

课程编号：A0802040060

《操作系统原理及安全》作业报告



姓	名
班	级
实	验 名 称
开	设 学 期
开	设 时 间
报	告 日 期
评	定 成 绩

东北大学软件学院

同步机制(1)

一、实验目的

探索、理解并掌握操作系统同步机制的设计和实现机理，针对所谓的银行账户转账同步问题，构建基于 Peterson 算法的同步解决方案以及基于 Windows(或 Linux)操作系统同步机制(主要是互斥机制)的解决方案，并进行分析和比较。

二、实验环境

1.实验类型：设计型。

2.实验环境：

1. 操作系统：

windows10

2. 调试软件名称及版本号

Visual Studio Code February 2022 (version 1.65)

3. 编程语言及版本号

C++/C

4. 上机地点

班级未集中进行实验课

5. 机器台号

华硕飞行堡垒 8

三、实验内容

针对所谓的银行账户转账同步问题，分析、设计和利用 C 语言编程实现基于 Peterson 算法的同步解决方案，以及基于 Windows(或 Linux)操作系统同步机制的相应解决方案，并就自编同步机制与操作系统自身同步机制的效率进行比较和分析。

四、实验流程

本部分实验包含以下几个部分：

(1) 银行账户转账同步问题的抽象及未采取同步控制情况下的编程实现，记作**实验 1.1**；

(2) 基于 Peterson 算法的银行账户转账同步问题解决方案，记作**实验 1.2**；

(3) 基于 Windows(或 Linux) 操作系统同步机制的银行账户转账同步问题解决方案，记作**实验 1.3**；

(4) Peterson 算法同步机制和 Windows (或 Linux) 操作系统同步机制的效率比较，记作**实验 1.4**。

1. 实验思路:

本实验的重点是针对所谓的银行账户转账同步问题。上述问题的大体思想，就是创建两个子线程，每个子线程分别对银行的账户(账户 1 和账户 2)进行资金的交互。对银行的账户 1 和账户 2 进行初始化这里为了更加直观的展现出结果，我们规定账户 1 只接受资金转入操作，账户 2 只进行资金转出操作。在账户资金发生变动时，采用一个临时变量来模拟对应账户的寄存器。两个子线程并发执行，设置线程循环的终止条件为账户 1 和账户 2 的资金总和不为 0。若循环终止，则说明两个子线程发生了数据的干扰或者冲突，由于资金转入和资金转出操作都不属于原子操作。会有一定的概率发生不满足循环条件的情况。

对于**实验 1.1**的部分，正常创建并运行两个子线程，统计数据即可。

对于**实验 1.2**，需要使用 Peterson 算法。Peterson 算法的核心思想是：双标志后检查法中，两个进程都争着想进入临界区，但是谁也不让谁，最后谁都无法进入临界区。Gary L. Peterson 想到了一种方法，如果双方都争着想进入临界区，那可以让进程尝试“孔融让梨”，主动让对方先使用临界区。Peterson 算法用软件方法解决了进程互斥问题，遵循了空闲让进、忙则等待、有限等待三个原则，但是依然未遵循让权等待的原则。但是总的来说，是实现进程互斥访问临界区的一种方法，避免了单标志法必须交替访问的限制，以及双标志法后检验的“饥饿”问题。因此在对应子线程运行前、子线程每次循环完、子线程运行结束，需要初始化对应子线程的数据。下面是 Peterson 算法的基本结构：

Pi 进程:

```
flag[i] = True, turn=j;//进入区
while(flag[j]&&turn==j);//进入区
critical section;//临界区
flag[i] = False;//退出区
remainder section;//剩余区
```

Pj 进程:

```
flag[j] = True, turn=i;//进入区
while(flag[i]&&turn==i);//进入区
critical section;//临界区
flag[j] = False;//退出区
remainder section;//剩余区
```

具体如下：考虑进程 Pi，一旦设置 flag[i]=true，则表示进程 Pi 想要进入临界区，同时 turn=j，此时若 Pj 已经在临界区中，则符合进程 Pi 的循环条件，则 Pi 不能进入临界区。若 Pj 不想进入临界区，即 flag[j]=false，循环条件不符合，则 Pi 可以顺利进入，反之亦然。

例如，根据 Peterson 算法，需要提前声明两组变量，数组 flag 和整数 turn。flag[i]=true 表示子线程 i 想要进入临界区。Turn=j，表示子线程 j 在临界区里。因此在对应子线程运行前、子线程每次循环完、子线程运行结束，需要初始化对应子线程的数据。来保证两个子线程有且仅能有一个在临界区里面。避免非原子操作会被干扰。

对于**实验 1.3**，需要使用 Windows 操作系统的锁，利用信号量机制，来解决线程的互斥与同步问题。其核心的操作是 P 操作和 V 操作。P 操作为加锁，V 操作为解锁。其基本结构为：

```
P 操作;//进入区
critical section;//临界区
V 操作;//退出区
remainder section;//剩余区
```

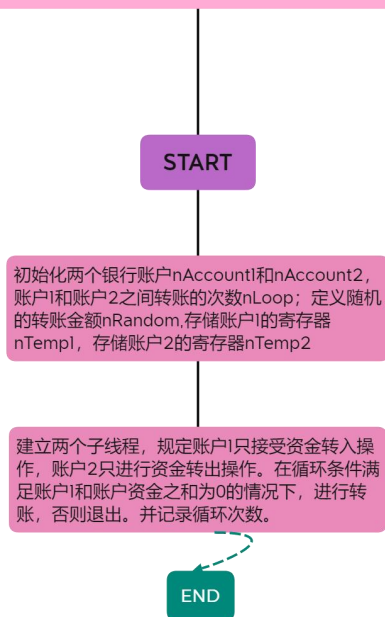
在同步问题中，若某个行为需要用到某种资源，则在这个行为前面 P 这种资源；若某个行为会提供某种资源，则在这个行为后面 V 这种资源。在互斥问题中，P, V 操作要紧夹使用互斥资源的那个行为，中间不能有其他冗余代码。

对于**实验 1.4**，是在实验 1.2 和实验 1.3 的基础上，比较 Peterson 算法同步机制和 Windows 操作系统同步机制的效率。我们分别在线程正式运行前记录系统时间（记作 begin），然后在线程运行完成后记录系统时间（记作 end），衡量效率我们采用时间的差值（end-begin）。时间越短，则相应的同步机制效率越高。

2. 实验流程图

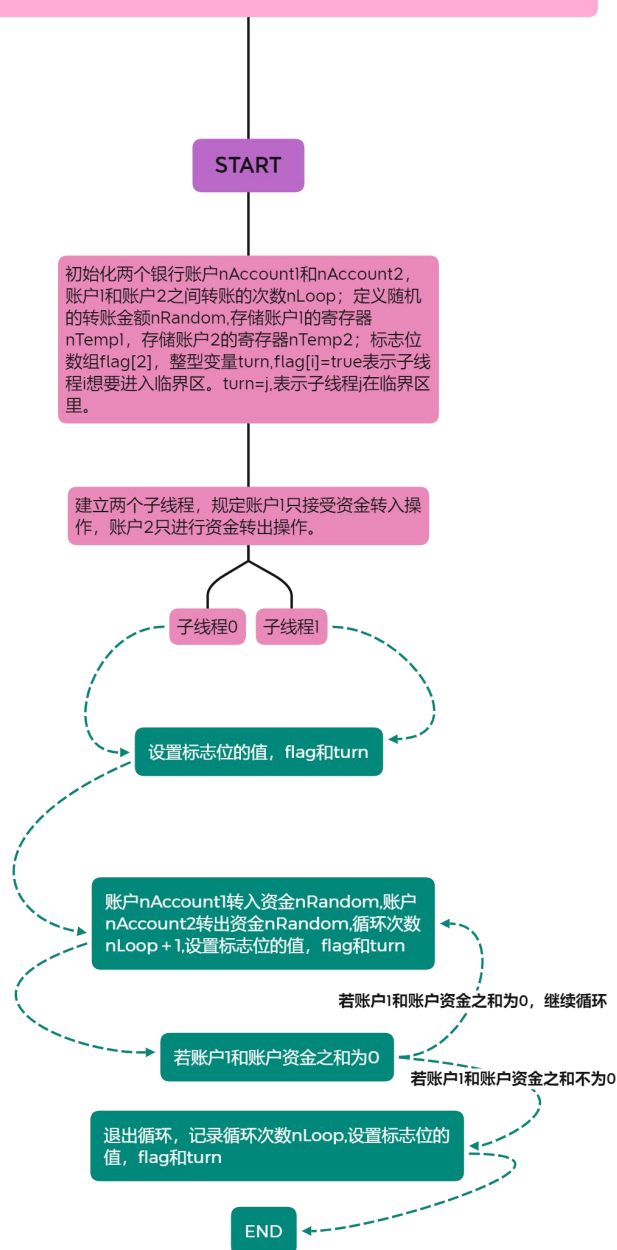
实验 1.1

实验1.1：银行账户转账同步问题的抽象及未采取同步控制情况下的编程实现



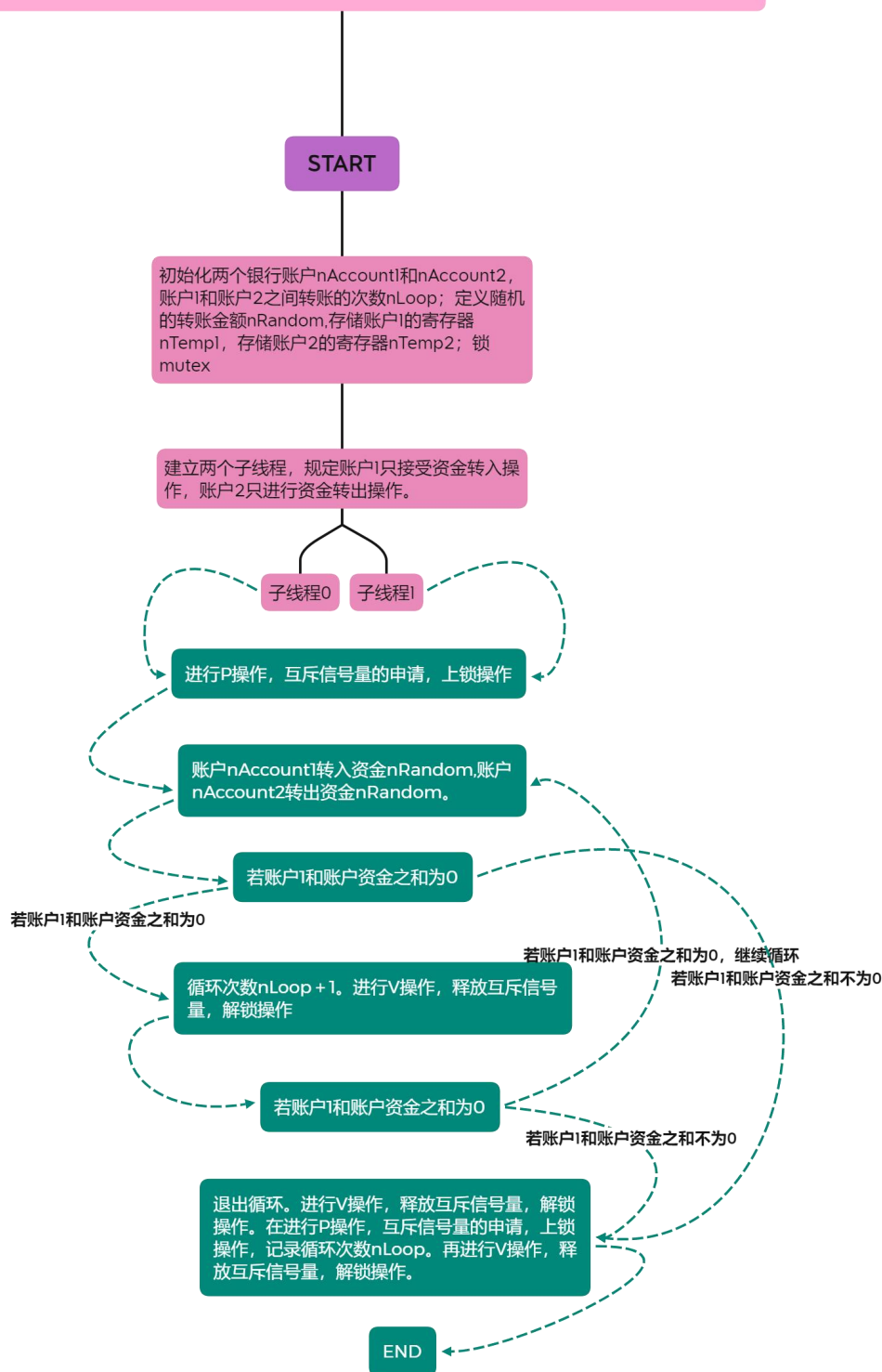
实验 1.2

实验1.2：基于Peterson算法的银行账户转账同步问题解决方案



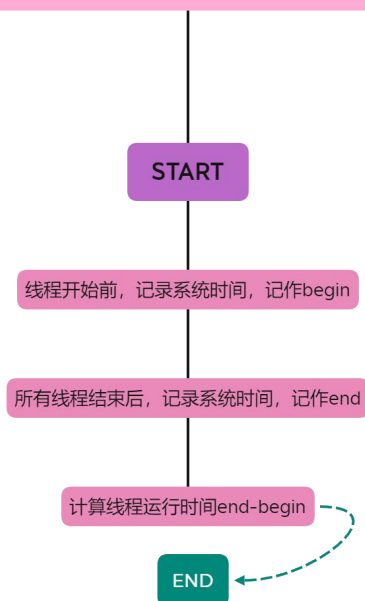
实验 1.3

实验1.3：基于Windows(或Linux) 操作系统同步机制的银行账户转账同步问题解决方案



实验 1.4

实验1.4：Peterson算法同步机制和 Windows（或Linux）操作系统同步机制的效率比较



五、关键代码及说明

1. 数据结构及符号说明

实验 1.1

```
int nAccount1=0;//初始账户1
int nAccount2=0;//初始账户2
int nLoop=0;//账户1和账户2之间转账的次数
int nTemp1,nTemp2,nRandom;//分别代表了：nTemp1模拟存储账户1的寄存器；nTemp2模拟存储账户2的寄存器
//nRandom代表随机转账的金额
```

实验 1.2(实验 1.4)

```
int nAccount1=0;//初始账户1
int nAccount2=0;//初始账户2
int nLoop=0;//账户1和账户2之间转账的次数
int nTemp1,nTemp2,nRandom;//分别代表了：nTemp1模拟存储账户1的寄存器；nTemp2模拟存储账户2的寄存器
//nRandom代表随机转账的金额
bool flag[2];//peterson算法标志位
int turn;//peterson算法标志位
```

定义宏：

```
#define TOTAL (100000)
```

实验 1.3 (实验 1.4)

```
int nAccount1=0;//初始账户1
int nAccount2=0;//初始账户2
int nTemp1,nTemp2,nRandom;//分别代表了：nTemp1模拟存储账户1的寄存器；nTemp2模拟存储账户2的寄存器
//nRandom代表随机转账的金额
HANDLE mutex;//windows互斥信号量
```

定义宏：

```
#define TOTAL (100000)
```

2. 函数说明

实验 1.1

函数	说明
DWORD WINAPI Thread_Func_Not_synced (HANDLE Thread)	银行账户转账同步问题的抽象及未采取同步控制情况下的编程实现，包含线程 0 和线程 1
void solution()	实验 1.1 主逻辑函数

实验 1.2 (实验 1.4)

函数	说明
DWORD WINAPI Thread_Func_peterson0 (HANDLE Thread)	基于 Peterson 算法的银行账户转账同步问题解决方案，子线程 0
void solution()	实验 1.2 主逻辑函数
DWORD WINAPI Thread_Func_peterson1 (HANDLE Thread)	基于 Peterson 算法的银行账户转账同步问题解决方案，子线程 1
void change(int num)	对于线程 ID 为 num 的线程，进行标志位的初始化

实验 1.3 (实验 1.4)

函数	说明
DWORD WINAPI Thread_Func_windows (HANDLE Thread)	基于 Windows (或 Linux) 操作系统同步机制的银行账户转账同步问题解决方案，包含线程 0 和线程 1
void solution()	实验 1.3 主逻辑函数

3. 关键代码

实验 1.1

1. DWORD WINAPI Thread_Func_Not_synced

```
DWORD WINAPI Thread_Func_Not_synced(HANDLE Thread){
    do{
        nRandom=rand();
        nTemp1=nAccount1;
        nTemp2=nAccount2;
        nAccount1=nTemp1+nRandom;
        nAccount2=nTemp2-nRandom;
        nLoop++;
    }while((nAccount1+nAccount2)==0);//转账线程的主体逻辑结构
    cout<<"The number of cycles is "<<nLoop<<endl;//记录转账的次数，即循环次数
    return 0;
} //银行账户转账同步问题的抽象及未采取同步控制情况下的编程实现
```

此函数主要是银行账户转账同步问题的抽象及未采取同步控制情况下的编程实现。函数已经存在注释。

2. void solution()

```
void solution(){
    HANDLE Thread[2]; //创建两个线程对象
    if((Thread[0]=CreateThread(NULL,0,Thread_Func_Not_synced,NULL,0,NULL))==NULL){
        cout<<"Thread: Thread_Func_Not_synced-0 failed to create!"<<endl;
        exit(0);
    } //实例化线程对象 0, 若实例化失败，则直接退出程序
    if((Thread[1]=CreateThread(NULL,0,Thread_Func_Not_synced,NULL,0,NULL))==NULL){
        cout<<"Thread: Thread_Func_Not_synced-1 failed to create!"<<endl;
        exit(0);
    } //实例化线程对象 1, 若实例化失败，则直接退出程序
    WaitForMultipleObjects(2,Thread,true,INFINITE); //实现主线程对两个银行账户转账的等待
    CloseHandle(Thread); //关闭线程句柄
}
```

此函数为实验 1.1 的主逻辑函数。首先创建两个线程并实例化，然后运行子线程，等待主线程对于子线程的结束响应。函数已经存在注释。

实验 1.2 (实验 1.4)

1. void change(int num)

```
void change(int num){
    int temp=1-num; //计算另外一个线程的 ID
    turn=temp; //赋值给 turn, 表示允许另外一个线程进入临界区
    flag[num]=true; //将本线程标志位设置为 true, 表示本线程想要进入临界区
    while(flag[temp]&&turn==temp){
```

```
}; //判断若另外一个线程想要进入临界区并且也允许其进入临界区，则本线程等待  
} //对于线程 ID 为 num 的线程，进行标志位的初始化
```

此函数为 Peterson 算法必要的对标志位数组 flag 和整数 turn 进行数据修改的操作。涵盖了 Peterson 算法的核心思想。

2. DWORD WINAPI Thread_Func_peterson0 和 DWORD WINAPI Thread_Func_peterson1

这两个函数结构比较相似，因此仅仅粘贴一个函数的代码。

```
DWORD WINAPI Thread_Func_peterson0(HANDLE Thread){  
    change(0); //对 ID 为 0 的线程进行数据初始化  
    do{  
        nRandom=rand();  
        nTemp1=nAccount1;  
        nTemp2=nAccount2;  
        nAccount1=nTemp1+nRandom;  
        nAccount2=nTemp2-nRandom; //进行转账  
        if(nAccount1+nAccount2!=0){  
            break;  
        } //若此时两个账户资金之和不为 0，跳出循环  
        nLoop++; //若仍然满足循环条件，循环次数+1  
        flag[0]=false; //执行完成一次，设置标志位为 false  
        change(0); //对 ID 为 0 的线程进行数据修改  
    } while((nAccount1+nAccount2)==0 && nLoop<(TOTAL));  
    flag[0]=false; //执行完成，设置标志位为 false  
    change(0); //对 ID 为 0 的线程进行数据修改  
    flag[0]=false; //设置标志位为 false  
    cout<<"The number of cycles (thread 0) is "<<nLoop<<endl;  
    return 0;  
}
```

此函数为基于 Peterson 算法的银行账户转账同步问题的算法。主要是在每次转帐前和转账后及子线程整个转账操作结束后，需要调用函数 void change(int num) 来重置标志位。并且记录循环次数。

3. void solution()

```
void solution(){  
    HANDLE Thread[2]; //创建两个线程对象  
    DWORD begin,end; //定义两个系统时间  
    begin=GetTickCount(); //获取线程运行开始前的系统时间  
    if((Thread[0]=CreateThread(NULL,0,Thread_Func_peterson0,NULL,0,NULL))==NULL){  
        cout<<"Thread: Thread_Func_peterson0-0 failed to create!"<<endl;  
        exit(0);  
    } //实例化线程对象 0, 若实例化失败，则直接退出程序  
    if((Thread[1]=CreateThread(NULL,0,Thread_Func_peterson1,NULL,0,NULL))==NULL){  
        cout<<"Thread: Thread_Func_peterson1-1 failed to create!"<<endl;  
        exit(0);  
    } //实例化线程对象 1, 若实例化失败，则直接退出程序
```

```

WaitForMultipleObjects(2,Thread,true,INFINITE);//主线程等待所有子线程的通知
end=GetTickCount();//获取线程运行结束的系统时间
cout<<"The total running time of the thread is "<<end-begin<<endl;//打印时间差
CloseHandle(Thread);//关闭子线程
}

```

此函数主要是实验 1.2 的主逻辑函数。和实验 1.1 中的不同点是，在子线程运行开始和运行结束记录下系统时间。

实验 1.3 (实验 1.4)

1. DWORD WINAPI Thread_Func_windows(HANDLE Thread)

```

DWORD WINAPI Thread_Func_windows(HANDLE Thread){
    int nLoop=0;//账户 1 和账户 2 之间转账的次数
    WaitForSingleObject(mutex,INFINITE);//对互斥信号量的申请，上锁操作
    do{
        nRandom=rand();
        nTemp1=nAccount1;
        nTemp2=nAccount2;
        nAccount1=nTemp1+nRandom;
        nAccount2=nTemp2-nRandom;//进行转账
        if(nAccount1+nAccount2!=0){
            break;
        }//若此时两个账户资金之和不为 0，跳出循环
        nLoop++;//若仍然满足循环条件，循环次数+1
        ReleaseMutex(mutex);//对互斥信号量的释放，解锁操作
        WaitForSingleObject(mutex,INFINITE);//对互斥信号量的申请，上锁操作
    }while((nAccount1+nAccount2)==0&& nLoop<(TOTAL));
    ReleaseMutex(mutex);//对互斥信号量的释放，解锁操作
    WaitForSingleObject(mutex,INFINITE);//对互斥信号量的申请，上锁操作
    cout<<"The number of cycles is "<<nLoop<<endl;//打印循环次数
    ReleaseMutex(mutex);//对互斥信号量的释放，解锁操作
    return 0;
}

```

此函数为基于 Windows (或 Linux) 操作系统同步机制的银行账户转账同步问题解决方案，包含线程 0 和线程 1。

2. void solution()

```

void solution(){
    HANDLE Thread[2];//创建两个线程对象
    DWORD begin,end;//定义两个系统时间
    begin=GetTickCount();//获取线程运行开始前的系统时间
    mutex=CreateMutex(NULL,false,NULL);//创建互斥信号量
    if((Thread[0]=CreateThread(NULL,0,Thread_Func_windows,NULL,0,NULL))==NULL){
        cout<<"Thread: Thread_Func_windows-0 failed to create!"<<endl;
        exit(0);
    }//实例化线程对象 0,若实例化失败，则直接退出程序
}

```

```

if((Thread[1]=CreateThread(NULL,0,Thread_Func_windows,NULL,0,NULL))==NULL){
    cout<<"Thread: Thread_Func_windows-1 failed to create!"<<endl;
    exit(0);
}
//实例化线程对象 1,若实例化失败,则直接退出程序
WaitForMultipleObjects(2,Thread,true,INFINITE);//主线程等待两个子线程的响应
end=GetTickCount();//获取线程运行结束的系统时间
cout<<"The total running time of the thread is "<<end-begin<<endl;//打印时间差
CloseHandle(Thread);//关闭子线程
CloseHandle(mutex);//关闭互斥信号量
}

```

此函数主要是实验 1.3 的主逻辑函数。和实验 1.1 中的不同点是,在子线程运行开始和运行结束记录下系统时间,并且新增了互斥信号量的相关操作。

六、实验结果

实验 1.1 子线程未同步操作

主函数:

```

int main(){
    solution();
    system("pause");
}

```

实验结果:

```

c:\VS\code\c and c++\os\work1.1.exe
The number of cycles is 195573
_

```

子线程的总运行次数为 195573 次。

下面进行多次实验:

<pre> c:\VS\code\c and c++\os\work1.1.exe The number of cycles is 28189 _ </pre>	<pre> c:\VS\code\c and c++\os\work1.1.exe The number of cycles is 26747 _ </pre>
--	--

实验 1.2 基于 Peterson 算法的同步操作

主函数:

```

int main(){
    solution();
    system("pause");
}

```

实验结果:

```
c:\VS\code\c and c++\os\work1.2.exe
The number of cycles (thread 0) is The number of cycles (thread 1) is 100000100000
The total running time of the thread is 16
```

下面进行多次实验，修改宏 TOTAL（分别为 100000，1000000，10000000）的值：

```
c:\VS\code\c and c++\os\work1.2.exe
The number of cycles (thread 0) is The number of cycles (thread 1) is 100000100000
The total running time of the thread is 16
```

```
c:\VS\code\c and c++\os\work1.2.exe
The number of cycles (thread 0) is The number of cycles (thread 1) is 695092695092
The total running time of the thread is 63
```

```
c:\VS\code\c and c++\os\work1.2.exe
The number of cycles (thread 1) is 1670597
The number of cycles (thread 0) is 10000000
The total running time of the thread is 266
```

实验 1.3 基于 Windows 互斥信号量的同步操作

主函数：

```
int main(){
    solution();
    system("pause");
}
```

实验结果：

```
c:\VS\code\c and c++\os\work1.3.exe
The number of cycles is 100000
The number of cycles is 100000
The total running time of the thread is 656
```

下面进行多次实验，修改宏 TOTAL（分别为 100000，1000000，10000000）的值：

```
c:\VS\code\c and c++\os\work1.3.exe
The number of cycles is 100000
The number of cycles is 100000
The total running time of the thread is 656
```

```
c:\VS\code\c and c++\os\work1.3.exe
The number of cycles is 1000000
The number of cycles is 1000000
The total running time of the thread is 5969
```

```
c:\VS\code\c and c++\os\work1.3.exe
The number of cycles is 1000000
The number of cycles is 1000000
The total running time of the thread is 60390
```

在实验收集数据中发现，结果打印出来需要等待的时间明显远远大于 Peterson 算法的时间，在宏 TOTAL 等于 10000000 时，甚至等待了大概 1 分钟结果才出来。

实验 1.4 Peterson 算法和 Windows 操作系统同步机制的效率比较

在这里我们建立了一个三线表格，使得数据更加直观：

宏 TOTAL	Peterson 算法同步机制时间差	Windows 操作系统同步机制
100000	16	656
1000000	63	5969
10000000	266	60390

为了避免实验的偶然性，我们对于确定的宏 TOTAL 进行多次实验：

宏 TOTAL	Peterson 算法同步机制时间差	Windows 操作系统同步机制
100000	0, 15, 15, 16	640, 656, 656, 671
1000000	0, 63, 63, 94	5934, 5969, 6004, 6117
10000000	266, 328, 390, 578	59997, 60390, 60667, 60419

实验 1.4 结果分析：没有应用互斥信号量对线程进行并发控制，运行会产生错误；利用 windows 互斥信号量后，两存取款线程可并发正确执行。成功转账。但加大了系统的时间开销。时间效率低；应用同步机制的 Peterson 算法后，两线程也可顺利的并发执行，成功转账，但相对于 Windows 互斥信号量，时间明显缩短。改善了系统的运行时间，提高了时间效率。

通过本次试验实现了两个线程的并发运行。对于共享数据资源但不受控制的两个线程的并发运行，会出现错误结果。因此同步机制是必要的。实验中利用 Windows 互斥信号量和同步机制的 Peterson 算法实现了关于该两个线程的有序控制，实现了同步机制和用于同一问题的解决。并了解了两种方案的效率。同步机制的 Peterson 算法要明显优于 Windows 互斥信号量。

七、实验总结

通过本次实验，我对于进程的同步与互斥有了更深的了解，加深巩固了对于 Peterson 算法的理解和认识，学习了基本的 Windows 编程，获益匪浅。

八、实验源代码

实验 1.1 源代码

```
#include<iostream>
#include<string>
#include<cstdlib>
#include<ctime>
#include<windows.h>
#include<stdlib.h>
#include<stdio.h>
```

```

#define random(a,b) (rand()%(b-a+1)+a)
using namespace std;

int nAccount1=0;//初始账户 1
int nAccount2=0;//初始账户 2
int nLoop=0;//账户 1 和账户 2 之间转账的次数
int nTemp1,nTemp2,nRandom;//分别代表了: nTemp1 模拟存储账户 1 的寄存器; nTemp2 模拟存储账户 2 的寄存器
//nRandom 代表随机转账的金额
DWORD WINAPI Thread_Func_Not_synced(HANDLE Thread){
    do{
        nRandom=rand();
        nTemp1=nAccount1;
        nTemp2=nAccount2;
        nAccount1=nTemp1+nRandom;
        nAccount2=nTemp2-nRandom;
        nLoop++;
    }while((nAccount1+nAccount2)==0);//转账线程的主体逻辑结构
    cout<<"The number of cycles is "<<nLoop<<endl;//记录转账的次数,即循环次数
    return 0;
}

//银行账户转账同步问题的抽象及未采取同步控制情况下的编程实现
void solution(){
    HANDLE Thread[2];//创建两个线程对象
    if((Thread[0]=CreateThread(NULL,0,Thread_Func_Not_synced,NULL,0,NULL))==NULL){
        cout<<"Thread: Thread_Func_Not_synced-0 failed to create!"<<endl;
        exit(0);
    }
    //实例化线程对象 0,若实例化失败,则直接退出程序
    if((Thread[1]=CreateThread(NULL,0,Thread_Func_Not_synced,NULL,0,NULL))==NULL){
        cout<<"Thread: Thread_Func_Not_synced-1 failed to create!"<<endl;
        exit(0);
    }
    //实例化线程对象 1,若实例化失败,则直接退出程序
    WaitForMultipleObjects(2,Thread,true,INFINITE);//实现主线程对两个银行账户转账的等待
    CloseHandle(Thread);//关闭线程句柄
}

int main(){
    solution();
    system("pause");
}

```

实验 1.2 (实验 1.4) 源代码

```

#include<iostream>
#include<string>
#include<cstdlib>
#include<ctime>
#include<windows.h>

```



```

#include<stdlib.h>
#include<stdio.h>
#define random(a,b) (rand()% (b-a+1)+a)
#define TOTAL (10000000)
using namespace std;

int nAccount1=0;//初始账户 1
int nAccount2=0;//初始账户 2
int nLoop=0;//账户 1 和账户 2 之间转账的次数
int nTemp1,nTemp2,nRandom;//分别代表了: nTemp1 模拟存储账户 1 的寄存器; nTemp2 模拟存储账户 2 的寄存器
//nRandom 代表随机转账的金额
bool flag[2]; //peterson 算法标志位
int turn; //peterson 算法标志位
void change(int num){
    int temp=1-num;//计算另外一个线程的 ID
    turn=temp;//赋值给 turn,表示允许另外一个线程进入临界区
    flag[num]=true;//将本线程标志位设置为 true,表示本线程想要进入临界区
    while(flag[temp]&&turn==temp){
        ;//判断若另外一个线程想要进入临界区并且也允许其进入临界区,则本线程等待
    }
} //对于线程 ID 为 num 的线程,进行标志位的初始化
DWORD WINAPI Thread_Func_peterson0(HANDLE Thread){
    change(0);//对 ID 为 0 的线程进行数据初始化
    do{
        nRandom=rand();
        nTemp1=nAccount1;
        nTemp2=nAccount2;
        nAccount1=nTemp1+nRandom;
        nAccount2=nTemp2-nRandom;//进行转账
        if(nAccount1+nAccount2!=0){
            break;
        } //若此时两个账户资金之和不为 0,跳出循环
        nLoop++; //若仍然满足循环条件,循环次数+1
        flag[0]=false;//执行完成一次,设置标志位为 false
        change(0);//对 ID 为 0 的线程进行数据修改
    }while((nAccount1+nAccount2)==0&& nLoop<(TOTAL));
    flag[0]=false;//执行完成,设置标志位为 false
    change(0);//对 ID 为 0 的线程进行数据修改
    flag[0]=false;//设置标志位为 false
    cout<<"The number of cycles (thread 0) is "<<nLoop<<endl;
    return 0;
}

```

```

DWORD WINAPI Thread_Func_peterson1(HANDLE Thread){
    change(1);//对 ID 为 1 的线程进行数据初始化
}

```



```

do{
    nRandom=rand();
    nTemp1=nAccount1;
    nTemp2=nAccount2;
    nAccount1=nTemp1+nRandom;
    nAccount2=nTemp2-nRandom;//进行转账
    if(nAccount1+nAccount2!=0){
        break;
    }//若此时两个账户资金之和不为0，跳出循环
    nLoop++;//若仍然满足循环条件，循环次数+1
    flag[1]=false;//执行完成一次，设置标志位为 false
    change(1);//对 ID 为 1 的线程进行数据修改
}while((nAccount1+nAccount2)==0&& nLoop<(TOTAL));
flag[1]=false;//执行完成，设置标志位为 false
change(1);//对 ID 为 1 的线程进行数据修改
flag[1]=false;//设置标志位为 false
cout<<"The number of cycles (thread 1) is "<<nLoop<<endl;
return 0;
}

void solution(){
    HANDLE Thread[2];//创建两个线程对象
    DWORD begin,end;//定义两个系统时间
    begin=GetTickCount();//获取线程运行开始前的系统时间
    if((Thread[0]=CreateThread(NULL,0,Thread_Func_peterson0,NULL,0,NULL))==NULL){
        cout<<"Thread: Thread_Func_peterson0-0 failed to create!"<<endl;
        exit(0);
    }//实例化线程对象 0,若实例化失败，则直接退出程序
    if((Thread[1]=CreateThread(NULL,0,Thread_Func_peterson1,NULL,0,NULL))==NULL){
        cout<<"Thread: Thread_Func_peterson1-1 failed to create!"<<endl;
        exit(0);
    }//实例化线程对象 1,若实例化失败，则直接退出程序
    WaitForMultipleObjects(2,Thread,true,INFINITE);//主线程等待所有子线程的通知
    end=GetTickCount();//获取线程运行结束的系统时间
    cout<<"The total running time of the thread is "<<end-begin<<endl;//打印时间差
    CloseHandle(Thread);//关闭子线程
}

int main(){
    solution();
    system("pause");
}

```

实验 1.3（实验 1.4）源代码

```
#include<iostream>
#include<string>
#include<cstdlib>
#include<ctime>
#include<windows.h>
#include<stdlib.h>
#include<stdio.h>
#define random(a,b) (rand()%(b-a+1)+a)
#define TOTAL (100000)
using namespace std;

int nAccount1=0;//初始账户 1
int nAccount2=0;//初始账户 2
int nTemp1,nTemp2,nRandom;//分别代表了: nTemp1 模拟存储账户 1 的寄存器; nTemp2 模拟存储账户 2 的寄存器
//nRandom 代表随机转账的金额
HANDLE mutex;//windows 互斥信号量
DWORD WINAPI Thread_Func_windows(HANDLE Thread){
    int nLoop=0;//账户 1 和账户 2 之间转账的次数
    WaitForSingleObject(mutex,INFINITE);//对互斥信号量的申请,上锁操作
    do{
        nRandom=rand();
        nTemp1=nAccount1;
        nTemp2=nAccount2;
        nAccount1=nTemp1+nRandom;
        nAccount2=nTemp2-nRandom;//进行转账
        if(nAccount1+nAccount2!=0){
            break;
        }//若此时两个账户资金之和不为 0,跳出循环
        nLoop++;//若仍然满足循环条件,循环次数+1
        ReleaseMutex(mutex);//对互斥信号量的释放,解锁操作
        WaitForSingleObject(mutex,INFINITE);//对互斥信号量的申请,上锁操作
    }while((nAccount1+nAccount2)==0&& nLoop<(TOTAL));
    ReleaseMutex(mutex);//对互斥信号量的释放,解锁操作
    WaitForSingleObject(mutex,INFINITE);//对互斥信号量的申请,上锁操作
    cout<<"The number of cycles is "<<nLoop<<endl;//打印循环次数
    ReleaseMutex(mutex);//对互斥信号量的释放,解锁操作
    return 0;
}

void solution(){
    HANDLE Thread[2];//创建两个线程对象
    DWORD begin,end;//定义两个系统时间
    begin=GetTickCount();//获取线程运行开始前的系统时间
    mutex=CreateMutex(NULL,false,NULL);//创建互斥信号量
```

```
if((Thread[0]=CreateThread(NULL,0,Thread_Func_windows,NULL,0,NULL))==NULL){
    cout<<"Thread: Thread_Func_windows-0 failed to create!"<<endl;
    exit(0);
} //实例化线程对象 0,若实例化失败, 则直接退出程序
if((Thread[1]=CreateThread(NULL,0,Thread_Func_windows,NULL,0,NULL))==NULL){
    cout<<"Thread: Thread_Func_windows-1 failed to create!"<<endl;
    exit(0);
} //实例化线程对象 1,若实例化失败, 则直接退出程序
WaitForMultipleObjects(2,Thread,true,INFINITE); //主线程等待两个子线程的响应
end=GetTickCount(); //获取线程运行结束的系统时间
cout<<"The total running time of the thread is "<<end-begin<<endl; //打印时间差
CloseHandle(Thread); //关闭子线程
CloseHandle(mutex); //关闭互斥信号量
}
int main(){
    solution();
    system("pause");
}
```

同步机制(2)

一、实验目的

探索、理解并掌握操作系统同步机制的应用编程方法，针对典型的同步问题，构建基于 Windows(或 Linux)操作系统同步机制的解决方案。

二、实验环境

1.实验类型：设计型。

2.实验环境：

操作系统：

windows10

调试软件名称及版本号

Visual Studio Code

February 2022 (version 1.65)

编程语言及版本号

C++/C

开发环境、运行环境及测试环境：

Visual Studio Code

February 2022 (version 1.65)

上机地点

班级未集中进行实验课

机器台号

华硕飞行堡垒 8

三、实验内容

了解、熟悉和运用 Windows(或 Linux)操作系统同步机制及编程方法，针对典型的同步问题，譬如生产者-消费者问题、读者优先的读者写者问题、写者优先的读者写者问题、哲学家就餐问题等(任选两个即可)，编程模拟实现相应问题的解决方案。

本实验选择了生产者-消费者问题(记作**实验 2.1**)，哲学家就餐问题(记作**实验 2.2**)，读者优先的读者写者问题(记作**实验 2.3**)、写者优先的读者写者问题(记作**实验 2.4**)。由于实验 2.3 和实验 2.4 十分相似，因而在本作业中我将**实验 2.3** 和**实验 2.4** 合并为**实验 2.3_4**。

四、实验流程

本次实验包含以下实验内容：

(1) 生产者-消费者问题

问题描述：

一组生产者进程和一组消费者进程共享了一个初始为空、大小为 n 的缓冲区，只有缓冲区没满时，生产者才能把消息放入缓冲区，否则必须等待；只有缓冲区不空，消费者才能从中取出消息，否则必须等待。由于缓冲区是临界资源，它允许一个生产者放入消息，或一个消费者从中取出消息。

问题分析：

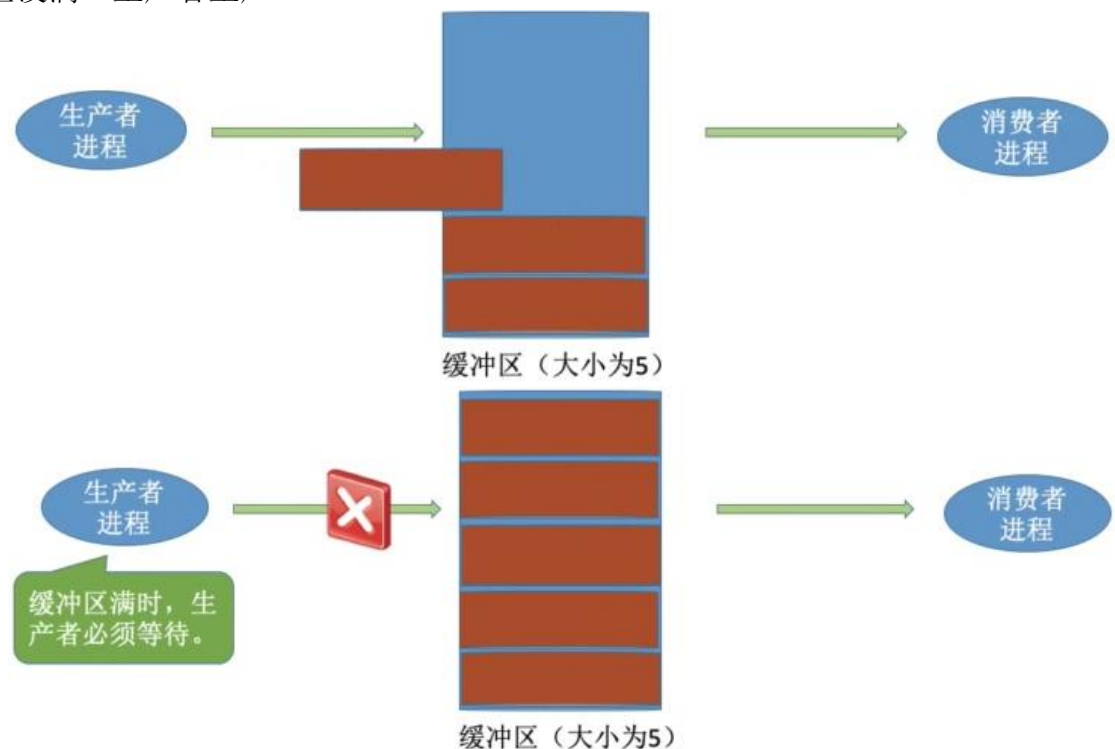
①关系分析。生产者和消费者对缓冲区互斥访问是互斥关系，同时生产者和消费者又是一个相互协作的关系。只有生产者生产后，消费者才能消费，他们也是同步关系。

②整理思路。只有生产者和消费者两个进程，正好是这两个进程存在着互斥关系和同步关系。那么需要解决的是互斥和同步 PV 操作的位置。

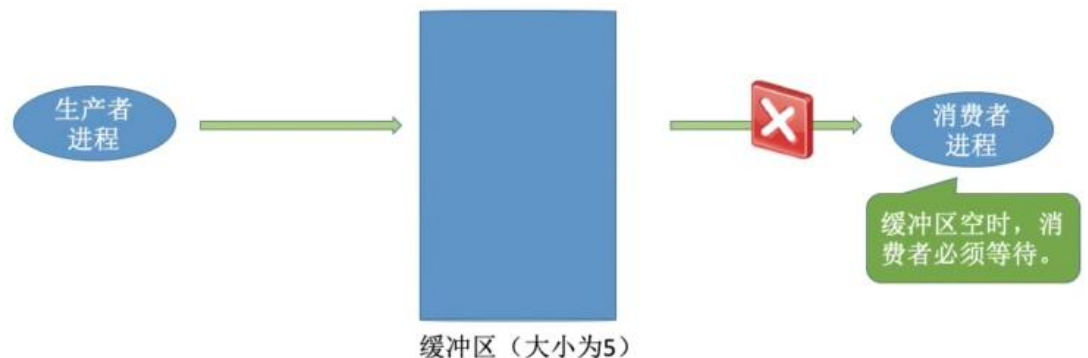
③信号量设置。信号量 mutex 作为互斥信号量，用于控制互斥访问缓冲区，互斥信号量初值为 1；信号量 full 用于记录当前缓冲池中的满缓冲区数量，初值为 0。信号量 empty 用于记录当前缓冲池中的空缓冲区数量，初值为 n。

实验流程图：

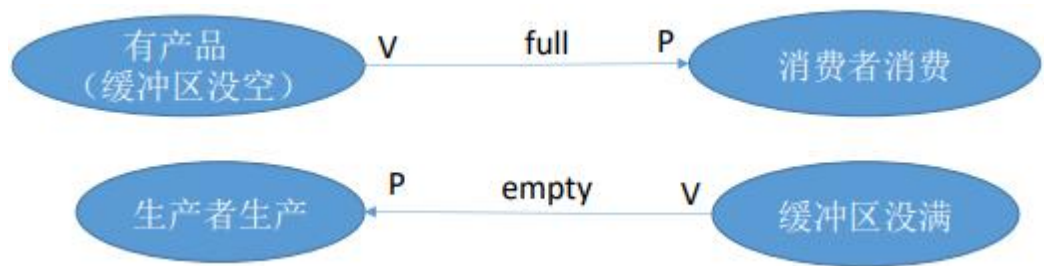
只有缓冲区没满时，生产者才能把产品放入缓冲区，否则必须等待。缓冲区没满→生产者生产。



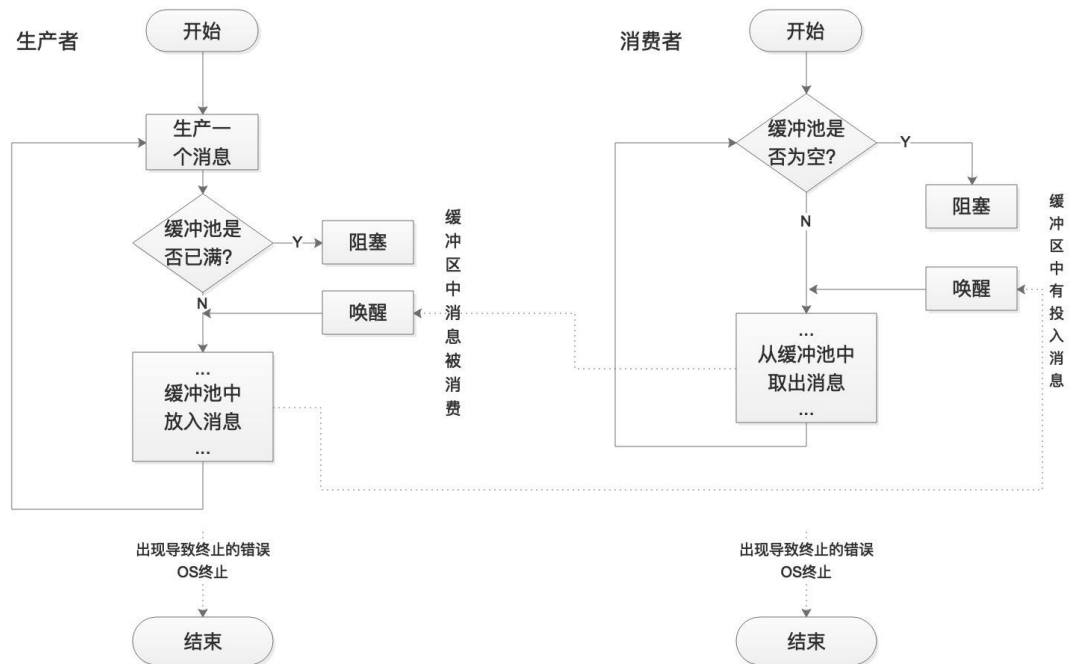
只有缓冲区不空时，消费者才能从中取出产品，否则必须等待。缓冲区没空→消费者消费。



缓冲区是临界资源，各进程必须互斥地访问。



总的来说，实验流程图如下：



(2) 哲学家就餐问题

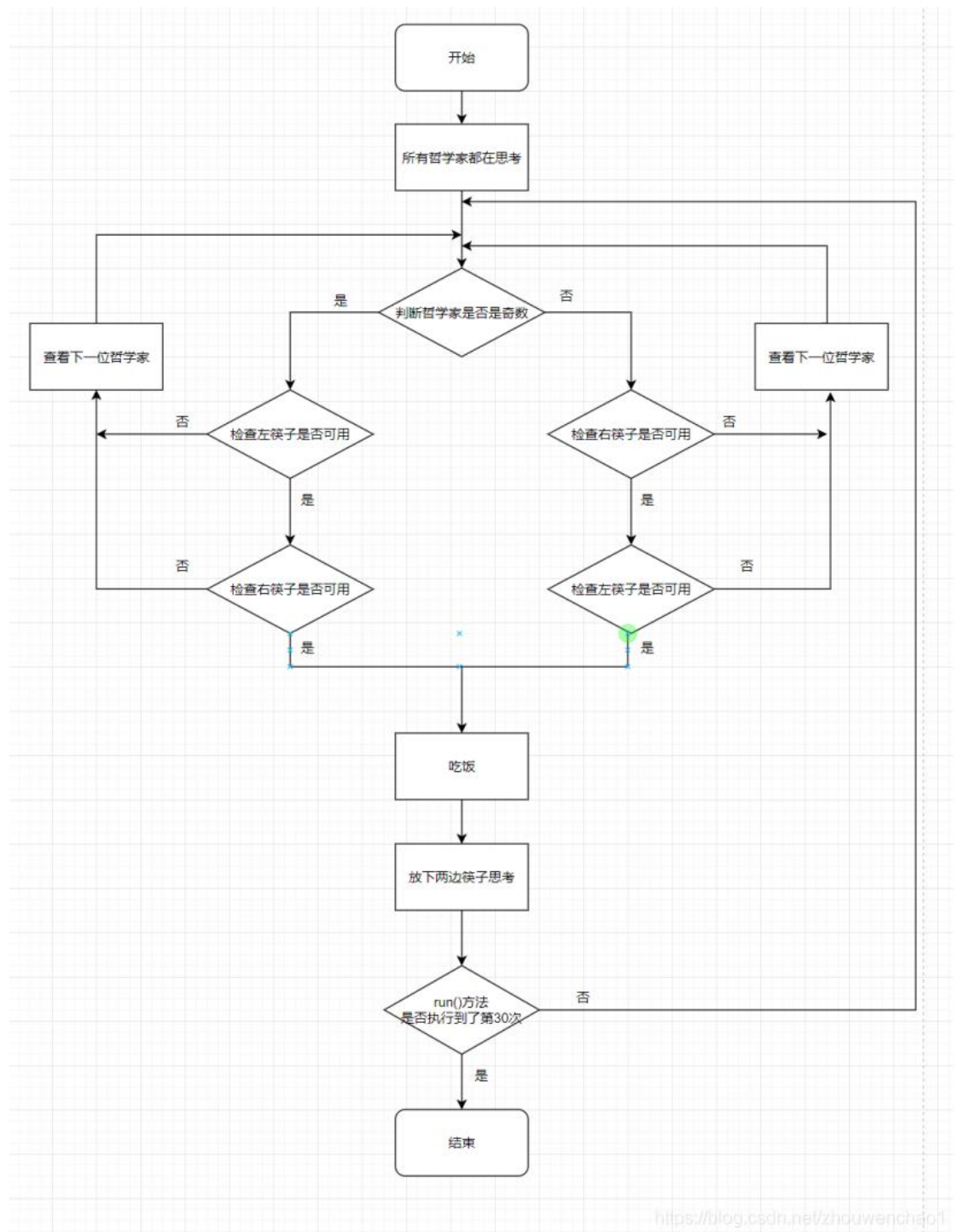
问题描述：

一张圆桌上坐着 5 名哲学家，每两名哲学家之间的桌上摆一根筷子，两根筷子中间是一碗白米饭。哲学家们倾注毕生精力用于思考和进餐，哲学家在思考时，不影响他人。只有当哲学家饥饿时，才视图拿起左右两根筷子（一根一根的拿起）。若筷子已经在其他人手上，则需要等待。饥饿的哲学家只有同时拿到了两根筷子才可以开始进餐，进餐完毕后，放下筷子继续思考。

问题分析：

- ①关系分析。5 名哲学家与左右邻居对其中间筷子的访问时互斥关系。
- ②整理思路。显然这里有 5 个进程。本题的关键是如何让一名哲学家拿到左右筷子而不造成死锁或饥饿现象。
- ③信号量设置。定义互斥信号量数组 `chopsticks[5]={1,1,1,1,1}`，用于对五个筷子的互斥访问，哲学家按照编号排序 0-4，哲学家啊 i 左边的筷子编号为 i ，右边为 $(i+1) \% 5$ 。

实验流程图:



(3)读者优先的读者写者问题

问题描述:

有一群写者和一群读者，写者在写同一本书，读者也在读这本书，多个读者可以同时读这本书，但是，只能有一个写者在写书，并且，读者必写者优先，也就是说，读者和写者同时提出请求时，读者优先。

问题分析:

即使写者发出了请求写的信号，但是只要还有读者在读取内容，就还允许其他读者继续读取内容，直到所有读者结束读取，才真正开始写

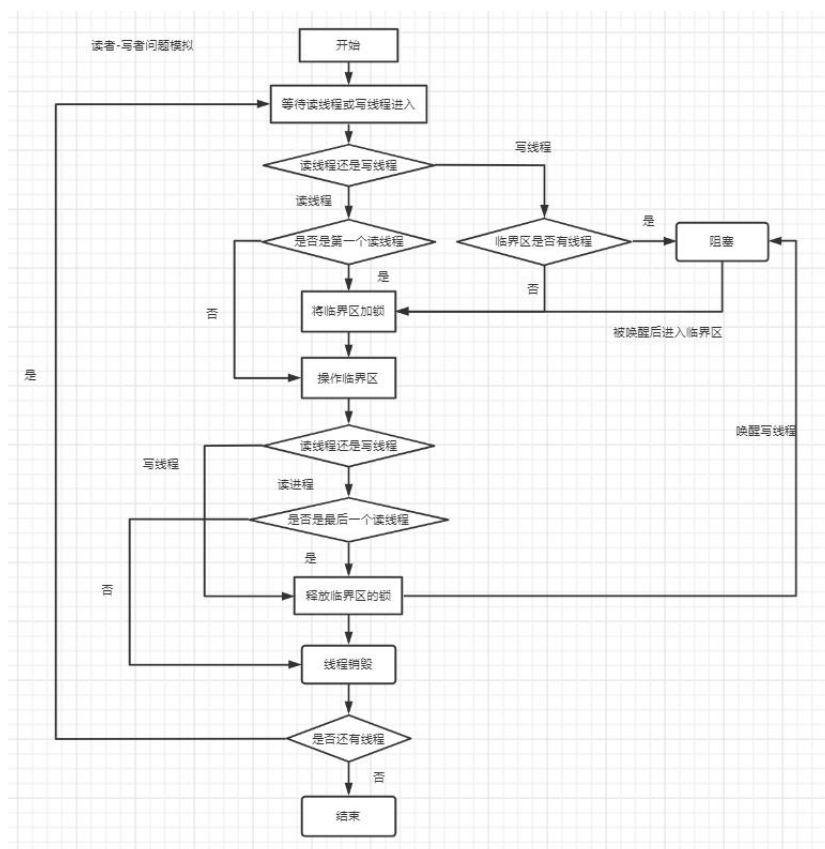
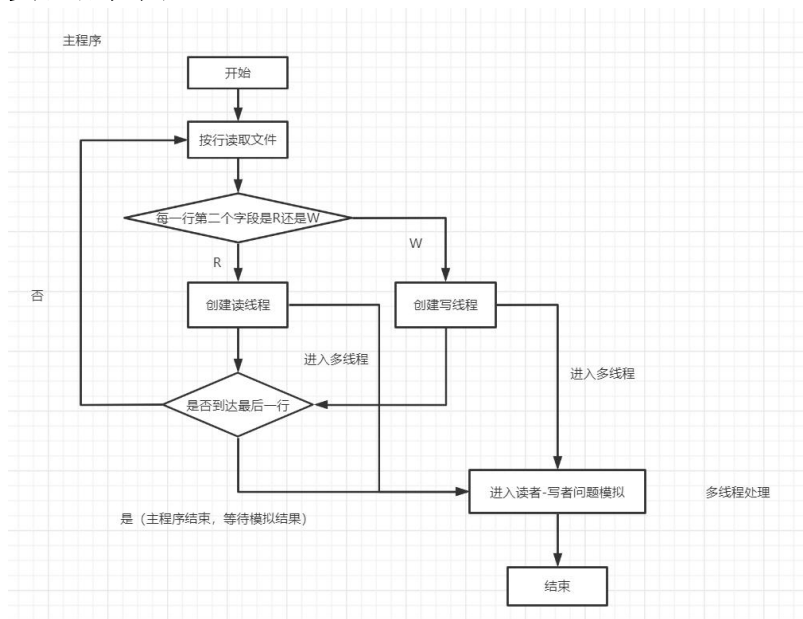
①有读者在读后面来的读者可以直接进入临界区，而已经在等待的写者继续等待直到没有任何一个读者时。

②读者之间不互斥，写者之间互斥，只能一个写，可以多个读，

③读者写者之间互斥，有写者写则不能有读者读

④如果在读访问非常频繁的场所，有可能造成写进程一直无法访问文件的局面

实验流程图：



(4) 读者优先的读者写者问题

问题描述:

有读者(reader)和写者(writer)两组并发进程, 共享一个文件, 当两个或以上的读进程同时访问共享数据时不会产生副作用, 但若某个写进程和其他进程(读进程或写进程)同时访问共享数据时则可能导致数据不一致的错误。

因此要求:

- ① 允许多个读进程可以同时同时对文件执行读操作;
- ② 只允许一个写进程往文件中写信息;
- ③ 任一写进程在完成写操作之前不允许其它读进程或写进程工作;
- ④ 写进程执行写操作前, 应让已有的读进程和写进程全部退出。

问题分析:

如果有写者申请写文件, 在申请之前已经开始读取文件的可以继续读取, 但是如果再有读者申请读取文件, 则不能够读取, 只有在所有的写者写完之后才可以读取

- ① 写者线程的优先级高于读者线程。
- ② 当有写者到来时应该阻塞读者线程的队列。
- ③ 当有一个写者正在写时或在阻塞队列时应当阻塞读者进程的读操作, 直到所有写者进程完成写操作时放开读者进程。
- ④ 当没有写者进程时读者进程应该能够同时读取文件。

实验流程图:

类似于上图, 此处省略流程图。

五、关键代码及说明

1. 数据结构及符号说明

实验 2.1 生产者消费者问题

```
const unsigned int BUFFER=5;//缓冲区长度
unsigned int PRODUCT_ID=0;//生产者ID
unsigned int CONSUME_ID=0;//消费者ID
unsigned int in;//产品进缓冲区时的缓冲区产品个数
unsigned int out;//产品出缓冲区时的缓冲区产品个数
int g_BUFFER[BUFFER];//缓冲区为循环队列
bool g_CONTINUE=true;//控制程序结束
HANDLE g_HMUTEX;//线程间的互斥对象
HANDLE g_HFULL_Semaphore;//缓冲区初始化为空, 满则生产者等待
HANDLE g_HEMPTY_Semaphore;//空闲缓冲区, 空则消费者等待

const unsigned int PRODUCE_COUNT=2;//生产者数量
const unsigned int CONSUMER_COUNT=1;//消费者数量
const unsigned int THREADS_COUNT=PRODUCE_COUNT+CONSUMER_COUNT;//线程的数据总量
```

实验 2.2 哲学家就餐问题

```
const unsigned int PERSON=5;//定义哲学家人数
const unsigned int MEALS=3;//定义每个哲学家总共需要吃三顿饭吃饱
HANDLE chopstick_semaphore[PERSON];//定义筷子的互斥信号量
HANDLE Thread[PERSON];//每个哲学家定义一个子线程
HANDLE mutex;//子线程的互斥信号量
unsigned int nums = 0;//访问的人数序号
```

定义宏：

```
#define random(a,b) (rand()%(b-a+1)+a)
```

实验 2.3_4 读者优先和写者优先的读者写者问题

```
int CountReader = 0; //用于记录当前的读者数量
int CountWriter = 0; //用于记录当前的写者数量
HANDLE readercount; //读者计数信号量
HANDLE writercount; //写者计数信号量
HANDLE rw; //用于保证读者和写者互斥地访问文件
HANDLE wait; //用于保证写优先（读者等待）
CRITICAL_SECTION FILES; //用于保护文件的临界区变量
unsigned const int Thread_NUM=10;//自定义的进程总数量
unsigned const int SLEEP_TIME=200;//模拟的运行时间

typedef struct PCB {
    string name; //进程名
    int requireTime; //请求时间
    int runningTime; //运行时间
}PCB;
```

2. 函数说明

实验 2.1 生产者消费者问题

函数	说明
int solution()	生产者消费者的主逻辑函数
DWORD WINAPI consumer(HANDLE Thread)	消费的子线程
DWORD WINAPI producer(HANDLE Thread)	生产的子线程
void consume()	消费函数
void produce()	生产函数

实验 2.2 哲学家就餐问题

函数	说明
int solution()	哲学家就餐问题的主逻辑函数
void eating(int id)	序列号为 id 的哲学家开始进餐
DWORD WINAPI Philosopher(LPVOID param)	哲学家进餐的子线程

实验 2.3_4 读者优先和写者优先的读者写者问题

函数	说明
int solution(int sign)	根据 sign 的值来决定运行读者优先还是写者优先
DWORD WINAPI WriterFisrt_writer(LPVOID lpParameter)	写者优先写者进程
DWORD WINAPI WriterFisrt_reader(LPVOID lpParameter)	写着优先读者进程
DWORD WINAPI ReaderFisrt_writer(LPVOID lpParameter)	读者优先写者进程
DWORD WINAPI ReaderFisrt_reader(LPVOID lpParameter)	读者优先读者进程

3. 关键代码

实验 2.1 生产者消费者问题

1. void produce() &&DWORD WINAPI producer(HANDLE Thread)

```
void produce(){
    cout<<"***** Produce start *****"<<endl;
    cout<<"create product: "<<++PRODUCT_ID<<endl;
    g_BUFFER[in]=PRODUCT_ID;
    in=(in+1)%BUFFER;
    cout<<"put new product in buffer....."<<endl;
    for(int i=0;i<BUFFER;i++){
        cout<<i<<" : "<<g_BUFFER[i];
        if(i==in){
            cout<<"<---- produce";
        }
        if(i==out){
            cout<<"<---- consume";
        }
        cout<<endl;
    }
    cout<<"***** Produce end *****"<<endl;
} //生产的主逻辑函数
```

此函数主要完成了在对应缓冲区进行生产。

```
DWORD WINAPI producer(HANDLE Thread){
    while(g_CONTINUE){
        WaitForSingleObject(g_HFULL_Semaphore,INFINITE);
        WaitForSingleObject(g_HMUTEX,INFINITE);//
        produce();
        Sleep(1000);
        ReleaseMutex(g_HMUTEX);
        ReleaseSemaphore(g_HEMPTY_Semaphore,1,NULL);
    }
    return 0;
} //生产的子线程函数
```

此函数为生产者的子线程。首先生产者如果要生产，则需要生产的数量先-1，这里对信号量 full 进行 P 操作，然后对子线程互斥信号量 mutex 上锁,调用函数 produce(), 对互斥信号量解锁，然后生产成功，对信号量 empty 进行 V 操作。

2. DWORD WINAPI consumer(HANDLE Thread)&&void consume()

```
void consume(){
    cout<<"***** Consume start *****"<<endl;
    cout<<"get product from the buffer....."<<endl;
    CONSUME_ID=g_BUFFER[out];
    out=(out+1)%BUFFER;
    for(int i=0;i<BUFFER;i++){
        cout<<i<<" : "<<g_BUFFER[i];
        if(i==in){
            cout<<"<---- produce";
        }
        if(i==out){
            cout<<"<---- consume";
        }
        cout<<endl;
    }
    cout<<"consume product:"<<CONSUME_ID<<endl;
    cout<<"***** Consume end *****"<<endl;
} //消费的主逻辑函数
```

此函数主要完成了在对应缓冲区进行消费。

```
DWORD WINAPI consumer(HANDLE Thread){
    while(g_CONTINUE){
        WaitForSingleObject(g_EMPTY_Semaphore,INFINITE);
        WaitForSingleObject(g_HMUTEX,INFINITE);
        consume();
        Sleep(1000);
        ReleaseMutex(g_HMUTEX);
        ReleaseSemaphore(g_HFULL_Semaphore,1,NULL);
    }
    return 0;
} //消费的子线程函数
```

此函数为消费者的子线程。首先消费者如果要生产，则需要消费的数量先-1，这里对信号量 empty 进行 P 操作，然后对子线程互斥信号量 mutex 上锁,调用函数 consume(), 对互斥信号量解锁，然后消费成功，对信号量 full 进行 V 操作。

实验 2.2 哲学家就餐问题

1. void eating(int id)

```
void eating(int id)
{
    //传进来为id的哲学家
    cout<<"%%%%%%%%%% Eat start %%%%%%%%%%"<<endl;
    cout<<"Philosopher start to eat....."<<endl;
    for(int i=1;i<=PERSON;i++){
        cout<<"Philosopher  "<<i;
        if(i==id){
            cout<<"<---- is eating";
        }else{
            cout<<"<---- is thinking";
        }
        cout<<endl;
    }
    //判断哪些哲学家在吃饭，哪些在思考
    int num = random(1,100); //吃饭吃一段随机的时间
    Sleep(num);
    cout<<"Philosopher "<<i<<" eat for "<<num<<" seconds"<<endl;
    cout<<"%%%%%%%%%% Eat end %%%%%%%%%%"<<endl;
}
```

此函数是哲学家编号为 id 的哲学家开始就餐，其他哲学家思考。就餐时间为随机数生成。

2. DWORD WINAPI Philosopher(LPVOID param)

```
DWORD WINAPI Philosopher(LPVOID param)
{
    nums++; //循环哲学家的 id
    int id = nums;
    int left_chopstick = id - 1; //左筷子的编号
    int right_chopstick = id % PERSON; //右筷子的编号
    int times = 0; //哲学家就餐的次数
    int ret1, ret2; //信号量的返回值
    while (true)
    {
        Sleep(100);
        if (times >= MEALS){
            break;
        }
        //如果哲学家已经吃了足够的饭，则直接退出
        if (id % 2 == 0){ //如果哲学家编号为偶数
            ret1 = WaitForSingleObject(chopstick_semaphore[left_chopstick], 0); //取走左边筷子
            if (ret1 == WAIT_OBJECT_0){ //如果能取走
                ret2 = WaitForSingleObject(chopstick_semaphore[right_chopstick], 0); //再取右边的
                //筷子
                if (ret2 == WAIT_OBJECT_0){ //如果能取走
                    WaitForSingleObject(mutex, INFINITE);
                }
            }
        }
        //取走筷子
        times++;
    }
}
```

```

        cout<<"Philosopher "<<id<<" gets two chopsticks and starts his "<<times+1<<"th
meal"<<endl;

        ReleaseMutex(mutex);
        times++;
        WaitForSingleObject(mutex, INFINITE);
        eating(id);
        ReleaseMutex(mutex);
        WaitForSingleObject(mutex, INFINITE);
        cout<<"Philosopher "<<id<<" has finished his/her meal"<<endl;
        ReleaseMutex(mutex);
        ReleaseSemaphore(chopstick_semaphore[rignt_chopstick], 1, NULL); //放回左边的
筷子

    } //开始就餐
    ReleaseSemaphore(chopstick_semaphore[left_chopstick], 1, NULL); //放回右边的筷子
}

else //如果哲学家编号为奇数，下面代码含义同上
{
    ret1 = WaitForSingleObject(chopstick_semaphore[rignt_chopstick], 0);
    if (ret1 == WAIT_OBJECT_0)
    {
        ret2 = WaitForSingleObject(chopstick_semaphore[left_chopstick], 0);
        if (ret2 == WAIT_OBJECT_0)
        {
            WaitForSingleObject(mutex, INFINITE);
            cout<<"Philosopher "<<id<<" gets two chopsticks and starts his "<<times+1<<"th
meal"<<endl;

            ReleaseMutex(mutex);
            times++;
            WaitForSingleObject(mutex, INFINITE);
            eating(id);
            ReleaseMutex(mutex);
            WaitForSingleObject(mutex, INFINITE);
            cout<<"Philosopher "<<id<<" has finished his/her meal"<<endl;
            ReleaseMutex(mutex);
            ReleaseSemaphore(chopstick_semaphore[left_chopstick], 1, NULL);
        }
        ReleaseSemaphore(chopstick_semaphore[rignt_chopstick], 1, NULL);
    }
}

WaitForSingleObject(mutex, INFINITE);
ReleaseMutex(mutex);
}

cout<<"***** Philosopher "<<id<<" is full and leave now *****"<<endl; //哲学家吃饱了

```

```

return 0;
}

```

此函数为哲学家就餐的子线程。核心思路是，编号为偶数的哲学家先拿左边筷子，如果拿得到左边筷子，再拿右边筷子，否则放下左边筷子；编号为奇数的哲学家先拿右边筷子，如果拿得到右边筷子，再拿左边筷子，否则放下右边筷子。若哲学家拿到了左右两个筷子，则可以开始进餐，进餐设置时间为随机时间。同时还设置了哲学家需要就餐的次数，达到了就餐的次数才能不再进餐，否则需要继续等待筷子可用，继续进餐。

实验 2.3_4 读者优先和写者优先的读者写者问题

1. void solution(int sign)

```

void solution(int sign) {
    int i = 0;
    HANDLE Thread[Thread_NUM];
    PCB pro[Thread_NUM] = { {"r1",0,15},
                              {"r2",1, 15},
                              {"w1",3,3},
                              {"r3",4, 2},
                              {"w2",5,6},
                              {"w3",6,10},
                              {"r4",7,8},
                              {"r5",9,2},
                              {"w4",10,18},
                              {"w5",12,2}
    };//初始化若干读写进程
    InitializeCriticalSection(&FILES); //初始化临界区变量
    readercount = CreateMutex(NULL, FALSE, NULL); //创建读者计数器信号量
    writercount = CreateMutex(NULL, FALSE, NULL); //创建写者计数器信号量
    rw = CreateMutex(NULL, FALSE, NULL); //创建互斥信号量，用于读者和写者互斥访问
    wait = CreateMutex(NULL, FALSE, NULL); //创建互斥信号量，用于确保写者优先
    cout << "Process application order:";
    for (i = 0; i < Thread_NUM; i++) {
        cout << pro[i].name << " ";
    }
    cout << endl;
    if (sign == 1) {
        cout << "Start reader priority process operation:" << endl;
        for (i = 0; i < Thread_NUM; i++) { //根据线程名称创建不同的线程
            if (pro[i].name[0] == 'r') //名称的首字母是'r'则创建读者线程
                Thread[i] = CreateThread(NULL, 0, ReaderFisrt_reader, &pro[i].name, 0, NULL);
            else {
                //名称的首字母是'w'则创建写者线程
                Thread[i] = CreateThread(NULL, 0, ReaderFisrt_writer, &pro[i].name, 0, NULL);
            }
        }
        WaitForMultipleObjects(Thread_NUM, Thread, TRUE, -1); //等待所有线程结束
    }
}

```



```

    }
} else if (sign == 2) {
    cout << "Start a writer-first process operation:" << endl;
    for (i = 0; i < Thread_NUM; i++) { //根据线程名称创建不同的线程
        if (pro[i].name[0] == 'r') //名称的首字母是'r'则创建读者线程
            Thread[i] = CreateThread(NULL, 0, WriterFisrt_reader, &pro[i].name, 0, NULL);
        else {
            //名称的首字母是'w'则创建写者线程
            Thread[i] = CreateThread(NULL, 0, WriterFisrt_writer, &pro[i].name, 0, NULL);
        }
        WaitForMultipleObjects(Thread_NUM, Thread, TRUE, -1); //等待所有线程结束
    }
}
CloseHandle(Thread); //关闭线程句柄
}

```

此函数根据入口参数 sign 的值来决定是进行读者优先模式还是写者优先模式。函数先初始化若干读写进程，这些读写进程按照一定的顺序随机排列。然后创建各种相关的锁和信号量。如果 sign 的值是 1，则进入模式读者优先，则优先创建读者进程；如果 sign 的值是 2，则进入模式写者优先，则优先创建写者进程。

六、实验结果

实验 2.1 生产者消费者问题

主函数：

```

int main(){
    solution();
    system("pause");
}

```

实验结果：

由于本实验线程未设置循环结束条件，因而消费者和生产者会一直运行。下面是实验结果，截取部分：

实验数据较多，下面仅仅截取部分实验成果，源代码在文中附录，可以运行观察更详细的实验结果。

选择 c:\VS\code\c and c++\os\work2.1.exe

```
***** Produce start *****
create product: 1
put new product in buffer.....
0 :1<---- consume
1 :0<---- produce
2 :0
3 :0
4 :0
***** Produce end *****
***** Produce start *****
create product: 2
put new product in buffer.....
0 :1<---- consume
1 :2
2 :0<---- produce
3 :0
4 :0
***** Produce end *****
***** Produce start *****
create product: 3
put new product in buffer.....
0 :1<---- consume
1 :2
2 :3
3 :0<---- produce
4 :0
***** Produce end *****
***** Consume start *****
get product from the buffer.....
0 :1
1 :2<---- consume
2 :3
3 :0<---- produce
4 :0
consume product:1
***** Consume end *****
***** Produce start *****
create product: 4
put new product in buffer.....
0 :1
1 :2<---- consume
2 :3
3 :4
4 :0<---- produce
***** Produce end *****
```

选择 c:\VS\code\c and c++\os\work2.1.exe

```
***** Consume start *****
get product from the buffer.....
0 :1
1 :2
2 :3<---- consume
3 :4
4 :0<---- produce
consume product:2
***** Consume end *****
***** Produce start *****
create product: 5
put new product in buffer.....
0 :1<---- produce
1 :2
2 :3<---- consume
3 :4
4 :5
***** Produce end *****
***** Produce start *****
create product: 6
put new product in buffer.....
0 :6
1 :2<---- produce
2 :3<---- consume
3 :4
4 :5
***** Produce end *****
***** Consume start *****
get product from the buffer.....
0 :6
1 :2<---- produce
2 :3
3 :4<---- consume
4 :5
consume product:3
***** Consume end *****
***** Consume start *****
get product from the buffer.....
0 :6
1 :2<---- produce
2 :3
3 :4
4 :5<---- consume
consume product:4
***** Consume end *****
```

选择 c:\VS\code\c and c++\os\work2.1.exe

```
***** Produce start *****
create product: 7
put new product in buffer.....
0 :6
1 :7
2 :3<---- produce
3 :4
4 :5<---- consume
***** Produce end *****
***** Consume start *****
get product from the buffer.....
0 :6<---- consume
1 :7
2 :3<---- produce
3 :4
4 :5
consume product:5
***** Consume end *****
***** Produce start *****
create product: 8
put new product in buffer.....
0 :6<---- consume
1 :7
2 :8
3 :4<---- produce
4 :5
***** Produce end *****
***** Consume start *****
get product from the buffer.....
0 :6
1 :7<---- consume
2 :8
3 :4<---- produce
4 :5
consume product:6
***** Consume end *****
***** Produce start *****
create product: 9
put new product in buffer.....
0 :6
1 :7<---- consume
2 :8
3 :9
4 :5<---- produce
***** Produce end *****
```

选择 c:\VS\code\c and c++\os\work2.1.exe

```
***** Produce start *****
create product: 10
put new product in buffer.....
0 :6<---- produce
1 :7<---- consume
2 :8
3 :9
4 :10
***** Produce end *****
***** Consume start *****
get product from the buffer.....
0 :6<---- produce
1 :7
2 :8<---- consume
3 :9
4 :10
consume product:7
***** Consume end *****
***** Consume start *****
get product from the buffer.....
0 :6<---- produce
1 :7
2 :8
3 :9<---- consume
4 :10
consume product:8
***** Consume end *****
***** Produce start *****
create product: 11
put new product in buffer.....
0 :11
1 :7<---- produce
2 :8
3 :9<---- consume
4 :10
***** Produce end *****
***** Produce start *****
create product: 12
put new product in buffer.....
0 :11
1 :12
2 :8<---- produce
3 :9<---- consume
4 :10
***** Produce end *****
```



```
选择 c:\VS\code\c and c++\os\work2.1.exe
***** Consume start *****
get product from the buffer.....
0 :11
1 :12
2 :8<----- produce
3 :9
4 :10<----- consume
consume product:9
***** Consume end *****
***** Consume start *****
get product from the buffer.....
0 :11<----- consume
1 :12
2 :8<----- produce
3 :9
4 :10
consume product:10
***** Consume end *****
***** Produce start *****
create product: 13
put new product in buffer.....
0 :11<----- consume
1 :12
2 :13
3 :9<----- produce
4 :10
***** Produce end *****
***** Produce start *****
create product: 14
put new product in buffer.....
0 :11<----- consume
1 :12
2 :13
3 :14
4 :10<----- produce
***** Produce end *****
***** Consume start *****
get product from the buffer.....
0 :11
1 :12<----- consume
2 :13
3 :14
4 :10<----- produce
consume product:11
***** Consume end *****
```

实验 2.2 哲学家就餐问题

主函数:

```
int main()
{
    solution();
    system("pause");
    return 0;
}
```

首先规定哲学家需要就餐 3 次：

```
const unsigned int MEALS=3;//定义每个哲学家总共需要吃三顿饭吃饱
```

实验结果：

选择 c:\VS\code\c and c++\os\work2.4_new.exe

```
Philosopher 5 gets two chopsticks and starts his 1th meal
Philosopher 2 gets two chopsticks and starts his 1th meal
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is thinking
Philosopher 3<---- is thinking
Philosopher 4<---- is thinking
Philosopher 5<---- is eating
Philosopher 5 eat for 42 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is eating
Philosopher 3<---- is thinking
Philosopher 4<---- is thinking
Philosopher 5<---- is thinking
Philosopher 2 eat for 42 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
Philosopher 5 has finished his/her meal
Philosopher 2 has finished his/her meal
Philosopher 2 gets two chopsticks and starts his 2th meal
Philosopher 5 gets two chopsticks and starts his 2th meal
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is eating
Philosopher 3<---- is thinking
Philosopher 4<---- is thinking
Philosopher 5<---- is thinking
Philosopher 2 eat for 68 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is thinking
Philosopher 3<---- is thinking
Philosopher 4<---- is thinking
Philosopher 5<---- is eating
Philosopher 5 eat for 68 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
Philosopher 2 has finished his/her meal
Philosopher 5 has finished his/her meal
Philosopher 1 gets two chopsticks and starts his 1th meal
Philosopher 3 gets two chopsticks and starts his 1th meal
```

对此截图的含义做出一些说明：首先是哲学家 2 和 5 先拿到筷子并进餐，二者均吃了 42s,然后哲学家 5 和 2 又拿到了筷子开始进餐第二次，进餐时间为 68s。然后哲学家 1 和 3 拿到了筷子。


```

%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is eating
Philosopher 2<---- is thinking
Philosopher 3<---- is thinking
Philosopher 4<---- is thinking
Philosopher 5<---- is thinking
Philosopher 1 eat for 42 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is thinking
Philosopher 3<---- is eating
Philosopher 4<---- is thinking
Philosopher 5<---- is thinking
Philosopher 3 eat for 42 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
Philosopher 1 has finished his/her meal
Philosopher 3 has finished his/her meal
Philosopher 5 gets two chopsticks and starts his 3th meal
Philosopher 3 gets two chopsticks and starts his 2th meal
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is thinking
Philosopher 3<---- is thinking
Philosopher 4<---- is thinking
Philosopher 5<---- is eating
Philosopher 5 eat for 35 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is thinking
Philosopher 3<---- is eating
Philosopher 4<---- is thinking
Philosopher 5<---- is thinking
Philosopher 3 eat for 68 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
Philosopher 5 has finished his/her meal
Philosopher 3 has finished his/her meal
***** Philosopher 5 is full and leave now *****
Philosopher 4 gets two chopsticks and starts his 1th meal
Philosopher 1 gets two chopsticks and starts his 2th meal

```

哲学家 1 和 3 开始进餐，均就餐了 42s。然后哲学家 5 和 3 拿到了筷子，哲学家 5 就餐了 35s，哲学家 3 就餐了 68s。然后哲学家 5 已经吃饱饭，并离开。最后哲学家 4 和 1 拿到了筷子。

```

%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is thinking
Philosopher 3<---- is thinking
Philosopher 4<---- is eating
Philosopher 5<---- is thinking
Philosopher 4 eat for 42 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is eating
Philosopher 2<---- is thinking
Philosopher 3<---- is thinking
Philosopher 4<---- is thinking
Philosopher 5<---- is thinking
Philosopher 1 eat for 68 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
Philosopher 4 has finished his/her meal
Philosopher 1 has finished his/her meal
Philosopher 2 gets two chopsticks and starts his 3th meal
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is eating
Philosopher 3<---- is thinking
Philosopher 4<---- is thinking
Philosopher 5<---- is thinking
Philosopher 2 eat for 35 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
Philosopher 2 has finished his/her meal
Philosopher 1 gets two chopsticks and starts his 3th meal
Philosopher 4 gets two chopsticks and starts his 2th meal
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is eating
Philosopher 2<---- is thinking
Philosopher 3<---- is thinking
Philosopher 4<---- is thinking
Philosopher 5<---- is thinking
Philosopher 1 eat for 35 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5

```

哲学家 4 和 1 开始就餐，哲学家 4 就餐 42s，哲学家 1 就餐 68s。然后 2 拿到了筷子就餐 35s。然后哲学家 1 和 4 拿到了筷子，哲学家 1 就餐 35s。


```

%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is thinking
Philosopher 3<---- is thinking
Philosopher 4<---- is eating
Philosopher 5<---- is thinking
***** Philosopher 2 is full and leave now *****
Philosopher 4 eat for 68 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
Philosopher 1 has finished his/her meal
Philosopher 4 has finished his/her meal
***** Philosopher 1 is full and leave now *****
Philosopher 4 gets two chopsticks and starts his 3th meal
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is thinking
Philosopher 3<---- is thinking
Philosopher 4<---- is eating
Philosopher 5<---- is thinking
Philosopher 4 eat for 35 seconds
%%%%%%%%%% Eat end %%%%%%%%%%5
Philosopher 4 has finished his/her meal
Philosopher 3 gets two chopsticks and starts his 3th meal
%%%%%%%%%% Eat start %%%%%%%%%%5
Philosopher start to eat.....
Philosopher 1<---- is thinking
Philosopher 2<---- is thinking
Philosopher 3<---- is eating
Philosopher 4<---- is thinking
Philosopher 5<---- is thinking
Philosopher 3 eat for 35 seconds
***** Philosopher 4 is full and leave now *****
%%%%%%%%%% Eat end %%%%%%%%%%5
Philosopher 3 has finished his/her meal
***** Philosopher 3 is full and leave now *****
请按任意键继续. . . █

```

哲学家 4 就餐 68s。哲学家 2、1 吃饱了离开。哲学家 4 拿到了筷子，哲学家 4 就餐 35s。
哲学家 4 吃饱了离开。哲学家 3 拿到筷子就餐，就餐 35s，然后吃饱了离开。

我们把结果汇总成一个表格如下：

哲学家编号	第几次就餐	就餐时间/s	哲学家状态
5	1	42	进餐/思考
2	1	42	进餐/思考
2	2	68	进餐/思考
5	2	68	进餐/思考
1	1	42	进餐/思考
3	1	42	进餐/思考
5	3	35	就餐完毕，离开
3	2	68	进餐/思考
4	1	42	进餐/思考
1	2	68	进餐/思考
2	3	35	就餐完毕，离开
1	3	35	就餐完毕，离开
4	2	68	进餐/思考
4	3	35	就餐完毕，离开
3	3	35	就餐完毕，离开

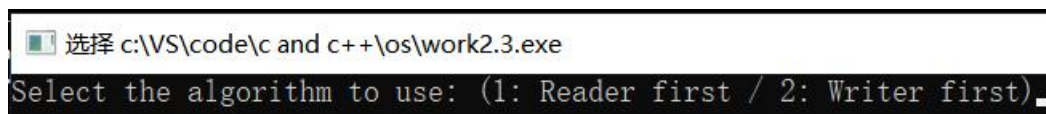
实验 2.3_4 读者优先和写者优先的读者写者问题

主函数：

```
int main() {  
    cout << "Select the algorithm to use: (1: Reader first / 2: Writer first)";  
    int sign;  
    cin >> sign;  
    solution(sign);  
    cout<<"The end....."<<endl;  
    system("pause");  
}
```

实验结果：

首先进入程序起始位置。



若输入 1，进入读者优先模式：

```
c:\VS\code\c and c++\os\work2.3.exe
Select the algorithm to use: (1: Reader first / 2: Writer first)1
Process application order:r1 r2 w1 r3 w2 w3 r4 r5 w4 w5
Start reader priority process operation:
r1 is reading ...
r2 is reading ...
r3 is reading ...
r4 is reading ...
r5 is reading ...
w1 is writing ...
w2 is writing ...
w3 is writing ...
w4 is writing ...
w5 is writing ...
The end.....
请按任意键继续. . .
```

若输入 2，进入写者优先模式：

```
c:\VS\code\c and c++\os\work2.3.exe
Select the algorithm to use: (1: Reader first / 2: Writer first)2
Process application order:r1 r2 w1 r3 w2 w3 r4 r5 w4 w5
Start a writer-first process operation:
r1 is writing ...
r2 is writing ...
w1 is writing ...
w2 is writing ...
w3 is writing ...
w4 is writing ...
w5 is writing ...
r3 is writing ...
r4 is writing ...
r5 is writing ...
The end.....
请按任意键继续. . .
```

七、技术难点及解决方案

技术难点：

熟悉 `window.h` 库的线程函数，并且熟练的去运用。

解决方案：

查询 MSDN，熟悉理解 `window` 库函数的运用。例如：

WaitForSingleObject

The **WaitForSingleObject** function returns when one of the following occurs:

- The specified object is in the signaled state.
- The time-out interval elapses.

```
DWORD WaitForSingleObject(  
    HANDLE hHandle,           // handle to object to wait for  
    DWORD dwMilliseconds     // time-out interval in milliseconds  
);
```

Parameters

hHandle

Handle to the object. For a list of the object types whose handles can be specified, see the following Remarks section.

Windows NT: The handle must have SYNCHRONIZE access. For more information, see [Standard Access Rights](#).

dwMilliseconds

Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is nonsignaled. If *dwMilliseconds* is zero, the function tests the object's state and returns immediately. If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.

CreateMutex

The **CreateMutex** function creates a named or unnamed mutex object.

```
HANDLE CreateMutex(  
    LPSECURITY_ATTRIBUTES lpMutexAttributes, // pointer to security attributes  
    BOOL bInitialOwner,    // flag for initial ownership  
    LPCTSTR lpName         // pointer to mutex-object name  
);
```

Parameters

lpMutexAttributes

Pointer to a [SECURITY_ATTRIBUTES](#) structure that determines whether the returned handle can be inherited by child processes. If *lpMutexAttributes* is NULL, the handle cannot be inherited.

Windows NT: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new mutex. If *lpMutexAttributes* is NULL, the mutex gets a default security descriptor.

bInitialOwner

Specifies the initial owner of the mutex object. If this value is TRUE and the caller created the mutex, the calling thread obtains ownership of the mutex object. Otherwise, the calling thread does not obtain ownership of the mutex. To determine if the caller created the mutex, see the Return Values section.

lpName

Pointer to a null-terminated string specifying the name of the mutex object. The

CreateSemaphore

The **CreateSemaphore** function creates a named or unnamed semaphore object.

```
HANDLE CreateSemaphore(  
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // pointer to security attributes  
    LONG lInitialCount,    // initial count  
    LONG lMaximumCount,    // maximum count  
    LPCTSTR lpName         // pointer to semaphore-object name  
);
```

Parameters

lpSemaphoreAttributes

Pointer to a [SECURITY_ATTRIBUTES](#) structure that determines whether the returned handle can be inherited by child processes. If *lpSemaphoreAttributes* is NULL, the handle cannot be inherited.

Windows NT: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new semaphore. If *lpSemaphoreAttributes* is NULL, the semaphore gets a default security descriptor.

lInitialCount

Specifies an initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to *lMaximumCount*. The state of a semaphore is signaled when its count is greater than zero and nonsignaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling the [ReleaseSemaphore](#) function.

此处不一一列举了。

八、疑难解惑即经验教训

生产者和消费者问题在现实系统中是很普遍的。例如在一个多媒体系统中，生产者编码视频帧，而消费者消费（解码）视频帧，缓冲区的目的就是减少视频流的抖动。又如在图形用户接口设计中，生产者检测到鼠标和键盘事件，并将其插入到缓冲区中。消费者以某种基于优先级的方式从缓冲区中取出这些事件并显示在屏幕上。

哲学家就餐问题是在计算机科学中的一个经典问题，用来演示在并行计算中多线程同步(Synchronization)时产生的问题。在 1971 年，著名的计算机科学家艾兹格·迪科斯彻提出了一个同步问题，即假设有五台计算机都试图访问五份共享的磁带驱动器。稍后，这个问题被托尼·霍尔重新表述为哲学家就餐问题。这个问题可以用来解释死锁和资源耗尽。

哲学家就餐问题可以这样表述，假设有五位哲学家围坐在一张圆形餐桌旁，做以下两件事情之一：吃饭，或者思考。吃东西的时候，他们就停止思考，思考的时候也停止吃东西。餐桌中间有一大碗意大利面，每两个哲学家之间有一只餐叉。因为用一只餐叉很难吃到意大利面，所以假设哲学家必须用两只餐叉吃东西。他们只能使用自己左右手边的那两只餐叉。哲学家就餐问题有时也用米饭和筷子而不是意大利面和餐叉来描述，因为很明显，吃米饭必须用两根筷子。

哲学家从来不交谈，这就很危险，可能产生死锁，每个哲学家都拿着左手的餐叉，永远都在等右边的餐叉（或者相反）。即使没有死锁，也有可能发生资源耗尽。例如，假设规定当哲学家等待另一只餐叉超过五分钟后就放下自己手里的那一只餐叉，并且再等五分钟后进行下一次尝试。这个策略消除了死锁（系统总会进入到下一个状态），但仍然有可能发生“活锁”。如果五位哲学家在完全相同的时刻进入餐厅，并同时拿起左边的餐叉，那么这些哲学家就会等待五分钟，同时放下手中的餐叉，再等五分钟，又同时拿起这些餐叉。

在实际的计算机问题中，缺乏餐叉可以类比为缺乏共享资源。一种常用的计算机技术是资源加锁，用来保证在某个时刻，资源只能被一个程序或一段代码访问。当一个程序想要使用的资源已经被另一个程序锁定，它就等待资源解锁。当多个程序涉及到加锁的资源时，在某些情况下就有可能发生死锁。例如，某个程序需要访问两个文件，当两个这样的程序各锁了一个文件，那它们都在等待对方解锁另一个文件，而这永远不会发生。

这些例子在我们身边都是十分常见的，我们要从中学习基本原理的同时，懂得如何将知识运用在生活中去！

九、实验总结（结论与心得体会）

通过本次实验，我对进餐的同步与互斥有了更深的理解，对于生产者消费者模型和哲学家就餐问题有了更深入的认识，受益匪浅。

十、附录

实验 2.1 源代码

```
#include<iostream>
#include<string>
#include<cstdlib>
#include<ctime>
#include<windows.h>
#include<stdlib.h>
#include<stdio.h>
using namespace std;

const unsigned int BUFFER=5;//缓冲区长度
```



```

unsigned int PRODUCT_ID=0;//生产者 ID
unsigned int CONSUME_ID=0;//消费者 ID
unsigned int in;//产品进缓冲区时的缓冲区产品个数
unsigned int out;//产品出缓冲区时的缓冲区产品个数
int g_BUFFER[BUFFER];//缓冲区为循环队列
bool g_CONTINUE=true;//控制程序结束
HANDLE g_HMUTEX;//线程间的互斥对象
HANDLE g_HFULL_Semaphore;//缓冲区初始化为空，满则生产者等待
HANDLE g_HEMPTY_Semaphore;//空闲缓冲区，空则消费者等待
const unsigned int PRODUCE_COUNT=2;//生产者数量
const unsigned int CONSUMER_COUNT=1;//消费者数量
const unsigned int THREADS_COUNT=PRODUCE_COUNT+CONSUMER_COUNT;//线程的数据总量
void produce(){
    cout<<"***** Procude start *****"<<endl;
    cout<<"create product: "<<++PRODUCT_ID<<endl;
    g_BUFFER[in]=PRODUCT_ID;
    in=(in+1)%BUFFER;
    cout<<"put new product in buffer....."<<endl;
    for(int i=0;i<BUFFER;i++){
        cout<<i<<" : "<<g_BUFFER[i];
        if(i==in){
            cout<<"<---- produce";
        }
        if(i==out){
            cout<<"<---- consume";
        }
        cout<<endl;
    }
    cout<<"***** Procude end *****"<<endl;
}
//生产的主逻辑函数

```

```

void consume(){
    cout<<"***** Consume start *****"<<endl;
    cout<<"get product from the buffer....."<<endl;
    CONSUME_ID=g_BUFFER[out];
    out=(out+1)%BUFFER;
    for(int i=0;i<BUFFER;i++){
        cout<<i<<" : "<<g_BUFFER[i];
        if(i==in){
            cout<<"<---- produce";
        }
        if(i==out){
            cout<<"<---- consume";
        }
    }
}

```

```

        cout<<endl;
    }
    cout<<"consume product:"<<CONSUME_ID<<endl;
    cout<<"***** Consume end *****"<<endl;
} //消费的主逻辑函数

DWORD WINAPI producer(HANDLE Thread){
    while(g_CONTINUE){
        WaitForSingleObject(g_HFULL_Semaphore,INFINITE);
        WaitForSingleObject(g_HMUTEX,INFINITE);
        produce();
        Sleep(1000);
        ReleaseMutex(g_HMUTEX);
        ReleaseSemaphore(g_HEMPTY_Semaphore,1,NULL);
    }
    return 0;
} //生产的子线程函数

DWORD WINAPI consumer(HANDLE Thread){
    while(g_CONTINUE){
        WaitForSingleObject(g_HEMPTY_Semaphore,INFINITE);
        WaitForSingleObject(g_HMUTEX,INFINITE);
        consume();
        Sleep(1000);
        ReleaseMutex(g_HMUTEX);
        ReleaseSemaphore(g_HFULL_Semaphore,1,NULL);
    }
    return 0;
} //消费的子线程函数

int solution(){
    g_HMUTEX=CreateMutex(NULL,false,NULL); //线程间的互斥对象,互斥锁的创建
    g_HFULL_Semaphore=CreateSemaphore(NULL,BUFFER-1,BUFFER-1,NULL); //生产者信号量
    g_HEMPTY_Semaphore=CreateSemaphore(NULL,0,BUFFER-1,NULL); //消费者信号量
    HANDLE h_Threads[THREADS_COUNT]; //创建若干线程对象
    DWORD PRODUCER_ID[PRODUCE_COUNT]; //生产者线程的标识
    DWORD CONSUMER_ID[CONSUMER_COUNT]; //消费者线程的标识
    for(int i=0;i<PRODUCE_COUNT;i++){
        if((h_Threads[i]=CreateThread(NULL,0,producer,NULL,0,&PRODUCER_ID[i]))==NULL){
            cout<<"Thread: Thread_Func_producer- "<<i<<" failed to create!"<<endl;
            exit(0);
        }
    }
    for(int i=0;i<CONSUMER_COUNT;i++){
        if((h_Threads[PRODUCE_COUNT+i]=CreateThread(NULL,0,consumer,NULL,0,&CONSUMER_ID[i]))==
NULL){
            cout<<"Thread: Thread_Func_consumer- "<<i<<" failed to create!"<<endl;

```

```

        exit(0);
    }
}

while(g_CONTINUE){
    if(getchar()){
        g_CONTINUE=false;
    }
}

CloseHandle(h_Threads);
CloseHandle(g_HFULL_Semaphore);
CloseHandle(g_HEMPTY_Semaphore);
CloseHandle(g_HMUTEX); //关闭信号量，锁和句柄
return 0;
}

int main(){
    solution();
    system("pause");
}

```

实验 2.2 源代码

```

#include <Windows.h>
#include <iostream>
#include <cstdio>
#include <stdlib.h>
#include <time.h>
#define random(a,b) (rand()%(b-a+1)+a)
using namespace std;
const unsigned int PERSON=5; //定义哲学家人数
const unsigned int MEALS=3; //定义每个哲学家总共需要吃三顿饭吃饱
HANDLE chopstick_semaphore[PERSON]; //定义筷子的互斥信号量
HANDLE Thread[PERSON]; //每个哲学家定义一个子线程
HANDLE mutex; //子线程的互斥信号量
unsigned int nums = 0; //访问的人数序号
void eating(int id)
{
    cout<<"%%%%%%%% Eat start %%%%%%%%%5"<<endl;
    cout<<"Philosopher start to eat....."<<endl;
    for(int i=1;i<=PERSON;i++){
        cout<<"Philosopher  "<<i;
        if(i==id){
            cout<<"<---- is eating";
        }else{

```

```

        cout<<"<---- is thinking";
    }
    cout<<endl;
}

int num = random(1,100);
Sleep(num);
cout<<"Philosopher "<<id<<" eat for "<<num<<" seconds"<<endl;
cout<<"%%%%%%%%%% Eat end %%%%%%%%%%%5"<<endl;
}

DWORD WINAPI Philosopher(LPVOID param)
{
    nums++;
    int id = nums;
    int left_chopstick = id - 1;
    int right_chopstick = id % PERSON;
    int times = 0;
    int ret1, ret2;
    while (true)
    {
        Sleep(100);
        if (times >= MEALS){
            break;
        }
        if (id % 2 == 0){
            ret1 = WaitForSingleObject(chopstick_semaphore[left_chopstick], 0);
            if (ret1 == WAIT_OBJECT_0){
                ret2 = WaitForSingleObject(chopstick_semaphore[right_chopstick], 0);
                if (ret2 == WAIT_OBJECT_0){
                    WaitForSingleObject(mutex, INFINITE);
                    cout<<"Philosopher "<<id<<" gets two chopsticks and starts his "<<times+1<<"th
meal"<<endl;

                    ReleaseMutex(mutex);
                    times++;
                    WaitForSingleObject(mutex, INFINITE);
                    eating(id);
                    ReleaseMutex(mutex);
                    WaitForSingleObject(mutex, INFINITE);
                    cout<<"Philosopher "<<id<<" has finished his/her meal"<<endl;
                    ReleaseMutex(mutex);
                    ReleaseSemaphore(chopstick_semaphore[right_chopstick], 1, NULL);
                }
                ReleaseSemaphore(chopstick_semaphore[left_chopstick], 1, NULL);
            }
        }
    }
}

```

```

        else
        {
            ret1 = WaitForSingleObject(chopstick_semaphore[right_chopstick], 0);
            if (ret1 == WAIT_OBJECT_0)
            {
                ret2 = WaitForSingleObject(chopstick_semaphore[left_chopstick], 0);
                if (ret2 == WAIT_OBJECT_0)
                {
                    WaitForSingleObject(mutex, INFINITE);
                    cout<<"Philosopher "<<id<<" gets two chopsticks and starts his "<<times+1<<"th
meal"<<endl;

                    ReleaseMutex(mutex);
                    times++;
                    WaitForSingleObject(mutex, INFINITE);
                    eating(id);
                    ReleaseMutex(mutex);
                    WaitForSingleObject(mutex, INFINITE);
                    cout<<"Philosopher "<<id<<" has finished his/her meal"<<endl;
                    ReleaseMutex(mutex);
                    ReleaseSemaphore(chopstick_semaphore[left_chopstick], 1, NULL);
                }
                ReleaseSemaphore(chopstick_semaphore[right_chopstick], 1, NULL);
            }
        }

        WaitForSingleObject(mutex, INFINITE);
        ReleaseMutex(mutex);
    }

    cout<<"***** Philosopher "<<id<<" is full and leave now *****"<<endl;
    return 0;
}

int solution(){
    srand((unsigned)time(NULL));
    mutex = CreateMutex(NULL, false, NULL);
    for (int i = 0; i < PERSON; ++i){
        chopstick_semaphore[i] = CreateSemaphore(NULL, 1, 1, NULL);
    }
    for (int i = 0; i < PERSON; ++i){
        Thread[i] = CreateThread(NULL, 0, Philosopher, NULL, 0, NULL);
    }
    Sleep(5000);
    for (int i = 0; i < PERSON; ++i){
        CloseHandle(Thread[i]);
        CloseHandle(chopstick_semaphore[i]);
    }
}

```

```

        CloseHandle(mutex);
        Sleep(1500);
    }
int main()
{
    solution();
    system("pause");
    return 0;
}

```

实验 2.3_4 源代码

```

#include <Windows.h>
#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <string.h>
using namespace std;

int CountReader = 0; //用于记录当前的读者数量
int CountWriter = 0; //用于记录当前的写者数量
HANDLE readercount; //读者计数信号量
HANDLE writercount; //写者计数信号量
HANDLE rw; //用于保证读者和写者互斥地访问文件
HANDLE wait; //用于保证写优先（读者等待）
CRITICAL_SECTION FILES; //用于保护文件的临界区变量
unsigned const int Thread_NUM=10; //自定义的进程总数量
unsigned const int SLEEP_TIME=200; //模拟的运行时间
typedef struct PCB {
    string name; //进程名
    int requireTime; //请求时间
    int runningTime; //运行时间
}PCB;
//读者优先
//代码起始点 A.....
//读者进程
DWORD WINAPI ReaderFisrt_reader(LPVOID lpParameter) {
    PCB* p = (PCB*)lpParameter;
    string name = p->name;
    Sleep((p->requireTime) * SLEEP_TIME); //模拟等待时间
}

```

```

        WaitForSingleObject(readercount, INFINITE); //P 操作, 互斥信号量申请 (上锁), 申请进入读者计数器临界区

        CountReader++; // 读者计数器+1

        if (CountReader == 1) { //当第一个读进程读取时
            EnterCriticalSection(&FILES); //申请进入关于文件的临界区
            WaitForSingleObject(rw, INFINITE); //P 操作, 读进程读取文件, 拒绝写进程进入
        }

        ReleaseMutex(readercount); //V 操作, 离开读者计数器临界区, 互斥信号量释放 (开锁)
        cout << name.c_str() << " is reading ..." << endl;
        Sleep((p->runningTime) * SLEEP_TIME); //模拟运行时间

        WaitForSingleObject(readercount, INFINITE); //P 操作, 互斥信号量申请 (上锁), 申请进入读者计数器临界区

        CountReader--; // 读者计数器-1

        if (CountReader == 0) { //当最后一个读进程读取完成时
            LeaveCriticalSection(&FILES); //离开关于文件的临界区
            ReleaseMutex(rw); //V 操作, 允许写进程进入, 互斥信号量释放 (开锁)
        }

        ReleaseMutex(readercount); //V 操作, 离开读者计数器临界区, 互斥信号量释放 (开锁)
        return 0;
    }
}

//写者进程
DWORD WINAPI ReaderFisrt_writer(LPVOID lpParameter) {
    PCB* p = (PCB*)lpParameter;
    string name = p->name;
    Sleep((p->requireTime) * SLEEP_TIME); //模拟等待时间
    EnterCriticalSection(&FILES); //申请进入关于文件的临界区
    WaitForSingleObject(rw, INFINITE);
    cout << name.c_str() << " is writing ..." << endl;
    LeaveCriticalSection(&FILES); //离开关于文件的临界区
    ReleaseMutex(rw); //V 操作, 允许写进程进入, 互斥信号量释放 (开锁)
    return 0;
}

//代码中止点 A.....
//写者优先
//代码起始点 B.....
//读者进程
DWORD WINAPI WriterFisrt_reader(LPVOID lpParameter) {
    PCB* p = (PCB*)lpParameter;
    string name = p->name;
    Sleep((p->requireTime) * SLEEP_TIME); //模拟等待时间
    WaitForSingleObject(rw, INFINITE); //P 操作, 读写互斥访问
    WaitForSingleObject(wait, INFINITE); //P 操作, 在没有写进程请求时申请进入临界区
    WaitForSingleObject(readercount, INFINITE); //P 操作, 进入读者计数器临界区
    CountReader++; // 读者计数器+1

```



```

    if (CountReader == 1) { //当第一个读进程读取时
        EnterCriticalSection(&FILES); //申请进入关于文件的临界区
    }
    ReleaseMutex(readercount); //V 操作，离开读者计数器临界区
    ReleaseMutex(wait); //V 操作
    ReleaseMutex(rw); //V 操作
    cout << name.c_str() << " is writing ..." << endl;
    Sleep((p->runningTime) * SLEEP_TIME); //模拟运行时间
    WaitForSingleObject(readercount, INFINITE); //P 操作，进入读者计数器临界区
    CountReader--; // 读者计数器-1
    if (CountReader == 0) { //当最后一个读进程读取完成时
        LeaveCriticalSection(&FILES); //离开关于文件的临界区
    }
    ReleaseMutex(readercount); //V 操作，离开读者计数器临界区
    return 0;
}

//写者进程
DWORD WINAPI WriterFisrt_writer(LPVOID lpParameter) {
    PCB* p = (PCB*)lpParameter;
    string name = p->name;
    Sleep((p->requireTime) * SLEEP_TIME); //模拟等待时间
    WaitForSingleObject(writercount, INFINITE); //P 操作，进入写者计数器临界区
    if (CountWriter == 0) { //如果为第一个写者
        WaitForSingleObject(wait, INFINITE); //P 操作，开始执行写优先操作
    }
    CountWriter++;
    ReleaseMutex(writercount); //V 操作，离开写者计数器临界区
    EnterCriticalSection(&FILES); //P 操作，申请进入关于文件的临界区
    cout << name.c_str() << " is writing ..." << endl;
    Sleep((p->runningTime) * SLEEP_TIME); //模拟运行时间
    //V 操作，允许写进程进入，互斥信号量释放（开锁）
    LeaveCriticalSection(&FILES); //离开关于文件的临界区
    //P 操作，进入写者计数器临界区
    WaitForSingleObject(writercount, INFINITE);
    CountWriter--;
    if (CountWriter == 0) { //如果为最后一个写者
        ReleaseMutex(wait); //V 操作，写操作完成
    }
    ReleaseMutex(writercount); //V 操作，离开写者计数器临界区
    return 0;
}

//代码终止点 B.....
void solution(int sign) {
    int i = 0;

```

```

HANDLE Thread[Thread_NUM];

PCB pro[Thread_NUM] = { {"r1",0,15},
                        {"r2",1, 15},
                        {"w1",3,3},
                        {"r3",4, 2},
                        {"w2",5,6},
                        {"w3",6,10},
                        {"r4",7,8},
                        {"r5",9,2},
                        {"w4",10,18},
                        {"w5",12,2}

};//初始化若干读写进程

InitializeCriticalSection(&FILES); //初始化临界区变量

readercount = CreateMutex(NULL, FALSE, NULL); //创建读者计数器信号量
writercount = CreateMutex(NULL, FALSE, NULL); //创建写者计数器信号量
rw = CreateMutex(NULL, FALSE, NULL); //创建互斥信号量, 用于读者和写者互斥访问
wait = CreateMutex(NULL, FALSE, NULL); //创建互斥信号量, 用于确保写者优先

cout << "Process application order:";
for (i = 0; i < Thread_NUM; i++) {
    cout << pro[i].name << " ";
}
cout << endl;

if (sign == 1) {
    cout << "Start reader priority process operation:" << endl;
    for (i = 0; i < Thread_NUM; i++) { //根据线程名称创建不同的线程
        if (pro[i].name[0] == 'r') //名称的首字母是'r'则创建读者线程
            Thread[i] = CreateThread(NULL, 0, ReaderFisrt_reader, &pro[i].name, 0, NULL);
        else {
            //名称的首字母是'w'则创建写者线程
            Thread[i] = CreateThread(NULL, 0, ReaderFisrt_writer, &pro[i].name, 0, NULL);
        }

        WaitForMultipleObjects(Thread_NUM, Thread, TRUE, -1); //等待所有线程结束
    }
}
else if (sign == 2) {
    cout << "Start a writer-first process operation:" << endl;
    for (i = 0; i < Thread_NUM; i++) { //根据线程名称创建不同的线程
        if (pro[i].name[0] == 'r') //名称的首字母是'r'则创建读者线程
            Thread[i] = CreateThread(NULL, 0, WriterFisrt_reader, &pro[i].name, 0, NULL);
        else {
            //名称的首字母是'w'则创建写者线程
            Thread[i] = CreateThread(NULL, 0, WriterFisrt_writer, &pro[i].name, 0, NULL);
        }

        WaitForMultipleObjects(Thread_NUM, Thread, TRUE, -1); //等待所有线程结束
    }
}
}

```

```

        //CloseHandle(Thread); //关闭线程句柄
    }
int main() {
    cout << "Select the algorithm to use: (1: Reader first / 2: Writer first)"; //
    int sign;
    cin >> sign;
    solution(sign);
    cout<<"The end....."<<endl;
    system("pause");
}

```

评价表格

考核标准	得分
(1) 正确理解和掌握实验所涉及的概念和原理（20%）；	
(2) 按实验要求合理设计数据结构和程序结构，运行结果正确（40%）；	
(3) 认真记录实验数据，原理及实验结果分析准确，实验报告规范（40%）；	
合计	