操作系统课程设计实验报告

实验名称：生产者消费者问题

姓名/学号：刘鑫/1120181208

一、 实验目的

使用进程模拟消费者生产者进程，掌握进程之间的通信方法，区分在不同操作系统下进程通信的不同之处，并熟练使用信号量的创建删除等一系列操作

二、 实验内容

- 一个大小为 3 的缓冲区，初始为空
- 2 个生产者
  - 随机等待一段时间，往缓冲区添加数据，
  - 若缓冲区已满，等待消费者取走数据后再添加
  - 重复 6 次
- 3 个消费者
  - 随机等待一段时间，从缓冲区读取数据
  - 若缓冲区为空，等待生产者添加数据后再读取
  - 重复 4 次

说明：

- 显示每次添加和读取数据的时间及缓冲区里的数据（指缓冲区里的具体内容）
- 生产者和消费者用进程模拟（不能用线程）
- Linux/Window 下都需要做

三、 实验环境

Windows：

Windows10

处理器 Inter core i5-8265u@1.60GHz

内存 8.00GB

系统类型 64 位操作系统，基于 x64 的处理器

Linux：

虚拟机软件：VMware workstation 15

虚拟机操作系统：Ubuntu20.04

虚拟机内存：4GB

虚拟机硬盘容量:60GB

四、 程序设计与实现

一、在 Windows 下的实现：

要实现生产者消费者模型的模拟，首先要创建文件缓冲区，在 Windows 中，通过文件映射的方式来实现创建缓冲区。

定义缓冲区结构

```
//定义缓冲结构
struct buffer {
    int buffer[BUFFER_LEGHTH];
    int head;
    int tail;
    boolean isEmpty;
};
```

定义共享内存的结构

```
//定义共享内存
struct sharedMemery {
    struct buffer data;
    HANDLE sharedMemery_empty;
    HANDLE sharedMemery_full;
    HANDLE sharedMemery_mutex;
};
```

然后使用 GreateFileMapping()创建文件映射对象,API 的解释如下：

```
HANDLE CreateFileMapping(
  HANDLE hFile,                    //物理文件句柄
  LPSECURITY_ATTRIBUTES lpAttributes, //安全设置
  DWORD flProtect,                 //保护设置
  DWORD dwMaximumSizeHigh,         //高位文件大小
  DWORD dwMaximumSizeLow,          //低位文件大小
  LPCTSTR lpName                   //共享内存名称
);
```

返回一个文件对象句柄，通过文件对象句柄对该映射对象进行初始化等一系列操作

```
//创建共享文件区
HANDLE buildSharedFile() {
  //创建文件映射对象
  HANDLE hMapping = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE,0,sizeof(struct sharedMemery), "Mybuffer");
  if (hMapping != INVALID_HANDLE_VALUE) {
    //在文件上创建视图,这样进程就可以像访问主存一样访问文件
    LPVOID pData = MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);
    if (pData != NULL) {//对视图进行初始化
      ZeroMemory(pData, sizeof(struct sharedMemery));
    }
    //关闭文件视图
    UnmapViewOfFile(pData);
  }
  return hMapping;
```

在主函数中，根据不同的进程号，判断是消费者进程还是生成者进程，并采取不同的操作

在主进程中，先创建映射视图和信号量并初始化

MapViewOfFile () 用于将视图映射到创建的共享文件，API 的解释如下：

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,
  DWORD  dwDesiredAccess,
  DWORD  dwFileOffsetHigh,
  DWORD  dwFileOffsetLow,
  SIZE_T dwNumberOfBytesToMap
);
```

参数：

hFileMappingObject 文件映射对象的句柄

dwDesiredAccess 文件映射对象的访问类型，取值为 FILE_MAP_ALL_ACCESS 表示对映射文件对象有读写的权利

dwFileOffsetHigh 文件高阶偏移量

dwFileOffsetLow 文件低阶偏移量

dwNumberOfBytesToMap 映射到视图的文件映射的字节数

CreateSemaphore() 此 API 用于创建或打开一个信号量，具体解释如下：

```
HANDLE CreateSemaphoreA(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG                  lInitialCount,
  LONG                  lMaximumCount,
  LPCSTR                lpName
);
```

参数：

lpSemaphoreAttributes：为信号量的属性，一般可以设置为 NULL

lInitialCount：信号量初始值，必须大于等于 0，而且小于等于 lpMaximumCount，如果 lInitialCount 的初始值为 0，则该信号量默认为 unsignal 状态，如果 lInitialCount 的初始值大于 0，则该信号量默认为 signal 状态

lMaximumCount： 此值为设置信号量的最大值，必须大于 0

lpName：信号量的名字，长度不能超出 MAX_PATH ，可设置为 NULL，表示无名的信号量。当 lpName 不为空时，可创建有名的信号量，若当前信号量名与已存在的信号量的名字相同时，则该函数表示打开该信号量，这时参数 lInitialCount 和 lMaximumCount 将被忽略。

在 main 函数的主线程中，创建缓冲区和信号量并初始化

```
if (nclone == 0) {//主进程
  printf("主进程开始运行\n");
  //创建共享数据文件
  hMapping = buildSharedFile();
  //创建映射视图
  HANDLE hFileMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, "Mybuffer");
  LPVOID pFile = MapViewOfFile(hFileMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);

  if (pFile == NULL) {//若返回空指针说明创建视图失败
    printf("OpenFileMapping failed!\n");
  }
  //对缓冲区进行初始化
  struct sharedMemery* shm = (struct sharedMemery*)pFile;
  shm->data.head = 0;
  shm->data.tail = 0;
  shm->data.isEmpty = TRUE;
  //创建信号量并且初始化
  shm->sharedMemery_mutex = CreateSemaphore(NULL, 1, 1, (LPCSTR)"SEM_MUTEX");
  shm->sharedMemery_full = CreateSemaphore(NULL, 0, BUFFER_LEGHTH, (LPCSTR)"SEM_FULL");
  shm->sharedMemery_empty = CreateSemaphore(NULL, BUFFER_LEGHTH, BUFFER_LEGHTH, (LPCSTR)"SEM_EMPTY");
  //关闭视图
  UnmapViewOfFile(pFile);
  pFile = NULL;
  CloseHandle(hFileMapping);
```

在生产者进程中，首先使用 OpenFileMapping()，此 API 的作用是打开一个命名的文件映射对象。该 API 的详细解释如下：

OpenFileMapping (

ByVal dwDesiredAccess As Long, //带有前缀 FILE_MAP_???的常数

ByVal bInheritHandle As Long, //如这个函数返回的句柄能由当前进程启动的新进程继承，则这个参数为 TRUE

ByVal lpName As String //指定要打开的文件映射对象名称

)

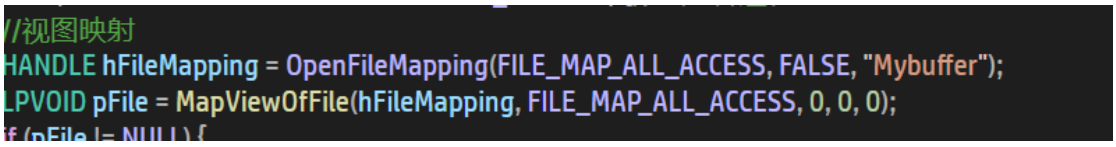
打开共享内存区后，再使用 MapViewOfFile()将共享内存区映射到当前进程的地址空间。MapViewOfFile()的作用是将一个文件映射对象映射到当前的应用程序的地址空间，该 API 的解释如下：

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,
  DWORD  dwDesiredAccess,
  DWORD  dwFileOffsetHigh,
  DWORD  dwFileOffsetLow,
  SIZE_T dwNumberOfBytesToMap
);
```

参数：

hFileMappingObject	文件映射对象的句柄
dwDesiredAccess	对文件映射对象的访问类型
dwFileOffsetHigh	视图开始处文件的高阶偏移量
dwFileOffsetLow	视图开始处文件的低阶偏移量
dwNumberOfBytesToMap	映射到视图的字节数。如果此参数为 0（零），则映射将从指定的偏移量扩展到文件映射的末尾。

相关部分代码如下：



然后使用 OpenSemaphore()打开信号量并对信号量进行初始化

```
//初始化信号量
struct sharedMemery* shm = (struct sharedMemery*)pFile;
HANDLE mutex= OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, (LPCSTR)"SEM_MUTEX");
HANDLE full=OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, (LPCSTR)"SEM_FULL");
HANDLE empty = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, (LPCSTR)"SEM_EMPTY");
```

在每一次写操作先调用生成随机数的函数，返回的随机数作为生产者写入缓冲区的数据，同时也作为进程运行前随机睡眠的时间。

使用 WaitforSingleObject 来实现信号量的 p 操作

```
//获取进程随机等待时间
int sleeptime = getRandom();
Sleep(sleeptime);
WaitForSingleObject(empty, INFINITE);//p(empty)
WaitForSingleObject(mutex, INFINITE);//p(mutex)
int data = getRandom();//生成写入的数据
//循环缓冲区的实现
shm->data.buffer[shm->data.tail] = data;
shm->data.isEmpty = FALSE;
shm->data.tail += 1;
shm->data.tail %= 3;
```

相对应的，最后通过 ReleaseSemaphore()来实现对信号量的 v 操作

然后关闭视图，断开与共享内存之间的连接

```
  ReleaseSemaphore(full, 1, NULL);//v(full)
  ReleaseSemaphore(mutex, 1, NULL);//v(mutex);

}
UnmapViewOfFile(pFile); //关闭视图
```

在消费者进程中，大致的步骤与生产者进程一样，要注意的是对信号量的操作是不同的

```
WaitForSingleObject(full, INFINITE);//P(full);
WaitForSingleObject(mutex, INFINITE);//p(mutex)
```

```
    ReleaseSemaphore(empty, 1, NULL);//v(full)
    ReleaseSemaphore(mutex, 1, NULL);//v(mutex);
}
UnmapViewOfFile(pFile);
```

实验运行结果：

```
主进程开始运行
当前系统时间：00时，29分，16秒
生产者进程1把597写入到缓冲区中
当前缓冲区内的数据：
597

当前系统时间：16时，29分，16秒
消费者进程2把597从缓冲区中取走
当前缓冲区已经没有数据


当前系统时间：00时，29分，16秒
生产者进程2把845写入到缓冲区中
当前缓冲区内的数据：
845

当前系统时间：16时，29分，16秒
消费者进程1把845从缓冲区中取走
当前缓冲区已经没有数据


当前系统时间：00时，29分，16秒
生产者进程1把597写入到缓冲区中
当前缓冲区内的数据：
597

当前系统时间：16时，29分，16秒
消费者进程3把597从缓冲区中取走
当前缓冲区已经没有数据


当前系统时间：00时，29分，17秒
生产者进程2把848写入到缓冲区中
当前缓冲区内的数据：
848

当前系统时间：16时，29分，17秒
消费者进程2把848从缓冲区中取走
当前缓冲区已经没有数据


当前系统时间：00时，29分，17秒
生产者进程1把600写入到缓冲区中
当前缓冲区内的数据：
600
```

```
当前系统时间：16时，29分，17秒
消费者进程1把600从缓冲区中取走
当前缓冲区已经没有数据


当前系统时间：00时，29分，18秒
生产者进程1把603写入到缓冲区中
当前缓冲区内的数据：
603

当前系统时间：16时，29分，18秒
消费者进程3把603从缓冲区中取走
当前缓冲区已经没有数据


当前系统时间：00时，29分，18秒
生产者进程2把851写入到缓冲区中
当前缓冲区内的数据：
851

当前系统时间：16时，29分，18秒
消费者进程2把851从缓冲区中取走
当前缓冲区已经没有数据


当前系统时间：00时，29分，18秒
生产者进程1把603写入到缓冲区中
当前缓冲区内的数据：
603

当前系统时间：16时，29分，18秒
消费者进程1把603从缓冲区中取走
当前缓冲区已经没有数据


当前系统时间：00时，29分，19秒
生产者进程2把855写入到缓冲区中
当前缓冲区内的数据：
855

当前系统时间：16时，29分，19秒
消费者进程2把855从缓冲区中取走
当前缓冲区已经没有数据
```

```
当前系统时间：00时，29分，19秒
生产者进程2把855写入到缓冲区中
当前缓冲区内的数据：
855

当前系统时间：16时，29分，19秒
消费者进程1把855从缓冲区中取走
当前缓冲区已经没有数据


当前系统时间：00时，29分，20秒
生产者进程2把858写入到缓冲区中
当前缓冲区内的数据：
858

当前系统时间：16时，29分，20秒
消费者进程3把858从缓冲区中取走
当前缓冲区已经没有数据


主进程运行结束

D:\c_c++\OS_experiment3\Debug\OS_experiment3.exe (进程 12316)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

实验结果：

成功在 Windows 下使用进程模拟了生产者消费者模型

在 Linux 下的实现：

首先定义缓冲区结构

```
//缓冲区结构
struct mybuffer {
    int buffer[BUFFER_SIZE];
    int head;
    int tail;
    int isEmpty;
};
```

然后使用 semget()创建信号量集，其中包含 SEM_FULL，SEM_EMPTY，SEM_MUTEX 三个信号量，并使用 semtcl()对信号量进行初始化

```
//创建信号量集,返回信号标识符
int sem_id = semget(SEM_KEY, 3, IPC_CREAT | 0660);
```

```
//初始化信号量集
union semun sem_val;
sem_val.value = BUFFER_SIZE;
//empty初始化为缓冲区长度
semtcl(sem_id, SEM_EMPTY, SETVAL, sem_val);
sem_val.value = 0;
//初始化full为0
semtcl(sem_id, SEM_FULL, SETVAL, sem_val);
sem_val.value = 1;
//初始化mutex为1
semtcl(sem_id, SEM_MUTEX, SETVAL, sem_val);
```

接着使用 shmget()创建共享内存区，改 API 的具体解释如下：

```
int shmget(key_t key, size_t size, int shmflg);
```

key　非 0 整数，为共享内存段命名

size　以字节为单位指定需要共享的内存容量

shmflg 权限标志，它的作用与 open 函数的 mode 参数一样，如果要想在 key 标识的共享内存不存在时，创建它的话，可以与 IPC_CREAT 做或操作。

相关代码如图：

```
//初始化信号量集
union semun sem_val;
sem_val.value = BUFFER_SIZE;
//empty初始化为缓冲区长度
semtcl(sem_id, SEM_EMPTY, SETVAL, sem_val);
sem_val.value = 0;
//初始化full为0
semtcl(sem_id, SEM_FULL, SETVAL, sem_val);
sem_val.value = 1;
//初始化mutex为1
semtcl(sem_id, SEM_MUTEX, SETVAL, sem_val);

//创建共享内存区
int shm_id = shmget(SHM_KEY, sizeof(struct mybuffer), 0600 | IPC_CREAT);
if (shm_id < 0) {
    //创建失败
    printf("创建共享内存区失败\n");
    exit(1);
}
```

再使用 shmat()函数将创建的共享内存区连接到父进程的地址空间上，然后初始化缓冲区。shmat()API 的解释如下：

```
void *shmat(int shm_id, const void *shm_addr, int shmflg);
```

该函数作用是用来启动对共享内存的访问，并把共享内存连接到当前进程的地址空间

参数：

shm_id　　由 shmget()函数返回的共享内存标识。

shm_addr　指定共享内存连接到当前进程中的地址位置，通常为空，表示让系统来选择共享内存的地址。

shm_flg　　一组标志位，通常为 0。

部分相关代码如图：

```
//初始化缓冲区,启动对该共享区的访问，并把它连接到当前进程的地址空间
struct mybuffer* shmaddr = shmat(shm_id, 0, 0);
if (shmaddr == (void*)-1) {
    printf("共享内存区连接失败\n");
    _Exit(1);
}
shmaddr->head = 0;
shmaddr->tail = 0;
shmaddr->isEmpty = 1;
```

信号量和缓冲区创建到这里就已经完成了，接下来是消费者和生产者进程。

首先源码中的 p 操作和 v 操作部分代码如图：

```
//p 操作
void p(int sem_id, int sem_num) {
    struct sembuf sbf;
    sbf.sem_num = sem_num;
    sbf.sem_flg = 0;
    sbf.sem_op = -1;
    semop(sem_id, &sbf, 1);
}
```

```
//v操作
void v(int sem_id, int sem_num) {
    struct sembuf sbf;
    sbf.sem_num = sem_num;
    sbf.sem_flg = 0;
    sbf.sem_op = 1;
    semop(sem_id, &sbf, 1);
}
```

在生产者进程中，首先使用 shmat()函数将共享内存连接到生产者进程的地址空间上，然后在依次对 SEM_EMPTY 和 SEM_MUTEX 进行 p 操作，随机睡眠一段时间后向缓冲区写入数据

```
//将共享内存区连接到生产者进程的地址空间上
if ((shmaddr =static_cast<struct mybuffer*>( shmat(shm_id, 0, 0))) == (void*)-1) {
    printf("共享内存连接失败\n");
    exit(1);
}
//写入数据
for (int i = 0; i < PRODUCER_TIMES; i++) {
    p(sem_id, SEM_EMPTY);//p(empty)
    int data = getRandom();
    sleep(data);
    p(sem_id, SEM_MUTEX);//p(mutex)
    shmaddr->buffer[shmaddr->tail] = data;
    shmaddr->isEmpty = 0;
    shmaddr->tail = (shmaddr->tail + 1) % BUFFER_SIZE;
```

完成操作后对 SEM_FULL 和 SEM_MUTEX 进行 v 操作，并断开共享内存和进程之间的连接

```
    v(shm_id, SEM_FULL);//v(full)
    v(shm_id, SEM_MUTEX);//v(mutex)

}
//断开共享内存连接
shmdt(shmaddr);
exit(0);
```

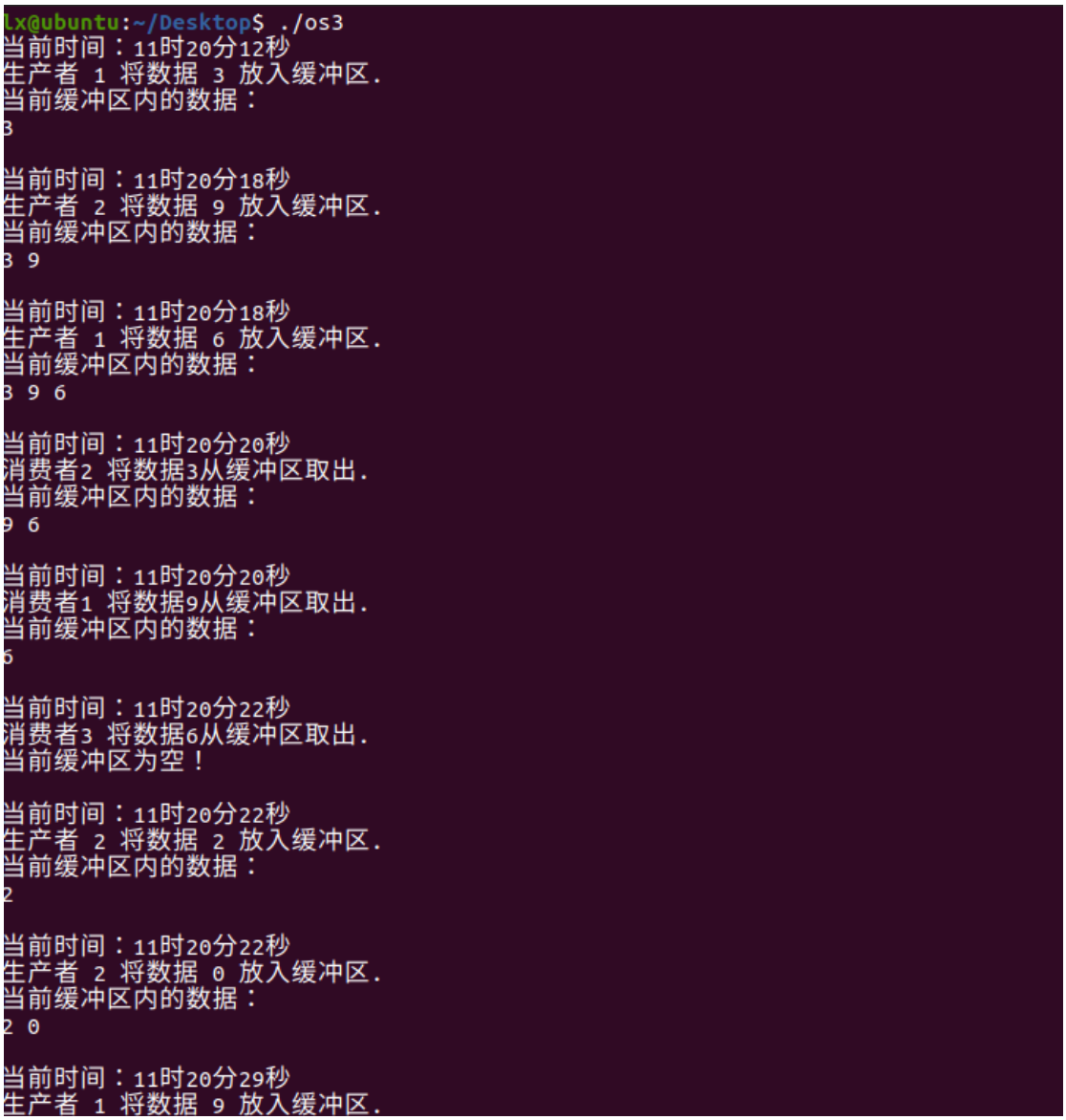
消费者进程与生产者进程基本的操作一致，也是先将共享内存和进程的地址空间进行连接，不过这里是对 SEM_FULL 和 SEM_EMPTY 先后进行 p 操作，随机睡眠一段时间后从缓冲区取出数据，最后再对 SEM_EMPTY 和 SEM_MUTEX 进行 v 操作，然后断开共享内存的连接

```
if ((shmaddr =static_cast<struct mybuffer*> (shmat(shm_id, 0, 0)) )== (void*)-1) {
    printf("共享内存连接失败\n");
    exit(1);
}
for (int i = 0; i < CUSTOMER_TIMES; i++) {//消费者读取数据
    p(shm_id, SEM_FULL);
    int sleeptime = getRandom();
    sleep(sleeptime);
    p(shm_id, SEM_MUTEX);
    data = shmaddr->buffer[shmaddr->head];
    shmaddr->head = (shmaddr->head + 1) % BUFFER_SIZE;
    shmaddr->isEmpty = (shmaddr->head == shmaddr->tail);
    nowtime = time(NULL);
    printf("当前时间：%02d时%02d分%02d秒\n", localtime(&nowtime)->tm_hour, localtime(&nowtime)->tm_min, localtime(&nowtime)->tm_sec);
    printf("消费者%d 将数据%d从缓冲区取出.\n", num_c, data);
    int bufflength = (shmaddr->tail + BUFFER_SIZE - shmaddr->head) % BUFFER_SIZE;
    if (bufflength == 0)
        printf("当前缓冲区为空！ \n\n");
    else{...}
    fflush(stdout);
    v(sem_id, SEM_MUTEX);  //解锁
    v(sem_id, SEM_EMPTY);

}
shmdt(shmaddr);
exit(0);
```

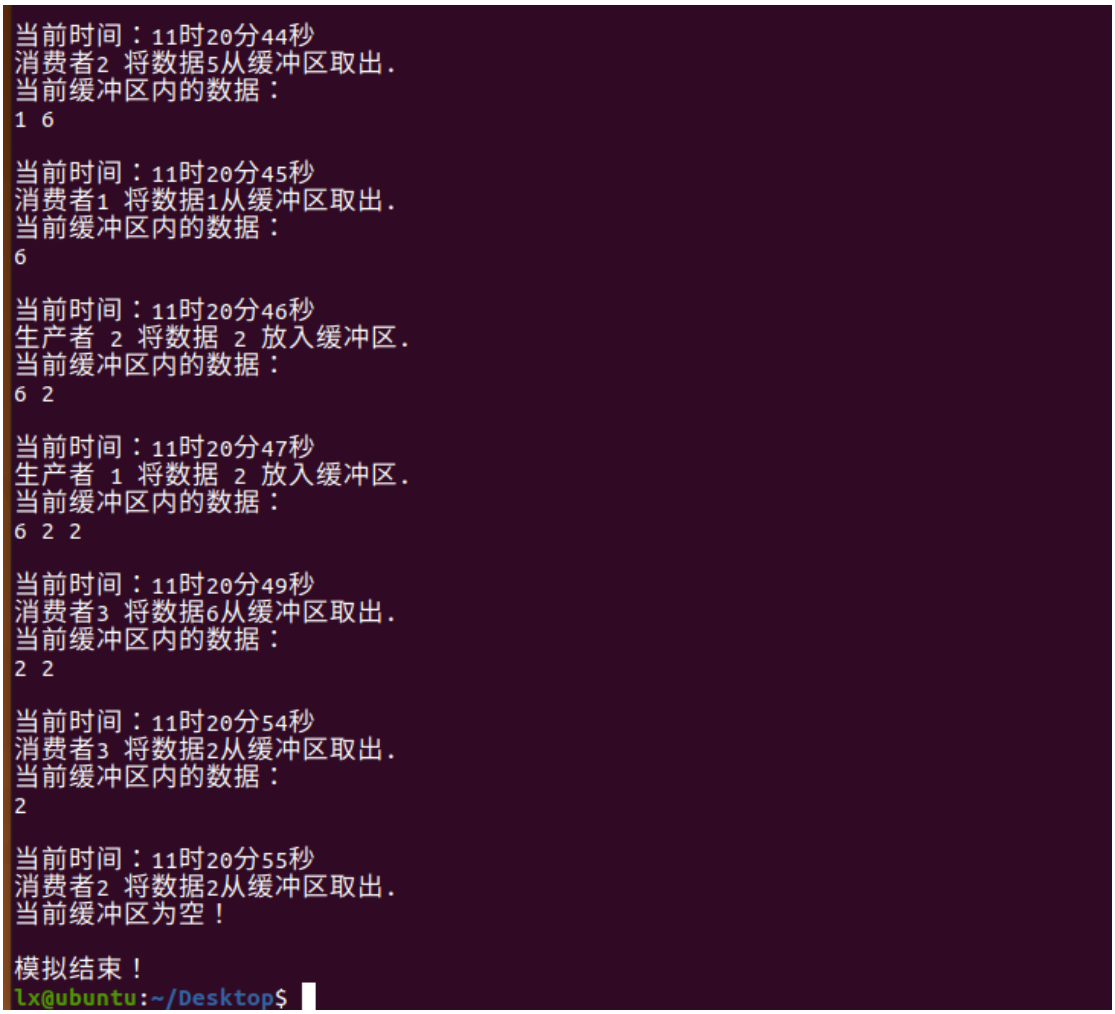
最后，在所有的子进程运行结束后，父进程要删除信号量集和共享内存

```
while (wait(0) != -1);
shmdt(shmaddr);
shmctl(shm_id, IPC_RMID, 0);
semctl(sem_id, IPC_RMID, 0);
```

实验运行结果如图：


lx@ubuntu:~/Desktop$ ./os3
当前时间：11时20分12秒
生产者 1 将数据 3 放入缓冲区.
当前缓冲区内的数据：
3
当前时间：11时20分18秒
生产者 2 将数据 9 放入缓冲区.
当前缓冲区内的数据：
3 9
当前时间：11时20分18秒
生产者 1 将数据 6 放入缓冲区.
当前缓冲区内的数据：
3 9 6
当前时间：11时20分20秒
消费者2 将数据3从缓冲区取出.
当前缓冲区内的数据：
9 6
当前时间：11时20分20秒
消费者1 将数据9从缓冲区取出.
当前缓冲区内的数据：
6
当前时间：11时20分22秒
消费者3 将数据6从缓冲区取出.
当前缓冲区为空！
当前时间：11时20分22秒
生产者 2 将数据 2 放入缓冲区.
当前缓冲区内的数据：
2
当前时间：11时20分22秒
生产者 2 将数据 0 放入缓冲区.
当前缓冲区内的数据：
2 0
当前时间：11时20分29秒
生产者 1 将数据 9 放入缓冲区.

```
当前缓冲区内的数据：
2 0 9

当前时间：11时20分31秒
消费者1 将数据2从缓冲区取出.
当前缓冲区内的数据：
0 9

当前时间：11时20分31秒
消费者2 将数据0从缓冲区取出.
当前缓冲区内的数据：
9

当前时间：11时20分34秒
生产者 1 将数据 3 放入缓冲区.
当前缓冲区内的数据：
9 3

当前时间：11时20分36秒
生产者 2 将数据 5 放入缓冲区.
当前缓冲区内的数据：
9 3 5

当前时间：11时20分37秒
消费者3 将数据9从缓冲区取出.
当前缓冲区内的数据：
3 5

当前时间：11时20分42秒
消费者1 将数据3从缓冲区取出.
当前缓冲区内的数据：
5

当前时间：11时20分43秒
生产者 2 将数据 1 放入缓冲区.
当前缓冲区内的数据：
5 1

当前时间：11时20分43秒
生产者 1 将数据 6 放入缓冲区.
当前缓冲区内的数据：
5 1 6
```



实验结果：

在 Linux 下成功用线程模拟了生产者消费者模型

五、 实验收获与体会

通过本次实验，我对 Windows 和 Linux 下的进程通信有了更加深入的认识，同时也熟悉了相关 API，共享内存区和信号量的使用。而且在操作实践中，对消

费者生产者这个经典模型有了更加全面的认识，掌握它是如何在不同的操作系统中实际运行的，对课程的理论知识的认识有了更加坚实的实践基础。