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： 宋尚儒/1120180717

一、 实验目的

学习生产者与消费者的运行基本原理，学习使用共享内存区和信号量机制，学习使用多进程以及进程间通信的方法，学会使用锁互斥访问对象。

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（指缓冲区里的具体内容）
- 生产者和消费者用进程模拟（不能用线程）
- Linux/Window 下都需要做

三、 实验环境

Windows10

VMWare15.5

Ubuntu-20.04

四、 程序设计与实现

通用的简要设计思路：

(1) 主进程负责创建公共信号量和共享内存区，以及创建并等待 2 个生产者子进程和 3 个消费者子进程

(2) 共享内存区的数据结构内需要包括大小为 3 的缓冲区数组 a、缓冲区数据头部指针 beg、缓冲区数据尾部指针 end

(3) 公共信号量共需 3 个，分别是：

同步信号量 fill，指示已填充的缓冲区个数，初值为 0

同步信号量 empty，指示为空的缓冲区个数，初值为 3

互斥信号量 rw，控制子进程互斥地使用缓冲区，初值为 1

(4) 生产者子进程需要对共享内存区进行读写操作并在缓冲区内填入数据，这还涉及到信号量的申请与释放，其具体每次生产执行过程可以抽象为

```
P(empty);        //申请空闲缓冲区资源
P(rw);           //申请对缓冲区修改的权限
array[end]=x     //在空缓冲区填入数据
end=(end+1)%3    //尾部指针递增
V(fill);         //填充了一个缓冲区，释放填充缓冲区信号量
V(rw);           //释放修改权限
```

(5) 消费者子进程需要对共享内存区进行读写操作并在缓冲区内读出数据，其每次消费具体执行过程可以抽象为

```
P(fill);         //申请已填充缓冲区资源
P(rw);           //申请对缓冲区修改的权限
y=array[beg]     //从已填充缓冲区读取数据
beg=(beg+1)%3    //头部指针递增
V(empty);        //读取了一个缓冲区，释放空缓冲区信号量
V(rw);           //释放修改权限
```

(6) 生产者消费者子进程全部运行结束后，主进程需要回收共享内存区和信号量

接下来开始介绍两个操作系统更具体的设计实现思路，为方便查看，先说明

一下通用的宏定义

```
#define Cnt_Producer 2
#define Rep_Producer 6
#define Cnt_Consumer 3
#define Rep_Consumer 4
#define Cnt_Process 5
#define Len_buffer 3
```

1. 在 Windows 下实现：

(1) 主进程需要负责创建各对象，首先需要创建共享内存区，具体创建方式会在之后结合共享内存区的结构体详细说明。共享内存区创建完成后，需要创建信号量，这里使用自定义的 Create_Mux 函数，具体使用方法会在之后结合信号量机制加以说明。

创建信号量完成后，需创建 5 个子进程，本次实验子进程和主进程都是通过一个程序实现，区别在于主进程只含一个参数，而子进程还包含参数 ID，指明子进程序号，主函数结构参考如下

```
 int main(int argc,char *argv[])
{
    if(argc==1)
    {
        主进程
    }
    else
    {
        //子进程
        int ID=atoi(argv[1]);
        if(ID 属于生产者)
            Producer(ID)
        else

            Consumer(ID)
    }
    return 0;
}
```

所以在主进程创建子进程调用 CreateProcess 函数以传入命令行的形式创建，并且需要保存创建的子进程句柄到 Handle_process 数组内，关键代码可参考如下

```
    GetModuleFileName(NULL, CurFile, sizeof(CurFile));
    sprintf(Cmdstr, "%s %d", CurFile, ID);
    CreateProcess(NULL,Cmdstr,NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
    Handle_process[ID]=pi.hProcess;
```

各对象创建完毕后主进程进入阻塞状态，调用 WaitForMultipleObjects 等待 Handle_process 数组内句柄对应的所有子进程结束。然后关闭子进程句柄、信号量和共享内存区句柄

(2) 共享内存区的具体结构参考如下

```
struct share_memory
{
    int a[Len_buffer];
    int beg;
    int end;
};
```

共享内存区在主进程中创建，需要函数 CreateFileMapping 创建内存中临时文件映射对象，对象名称为“SHARE_MEM”，大小为自定义的共享内存区结构 share_memory，具体结构会在之后具体说明，然后通过函数 MapViewOfFile 将文件映射对象的一个视口映射到主进程，完成临时文件初始化操作

```
    HANDLE hMapping = CreateFileMapping(INVALID_HANDLE_VALUE,NULL,
    PAGE_READWRITE,0, sizeof(struct share_memory),"SHARE_MEM");
    LPVOID pData=MapViewOfFile(hMapping,FILE_MAP_ALL_ACCESS,0,0,0);
    ZeroMemory(pData, sizeof(struct share_memory));
    UnmapViewOfFile(pData);
```

共享内存区又各生产者和消费者子进程共享使用，在生产者和消费者子进程开始执行操作前，必须先通过函数 OpenFileMapping 打开之前创建的临时文件对象并获取句柄，然后通过函数 MapViewOfFIle 将共享的临时文件对象的一个视口映射到当前进程的地址空间，并定义一个相同结构体的指针指向该地址，然后就可以读取共享内存区数据了，注意最后要关闭句柄，解除映射

```
HANDLE hMapping=OpenFileMapping(FILE_MAP_ALL_ACCESS,FALSE,"SHARE_MEM");
LPVOID pFile = MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);
struct share_memory *sm=(struct share_memory*)(pFile);
执行各种操作
```

```
UnmapViewOfFile(pFile);
CloseHandle(hMapping);
```

(3) 在本实验的设计中将信号量句柄在进程内定义为全局句柄变量

```
HANDLE s_empty,s_fill,s_rw;
```

在主进程中调用自定义的 Create_Mux 函数创建信号量，在其中调用了 CreateSemaphore 函数来创建各个信号量，具体使用及各信号量命名参考如下

```
s_empty=CreateSemaphore(NULL, Len_buffer, Len_buffer, "SEM_EMPTY");
s_fill=CreateSemaphore(NULL, 0, Len_buffer, "SEM_FILL");
s_rw=CreateSemaphore(NULL, 1, 1, "SEM_RW");
```

之后在所有子进程的使用中，首先需要使用 OpenSemapHore 打开对应信号量，在本实验设计中已集成为 Open_Mux 函数，结构与 Create_Mux 相似，省略具体代码说明。

在子进程信号量使用相关内容结束后，同样需要关闭信号量句柄，已集成为 Close_Mux 函数，此处省略代码说明

(4) 对于每个生产者子进程，获取参数 ID 后，需要先在进程打开共享内存区的临时文件映射对象并将一个视口映射到当前进程地址空间，具体操作已在上文说明，最终会获得指向共享内存区结构的指针 sm，然后调用自定义函数 Open_Mux 在当前进程内获取各个信号量句柄，然后即可开始执行操作。每个生产者均需要重复 Rep_Producer 次生产操作，故需要一个循环结构，在每次循环内，生产者首先调用 Sleep 等待随机的一段时间，然后开始正式的生产执行操作，首先使用函数 WaitForSingleObject 模拟 P 操作

```
WaitForSingleObject(s_empty,INFINITE);
WaitForSingleObject(s_rw,INFINITE);
```

然后就是往缓冲区生产填充数据，这里需要通过指针 sm 进行操作，思路已在通用设计思路中提及，这里省略具体操作

然后用 ReleaseSemaphore 函数释放信号量

```
ReleaseSemaphore(s_fill,1,NULL);
ReleaseSemaphore(s_rw,1,NULL);
```

重复 Rep_Producer 次生产过程，关闭信号量句柄，解除文件映射，关闭临时文件句柄，该生产者子进程结束

(5) 对于每个消费者子进程，执行操作之前同生产者子进程.。每个消费者均需要重复 Rep_Consumer 次消费操作，故需要一个循环结构，在每次循环内，消费者首先调用 Sleep 等待随机的一段时间，然后开始正式的消费执行操作，首先使用函数 WaitForSingleObject 模拟 P 操作

```
WaitForSingleObject(s_fill,INFINITE);
WaitForSingleObject(s_rw,INFINITE);
```

取出数据的具体操作省略，理由同生产数据

然后用 ReleaseSemaphore 函数释放信号量

```
ReleaseSemaphore(s_empty,1,NULL);
ReleaseSemaphore(s_rw,1,NULL);
```

重复 Rep_Consumer 次消费过程，关闭信号量句柄，解除文件映射，关闭临时文件句柄，该消费者子进程结束

(6) 在所有子进程结束后，WaitForMultipleObject 函数结束，关闭子进程句柄、信号量和共享内存区句柄，主进程结束

编译生成可执行文件后执行结果如下所示（因文本量较大故只展示开头结尾）

```
0号进程:创建子进程，1~2号为生产者进程，3~5号为消费者进程
2号进程:生产者在0号缓冲区生产2000
现缓冲区数据为: 2000 0 0
3号进程:消费者在0号缓冲区消费2000
现缓冲区数据为: 0 0 0
1号进程:生产者在1号缓冲区生产2500
现缓冲区数据为: 0 2500 0
5号进程:消费者在1号缓冲区消费2500
现缓冲区数据为: 0 0 0
1号进程:生产者在2号缓冲区生产1000
现缓冲区数据为: 0 0 1000
2号进程:生产者在0号缓冲区生产2500
现缓冲区数据为: 2500 0 1000
4号进程:消费者在2号缓冲区消费1000
```

```
3号进程:消费者在0号缓冲区消费2500
现缓冲区数据为: 0 0 0
2号进程:生产者在1号缓冲区生产500
现缓冲区数据为: 0 500 0
1号进程:生产者在2号缓冲区生产2500
现缓冲区数据为: 0 500 2500
5号进程:消费者在1号缓冲区消费500
现缓冲区数据为: 0 0 2500
4号进程:消费者在2号缓冲区消费2500
现缓冲区数据为: 0 0 0
0号进程:子进程运行完成
```

2. 在 Linux 下实现：

(1) 主进程需要负责创建各对象，首先需要创建共享内存区，具体创建方式会在之后结合共享内存区的结构体详细说明。共享内存区创建完成后，需要创建信号量，具体创建方法会在之后结合信号量机制加以说明。

在本实验中使用一个程序运行主进程和各子进程，采用 fork 函数创建子进程，故主进程在运行到 fork 时会产生分支，以 fork 返回的 pid 号区分进程，故主函数结构参考如下，注意循环使用 fork 函数产生子进程的操作结束后必须在循环内结束进程，不然子进程也会运行 fork 函数

```
int main(int argc,char *argv[])
{
    创建共享内存区
    创建信号量
    for(int i=1;i<=Cnt_Process;i++)
    {
        Int pid=fork();
        if(pid==0)
        {
            //子进程
            if(i 对应生产者 id)
                Producer 操作
            else
                Consumer 操作
            return 0;
        }
    }
    //主进程
    主进程其余操作
    return 0;
}
```

主进程在各个对象创建完毕后需要用函数 wait 等待子进程结束，由于创建了 5 个子进程，只需要 5 次循环使用 wait(NULL)即可，最后注意解除共享内存区和信号量

(2) 共享内存区具体结构参考如下

```
struct share_memory
{
    int a[Len_buffer];
```

```
    int beg;
    int end;
};
```

共享内存区通过函数 shmget 获取，第一个参数为随意设置的一个关键值，第二个参数即为 share_memory 大小，第三个参数设置为 0666|IPC_CREAT，表示创建新内存区，所有用户均有读写权限，shmget 会返回引用标识符。创建完成后可使用函数 shmat 通过引用标识符将该内存区附加到进程内，shmat 返回值为指向实际连接到的地址的指针，然后可以通过该指针对共享内存区的对象做各种具体操作，关键代码参考如下

```
    //创建内存区
    int shmid=shmget(SHMKEY,sizeof(struct share_memory),0666|IPC_CREAT);
    //将内存区附加到进程，获取指针
    struct share_memory* sm=(struct share_memory*)shmat(shmid,0,0);
```

对于主进程，需要初始化共享内存区，对于子进程，可以生产或消费缓冲区数据，但注意在操作结束后使用 shmdt 解除进程对该共享内存区的附加。并且在主进程的最后需要使用函数 shmctl 删除内存区，参考使用如下，参数值与上文代码对应，其中 IPC_RMID 参数表示标记该内存区可删除

```
    shmdt(sm);
    shmctl(shmid,IPC_RMID,0);
```

(3) 信号量通过函数 semget 创建，第一个参数为随意设置的一个关键值，第二个参数为信号总数量，此处设置为 3，第三个参数设置为 0666|IPC_CREAT，意义同共享内存区内相同参数，其返回值为引用标识符。之后即可通过 semctl 函数对信号量做具体设置，具体参考如下

```
    int semid=semget(SEMKEY,3,0666|IPC_CREAT);
    semctl(semid,0,SETVAL,3); //empty
    semctl(semid,1,SETVAL,0); //fill
    semctl(semid,2,SETVAL,1); //rw
```

由于创建之后子进程需要对信号量做 PV 操作，需要用到 semmop 函数和 sembuf 结构，这里将 P、V 操作封装为两个函数方便调用，输入信号量集应用标识符 semid 和信号量索引 n，可对第 n 个信号量单独做 P、V 操作。

```
 void P(int semid,int n)
{
    struct sembuf temp;
    temp.sem_num=n;
    temp.sem_op=-1;
    temp.sem_flg=0;
    semop(semid,&temp,1);
}

void V(int semid,int n)
{
    struct sembuf temp;
    temp.sem_num=n;
    temp.sem_op=1;
    temp.sem_flg=0;
    semop(semid,&temp,1);
}
```

在主函数的最后需要使用函数 semctl(semid,IPC_RMID,0)删除 semid 对应的信号量

(4) 对于每个生产者进程，首先需要指向共享内存区的指针 sm，具体方法已在上文说明，然后每个生产者均需要重复 Rep_Producer 次生产操作，故需要一个循环结构，在每次循环内，生产者首先调用 Sleep 等待随机的一段时间，然后开始正式的生产执行操作，首先进行 P 操作

```
P(semid,0); //empty
P(semid,2); //rw
```

然后就是往缓冲区生产填充数据，这里需要通过指针 sm 进行操作，思路已在通用设计思路中提及，这里省略具体操作

然后进行 V 操作

```
V(semid,1); //fill
V(semid,2); //rw
```

重复 Rep_Producer 次生产过程，解除共享内存区附加，该生产者子进程结束。

(5) 对于每个消费者进程，首先需要指向共享内存区的指针，具体方法已在上文说明，然后每个消费者均需要重复 Rep_Consumer 次消费操作，故需要一个循环结构，在每次循环内，消费者首先调用 Sleep 等待随机的

一段时间，然后开始正式的消费执行操作，首先进行 P 操作

```
P(semid,1); //fill
P(semid,2); //rw
```

取出数据的具体操作省略，理由同生产数据

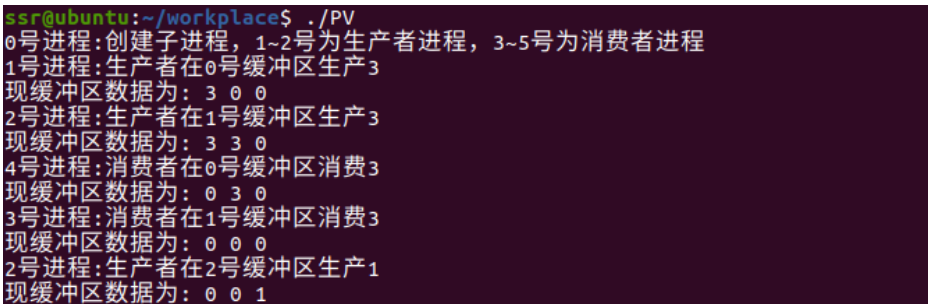
然后进行 V 操作

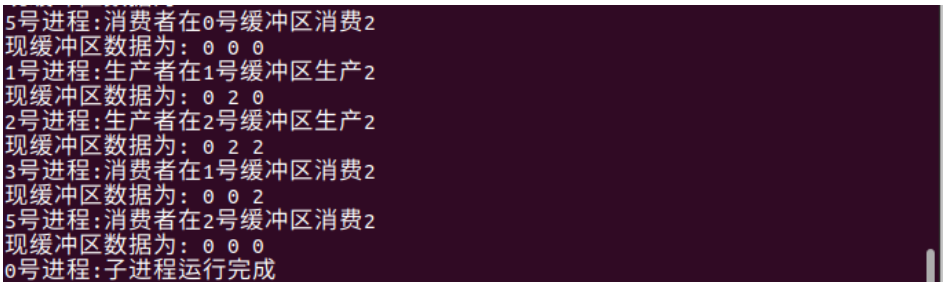
```
V(semid,0); //empty
V(semid,2); //rw
```

重复 Rep_Consumer 次消费过程，解除共享内存区附加，该消费者子进程结束。

(6) 调用 5 次 wait(NULL)等待 5 个子进程结束，然后用上文说明的方法删除共享内存区和信号量集。

编译生成可执行文件后执行结果如下所示（因文本量较大故只展示开头结尾）





五、 实验收获与体会

这次实验首先复习了如何使用信号量模拟生产者消费者的同步过程，PV 操作的顺序必须注意，否则可能造成死锁。但大部分时间还是用在了学习 api 调用上，不得不说 windows 的 api 实在麻烦，不仅需要考虑的参数多，收尾操作也繁琐许多，相较之下 linux 简单多了。