

C#多线程同步，C#中四种进程或线程同步互斥的控制方法

疯狂代码 <http://CrazyCoder.cn/> j:<http://CrazyCoder.cn/crazycoder/Article1784.html>

现在流行的进程线程同步互斥的控制机制，其实是由最原始最基本的4种方法实现的。由这4种方法组合优化就有了.Net和Java下灵活多变的，编程简便的线程进程控制手段。这4种方法具体定义如下 在《操作系统教程》ISBN 7-5053-6193-7 一书中可以找到更加详细的解释

- 1临界区:通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。
- 2互斥量:为协调共同对一个共享资源的单独访问而设计的。
- 3信号量:为控制一个具有有限数量用户资源而设计。
- 4事件:用来通知线程有一些事件已发生，从而启动后继任务的开始。

临界区 (Critical Section) 保证在某一时刻只有一个线程能访问数据的简便办法。在任意时刻只允许一个线程对共享资源进行访问。如果有多个线程试图同时访问临界区，那么在有一个线程进入后其他所有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线程离开。临界区在被释放后，其他线程可以继续抢占，并以此达到用原子方式操作共享资源的目的。临界区包含两个操作原语：EnterCriticalSection () 进入临界区 LeaveCriticalSection () 离开临界区 EnterCriticalSection () 语句执行后代码将进入临界区以后无论发生什么，必须确保与之匹配的LeaveCriticalSection () 都能够被执行到。否则临界区保护的共享资源将永远不会被释放。虽然临界区同步速度很快，但却只能用来同步本进程内的线程，而不可用来同步多个进程中的线程。MFC提供了很多功能完备的类，我用MFC实现了临界区。MFC为临界区提供有一个CCriticalSection类，使用该类进行线程同步处理是非常简单的。只需在线程函数中用CCriticalSection类成员函数Lock () 和UnLock () 标定出被保护代码片段即可。Lock () 后代码用到的资源自动被视为临界区内的资源被保护。UnLock后别的线程才能访问这些资源。代码：

```
;//CriticalSection
;CCriticalSection global_CriticalSection;
;;;
;// 共享资源
;char global_Array[256];
;;;
;//初始化共享资源
;void InitializeArray()
;{
; ; for(int i = 0;i<256;i++)
; ; {
; ; global_Array[i]=I;
; ; }
; }
; ;
;//写线程
;UINT Global_ThreadWrite(LPVOID pParam)
```

```

;{
;; CEdit *ptr=(CEdit *)pParam;
;; ptr->SetWindowText("");
;; //进入临界区
; global_CriticalSection.Lock();
;; for(int i = 0;i<256;i++)
;; {
;; global_Array[i]=W;
;; ptr->SetWindowText(global_Array);
;; Sleep(10);
;; } //离开临界区
;; global_CriticalSection.Unlock();
;; return 0;
;}
;;;
//删除线程
; UINT Global_ThreadDelete(LPVOID pParam)
;{
;; CEdit *ptr=(CEdit *)pParam;
;; ptr->SetWindowText("");
;; //进入临界区
;; global_CriticalSection.Lock();
;; for(int i = 0;i<256;i++)
;; {
;; global_Array[i]=D;
;; ptr->SetWindowText(global_Array);
;; Sleep(10);
;; }
;;;
//离开临界区
;; global_CriticalSection.Unlock();
;; return 0;
;}
;;;
//创建线程并启动线程
; void CCriticalSectionDlg::OnBnClickedButtonLock()

```

```

;{
;; //Start the first Thread
;; CWinThread *ptrWrite = AfxBeginThread(Global_ThreadWrite,
;; &m_Write,
;; THREAD_PRIORITY_NORMAL,
;; 0,
;; CREATE_SUSPENDED);
;; ptrWrite->ResumeThread();
;;
;; //Start the second Thread
;; CWinThread *ptrDelete = AfxBeginThread(Global_ThreadDelete,
;; &m_Delete,
;; THREAD_PRIORITY_NORMAL,
;; 0,
;; CREATE_SUSPENDED);
;; ptrDelete->ResumeThread();
};

```

在测试程序中，Lock UnLock两个按钮分别实现，在有临界区保护共享资源的执行状态，和没有临界区保护共享资源的执行状态。互斥量（Mutex） 互斥量跟临界区很相似，只有拥有互斥对象的线程才具有访问资源的权限，由于互斥对象只有一个，因此就决定了任何情况下此共享资源都不会同时被多个线程所访问。当前占据资源的线程在任务处理完后应将拥有的互斥对象交出，以便其他线程在获得后得以访问资源。互斥量比临界区复杂。因为使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。 互斥量包含的几个操作原语：

CreateMutex（ ） 创建一个互斥量

OpenMutex（ ） 打开一个互斥量

ReleaseMutex（ ） 释放互斥量

WaitForMultipleObjects（ ） 等待互斥量对象 同样MFC为互斥量提供一个CMutex类。使用CMutex类实现互斥量操作非常简单，但是要特别注意对CMutex的构造函数的调用

CMutex(BOOL bInitiallyOwn = FALSE, LPCTSTR lpszName = NULL, LPSECURITY_ATTRIBUTES
lpsaAttribute = NULL) 不用的参数不能乱填，乱填会出现一些意想不到的运行结果。代码:

```

//创建互斥量
; CMutex global_Mutex(0,0,0);
;;;
// 共享资源
; char global_Array[256];

```

```

;;;
; void InitializeArray()
; {
;; for(int i = 0;i<256;i++)
;; {
;; global_Array[i]=I;
;; }
; }
; UINT Global_ThreadWrite(LPVOID pParam)
; {
;; CEdit *ptr=(CEdit *)pParam;
;; ptr->SetWindowText("");
;; global_Mutex.Lock();
;; for(int i = 0;i<256;i++)
;; {
;; global_Array[i]=W;
;; ptr->SetWindowText(global_Array);
;; Sleep(10);
;; }
;; global_Mutex.Unlock();
;; return 0;
; }
;;;
; UINT Global_ThreadDelete(LPVOID pParam)
; {
;; CEdit *ptr=(CEdit *)pParam;
;; ptr->SetWindowText("");
;; global_Mutex.Lock();
;; for(int i = 0;i<256;i++)
;; {
;; global_Array[i]=D;
;; ptr->SetWindowText(global_Array);
;; Sleep(10);
;; }
;; global_Mutex.Unlock();
;; return 0;

```

```
};
```

同样在测试程序中，Lock UnLock两个按钮分别实现，在有互斥量保护共享资源的执行状态，和没有互斥量保护共享资源的执行状态。

信号量 (Semaphores) 信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中的PV操作相同。它指出了同时访问共享资源的线程最大数目。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。在用CreateSemaphore () 创建信号量时即要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减1，只要当前可用资源计数是大于0的，就可以发出信号量信号。但是当前可用计数减小到0时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离开的同时通过ReleaseSemaphore () 函数将当前可用资源计数加1。在任何时候当前可用资源计数决不可能大于最大资源计数。 PV操作及信号量的概念都是由荷兰科学家E.W.Dijkstra提出的。信号量S是一个整数，S大于等于零时代表可供并发进程使用的资源实体数，但S小于零则表示正在等待使用共享资源的进程数。 P操作申请资源：

- (1) S减1；
- (2) 若S减1后仍大于等于零，则进程继续执行；
- (3) 若S减1后小于零，则该进程被阻塞后进入与该信号相对应的队列中，然后转入进程调度。

V操作 释放资源：

- (1) S加1；
 - (2) 若相加结果大于零，则进程继续执行；
 - (3) 若相加结果小于等于零，则从该信号的等待队列中唤醒一个等待进程，然后再返回原进程继续执行或转入进程调度。
- 信号量包含的几个操作原语：

CreateSemaphore () 创建一个信号量
OpenSemaphore () 打开一个信号量
ReleaseSemaphore () 释放信号量
WaitForSingleObject () 等待信号量代码:

```
;//信号量句柄  
;HANDLE global_Semaphore;  
;;;  
;// 共享资源  
; char global_Array[256];  
; void InitializeArray()  
;{  
;; for(int i = 0;i<256;i++)  
;;{
```

```

;; global_Array[i]=I;
;;}
;}
;
//线程1
; UINT Global_ThreadOne(LPVOID pParam)
;{
;; CEdit *ptr=(CEdit *)pParam;
;; ptr->SetWindowText("");
;; //等待对共享资源请求被通过 等于 P操作
; WaitForSingleObject(global_Semaphore, INFINITE);
;; for(int i = 0;i<256;i++)
;;{
;; global_Array[i]=O;
;; ptr->SetWindowText(global_Array);
;; Sleep(10);
;;}
;
//释放共享资源 等于 V操作
;; ReleaseSemaphore(global_Semaphore, 1, NULL);
;; return 0;
;}
;;;
; UINT Global_ThreadTwo(LPVOID pParam)
;{
;; CEdit *ptr=(CEdit *)pParam;
;; ptr->SetWindowText("");
;; WaitForSingleObject(global_Semaphore, INFINITE);
;; for(int i = 0;i<256;i++)
;;{
;; global_Array[i]=T;
;; ptr->SetWindowText(global_Array);
;; Sleep(10);
;;}
;; ReleaseSemaphore(global_Semaphore, 1, NULL);
;; return 0;

```

```

; }
;;;
; UINT Global_ThreadThree(LPVOID pParam)
; {
; ; CEdit *ptr=(CEdit *)pParam;
; ; ptr->SetWindowText("");
; ; WaitForSingleObject(global_Semaphore, INFINITE);
; ; for(int i = 0;i<256;i++)
; ; {
; ; global_Array[i]=H;
; ; ptr->SetWindowText(global_Array);
; ; Sleep(10);
; ; }
; ; ReleaseSemaphore(global_Semaphore, 1, NULL);
; ; return 0;
; }
;;;
; void CSemaphoreDlg::OnBnClickedButtonOne()
; { ; ; //设置信号量 1 个资源 1同时只可以有一个线程访问
; ; global_Semaphore= CreateSemaphore(NULL, 1, 1, NULL);
; ; this->StartThread();
;
; // TODO: Add your control notification handler code here
; }
;;;
; void CSemaphoreDlg::OnBnClickedButtonTwo()
; { ; ; //设置信号量 2 个资源 2 同时只可以有二个线程访问
; ; global_Semaphore= CreateSemaphore(NULL, 2, 2, NULL);
; ; this->StartThread(); ; ; // TODO: Add your control notification handler code here
; }
;;;
; void CSemaphoreDlg::OnBnClickedButtonThree()
; { ; //设置信号量 3 个资源 3 同时只可以有三个线程访问
; ; global_Semaphore= CreateSemaphore(NULL, 3, 3, NULL);
; ; this->StartThread();
;
;

```

```
; // TODO: Add your control notification handler code here
};
```

信号量的使用特点使其更适用于对Socket（套接字）程序中线程的同步。例如，网络上的HTTP服务器要对同一时间内访问同一页面的用户数加以限制，这时可以为每一个用户对服务器的页面请求设置一个线程，而页面则是待保护的共享资源，通过使用信号量对线程的同步作用可以确保在任一时刻无论有多少用户对某一页面进行访问，只有不大于设定的最大用户数目的线程能够进行访问，而其他的访问企图则被挂起，只有在有用用户退出对此页面的访问后才有可能进入。

事件（Event） 事件对象也可以通过通知操作的方式来保持线程的同步。并且可以实现不同进程中的线程同步操作。 信号量包含的几个操作原语：

CreateEvent（） 创建一个信号量

OpenEvent（） 打开一个事件

SetEvent（） 回置事件

WaitForSingleObject（） 等待一个事件

WaitForMultipleObjects（） 等待多个事件 WaitForMultipleObjects 函数原型：

WaitForMultipleObjects（

IN DWORD nCount, // 等待句柄数

IN CONST HANDLE *lpHandles, //指向句柄数组

IN BOOL bWaitAll, //是否完全等待标志

IN DWORD dwMilliseconds //等待时间

） 参数nCount指定了要等待的内核对象的数目，存放这些内核对象的数组由lpHandles来指向。

bWaitAll对指定的这nCount个内核对象的两种等待方式进行了指定，为TRUE时当所有对象都被通知时函数才会返回，为FALSE则只要其中任何一个得到通知就可以返回。dwMilliseconds在这里的作用与在WaitForSingleObject（）中的作用是完全一致的。如果等待超时，函数将返回WAIT_TIMEOUT。代码：

```
; //事件数组
; HANDLE global_Events[2];
;;;
; // 共享资源
; char global_Array[256];
;;;
; void InitializeArray()
; {
; ; for(int i = 0;i<256;i++)
; ; {
; ; global_Array[i]=I;
; ; }
; }
```

```
;;;
; UINT Global_ThreadOne(LPVOID pParam)
; {
;; CEdit *ptr=(CEdit *)pParam;
;; ptr->SetWindowText("");
;; for(int i = 0;i<256;i++)
;; {
;; global_Array[i]=O;
;; ptr->SetWindowText(global_Array);
;; Sleep(10);
;; }
;
; //回置事件
;; SetEvent(global_Events[0]);
;; return 0;
; }
;;;
; UINT Global_ThreadTwo(LPVOID pParam)
; {
;; CEdit *ptr=(CEdit *)pParam;
;; ptr->SetWindowText("");
;; for(int i = 0;i<256;i++)
;; {
;; global_Array[i]=T;
;; ptr->SetWindowText(global_Array);
;; Sleep(10);
;; }
;
; //回置事件
;; SetEvent(global_Events[1]);
;; return 0;
; }
;;;
; UINT Global_ThreadThree(LPVOID pParam)
; {
;; CEdit *ptr=(CEdit *)pParam;
```

```

;; ptr->SetWindowText("");
;
;; //等待两个事件都被回置
;; WaitForMultipleObjects(2, global_Events, true, INFINITE);
;; for(int i = 0;i<256;i++)
;; {
;; global_Array[i]=H;
;; ptr->SetWindowText(global_Array);
;; Sleep(10);
;; }
;; return 0;
;}
; void CEventDlg::OnBnClickedButtonStart()
; {
;; for (int i = 0; i < 2; i++)
;; {
;
;; //实例化事件
;; global_Events[i]=CreateEvent(NULL,false,false,NULL);
;; }
;; CWinThread *ptrOne = AfxBeginThread(Global_ThreadOne,
;; &m_One,
;; THREAD_PRIORITY_NORMAL,
;; 0,
;; CREATE_SUSPENDED);
;; ptrOne->ResumeThread();
;;;
;; //Start the second Thread
;; CWinThread *ptrTwo = AfxBeginThread(Global_ThreadTwo,
;; &m_Two,
;; THREAD_PRIORITY_NORMAL,
;; 0,
;; CREATE_SUSPENDED);
;; ptrTwo->ResumeThread();
;;;
;; //Start the Third Thread

```

```
;; CWinThread *ptrThree = AfxBeginThread(Global_ThreadThree,  
;; &m_Three,  
;; THREAD_PRIORITY_NORMAL,  
;; 0,  
;; CREATE_SUSPENDED);  
;; ptrThree->ResumeThread(); ; ; // TODO: Add your control notification handler code here  
;}
```

事件可以实现不同进程中的线程同步操作，并且可以方便的实现多个线程的优先比较等待操作，例如写多个WaitForSingleObject来代替WaitForMultipleObjects从而使编程更加灵活。

总结： 1. 互斥量与临界区的作用非常相似，但互斥量是可以命名的，也就是说它可以跨越进程使用。所以创建互斥量需要的资源更多，所以如果只为了在进程内部是用的话使用临界区会带来速度上的优势并能够减少资源占用量。因为互斥量是跨进程的互斥量一旦被创建，就可以通过名字打开它。 2. 互斥量

(Mutex)，信号灯 (Semaphore)，事件 (Event) 都可以被跨越进程使用来进行同步数据操作，而其他的对象与数据同步操作无关，但对于进程和线程来讲，如果进程和线程在运行状态则为无信号状态，在退出后为有信号状态。所以可以使用WaitForSingleObject来等待进程和线程退出。 3. 通过互斥量可以指定资源被独占的方式使用，但如果有下面一种情况通过互斥量就无法处理，比如现在一位用户购买了一份三个并发访问许可的数据库系统，可以根据用户购买的访问许可数量来决定有多少个线程/进程能同时进行数据库操作，这时候如果利用互斥量就没有办法完成这个要求，信号灯对象可以说是一种资源计数器。 2008-6-13

16:40:54

疯狂代码 <http://CrazyCoder.cn/>