北京理工大學

# 实验三 生产者消费者


班　　级：07111505

姓　　名：徐宇恒

学　　号：1120151839

# 目录

# 一. 实验目的

在 Windows 和 Linux 环境下利用进程模拟生产者消费者问题，通过自学互斥信号量来实现进程之间的互斥和同步操作，并采用共享内存来实现缓冲区的数据存储

# 二. 实验内容

## 2.1. 创建两个生产者

- 随机等待一段时间，向缓冲池中添加数据。
- 若缓冲区已满，等待消费者取走数据后在添加
- 重复 12 次

## 2.2. 创建三个消费者

- 速记等待一段时间，从缓冲区中取出数据。
- 若缓冲区为空，则等待生产者添加数据
- 重复 8 次

## 2.3. 创建一个大小为 6 的缓冲区

- 初始为空
- 允许消费者进程读数据
- 允许生产者进程写数据

## 2.4. 显示结果

- 显示每次添加和读取数据的时间
- 显示缓冲区的状态

# 三. 实验环境

| | Windows | Linux |
|---|---|---|
| 操作系统 | Windows10 Pro 64bit | Ubuntu 16.04LTS |

| 编译器 | Visual Studio 2017 IDE | Gcc |
| --- | --- | --- |

# 四. 实验过程

## 4.1. 基本思路

在 Windows 和 Linux 下分别编写三个程序，主程序命名为“Program”，两个子程序“Producer”，“Customer”。

### 4.1.1. “Program”程序：

- 建立并初始化缓冲区
- 创建互斥信号量 Mutex，空信号量 Empty，满信号量 Full
- 创建生产者和消费者子进程

### 4.1.2. “Producer”程序：

- 打开缓冲区
- 打开互斥信号量、空信号量、满信号量
- 对空信号量和互斥信号量进行 P 操作
- 申请成功则放入数据
- 对满信号量和互斥信号量进行 V 操作
- 返回到第三步，重复 12 次

### 4.1.3. “Customer”程序：

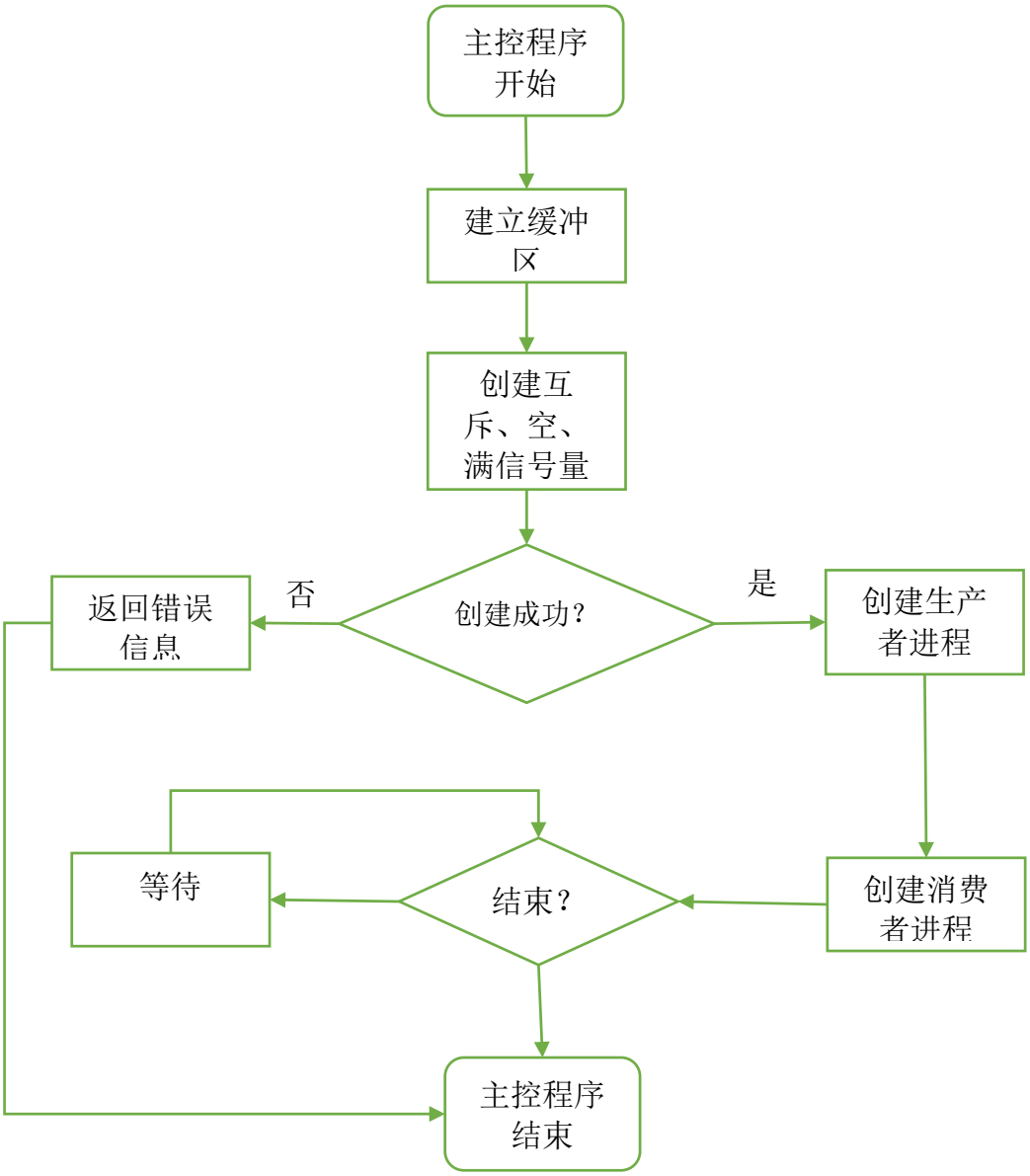
- 打开缓冲区
- 打开互斥访问信号量、空信号量、满信号量
- 若打开成功，对满信号量做 P 操作，对互斥信号量做 P 操作
- 若申请成功，则读取数据，并将当前缓冲区置零
- 对空信号量做 V 操作，对互斥信号量做 V 操作。

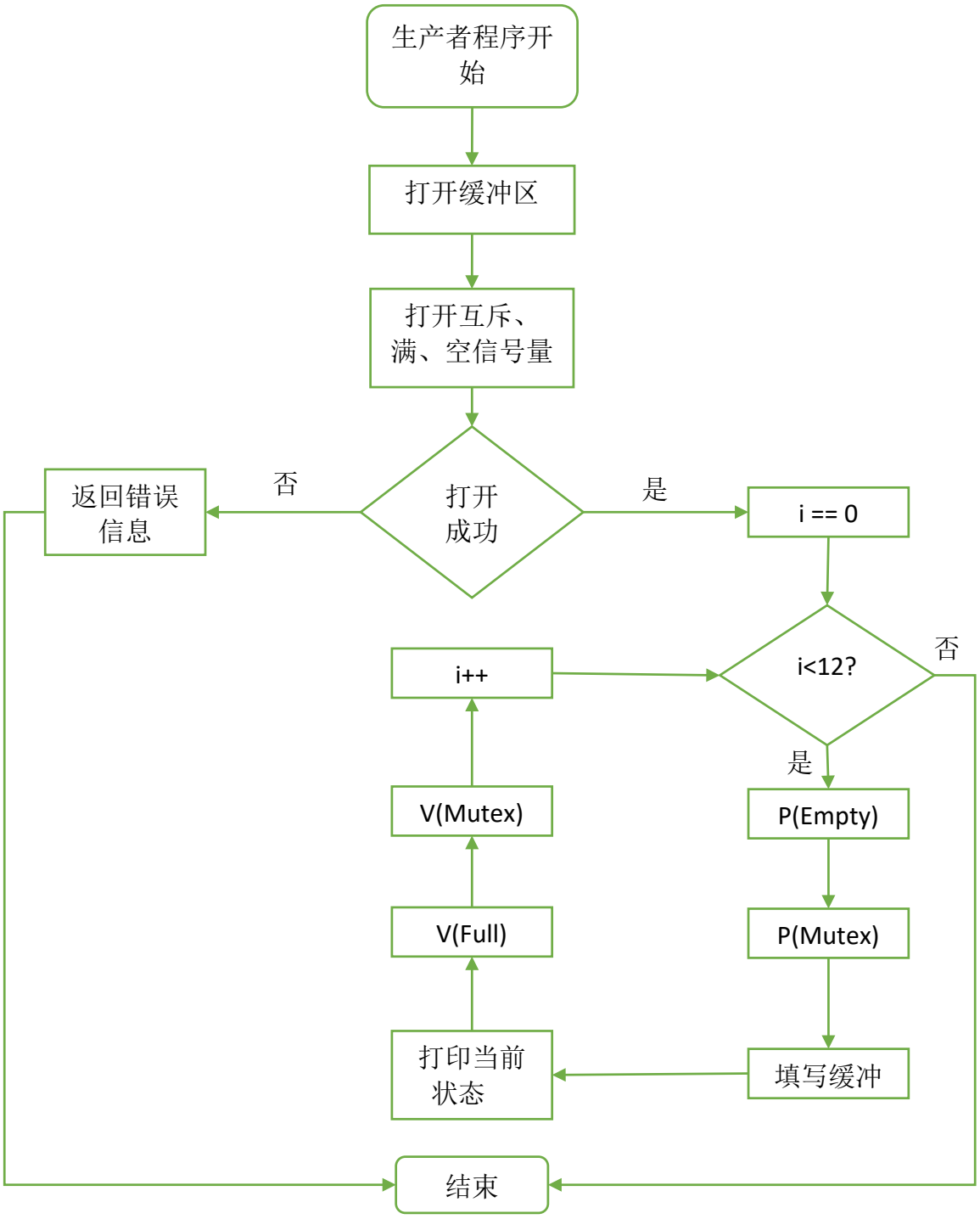
- 返回第三步，重复 8 次。

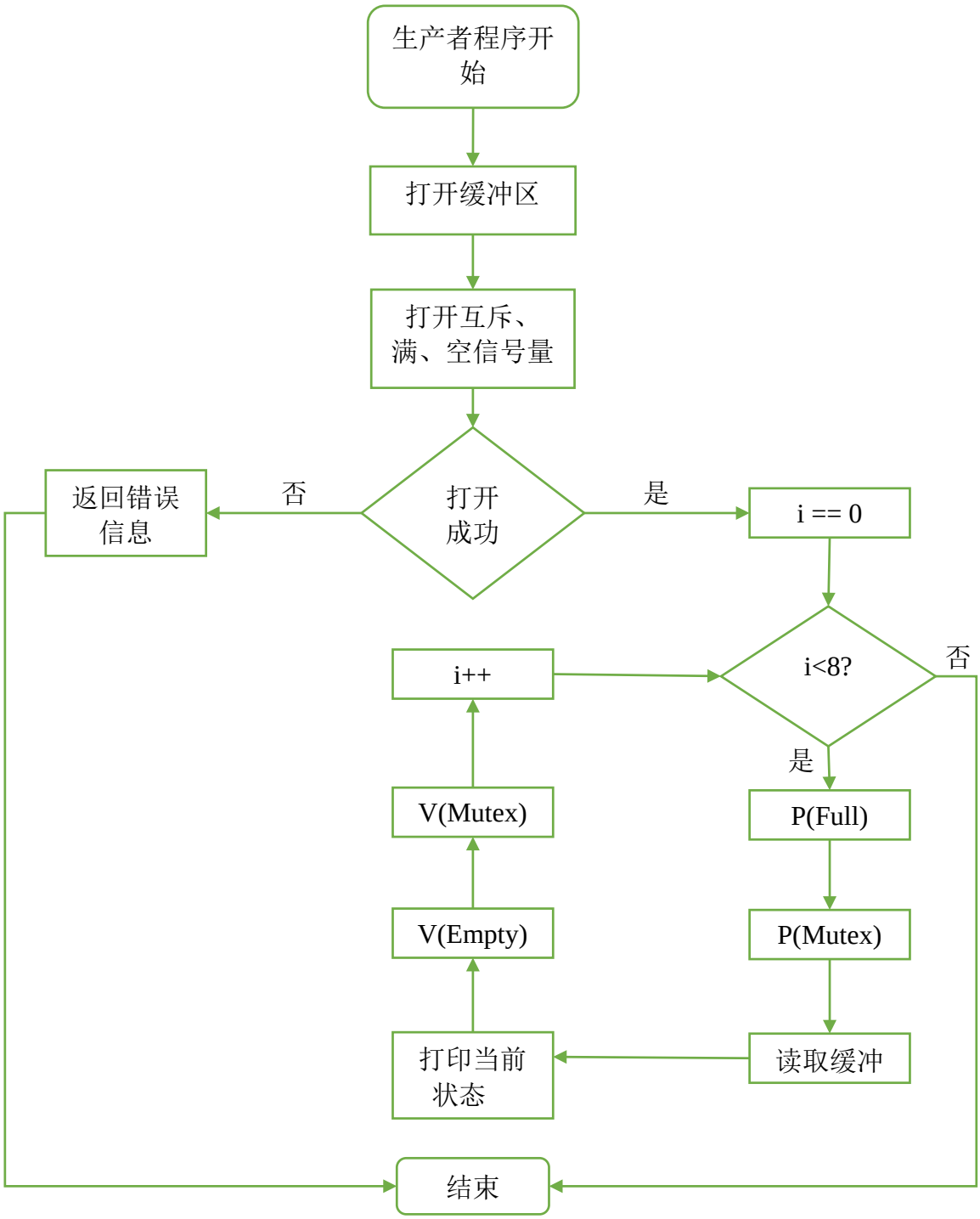
## 4. 2. 伪代码

```
semaphore Mutex = 1;
semaphore Fill = 0;
semaphore Empty = 3;
procedure producer()
{
    while (i<6)
    {
        P(Empty);
        P(Mutex);
        PutItemIntoBuffer(item);
        V(Full);
        V(Mutex);
    }
}
procedure consumer()
{
    while (i<4)
    {
    P(Full);
    P(Mutex);
    RemoveItemFromBuffer();
    V(Empty);
    V(mutex);
    }
}
```

## 4. 3. 程序流程图


主控程序
开始
建立缓冲
区
创建互
斥、空、
满信号量
创建成功?
否
返回错误
信息
是
创建生产
者进程
等待
结束?
创建消费
者进程
主控程序
结束


生产者程序开始
打开缓冲区
打开互斥、满、空信号量
打开成功
否
返回错误信息
是
i == 0
i<12?
否
是
i++
P(Empty)
V(Mutex)
P(Mutex)
V(Full)
填写缓冲
打印当前状态
结束




## 4.4. Windows 调用系统 API

#### 4.4.1. sharemen

```
struct sharemen
{
    int put;            //记录放数据位置
    int take;           //记录取数据位置
    int number[3];      //数据缓存去
};
```

#### 4.4.2. Process_INFORMATION

```
typedef struct_PROCESS_INFORMATION{
    //新创建进程的句柄
    HANDLE hProcess;
    //新创建进程的主线程的句柄
    HANDLE hThread;
    //新创建进程的标识
    DWORD dwProcessId;
    //新创建进程的主线程的标识
    DWORD dwThreadId;
}PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

#### 4.4.3. CreateFileMapping

创建新的文件映射内核对象

```
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, //□ □ □ □ □ □
    LPCTSTR lpName //□ □ □ □ □ □
);
```

#### 4.4.4. MapViewOfFlie

将一个文件映射对象映射到当前应用程序的地址空间。

```
LPVOID MapViewOfFile(
    HANDLE hFileMappingObject, //共享文件对象
    DWORD dwDesiredAccess, //文件共享属性
    DWORD dwFileOffsetHigh, //文件共享区的偏移地址的高32位
    DWORD dwFileOffsetLow, //文件共享区的偏移地址的低32位
    SIZE_T dwNumberOfBytesToMap //映射文件的字节数
);
```

#### 4.4.5. OpenFileMapping

```
HANDLE OpenFileMapping(
    DWORD dwDesiredAccess, //映射对象的文件数据的访问方式
    BOOL bInheritHandle, //返回句柄能否由当前进程启动的新进程继承
    LPCSTR lpName //要打开的文件映射对象名称
```

```
);
```

#### 4.4.6. CreateSemaphore

创建一个新的信号量。

```
HANDLE WINAPI CreateSemaphore(
        LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, //□ □ □ □
        LONG lInitialCount, //□ □ □ □ □ □ □
        LONG lMaximumCount, //□ □ □ □ □ □ □
        LPCTSTR lpName //□ □ □ □ □ □
);
```

#### 4.4.7. OpenSemaphore

为现有的一个已命名信号量对象创建一个新句柄。

```
HANDLE WINAPI OpenSemaphore(
        DWORD dwDesiredAccess, //□ □ □ □ □ □
        BOOL bInheritHandle, //□ □ □ □ □ □ □ □ □ □ □
        LPCTSTR lpName //□ □ □ □ □ □ □ □ □
);
```

#### 4.4.8. ReleaseSemaphore

用于对指定的信号量增加指定的值。

```
BOOL WINAPI ReleaseSemaphore(
        HANDLE hSemaphore, //所要操作的信号量对象的句柄
        LONG lReleaseCount, //这个信号量对象在当前基础上所要增加的值
        LPLONG lpPreviousCount //指向返回信号量上次值的变量的指针
);
```

### 4.5. Linux 调用系统 API

#### 4.5.1. semget

得到一个信号量集标识符或创建一个信号量集对象并返回信号量集标识符

```
int semget(
    key_t key, //□ □ □ □ □ □ □ □ □ □ □ □ □ 。
    int nsems, //□ □ □ □ □ □ □ □ □ □ □ □ □ □ □
    int semflg //□ □ □ □ □ □ □ □ □
);
```

#### 4.5.2. semctl

```
得到一个信号量集标识符或创建一个信号量集对象并返回信号量集标识符。
int semctl(
    int semid, //要操作的信号量
    int semnum, //信号量编号
    int cmd, //指定对信号量的操作
    union semun arg //
);
```

### 4.5.3. semget

```
得到一个共享内存标识符或创建一个共享内存对象并返回共享内存标识符

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

### 4.5.4. semop

```
操作一个或一组信号。

int semop(
    int semid, //□ □ □ □ □ □ □
    struct sembuf *sops, //□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
    unsigned nsops //□ □ □ □ □ □ □ □ □ □
);
struct sembuf {
    short semnum;
    /*□ □ □ □ □ □ □ □ □ □ □ □ □ ，0□ □ □ 1□ □ □ □ □ */
    short val;
    /*□ val>0□ □ V□ □ □ □ □ □ □ val ，□ □ □ □ □ □ □ □ □ □ □ □ */
    /*□ val<0□ □ P□ □ □ □ □ □ □ val ，□ (semval-val)<0（semval□ □ □ □ □ □ ），□ □ □ □ □ □
□ ，□ □ □ □ □ □ ；□ □ □ IPC_NOWAIT□ □ □ □ ，□ □ □ □ □ □ EAGAIN□ □ */
    /*□ val==0□ □ □ □ □ □ □ □ □ 0 ，□ □ □ □ □ □ □ □ □ □ ，□ □ □ □ □ □ 0 ；□ □ □
IPC_NOWAIT，□ □ □ □ □ □ ，□ □ □ □ EAGAIN□ □ */
    short flag;
    /*0 □ □ □ □ □ □ □ □ □ □ */
    /*IPC_NOWAIT□ □ □ □ □ □ □ □ □ */
    /*SEM_UNDO □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ UNDO□ □ ，□ □ □ □ □ □ □ ，□ □ □
□ □ □ □ □ UNDO□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ */
};
```

### 4.5.5. shmat

连接共享内存标识符为 shmid 的共享内存，连接成功后把共享内存区对象映射到调用进程的地址空间，随后可像本地空间一样访问.

```
void *shmat(
    int shmid, //共享内存标识符
    const void *shmaddr, //指定共享内存出现在进程内存地址的什么位置
    int shmflg //SHM_RDONLY：为只读模式，其他为读写模式
);
```

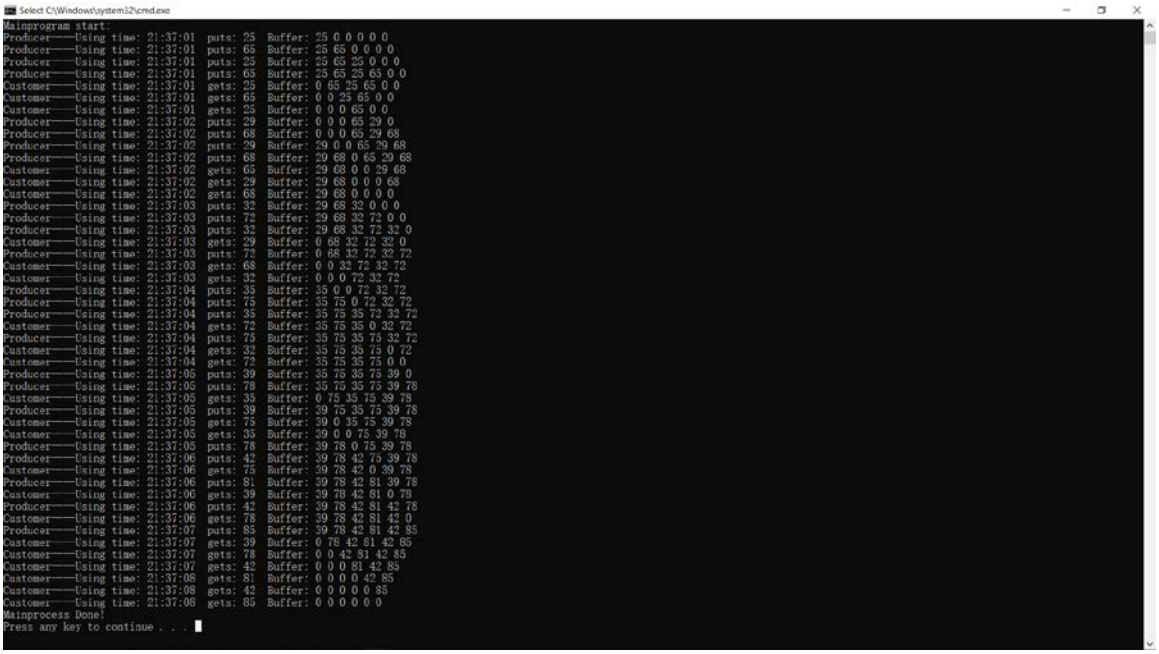
### 4.5.6. shmdt

与 shmat 函数相反，是用来断开与共享内存附加点的地址，禁止本进程访问此片共享内存

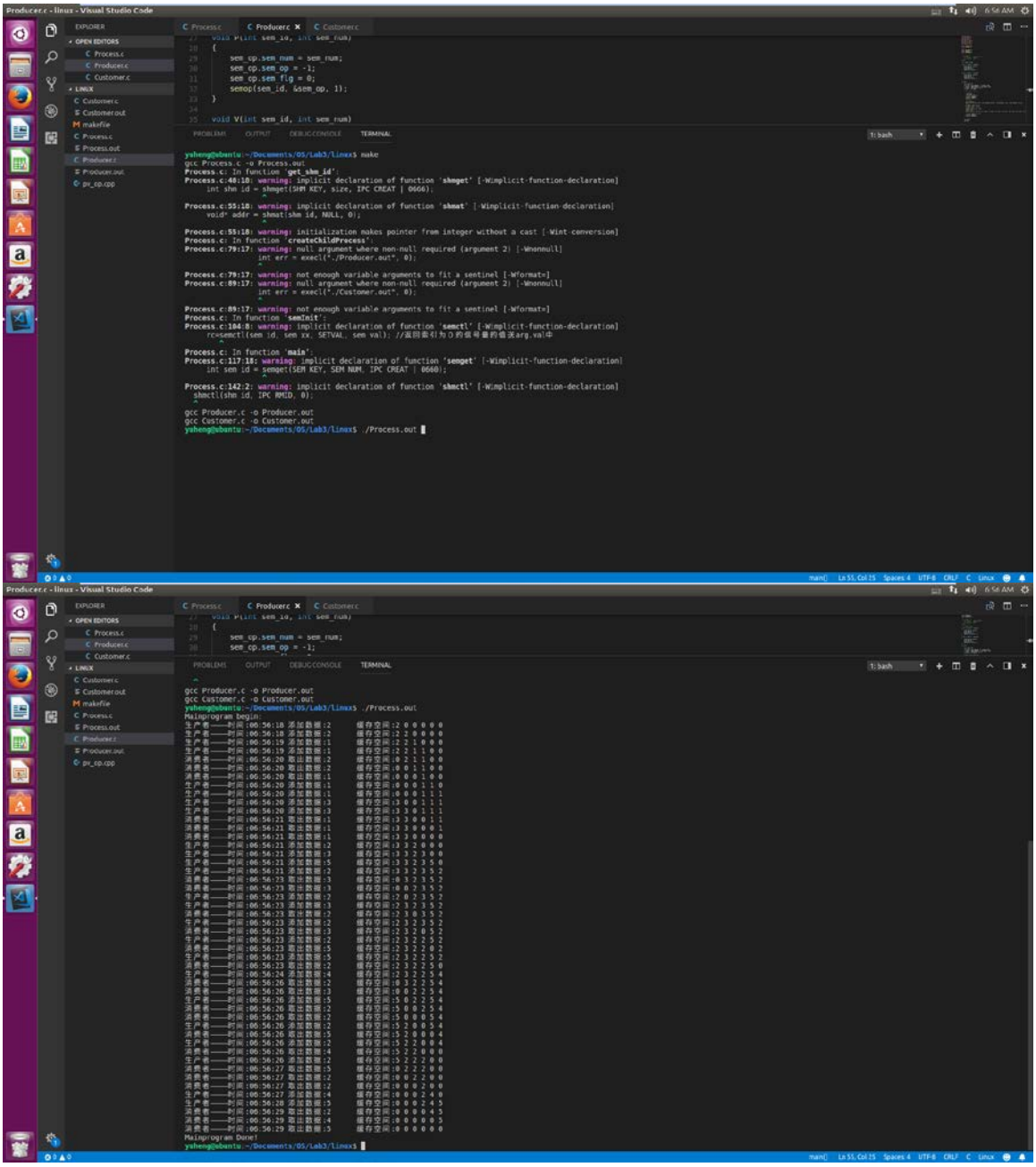
```
int shmdt(
    const void *shmaddr //连接的共享内存的起始地址
)
```

# 五. 实验结果

## 5.1. Windows 实验结果




## 5.2. Linux 实验结果



## 六. 心得体会

通过本次实验，基掌握了通过本次实验，基掌握了 Windows 和 Linux 中通过互斥体和信号量实现控制进程同步和互斥的方法，并初步学习并实现了共享主存，为多进程通信和共享数据提供了更加快捷方便的方法。

体会 1：在子进程中，应当先申请信号量，再申请互斥体，顺序不能改变。否则会发生死锁。

体会 2：在申请信号量和互斥体成功之前，子程序并未开始真正的执行，第一次写程序时，将 printf（“生产者——”);放在了申请之前，造成输出结果混乱。

体会 3：加深了 int 型数据占四个字节的知识点，第一次写程序时忘了每次读取一个新的 int 型数据时，地址需要加 4，导致出现错误。