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	1120191600	姓名	张驰

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

- 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。
- 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
- 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

说明：

- 分别在 Windows 和 Linux 平台上做。
- 显示每次添加或读取数据的时间及缓冲区的映像。
- 生产者和消费者用进程模拟。

三、实验环境及配置方法

Windows10

VMware workstation pro

Ubuntu18

四、实验方法

操作系统课程设计实验报告

1.对生产者和消费者执行过程的描述：

设置三个公共信号量，三个信号量为 full、empty、mutex。

设置一个公共缓冲区，缓冲区的组成由 char 类型的二维数组（可存放多个字符串），数组的头指针 head，数组的尾指针 tail。

生产者生产字符串的过程可以描述为：

```
P(empty);        //申请空闲缓冲区资源
P(mutex);          //申请对缓冲区修改的权限
strcpy(str_buffer[tail] , str);    //在空缓冲区填入字符串
tail = (tail + 1) % buffer_size    //尾部指针递增
V(full);         //填充了一个缓冲区，释放填充缓冲区信号量
V(mutex);          //释放修改权限
```

消费者消费字符串的过程可以描述为：

```
P(full);         //申请已填充缓冲区资源
P(mutex);          //申请对缓冲区修改的权限
str = str_buffer[head]     //从已填充缓冲区读取字符串
str_buffer[head][0] = '\0';          //清空缓冲区内的字符串
head = (head + 1) % buffer_size    //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(mutex);          //释放修改权限
```

之后主进程回收共享内存区和信号量。

2. 实验方法具体描述：

主进程负责创建共享内存区和信号量。创建信号量和共享内存区后，创建五个子进程，将保存创建的子进程句柄放置在进程句柄的数组中。

其中有 2 个生产者进程，3 个消费者进程，根据子进程的 ID 可以判断其为生产者进程还是消费者进程。生产者进程调用生产者函数，根据随机数从 6 个字符串中随机生成一个字符串，休眠随机的时间，之后生产者执行 PV 操作并将生成的字符串，将相应的字符串放置在共享内存区中。消费者进程调用消费者函数，随即休眠一定时间，休眠后去共享内存区的字符串，并将共享内存区相应位置置为空串。

主进程结束之后关闭共享内存区和信号量相应的句柄。

五、实验步骤：（代码实现）

1. windows 系统下的实验步骤：

(1)宏定义和共享缓冲区定义，创建信号量和进程的句柄数组：

```
#define Producer_num 2
#define Producer_repeat 12
// 两个生产者 重复六次
#define Consumer_num 3
#define Consumer_repeat 8
//三个消费者 重复八次
#define Process_num 5
//共五个进程
#define Buffer_size 6
//大小为6 的缓冲区
struct share_memory
{
    char str[Buffer_size][15];
    int head;
    int tail;
};
HANDLE s_empty, s_full, s_mutex;//信号量句柄
HANDLE Handle_process[Process_num + 5];//进程的句柄数组
char pro_str[6][15] = { "Apple","Banana","Circle","Dog","Enter","Fruit
" };
```

(2)创建共享缓存区函数：

使用 CreateFileMapping()函数来创建共享缓存区的映射，返回相应的句柄。在代码的注释中包含对函数 CreateFileMapping 中参数的描述。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 MakeSharedFile()
{
    HANDLE hMapping = CreateFileMapping(
        INVALID_HANDLE_VALUE, //创建与物理文件无关的内存映射
        NULL, //使用默认的安全设置
        PAGE_READWRITE, //保护设置
        0,
        sizeof(struct share_memory), //高低位文件大小
        "SHARE_MEM" //共享内存名称
    );
    //将文件映射对象的一个视口映射到主进程，完成共享缓存区初始化
    LPVOID pData = MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS, 0, 0,
0);
    //将指定的内存块清零
    ZeroMemory(pData, sizeof(struct share_memory));
    //停止映射
    UnmapViewOfFile(pData);
```

```
    return(hMapping);
}
```

(3)创建信号量句柄的三个函数，创建、打开、关闭：

使用 CreateSemaphore 函数，中间两个参数分别是初始值和最大值。

其中，s_empty 表示的是信号量 empty，s_full 表示的是信号量 full，s_mutex 表示的是信号量

```
//下面是关于信号量句柄的三个函数—创建、打开、关闭;
void Create_sig()
{
    s_empty = CreateSemaphore(NULL, Buffer_size, Buffer_size, "SEM_EMPTY");
    s_full = CreateSemaphore(NULL, 0, Buffer_size, "SEM_FULL");
    s_mutex = CreateSemaphore(NULL, 1, 1, "SEM_MUTEX");
}

void Open_sig()
{
    s_empty = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SEM_EMPTY");
    s_full = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SEM_FULL");
    s_mutex = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "SEM_MUTEX");
}

void Close_sig()
{
    CloseHandle(s_empty);
    CloseHandle(s_full);
    CloseHandle(s_mutex);
}
```

(4)创建子进程的函数

仍旧使用 CreateProcess()函数创建子进程，将创建后的子进程句柄放置在子进程句柄的数组中。同时，其中的 Cmdstr[]和 CurFile[]

```
//创建子进程的函数
void New_SubProcess(int ID)
{
    STARTUPINFO si;
    memset(&si, 0, sizeof(si));
    si.cb = sizeof(si);
    PROCESS_INFORMATION pi;

    char Cmdstr[105];
    char CurFile[105];
```

```
    //构造一个 Cmdstr 来模拟创建进程的操作
    GetModuleFileName(NULL, CurFile, sizeof(CurFile));
    sprintf(Cmdstr, "%s %d", CurFile, ID);

    //创建进程并存储进程的句柄
    CreateProcess(NULL, Cmdstr, NULL, NULL, FALSE, 0, NULL, NULL, &si,
 &pi);
    Handle_process[ID] = pi.hProcess;

    return;
}
```

(5) 生产者生产字符串的函数

共享内存区由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，这非常重要。

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，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_sig()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 Producer_reaper 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()函数等待随机的一段时间，然后开始正式的生产执行操作，使用函数 WaitForSingleObject 模拟 P 操作，使用 ReleaseSemaphore 函数释放信号量模拟 V 操作。

```
void Producer(int ID)
{
    //打开映射
    HANDLE hMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, "SHA
RE_MEM");
    LPVOID pFile = MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS, 0, 0,
0);
    struct share_memory* sm = (struct share_memory*)(pFile);
    Open_sig();
```

```
    for (int i = 0; i < Producer_repeat; i++)
    {
        srand((unsigned int)time(NULL));
        int temp = (rand() % 3 + 1) * 500;
        Sleep(temp);

        WaitForSingleObject(s_empty, INFINITE);//P(s_empty)
        WaitForSingleObject(s_mutex, INFINITE);//P(s_mutex)

        //随机从六个可能的字符串，放入缓冲区中
        srand((unsigned int)time(NULL));
        temp = rand() % 6;
        printf("%d 号进程:生产者在%d 号缓冲区生产字符
串: %s\n", ID, sm->tail, pro_str[temp]);
        strcpy(sm->str[sm->tail], pro_str[temp]);

        sm->tail = (sm->tail + 1) % Buffer_size;
        print(sm);

        ReleaseSemaphore(s_full, 1, NULL);//V(s_full)
        ReleaseSemaphore(s_mutex, 1, NULL);//V(s_mutex)
    }
    Close_sig();
    //停止映射 关闭句柄
    UnmapViewOfFile(pFile);
    CloseHandle(hMapping);
}
```

(6)消费者消费字符串的函数：

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 Consumer_repeat 次消费操作。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，使用函数 WaitForSingleObject 模拟 P 操作，使用 ReleaseSemaphore 函数释放信号量模拟 V 操作。

```
void Consumer(int ID)
{
    //建立映射
    HANDLE hMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, "SHA
RE_MEM");
    LPVOID pFile = MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS, 0, 0,
```

```
0);
    struct share_memory* sm = (struct share_memory*)(pFile);
    Open_sig();
    for (int i = 0; i < Consumer_repeat; i++)
    {
        srand((unsigned int)time(NULL) + ID);
        int temp = (rand() % 3 + 1) * 1000;
        Sleep(temp);

        WaitForSingleObject(s_full, INFINITE);//P(s_full)
        WaitForSingleObject(s_mutex, INFINITE);//P(s_mutex)

        printf("%d 号进程:消费者在%d 号缓冲区消费字符
串: %s\n", ID, sm->head, sm->str[sm->head]);
        strcpy(sm->str[sm->head], "\0");
        sm->head = (sm->head + 1) % Buffer_size;
        print(sm);

        ReleaseSemaphore(s_empty, 1, NULL);//V(s_empty)
        ReleaseSemaphore(s_mutex, 1, NULL);//V(s_mutex)
    }
    Close_sig();
    //停止映射 关闭句柄
    UnmapViewOfFile(pFile);
    CloseHandle(hMapping);
}
```

(7)主函数:

主函数要在创建进程之前，创建共享缓存区，将其临时文件映射到句柄之中，创建信号量调用 Create_sig 函数，将创建的信号量保存在 s_empty,s_full,s_mutex 句柄之中，之后根据创建子进程的 ID 来识别子进程是生产者还是消费者进程。

```
int main(int argc, char* argv[])
{
    if (argc == 1)
    {
        //在父进程中创建映射，创建信号量对应的句柄
        HANDLE hMapping = MakeSharedFile();
        Create_sig();
        printf("我们设定 1 号 2 号进程为生产者进程，3 号 4 号 5 号为消费者进程
\n");
        //创建 5 个子进程，分别对应两个生产者进程，三个消费者进程;
        for (int i = 1; i <= Process_num; i++)
        {
```

```
            New_SubProcess(i);
        }
        //父进程等待多个子进程结束
        WaitForMultipleObjects(Process_num, Handle_process + 1, TRUE,
INFINITE);

        for (int i = 1; i <= Process_num; i++)
        {
            CloseHandle(Handle_process[i]);
        }
        Close_sig();
        printf("父进程结束\n");
        CloseHandle(hMapping);
    }
    else
    {
        int ID = atoi(argv[1]);
        if (ID <= Producer_num)
        {
            Producer(ID);
        }
        else
        {
            Consumer(ID);
        }
    }
    return 0;
}
```

2.Linux 系统下的实验步骤：

Linux 系统上整体思路大体一致，下面主要描述 Linux 系统中与 Windows 系统中不同的调用方式。

(1)主函数：

使用 semid 和 shmid 来进行信号量和共享缓存区初始化：

```
int semid = Create_sig();//信号量初始化
int shmid = Create_Share_Memory();// 共享缓冲区初始化
```

使用 fork()函数来创建子进程，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；

使用 wait()来等待子进程，在等待子进程结束后要使用

```
semctl(semid, IPC_RMID, 0);
```

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

解除信号量和内存共享区。

根据 pid 标志，对子进程进行区分，调用 Producer()和 Consumer()函数：

```
//创建 5 个子进程，分别对应两个生产者进程，三个消费者进程;
for (int i = 1; i <= Process_num; i++)
{
 int pid = fork();
 if (pid == 0)
 {
  //子进程
  if (i <= Producer_num)
  {
   Producer(i, shmid, semid);
  }
  else
  {
   Consumer(i, shmid, semid);
  }
  return 0;
 }
}
```

(2)创建共享缓存区：

共享内存区通过函数 shmget 获取，第一个参数为设置的一个关键值，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具体操作。

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**。

在主进程的最后需要使用函数 **shmctl 删除内存区。**

```
int Create_Share_Memory()
{
 //创建一个共享内存对象
 int shmid = shmget(
  SHMKEY,//标识符关键字
  sizeof(struct share_memory),//共享存储段字节数
  0666 | IPC_CREAT//读写权限
 );
 struct share_memory* sm = (struct share_memory*)shmat(shmid, 0, 0);
```

```
 //引用标识符将内存区附加到进程中
 sm->head = sm->tail = 0;
 for (int i = 0; i < Buffer_size; i++)
 {
  sm->str[i][0] = '\0';
 }
 return shmid;
}
```

(2)信号量的相关操作：

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。之后可通过 semctl 函数对信号量做具体设置。

PV 操作需要用的 semop 函数以及，semop 函数可以对信号量进行操作，sembuf 结构如以下代码中的注释所示：

```
int Create_sig()
{
 //创建信号量集对象
 int semid = semget(
  SEMKEY,//关键值
  3,//信号量的数目
  0666 | IPC_CREAT// 0666 表示用户和同组用户有读写执行权限，其他用户没有任何访问权限
 );
 //信号量集标识符  下标    初始值
 semctl(semid, 0, SETVAL, 6); //empty
 semctl(semid, 1, SETVAL, 0); //full
 semctl(semid, 2, SETVAL, 1); //mutex
 return semid;
}

void P(int semid, int n)
{
 struct sembuf temp;
 //制定要对其进行的操作
 temp.sem_num = n;//信号量编号
 temp.sem_op = -1;// -1 -- P  || +1 -- V
 temp.sem_flg = 0;
 //执行 temp 对应的 sem 操作
 semop(semid, &temp, 1);
}
```

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

(3)生产者进程和消费者进程的相关改变：

生产者进程的 PV 相关操作改为：

```
//P 操作
  P(semid, 0);//empty
  P(semid, 2);//mutex
//V 操作
  V(semid, 1);//full
  V(semid, 2);//mutex
```

消费者进程的 PV 相关操作改为：

```
//P 操作
  P(semid, 1);//full
  P(semid, 2);//mutex
//V 操作
  V(semid, 0);//empty
  V(semid, 2);//mutex
```

注意生产者和消费者都要通过 shmdt(sm)解除链接。

六、实验结果和分析

1. 在 Windows 系统中的运行结果：

调用 PV.exe,可以看到执行结果如下所示，由于输出量较大，只展示部分输出结果：

```
我们设定1号 2号进程为生产者进程，3号 4号 5号为消费者进程
1号进程:生产者在0号缓冲区生产字符串：Banana

----------展示目前的缓冲区数据---------
Banana 空串 空串 空串 空串 空串
------------展示目前数据完毕------------

2号进程:生产者在1号缓冲区生产字符串：Banana

----------展示目前的缓冲区数据---------
Banana Banana 空串 空串 空串 空串
------------展示目前数据完毕------------

2号进程:生产者在2号缓冲区生产字符串：Enter

----------展示目前的缓冲区数据---------
Banana Banana Enter 空串 空串 空串
------------展示目前数据完毕------------

1号进程:生产者在3号缓冲区生产字符串：Enter

----------展示目前的缓冲区数据---------
Banana Banana Enter Enter 空串 空串
------------展示目前数据完毕------------

3号进程:消费者在0号缓冲区消费字符串：Banana

----------展示目前的缓冲区数据---------
空串 Banana Enter Enter 空串 空串
------------展示目前数据完毕------------
```

```
3号进程:消费者在2号缓冲区消费字符串：Apple

----------展示目前的缓冲区数据---------
空串 空串 空串 Apple Dog Dog
------------展示目前数据完毕------------

5号进程:消费者在3号缓冲区消费字符串：Apple

----------展示目前的缓冲区数据---------
空串 空串 空串 空串 Dog Dog
------------展示目前数据完毕------------

4号进程:消费者在4号缓冲区消费字符串：Dog

----------展示目前的缓冲区数据---------
空串 空串 空串 空串 空串 Dog
------------展示目前数据完毕------------

4号进程:消费者在5号缓冲区消费字符串：Dog

----------展示目前的缓冲区数据---------
空串 空串 空串 空串 空串 空串
------------展示目前数据完毕------------

父进程结束
```

2. 在 Linux 系统中的运行结果：

同样由于文本量过大，只展示部分输出内容。

```
zc1120191600@ubuntu:~$ cd code
zc1120191600@ubuntu:~/code$ ls
a.out  mytime  mytime.c  program  program2.c  programm  PV  PV.c
zc1120191600@ubuntu:~/code$ ./PV
我们设定1号 2号进程为生产者进程，3号 4号 5号为消费者进程
这里是主进程
2号进程:生产者在0号缓冲区生产字符串：Dog

----------展示目前的缓冲区数据---------
Dog 空串 空串 空串 空串 空串
------------展示目前数据完毕------------

4号进程:消费者在0号缓冲区消费字符串：Dog

----------展示目前的缓冲区数据---------
空串 空串 空串 空串 空串 空串
------------展示目前数据完毕------------

1号进程:生产者在1号缓冲区生产字符串：Dog

----------展示目前的缓冲区数据---------
空串 Dog 空串 空串 空串 空串
------------展示目前数据完毕------------

5号进程:消费者在1号缓冲区消费字符串：Dog

----------展示目前的缓冲区数据---------
空串 空串 空串 空串 空串 空串
------------展示目前数据完毕------------
```

```
------------展示目前数据完毕------------

2号进程:生产者在4号缓冲区生产字符串：Banana

----------展示目前的缓冲区数据---------
空串 空串 空串 空串 Banana 空串
------------展示目前数据完毕------------

1号进程:生产者在5号缓冲区生产字符串：Banana

----------展示目前的缓冲区数据---------
空串 空串 空串 空串 Banana Banana
------------展示目前数据完毕------------

3号进程:消费者在4号缓冲区消费字符串：Banana

----------展示目前的缓冲区数据---------
空串 空串 空串 空串 空串 Banana
------------展示目前数据完毕------------

3号进程:消费者在5号缓冲区消费字符串：Banana

----------展示目前的缓冲区数据---------
空串 空串 空串 空串 空串 空串
------------展示目前数据完毕------------

End
zc1120191600@ubuntu:~/code$
```

操作系统课程设计实验报告

七、讨论、心得

本次实验过程较为复杂，在这次实验中，我运用了操作系统课程中进程控制中的相关内容，使用信号量的方式对生产者和消费者的同步过程进行模拟，在 PV 过程中设置了 full,empty,mutex 三个信号量，PV 操作的顺序必须要注意，否则可能会产生死锁，导致子进程相互等待无法终止。

在本次学习的过程中，我进一步了解了 windows 系统和 linux 系统中相关的 API 操作，了解相关的函数，对双系统上的进程控制有了进一步的了解。