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：曾煜瑾/1120172765

一、 实验目的

1. 学习掌握操作系统中进程之间的通信

2. 掌握使用信号量基址进行多进程之间互斥访问共享内存区域的控制

3. 学习进程的创建和控制，共享内存区域的创建、使用和删除，信号量的创建使用和删除

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

生产者和消费者用进程模拟

三、 实验环境

	名称	版本
Windows 操作系统	Windows 10	企业版

Windows IDE	VSCode	1.40.1.0
Linux 操作系统	Ubuntu 16.04	内核 4.13.0
Linux IDE	gcc	5.4.0

四、 程序设计与实现

Windows 系统：

•总的思路为主程序先建立所和共享内存区，然后调用自身形成两个生产者和三个消费者，生产者和消费者请求信号量，请求满足后，打开共享内存区，如果满足生产或消费条件（即生产时共享内存区有空位，消费时共享内存区非空），就修改数据然后释放信号量。

（1）API 介绍

• CreateProgess( )用于创建进程并为进程指定运行程序

• CreateMutex( )用于创建互斥体对象

• OpenMutex( )用于为一个已经存在的互斥体对象创建一个新的句柄

• ReleaseMutex( )用于释放互斥体对象

• CreateFileMapping( )用于创建一个文件映射内核对象

• OpenFileMapping( )用于打开一个已经存在的文件映射对象，返回打开的句柄

• MapViewofFile( )用于将一个文件映射对象映射到当前程序地址空间

• UnmapViewOfFIle( )用于解除当前程序的一个内存映射

• CreateSemaphore( )用于创建一个信号量，返回相应信号量的句柄

• OpenSemaphore( )用于打开一个已经存在的信号量对象

• ReleaseSemaphore( )用于释放信号量

• WaitForSingleObject( )用于等待对象信号状态

• CloseHandle( )用于关闭现有已打开句柄

（2）过程介绍

• 对于主进程：

1. 创建互斥体对象和共享内存区的文件映射

2. 把文件映射到进程的地址空间，用读写内存的方式操作和处理文件数据

3. 建立共享内存区并初始化相关数据

4. 创建 2 个生产者子进程和 3 个消费者子进程

5. 解除文件映射，释放互斥体

• 对于生产者进程和消费者进程：

1. 为现有的互斥体对象创建一个新句柄

2. 打开已经存在的信号量对象

3. 循环生产者进程和消费者进程的每个子操作，把文件映射到进程的地址空间后，进行相应的读写操作和打印操作

4. 解除文件映射，释放信号量，释放互斥体

（3）代码实现

• 定义宏变量

```
//2个生产者，每个生产者工作6次
#define Need_Producer 2
#define Works_Producer 6
//3个消费者，每个消费者工作4次
#define Need_Customer 3
#define Works_Customer 4
//缓冲区为3
#define buffer_len 3
```

• 定义缓冲区结构

```
struct buf
{
    int num;
    int read;
    int write;
    int buffer[buffer_len];
};
```

其中 num 表示缓冲区现有产品的数量，read 表示生产者的当前指针，write

表示消费者的当前指针。buffer 是一个长度为 buffer_len 的数组，1 表示有产品，0 表示无产品。

• 创建主进程和缓冲区

```
void Parent()
{
    hMutexMapping = CreateMutex(NULL, true, "mutex"); //创建互斥体对象
    //创建共享内存区的文件映射
    HANDLE hMapping = CreateFileMapping(NULL, NULL, PAGE_READWRITE, 0, sizeof(LONG), "map");
    if (hMapping != INVALID_HANDLE_VALUE)
    {
        //把文件映射到进程的地址空间，用读写内存的方式操作和处理文件数据
        LPVOID pData = MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);
        if (pData != NULL)
            ZeroMemory(pData, sizeof(LONG));
        struct buf *shmptr = reinterpret_cast<struct buf *>(pData);
        shmptr->read = 0;
        shmptr->write = 0;
        shmptr->num = 0;
        memset(shmptr->buffer, 0, sizeof(shmptr->buffer));
        UnmapViewOfFile(pData); //从进程的地址空间撤销文件数据的映像
    }
    CreateSemaphore(NULL, 3, 3, "EMPTY");
    CreateSemaphore(NULL, 0, 3, "FULL");
    //创建5个子进程
    StartClone();
    ReleaseMutex(hMutexMapping); //释放互斥体
}
```

其中 StartClone()用于创建两个生产者进程和三个消费者进程，创建两个生产者进程的代码如下：

```
for (i = 1; i <= Need_Producer; i++)
{
    sprintf(szCmdLine, "\"%s\" Producer %d", szFilename, i);
    STARTUPINFO si;
    ZeroMemory(reinterpret_cast<void *>(&si), sizeof(si));
    si.cb = sizeof(si);
    bCreateOK = CreateProcess(szFilename, szCmdLine, NULL, NULL,
        FALSE, CREATE_DEFAULT_ERROR_MODE, NULL, NULL, &si, &pi);
    if (!bCreateOK)
        return false;
    else
    {
        printf("Producer %d is created\n", i);
        lpHandle[num] = pi.hProcess;
        num++;
    }
}
```

• 生产者往缓冲区生产数据

```
struct buf *shmptr = reinterpret_cast<struct buf *>(pFile);
shmptr->buffer[shmptr->read] = 0;
shmptr->num--;
```

• 消费者从缓冲区消费数据

```
struct buf *shmptr = reinterpret_cast<struct buf *>(pFile);
shmptr->buffer[shmptr->write] = 1;
shmptr->num++;
```

• 显示当前时间

```
SYSTEMTIME time;
GetSystemTime(&time);
printf("\n%04d/%02d/%02d-%02d:%02d:%02d", time.wYear, time.wMonth,
    time.wDay, time.wHour+8, time.wMinute, time.wSecond);
```

• 打印当前缓冲区数据情况

```
printf("\tConsumer %d gets data from Position [%d]", n, shmptr->read + 1);
printf("\tCurrent buffer: ");
for (j = 0; j < buffer_len; j++)
    printf("%d ", shmptr->buffer[j]);
```

（4）实现效果：

• 编译源文件

```
C:\Users\lenovo\Desktop\windows>gcc cus.cpp -o cus

C:\Users\lenovo\Desktop\windows>
```

• 运行程序

```
Producer 1 is created
Producer 2 is created
Consumer 1 is created
Consumer 2 is created
Consumer 3 is created

2019/12/04-19:23:36     Producer 1 puts data at Position [1]    Current buffer: 1 0 0
2019/12/04-19:23:37     Producer 2 puts data at Position [2]    Current buffer: 1 1 0
2019/12/04-19:23:37     Consumer 1 gets data from Position [1]  Current buffer: 0 1 0
2019/12/04-19:23:37     Producer 1 puts data at Position [3]    Current buffer: 0 1 1
2019/12/04-19:23:37     Consumer 2 gets data from Position [2]  Current buffer: 0 0 1
2019/12/04-19:23:37     Producer 2 puts data at Position [1]    Current buffer: 1 0 1
2019/12/04-19:23:37     Consumer 3 gets data from Position [3]  Current buffer: 1 0 0
2019/12/04-19:23:37     Producer 1 puts data at Position [2]    Current buffer: 1 1 0
2019/12/04-19:23:38     Producer 2 puts data at Position [3]    Current buffer: 1 1 1
2019/12/04-19:23:38     Consumer 1 gets data from Position [1]  Current buffer: 0 1 1
2019/12/04-19:23:38     Consumer 2 gets data from Position [2]  Current buffer: 0 0 1
2019/12/04-19:23:38     Consumer 3 gets data from Position [3]  Current buffer: 0 0 0
2019/12/04-19:23:38     Producer 1 puts data at Position [1]    Current buffer: 1 0 0
2019/12/04-19:23:38     Producer 2 puts data at Position [2]    Current buffer: 1 1 0
2019/12/04-19:23:38     Producer 1 puts data at Position [3]    Current buffer: 1 1 1
2019/12/04-19:23:38     Consumer 1 gets data from Position [1]  Current buffer: 0 1 1
2019/12/04-19:23:38     Consumer 2 gets data from Position [2]  Current buffer: 0 0 1
2019/12/04-19:23:39     Consumer 3 gets data from Position [3]  Current buffer: 0 0 0
2019/12/04-19:23:39     Producer 2 puts data at Position [1]    Current buffer: 1 0 0
2019/12/04-19:23:39     Producer 1 puts data at Position [2]    Current buffer: 1 1 0   Producer 1 works over.
2019/12/04-19:23:39     Producer 2 puts data at Position [3]    Current buffer: 1 1 1   Producer 2 works over.
2019/12/04-19:23:39     Consumer 1 gets data from Position [1]  Current buffer: 0 1 1   Consumer 1 works over.
2019/12/04-19:23:39     Consumer 2 gets data from Position [2]  Current buffer: 0 0 1   Consumer 2 works over.
2019/12/04-19:23:39     Consumer 3 gets data from Position [3]  Current buffer: 0 0 0   Consumer 3 works over.
```

可以看到程序一开始输出两个生产者和三个消费者被产生，之后每次生产者生产数据或者消费者消费数据都输出一行，最开始是当前时间，之后表明具体操作，最后打印当前缓冲区的数据情况，0 表示没有数据，1 表示有数据。另外当生产者生产完 6 次数据或者消费者消费完 4 次数据，都打印一条语句表示对应的工作结束。

Linux 系统：

• 同 windows 系统一样，Linux 也是用主函数创建信号量集合和共享内存区，之后创建 2 个生产者进程和 3 个消费者进程，然后进行一系列的相关操作。

（1）API 介绍

• fork( )创建子进程

• segmet( )用于创建信号量集合

• semctl( )用于执行在信号量集合上的控制操作

• shmget( )用于创建共享内存对象

• shmat( )用于把共享内存区对象映射到调用进程的地址空间

• shmdt( )用于断开共享内存连接

• shmctl( )用于结束对共享内存的控制

（2）过程介绍

• 对于主进程：

1. 创建一个信号量集合，如果返回的信号量集合的标识号不小于 0，输出提示。

2. 对信号量进行控制操作。

3. 申请一个共享内存区，返回共享内存区的标识号，如果标识号小于 0，则申请共享内存区失败并输出相应提示。

4. 将共享段附加到申请通信的进程空间，成功时返回共享内存附加到进程空间的虚地址，失败时返回-1，若返回-1 则输出相应提示。

5. 初始化环形缓冲区中的数据成员。

• 对于生产者进程和消费者进程：

1. 创建新的进程，若所创建的进程标识符小于 0，则创建进程失败，并输出相应提示。

2. 若此进程为子进程，将共享段附加到申请通信的进程空间，成功时返回共享内存附加到进程空间的虚地址。

3. 对于生产者进程和消费者进程的每个子操作，利用 P 操作实现，然后睡眠一段随机时间，之后往缓冲区中生产一个数据，并进行相关打印，然后执行 V 操作。

4. 将共享段与进程之前解除连接。

（3）代码实现

• 变量宏定义

```
//2个生产者，每个生产者工作6次
#define Need_Producer 2
#define Works_Producer 6
//3个消费者，每个消费者工作4次
#define Need_Customer 3
#define Works_Customer 4
//缓冲区为3
#define buffer_len 3
```

- 缓冲区结构

```
//缓冲区的结构 （循环队列）
struct buf
{
    int buffer[buffer_len];
    int write;
    int read;
};
```

其中 read 表示生产者的当前指针，write 表示消费者的当前指针。buffer 是一个长度为 buffer_len 的数组，1 表示有产品，0 表示无产品。

- P、V 操作

```
//P操作
void P(int sem_id, int sem_num)
{
    struct sembuf xx;
    xx.sem_num = sem_num; //信号量的索引
    xx.sem_op = -1; //信号量的操作值
    xx.sem_flg = 0; //访问标志
    semop(sem_id, &xx, 1); //一次需进行的操作的数组sembu
}

//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);
}
```

- 初始化操作

```
//创建一个信号量集合(信号量数为2)，返回值为信号量集合的标识号(关键字，信号量数，创建或打开的标志)
sem_id = semget(SEM_ALL_KEY, 2,IPC_CREAT|0660);
//对信号量执行控制操作(信号量集合标识，信号量的索引，要执行的操作命令，设置或返回信号量的参数)
semctl(sem_id, SEM_EMPTY, SETVAL, buffer_len);
semctl(sem_id, SEM_FULL, SETVAL, 0);

//申请一个共享内存区，成功返回为共享内存区的标识
shm_id = shmget(IPC_PRIVATE, MYBUF_LEN, SHM_MODE);
if (shm_id < 0)
{
    printf("Error on shmget.\n"); //申请共享内存区失败
    exit(1);
}
//将共享段附加到申请通信的进程空间；成功时返回共享内存附加到进程空间的虚地址，失败时返回-1
shmptr = shmat(shm_id, 0, 0);
```

• 创建子进程（以生产者子进程为例）

```
//2个生产者进程
while (num_Producer < Need_Producer)
{
   pid_p = fork();
   num_Producer++;
   if (pid_p < 0)
   {
      printf("Error on fork.\n"); //创建进程失败
      exit(1);
   }
```

• 显示当前时间

```
time(&now);
timenow = localtime(&now);
printf("%04d/%02d/%02d-%02d:%02d:%02d", timenow->tm_year+1900,
   timenow->tm_mon+1, timenow->tm_mday, timenow->tm_hour,
   timenow->tm_min, timenow->tm_sec);
```

• 生产者往缓冲区生产数据

```
shmptr->buffer[shmptr->write] = 1;
printf("\tProducer %d puts data at Position [%d]", num_Producer, shmptr->write + 1);
shmptr->write = (shmptr->write + 1) % buffer_len;
```

• 消费者从缓冲区消费数据

```
shmptr->buffer[shmptr->read] = 0;
printf("\tConsumer %d gets data from Position [%d]", num_Customer, shmptr->read + 1);
shmptr->read = (shmptr->read + 1) % buffer_len;
```

• 打印当前缓冲区数据情况

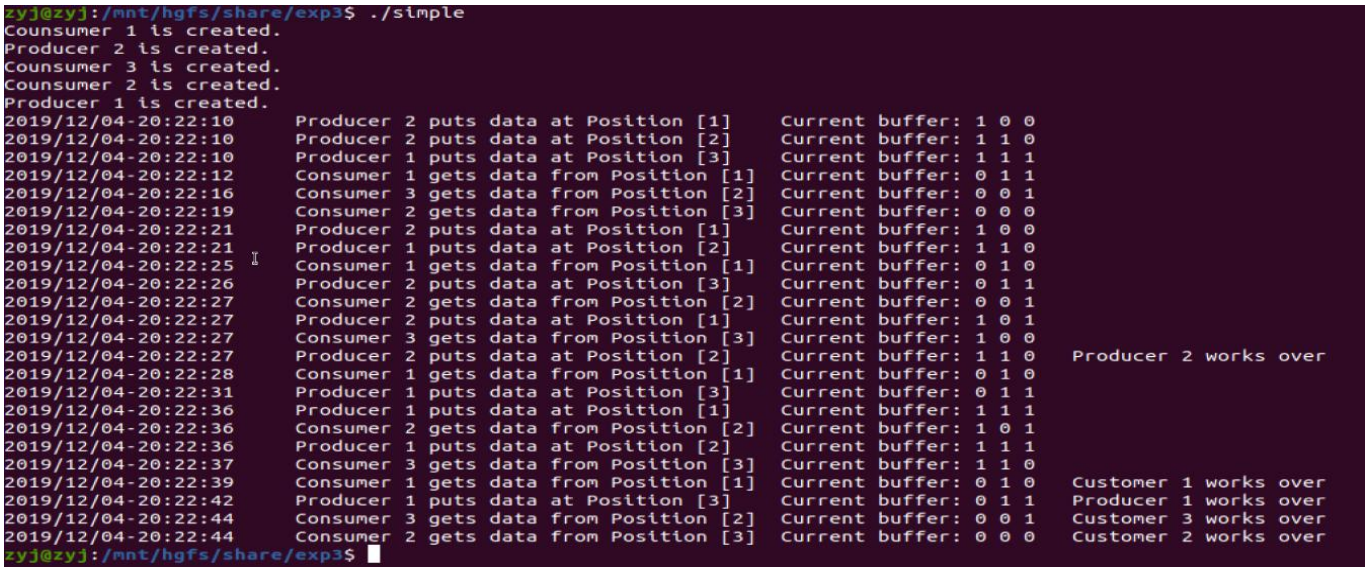
```
printf("\tCurrent buffer: ");
for (j = 0; j < buffer_len; j++)
   printf("%d ", shmptr->buffer[j]);
```

（4）实现效果

• 编译源文件

```
zyj@zyj:/mnt/hgfs/share/exp3$ gcc simple.c -o simple
zyj@zyj:/mnt/hgfs/share/exp3$
```

• 运行程序



同 Windows 运行结果一样，程序一开始输出两个生产者和三个消费者被产生，之后每次生产者生产数据或者消费者消费数据都打印一行，最开始是当前时间，之后表明具体操作，最后打印当前缓冲区的数据情况，0 表示没有数据，1 表示有数据。另外当生产者生产完 6 次数据或者消费者消费完 4 次数据，都打印一条语句表示对应的工作结束。

五、 实验收获与体会

这个实验总算做完了，花了相当多时间，主要花在寻找相关接口函数。总的来说，windows 系统和 linux 系统处理生产者消费者问题的过程类似，但是接口的相关性和调用参数有很大的不同，在前文中已有说明，而且 windows 所需要的的接口函数比 linux 多了很多，程序也更复杂。通过这个实验，掌握和理解了不同进程之间的通信机、利用信号量进行多进程之间互斥访问共享内存区域的控制，收获很大。