间共享资源的另一种进程同步机制。**linux** 中可用命令 **ipcs -q** 观察消息队列的情况。

```
$ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x000001c8 0          root       644        8            1
```

msgmid 消息队列的标识号
used-bytes 消息的字节长度
messages 消息队列中的消息条数
其他字段意义与以上两种机制所述相同。
上例说明当前系统中有一条建立消息队列，标号为 0，为 root 所建立，使用权限为 644，每条消息 8 个字节，现有一条消息。

4) 在权限允许的情况下您可以使用 **ipcrm** 命令删除系统当前存在的 **IPC** 对象中的任一个对象。

```
ipcrm -m 21482        删除标号为 21482 的共享内存。
ipcrm -s 32673        删除标号为 32673 的信号灯数组。
ipcrm -q 18465        删除标号为 18465 的消息队列。
```

5) 在 **linux** 的 **proc** 文件系统中有 **3** 个虚拟文件动态记录了由以上 **ipcs** 命令显示的当前 **IPC** 对象的信息，它们分别是：

```
/proc/sysvipc/shm  共享内存
/proc/sysvipc/sem  信号量
/proc/sysvipc/msg  消息队列
```

我们可以利用它们在程序执行时获取有关 IPC 对象的当前信息。

6) IPC 对象有关的系统调用函数原型都声明在以下的头文件中 **：**

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

创建一段共享内存系统调用语法 **：**

```
#include <sys/shm.h>
int shmget(key_t key,int size,int flags);
```

key 共享内存的键值,可以为 IPC_PRIVATE,也可以用整数指定一个
size 共享内存字节长度
flags 共享内存权限位。
shmget 调用成功后，如果 key 用新整数指定，且 flags 中设置了 IPC_CREAT 位，则返回一个新建立的共享内存段标识符。 如果指定的 key 已存在则返回与 key 关联的标识符。 不成功返回-1

令一段共享内存附加到调用进程中的系统调用语法：

```
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr,int flags)
        shmid    由 shmget 创建的共享内存的标识符
        shmaddr  总为 0,表示用调用者指定的指针指向共享段
        flags  共享内存权限位
```

shmat 调用成功后返回附加的共享内存首地址

令一段共享内存从到调用进程中分离出去的系统调用语法：

```
#include <sys/shm.h>
int shmdt(char *shmadr);
   shmadr    进程中指向附加共享内存的指针
   shmdt  调用成功将递减附加计数，当计数为 0，将删除共享内存。调用不成功返回-1。
```

创建一个信号灯数组的系统调用有语法：

```
#include <sys/sem.h>
int semget(key_t key,int nsems, int flags);
key  信号灯数组的键值,可以为 IPC_PRIVATE,也可以用整数指定一个
nsems  信号灯数组中信号灯的个数
flags  信号灯数组权限位。如果 key 用整数指定，应设置 IPC_CREAT 位。
```

semget 调用成功，如果 key 用新整数指定，且 flags 中设置了 IPC_CREAT 位，则返回一个新建立的信号等数组标识符。 如果指定的整数 key 已存在则返回与 key 关联的标识符。 不成功返回-1

操作信号灯数组的系统调用语法:

```
#include <sys/sem.h>
int semop(int semid,struct sembuf *semop, unsigned nops);
    semid    由 semget 创建的信号灯数组的标识符
    semop    指向 sembuf 数据结构的指针
    nops     信号灯数组元素的个数。
```

semop 调用成功返回 0，不成功返回-1。

控制信号灯数组的系统调用语法:

```
#include <sys/sem.h>
int semctl(int semid,int semnum,int cmd, union semun arg);
    semid    由 semget 创建的信号灯数组的标识符
    semnum  该信号灯数组中的第几个信号灯
    cmd       对信号灯发出的控制命令。例如:
              GETVAL  返回当前信号灯状态
              SETVAL  设置信号灯状态
              IPC_RMD    删除标号为 semid 的信号灯
    arg       保存信号灯状态的联合体,信号灯的值是其中一个基本成员
              union semun {
               int val;          /* value for SETVAL */
```

```
        ......
    };
```

semctl 执行不成功返回-1，否则返回指定的 cmd 的值。

创建消息队列的系统调用语法:

#include<sys/msg.h>
int msgget(key_t key,int flags)
key 消息队列的键值,可以为 IPC_PRIVATE,也可以用整数指定一个
flags 消息队列权限位。
msgget 调用成功，如果 key 用新整数指定，且 flags 中设置了 IPC_CREAT 位，则返回一个新建立的消息队列标识符。 如果指定的整数 key 已存在则返回与 key 关联的标识符。成功返回-1。

追加一条新消息到消息队列的系统调用语法：

```
#include <sys.msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

msqid 由消息队列的标识符
msgp 消息缓冲区指针。消息缓冲区结构为：

```
struct msgbuf {
    long mtype;       /* 消息类型，必须大于 0   */
    char mtext[1];    /* 消息数据，长度应于 msgsz 声明的一致*/
}
```

msgsz 消息数据的长度
msgflg 为 0 表示阻塞方式，设置 IPC_NOWAIT 表示非阻塞方式
msgsnd 调用成功返回 0，不成功返回-1。

从到消息队列中读出一条新消息的系统调用语法：

```
#include <sys.msg.h>
int msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz,   long msgtype, int msgflg);
```

msqid 由消息队列的标识符
msgp 消息缓冲区指针。消息缓冲区结构为：

```
struct msgbuf {
    long mtype;       /* 消息类型，必须大于 0   */
    char mtext[1];    /* 消息数据，长度应于 msgsz 声明的一致*/
}
```

msgsz 消息数据的长度
msgtype 决定从队列中返回哪条消息：
=0 返回消息队列中第一条消息
>0 返回消息队列中等于 mtype 类型的第一条消息。
<0 返回 mtype<=type 绝对值最小值的第一条消息。
msgflg 为 0 表示阻塞方式，设置 IPC_NOWAIT 表示非阻塞方式
msgrcv 调用成功返回 0，不成功返回-1。

删除消息队列的系统调用语法：

```
#include <sys/msg.h>
```

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
msqid 由消息队列的标识符
cmd 控制命令。常用的有：
IPC_RMID 删除 msgid 标识的消息队列
IPC_STAT 为非破坏性读，从队列中读出一个 msgid_ds
结构填充缓冲 buf
IPC_SET 改变队列的 UID，GID，访问模式和最大字节数。
msgctl 调用成功返回 0，不成功返回-1。

4.3 示例实验

以下示例实验程序应能模拟多个生产/消费者在有界缓冲上正确的操作。它利用 N 个字节的共享内存作为有界循环缓冲区，利用写一字符模拟放一个产品，利用读一字符模拟消费一个产品。当缓冲区空时消费者应阻塞睡眠，而当缓冲区满时生产者应当阻塞睡眠。一旦缓冲区中有空单元，生产者进程就向空单元中入写字符，并报告写的内容和位置。一旦缓冲区中有未读过的字符，消费者进程就从该单元中读出字符，并报告读取位置。生产者不能向同一单元中连续写两次以上相同的字符，消费者也不能从同一单元中连续读两次以上相同的字符。

1) 在当前新建文件夹中建立以下名为 **ipc..h** 的 **C** 程序的头文件，该文件中定义了生产者/消费者共用的 **IPC** 函数的原型和变量：

```
/*
   *    Filename              :    ipc.h
   *    copyright             : (C) 2006 by zhonghonglie
    * Function               : 声明 IPC 机制的函数原型和全局变量
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/msg.h>

#define BUFSZ     256

//建立或获取 ipc 的一组函数的原型说明

int get_ipc_id(char *proc_file,key_t key);

char *set_shm(key_t shm_key,int shm_num,int shm_flag);

int set_msq(key_t msq_key,int msq_flag);

int set_sem(key_t sem_key,int sem_val,int sem_flag);
int down(int sem_id);
```

```
int up(int sem_id);

/*信号灯控制用的共同体*/
typedef union semuns {
        int val;
} Sem_uns;

/*  消息结构体*/
typedef struct msgbuf {
        long mtype;
        char mtext[1];
} Msg_buf;

//生产消费者共享缓冲区即其有关的变量
  key_t buff_key;
  int buff_num;
  char *buff_ptr;

//生产者放产品位置的共享指针
  key_t pput_key;
  int pput_num;
  int *pput_ptr;

//消费者取产品位置的共享指针
  key_t cget_key;
  int cget_num;
  int *cget_ptr;

//生产者有关的信号量
  key_t prod_key;
  key_t pmtx_key;
  int prod_sem;
  int pmtx_sem;

//消费者有关的信号量
  key_t cons_key;
  key_t cmtx_key;
  int cons_sem;
  int cmtx_sem;

  int sem_val;
  int sem_flg;
  int shm_flg;
```

2) 在当前新建文件夹中建立以下名为 **ipc.c** 的 **C** 程序，该程序中定义了生产者/消费者共用的 **IPC** 函数：

```
   /*
```

```
   *    Filename              :    ipc.c
   *    copyright             : (C) 2006 by zhonghonglie
    * Function              :  一组建立 IPC 机制的函数
*/
#include "ipc.h"
/*
    * get_ipc_id()  从/proc/sysvipc/文件系统中获取 IPC 的 id 号
    * pfile:  对应/proc/sysvipc/目录中的 IPC 文件分别为
    *              msg-消息队列,sem-信号量,shm-共享内存
    * key:     对应要获取的 IPC 的 id 号的键值
*/
int get_ipc_id(char *proc_file,key_t key)
{
    FILE *pf;
    int i,j;
    char line[BUFSZ],colum[BUFSZ];

    if((pf = fopen(proc_file,"r")) == NULL){
  perror("Proc file not open");
        exit(EXIT_FAILURE);
     }
    fgets(line, BUFSZ,pf);
    while(!feof(pf)){
       i = j = 0;
       fgets(line, BUFSZ,pf);
       while(line[i] == ' ') i++;
       while(line[i] !=' ') colum[j++] = line[i++];
       colum[j] = '\0';
       if(atoi(colum) != key) continue;
       j=0;
       while(line[i] == ' ') i++;
       while(line[i] !=' ') colum[j++] = line[i++];
       colum[j] = '\0';
       i = atoi(colum);
       fclose(pf);
       return i;
     }
     fclose(pf);
     return -1;
}
/*
 *  信号灯上的 down/up 操作
 * semid:信号灯数组标识符
 * semnum:信号灯数组下标
 * buf:操作信号灯的结构
*/
int down(int sem_id)
```

```
{
   struct sembuf buf;
   buf.sem_op = -1;
   buf.sem_num = 0;
   buf.sem_flg = SEM_UNDO;
   if((semop(sem_id,&buf,1)) <0) {
      perror("down error ");
      exit(EXIT_FAILURE);
   }
   return EXIT_SUCCESS;
}

int up(int sem_id)
{
   struct sembuf buf;
   buf.sem_op = 1;
   buf.sem_num = 0;
   buf.sem_flg = SEM_UNDO;
   if((semop(sem_id,&buf,1)) <0) {
      perror("up error ");
      exit(EXIT_FAILURE);
   }
   return EXIT_SUCCESS;
}

/*
*     set_sem 函数建立一个具有 n 个信号灯的信号量
*           如果建立成功，返回 一个信号灯数组的标识符 sem_id
*           输入参数：
*     sem_key 信号灯数组的键值
*     sem_val 信号灯数组中信号灯的个数
*     sem_flag 信号等数组的存取权限
*/
int set_sem(key_t sem_key,int sem_val,int sem_flg)
{
 int sem_id;
 Sem_uns sem_arg;

      //测试由 sem_key 标识的信号灯数组是否已经建立
   if((sem_id = get_ipc_id("/proc/sysvipc/sem",sem_key)) < 0 )
      {
         //semget 新建一个信号灯,其标号返回到 sem_id
         if((sem_id = semget(sem_key,1,sem_flg)) < 0)
                  {
            perror("semaphore create error");
            exit(EXIT_FAILURE);
                  }
```

```
            //设置信号灯的初值
               sem_arg.val = sem_val;
               if(semctl(sem_id,0,SETVAL,sem_arg) <0)
                 {
            perror("semaphore set error");
            exit(EXIT_FAILURE);
                    }
        }

    return sem_id;
}

/*
*    set_shm 函数建立一个具有 n 个字节 的共享内存区
*          如果建立成功，返回 一个指向该内存区首地址的指针 shm_buf
*          输入参数：
*    shm_key 共享内存的键值
*    shm_val 共享内存字节的长度
*    shm_flag 共享内存的存取权限
*/
char * set_shm(key_t shm_key,int shm_num,int shm_flg)
{
    int i,shm_id;
    char * shm_buf;

          //测试由 shm_key 标识的共享内存区是否已经建立
    if((shm_id = get_ipc_id("/proc/sysvipc/shm",shm_key)) < 0 )
          {
        //shmget 新建 一个长度为 shm_num 字节的共享内存,其标号返回到 shm_id
        if((shm_id = shmget(shm_key,shm_num,shm_flg)) <0)
                  {
          perror("shareMemory set error");
          exit(EXIT_FAILURE);
                  }
        //shmat 将由 shm_id 标识的共享内存附加给指针 shm_buf
        if((shm_buf = (char *)shmat(shm_id,0,0)) < (char *)0)
                  {
          perror("get shareMemory error");
          exit(EXIT_FAILURE);
                  }
        for(i=0; i<shm_num; i++) shm_buf[i] = 0; //初始为 0
           }
    //shm_key 标识的共享内存区已经建立,将由 shm_id 标识的共享内存附加给指
针 shm_buf
    if((shm_buf = (char *)shmat(shm_id,0,0)) < (char *)0)
         {
         perror("get shareMemory error");
```

```
            exit(EXIT_FAILURE);
            }

    return shm_buf;
}

/*
*    set_msq 函数建立一个消息队列
*    如果建立成功，返回 一个消息队列的标识符 msq_id
*    输入参数：
*    msq_key 消息队列的键值
*    msq_flag 消息队列的存取权限
*/
int set_msq(key_t msq_key,int msq_flg)
{
    int msq_id;

     //测试由 msq_key 标识的消息队列是否已经建立
    if((msq_id = get_ipc_id("/proc/sysvipc/msg",msq_key)) < 0 )
     {
       //msgget 新建一个消息队列,其标号返回到 msq_id
       if((msq_id = msgget(msq_key,msq_flg)) < 0)
          {
            perror("messageQueue set error");
        exit(EXIT_FAILURE);
          }
     }
    return msq_id;
}
```

3) 在当前新文件夹中建立生产者程序 **producer.c**

```
/*
   *   Filename              : producer.c
   *   copyright             : (C) 2006 by zhonghonglie
   *   Function              : 建立并模拟生产者进程
*/

#include "ipc.h"

int main(int argc,char *argv[])
{
   int rate;
   //可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度
   if(argv[1] != NULL)    rate = atoi(argv[1]);
   else rate = 3;    //不指定为 3 秒

   //共享内存使用的变量
```

```
buff_key = 101;//缓冲区任给的键值
buff_num = 8;//缓冲区任给的长度
pput_key = 102;//生产者放产品指针的键值
pput_num = 1; //指针数
shm_flg = IPC_CREAT | 0644;//共享内存读写权限

//获取缓冲区使用的共享内存，buff_ptr 指向缓冲区首地址
buff_ptr = (char *)set_shm(buff_key,buff_num,shm_flg);
//获取生产者放产品位置指针 pput_ptr
pput_ptr = (int *)set_shm(pput_key,pput_num,shm_flg);


//信号量使用的变量
prod_key = 201;//生产者同步信号灯键值
pmtx_key = 202;//生产者互斥信号灯键值
cons_key = 301;//消费者同步信号灯键值
cmtx_key = 302;//消费者互斥信号灯键值
sem_flg = IPC_CREAT | 0644;

//生产者同步信号灯初值设为缓冲区最大可用量
sem_val = buff_num;
//获取生产者同步信号灯，引用标识存 prod_sem
prod_sem = set_sem(prod_key,sem_val,sem_flg);

//消费者初始无产品可取，同步信号灯初值设为 0
sem_val = 0;
//获取消费者同步信号灯，引用标识存 cons_sem
cons_sem = set_sem(cons_key,sem_val,sem_flg);

//生产者互斥信号灯初值为 1
sem_val = 1;
//获取生产者互斥信号灯，引用标识存 pmtx_sem
pmtx_sem = set_sem(pmtx_key,sem_val,sem_flg);

//循环执行模拟生产者不断放产品
while(1){
      //如果缓冲区满则生产者阻塞
    down(prod_sem);
    //如果另一生产者正在放产品，本生产者阻塞
    down(pmtx_sem);

     //用写一字符的形式模拟生产者放产品，报告本进程号和放入的字符及存放的位置
       buff_ptr[*pput_ptr] = 'A'+ *pput_ptr;
         sleep(rate);
printf("%d producer put: %c to Buffer[%d]\n",getpid(),buff_ptr[*pput_ptr],*pput_ptr);
```

```
            //存放位置循环下移
            *pput_ptr = (*pput_ptr+1) % buff_num;

           //唤醒阻塞的生产者
           up(pmtx_sem);
           //唤醒阻塞的消费者
           up(cons_sem);
          }

       return EXIT_SUCCESS;
}
```

4) 在当前新文件夹中建立消费者程序 **consumer.c**

```
/*
      Filename          : consumer.c
      copyright         : (C) by zhanghonglie
      Function          : 建立并模拟消费者进程
*/

#include "ipc.h"

int main(int argc,char *argv[])
{
   int rate;
   //可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度
   if(argv[1] != NULL)    rate = atoi(argv[1]);
   else rate = 3;   //不指定为 3 秒

      //共享内存 使用的变量
   buff_key = 101; //缓冲区任给的键值
   buff_num = 8;     //缓冲区任给的长度
   cget_key = 103; //消费者取产品指针的键值
   cget_num = 1;     //指针数
   shm_flg = IPC_CREAT | 0644; //共享内存读写权限

       //获取缓冲区使用的共享内存，buff_ptr 指向缓冲区首地址
   buff_ptr = (char *)set_shm(buff_key,buff_num,shm_flg);
    //获取消费者取产品指针，cget_ptr 指向索引地址
   cget_ptr = (int *)set_shm(cget_key,cget_num,shm_flg);

      //信号量使用的变量
   prod_key = 201;   //生产者同步信号灯键值
   pmtx_key = 202;   //生产者互斥信号灯键值
   cons_key = 301;   //消费者同步信号灯键值
   cmtx_key = 302;   //消费者互斥信号灯键值
   sem_flg = IPC_CREAT | 0644; //信号灯操作权限
```

```
    //生产者同步信号灯初值设为缓冲区最大可用量
    sem_val = buff_num;
    //获取生产者同步信号灯，引用标识存 prod_sem
    prod_sem = set_sem(prod_key,sem_val,sem_flg);

      //消费者初始无产品可取，同步信号灯初值设为 0
    sem_val = 0;
  //获取消费者同步信号灯，引用标识存 cons_sem
    cons_sem = set_sem(cons_key,sem_val,sem_flg);

    //消费者互斥信号灯初值为 1
    sem_val = 1;
    //获取消费者互斥信号灯，引用标识存 pmtx_sem
    cmtx_sem = set_sem(cmtx_key,sem_val,sem_flg);

       //循环执行模拟消费者不断取产品
    while(1){
                  //如果无产品消费者阻塞
        down(cons_sem);
                     //如果另一消费者正在取产品，本消费者阻塞
         down(cmtx_sem);

                           //用读一字符的形式模拟消费者取产品，报告本进程号和
  获取的字符及读取的位置
            sleep(rate);
            printf("%d consumer get: %c from
  Buffer[%d]\n",getpid(),buff_ptr[*cget_ptr],*cget_ptr);
                              //读取位置循环下移
            *cget_ptr = (*cget_ptr+1) % buff_num;

          //唤醒阻塞的消费者
          up(cmtx_sem);
            //唤醒阻塞的生产者
        up(prod_sem);
       }

     return EXIT_SUCCESS;
  }
```

5）在当前文件夹中建立 **Makefile** 项目管理文件

```
    hdrs = ipc.h
    opts = -g -c
    c_src = consumer.c ipc.c
    c_obj = consumer.o ipc.o
    p_src = producer.c ipc.c
    p_obj = producer.o ipc.o
```

```
all:        producer consumer

consumer:   $(c_obj)
            gcc $(c_obj) -o consumer
consumer.o: $(c_src) $(hdrs)
            gcc $(opts) $(c_src)

producer: $(p_obj)
            gcc $(p_obj) -o producer
producer.o: $(p_src) $(hdrs)
            gcc $(opts) $(p_src)
clean:
            rm consumer producer *.o
```

6）使用 **make** 命令编译连接生成可执行的生产者、消费者程序

```
$ gmake
gcc -g -c producer.c ipc.c
gcc producer.o ipc.o -o producer
gcc -g -c consumer.c ipc.c
gcc consumer.o ipc.o -o consumer
```

7）在当前终端窗体中启动执行速率为 **1** 秒的一个生产者进程

```
$./producer   1
12263 producer put: A to Buffer[0]
12263 producer put: B to Buffer[1]
12263 producer put: C to Buffer[2]
12263 producer put: D to Buffer[3]
12263 producer put: E to Buffer[4]
12263 producer put: F to Buffer[5]
12263 producer put: G to Buffer[6]
12263 producer put: H to Buffer[7]
```

可以看到 12263 号进程在向共享内存中连续写入了 8 个字符后因为缓冲区满而阻塞。

8） 打开另一终端窗体，进入当前工作目录，从中再启动另一执行速率为 **3** 的生产者进程**:**

```
$ ./producer 3
```

可以看到该生产者进程因为缓冲区已满而立即阻塞。

9)再打开另外两个终端窗体，进入当前工作目录，从中启动执行速率为 **2** 和 **4** 的两个消费者进程**:**

```
$ ./consumer 2
12528 consumer get: B from Buffer[1]
12528 consumer get: D from Buffer[3]
12528 consumer get: F from Buffer[5]
```

12528 consumer get: H from Buffer[7]
……
$./consumer 4
12529 consumer get: A from Buffer[0]
12529 consumer get: C from Buffer[2]
12529 consumer get: E from Buffer[4]
12529 consumer get: G from Buffer[6]
……

在第一个生产者窗体中生产者 1 被再此唤醒输出:

12263 producer put: B to Buffer[1]
12263 producer put: D to Buffer[3]
12263 producer put: F to Buffer[5]
12263 producer put: H to Buffer[7]
……

在第二个生产者窗体中生产者 2 也被再此被唤醒输出

12264 producer put: A to Buffer[0]
12264 producer put: C to Buffer[2]
12264 producer put: E to Buffer[4]
12264 producer put: G to Buffer[6]

可以看到由于消费者进程读出了写入缓冲区的字符，生产者从新被唤醒继续向读过的缓冲区单元中同步的写入字符。

请用 ctrl+C 将两生产者进程打断，观察两消费者进程是否在读空缓冲区后而阻塞。反之，请用 ctrl+C 将两消费者进程打断，观察两生产者进程是否在写满缓冲区后而阻塞。

4.4 独立实验

抽烟者问题。假设一个系统中有三个抽烟者进程，每个抽烟者不断地卷烟并抽烟。抽烟者卷起并抽掉一颗烟需要有三种材料：烟草、纸和胶水。一个抽烟者有烟草，一个有纸，另一个有胶水。系统中还有两个供应者进程，它们无限地供应所有三种材料，但每次仅轮流提供三种材料中的两种。得到缺失的两种材料的抽烟者在卷起并抽掉一颗烟后会发信号通知供应者，让它继续提供另外的两种材料。这一过程重复进行。 请用以上介绍的 IPC 同步机制编程，实现该问题要求的功能。

4.5 实验要求

根据示例实验程序和独立实验程序中观察和记录的信息，结合生产者/消费者问题和抽烟者问题的算法的原理，说明真实操作系统中提供的并发进程同步机制是怎样实现和解决同步问题的，它们是怎样应用操作系统教材中讲解的进程同步原理的？对应教材中信号灯的定义，说明信号灯机制是怎样完成进程的互斥和同步的？其中信号量的初值和其值的变化的物理意义是什么？使用多于 4 个的生产者和消费者，以各种不同的启动顺序、不同的执行速率检测以上示例程序和独立实验程序是否都能满足同步的要求。根据实验程序、调试过程和结果分析写出实验报告。