**Windows线程同步的几种方式**

仲达超

QQ:814865808

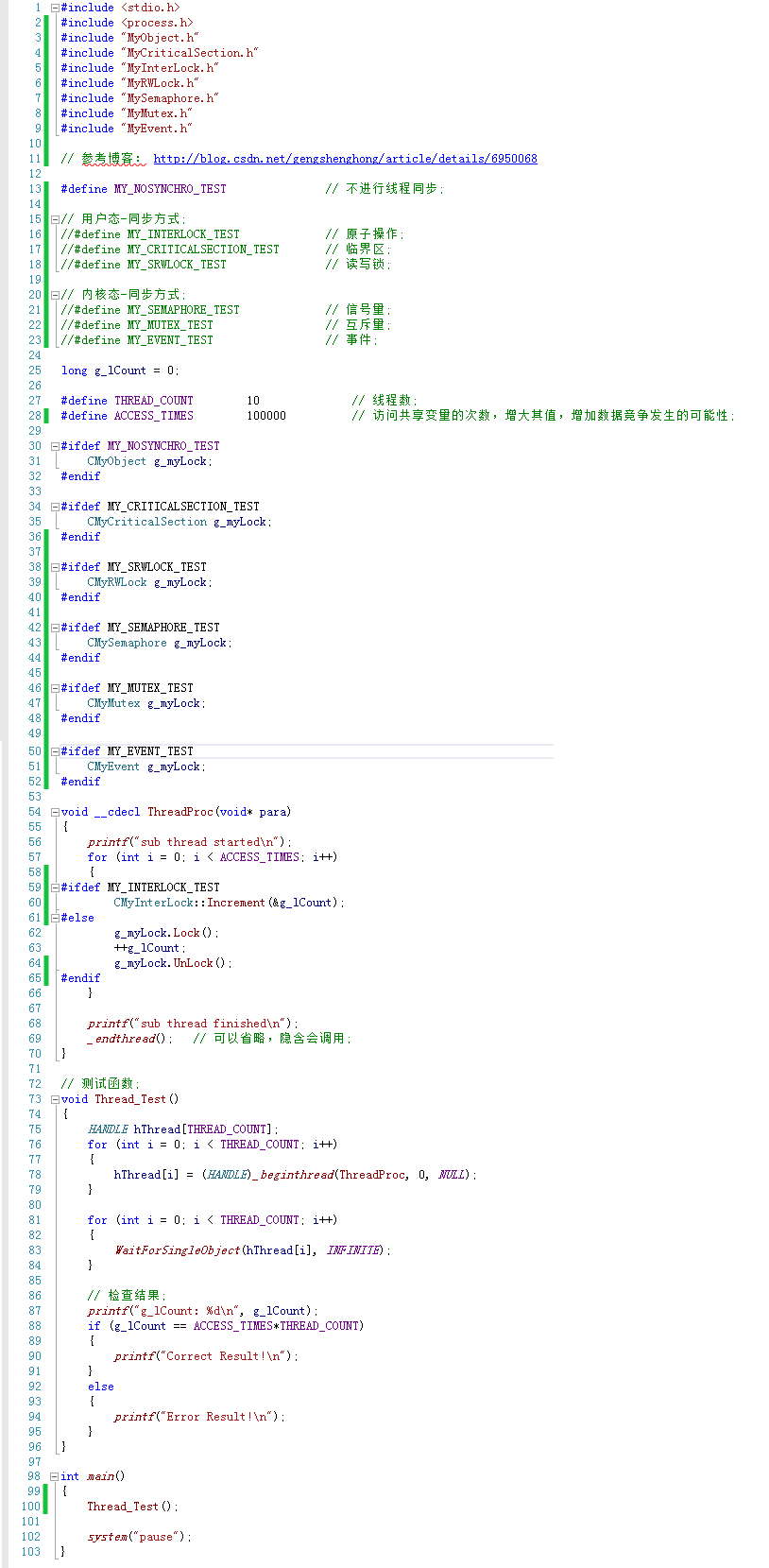
2017年2月

对于多线程编程，一个很重要的问题就是解决由数据共享引起的数据竞争的问题，通过一定的线程同步的方法能避免数据竞争。在Win32多线程中，同步方法包括**用户态同步方式**：InterLock、CriticalSection、SRWLock和**内核态同步方式**：Event、Semaphore、Mutex等。

本文主要目的是收集这些同步的方法和涉及到API的含义和使用，可能不会涉及到更深入的比如何时使用何种方式更好以及为什么，更深入的问题，单独在以后积累并记录。

一、数据竞争的例子

在分析同步的方法之前，先给出要解决的数据竞争的例子：

  
如果程序输出Error Result，说明发生了数据竞争。

二、用户态同步

需要注意的是，由于用户态到内核态的转换需要一定的开销，所以如果能在用户态进行同步控制，尽量选择用户态的方式。

（1）InterLocked原子操作

InterLocked是一系列的方法，提供了原子操作的实现。原子操作比较容易理解，直接调用对应的API就能进行简单的同步了，但是这种方式，显然只能适用于简单的预定义的一些操作方法，比如自增、自减、加法等，具体有哪些方法，可以参考MSDN（<http://msdn.microsoft.com/zh-cn/site/ms684122>）

对于上面的数据竞争的例子，显然可以使用自增的原子操作来解决了，对于自增操作，MS提供了InterlockedIncrement()和InterlockedIncrement64()来完成原子操作，分别操作32位和64位的整数(long,long long)，参数为要进行自增的变量的地址。

（2）Critical Section临界区

不论是硬件临界资源，还是软件临界资源，多个进程必须互斥地对它进行访问。每个进程中访问临界资源的那段代码称为临界区（Critical Section）（临界资源是一次仅允许一个进程使用的共享资源）。每次只准许一个进程进入临界区，进入后不允许其他进程进入。不论是硬件临界资源，还是软件临界资源，多个进程必须互斥地对它进行访问。

与临界区相关的函数有：

EnterCriticalSection

LeaveCriticalSection

InitializeCriticalSection

DeleteCriticalSection

TryEnterCriticalSection。

EnterCriticalSection和LeaveCriticalSection是一对操作，表示进入和离开临界区，同步控制一段代码的访问，即在这两个函数之间调用的代码，每次只允许一个线程执行。

InitializeCriticalSection和DeleteCriticalSection也是一对操作，分别是初始化和删除临界区（变量），在使用临界区的时候，需要定义一个临界区变量，代表一个临界区资源。所以，一个程序可以使用多个临界区变量来进行不同的代码段的保护。在调用EnterCriticalSection的时候，如果临界区被其它线程占有，那么当前线程会等待资源，知道其它线程退出临界区。TryEnterCriticalSection函数就是用来尝试进入临界区，如果无法进入临界区(临界区被占有)，那么返回FALSE，不会阻塞线程。

下面是使用临界区解决上述问题的代码：

可见，临界区能保护一个代码块，使用起来比原子操作更灵活。

（3）SRWLOCK读写锁

关于读写锁：<http://baike.baidu.com/view/2214179.htm>（百度百科）

读写锁实际是一种特殊的自旋锁，它把对共享资源的访问者划分成读者和写者，读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。这种锁相对于自旋锁而言，能提高并发性，因为在多处理器系统中，它允许同时有多个读者来访问共享资源，最大可能的读者数为实际的逻辑CPU数。写者是排他性的，一个读写锁同时只能有一个写者或多个读者（与CPU数相关），但不能同时既有读者又有写者。

说明：在win32中，自旋锁是内核定义，且只能在内核状态下使用的一种锁机制。所以一般的程序不会用到，需要一定的要求。而SRWLOCK读写锁是用户态下的读写锁。

从使用上，SRWLOCK和临界区的使用非常类似，读写锁和临界区的主要区别是，读写锁对共享资源的访问区分了读者和写者分离的功能。所以，SRWLOCK比临界区多了一些方法。

与SRWLOCK相关的函数主要有：（具体可参考<http://msdn.microsoft.com/zh-cn/library/aa904937.aspx>介绍所有的读写锁相关的API）

AcquireSRWLockShared  
AcquireSRWLockExclusive  
ReleaseSRWLockShared  
ReleaseSRWLockExclusive  
InitializeSRWLock  
TryAcquireSRWLockExclusive  
TryAcquireSRWLockShared  
SleepConditionVariableSRW

其中，AcquireSRWLockShared和AcquireSRWLockExclusive表示获取读锁和获取写锁（共享锁和排他锁）。ReleaseSRWLockShared和ReleaseSRWLockExclusive表示释放读锁和写锁。和临界区一样，InitializeSRWLock是初始化。但是，SRWLock没有提供删除读写锁的方法，不需要删除。TryAcquireSRWLockExclusive和TryAcquireSRWLockShared也是用于非阻塞的方式获取读锁和写锁，失败返回FALSE。SleepConditionVariableSRW在下面再介绍。需要说明的是：TryAcquireSRWLockExclusive和TryAcquireSRWLockShared是Win7才开始提供支持的，详细信息参考MSDN。

读写锁的适用情况：读写锁适合于对数据结构的读次数比写次数多得多的情况。更多相关问题，参考关于读写锁的特性等的分析介绍。

（4）Condition Variable条件变量

MSDN:<http://msdn.microsoft.com/zh-cn/site/ms682052>

条件变量一开始是在Linux中有，Window平台是从Vista才开始支持条件变量（所以XP不支持）。

关于条件变量本身的解释，参考百度百科（<http://baike.baidu.com/view/4025952.htm>），当然，百度百科里面都是说的Linux，但是概念本身其实是类似的。如下：

条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待"**条件变量的条件成立"而挂起**；另一个线程**使"条件成立"（给出条件成立信号）**。为了防止竞争，**条件变量的使用总是和一个互斥锁结合在一起**。其实，**在windows下，条件变量的使用总是和读写锁或临界区结合使用**（因为Linux下没有临界区，所以这里只说了互斥锁）。

和条件变量相关的函数有：

Condition variable function  
InitializeConditionVariable  
SleepConditionVariableCS  
SleepConditionVariableSRW  
WakeAllConditionVariable  
WakeConditionVariable

具体使用说明可以参考MSDN，由于条件变量和这里的数据竞争的例子很难结合起来，这里就不举例了，而且它单独是无法完成同步的，所以这里也没必要单独作为一种同步的方法来说明，关于条件变量的使用场合和使用方法，参考其它内容。

三、内核态同步

上面介绍的都是用户态的同步方法，win32多线程还提供了一些内核态同步的方式。从性能上来说，内核态同步方式比用户态更低，原因是用户态到内核态的转换是有开销的。但是内核态的优点在于是可以**跨进程同步**的，所以不仅仅是线程同步方式，也是一种进程同步方式。

（1）内核对象和状态

在了解内核态同步之前，首先需要了解很重要的两个函数：WaitForSingleObject和WaitForMultipleObjects。

1. 内核对象

**内核对象只是内核分配的一个内存块，并且只能由该内核访问。该内存块是一种数据结构，它的成员负责维护该对象的各种信息。**有些数据成员（如安全性描述符、使用计数等）在所有对象类型中是相同的，但**大多数数据成员属于特定的对象类型**。例如，进程对象有一个进程I D 、一个基本优先级和一个退出代码，而文件对象则拥有一个字节位移、一个共享模式和一个打开模式。

参考：

<http://www.cnblogs.com/fangyukuan/archive/2010/08/31/1813117.html>

<http://i.mtime.com/MyFighting/blog/1793762/>

总之，这里要提到的内核态同步的对象，都是属于内核对象，包括进程对象和线程对象也是属于内核对象。另外要知道的是，内核对象使用相应的创建函数创建，返回都是句柄，即HANDLE对象。

2. 内核同步对象

在Windows NT内核对象中，提供了五种内核同步对象（Kernel Dispatcher Object），为：Event(事件)、Semaphore(信号灯/信号量)、Mutex(互斥)、Timer(定时器)、Thread(线程)。

关于内核同步对象可以参考：<http://hi.baidu.com/klksys/blog/item/2c470ad25808cdd6a9ec9aaa.html>

3. 内核对象的状态

在任何时刻，任何对象都处于两种状态中的一种：信号态或非信号态（参考上面的链接的说明，没有找到官方证实这句话，但是至少对于内核同步对象，所有的对象应该都是有这两个状态的）。有时候，这两个状态称为受信状态（signaled state)和非受信状态(nonsignaled state)或者通知状态和未通知状态。

（参考<http://st251256589.blog.163.com/blog/static/16487644920111244054511/>）。

到了这里，我们就可以讨论WaitForSingleObject了。WaitForSingleObject的参数是一个内核对象句柄，它的作用是：Waits until the specified object is in the signaled state or the time-out interval elapses。即等待指定的对象处于受信状态或者出现超时，等待，表明如果执行WaitForSingleObject的时候，对象处于非受信状态，那么当前线程处于阻塞状态。当然，WaitForMultipleObjects的作用就是等待多个状态了。

说明，WaitForSingleObject对于某些内核对象是由副作用的，比如对于信号量，等待成功会使得信号量减1。

参考MSDN：<http://msdn.microsoft.com/zh-cn/site/ms687032>可以知道WaitForSingleObject的用法和它能等待的所有内核对象：

Change notification  
Console input  
**Event**  
Memory resource notification  
**Mutex**  
Process  
**Semaphore  
Thread**  
Waitable timer

其中加粗的几个内核对象是多线程编程中会遇到的（三个内核态同步对象和一个线程对象）。理解了signaled state和nonsignaled state之后，下面的三种内核态同步方式就很容易理解了。

（2）Event事件

MSDN：<http://msdn.microsoft.com/zh-cn/site/ms682655> （Event Objects）

事件内核对象包括两种：人工重置的事件和自动重置的事件。

当人工重置事件得到通知时，等待该事件的所有线程成为可调度线程；它没有成功等待副作用。  
当自动重置的事件得到通知时，等待该事件的线程中只有一个线程变为可调度线程。其成功等待的副作用是该对象自动重置为未通知状态。

事件内核对象通过CreateEvent创建，初始可以是通知或未通知状态。

人工事件一般用于一个线程通知另一个线程或者一个线程通知多个线程进行某一操作。自动事件适用于保护资源在同一时间只能有一个线程可以访问，它能保证只有一个线程被激活。

事件对象分为命名事件对象和未命名对象（named and unnamed event objects）。

和事件对象相关的函数有：CreateEvent/OpenEvent、ResetEvent、SetEvent、PulseEvent等。

其中，CreateEvent创建或打开一个事件对象，其中，四个参数分别为安全属性（内核对象都有安全属性），是否为人工重置事件，初始状态（TRUE表示signaled，FALSE表示nonsignled），事件对象的名字，如果为NULL，创建一个未命名事件对象。如果指定name的事件已经存在，则获得EVENT\_ALL\_ACCESS的访问权限，第一个参数部分有效，后两个参数忽略。OpenEvent用于打开已存在的事件（所以一般用CreateEvent即可）。ResetEvent/SetEvent分别设置事件的状态为nonsignaled和signaled。PulseEvent根据MSDN不可信，所以不推荐使用（相当于reset然后set的功能，但是并不可靠）。

总之，对于事件来说，常用的方法是CreateEvent，ResetEvent，SetEvent，然后利用WaitForSingleObject来等待事件（变成singled状态）。  
说明：实际运行就会发现，内核态事件对象同步的方法比用户态的方法的效率低很多，如果这里的ACCESS\_TIMES如果太大，运行时间相比用户态的方法多很多。当然，再次强调，这里都是用这一个例子，只是为了分析各种方法实现同步的实现，实际应用，显然是有所取舍的，不同的同步方法有不同的合适的使用情景。

（3）Semaphore信号量

MSDN：<http://msdn.microsoft.com/zh-cn/site/ms685129>（Semaphore Objects）

信号量用来对资源进行计数。它包含两个32位值，一个表示能够使用的最大资源数量，一个表示当前可用的资源数量。  
**信号量的使用规则为：如果当前资源数量大于0，发出信号量信号；如果当前资源数量是0，不发出信号量信号；不允许当前资源数量为负值  
 当前资源数量不能大于最大信号数量**。

当调用等待函数时，它会检查信号量的当前资源数量。如果它的值大于0，那么计数器减1，调用线程处于可调度状态。如果当前资源是0，则调用函数的线程进入等待状态。当另一个线程对信号量的当前资源通过ReleaseSemaphore进行递增时，系统会记住该等待线程，并将其变为可调度状态。

对于信号量，相关的函数有：CreateSemaphore/OpenSemaphore、ReleaseSemaphore。WaitForSingleObject对于信号量，成功等待的副作用是使得信号量减1。从某种角度理解，事件相当于最大计数为1的信号量。

（4）Mutex互斥（互斥对象，互斥体）

MSDN：<http://msdn.microsoft.com/zh-cn/site/ms684266>（Mutex Objects）

互斥器保证线程拥有对单个资源的互斥访问权。**互斥对象类似于关键代码区（临界区），但它是一个内核对象**。 互斥器不同于其他内核对象，它有一个“**线程所有权**”的概念。它如果被某个线程等待成功，就属于该线程。

由于和临界区和读写锁很类似，使用也是很类似的。和Mute相关的函数主要有：CreateMutex/OpenMutex，ReleaseMutex。很显然，Create是创建，Open是打开已存在的命名互斥对象，ReleaseMutex是释放互斥对象。几乎和临界区的函数一样，当然，用WaitForSingleObject等待互斥体，类似于进入临界区的操作了。

总结：这里介绍了用户态和内核态的同步对象和基本函数的使用。这里只是为了演示其使用和理解其概念，针对实际应用，需要根据实际的case选择合适的方法进行同步。

MFC提供了多种同步对象，下面我们只介绍最常用的四种：

* 临界区（CCriticalSection）
* 事件（CEvent）
* 互斥量（CMutex）
* 信号量（CSemaphore）

***A、使用 CCriticalSection 类***

　　当多个线程访问一个独占性共享资源时,可以使用“临界区”对象。任一时刻只有一个线程可以拥有临界区对象，拥有临界区的线程可以访问被保护起来的资源或代码段，其他希望进入临界区的线程将被挂起等待，直到拥有临界区的线程放弃临界区时为止，这样就保证了不会在同一时刻出现多个线程访问共享资源

***B、使用 CEvent 类***

　　CEvent 类提供了对事件的支持。事件是一个允许一个线程在某种情况发生时，唤醒另外一个线程的同步对象。例如在某些网络应用程序中，一个线程（记为A）负责监听通讯端口，另外一个线程（记为B）负责更新用户数据。通过使用CEvent 类，线程A可以通知线程B何时更新用户数据。每一个CEvent 对象可以有两种状态：有信号状态和无信号状态。线程监视位于其中的CEvent 类对象的状态，并在相应的时候采取相应的操作。  
　　在MFC中，CEvent 类对象有两种类型：人工事件和自动事件。一个自动CEvent 对象在被至少一个线程释放后会自动返回到无信号状态；而人工事件对象获得信号后，释放可利用线程，但直到调用成员函数ReSetEvent()才将其设置为无信号状态。在创建CEvent 类的对象时，默认创建的是自动事件。 CEvent 类的各成员函数的原型和参数说明如下：

1、CEvent(BOOL bInitiallyOwn=FALSE,

BOOL bManualReset=FALSE,

LPCTSTR lpszName=NULL,

LPSECURITY\_ATTRIBUTES lpsaAttribute=NULL);

* bInitiallyOwn:指定事件对象初始化状态，TRUE为有信号，FALSE为无信号；
* bManualReset：指定要创建的事件是属于人工事件还是自动事件。TRUE为人工事件，FALSE为自动事件；
* 后两个参数一般设为NULL，在此不作过多说明。

2、BOOL CEvent：：SetEvent();

　　将 CEvent 类对象的状态设置为有信号状态。如果事件是人工事件，则 CEvent 类对象保持为有信号状态，直到调用成员函数ResetEvent()将 其重新设为无信号状态时为止。如果CEvent 类对象为自动事件，则在SetEvent()将事件设置为有信号状态后，CEvent 类对象由系统自动重置为无信号状态。  
  
如果该函数执行成功，则返回非零值，否则返回零。

3、BOOL CEvent：：ResetEvent();

　　该函数将事件的状态设置为无信号状态，并保持该状态直至SetEvent()被调用时为止。由于自动事件是由系统自动重置，故自动事件不需要调用该函数。如果该函数执行成功，返回非零值，否则返回零。我们一般通过调用WaitForSingleObject函数来监视事件状态。

***C、使用CMutex 类***

　　互斥对象与临界区对象很像.互斥对象与临界区对象的不同在于:**互斥对象可以在进程间使用,而临界区对象只能在同一进程的各线程间使用。当然，互斥对象也可以用于同一进程的各个线程间，但是在这种情况下，使用临界区会更节省系统资源，更有效率。**

***D、使用CSemaphore 类***

　　当需要一个计数器来限制可以使用某个线程的数目时，可以使用“信号量”对象。CSemaphore 类的对象保存了对当前访问某一指定资源的线程的计数值，该计数值是当前还可以使用该资源的线程的数目。如果这个计数达到了零，则所有对这个CSemaphore 类对象所控制的资源的访问尝试都被放入到一个队列中等待，直到超时或计数值不为零时为止。一个线程被释放已访问了被保护的资源时，计数值减1；一个线程完成了对被控共享资源的访问时，计数值增1。这个被CSemaphore 类对象所控制的资源可以同时接受访问的最大线程数在该对象的构建函数中指定。  
  
CSemaphore 类的构造函数原型及参数说明如下：

CSemaphore (LONG lInitialCount=1,

LONG lMaxCount=1,

LPCTSTR pstrName=NULL,

LPSECURITY\_ATTRIBUTES lpsaAttributes=NULL);

* lInitialCount:信号量对象的初始计数值，即可访问线程数目的初始值；
* lMaxCount：信号量对象计数值的最大值，该参数决定了同一时刻可访问由信号量保护的资源的线程最大数目；
* 后两个参数在同一进程中使用一般为NULL，不作过多讨论；

　　在用CSemaphore 类的构造函数创建信号量对象时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，不能再允许其它线程的进入，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数加1。