操作系统课程设计实验报告

实验名称	生产者消费者问题		
学号	1120141831	姓名	朴泉宇

一、实验目的

在 Windows 以及 Linux 平台下实现多进程经典问题——生产者消费者问题，掌握在两个平台下进程之间的通信方法。

二、实验内容

1. 分别编写 Windows 下的多进程代码、Linux 下的多进程代码
2. 在 Windows 下编译运行程序，得到相应结果
3. 在 Linux 下编译运行程序，得到相应结果
4. 对比 Windows 下以及 Linux 下的进程之间通信方式的区别

三、实验环境及配置方法

1. 利用 Visual Studio 编写 Windows 下的代码，并编译运行
2. 利用 Notepad++编写 Linux 下的代码，在 Ubuntu 中调用 g++编译，运行

四、实验方法和实验步骤（程序设计与实现）

1. 编写 Windows 下的多进程代码

(1) 思路

1) 在 CreateProcess()的时候，借鉴 Linux 的 fork()的思路，将主程序复制成多份子程序,在创建子程序的时候将递增的程序序列号（Serial No.，即 nCloneID）分配给每个子程序。这个序列号用于判断进入 Producer 分支或者 Consumer 分支。

2) 进程间的通信采用共享内存及文件映射的方式。父进程创建共享内存及文件映射，还有对应的信号量，供子进程使用、维护。

3) 缓冲区采用循环队列形式。

4) Producer 子进程获取共享内存和文件映射的句柄，用于操作缓冲区;也获取信号量的句柄,用于限制其他进程缓冲区的使用权。生产特定字母（P、Q、Y），放入缓冲区，并释放缓冲区的使用权。

5) Consumer 子进程获取共享内存和文件映射的句柄，用于操作缓

冲区；也获取信号量句柄，用于限制其他进程对缓冲区的使用。消费已在缓冲区中的字母（按照先生产先消费原则），并释放缓冲区的使用权。

6）父进程等待所有 Producer 以及 Consumer 进程都完成各自的生产和消费任务。关闭相关句柄，结束程序。

(2) #define 及数据结构定义解释

```
#include <stdio.h>
#include <Windows.h>
#include <time.h>
#include <tchar.h>

#define ID_MAIN 0  //Main 函数的序列号
#define ID_PRODUCER_START 1  //Producer 进程开始序列号
#define ID_PRODUCER_ENDS 3  //Producer 进程结束序列号
#define ID_CONSUMER_START 4  //Consumer 进程开始序列号
#define ID_CONSUMER_ENDS 7  //Consumer 进程结束序列号

#define PRODUCER_WORKS_TIMES 4  //每一个 Producer 生产 4 次
#define CONSUMER_WORKS_TIMES 3  //每一个 Consumer 消费 3 次

#define BUFFER_SIZE 4  //缓冲区大小
#define PROCESS_NUM 7  //子进程总数量 用于创建子进程列表
TCHAR sharedMemName[] = TEXT("Global\\MyFileMappingObject");  //共享内存映射文件名

/*The Buffer, use to cache.*/
struct myBuffer
{
    char Buffer[BUFFER_SIZE + 1];
    int head;
    int tail;
    int isEmpty;
};

/*The sharedMemory used to realize Buffer.*/
struct sharedMem
{
    struct myBuffer bufferData;
    int index;
};

```

```
35. /*File Mapping HANDLE.*/
36. static HANDLE hMapping;
37.
38. /*The HANDLE array of Child Process.*/
39. static HANDLE hChildProcess[PROCESS_NUM + 1];
```

(3) 函数定义解释

1) 随机数函数

```
1.  /*Get the number between 0 - 3000, used to stop between 0 - 3 seconds.*/
2.  int getRandomInt()
3.  {
4.      int randnum;
5.      //srand((unsigned)(GetCurrentProcessId() + time(NULL)));
6.      randnum = rand() % 3001;
7.
8.      return randnum;
9.  }
```

2) 随机获取字母函数

```
1.  /*Get the Letter P or Q or Y, is the product.*/
2.  char getRandomLetter()
3.  {
4.      char letterMap[4] = { 'P','Q','Y','\0' };
5.      int randNum;
6.      //srand((unsigned)(GetCurrentProcessId() + time(NULL)));
7.      randNum = rand() % 3;
8.
9.      return letterMap[randNum];
10. }
```

3) 创建共享内存函数

```
1.  /*Make shared memory.*/
2.  HANDLE MakeSharedMem()
3.  {
4.      /*Make view of file.*/
5.      HANDLE fMapping = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE
    , 0, sizeof(struct sharedMem), sharedMemName);
6.      if (fMapping != INVALID_HANDLE_VALUE)
7.      {
8.          LPVOID pData = MapViewOfFile(fMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);
```

```
        if (pData != NULL)
        {
            memset(pData, 0, sizeof(struct sharedMem));
        }
        UnmapViewOfFile(pData);
    }

    return fMapping;
}
```

4）创建子进程函数

```
/*Clone the Process.*/
void cloneChildProcess(int nCloneID)
{
    TCHAR szFilename[MAX_PATH];
    TCHAR szCmdLine[MAX_PATH];
    char tempFilename[MAX_PATH];
    char tempCmdLine[MAX_PATH];
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    GetModuleFileName(NULL, szFilename, MAX_PATH);
    TCHAR_to_char(szFilename, tempFilename);
    sprintf(tempCmdLine, "\"%s\" %d", tempFilename, nCloneID);  //给子进程传入参数
    char_to_TCHAR(tempCmdLine, szCmdLine);
    memset(&si, 0, sizeof(si));
    si.cb = sizeof(si);

    /*Create Child Process.*/
    BOOL bCreateOK = CreateProcess(
        szFilename,
        szCmdLine,
        NULL,
        NULL,
        FALSE,
        0,
        NULL,
        NULL,
        &si,
        &pi
    );
    hChildProcess[nCloneID] = pi.hProcess;  //向子进程列表中对应子进程填充 pi.hProcess 信息
```

```
32.
33.     return;
34. }
```

5）THCAR 转 char 函数

```
1.  /*Change TCHAR to char*/
2.  void TCHAR_to_char(const TCHAR * tchar, char * _char)
3.  {
4.      int iLength;
5.
6.      iLength = WideCharToMultiByte(CP_ACP, 0, tchar, -1, NULL, 0, NULL, NULL);   //
    Get the length.
7.      WideCharToMultiByte(CP_ACP, 0, tchar, -1, _char, iLength, NULL, NULL);  //Give
     tchar value to _char.
8.  }
```

6）char 转 TCAHR 函数

```
1.  /*Change char to TCHAR*/
2.  void char_to_TCHAR(const char * _char, TCHAR * tchar)
3.  {
4.      int iLength;
5.
6.      iLength = MultiByteToWideChar(CP_ACP, 0, _char, strlen(_char) + 1, NULL, 0);
7.      MultiByteToWideChar(CP_ACP, 0, _char, strlen(_char) + 1, tchar, iLength);
8.  }
```

（4）main 函数解释

1）信号量句柄声明以及打印进程信息（进程 PID 以及序列号 nCloneID）

```
1.      int nClone = ID_MAIN;
2.      SYSTEMTIME nowTime;
3.      HANDLE semEmpty;  //声明取空缓冲区，同步信号量句柄
4.      HANDLE semFull;  //声明取有产品的缓冲区，同步信号量句柄
5.      HANDLE semMutex;  //声明对整个缓冲区的互斥信号量
6.
7.      /*Give the  parameter to nClone.*/
8.      if (argc > 1)
9.      {
10.         sscanf(argv[1], "%d", &nClone);
```

```
    }

    /*Print Process ID and Serial Number.*/
    printf("Process ID: %d, Serial No: %d.\n", GetCurrentProcessId(), nClone);
```

2）父进程：创建内存共享，创建文件映射，初始化缓冲区，创建信号量，创建生产者、消费者子进程，等待子进程结束。

```
    if (nClone == ID_MAIN)
    {
        printf("Main Process starts.\n");

        /*Start the shared memory.*/
        hMapping = MakeSharedMem();
        /*Mapping the view*/
        HANDLE hFileMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, sharedMemName);
        LPVOID pFile = MapViewOfFile(hFileMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);

        if (pFile == NULL)
        {
            printf("OpenFileMapping Error!\n");

            return -1;
        }
        else
        {
            struct sharedMem * SHM = (struct sharedMem *)pFile;
            memset(SHM->bufferData.Buffer, '-', sizeof(SHM->bufferData.Buffer));
            SHM->bufferData.Buffer[BUFFER_SIZE] = '\0';
            SHM->bufferData.head = 0;
            SHM->bufferData.tail = 0;
            SHM->index = 0;
            semEmpty = CreateSemaphore(NULL, BUFFER_SIZE, BUFFER_SIZE, TEXT("SEM_EMPTY"));
            semFull = CreateSemaphore(NULL, 0, BUFFER_SIZE, TEXT("SEM_FULL"));
            semMutex = CreateMutex(NULL, FALSE, TEXT("SEM_MUTEX"));
            UnmapViewOfFile(pFile);
            pFile = NULL;
        }
        CloseHandle(hFileMapping);

        /*Clone the Child Process.*/
```

```
        int childprocessNum = 1;
        for (childprocessNum = 1;childprocessNum <= PROCESS_NUM;childprocessNum++)

        {
            cloneChildProcess(childprocessNum);
        }

        /*Wait child process ends.*/
        int i;
        for (i = 1; i <= PROCESS_NUM;i++)
        {
            WaitForSingleObject(hChildProcess[i], INFINITE);
            CloseHandle(hChildProcess[i]);
        }
        printf("Main Process ends.\n");
    }
```

3) Producer 进程：获取内存共享及内存共享的句柄，获取信号量句柄，每个生产者进程循环 `PRODUCER_WORKS_TIMES` 生产产品，打印时间、缓冲区、生产信息。生产完毕后结束进程。

```
        /*The Producer Process.*/
        if (nClone >= ID_PRODUCER_START&&nClone <= ID_PRODUCER_ENDS)
        {
            printf("Producer No.%d starts.\n", nClone - ID_MAIN);

            /*Mapping the view.*/
            HANDLE hFileMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, shar
edMemName);
            LPVOID pFile = MapViewOfFile(hFileMapping, FILE_MAP_ALL_ACCESS, 0, 0,
0);
            if (pFile == NULL)
            {
                printf("OpenFileMapping Error!\n");

                return -1;
            }
            else
            {
                struct sharedMem * SHM = (struct sharedMem*)pFile;
                semEmpty = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, TEXT("SEM_EM
PTY"));
                semFull = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, TEXT("SEM_FUL
```

```
    L"));
20.                 semMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, TEXT("SEM_MUTEX"));
21.
22.                 /*Start Producer Work Times.*/
23.                 srand((unsigned)(GetCurrentProcessId() + time(NULL)));
24.                 char productLetter;
25.                 int i;
26.                 for (i = 0;i < PRODUCER_WORKS_TIMES;i++)
27.                 {
28.                     Sleep(getRandomInt());
29.                     WaitForSingleObject(semEmpty, INFINITE);
30.                     WaitForSingleObject(semMutex, INFINITE);
31.
32.                     /*Produce*/
33.                     productLetter = getRandomLetter();
34.                     SHM->index++;
35.                     SHM->bufferData.Buffer[SHM->bufferData.tail] = productLetter;
36.                     SHM->bufferData.tail = (SHM->bufferData.tail + 1) % BUFFER_SIZ
    E;
37.                     SHM->bufferData.isEmpty = 0;
38.
39.                     /*Print the index, time, and the product.*/
40.                     GetLocalTime(&nowTime);
41.                     printf("[%02d]\t", SHM->index);  //Print index.
42.                     printf("%02d:%02d:%02d\t", nowTime.wHour, nowTime.wMinute, now
    Time.wSecond);    //Print now time.
43.                     int j;
44.                     for (j = 0;j < BUFFER_SIZE;j++)
45.                     {
46.                         printf("%c", SHM->bufferData.Buffer[j]);
47.                     }
48.                     printf("\tProducer No.%d produced '%c'.\n", nClone - ID_MAIN,
    productLetter);
49.
50.                     ReleaseSemaphore(semFull, 1, NULL);
51.                     ReleaseMutex(semMutex);
52.                 }
53.                 UnmapViewOfFile(pFile);
54.                 pFile = NULL;
55.             }
56.             CloseHandle(hFileMapping);
57.             printf("Producer No.%d ends.\n", nClone - ID_MAIN);
```

```
58.         }
```

4) Consumer 进程：获取内存共享及内存共享的句柄，获取信号量句柄，每个消费者进程循环 CONSUMER_WORKS_TIMES 3 消费产品，打印时间、缓冲区、消费信息。消费完毕后结束进程。

```
1.              /*The Consumer Process.*/
2.              if (nClone >= ID_CONSUMER_START&&nClone <= ID_CONSUMER_ENDS)
3.              {
4.                  printf("Consumer No.%d starts.\n", nClone - ID_PRODUCER_ENDS);
5.
6.                  /*Mapping the view.*/
7.                  HANDLE hFileMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE,
    sharedMemName);
8.                  LPVOID pFile = MapViewOfFile(hFileMapping, FILE_MAP_ALL_ACCESS, 0,
     0, 0);
9.                  if (pFile == NULL)
10.                 {
11.                     printf("OpenFileMapping Error!\n");
12.
13.                     return -1;
14.                 }
15.                 else
16.                 {
17.                     struct sharedMem * SHM = (struct sharedMem*)pFile;
18.                     semEmpty = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, TEXT("SE
    M_EMPTY"));
19.                     semFull = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, TEXT("SEM
    _FULL"));
20.                     semMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, TEXT("SEM_MUTEX"
    ));
21.
22.                     /*Start Consumer Work Times.*/
23.                     srand((unsigned)(GetCurrentProcessId() + time(NULL)));
24.                     char ConsumptionLetter;
25.                     int i;
26.                     for (i = 0; i < CONSUMER_WORKS_TIMES;i++)
27.                     {
28.                         Sleep(getRandomInt());
29.                         WaitForSingleObject(semFull, INFINITE);
30.                         WaitForSingleObject(semMutex, INFINITE);
31.
32.                         /*Consume*/
```

```
33.                         SHM->index++;
34.                         ConsumptionLetter = SHM->bufferData.Buffer[SHM->bufferData
    .head];
35.                         SHM->bufferData.Buffer[SHM->bufferData.head] = '-';   //Se
    t Used!
36.                         SHM->bufferData.head = (SHM->bufferData.head + 1) % BUFFER
    _SIZE;
37.                         SHM->bufferData.isEmpty = (SHM->bufferData.head == SHM->bu
    fferData.tail);
38.
39.                         /*Print the index, time, and the consumption.*/
40.                         GetLocalTime(&nowTime);
41.                         printf("[%02d]\t", SHM->index);
42.                         printf("%02d:%02d:%02d\t", nowTime.wHour, nowTime.wMinute,
     nowTime.wSecond);
43.                         int j;
44.                         for (j = 0;j < BUFFER_SIZE;j++)
45.                         {
46.                             printf("%c", SHM->bufferData.Buffer[j]);
47.                         }
48.                         printf("\tConsumer No.%d consumed '%c'.\n", nClone - ID_PR
    ODUCER_ENDS, ConsumptionLetter);
49.
50.                         ReleaseSemaphore(semEmpty, 1, NULL);
51.                         ReleaseMutex(semMutex);
52.                     }
53.                     UnmapViewOfFile(pFile);
54.                     pFile == NULL;
55.                 }
56.                 CloseHandle(hFileMapping);
57.                 printf("Consumer No.%d ends.\n", nClone - ID_PRODUCER_ENDS);
58.             }
59.             else
60.             {
61.                 printf("nClone Error!\n");
62.
63.                 return -1;
64.             }
```

5）结束：关闭句柄，结束。

```
1.      CloseHandle(hMapping);
2.      hMapping == INVALID_HANDLE_VALUE;
3.
```

```
4.      return 0;
```

2. 编写 Linux 下的多进程代码

(1) 思路

1) Linux 的 fork()的继承特性，使得无需在进程之间的关联性上像 Windows 一样进行命名等标记，也无需通过传入序列号来进行生产者消费者进程的控制。

2) 进程间的通信同样采用共享内存的方式。父进程创建共享内存，还有对应的信号量，供子进程使用、维护。

3) 缓冲区同样采用循环队列形式。

4) Producer 子进程获取共享内存的指针，用于操作缓冲区；信号量的控制模拟理论上的 P、V 操作，用于限制或释放其他进程缓冲区的使用权。生产特定字母（P、Q、Y），放入缓冲区，并释放缓冲区的使用权。

5) Consumer 子进程获取共享内存指针，用于操作缓冲区；信号量的控制模拟理论上的 PV 操作，用于限制或释放其他进程缓冲区的使用权。消费已在缓冲区中的字母（按照先生产先消费原则），并释放缓冲区的使用权。

6) 父进程等待所有 Producer 以及 Consumer 进程都完成各自的生产和消费任务。解绑共享内存及信号量，结束。

(2) #define 及数据结构定义解释

```
1.  #include <stdio.h>
2.  #include <time.h>
3.  #include <unistd.h>
4.  #include <stdlib.h>
5.  #include <sys/types.h>
6.  #include <sys/wait.h>
7.  #include <sys/time.h>
8.  #include <sys/ipc.h>
9.  #include <sys/shm.h>
10. #include <sys/sem.h>
11.
12. //#define ID_MAIN 0
13. //#define ID_PRODUCER_START 1
```

```
//#define ID_PRODUCER_ENDS 3
//#define ID_CONSUMER_START 4
//#define ID_CONSUMER_ENDS 7
#define ID_PRODUCER_NEED 3  //生产者数量
#define ID_CONSUMER_NEED 4  //消费者数量

#define PRODUCER_WORKS_TIMES 4  //每一个 Producer 生产 4 次
#define CONSUMER_WORKS_TIMES 3  //每一个 Consumer 消费 3 次

#define BUFFER_SIZE 4  //缓冲区大小
//#define PROCESS_NUM 7
#define SHM_MODE 0600

#define SEM_EMPTY 0  //缓冲区空位同步信号量
#define SEM_FULL 1  //缓冲区满位同步信号量
#define SEM_MUTEX 2  //使用缓冲区互斥信号量

/*The Buffer, use to catch.*/
struct myBuffer
{
    int index;
    char Buffer[BUFFER_SIZE + 1];
    int head;
    int tail;
    int isEmpty;
};
```

(3) 函数定义解释

1) P、V 操作

```
/*P operation*/
void op_P(int sem_id, int sem_num)
{
    struct sembuf TMP;
    TMP.sem_num = sem_num;
    TMP.sem_op = -1;
    TMP.sem_flg = 0;
    semop(sem_id, &TMP, 1);
}

/*V operation*/
void op_V(int sem_id, int sem_num)
{
```

```
    struct sembuf TMP;
    TMP.sem_num = sem_num;
    TMP.sem_op = 1;
    TMP.sem_flg = 0;
    semop(sem_id, &TMP, 1);
}
```

2）取随机数函数

```
/*Get the number between 0 - 3000000, used to stop between 0 - 3 seconds.*/
int getRandomInt()
{
    int randnum;
    //srand((unsigned)(getpid() + time(NULL)));
    randnum = rand() % 3000001;

    return randnum;
}
```

3）取随机字母函数

```
/*Get the Letter P or Q or Y, is the product.*/
char getRandomLetter()
{
    char letterMap[4] = { 'P','Q','Y','\0' };
    int randNum;
    //srand((unsigned)(getpid() + time(NULL)));
    randNum = rand() % 3;

    return letterMap[randNum];
}
```

(4) main 函数解释

1）父进程：信号量集的创建，共享内存的创建，初始化

```
    time_t nowTime;
    pid_t pid_Producer, pid_Consumer;

    int SEM_ALL_KEY = ftok("/tmp", 0x66);
    if (SEM_ALL_KEY < 0)
    {
        printf("ftok SEM_ALL_KEY Error!\n");

```

```
9.          return -1;
10.     }
11.     int SEM_ID = semget(SEM_ALL_KEY, 3, IPC_CREAT | 0600);
12.     if (SEM_ID >= 0)
13.     {
14.         printf("Main process starts.\n");
15.     }
16.     else
17.     {
18.         printf("Semaphore Create Error!\n");
19.
20.         return -1;
21.     }
22.     semctl(SEM_ID, SEM_EMPTY, SETVAL, BUFFER_SIZE);
23.     semctl(SEM_ID, SEM_FULL, SETVAL, 0);
24.     semctl(SEM_ID, SEM_MUTEX, SETVAL, 1);
25.
26.     int SHM_ID = shmget(IPC_PRIVATE, sizeof(struct myBuffer), SHM_MODE);
27.     if (SHM_ID < 0)
28.     {
29.         printf("SharedMemory Create Error!\n");
30.
31.         return -1;
32.     }
33.
34.     struct myBuffer *SHMPTR;
35.     SHMPTR = (struct myBuffer *)shmat(SHM_ID, 0, 0);
36.     if (SHMPTR == (void *)-1)
37.     {
38.         printf("shmat Error!\n");
39.
40.         return -1;
41.     }
42.     SHMPTR->index = 0;
43.     SHMPTR->head = 0;
44.     SHMPTR->tail = 0;
45.     SHMPTR->isEmpty = 1;
46.     int i;
47.     for (i = 0;i<BUFFER_SIZE;i++)
48.     {
49.         SHMPTR->Buffer[i] = '-';
50.     }
51.     SHMPTR->Buffer[BUFFER_SIZE] = '\0';
```

2）Producer 进程：与 Windows 下的操作一致，仅在调用上有所区别，故不赘述。

```
/*The Producer Process.*/
int producerNum;
for (producerNum = 0;producerNum < ID_PRODUCER_NEED; producerNum++)
{
    pid_Producer = fork();
    if (pid_Producer<0)
    {
        printf("fork Error!\n");

        return -1;
    }

    if (pid_Producer == 0)
    {
        SHMPTR = (struct myBuffer *)shmat(SHM_ID, 0, 0);
        if (SHMPTR == (void *)-1)
        {
            printf("shmat Error!\n");

            return -1;
        }

        /*Start Producer Work Times.*/
        srand((unsigned)(getpid() + time(NULL)));
        char productLetter;
        int i;
        for (i = 0;i<PRODUCER_WORKS_TIMES;i++)
        {
            usleep(getRandomInt());
            op_P(SEM_ID, SEM_EMPTY);
            op_P(SEM_ID, SEM_MUTEX);

            /*Produce*/
            productLetter = getRandomLetter();
            SHMPTR->Buffer[SHMPTR->tail] = productLetter;
            SHMPTR->tail = (SHMPTR->tail + 1) % BUFFER_SIZE;
            SHMPTR->isEmpty = 0;
            SHMPTR->index++;

            /*Print the index, time, and the product.*/
```

```
                nowTime = time(NULL);
                printf("[%02d]\t", SHMPTR->index);   //Print index.
                printf("%02d:%02d:%02d\t", localtime(&nowTime)->tm_hour, localtime
(&nowTime)->tm_min, localtime(&nowTime)->tm_sec);    //Print now time.
                int j;
                for (j = 0; j<BUFFER_SIZE; j++)
                {
                    printf("%c", SHMPTR->Buffer[j]);
                }
                printf("\tProducer No.%d produced '%c'.\n", producerNum + 1, produ
ctLetter);

                op_V(SEM_ID, SEM_FULL);
                op_V(SEM_ID, SEM_MUTEX);
            }
            shmdt(SHMPTR);
            printf("Producer No.%d ends.\n", producerNum + 1);

            return 0;
        }
    }
```

3) Consumer 进程：与 Windows 下的操作一致，仅在调用上有所区别，故不赘述。

```
    /*The Consumer Process.*/
    int consumerNum;
    for (consumerNum = 0;consumerNum < ID_CONSUMER_NEED; consumerNum++)
    {
        pid_Consumer = fork();
        if (pid_Consumer<0)
        {
            printf("fork Error!\n");

            return -1;
        }

        if (pid_Consumer == 0)
        {
            SHMPTR = (struct myBuffer *)shmat(SHM_ID, 0, 0);
            if (SHMPTR == (void *)-1)
            {
                printf("shmat Error!\n");
```

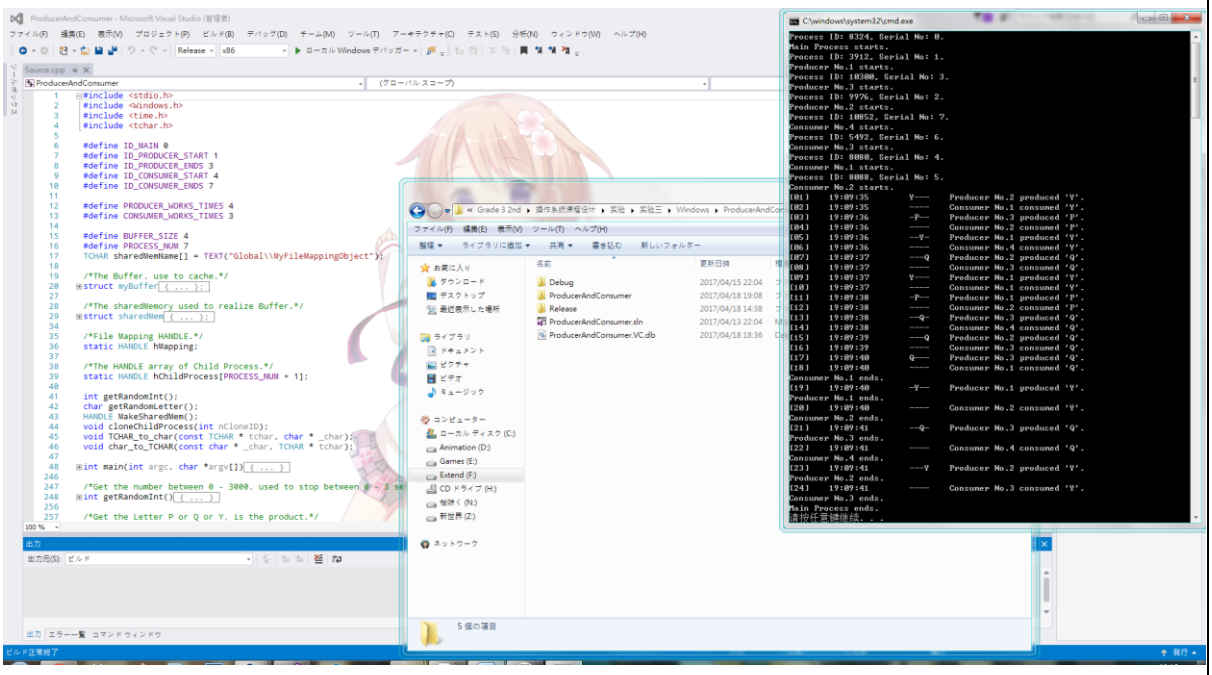
```
19.
20.                 return -1;
21.             }
22.
23.             /*Start Consumer Work Times.*/
24.             srand((unsigned)(getpid() + time(NULL)));
25.             char consumptionLetter;
26.             int i;
27.             for (i = 0;i<CONSUMER_WORKS_TIMES;i++)
28.             {
29.                 usleep(getRandomInt());
30.                 op_P(SEM_ID, SEM_FULL);
31.                 op_P(SEM_ID, SEM_MUTEX);
32.
33.                 /*Consume*/
34.                 consumptionLetter = SHMPTR->Buffer[SHMPTR->head];
35.                 SHMPTR->Buffer[SHMPTR->head] = '-';
36.                 SHMPTR->head = (SHMPTR->head + 1) % BUFFER_SIZE;
37.                 SHMPTR->isEmpty = (SHMPTR->head == SHMPTR->tail);
38.                 SHMPTR->index++;
39.
40.                 /*Print the index, time, and the product.*/
41.                 nowTime = time(NULL);
42.                 printf("[%02d]\t", SHMPTR->index);   //Print index.
43.                 printf("%02d:%02d:%02d\t", localtime(&nowTime)->tm_hour, localtime
    (&nowTime)->tm_min, localtime(&nowTime)->tm_sec);    //Print now time.
44.                 int j;
45.                 for (j = 0; j<BUFFER_SIZE; j++)
46.                 {
47.                     printf("%c", SHMPTR->Buffer[j]);
48.                 }
49.                 printf("\tConsumer No.%d consumed '%c'.\n", consumerNum + 1, consu
    mptionLetter);
50.
51.                 op_V(SEM_ID, SEM_EMPTY);
52.                 op_V(SEM_ID, SEM_MUTEX);
53.             }
54.             shmdt(SHMPTR);
55.             printf("Consumer No.%d ends.\n", consumerNum + 1);
56.
57.             return 0;
58.         }
59.     }
```

4）结束：等待所有子进程结束，并解绑信号量集、共享内存，结束。

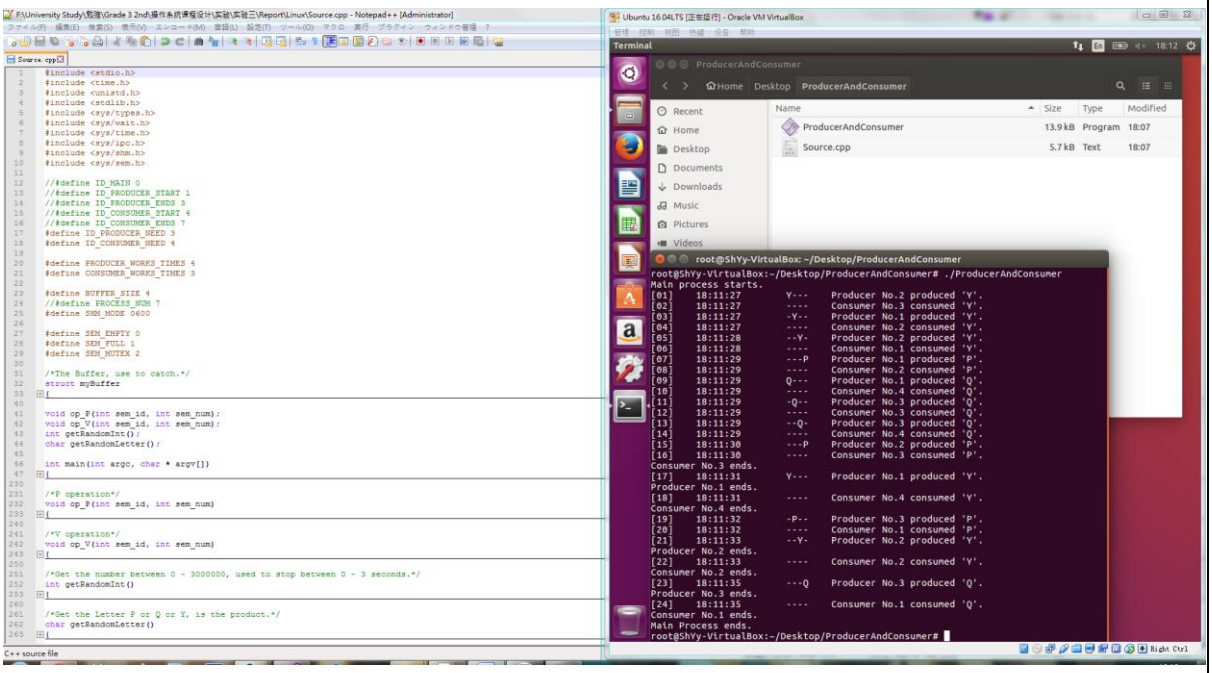
```
1.      /*Main process ends.*/
2.      while (wait(NULL) != -1);
3.      shmdt(SHMPTR);
4.      shmctl(SHM_ID, IPC_RMID, 0);
5.      shmctl(SEM_ID, IPC_RMID, 0);
6.      printf("Main Process ends.\n");
7.      fflush(stdout);
8.
9.      return 0;
```

五、 实验结果和分析

1. Windows 下的实验结果



2. Linux 下的实验结果



3. 对比分析

Windows 与 Linux 下的区别在于内存共享部分的调用方法以及对于信号量的操作上，相似度较高，若掌握了一个平台的实现方法，则在另一个平台下，正确调用 API 即可实现。

六、 讨论、心得

此次实验，实现了操作系统中的经典问题。在实现这个问题的过程中，了解并掌握了 Windows 和 Linux 平台下的信号量操作方式，掌握了共享内存的进程间通信方式。

并且，在调用 API 的时候，通过查阅 MSDN 以及各方资料，阅读了一些源码，对两种平台下的进程操作方式有了更深的理解。

在修改的过程中，有同学为我指出了在 Windows 下句柄操作中存在的重要问题，并与我讨论、帮助我修改，让我加深了对句柄的理解，感激不尽。

学路漫漫，还需精进。