实验三 生产者消费者问题

徐恒达 1120151811

目录

1 实验目的

在 Windows 和 Linux 环境下利用进程模拟生产者和消费者，利用多进程同步机制完成生产者消费者问题。

2 实验内容

- 创建一个有 6 个缓冲区的缓冲池，初始为空，每个缓冲区能存放一个长度若为 10 个字符的字符串。
- 创建 2 个生产者进程：
  - 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 12 次。
- 创建 3 个消费者进程：
  - 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 8 次。
- 显示每次添加和读取数据式当前缓冲区状态。

3 实验环境

3.1 硬件配置

处理器：Intel(R) Core(TM) i3-4020Y CPU @ 1.5GHz
内存：4.0GB DDR3

3.2 Windwos 配置

Windows 10 Pro Version 1709 64bit

3.3 Linux 配置

Ubuntu 16.04 LTS

4 实验过程

4.1 实现思路

三个程序文件分别为 master、producer 和 consumer，由 master 程序创建 Master 进程，负责创建缓冲区，初始化信号量以及启动指定数量的生产者和消费者进程，分别由 producer 程序和 consumer 程序创建生产者和消费者进程，第 i 个生产者进程记为 $Producer_i$，第 i 个消费者进程记为 $Consumer_i$。

4.1.1 主进程

Algorithm 1 主程序

Require:

缓冲区数量 N_{buffer}

生产者数量 $N_{producer}$

消费者数量 $N_{consumer}$

创建 N_{buffer} 个共享缓冲区
创建空缓冲区信号量 $S_{empty} \leftarrow N_{buffer}$
创建满缓冲区信号量 $S_{full} \leftarrow 0$
创建互斥信号量 S_{mutex}
for $i = 1$ **to** $N_{producer}$ **do**
    创建生产者进程 $Producer_i$
end for
for $i = 1$ **to** $N_{consumer}$ **do**
    创建消费者进程 $Consumer_i$
end for
阻塞等待所有生产者和消费者进程结束
释放信号量 S_{empty}、S_{full} 和 S_{mutex}
释放所有缓冲区

4.1.2 生产者进程

Algorithm 2 生产者

Require: 单个生产者的生产次数 $M_{producer}$

1: 获取信号量 S_{empty}、S_{full} 和 S_{mutex}
2: **for** $i = 1$ **to** $M_{producer}$ **do**
3: P(S_{empty})
4: P(S_{mutex})
5: 打开缓冲区
6: 写入数据
7: 关闭缓冲区
8: V(S_{full})
9: V(S_{mutex})
10: **end for**
11: 关闭信号量 S_{empty}、S_{full} 和 S_{mutex}

4.1.3 消费者进程

Algorithm 3 消费者

Require: 单个消费者的消费次数 $M_{consumer}$

1: 获取信号量 S_{empty}、S_{full} 和 S_{mutex}
2: **for** $i = 1$ **to** $M_{consumer}$ **do**
3: P(S_{full})
4: P(S_{mutex})
5: 打开缓冲区
6: 读出数据
7: 关闭缓冲区
8: V(S_{empty})
9: V(S_{mutex})
10: **end for**
11: 关闭信号量 S_{empty}、S_{full} 和 S_{mutex}

4.2 Windows 实现

包括 4 个源程序文件：master.c 实现主进程，producer.c 实现生产者进程，consumer.c 实现消费者进程，另外用一个头文件 header.h 包含进程之间的公共信息，包括缓冲区数量，

缓冲区名字，消费者数量，生产者数量等。三个 C 源文件均包含 header.h 文件。

4.2.1 头文件

进程之间要共享的信息包含在 header.h 文件中，包括生产者和消费者程序文件的名字，缓冲区的名字，空信号量、满信号量和互斥锁的名字，不同的进程由这些名字就可以获取到相应的对象句柄。

还包括生产者消费者进程的数量以及它们各自生产和消费的次数。

```
#define PRODUCER_NAME "producer" // 生产者程序名
#define CONSUMER_NAME "consumer" // 消费者程序名

#define SHM_NAME "SharedMemory" // 缓冲区名
#define SEM_EMPTY_NAME "Empty" // 空信号量名
#define SEM_FULL_NAME "Full" // 满信号量名
#define MUTEX_NAME "Mutex"   // 互斥锁名

#define PRODUCER_NUM 2       // 生产者数量
#define PRODUCER_CNT 12      // 单个生产者生产次数
#define CONSUMER_NUM 3       // 消费者数量
#define CONSUMER_CNT 8       // 单个消费之消费次数
```

对单个缓冲区的定义为一个结构体，结构体中仅包含一个长度为 10 的字符型数组，命名为 Data。缓冲区的数量定义为 6 个。

生产者要写入和消费者要读出的数据定义为 12 个字符串，从”dataA” 到”dataL”，以便在程序执行过程中显示在控制台中。

```
typedef struct data {        // 缓冲区数据结构
    char s[10];
} Data;
#define DATA_NUM 6           // 缓冲区数量
char *msg[12] = {            // 生产或消费的数据
       "dataA", "dataB", "dataC", "dataD",
       "dataE", "dataF", "dataG", "dataH",
       "dataI", "dataJ", "dataK", "dataL"
};
```

此外还分别定义了生产者和消费者进程在缓冲区中停留的时间和在缓冲区外停留的时间，在缓冲区内停留一定时间来模拟进程读写数据的时间开销，在缓冲区外停留一定时间来

模拟进程消耗数据或产生数据需求的时间。

这里定义的数值单位为毫秒，并且表示停留时间的最大值。即若停留时间定义为 t，则实际停留时间为 0 到 t 毫秒之间的一个随机时间。

```
#define PRODUCER_IN_TIME 500 // 生产者在缓冲区内的最大停留时间
#define PRODUCER_OUT_TIME 800 // 生产者在缓冲区外的最大停留时间
#define CONSUMER_IN_TIME 1500 // 消费者在缓冲区内的最大停留时间
#define CONSUMER_OUT_TIME 2000 //消费者在缓冲区外的最大停留时间
```

4.2.2 主进程

创建缓冲区 主进程首先创建缓冲区并初始化。缓冲区的主体部分为DATA_NUM个Data结构体，但为了能够确定哪些缓冲区为满哪些为空，故将缓冲区域作为循环队列来使用，数据总是从队尾写入，从队头取出。因此在缓冲区开始位置额外开辟两个 int 整型变量的空间，前一个 int 变量存储队头元素索引值，后一个 int 变量存储队尾元素的下一个元素的索引值。每次写入数据或取出数据时队头索引或队尾索引加 1 指向下一个位置，超出范围则跳到 0 位置。因为有信号量的保证，这里的队头队尾指针便可以只负责指示位置，不用判断循环队列是否为空或满，这也意味着这里不必像传统的循环队列那样需要留出一个空间来区分全空或全满的状态，即DATA_NUM个缓冲区全部用来存储数据。

在 Windows 下，创建共享缓冲区的过程是先使用CreateFileMapping()函数创建一个文件映射对象，传入INVALID_HANDLE_VALUE参数表示不映射到具体的磁盘文件，而是映射到交换区页文件的一部分。然后使用MapViewOfFile()函数将共享内存区映射到进程地址空间，该函数返回一个指向共享内存区域首地址的指针，由这个指针就可以访问共享内存中的数据了。

为了便于查看缓冲区中的数据情况，如果缓冲区中没有数据就在其中写入字符串-----。初始化时将所有缓冲区均写入-----。最后使用UnmapViewOfFile()函数将共享内存区从进程地址空间接触映射。

```
void CreateBuffer() {
    DWORD dwMapSize = 2 * sizeof(int) // 缓冲区大小
                    + DATA_NUM * sizeof(Data);
    hMapFile = CreateFileMapping(   // 创建缓冲区
           INVALID_HANDLE_VALUE, NULL,
           PAGE_READWRITE, 0, dwMapSize, SHM_NAME);
    void *pBuf = MapViewOfFile(     // 加载到进程地址空间
           hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
    int *pInt = pBuf;
    Data *pData = pBuf + 2 * sizeof(int);
```

```
    pInt[0] = pInt[1] = 0;          // 初始化头尾指针
    for (int i = 0; i < DATA_NUM; i++) // 初始化数据
        strcpy(pData[i].s, "-----");
    UnmapViewOfFile(pBuf);          // 从进程地址空间卸载
}
```

创建同步对象 使用CreateSemaphore()函数创建信号量，使用CreateMutex()函数创建互斥锁。

```
void CreateSyncObjects() {
    hEmptySemaphore = CreateSemaphore( // 创建空缓冲区信号量
            NULL, DATA_NUM, DATA_NUM, SEM_EMPTY_NAME);
    hFullSemaphore = CreateSemaphore( // 创建空缓冲区信号量
            NULL, 0, DATA_NUM, SEM_FULL_NAME);
    hMutex = CreateMutex(           // 创建互斥锁
            NULL, FALSE, MUTEX_NAME);
}
```

主体流程 如上节伪代码中所示，Master 进程依次创建缓冲区，创建信号量和同步对象，启动消费者和生产者进程，等待进程结束最后释放资源。主体流程如下所示。

```
int main(int argc, char *argv[]) {
    CreateBuffer();                     // 创建缓冲区
    CreateSyncObjects();                // 创建同步信号量
    HANDLE hProcesses[PRODUCER_NUM + CONSUMER_NUM];
    int iProcessTop = 0;
    for (int i = 1; i <= PRODUCER_NUM; i++) { // 启动生产者进程
        HANDLE hProcess = StartProcess(PRODUCER_NAME, i);
        hProcesses[iProcessTop++] = hProcess;
    }
    for (int i = 1; i <= CONSUMER_NUM; i++) { // 启动消费者进程
        HANDLE hProcess = StartProcess(CONSUMER_NAME, i);
        hProcesses[iProcessTop++] = hProcess;
    }
    for (int i = 0; i < iProcessTop; i++) { // 等待子进程结束
        WaitForSingleObject(hProcesses[i], INFINITE);
```

```
        CloseHandle(hProcesses[i]);
    }
    CloseHandle(hEmptySemaphore);        // 关闭所有句柄
    CloseHandle(hFullSemaphore);
    CloseHandle(hMutex);
    CloseHandle(hMapFile)
    return 0;
}
```

其中创建子进程函数StartProcess()的实现如下。

```
HANDLE StartProcess(char *name, int id) {
    STARTUPINFO si;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    PROCESS_INFORMATION pi;
    ZeroMemory(&pi, sizeof(pi));
    char commandLine[10];  // 存储命令行参数
    sprintf(commandLine, "%s %d", name, id);
    CreateProcess(          // 创建子进程
           NULL, commandLine, NULL, NULL,
           FALSE, 0, NULL, NULL, &si, &pi);
    return pi.hProcess;     // 返回进程句柄
}
```

部分负责打印信息的代码没有给出，完整代码请见附录。

4.2.3 生产者

打开句柄 如上节伪代码中所述，生产者进程首先打开各种进程间公共对象，包括文件映射对象，信号量对象和互斥锁对象，分别使用使用函数OpenFileMapping()，OpenSemaphore()和OpenMutex()。

```
HANDLE hMapFile = OpenFileMapping( // 打开文件映射对象
    FILE_MAP_ALL_ACCESS, FALSE, SHM_NAME);
HANDLE hEmptySemaphore = OpenSemaphore( // 打开空缓冲区信号量
    SEMAPHORE_ALL_ACCESS, FALSE, SEM_EMPTY_NAME);
HANDLE hFullSemaphore = OpenSemaphore( // 打开满缓冲区信号量
    EMAPHORE_ALL_ACCESS, FALSE, SEM_FULL_NAME);
```

```
HANDLE hMutex = OpenMutex(          // 打开互斥锁
    MUTEX_ALL_ACCESS, FALSE, MUTEX_NAME);
```

P 操作 生产者每次写入数据前先对空缓冲区信号量进程 P 操作，再取得互斥锁的使用权，在 Windows 中均用函数`WaitForSingleObject()`实现。

```
WaitForSingleObject(hEmptySemaphore, INFINITE); // P(空信号量)
WaitForSingleObject(hMutex, INFINITE);     // P(互斥锁)
```

写数据 获得了缓冲区的写入权后，生产者使用`MapViewOfFile()`函数将共享缓冲区映射到自己的地址空间，函数返回缓冲区首地址。生产者读取尾指针，然后向尾指针指向的区域写入数据并将尾指针指向下一个缓冲区，最后使用`UnmapViewOfFile()`函数将共享内存从地址空间解除映射。

```
void *pBuf = MapViewOfFile(  // 加载共享内存区到进程地址空间
    hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
int *pInt = pBuf;
Data *pData = pBuf + 2 * sizeof(int);
int tail = pInt[1];           // 取出尾指针
strcpy(pData[tail].s, msg[i]); // 写入数据
tail = (tail + 1) % DATA_NUM; // 推进尾指针
pInt[1] = tail;               // 写会尾指针
UnmapViewOfFile(pBuf);        // 卸载共享内存区
```

V 操作 写入数据完成后，生产者对满信号量进行依次 V 操作，然后释放互斥锁，在 Windows 中分别使用函数`ReleaseSemaphore()`和`ReleaseMutex()`实现。

```
ReleaseSemaphore(hFullSemaphore, 1, NULL); // V(满信号量)
ReleaseMutex(hMutex);                      // V(互斥锁)
```

关闭句柄 最后使用`CloseHandle()`函数关闭所有句柄。

```
CloseHandle(hEmptySemaphore);
CloseHandle(hFullSemaphore);
CloseHandle(hMutex);
CloseHandle(hMapFile);
```

以上描述中略去了输出状态信息和暂停等待的语句，完整代码请见附件。

4.2.4 消费者

消费者进程的操作于生产者在流程上相同，只是把 P/V 操作为消费者的操作模式，即开始先对空信号量进行 P 操作，读出数据后对满信号量进行一次 V 操作。

```
WaitForSingleObject(hFullSemaphore, INFINITE); // P(满信号量)
WaitForSingleObject(hMutex, INFINITE);    // P(互斥锁)

void *pBuf = MapViewOfFile(               // 加载共享内存区
       hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
int *pInt = pBuf;
Data *pData = pBuf + 2 * sizeof(int);
int head = pInt[0];
printf("%s", pData[head].s);              // 读数据
strcpy(pData[head].s, "-----")
head = (head + 1) % DATA_NUM;
pInt[0] = head;
UnmapViewOfFile(pBuf);                    // 卸载共享内存区

ReleaseSemaphore(hEmptySemaphore, 1, NULL); // V(空信号量)
ReleaseMutex(hMutex);                     // V(互斥锁)
```

完整代码请见附录。

4.3 Linux 实现

Linux 实现的框架于 Windows 相同，4 个源文件，master.c 实现主进程，producer.c 实现生产者进程，consumer.c 实现消费者进程，header.h 包含进程间的公共信息。三个 C 文件均包好 header.h 文件。

4.3.1 头文件

头文件 header.h 中记录了生产者消费者进程名，数量，生产或消费次数，缓冲区定义，缓冲区数量等等信息，于 Windows 基本一致。

```
#define PRODUCER_NAME "producer" // 生产者程序名
#define CONSUMER_NAME "consumer" // 消费者程序名

#define PRODUCER_NUM 2           // 生产者数量
#define PRODUCER_CNT 12          // 单个生产者生产次数
```

```
#define CONSUMER_NUM 3          // 消费者数量
#define CONSUMER_CNT 8          // 单个消费者消费次数

#define SEM_KEY 2018            // 信号量集 ID
#define SEM_NUM 3               // 信号量数目
#define SEM_EMPTY 0             // 空信号量 ID
#define SEM_FULL 1              // 满信号量 ID
#define SEM_MUTEX 2             // 互斥信号量 ID
#define SHM_KEY 2019            // 共享内存区 ID

struct data {                   // 缓冲区数据结构
    char s[10];
};
#define DATA_NUM 6              // 缓冲区数量

char *msg[12] = {               // 生产或消费的数据
    "dataA", "dataB", "dataC", "dataD",
    "dataE", "dataF", "dataG", "dataH",
    "dataI", "dataJ", "dataK", "dataL"
};

#define PRODUCER_IN_TIME 500 // 生产者在缓冲区内的最大停留时间
#define PRODUCER_OUT_TIME 800 // 生产者在缓冲区外的最大停留时间
#define CONSUMER_IN_TIME 1500 // 消费者在缓冲区内的最大停留时间
#define CONSUMER_OUT_TIME 2000 //消费者在缓冲区外的最大停留时间
```

4.3.2 主进程

创建缓冲区 缓冲区的结构于 Windows 相同，两个 int 整型变量加DATA_NUM个data结构体，缓冲区作为循环队列使用，两个 int 变量作为头尾指针。

Linux 中使用shmget()函数创建共享内存，shmat()函数将共享内存映射到进程地址空间，初始化每个缓冲区中为字符串"-----"，最后使用shmdt()函数卸载缓冲区。

```
int shminit() {                     // 创建共享内存区区函数
size_t size = 2 * sizeof(int) +     // 共享内存区大小
              DATA_NUM * sizeof(struct data);
int shm_id = shmget(                // 创建共享内存区
```

```
        SHM_KEY, size, IPC_CREAT | 0666);
void *pBuf = shmat(shm_id, NULL, 0); // 加载共享内存区
int *pInt = pBuf;
struct data *pData = pBuf + 2 * sizeof(int);
pInt[0] = pInt[1] = 0;              // 初始化头尾指针
for (int i = 0; i < DATA_NUM; i++) // 初始化数据
    strcpy(pData[i].s, "-----");
shmdt(pBuf);                        // 卸载共享内存区
return shm_id;                      // 返回共享内存区ID
}
```

创建信号量 Linux 中使用semget()函数创建信号量集合，程序中创建SEM_NUM也就是 3 个信号量，将空信号量初始值设为 3，满信号量初始化为 0，互斥信号量初始化为 1。

```
int seminit() {
    int sem_id = semget(SEM_KEY, SEM_NUM, IPC_CREAT | 0660);
                                    // 创建信号量集合
    semctl(sem_id, SEM_EMPTY, SETVAL, DATA_NUM);
                                    // 初始化空缓冲区信号量
    semctl(sem_id, SEM_FULL, SETVAL, 0);
                                    // 初始化满缓冲区信号量
    semctl(sem_id, SEM_MUTEX, SETVAL, 1);
                                    // 初始化互斥锁信号量
    return sem_id;
}
```

主体流程 Master 进程依次创建缓冲区，创建信号量和同步对象，启动消费者和生产者进程，等待进程结束最后释放资源，如下所示。

```
int main(int argc, char *argv[]) {
    int shm_id = shminit();             // 创建化共享内存区
    int sem_id = seminit();             // 创建信号量集合
    for (int i = 1; i <= PRODUCER_NUM; i++) // 创建生产者进程
        start(PRODUCER_NAME, i);
    for (int i = 1; i <= CONSUMER_NUM; i++) // 创建消费者进程
        start(CONSUMER_NAME, i);
```

```
    for (int i = 0; i < PRODUCER_NUM + CONSUMER_NUM; i++)
        wait(NULL);                    // 等待所有子进程结束
    semctl(sem_id, 0, IPC_RMID);       // 释放信号量集合
    shmctl(shm_id, IPC_RMID, 0);       // 释放共享内存区
    return 0;
}
```

其中创建子进程函数 start 的实现如下。

```
void start(char *name, int id) {
    if (fork() == 0) {
        char index[5];
        sprintf(index, "%d", id);
        execl(name, name, index, NULL);
    }
}
```

以上描述中略去了输出状态信息和暂停等待的语句，完整代码请见附件。

4.3.3 生产者

打开缓冲区和同步对象 生产者进程从头文件 header.h 中获取共享内存和信号量集合的名字，然后使用函数`semget()`获取信号量集合，使用函数`shmget()`获取共享内存区。

```
int sem_id = semget(SEM_KEY, SEM_NUM, 0666); // 获取信号量集合
size_t size = 2 * sizeof(int) +          // 共享内存区大小
              DATA_NUM * sizeof(struct data);
int shm_id = shmget(SHM_KEY, size, 0);   // 获取共享内存区
```

主体流程 生产者进程首先对空信号量和互斥信号量进行 P 操作，然后加载共享内存区，读取尾指针，向队尾写入数据，队尾指针前进，卸载缓冲区，最后对满信号量和互斥锁各进行一次 V 操作。

为了代码清晰，程序中将信号量的 P 操作和 V 操作各自单独定义了一个子程序。

```
semp(sem_id, SEM_EMPTY);           // P(空缓冲区信号量)
semp(sem_id, SEM_MUTEX);           // P(互斥锁信号量)

void *pBuf = shmat(shm_id, NULL, 0); // 加载共享内存区
int *pInt = pBuf;
```

```
struct data *pData = pBuf + 2 * sizeof(int);
int tail = pInt[1];                 // 读取队尾指针
strcpy(pData[tail].s, msg[i]);      // 写如数据
tail = (tail + 1) % DATA_NUM;       // 推进队尾指针
pInt[1] = tail;                     // 写会队尾指针
shmdt(pBuf);                        // 卸载共享内存区

semv(sem_id, SEM_FULL);             // V(满缓冲区信号量)
semv(sem_id, SEM_MUTEX);            // V(互斥锁信号量)
```

P/V 操作 要对某个信号量进行操作，需要把信号量名称和要修改的值写入一个sembuf结构体中，然后传入semop()函数执行操作。进行 P 操作要修改的值就为-1，V 操作要修改的值就为 1。

```
void semp(int sem_id, ushort sem_num) { // P操作
    struct sembuf sop = {sem_num, -1, 0}; // 信号量值-1
    semop(sem_id, &sop, 1);
}
void semv(int sem_id, ushort sem_num) { // V操作
    struct sembuf sop = {sem_num, 1, 0}; // 信号量值+1
    semop(sem_id, &sop, 1);
}
```

完整代码详见附件。

4.3.4 消费者

消费者进程的代码与生产者基本一致，只是调整 P/V 操作的顺序并将读操作改为写操作。核心代码如下所示。完整程序见附录。

```
semp(sem_id, SEM_FULL);             // P(空缓冲区信号量)
semp(sem_id, SEM_MUTEX);            // P(互斥锁信号量)

void *pBuf = shmat(shm_id, NULL, 0);
int *pInt = pBuf;
struct data *pData = pBuf + 2 * sizeof(int);
int head = pInt[0];
printf("%s", pData[head].s);        // 读取数据
```

```
strcpy(pData[head].s, "-----");
head = (head + 1) % DATA_NUM;
pInt[0] = head;
shmdt(pBuf);

semv(sem_id, SEM_EMPTY);            // V(满缓冲区信号量)
semv(sem_id, SEM_MUTEX);            // V(互斥锁信号量)
```

5 实验结果

5.1 编译生成

编译主程序，生产者程序和消费者程序的 makefile 文件如下所示。其中为了在一次 make 过程中同时生成三个目标文件，将三个目标 master，producer 和 consumer 共同指向虚（phony）根节点 all。Windows 下和 Linux 下的 makefile 均是如此。

```
all: master producer consumer
.phony: all
master: master.c header.h
    gcc master.c -o master
producer: producer.c header.h
    gcc producer.c -o producer
consumer: consumer.c header.h
    gcc consumer.c -o consumer
```

5.2 实验输出

在命令行中启动 master，master 进程初始化资源，分别启动生产者进程和消费者进程，输出结果如下所示。Windows 下的输出格式和 Linux 下的完全相同，这里只取一次运行的结果展示说明。

输出中每一行表示一次动作，第一列表示生产者进程的动作，共创建了 2 个生产者进程，第一个生产者用 Producer1 表示，第二个用 Producer2 表示，=> 表示像缓冲区写入数据；第二列表示消费者的动作，三个消费者进程分别用 Consumer1，Consumer2 和 Consumer3 表示，<= 表示从缓冲区取走数据；第三列显示完成此次操作后缓冲区的实时状态，—表示对应缓冲区为空。

从中可以清晰地看出缓冲区作为循环队列数据写入和读出的情况。因为进程间推进的异步性和等待时间的随机性，每一次运行的输出顺序会有不同，但不管进程间如何执行，都

能保证开始和最后缓冲区一定为空，而且不会出现当缓冲区为空消费者还在取数据或当缓冲区已满生产者还在写数据的情况，也不会出现生产者写入数据覆盖和消费者重复取数据的情况。程序已经成功得实现了生产者消费者问题。

```
Producer           Consumer           Buffer
------------------ ------------------ ----------------------------------
Producer1 => dataA                    dataA ----- ----- ----- ----- -----
Producer2 => dataA                    dataA dataA ----- ----- ----- -----
                   Consumer1 <= dataA ----- dataA ----- ----- ----- -----
Producer1 => dataB                    ----- dataA dataB ----- ----- -----
                   Consumer2 <= dataA ----- ----- dataB ----- ----- -----
Producer2 => dataB                    ----- ----- dataB dataB ----- -----
                   Consumer3 <= dataB ----- ----- ----- dataB ----- -----
Producer1 => dataC                    ----- ----- ----- dataB dataC -----
Producer2 => dataC                    ----- ----- ----- dataB dataC dataC
Producer1 => dataD                    dataD ----- ----- dataB dataC dataC
                   Consumer1 <= dataB dataD ----- ----- ----- dataC dataC
Producer2 => dataD                    dataD dataD ----- ----- dataC dataC
Producer1 => dataE                    dataD dataD dataE ----- dataC dataC
                   Consumer2 <= dataC dataD dataD dataE ----- ----- dataC
                   Consumer3 <= dataC dataD dataD dataE ----- ----- -----
Producer2 => dataE                    dataD dataD dataE dataE ----- -----
                   Consumer1 <= dataD ----- dataD dataE dataE ----- -----
Producer1 => dataF                    ----- dataD dataE dataE dataF -----
Producer2 => dataF                    ----- dataD dataE dataE dataF dataF
                   Consumer2 <= dataD ----- ----- dataE dataE dataF dataF
                   Consumer3 <= dataE ----- ----- ----- dataE dataF dataF
Producer1 => dataG                    dataG ----- ----- dataE dataF dataF
                   Consumer1 <= dataE dataG ----- ----- ----- dataF dataF
Producer2 => dataG                    dataG dataG ----- ----- dataF dataF
                   Consumer2 <= dataF dataG dataG ----- ----- ----- dataF
Producer1 => dataH                    dataG dataG dataH ----- ----- dataF
                   Consumer3 <= dataF dataG dataG dataH ----- ----- -----
Producer2 => dataH                    dataG dataG dataH dataH ----- -----
                   Consumer1 <= dataG ----- dataG dataH dataH ----- -----
                   Consumer2 <= dataG ----- ----- dataH dataH ----- -----
Producer1 => dataI                    ----- ----- dataH dataH dataI -----
```

```
                  Consumer3 <= dataH ----- ----- ----- dataH dataI -----
Producer2 => dataI                   ----- ----- ----- dataH dataI dataI
                  Consumer1 <= dataH ----- ----- ----- ----- dataI dataI
Producer1 => dataJ                   dataJ ----- ----- ----- dataI dataI
                  Consumer2 <= dataI dataJ ----- ----- ----- ----- dataI
Producer2 => dataJ                   dataJ dataJ ----- ----- ----- dataI
                  Consumer1 <= dataI dataJ dataJ ----- ----- ----- -----
                  Consumer3 <= dataJ ----- dataJ ----- ----- ----- -----
Producer1 => dataK                   ----- dataJ dataK ----- ----- -----
                  Consumer2 <= dataJ ----- ----- dataK ----- ----- -----
Producer2 => dataK                   ----- ----- dataK dataK ----- -----
                  Consumer3 <= dataK ----- ----- ----- dataK ----- -----
Producer1 => dataL                   ----- ----- ----- dataK dataL -----
                  Consumer1 <= dataK ----- ----- ----- ----- dataL -----
Producer2 => dataL                   ----- ----- ----- ----- dataL dataL
                  Consumer2 <= dataL ----- ----- ----- ----- ----- dataL
                  Consumer3 <= dataL ----- ----- ----- ----- ----- -----
```

6 心得体会

通过本次实验，我简单了解了部分 WIN32、POSIX/System V 下的进程间通信技术，并亲手实现了简单的进程间同步或互斥访问共享主存区。

生产者消费者问题是进程控制这一个大内容下的经典问题，虽然思路非常简单，但是在实现的过程中，还是会存在非常多的细节问题，比如信号量的访问顺序、开关量、共享主存区的访问权限等等。

这一问题在现实生活中的意义非常大，可以帮助我们在将来工作中更好的并行完成多项任务，并且在进行任务的时候，通过设定标记来更好的完成各项复杂的工作，能够有条不紊。

此次实验所使用的各项技术还只是进程管理的初步，要想实现和 Windows 和 Linux 一样的强大的进程管理控制，还需要学习更多知识。

A 附录

A.1 Windows 版本源代码

A.1.1 header.h

```
#ifndef PRODUCERCONSUMER_HEADER_H
#define PRODUCERCONSUMER_HEADER_H

#define PRODUCER_NAME "producer"
#define CONSUMER_NAME "consumer"

#define PRODUCER_NUM 2
#define PRODUCER_CNT 12
#define CONSUMER_NUM 3
#define CONSUMER_CNT 8

#define SHM_NAME "SharedMemory"
#define SEM_EMPTY_NAME "Empty"
#define SEM_FULL_NAME "Full"
#define MUTEX_NAME "Mutex"

typedef struct Data
{
    char s[11];
} Data;
#define DATA_NUM 6

char *msg[12] = {
       "dataA", "dataB", "dataC", "dataD",
       "dataE", "dataF", "dataG", "dataH",
       "dataI", "dataJ", "dataK", "dataL"
};

#define PRODUCER_IN_TIME 500
#define PRODUCER_OUT_TIME 800
#define CONSUMER_IN_TIME 1500
```

```
#define CONSUMER_OUT_TIME 2000

#endif //PRODUCERCONSUMER_HEADER_H
```

A.1.2 master.c

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#include "header.h"

HANDLE hMapFile;
HANDLE hEmptySemaphore;
HANDLE hFullSemaphore;
HANDLE hMutex;

void PrintHeader()
{
    char *head[] = {"Producer", "Consumer", "Buffer"};
    int width[] = {18, 18, 6 * DATA_NUM - 1};
    int num = 3;
    for (int i = 0; i < num; i++)
    {
        char fmt[10];
        sprintf(fmt, "%%-%ds", width[i]);
        printf(fmt, head[i]);
        printf(" ");
    }
    printf("\n");
    for (int i = 0; i < num; i++)
    {
        for (int j = 0; j < width[i]; j++) printf("-");
        printf(" ");
    }
    printf("\n");
}
```

```
void CreateBuffer()
{
    DWORD dwMapSize = 2 * sizeof(int)
                    + DATA_NUM * sizeof(Data);
    hMapFile = CreateFileMapping(
           INVALID_HANDLE_VALUE, NULL,
           PAGE_READWRITE, 0, dwMapSize, SHM_NAME);
    void *pBuf = MapViewOfFile(
           hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
    int *pInt = pBuf;
    Data *pData = pBuf + 2 * sizeof(int);

    pInt[0] = pInt[1] = 0;
    for (int i = 0; i < DATA_NUM; i++)
       strcpy(pData[i].s, "-----");
    UnmapViewOfFile(pBuf);
}

void CreateSyncObjects()
{
    hEmptySemaphore = CreateSemaphore(
           NULL, DATA_NUM, DATA_NUM, SEM_EMPTY_NAME);
    hFullSemaphore = CreateSemaphore(
           NULL, 0, DATA_NUM, SEM_FULL_NAME);
    hMutex = CreateMutex(
           NULL, FALSE, MUTEX_NAME);
}

HANDLE StartProcess(char *name, int id)
{
    STARTUPINFO si;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);

    PROCESS_INFORMATION pi;
```

```
    ZeroMemory(&pi, sizeof(pi));

    char commandLine[10];
    sprintf(commandLine, "%s %d", name, id);

    BOOL bSucceed = CreateProcess(
           NULL, commandLine, NULL, NULL,
           FALSE, 0, NULL, NULL, &si, &pi);
    if (!bSucceed)
    {
        printf("Error: Master: Create process '%s' failed!\n",
            name);
        exit(1);
    }
    return pi.hProcess;
}

void CloseHandles()
{
    CloseHandle(hEmptySemaphore);
    CloseHandle(hFullSemaphore);
    CloseHandle(hMutex);
    CloseHandle(hMapFile);
}

int main(int argc, char *argv[])
{
    CreateBuffer();
    CreateSyncObjects();
    PrintHeader();

    HANDLE hProcesses[PRODUCER_NUM + CONSUMER_NUM];
    int iProcessTop = 0;
    for (int i = 1; i <= PRODUCER_NUM; i++)
    {
        HANDLE hProcess = StartProcess(PRODUCER_NAME, i);
```

```
            hProcesses[iProcessTop++] = hProcess;
        }
        for (int i = 1; i <= CONSUMER_NUM; i++)
        {
            HANDLE hProcess = StartProcess(CONSUMER_NAME, i);
            hProcesses[iProcessTop++] = hProcess;
        }

        for (int i = 0; i < iProcessTop; i++)
        {
            WaitForSingleObject(hProcesses[i], INFINITE);
            CloseHandle(hProcesses[i]);
        }
        CloseHandles();
        return 0;
    }
```

A.1.3 producer.c

```
    #include <stdio.h>
    #include <string.h>
    #include <stdlib.h>
    #include <time.h>
    #include <windows.h>
    #include "header.h"

    int main(int argc, char *argv[])
    {
        int id = 0;
        if (argc > 1) id = strtol(argv[1], NULL, 10);
        srand(time(NULL));

        HANDLE hMapFile = OpenFileMapping(
               FILE_MAP_ALL_ACCESS, FALSE, SHM_NAME);

        HANDLE hEmptySemaphore = OpenSemaphore(
```

```
        SEMAPHORE_ALL_ACCESS, FALSE, SEM_EMPTY_NAME);
HANDLE hFullSemaphore = OpenSemaphore(
        SEMAPHORE_ALL_ACCESS, FALSE, SEM_FULL_NAME);
HANDLE hMutex = OpenMutex(
        MUTEX_ALL_ACCESS, FALSE, MUTEX_NAME);

for (int i = 0; i < PRODUCER_CNT; i++)
{
    WaitForSingleObject(hEmptySemaphore, INFINITE);
    WaitForSingleObject(hMutex, INFINITE);

    void *pBuf = (LPTSTR)MapViewOfFile(
           hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
    int *pInt = pBuf;
    Data *pData = pBuf + 2 * sizeof(int);

    Sleep(rand() % (PRODUCER_IN_TIME + 1));
    int tail = pInt[1];
    strcpy(pData[tail].s, msg[i]);
    printf("Producer%d => %s ", id, pData[tail].s);
    tail = (tail + 1) % DATA_NUM;
    pInt[1] = tail;

    for (int j = 0; j < 18; j++) printf(" ");
    printf("  ");
    for (int j = 0; j < DATA_NUM; j++)
        printf("%s ", pData[j].s);
    printf("\n");
    UnmapViewOfFile(pBuf);

    ReleaseSemaphore(hFullSemaphore, 1, NULL);
    ReleaseMutex(hMutex);

    Sleep(rand() % (PRODUCER_OUT_TIME + 1));
}
```

```
    CloseHandle(hEmptySemaphore);
    CloseHandle(hFullSemaphore);
    CloseHandle(hMutex);
    CloseHandle(hMapFile);
    return 0;
}
```

A.1.4 consumer.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>
#include "header.h"

int main(int argc, char *argv[])
{
    int id = 0;
    if (argc > 1) id = strtol(argv[1], NULL, 10);
    srand(time(NULL));

    HANDLE hMapFile = OpenFileMapping(
           FILE_MAP_ALL_ACCESS, FALSE, SHM_NAME);

    HANDLE hEmptySemaphore = OpenSemaphore(
           SEMAPHORE_ALL_ACCESS, FALSE, SEM_EMPTY_NAME);
    HANDLE hFullSemaphore = OpenSemaphore(
           SEMAPHORE_ALL_ACCESS, FALSE, SEM_FULL_NAME);
    HANDLE hMutex = OpenMutex(
           MUTEX_ALL_ACCESS, FALSE, MUTEX_NAME);

    for (int i = 0; i < CONSUMER_CNT; i++)
    {
        WaitForSingleObject(hFullSemaphore, INFINITE);
        WaitForSingleObject(hMutex, INFINITE);
```

```
            void *pBuf = MapViewOfFile(
                   hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
            int *pInt = pBuf;
            Data *pData = pBuf + 2 * sizeof(int);

            Sleep(rand() % (CONSUMER_IN_TIME + 1));
            int head = pInt[0];
            for (int j = 0; j < 21; j++) printf(" ");
            printf("Consumer%d <= %s ", id, pData[head].s);
            strcpy(pData[head].s, "-----");
            head = (head + 1) % DATA_NUM;
            pInt[0] = head;

            for (int j = 0; j < DATA_NUM; j++)
                printf("%s ", pData[j].s);
            printf("\n");
            UnmapViewOfFile(pBuf);

            ReleaseSemaphore(hEmptySemaphore, 1, NULL);
            ReleaseMutex(hMutex);

            Sleep(rand() % (CONSUMER_OUT_TIME + 1));
        }

        CloseHandle(hEmptySemaphore);
        CloseHandle(hFullSemaphore);
        CloseHandle(hMutex);
        CloseHandle(hMapFile);
        return 0;
    }
```

A.1.5 makefile

```
all: master producer consumer
.phony: all
```

```
master: master.c header.h
    gcc master.c -o master
producer: producer.c header.h
    gcc producer.c -o producer
consumer: consumer.c header.h
    gcc consumer.c -o consumer
```

A.2 Linux 版本源代码

A.2.1 header.h

```
#ifndef PRODUCERCONSUMER_HEADER_H
#define PRODUCERCONSUMER_HEADER_H

#define PRODUCER_NAME "producer"
#define CONSUMER_NAME "consumer"

#define PRODUCER_NUM 2
#define PRODUCER_CNT 12
#define CONSUMER_NUM 3
#define CONSUMER_CNT 8

#define SEM_KEY 2018
#define SEM_NUM 3
#define SEM_EMPTY 0
#define SEM_FULL 1
#define SEM_MUTEX 2

#define SHM_KEY 2019

struct data
{
    char s[8];
};
#define DATA_NUM 6
```

```
char *msg[12] = {
    "dataA", "dataB", "dataC", "dataD",
    "dataE", "dataF", "dataG", "dataH",
    "dataI", "dataJ", "dataK", "dataL"
};

#define PRODUCER_IN_TIME 500
#define PRODUCER_OUT_TIME 800
#define CONSUMER_IN_TIME 1500
#define CONSUMER_OUT_TIME 2000

#endif //PRODUCERCONSUMER_HEADER_H
```

A.2.2 master.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include "header.h"

void showheader()
{
    char *head[] = {"Producer", "Consumer", "Buffer"};
    int width[] = {18, 18, 6 * DATA_NUM - 1};
    int num = 3;
    for (int i = 0; i < num; i++)
    {
        char fmt[10];
        sprintf(fmt, "%%-%ds", width[i]);
        printf(fmt, head[i]);
        printf(" ");
    }
```

```
    printf("\n");
    for (int i = 0; i < num; i++)
    {
        for (int j = 0; j < width[i]; j++) printf("-");
        printf(" ");
    }
    printf("\n");
}

int shminit()
{
    size_t size = 2 * sizeof(int) + DATA_NUM * sizeof(struct
        data);
    int shm_id = shmget(SHM_KEY, size, IPC_CREAT | 0666);
    void *pBuf = shmat(shm_id, NULL, 0);
    int *pInt = pBuf;
    struct data *pData = pBuf + 2 * sizeof(int);

    pInt[0] = pInt[1] = 0;
    for (int i = 0; i < DATA_NUM; i++)
        strcpy(pData[i].s, "-----");

    shmdt(pBuf);
    return shm_id;
}

int seminit()
{
    int sem_id = semget(SEM_KEY, SEM_NUM, IPC_CREAT | 0660);
    semctl(sem_id, SEM_EMPTY, SETVAL, DATA_NUM);
    semctl(sem_id, SEM_FULL, SETVAL, 0);
    semctl(sem_id, SEM_MUTEX, SETVAL, 1);
    return sem_id;
}

void start(char *name, int id)
```

```
{
    if (fork() == 0)
    {
        char index[5];
        sprintf(index, "%d", id);
        int s = execl(name, name, index, NULL);
        if (s == -1)
        {
            printf("Exec %s failed!\n", name);
            exit(1);
        }
    }
}

int main(int argc, char *argv[])
{
    int shm_id = shminit();
    int sem_id = seminit();
    showheader();

    for (int i = 1; i <= PRODUCER_NUM; i++)
    {
        start(PRODUCER_NAME, i);
    }
    for (int i = 1; i <= CONSUMER_NUM; i++)
    {
        start(CONSUMER_NAME, i);
    }

    for (int i = 0; i < PRODUCER_NUM + CONSUMER_NUM; i++)
    {
        wait(NULL);
    }

    semctl(sem_id, 0, IPC_RMID);
    shmctl(shm_id, IPC_RMID, 0);
```

```
    return 0;
}
```

A.2.3 producer.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "header.h"

void semp(int sem_id, ushort sem_num)
{
    struct sembuf sop = {sem_num, -1, 0};
    semop(sem_id, &sop, 1);
}

void semv(int sem_id, ushort sem_num)
{
    struct sembuf sop = {sem_num, 1, 0};
    semop(sem_id, &sop, 1);
}

int main(int argc, char *argv[])
{
    int id = 0;
    if (argc > 1) id = strtol(argv[1], NULL, 10);
    srand(time(NULL));

    int sem_id = semget(SEM_KEY, SEM_NUM, 0666);
    size_t size = 2 * sizeof(int) + DATA_NUM * sizeof(struct
        data);
    int shm_id = shmget(SHM_KEY, size, 0);
```

```
    for (int i = 0; i < PRODUCER_CNT; i++)
    {
        semp(sem_id, SEM_EMPTY);
        semp(sem_id, SEM_MUTEX);

        void *pBuf = shmat(shm_id, NULL, 0);
        int *pInt = pBuf;
        struct data *pData = pBuf + 2 * sizeof(int);

        int tail = pInt[1];
        strcpy(pData[tail].s, msg[i]);
        printf("Producer%d => %s ", id, pData[tail].s);
        tail = (tail + 1) % DATA_NUM;
        pInt[1] = tail;


        for (int j = 0; j < 18; j++) printf(" ");
        printf("  ");
        for (int j = 0; j < DATA_NUM; j++)
            printf("%s ", pData[j].s);
        printf("\n");

        usleep(rand() % (PRODUCER_IN_TIME + 1) * 1000);
        shmdt(pBuf);

        semv(sem_id, SEM_FULL);
        semv(sem_id, SEM_MUTEX);

        usleep(rand() % (PRODUCER_OUT_TIME + 1) * 1000);
    }
    return 0;
}
```

A.2.4 consumer.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "header.h"

void semp(int sem_id, ushort sem_num)
{
    struct sembuf sop = {sem_num, -1, 0};
    semop(sem_id, &sop, 1);
}

void semv(int sem_id, ushort sem_num)
{
    struct sembuf sop = {sem_num, 1, 0};
    semop(sem_id, &sop, 1);
}

int main(int argc, char *argv[])
{
    int id = 0;
    if (argc > 1) id = strtol(argv[1], NULL, 10);
    srand(time(NULL));

    int sem_id = semget(SEM_KEY, SEM_NUM, 0600);
    size_t size = 2 * sizeof(int) + DATA_NUM * sizeof(struct
        data);
    int shm_id = shmget(SHM_KEY, size, 0);

    for (int i = 0; i < CONSUMER_CNT; i++)
    {
        semp(sem_id, SEM_FULL);
        semp(sem_id, SEM_MUTEX);
```

```
        void *pBuf = shmat(shm_id, NULL, 0);
        int *pInt = pBuf;
        struct data *pData = pBuf + 2 * sizeof(int);
        usleep(rand() % (CONSUMER_IN_TIME + 1) * 1000);

        int head = pInt[0];
        for (int j = 0; j < 21; j++) printf(" ");
        printf("Consumer%d <= %s ", id, pData[head].s);
        strcpy(pData[head].s, "-----");
        head = (head + 1) % DATA_NUM;
        pInt[0] = head;

        for (int j = 0; j < DATA_NUM; j++)
            printf("%s ", pData[j].s);
        printf("\n");
        shmdt(pBuf);

        semv(sem_id, SEM_EMPTY);
        semv(sem_id, SEM_MUTEX);

        usleep(rand() % (CONSUMER_OUT_TIME + 1) * 1000);
    }
    return 0;
}
```

A.2.5 makefile

```
all: master producer consumer
.phony: all
master: master.c header.h
    gcc master.c -o master
producer: producer.c header.h
    gcc producer.c -o producer
consumer: consumer.c header.h
    gcc consumer.c -o consumer
```