操作系统课程设计实验报告

实验三 生产者与消费者进程模拟

指导教师：陆慧梅老师

班　　级：07111507

学　　号：1120151880

姓　　名：廖汉龙

邮　　箱：liaohanlong@outlook.com

2018 年 3 月 20 日

一、实验目的

1. 掌握 Windows、Linux 下利用信号量机制实现进程的同步，实践操作系统中经典的生产者与消费者的问题。

2. 掌握信号量、互斥量、缓冲区的建立及相关操作。

3. 掌握 Windows、Linux 不同环境下实现内存共享的方法，创建和使用共享内存区，使多个进程可以使用同一块数据。

二、实验内容

1. 总体要求：

在 Windows 和 Linux 系统下通过进程编程模拟生产者消费者算法。

设计一个大小为 3 的缓冲区。

2. 生产者要求：

创建 2 个生产者；

每个生产者随机等待一段时间，向缓冲区添加一个大写字母，若缓冲区已满，等待消费者取走字母后再添加；

重复 6 次；

3. 消费者要求：

创建 3 个消费者；

每个消费者随机等待一段时间，从缓冲区读取字母；

若缓冲区为空，等待生产者添加字母后再读取；

重复 4 次；

4. 打印内容要求；

生产者打印：生产者本次写入缓冲区字母；

消费者打印：消费者本次取走的字母；

需打印缓冲区内容；

按先生产的商品先消费原则；

三、实验环境及配置方法

1. Windows 环境：

操作系统：Windows 10 家庭中文版

编译环境：Visual Studio Code

gcc 4.9.2

2. Linux 环境：

操作系统：Ubuntu 16.04 LTS

编译环境：GCC 5.4.0

编写代码软件：Visual Studio Code

四、实验方法与步骤

1.实验原理

生产者进程与消费者进程存在同步与互斥关系：进程互斥地访问缓冲区，对缓冲区内容进行读写操作；而各个进程之间是同步运行的关系。根据操作系统课程的学习内容，需要设定以下信号量进行进程间的操作：

MUTEX：互斥信号量，控制生产者进程与消费者进程互斥的访问缓冲区，初始值为 1；

EMPTY：表示缓冲区中空闲区的大小，初始值为 3；

FULL：表示缓冲区中非空闲区的大小，初始值为 0

伪代码如下：

```
1). 生产者进程：
Process:
{
    P(EMPTY);//检查缓冲区是否有空闲区域
    P(MUTEX);//申请互斥访问缓冲区
    添加字母操作；
    V(FULL);//增加缓冲区非空闲区的数量
    V(MUTEX);//释放缓冲区
}
2).消费者进程：
Consumer:
{
    P(FULL);//检查缓冲区中是否有字母
    P(MUTEX);//申请互斥访问缓冲区
    读取字母操作；
    V(EMPTY);//增加缓冲区空间区的数量
    V(MUTEX);//释放缓冲区
}
```

各个过程的流程图如下：

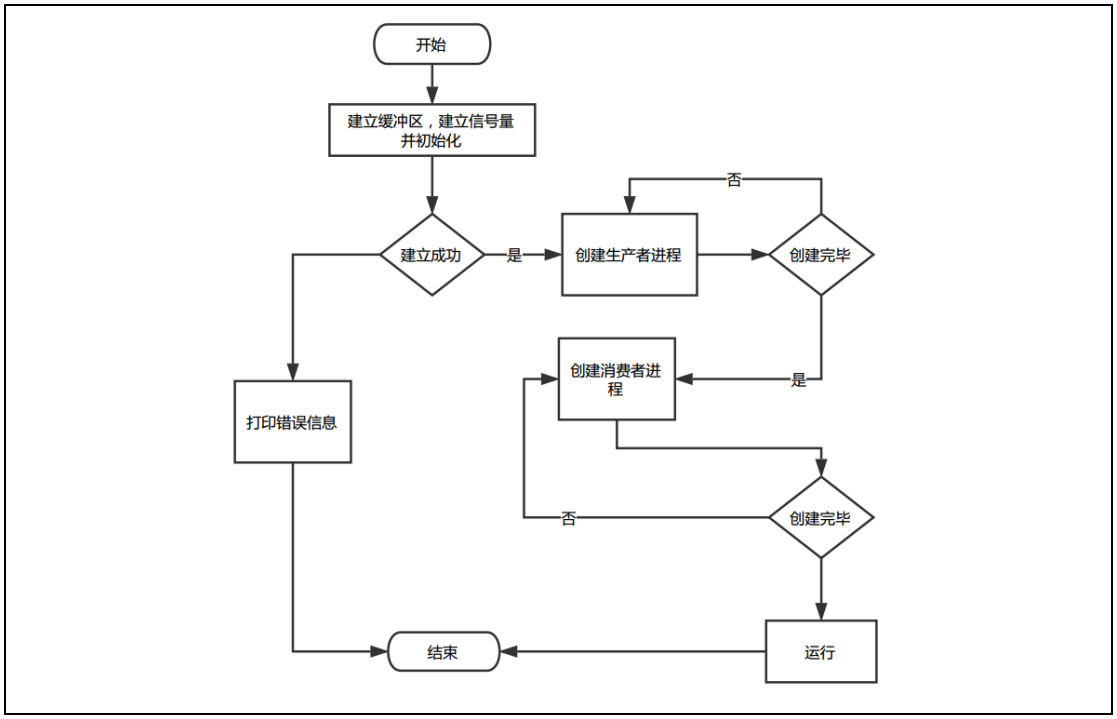


图-主控过程

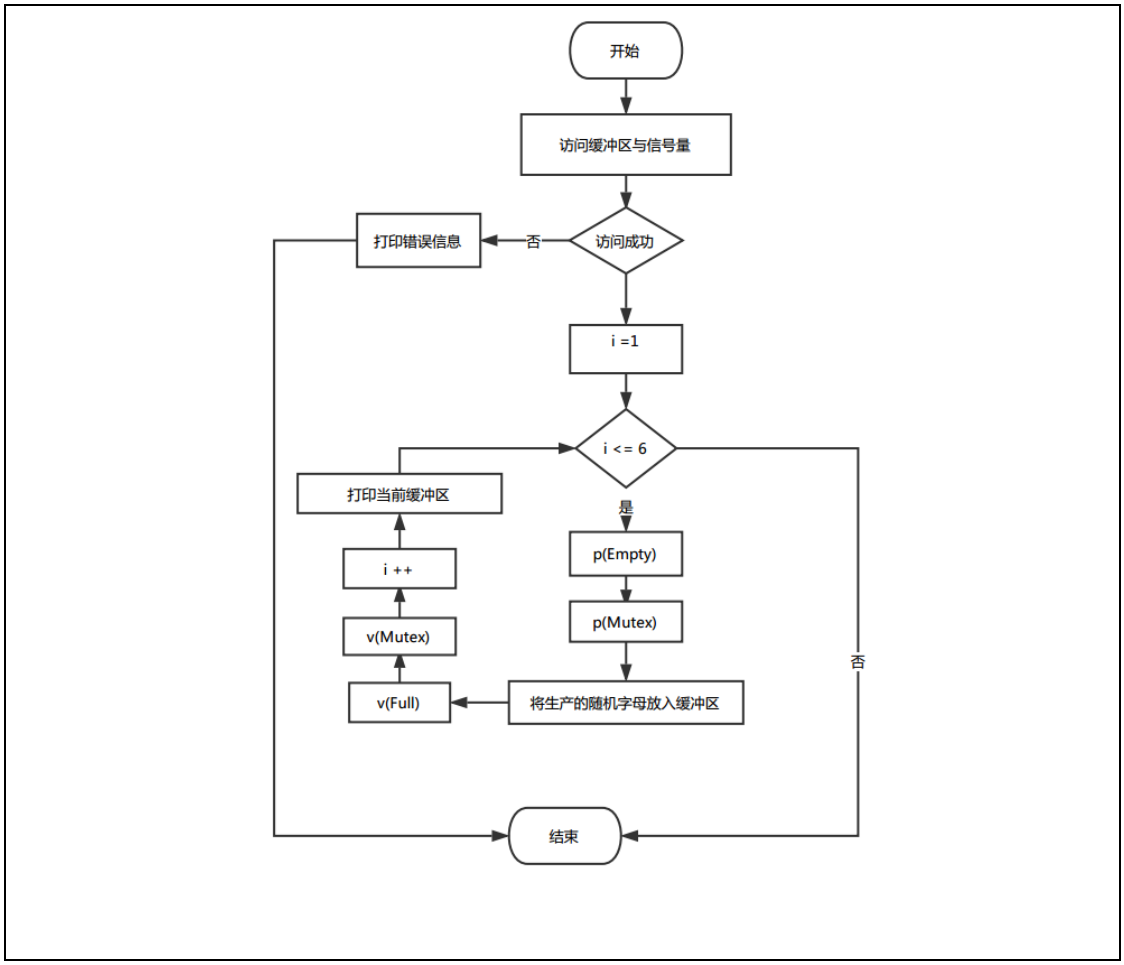


图-生产者过程

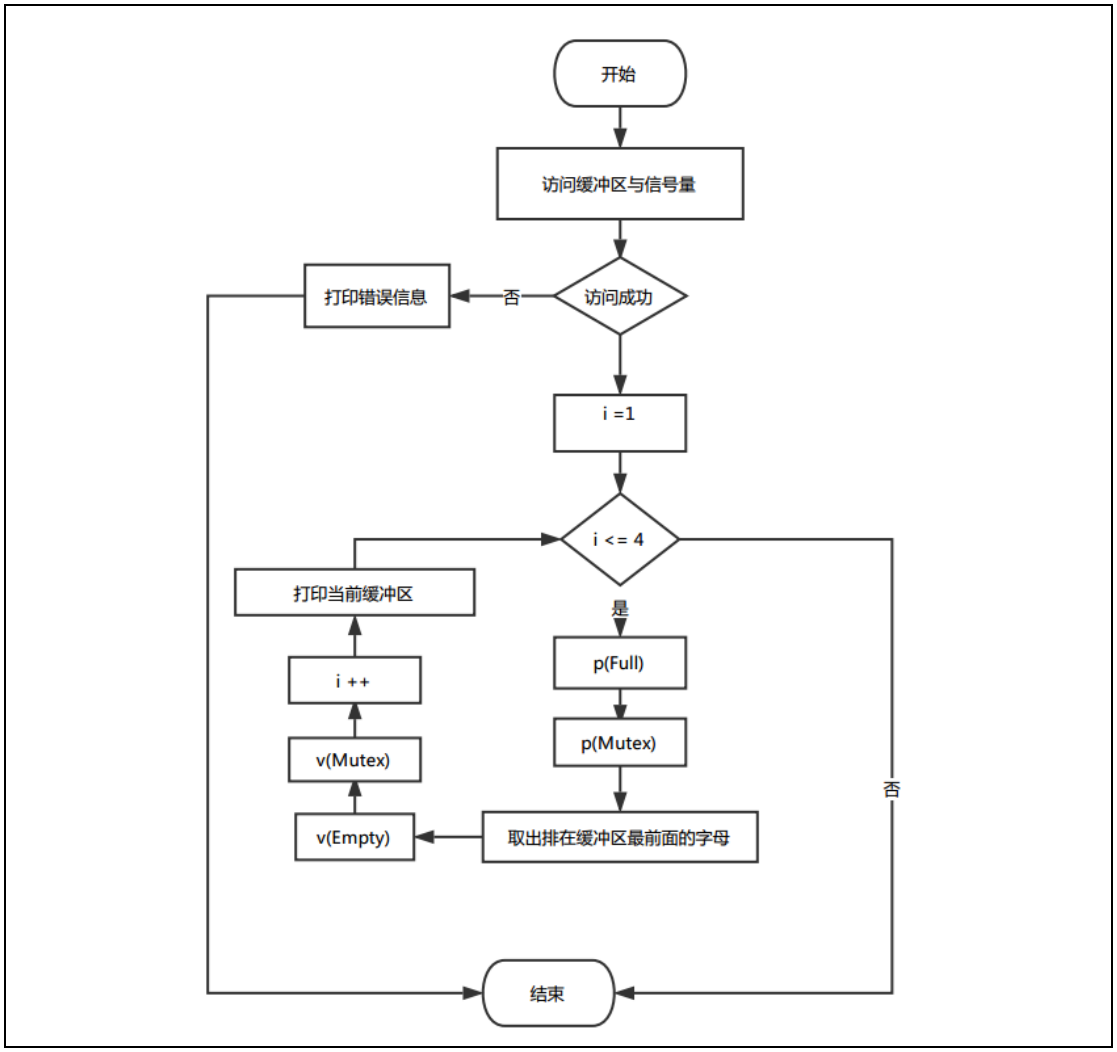


图-消费者过程

2.window 系统与 Linux 下的 API 介绍

（1）windows 系统下的 API

- CreateFileMapping()，打开或创建一个文件映射对象。

```
HANDLE WINAPI CreateFileMapping(
    HANDLE                hFile,              //为指定欲在其中映射创建映射的一个文件句柄，
                                              //也就是CreateFile()函数返回的句柄
    LPSECURITY_ATTRIBUTES lpAttributes,       //为指向SECURITY_ATTRIBUTES结构的指针
    DWORD                 flProtect,          //为保护参数，可取的值为PAGE_READONLY，
                                              //PAGE_READWEITE或PAGE_WRITECOPY
    DWORD                 dwMaximumSizeHigh,//为文件映射对象的最大长度（高32位）
    DWORD                 dwMaximumSizeLow, //为文件映射对象的最小长度（低32位），如果和
                                              //dwMaximumSizeHigh都为0，表示磁盘实际长度
    LPCTSTR               lpName              //为指定文件映射对象的名字
);
```

若函数调用成功，返回新建文件映射对象句柄。如果函数调用失败，返回值为 0，并且设置错误信息，错误信息可以通过 GetLastError 返回错误原因。即使函数成功，但倘若返回的句柄属于一个现成的文件映射对象，那么 GetLastError 也会设置成 ERROTR_ALREADY_EXISTS.在这种情况下，文件的映射长度就是先有对象的长度，而不是这个函数指定的尺寸。

● OpenFileMapping（）打开一个已经存在的文件映射对象

```
HANDLE WINAPI OpenFileMapping(
    dwDesiredAccess,    //为指定文件映射对象的存取访问方式
    bInheritHandle,     //为指定继承标记，用于明确返回的句柄是否在
                        //进程创建期间由其他进程继承。若设置为true,
                        //则表明新创建进程继承句柄
    LPCTSTR lpName      //为指定要打开的文件映射对象的名称。
);
```

如果函数调用成功，则返回要打开的文件映射对象的句柄；否则返回值为 0，则设置错误信息，并通过 GetLastError 返回错误原因。

● MapViewofFile() 将文件对象映射到进程的地址空间

```
LPVOID WINAPI MapViewOfFile(
    hFileMappingObject,  //为文件映射对象的句柄
    dwDesiredAccess,     //为指定对文件的访问权限
    dwFileOffsetHigh,    //为文件内映射起点的高32位地址
    dwFileOffsetLow,     //为文件内映射低32位地址
    dwNumberOfBytesToMap //为文件要要映射的字节数；0表示整个文件
);
```

如果调用成功，返回文件映射在内存中的其实地址，否则返回值为 0。且会设置错误信息，并通过 GetLastError 返回错误原因，MapViewofFileEX()函数允许指定一个基本地址来进行映射。

● UnmapView() 解除进程对于一个文件映射对象的映射

```
BOOL WINAPI UnmapViewOfFile(
    LPCVOID lpBaseAddress  //为指定要解除映射文件的一个文件映射的
                           //基准地址，该地址是用MapViewofFile()函数来获得的
);
```

如果函数调用成功，返回值为非 0，否则返回值为 0，切会设置错误信息，并通过 GetlastError 返回错误信息

● createSemphore() 创建一个信号量对象

```
HANDLE WINAPI CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, //为指定了应用于信号量的安全属性，包括访问权
                                                 //等，如果是NULL，就表示使用默认属性
    LONG                  lInitialCount,         //为信号量的初始值
    LONG                  lMaximumCount,         //为信号量的最大值
    LPCTSTR               lpName                 //为信号量的名称（一个字符串）
);
```

如果函数调用成功，则会返回一个信号量对象的句柄，否则返回 NULL.不论是哪种情况，GetLastError()函数都会返回一个合理的结果。

● ReleaseSemaphore() 释放信号量

```
BOOL WINAPI ReleaseSemaphore(
    HANDLE hSemaphore,      //为信号量对象的句柄
    LONG   lReleaseCount,   //为信号量技术要增加的值
    LPLONG lpPreviousCount  //为返回信号量计数增加前的值
);
```

如果函数成功调用，则返回 true， 否则返回 false，失败时可以通过调用 GetLastError（）函数获得原因。

- OpenSemaphore() 打开一个信号量

```
HANDLE WINAPI OpenSemaphore(
    dwDesiredAccess,      //为指定信号量对象期望的访问形式
    bInheritHandle,       //为指定的继承标识
    LPCTSTR lpName        //为信号量的名称（一个字符串）
);
```

如果调用成功，返回信号量对象的句柄，否则，返回为 NULL。一旦不再需要，一定要使用 CloseHandle 关闭信号量句柄。如果对象的所有句柄都已经关闭那么该对象就会被删除。

（2）Linux 系统的 API

- semget() 创建一个信号量集合

```
semget(
    key_t key,    //为用户进程指定信号量集合的关键字
    int nsems,    //为信号量集合中的信号量数
    int semflg    //为规定的创建和打开标志
);
```

返回值为信号量集合的标志号，出错返回-1;

- semop() 对信号量进程 P/V 操作

```
semop(
    int semid,                //是进程调用semget后返回的信号量集合的标识符
    struct sembuf * sops,     //是用户提供的操作信号量的模板数组的指针
    unsigned nsops            //为一次操作的数组 sembuf 中的元素数
);
```

其操作命令由用户提供的信号量操作模板（sembuf）定义，改模板的结构如下

```
struct sembuf{
    ushort sem_num;   //信号量集合中要操作的信号量的索引
    short sem_op;     //信号量的操作值
    short sem_fly;    //访问标志
};
```

正常返回值为 0，错误返回值为-1

- semctl() 对信号量执行控制操作

```
int semctl(
    int semid,      //信号量集合的标志
    int semnum,     //信号量的索引
    int cmd,        //为要执行的操作命令
    union semun arg //用于设置或返回信号量信息的参数
);
```

其中，semun 定义如下：

```
union semun{
    int val;                  //SETVAL 的值
    struct semid_ds *buf;     //为 IPC_STAT 和 IPC_SET 的缓冲区
    ushort * array;           //为获得GETALL 和设置SETALL 信号量值的数组
}arg;
```

- shmget() 申请一个共享内存区

```
shmget(
    key,    //为共享内存区的关键字
    size,   //创建或者打开的标志
    shmflg  //共享内存区域字节长度
);
```

成功时，为共享内存区域标识区的标识，不成功返回-1，errno 存储错误原因

- shmat() 将共享段附加到申请通信的进程空间

```
shmat(
    shmid,      //是进程调用shmget后返回的共享段标识
    shmadd,     //给出了应附加到进程虚空间的地址，若为0,则将该共享段附加到系统选
                //择的进程的第一个可用地址之后，若为非0，则附加到指定的地址上，
                //且shmflg指定了SHM_RND标志则将其附加到按页取值的地址上，通常shmadd
                //的值为0
    shmflg      //shmflg为允许对共享段的访问方式
);
```

- shmdt() 将共享段与进程之间接触连接

```
shmat(
    shmaddr    //共享段在进程地址空间的虚地址
);
```

函数的返回值为 0

- shmctl() 对共享内存区执行控制操作

```
shctl(
    int shmid,              //表示共享区的标识
    int cmd,                //要执行的命令
    struct shmid_ds *buf
);
```

若成功，返回 0，失败返回-1.

3.代码实现过程

(1)windows 环境

1). 初始化缓冲区

定义缓冲区结构，采用循环队列实现先生产的商品先消费的原则。
结构体定义代码截图：

```
//共享缓冲区结构
struct mybuffer
{
    char letter[LETTER_NUM];
    int head;
    int tail;
    int is_empty;
    int index;              // 步骤标记
    HANDLE semEmpty;        //空信号量句柄
    HANDLE semFull;         //满信号量句柄
    HANDLE semMutex;        //互斥访问信号量
};
```

在此处，我没有像网上给的写法，把信号量和共享内存的区域分开定义，而是统一定义在了一个结构体里边，因为信号量也是所有进程共享的资源，这样构

造，使得后续的代码简洁许多。

对缓冲区进行初始化，创建共享内存区域，使用函数 CreateFileMapping()来创建一个共享的文件数据句柄，使用 MapViewOfFile()来获取共享的内存地址

```
/**
 * 创建一个共享文件映射对象
 * 返回共享文件映射对象的句柄
 */
HANDLE MakeSharedFile()
{
    // 创建文件映射对象
    // INVALID_HANDLE_VALUE 表示 0xFFFFFFFF 表示无效的句柄，
    // 此处可以创建一个进程间共享的内存映射对象

    HANDLE hMapping = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, sizeof(struct mybuffer), SHM_NAME)

    if (hMapping != INVALID_HANDLE_VALUE)
    {
        // 将文件对象映射到进程的地址空间
        // 第一个参数为映射对象的句柄，第二个参数为制定文件的访问权限FILE_MAP_ALL_ACCESS表示映射
        // 文件可读可写
        // 第三个参数表示文件映射起点的高32位地址，第四个参数为低32位地址
        // 第四个参数表示文件中要映射的字节数，0表示映射整个文件
        // 如果调用成功，返回值为在文件在内存中的其实地址

        LPVOID pData = MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);
        if (pData != NULL)
        {
            ZeroMemory(pData, sizeof(struct mybuffer));
        }
        //关闭文件视图
        UnmapViewOfFile(pData);
    }
    return (hMapping);
}
```

对信号量的初始化在主进程中实现，当当前为父进程的时候，则会对信号量进程初始化。

分别创建互斥信号量 Mutex，表示缓冲区空闲区数量的信号量 Empty，表示缓冲区非空闲区的信号量 Full。

```
if (pFile != NULL)
        {
            //初始化共享内存区域
            struct mybuffer * shmp = (struct mybuffer *)(pFile);
            shmp->head = 0;
            shmp->tail = 0;
            shmp->index = 0;

            // 创建一个信号量对象
            // 第一个参数表示安全属性
            // 第二个参数为信号量的初始值，必须大于或者等于0
            // 第三个参数为信号量的最大值
            // 第四个参数为信号量的名称

            shmp->semEmpty = CreateSemaphore(NULL, LETTER_NUM, LETTER_NUM, "SEM_EMPTY")
            shmp->semFull = CreateSemaphore(NULL, 0, LETTER_NUM, "SEM_FULL");
            shmp->semMutex = CreateSemaphore(NULL, 1, 1, "SEM_MUTEX");

            UnmapViewOfFile(pFile);
            pFile = NULL;
        }
```

2) 创建子进程

通过自定义一个 StartClone 函数，创建子进程

```
//进程克隆（通过参数传递进程的序列号）
void StartClone(int nCloneID)
{
    char szFilename[MAX_PATH];
    char szCmdLine[MAX_PATH];
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    GetModuleFileName(NULL, szFilename, MAX_PATH);
    sprintf(szCmdLine, "\"%s\" %d", szFilename, nCloneID);
    memset(&si, 0, sizeof(si));
    si.cb = sizeof(si);
    //创建子进程
    BOOL bCreateOK = CreateProcess(szFilename, szCmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
    hs[nCloneID] = pi.hProcess;
    return;
}
```

这个过程在父进程中使用一个循环实现一次性创建所有的子进程，包括生产者与消费者

```
while (pindex <= PROCESS_NUM)
        {
            StartClone(pindex++);
        }
        //等待子进程完成
        for (k = 1; k < PROCESS_NUM + 1; k++)
        {
            WaitForSingleObject(hs[k], INFINITE);
            CloseHandle(hs[k]);
        }
```

3) 生产者进程实现

生产者进程的主要工作流程：

申请一个空缓冲区->申请对于缓冲区的操作->在缓冲区中写入字母->释放缓冲区->增加缓冲区中非空闲区的数量。

在这核心操作的过程前后，关键是对信号量的操作，可以发现，在网上与学长的一些代码中非常奇怪的并没有考虑到互斥信号量 Mutex 的作用，经过实验发现，没有考虑互斥信号量在 windows 系统的小数量级别的实验中不会出现问题，但是在大数量级别的时候或者在 Linux 系统下，则会出现明显的错误。

由于在开始时候的缓冲区的重新定义，所以此处对于缓冲区的操作则会简单许多。此处关键代码截图如下：

```
else if (nClone >= ID_P_FROM && nClone <= ID_P_TO)
    {
        //printf("Producer %d process starts.\n", nClone - ID_M);
        //映射视图
        HANDLE hFileMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, SHM_NAME);
        LPVOID pFile = MapViewOfFile(hFileMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);

        if (pFile != NULL)
        {
            struct mybuffer * shmp = (struct mybuffer *)(pFile);

            HANDLE semEmpty = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SEM_EMPTY")
            HANDLE semFull = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SEM_FULL");
            HANDLE semMutex = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SEM_MUTEX")

            for (i = 0; i < WORKS_P; i++)
            {
                Sleep(get_random());

                WaitForSingleObject(semEmpty, INFINITE);  // 信号量操作
                WaitForSingleObject(semMutex, INFINITE);
                //生产者向缓冲区投入字母
                shmp->index++;
                shmp->letter[shmp->tail] = lt = get_letter();
                shmp->tail = (shmp->tail + 1) % LETTER_NUM;
                shmp->is_empty = 0;
                GetLocalTime(&systime);

                printInfo(systime, shmp, nClone, lt, 0);

                ReleaseSemaphore(semFull, 1, NULL);    //信号量操作
                ReleaseSemaphore(semMutex, 1, NULL);
            }

            UnmapViewOfFile(pFile);
            pFile = NULL;
        }
        else
        {
```

```
            printf("Error on OpenFileMapping.\n");
        }
        CloseHandle(hFileMapping);
        //printf("Producer %d process ends.\n", nClone - ID_M);
    }
```

4) 消费者进程实现

消费者的进程实现基本和生产者相似，代码可以直接拷贝修改，除了对于信号量的操作意外，就是对于缓冲区的操作，操作大致如下：

消费者进程的主要工作流程：申请一个缓冲区的非空闲区->申请对于缓冲区的操作->从缓冲区中读取字母->释放缓冲区->增加一个缓冲区中空闲区的数量。关键代码截图如下：

```
//对于消费者进程
    else if (nClone >= ID_C_FROM && nClone <= ID_C_TO)
    {
        //printf("Consumer %d process starts.\n", nClone - ID_P_TO);
        //映射视图
        HANDLE hFileMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, SHM_NAME);
        LPVOID pFile = MapViewOfFile(hFileMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);
        if (pFile != NULL)
        {
            struct mybuffer * shmp = (struct mybuffer *)(pFile);

            HANDLE semEmpty = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SEM_EMPTY");
            HANDLE semFull = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SEM_FULL");
            HANDLE semMutex = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SEM_MUTEX");

            for (i = 0; i < WORKS_C; i++)
            {
                Sleep(get_random());
                WaitForSingleObject(semFull, INFINITE);   //信号量操作
                WaitForSingleObject(semMutex, INFINITE);


                //缓冲区操作
                shmp->index++;
                lt = shmp->letter[shmp->head];
                shmp->head = (shmp->head + 1) % LETTER_NUM;
                shmp->is_empty = (shmp->head == shmp->tail);
                GetLocalTime(&systime);

                printInfo(systime, shmp, nClone, lt, 1);

                ReleaseSemaphore(semEmpty, 1, NULL);    //信号量操作
                ReleaseSemaphore(semMutex, 1, NULL);
            }
            UnmapViewOfFile(pFile);
            pFile = NULL;
        }
```

6) 输出函数

为了对于缓冲区以及当前的进程的变化的过程有更加直观的认识，定义了一个输出函数。

```
void printInfo(SYSTEMTIME systime, struct mybuffer * shmp , int nClone, char lt, int type){
    printf("[%02d]\t", shmp->index);
    printf("Process ID: %d\t", GetCurrentProcessId());
    printf("Time: %02d:%02d:%02d\n", systime.wHour, systime.wMinute, systime.wSecond);
    if(type == 0) printf("Producer %d puts '%c'.\n", nClone - ID_M, lt);
    else{ printf("Consumer %d gets '%c'.\n", nClone - ID_P_TO, lt);}


    printf("+---+---+---+---+\n");

    int n, j;
    for (n = 0, j = (shmp->tail - 1 >= shmp->head) ? (shmp->tail - 1) : (shmp->tail - 1 + LETTER_NUM); !(shmp->is_empty) && j >= shmp->head; j-
    {
    n++;
    printf("|");
    printf("%3c", shmp->letter[j % LETTER_NUM]);
    }
    for(int k = n + 1; k <= 3; k++ ){
        printf("|   ");
    }
    printf("|\n");
    printf("+---+---+---+---+\n");

    printf("----------------------------------------------------------------------\n");
}
```

7) 编译运行代码，观察实验结果。

(2)Linux 系统环境

1)初始化缓冲区

定义缓冲区结构，采用循环队列实现先生产的商品先消费的原则。

结构体定义代码截图：

```c
//缓冲区结构（循环队列）
struct mybuffer
{
    char letter[LETTER_NUM];
    int head;
    int tail;
    int is_empty;
    int index;
};
```

初始化共享内存区域，利用函数 shmget()申请共享区域，使用函数 shmat()得到映射到共享区域的指针，并对缓冲区进行初始化。代码截图如下：

```c
/*  建立共享内存区，如果 key == IPC_PRIVATE时，表示总是会创建一个新的共享内核对象
    *   第二个参数表示共享内存区域的长度，第三个参数表示共享内存区域的创建或者打开标志
    *   表示对共享内存区域的特殊要求
    *   函数返回共享段标识
    */
    if ((shm_id = shmget(IPC_PRIVATE, BUF_LENGTH, SHM_MODE)) < 0)
    {
        printf("Error on shmget.\n");
        exit(1);
    }

    /*  shamt()API将共享内存空间映射到了当前进程的虚拟空间，返回的是对应的内存的地址
    *   第二个参数为0表示附加到的地址为进程虚拟空间的
    *   第一个可用地址空间，第三个参数则为允许对共享段的访问方式
    *   shmper 是一个缓冲区类型的指针
    */
    if ((shmptr = (mybuffer*)shmat(shm_id, 0, 0)) == (void *)-1)
    {
        printf("Error on shmat.\n");
        exit(1);
    }

    //  初始化
    shmptr->head = 0;
    shmptr->tail = 0;
    shmptr->is_empty = 1;
    shmptr->index = 0;
```

2）初始化信号量

利用函数 semget()创建信号量集合，利用函数 semctl()分别创建互斥信号量 mutex、空信号量 empty、满信号量 full。

```c
/*  创建了一个信号量集合，关键字为SEM_ALL_KEY 1234，
*   信号量集合中的信号量数目为2，第三个参数为创
*   建或者打开的标志
*   返回值为信号量集合的标志号，出错返回1
*/
sem_id = semget(SEM_ALL_KEY, 3, IPC_CREAT | 0660);
//创建成功
if (sem_id >= 0)
{
    printf("Main process starts. Semaphore created.\n");
}
/*  对信号量执行控制操作
*   信号量标志为sem_id,信号量索引为SEM_EMPTY与SEM_FULL
*   SETVAL 为需要执行的操作命令，为 SETVAL 时表示设置信号量为
*   一个初始值，为arg.val(第4个参数中的一个值)，第四个参数一般是一个semun的
*   union,此处直接进行了赋值，并且初始化给了信号量
*   在上一个函数中创建了的信号量为2，此处第二个参数分别对其
*   索引值为0和1的两个信号量进行初始化
*/
semctl(sem_id, SEM_EMPTY, SETVAL, LETTER_NUM);
semctl(sem_id, SEM_FULL, SETVAL, 0);
semctl(sem_id, SEM_MUTEX, SETVAL, 1);
```

3）定义 P 操作与 V 操作

P 操作使信号量减一，代码截图如下：

```
//P操作
void p(int sem_id, int sem_num)
{
    struct sembuf xx;
    xx.sem_num = sem_num;   //要操作的信号量的索引
    xx.sem_op = -1;         //操作值
    xx.sem_flg = 0;         //访问标志
    /*  semop() 进程对信号集合的一个或者多个信号量执行P/V操作
    *   sem_id 为信号量的标识符，xx是用户提供的模板数组指针
    *   1为数组中的元素数量
    */
    semop(sem_id, &xx, 1);
}
```

V 操作使信号量加一，代码截图如下：

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

4)创建子进程

创建生产者并且进程对于缓冲区的操作如下：

生产者进程的主要工作流程：申请一个空缓冲区->申请对于缓冲区的操作->在缓冲区中写入字母->释放缓冲区->增加缓冲区中非空闲区的数量。

与 Window 操作系统下一致的是对于信号量的操作，关键代码截图如下：

```
while ((num_p++) < NEED_P)
    {
        if ((pid_p = fork()) < 0)
        {
            printf("Error on fork.\n");
            exit(1);
        }
        //如果是子进程，开始创建生产者
        if (pid_p == 0)
        {
            if ((shmptr = (mybuffer*)shmat(shm_id, 0, 0)) == (void *)-1)
            {
                printf("Error on shmat.\n");
                exit(1);
            }

            for (i = 0; i < WORKS_P; i++)
            {
                p(sem_id, SEM_EMPTY);   // 信号量操作
                p(sem_id, SEM_MUTEX);
                sleep(get_random());

                shmptr->letter[shmptr->tail] = lt = get_letter();   //对于缓冲区的操作
                shmptr->tail = (shmptr->tail + 1) % LETTER_NUM;
                shmptr->is_empty = 0;
                shmptr->index++;
                pid_t pid = getpid();
                now = time(NULL);

                printInfo(now, shmptr, num_p, lt, pid, 0);

                fflush(stdout);
                v(sem_id, SEM_FULL);  //信号量操作
                v(sem_id, SEM_MUTEX);
            }
            shmdt(shmptr);
            exit(0);
        }
    }
```

5)消费者进程实现

消费者进程的主要工作流程：申请一个缓冲区的非空闲区->申请对于缓冲区的操作->从缓冲区中读取字母->释放缓冲区->增加一个缓冲区中空闲区的数量。关键代码截图如下：

```
while (num_c++ < NEED_C)
    {
        if ((pid_c = fork()) < 0)
        {
            printf("Error on fork.\n");
            exit(1);
        }
        //如果是子进程，开始创建消费者
        if (pid_c == 0)
        {
            if ((shmptr = (mybuffer*)shmat(shm_id, 0, 0)) == (void *)-1)
            {
                printf("Error on shmat.\n");
                exit(1);
            }
            for (i = 0; i < WORKS_C; i++)
            {
                p(sem_id, SEM_FULL);    //信号量操作
                p(sem_id, SEM_MUTEX);

                sleep(get_random());

                lt = shmptr->letter[shmptr->head];          //缓冲区操作
                shmptr->head = (shmptr->head + 1) % LETTER_NUM;
                shmptr->is_empty = (shmptr->head == shmptr->tail);
                shmptr->index++;
                pid_t pid = getpid();
                now = time(NULL);
                printInfo(now, shmptr, num_p, lt, pid, 1);
                fflush(stdout);

                v(sem_id, SEM_EMPTY);  //信号量操作
                v(sem_id, SEM_MUTEX);
            }
            shmdt(shmptr);
            exit(0);
        }
    }
```

6)自定义输出函数

为了与 Windows 系统下的输出保持一致，我同样在 Linux 下定义了相同的输出方式函数：

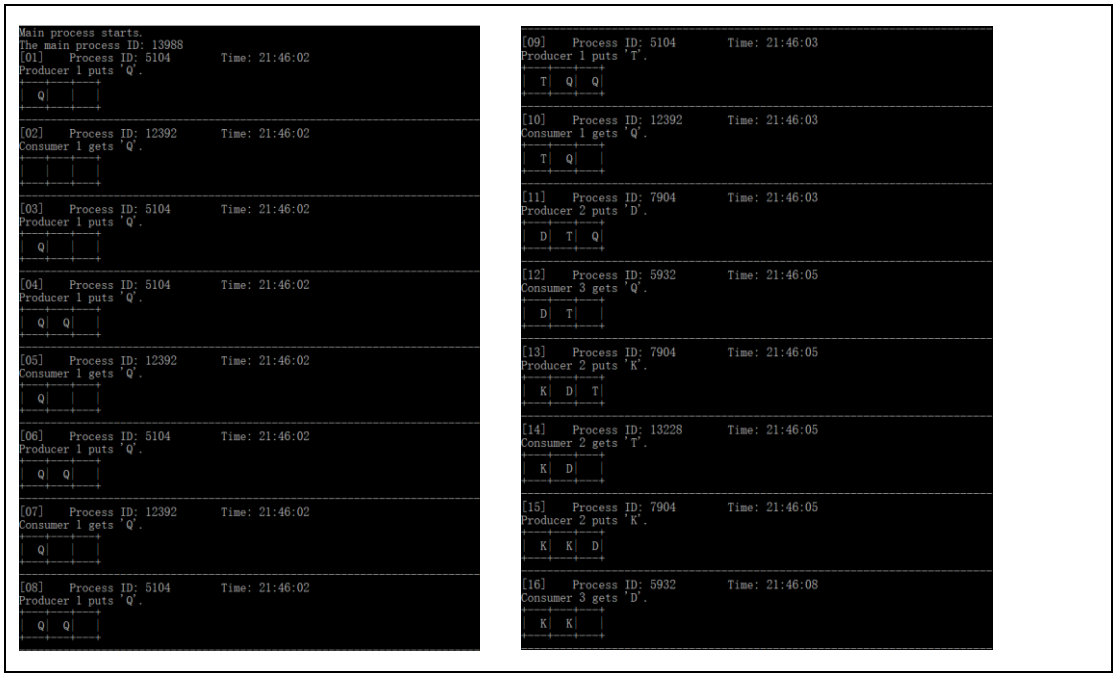
```
void printInfo(time_t now, mybuffer * shmptr, int num, char lt, pid_t mypid, int type){
    printf("[%02d]\tProcess ID: %d\t", shmptr->index, mypid);

    printf("Time: %02d:%02d:%02d\n", localtime(&now)->tm_hour, localtime(&now)->tm_min, localtime(&now)->tm_sec);

    if(type == 0) printf("Producer %d puts '%c'.\n", num, lt);
    else{  printf("Consumer %d gets '%c'.\n", num, lt); }
    int i, j;

    printf("+---+---+---+\n");
    for (i = 0, j = (shmptr->tail - 1 >= shmptr->head) ? (shmptr->tail - 1) : (shmptr->tail - 1 + LETTER_NUM); !(shmptr->is_empty) && j >= shmptr->head; j--)
    {
        i++;
        printf("|");
        printf("%3c", shmptr->letter[j % LETTER_NUM]);
    }
    for(int k = i + 1; k <= 3; k++){
        printf("|   ");
    }
    printf("|\n");
    printf("+---+---+---+\n");

    printf("--------------------------------------------------------------------------\n");
}
```

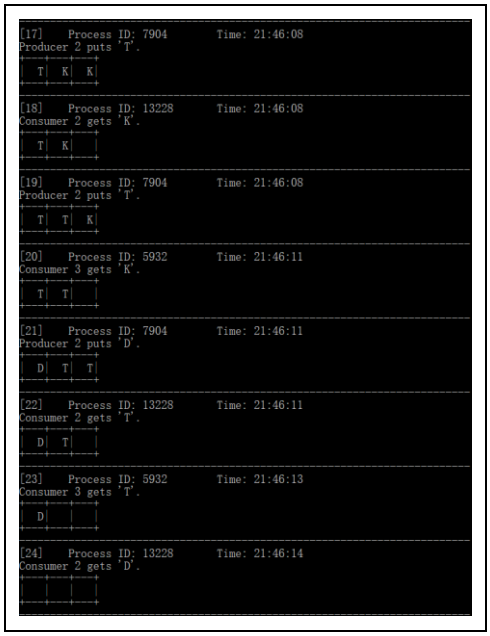
7)编译运行代码，观察实验结果。

五、实验结果和分析

1.Windows 环境

程序运行结果截图：



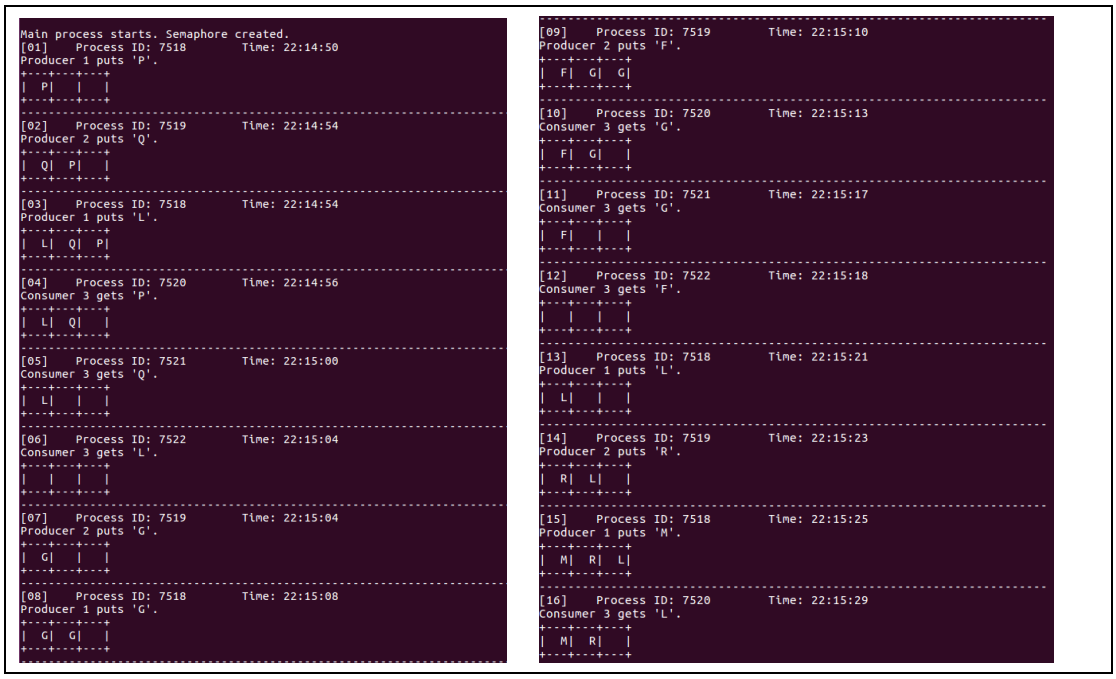


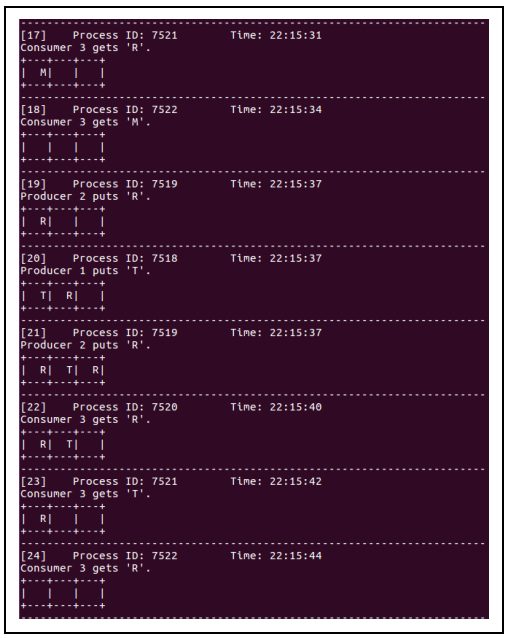
结果分析：

由图可见，缓冲区初始化为空，按照先生产先消费的顺序，生产者进程向缓冲区中写入字母，消费者进程从缓冲区中读取字母；缓冲区存满之后，生产者无法继续写入，只有等待消费者读取之后才可以继续写入。在进程运行的整个过程中，生产者共有 2 个，每一个生产者随机进行 6 次写字母操作；消费者有 3 个，每一个消费者随机进行 4 次读字母操作。共有二十四次进程操作，缓冲区最后的内容为空。

实现了生产者进程与消费者对于缓冲区的互斥操作，与预期结果一致。

1. Linux 环境





结果分析：

由图可见，运行结果与 Windows 环境下结果相同，说明在 Linux 系统下利用信号量机制实现了生产者进程与消费者进程对于缓冲区的互斥读写操作，与预期结果一致。

六、讨论、心得

本次实验内容利用信号量机制，实现进程之间的同步和互斥关系。

对于信号量机制，在之前的操作系统课程中，有了一定的理论上的了解，但对于代码实现，确实有很多不懂得地方，这个实验确实比较难。我通过阅读课本上的函数示例与网上大量的参考资料，也参考了一些学长的代码，也发现了一些存在的问题。

另外，通过这次实验，我还学会了共享内存的使用，使得不同的进程可以访问一块相同的内存区域，来进行数据交换，提高了效率，但为了正确使用共享内存，就需要使用信号量机制对其进行互斥与同步关系控制。

另外还有一个问题是始终还没有解决的，在小规模的数据量下，windows 系统下的随机性并没有 Linux 表现好，总是出现生产者与消费者交替访问缓冲区的情况，在规模较大的时候，随机性较好。