

2020/07/30_Windows编程_第10课_线程间的竞争、解决同步问题的各种办法

笔记本: Windows编程

创建时间: 2020/7/30 星期四 10:42

作者: ileemi

- [资源竞争存在的问题](#)
- [线程同步问题](#)
- [锁变量](#)
- [Dekker算法](#)
- [临界区](#)
- [自旋锁](#)
- [内存锁](#)
- [生产者-消费者问题](#)
 - [信号量](#)
 - [使用信号量解决前面的同步问题](#)
 - [互斥体](#)
 - [事件对象](#)
 - [将临界区事件对象封装成类](#)
- [总结](#)

资源竞争存在的问题

两个进程同时访问一个文件，资源竞争在多进程年代就已经存在。

通过分析，再程序线程中的加加操作，程序会从物理内存中取出全局变量的数值进行"加加"操作，加完后的数据值再写回到物理内存中，但是由于各种原因，可能加加后的数据没能即使的写回到内存中去（切换时间片）。等待下次再进行"加加"操作，从内存中取出的数据还是之前的数据。

线程同步问题

在对全局变量进行操作的时候，系统会从物理内存中将全局变量的数据读取出来，然后将其进行对应的更改，更改后的数据还需要重写写入到内存中，如果当时间片到了，更改后的全局变量数据还没有来的及写入到内存中，另一个线程再次访问该全局变量，访问到的数据还是原来的数据。对全局变量的操作（加加，减减等）是需要一个过程的，不是立马可以完成的。这个时候对全局变量的结果来说就出现：**线程间的同步问题**。最后的值越小，不同步的次数就越多。

代码示例：

```

#include <stdio.h>
#include <Windows.h>
#define MAX_NUM 1000000
int g_nNum = 0;

// 工作线程
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    for (int i = 0; i < MAX_NUM; i++)
    {
        //printf("WorkThread:%d\r\n", (int)lpParameter);
        g_nNum++; // 解决这个问题的办法就是让这条语句完整的执行
    }
    return 0;
}

int main()
{
    HANDLE hThread[2];
    for (int i = 0; i < (sizeof(hThread) / sizeof(hThread[0])); i++)
    {
        hThread[i] = CreateThread(NULL, 0, &WorkThread, (LPVOID)i, 0, NULL);
    }

    // 等待线程全部结束
    WaitForMultipleObjects((sizeof(hThread) / sizeof(hThread[0])),
        hThread, NULL, INFINITE);
    printf("g_nNum = %d\r\n", g_nNum);

    return 0;
}

```

代码中的宏 MAX_NUM 值不断增加的时候，程序计算的结果和正确的结果有很大的差距。

当 MAX_NUM 的值为 10 时：程序运算结果为 20

当 MAX_NUM 的值为 1000 时：程序运算结果为 2000

当 MAX_NUM 的值为 10000 时：程序运算结果为 20000

当 MAX_NUM 的值为 1000000 时：程序运算结果为 1177362

```
3
4 #include <stdio.h>
5 #include <Windows.h>
6 #define MAX_NUM 1000000
7 int g_nNum = 0;
8
9 // 线程同步问题
10 DWORD WINAPI WorkThread(LPVOID lpParameter)
11 {
12     for (int i = 0; i < MAX_NUM; i++)
13     {
14         //printf("WorkThread:%d\r\n", (int)lpParameter);
15         g_nNum++; // 解决这个问题的办法
16     }
17     return 0;
18 }
19
```

Microsoft Visual Studio 调试控制台
g_nNum = 1131317
结果应为 2000000
D:\CR37\Works\第二阶段\Windows编程\Codes\20200730 -\Test1\Debug\Test1.exe (进程 11528) 已...
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”
按任意键关闭此窗口...

数值和正确结果相差越大说明其不同步问题越严重。

解决上述问题的办法：让计算全局变量数据的那条语句完整的执行（将计算后的结果成功写入的内存中再切换线程）

这个时候就提出了 **线程互斥** 的概念。

为了解决线程间的同步问题，为此科学家提出了**临界区**：

同步操作，一个进程不应该去影响另外一个进程

临界区：

1. 一个进程不应该阻塞另一个进程
2. 临界区同时只能一个进程进入
3. 不应该受CPU运行速度或者数量的影响

可以尝试封装两个函数（进入临界区，退出临界区），在对“区域”进程操作之前，首先进入临界区（在区域代码没有执行完毕，没有退出临界区前，其它线程不能进入进行操作），区域操作完成后，退出临界区。

锁变量

依然没有解决同步问题，判断的标志同样存在两个线程同时可以操控。

代码示例：

```
// 设置一个变量当作一个锁
int nTrue = 0;

// 进入临界区
void enter_region()
{
    while (nTrue != 0);
    nTrue = 1;
}

// 退出临界区
void leave_region()
{
    nTrue = 0;
}
```

```

}

// 线程同步问题
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    for (int i = 0; i < MAX_NUM; i++)
    {
        //printf("WorkThread:%d\r\n", (int)lpParameter);
        enter_region();
        g_nNum++; // 解决这个问题的办法就是让这条语句完整的执行
        leave_region();
    }
    return 0;
}

```

Dekker算法

1986年 荷兰数学家 Dekker 的解决办法（Dekker互斥算法）：**谦让算法（给两个标志）**，先抢到的把抢到的机会让给没有抢到的，解决线程间同时进入的问题。

1981年 Peterson 的算法是一个实现互斥锁的并发程序设计算法，可以控制两个线程访问一个共享的单用户资源而不发生访问冲突。第一个线程抢到让给第二个线程，第二个线程抢到让给第一个线程。

代码示例：

```

#include <stdio.h>
#include <Windows.h>
#define MAX_NUM 1000000
#define N 2

int g_nNum = 0;

// 设置一个变量当作一个锁
// Peterson 算法
int nTrue = 0;
int nInterested[N]; // 谁进入临界区
// 进入临界区（线程 0 1做测试）
void enter_region(int Process)
{
    // other 表示 谁进入
    int other = 1 - Process;
    nInterested[Process] = true;
    nTrue = Process;
    while (nTrue == Process && nInterested[other] == true); // 忙等待
}

```

```

// 退出临界区
void leave_region(int Process)
{
    nInterested[Process] = false;
}

// 线程同步问题
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    for (int i = 0; i < MAX_NUM; i++)
    {
        // 上锁
        enter_region((int)lpParameter);
        g_nNum++;
        // 解锁
        leave_region((int)lpParameter);
    }
    return 0;
}

int main()
{
    // 测试 Peterson 算法
    // 设置程序在第一核CPU上跑
    //SetProcessAffinityMask(GetCurrentProcess(), 1);

    HANDLE hThread[2];
    for (int i = 0; i < (sizeof(hThread) / sizeof(hThread[0])); i++)
    {
        hThread[i] = CreateThread(NULL, 0, &WorkThread, (LPVOID)i, 0, NULL);
    }

    WaitForMultipleObjects((sizeof(hThread) / sizeof(hThread[0])),
        hThread, NULL, INFINITE);
    printf("g_nNum = %d\r\n", g_nNum);

    return 0;
}

```



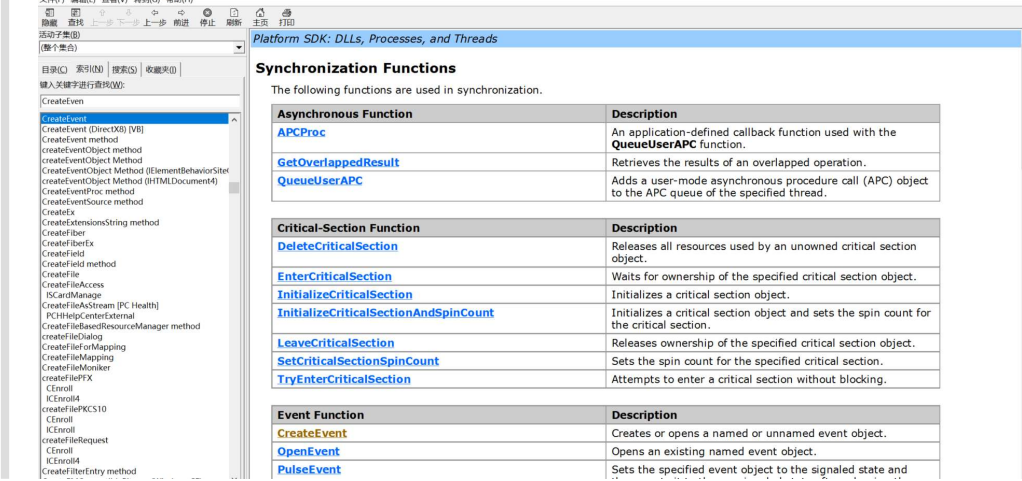
这种算法依然存在同步问题，同时效率上也有一定的问题，在进入临界区的时候存在忙等待。

线程的控制由操作系统管理，为此同步的代码应该有操作系统系统去完成。

一般的临界区算法都不自己做，自己做控制力比较弱，程序的性能发挥不到极致。

为此操作系统也就提供了对应的同步API，供使用。

EnterCriticalSection 等



Asynchronous Function	Description
APCProc	An application-defined callback function used with the QueueUserAPC function.
GetOverlappedResult	Retrieves the results of an overlapped operation.
QueueUserAPC	Adds a user-mode asynchronous procedure call (APC) object to the APC queue of the specified thread.

Critical-Section Function	Description
DeleteCriticalSection	Releases all resources used by an unowned critical section object.
EnterCriticalSection	Waits for ownership of the specified critical section object.
InitializeCriticalSection	Initializes a critical section object.
InitializeCriticalSectionAndSpinCount	Initializes a critical section object and sets the spin count for the critical section.
LeaveCriticalSection	Releases ownership of the specified critical section object.
SetCriticalSectionSpinCount	Sets the spin count for the specified critical section.
TryEnterCriticalSection	Attempts to enter a critical section without blocking.

Event Function	Description
CreateEvent	Creates or opens a named or unnamed event object.
OpenEvent	Opens an existing named event object.
PulseEvent	Sets the specified event object to the signaled state and...

临界区

- GetTickCount -- 获取当前程序运行的时间，还可以使用库函数：clock()
- EnterCriticalSection -- 等待指定临界区对象的所有权。当调用线程被授予所有权时，函数返回。进入临界区，在进入临界区的时候，进不去的时候（临界区内有其它线程正在操作，系统会将这个线程挂起），时间片到的时候会切走。
- InitializeCriticalSection -- 初始化一个临界区对象
- DeleteCriticalSection -- 释放无主临界区对象使用的所有资源
- LeaveCriticalSection -- 释放指定临界区对象的所有权

代码示例：

```
#include <stdio.h>
#include <Windows.h>
#include < time.h >

#define MAX_NUM 2
#define N 2

int g_nNum = 0;
CRITICAL_SECTION g_cs;    // 定义一个临界区的结构体对象
// 设置一个变量当作一个锁
// Peterson 算法
int nTrue = 0;
int nInterested[N];
// 进入临界区 (线程 0 1做测试)
void enter_region(int Process)
{
    int other = 1 - Process;
```

```

    nInterested[Process] = true;
    nTrue = Process;
    while (nTrue == Process && nInterested[other] == true); //忙等待
    //Sleep(100);
}

// 退出临界区
void leave_region(int Process)
{
    nInterested[Process] = false;
}

// 线程同步问题
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    for (int i = 0; i < MAX_NUM; i++)
    {
        EnterCriticalSection(&g_cs);    // 进入临界区
        g_nNum++;
        LeaveCriticalSection(&g_cs);    // 退出临界区
    }
    return 0;
}

int main()
{
    // 测试 Peterson 算法
    // 设置程序在第一核CPU上跑
    //SetProcessAffinityMask(GetCurrentProcess(), 1);
    InitializeCriticalSection(&g_cs);    // 初始化临界区(初始化同步对象)

    clock_t start, finish;
    start = clock();
    HANDLE hThread[2];
    for (int i = 0; i < (sizeof(hThread) / sizeof(hThread[0])); i++)
    {
        hThread[i] = CreateThread(NULL, 0, &WorkThread, (LPVOID)i, 0, NULL);
    }

    WaitForMultipleObjects((sizeof(hThread) / sizeof(hThread[0])),
hThread, NULL, INFINITE);
    finish = clock();
    printf("g_nNum = %d clock = %d\r\n", g_nNum, finish - start);
    // 释放无主临界区对象使用的所有资源
    DeleteCriticalSection(&g_cs);
    return 0;
}

```

2个线程同步的运行结果如下：

1. g_nNum = 2000000 耗时：66 mm
2. g_nNum = 2000000 耗时：57 mm
3. g_nNum = 2000000 耗时：69 mm

10个线程同步的运行结果如下：

1. g_nNum = 2000000 耗时：1966 mm
2. g_nNum = 2000000 耗时：1846 mm
3. g_nNum = 2000000 耗时：1766 mm

自旋锁

适合用在循环代码量较少的程序中，不需要切换线程而挂起。

一般通过硬件去实现，进程在跑的时候，硬件和CPU之间有一条总线，将这条总线关闭，等当前线程代码跑完，再将这条总线连接。这种做法微软没有提供对应的API，这种做法存在一定的局限性，当一个线程将总线关闭后没有连接，进程中后面的所有线程都不能在正常运行。

这种做法的效率要比临界区的效率要高，只限操作系统去使用。

内存锁

解决自旋锁的安全问题，硬件提供的同步功能。

导致局部变量在线程中不同步的原因主要是，去内存写内存的时间不一致造成的。

硬件提供了一种功能，内存和CPU之间通过内存总线连接，当CPU通过这个内存总线访问内存中的数据时，别的线程想要访问同位置上的数据，CPU会拒绝访问（给这个内存上“锁”）。这种办法只提供一条指令，内存解锁的过程不是有用户来决定，是由CPU决定，当CPU对其操作完成后，会自动为这个内存解锁（锁内存总线）。

存在缺点：**只能写一行代码**（只能保证一行代码是同步的，一次内存操作是同步的）

对变量的加加减减操作由CPU去帮助我们执行。

`InterlockedIncrement` -- 增加指定变量的值并检查结果值，该函数防止多个线程同时使用同一个变量（只能加一）。

参数：需要进行操作的变量的地址（`InterlockedIncrement((long*)&g_nNum);`）

`InterlockedExchangePointer` -- 原子地交换一对值

`InterlockedDecrement` -- 将指定变量的值递减(减少1)并检查结果值。

由于不存在切换线程，让时间片其效率要比临界区要高。

运行结果：

1. g_nNum = 2000000 耗时: 9 mm
2. g_nNum = 2000000 耗时: 10 mm
3. g_nNum = 2000000 耗时: 9 mm

示例代码:

```
#include <stdio.h>
#include <Windows.h>
#include <time.h>

#define MAX_NUM 1000000

int g_nNum = 0;

// 线程同步问题
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    for (int i = 0; i < MAX_NUM; i++)
    {
        InterlockedIncrement((long*)&g_nNum);
    }
    return 0;
}

int main()
{
    // 设置程序在第一核CPU上跑
    SetProcessAffinityMask(GetCurrentProcess(), 1);

    clock_t start, finish;
    start = clock();
    HANDLE hThread[2];
    for (int i = 0; i < (sizeof(hThread) / sizeof(hThread[0])); i++)
    {
        hThread[i] = CreateThread(NULL, 0, &WorkThread, (LPVOID)i, 0, NULL);
    }

    WaitForMultipleObjects((sizeof(hThread) / sizeof(hThread[0])),
        hThread, NULL, INFINITE);
    finish = clock();
    printf("g_nNum = %d clock = %d\r\n", g_nNum, finish - start);
    return 0;
}
```

生产者-消费者问题

存在问题：

生产者（线程1） 向一个数组或者链表中写入数据，消费者2、3（线程2，线程3）从数组中读取数据，线程1写入数据的时候，线程2，3从链表中读取数据，读取到的数据存在同步问题，读取到的数据存在缺陷，不全。同时，线程2，3不知道数组中的数据量。这个时候，在读取数据时，易陷入到死循环中出不来，线程卡死，线程1也不能继续向数组链表中插入数据。

这种做法效率比较低，时间都浪费到了检查上。

解决办法：可以添加一个引用计数，当写入一条数据，数据量计数加一，读入一条数据，数据量计数减一。这种办法称为：**信号量**（其不受进程的限制），这个信号别的进程也可以收到。

信号量

可以跨进程同步，临界区不可以

从本质上来讲，其不是在解决数据的同步问题，是在解决进程间的通讯的问题，同样可以利用在 **同步** 上。

信号量相关函数：

CreateSemaphore：创建或打开命名或未命名信号量对象

ReleaseSemaphore：将指定信号量对象的计数增加指定数量（投递信号量）

OpenSemaphore：打开一个信号量

WaitForSingleObject：等待线程

当信号量为1的时候，一个线程拿到信号的时候，另一个线程就会被挂起。当然也可以指定多个信号量。

锁之间可以混合使用，可以将临界区用在信号量中，可以将信号量也可以用来数据同步问题中。

信号量可以理解作为一种跨进程通知的一种东西。

代码实例：

```
#include <stdio.h>
#include <Windows.h>

int g_nAry[10] = { 0 };
int g_nCount = 0;

// 保存创建信号量成功后的句柄
HANDLE g_hSemp = NULL;
CRITICAL_SECTION g_cs;
// 线程同步问题
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    while (true)
    {
```

```

    // 等待信号, 判断数据
    WaitForSingleObject(g_hSemp, INFINITE);

    // 进入临界区
    EnterCriticalSection(&g_cs);
    printf("ThreadPID = %d %d\r\n",
        GetCurrentThreadId(), g_nAry[g_nCount - 1]);
    // 防止两个线程同时抢信号量产生同步问题, 这里使用临界区
    g_nCount--;
    // 退出临界区
    LeaveCriticalSection(&g_cs);
}

return 0;
}

int main()
{
    InitializeCriticalSection(&g_cs); //初始化同步对象
    // 创建一个信号量
    g_hSemp = CreateSemaphore(
        NULL, // 是否继承
        0,    // 初始信号量
        2,    // 最大信号量
        NULL // 信号量对象的名称, 支持多进程操作这个信号量
    );

    // 创建两个线程
    HANDLE hThread[2];
    for (int i = 0; i < (sizeof(hThread) / sizeof(hThread[0])); i++)
    {
        hThread[i] = CreateThread(NULL, 0, &WorkThread, (LPVOID)i, 0, NULL);
    }
    int nNum = 1;
    while (true)
    {
        scanf_s("%d", &nNum, sizeof(nNum));
        g_nAry[g_nCount++] = nNum;
        g_nAry[g_nCount++] = nNum + 1;
        // 输入一个数值, 就投递一个信号
        ReleaseSemaphore(g_hSemp, 2, NULL);
    }

    WaitForMultipleObjects((sizeof(hThread) / sizeof(hThread[0])),
        hThread, TRUE, INFINITE);
    return 0;
}

```

使用信号量解决前面的同步问题

需要创建信号量，使用信号量，投递信号量。

代码示例：

```
// 线程同步问题
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    for (int i = 0; i < MAX_NUM; i++)
    {
        //printf("WorkThread:%d\r\n", (int)lpParameter);
        //enter_region((int)lpParameter);
        //EnterCriticalSection(&g_cs);    // 进入临界区
        // 使用信号量
        // 等待信号，判断数据
        WaitForSingleObject(g_hSemp, INFINITE);
        g_nNum++;    // 这个问题的办法就是让这条语句完整的执行
        ReleaseSemaphore(g_hSemp, 1, NULL);
        //while (true);
        //InterlockedIncrement((long*)&g_nNum);
        //LeaveCriticalSection(&g_cs);    // 退出临界区
        //leave_region((int)lpParameter);
    }
    return 0;
}
```

运行结果：

1. g_nNum = 2000000 clock = 11355

通过运行结果可以看出，此方法虽然不可以解决数据的同步问题，但是其效率要比临界区处理同步问题还要低。

互斥体

信号量可以跨进程同步，临界区不支持。

简单版的信号量（其信号量只能是1），可以解决多进程同步问题。当给投递信号一个名字的时候，别的进程也可以收到。

当没有数量上的需求的时候，适合使用互斥体。不存在投递多少信息。

Mutex Function：

CreateMutex：创建或打开命名或未命名的互斥对象

ReleaseMutex：释放指定互斥对象的所有权

OpenMutex：打开一个现有的已命名互斥对象

代码示例：

```
// Test1.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结
束。
//
#include <stdio.h>
#include <Windows.h>
#include < time.h >

#define MAX_NUM 1000000
#define N 2

int g_nNum = 0;

// 存储创建互斥体的句柄
HANDLE g_hMutex = NULL;
// 线程同步问题
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    for (int i = 0; i < MAX_NUM; i++)
    {
        // 等待互斥体
        WaitForSingleObject(g_hMutex, INFINITE);
        g_nNum++;
        // 释放互斥体
        ReleaseMutex(g_hMutex);
    }
    return 0;
}

int main()
{
    // 设置程序在第一核CPU上跑
    //SetProcessAffinityMask(GetCurrentProcess(), 1);
    // 创建一个互斥体
    g_hMutex = CreateMutex(NULL, FALSE, NULL);
    clock_t start, finish;
    start = clock();
    HANDLE hThread[2];
    for (int i = 0; i < (sizeof(hThread) / sizeof(hThread[0])); i++)
    {
        hThread[i] = CreateThread(NULL, 0,
            &WorkThread, (LPVOID)i, 0, NULL);
    }

    WaitForMultipleObjects((sizeof(hThread) / sizeof(hThread[0])),
        hThread, TRUE, INFINITE);
}
```

```

        finish = clock();

        printf("g_num = %d  clock = %d\r\n", g_num, finish - start);

        return 0;
    }

```

运行结果：

1. g_num = 2000000 耗时：11151 ms
2. g_num = 2000000 耗时：10946 ms

通过结果可以观察到其和 信号量 处理同步的效率上差不过。

事件对象

和互斥体的用法基本类似，事件对象当信号来的时候产生一个事件。

自动用来做同步（参数：TRUE -- 手动，FALSE -- 自动），**手动用来跨进程通讯**。自动 -- 等到的信号会变成没信号，事件操作完成可以将其重置为初始的状态。

其和信号量存在实现的方式上有差异，一种是有信号或者没信号，另一种是有事件或者没事件，使用方式上基本类似。

Event Function：

CreateEvent -- 创建或打开已命名或未命名的事件对象

OpenEvent -- 打开现有的已命名事件对象

ResetEvent -- 将指定的事件对象设置为无信号状态

SetEvent -- 将指定的事件对象设置为有信号状态

代码示例：

```

// Test1.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结
束。
//
#include <stdio.h>
#include <Windows.h>
#include < time.h >

#define MAX_NUM 1000000
#define N 2

int g_num = 0;

// 存储创建事件对象的句柄
HANDLE g_hEvent = NULL;

```

```

// 线程同步问题
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    for (int i = 0; i < MAX_NUM; i++)
    {
        // 等待事件对象
        WaitForSingleObject(g_hEvent, INFINITE);
        g_nNum++;
        // 将指定的事件对象设置为有信号状态
        SetEvent(g_hEvent);
    }
    return 0;
}

int main()
{
    // 设置程序在第一核CPU上跑
    //SetProcessAffinityMask(GetCurrentProcess(), 1);
    // 创建一个事件对象 自动模式
    g_hEvent = CreateEvent(NULL, FALSE, TRUE, NULL);

    clock_t start, finish;
    start = clock();
    // 创建两个线程
    HANDLE hThread[2];
    for (int i = 0; i < (sizeof(hThread) / sizeof(hThread[0])); i++)
    {
        hThread[i] = CreateThread(NULL, 0,
            &WorkThread, (LPVOID)i, 0, NULL);
    }
    WaitForMultipleObjects((sizeof(hThread) / sizeof(hThread[0])),
        hThread, TRUE, INFINITE);
    finish = clock();
    printf("g_nNum = %d clock = %d\r\n", g_nNum, finish - start);

    return 0;
}

```

运行结果：

单核模式下：g_nNum = 2000000 耗时：3134 mm

非单核模式下：g_nNum = 2000000 耗时：12137 mm

跨进程通讯适合使用事件对象（支持手动自动），互斥体可以认为只能手动（互斥体一旦拿到，信号就没了）。而事件对象拿到事件后，还可以指定事件对象的状态（有

信号或者无信号)。

自己手动处理会存在同步问题，只适合进程间的通讯使用。

总结：事件对象可以用来处理同步问题，同时还可以用来跨进程通讯。

将临界区事件对象封装成类

代码示例：

```
#include <stdio.h>
#include <Windows.h>
#include < time.h >

class CCritical
{
public:
    CCritical()
    {
        // 初始化临界区(初始化同步对象)
        InitializeCriticalSection(&m_cs);
    }
    ~CCritical()
    {
        // 释放无主临界区对象使用的所有资源
        DeleteCriticalSection(&m_cs);
    }
    void lock()
    {
        EnterCriticalSection(&m_cs); // 进入临界区
    }

    void un_lock()
    {
        LeaveCriticalSection(&m_cs); // 退出临界区
    }
private:
    CRITICAL_SECTION m_cs; // 定义一个临界区的结构体对象
};

// 事件对象类
class CEvent
{
public:
    CEvent()
    {
        // 创建一个事件对象
    }
};
```



```

        m_hEvent = CreateEvent(NULL, FALSE, TRUE, NULL);
    }
    ~CEvent()
    {
        CloseHandle(m_hEvent);
    }
    void lock()
    {
        // 等待事件对象
        WaitForSingleObject(m_hEvent, INFINITE);
    }
    void un_lock()
    {
        // 将指定的事件对象设置为有信号状态
        SetEvent(m_hEvent);
    }
private:
    HANDLE m_hEvent;
};

```

使用示例:

```

#include "lock.h"

#define MAX_NUM 1000000
int g_nNum = 0;
CCritical g_Lock;

// 线程同步问题
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    for (int i = 0; i < MAX_NUM; i++)
    {
        g_Lock.lock();
        g_nNum++;
        g_Lock.un_lock();
    }
    return 0;
}

int main()
{
    clock_t start, finish;
    start = clock();
    HANDLE hThread[2];
    for (int i = 0; i < (sizeof(hThread) / sizeof(hThread[0])); i++)
    {

```

```

        hThread[i] = CreateThread(NULL, 0, &WorkThread, (LPVOID)i, 0, NULL);
    }

    WaitForMultipleObjects((sizeof(hThread) / sizeof(hThread[0])),
hThread, TRUE, INFINITE);
    finish = clock();
    printf("g_num = %d clock = %d\r\n", g_num, finish - start);
    return 0;
}

```

总结

MFC中同步对象封装的有对应的类：

CEvent 继承 CSyncObject

- CCriticalSection
- CEvent
- CMutex
- CSemaphore

1.临界区

2.自旋锁 硬件

3.内存锁 锁内存总线

4.信号量

5.互斥体

6.事件对象

不跨进程：临界区、内存锁（锁内存总线，两个进程操作的内存要是同一个，所以需要共享，共享比较麻烦）

跨进程：自旋锁、信号量、互斥体、事件对象 ==> 同步 通讯

PostThreadMessage 发送线程消息，接收消息的进程需要在线程中写消息循环。