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	1120180207	姓名	唐小娟

一、 实验目的

1. 掌握进程间通信的方法
2. 熟悉生产者消费者该经典 IPC 问题
3. 了解 Linux 和 Windows 下信号量和互斥体的使用
4. 了解 Linux 和 Windows 共享内存机制的相关操作

二、 实验内容

1. Windows 下的生产者消费者问题

(1). 创建进程、信号量和互斥体

(2). 实现共享内存

(3). PV 操作

2. Linux 下的生产者消费者问题

(1). 创建进程、信号量和互斥体

(2). 实现共享内存

(3). PV 操作

三、实验环境及配置方法

操作系统：Windows 10，Ubuntu 20.04，Linux 5.4.0-42

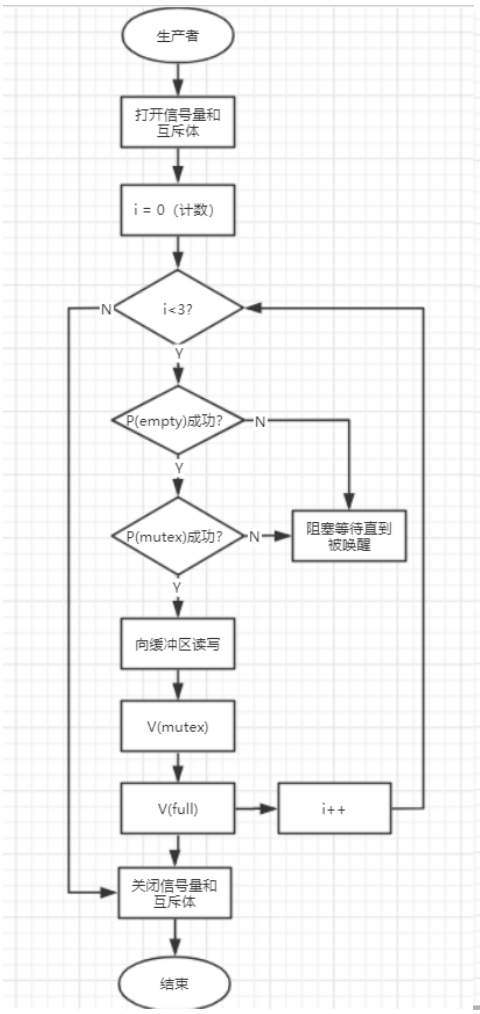
集成开发环境：Microsoft VS Code

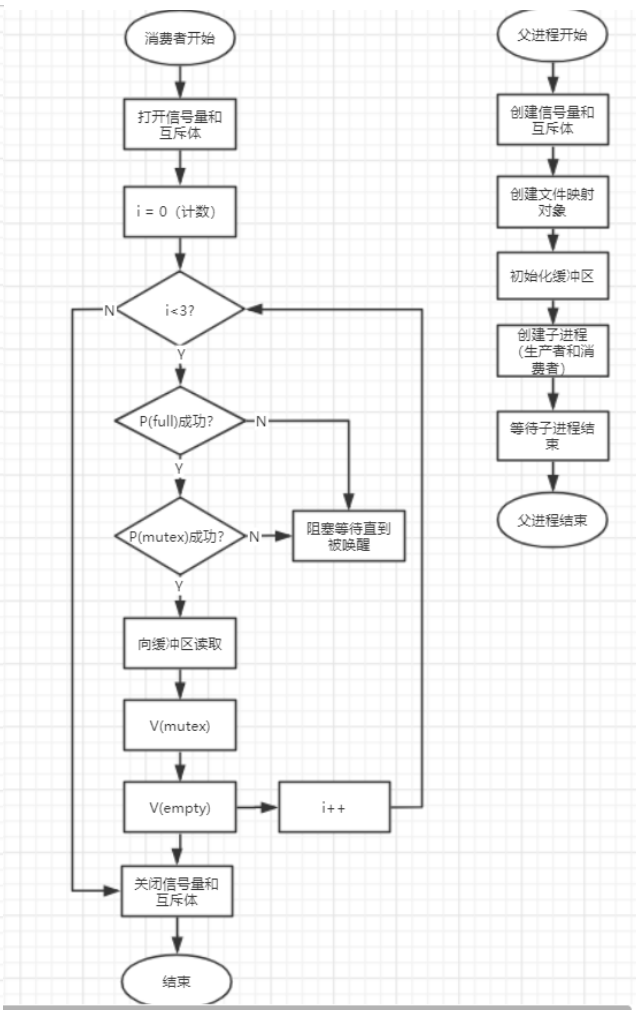
编译器：gcc 9.3.0

四、 实验方法和实验步骤（程序设计与实现）

1. 流程图

根据课本上的经典 IPC 问题，可以得到如下控制流程图：





2. Windows 下的生产者消费者问题

代码整体思路：由父进程创建具有相同可执行文件路径的子进程（7 个，分别是 3 个生产者，4 个消费者），进程运行时根据传入的参数 argv[1]，得知该进程是父进程、生产者还是消费者：

父进程：执行 Parent()函数，创建互斥体、信号量和文件映射对象，并且对他们相应的初始化；

生产者：实行 Producer()函数，打开相应的信号量和互斥体，申请互斥体和信号量，若申请成功则将内容写入缓冲区，否则阻塞等待直到被唤醒，写完相应内容后，释放互斥体和信号量，该过程重复 4 次；

消费者：执行 Consumer()函数，同生产者步骤一致，只是申请和释放的信号量和生产者有所不同。

(1). 父进程

1). 创建互斥体

调用 CreateMutex()创建互斥体，命名为 mutex。初始时，由于 bInitialOwner = true，互斥体被父进程所持有，所以要进行释放 mutex，生产者和消费者才能获得该资源。

```
HANDLE CreateMutex(
LPSECURITY_ATTRIBUTES lpMutexAttributes, // 指向安全属性的指针(NULL)
BOOL bInitialOwner,                      // 初始化互斥对象的所有者
LPCTSTR lpName                           // 指向互斥对象名的指针
);
```

2). 创建信号量

调用 CreateSemaphore()创建两个信号量 empty 和 full，初始值分别为 4 和 0。

```
HANDLE CreateSemaphore(
LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // 指向安全属性的指针
LONG lInitialCount,                          // 初始信号量大小
LONG lMaximumCount,                          // 信号量的最大值
LPCTSTR lpName                               // 信号量的名字
);
```

3). 创建文件映射对象

Windows 中共享内存需要调用 CreateFileMapping()创建文件映射对象，命名为 buffer，大小为结构体 buf 的大小。结构体如下：

```
struct buf
{
    char text[4];
    int read_op;
    int write_op;
};
```

上述结构是一个长度为 4 的环形队列，写指针和读指针均初始化为 0。

```
HANDLE CreateFileMapping(
HANDLE hFile,                       //物理文件句柄
LPSECURITY_ATTRIBUTES lpAttributes, //安全设置(NULL)
DWORD flProtect,                    //保护设置
DWORD dwMaximumSizeHigh,            //高位文件大小
DWORD dwMaximumSizeLow,             //低位文件大小
LPCTSTR lpName                      //共享内存名称
);
```

4). 初始化文件映射对象

创建文件映射对象之后，调用 MapViewOfFile()将该对象映射到自己进程的地址空间，得到相应地址后，进行初始化。最后，调用 UnmapViewOfFile()取消映射。

```
HANDLE CreateFileMapping(
HANDLE hFile,                       //物理文件句柄
LPSECURITY_ATTRIBUTES lpAttributes, //安全设置(NULL)
DWORD flProtect,                    //保护设置
DWORD dwMaximumSizeHigh,            //高位文件大小
DWORD dwMaximumSizeLow,             //低位文件大小
LPCTSTR lpName                      //共享内存名称
```

```
);
```

5). 创建消费者和生产者进程

单独使用函数接口 createChildProcess()创建子进程，传入相应 Producer、Consumer 字符串参数来表示该进程属于消费者还是生产者。

6). 调用 WaitForMultipleObjects

父进程等待生产者和消费者完成后，关闭互斥体、信号量、文件映射对象的句柄。

(2). 生产者

1). 打开信号量和互斥体

调用 OpenMutex()和 OpenSemaphor()打开互斥体和信号量。

```
HANDLE OpenMutex(
DWORD dwDesiredAccess, // 访问权限
BOOL bInheritHandle,   // 子进程是否继承句柄
LPCTSTR lpName         // 互斥体的名字
);

HANDLE OpenSemaphore(
DWORD dwDesiredAccess, // 访问权限
BOOL bInheritHandle,   // 子进程是否继承句柄
LPCTSTR lpName         // 信号量名字
);
```

2). 循环写入缓冲区

调用 Sleep()等待 3s 后，通过 WaitForSingleObject()先等待信号量 empty，之后等待互斥体 mutex，若失败则一直阻塞等待，若成功则调用 OpenFileMapping()打开文件映射对象，并且通过 MapViewOfFile()映射到该进程的地址空间，获取地址指针写入数据。写入完成后，释放 mutex 互斥体和 full 信号量。

```
HANDLE WINAPI OpenFileMapping(
DWORD dwDesiredAccess, //访问权限
BOOL bInheritHandle,   //子进程是否继承句柄
LPCTSTR lpName         //文件映射对象的名字
);
```

3). 关闭信号量和互斥体

调用 CloseHandle()关闭相应句柄对象。

(3). 消费者

1). 打开信号量和互斥体

调用 OpenMutex()和 OpenSemaphor()打开互斥体和信号量。

2). 循环从缓冲区读出

调用 Sleep()等待 3s 后，通过 WaitForSingleObject()先等待信号量 full，之后等待互斥体 mutex，若失败则一直阻塞等待，若成功则调用 OpenFileMapping()打开文件映射对象，并且通过 MapViewOfFile()映射到该进程的地址空间，获取地址指针读取数据。读取完成后，释放 mutex 互斥体和 empty 信号量。

3). 关闭信号量和互斥体

调用 CloseHandle()关闭相应句柄对象。

3. Linux 下的生产者和消费者问题

(1). 父进程

1). 创建信号量集合

调用 semget()创建信号量集合，包括 mutex、full、empty，成功调用后返回信号量集合的标识码 sem_id。

```
int semget(
  key_t  _key,       //键值
  int  _nsems,       //信号量的个数
  int _semflg        //访问权限
  );
```

2). 初始化信号量集合

调用 semctl(),利用结构体 semun 进行初始化信号量集合。

```
int semctl(
  int _semid,       //信号量的标识符，也就是 semget()的返回值
  int _semnum,      //操作信号在信号集中的开始编号
  int _cmd,         //要进行的操作
  union semun arg      //可选
  );
```

3). 创建共享内存区

调用 shmget()创建共享内存区，大小为结构体 buf 的大小，返回值为内存区的标识符，用 shm_id 表示。

```
int shmget(
  key_t key,       //共享内存区的键值
  size_t size,     //共享内存区的大小
  int shmflg       //标志(IPC_CREATED)
  );
```

4). 初始化共享内存区

利用 shmat()映射内存区到进程的地址空间，返回为地址后，进行初始化赋予初值，完成后调用 shmdt()解除映射关系。

```
void *shmat(
  int shmid,            //共享内存标识符
  const void *shmaddr,  //指定共享内存在进程内存地址的位置(NULL)
  int shmflg            //权限模式
  );
```

5). 创建子进程

创建 3 个生产者子进程，4 个消费者子进程，分别执行 Producer()和 Consumer()的函数。

6). 等待子进程结束

调用 wait()等待子进程结束后，删除信号量集合和共享内存区。

(2). 生产者

1). 打开信号量集合

调用 semget()打开具有相同键值的信号量集合，得到信号量集合的标识。

2). 循环向缓冲区写入

在等待一段时间后，申请相应的信号量。首先申请 empty 信号量，即 P(sem_id,2)，再申请 mutex 信号量，即 P(sem_id,0),之后打开共享内存区，并且映射到本进程的地址空间写入数据，写入完成后，释放 mutex 信号量和 full 信号量，即 V(sem_id,0),V(sem_id,1)。

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

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

semop()功能是对信号量集合进行操作,定义如下:

```
int semop（
  int semid,            //信号量集合的标识符
  struct sembuf *sops,  //指向存储信号操作结构
  size_t nsops          //操作信号的数量
  );
```

(3). 消费者

1). 打开信号量集合

调用 semget()打开具有相同键值的信号量集合，得到信号量集合的标识。

2). 循环从缓冲区读取

在等待一段时间后，申请相应的信号量。首先申请 full 信号量，即 P(sem_id,1)，再申请 mutex 信号量，即 P(sem_id,0),之后打开共享内存区，并且映射到本进程的地

址空间写入数据，写入完成后，释放 mutex 信号量和 empty 信号量，即 V(sem_id,0),V(sem_id,2)。

五、实验结果和分析

1. Windows 下的生产者和消费者问题

运行 PV 后，我们可以看到生产者和消费者交替的写数据读数据，并且不会存在有消费者读取到初始字符-，生产者和消费者一共分别进行了 12 次读写操作。

```
the producer: push T
the buffer-text:T---
the producer: push T
the buffer-text:TT--
the consumer: pop T
the buffer-text:-T--
the consumer: pop T
the buffer-text:----
the producer: push T
the buffer-text:--T-
the consumer: pop T
the buffer-text:----
the producer: push X
the buffer-text:---X
the producer: push X
the buffer-text:X--X
the consumer: pop X
the buffer-text:X---
the producer: push X
the buffer-text:XX--
the consumer: pop X
the buffer-text:-X--
the consumer: pop X
the buffer-text:----
the producer: push X
the buffer-text:--X-
the producer: push X
the buffer-text:--XX
the consumer: pop X
the buffer-text:---X
the consumer: pop X
the buffer-text:----
the producer: push X
the buffer-text:X---
the consumer: pop X
the buffer-text:----
the producer: push J
the buffer-text:-J--
the consumer: pop J
the buffer-text:----
the producer: push J
the buffer-text:--J-
the producer: push J
the buffer-text:--JJ
the consumer: pop J
the buffer-text:---J
the consumer: pop J
the buffer-text:----
```

2. Linux 下的生产者和消费者问题

运行./PV 后，生产者和消费者交替从缓冲区读数据和写数据，分别进行了 12 次。

```
ostxj@ostxj-virtual-machine:~/studycodes/vscode$ ./PV
the producer: push J
the buffer-text:J---
the producer: push J
the buffer-text:JJ--
the producer: push J
the buffer-text:JJJ-
the consumer: pop J
the buffer-text:-JJ-
the consumer: pop J
the buffer-text:--J-
the consumer: pop J
the buffer-text:----
the producer: push T
the buffer-text:---T
the producer: push T
the buffer-text:T--T
the producer: push T
the buffer-text:TT-T
the consumer: pop T
the buffer-text:TT--
the consumer: pop T
the buffer-text:-T--
the producer: push J
the buffer-text:-TJ-
the producer: push J
the buffer-text:-TJJ
the consumer: pop T
the buffer-text:--JJ
the producer: push J
the buffer-text:J-JJ
the consumer: pop J
the buffer-text:J--J
the consumer: pop J
the buffer-text:J---
the consumer: pop J
the buffer-text:----
the producer: push X
the buffer-text:-X--
the producer: push X
the buffer-text:-XX-
the producer: push X
the buffer-text:-XXX
the consumer: pop X
the buffer-text:--XX
the consumer: pop X
the buffer-text:---X
the consumer: pop X
the buffer-text:----
```

六、讨论、心得

在这个实验过程中，首先就要明白需要运用哪些原理，是信号量？是消息缓冲？还是共享内存区。其二 linux 上的代码实现容易，但是 windows 上的函数说明太过复杂，尤其面对极为陌生的数据结构，这时候需要细心的翻看微软的文档。其三，我在实现 windows 的生产者和消费者问题时，由于之前翻阅书知道，创建进程后，如果不对进程内核操作，要马上关闭进程和线程句柄，所以我便想当然的，当我创建信号量

和互斥体等对象时，也马上关闭了该句柄，导致程序代码运行出错。后来查阅资料得知，每打开一次句柄，句柄计数加 1，关闭句柄，则计数减 1，如果句柄计数为 0，那么该对象会被删除。那为什么线程句柄关闭，线程不会被删除呢？这是由于如下：

==========

线程作为一种资源创建后不只被创建线程引用，我想系统自身为了管理线程也会有一个引用，所以用户线程释放线程句柄后，引用计数也不会是零。引用计数是资源自我管理的一种机制，资源本身以引用计数为零来得知别人不再需要自己，从而把自己kill掉。

==========

CreateThread后那个线程的引用计数不是1，调用CloseHandle只是说自己对这个线程没有兴趣了，线程还是正常运行的

==========

CreateThread后那个线程的引用计数不是1，而是2。

虽然调试代码花了我很长的时间，也不明白为什么自己会出现那么多 bug，但是既然自己完成一件事比别人花费的时间和踩得坑都多，那么在实验的理解程度上也会更加深刻。