

第8章 用户方式中线程的同步

当所有的线程在互相之间不需要进行通信的情况下就能够顺利地运行时，Microsoft Windows的运行性能最好。但是，线程很少能够在所有的时间都独立地进行操作。通常情况下，要生成一些线程来处理某个任务。当这个任务完成时，另一个线程必须了解这个情况。

系统中的所有线程都必须拥有对各种系统资源的访问权，这些资源包括内存堆栈，串口，文件，窗口和许多其他资源。如果一个线程需要独占对资源的访问权，那么其他线程就无法完成它们的工作。反过来说，也不能让任何一个线程在任何时间都能访问所有的资源。如果在一个线程从内存块中读取数据时，另一个线程却想要将数据写入同一个内存块，那么这就像你在读一本书时另一个人却在修改书中的内容一样。这样，书中的内容就会被搞得乱七八糟，结果什么也看不清楚。

线程需要在下面两种情况下互相进行通信：

- 当有多个线程访问共享资源而不使资源被破坏时。
- 当一个线程需要将某个任务已经完成的情况通知另外一个或多个线程时。

线程的同步包括许多方面的内容，下面几章将分别对它们进行介绍。值得高兴的是，Windows提供了许多方法，可以非常容易地实现线程的同步。但是，要想随时了解一连串的线程想要做什么，那是非常困难的。我们的头脑的工作不是异步的，我们希望以一种有序的方式来思考许多事情，每次前进一步。不过多线程环境不是这样运行的。

我是在大约1992年的时候开始从事多线程的编程工作的。最初，我犯过许多编程错误，在我出版的书籍和杂志文章中实际上都存在着与线程同步相关的错误。现在我的编程工作熟练了许多，但是并未做到完美无缺。希望本书中的全部内容不存在任何错误（尽管现在我知道我可以做得更好些）。要搞好线程的同步，唯一的办法是通过实践。下面几章将要介绍系统是如何运行的，并展示如何实现线程的正确同步，不过应该面对这样一个问题：即取得经验的同时，难免要犯错误。

8.1 原子访问：互锁的函数家族

线程同步问题在很大程度上与原子访问有关，所谓原子访问，是指线程在访问资源时能够确保所有其他线程都不在同一时间内访问相同的资源。让我们来看一看下面这个简单例子：

```
// Define a global variable.
long g_x = 0;

DWORD WINAPI ThreadFunc1(PVOID pvParam) {
    g_x++;
    return(0);
}

DWORD WINAPI ThreadFunc2(PVOID pvParam) {
    g_x++;
    return(0);
}
```

在这个代码中，声明了一个全局变量 `g_x`，并将它初始化为0。现在，假设创建两个线程，一个线程执行 `ThreadFunc1`，另一个线程执行 `ThreadFunc2`。这两个函数中的代码是相同的，它们都将1添加给全局变量 `g_x`。因此，当两个线程都停止运行时，你可能希望在 `g_x` 中看到2这个值。但是你真的看到了吗？回答是，也许看到了。根据代码的编写方法，你无法说明 `g_x` 中最终包含了什么东西。下面我们来说明为什么会出现这种情况。假设编译器生成了下面这行代码，以便将 `g_x` 递增1：

```
MOV EAX, [g_x]      ; Move the value in g_x into a register.
INC EAX             ; Increment the value in the register.
MOV [g_x], EAX      ; Store the new value back in g_x.
```

两个线程不可能在完全相同的时间内执行这个代码。因此，如果一个线程在另一个线程的后面执行这个代码，那么下面就是实际的执行情况：

```
MOV EAX, [g_x]      ; Thread 1: Move 0 into a register.
INC EAX             ; Thread 1: Increment the register to 1.
MOV [g_x], EAX      ; Thread 1: Store 1 back in g_x.

MOV EAX, [g_x]      ; Thread 2: Move 1 into a register.
INC EAX             ; Thread 2: Increment the register to 2.
MOV [g_x], EAX      ; Thread 2: Store 2 back in g_x.
```

当两个线程都将 `g_x` 的值递增之后，`g_x` 中的值就变成了2。这很好，并且正是我们希望的：即取出零（0），两次将它递增1，得出的值为2。太好了。不过不要急，Windows是个抢占式多线程环境。一个线程可以随时中断运行，而另一个线程则可以随时继续执行。这样，上面的代码就无法完全按编写的那样来运行。它可能按下面的形式运行：

```
MOV EAX, [g_x]      ; Thread 1: Move 0 into a register.
INC EAX             ; Thread 1: Increment the register to 1.

MOV EAX, [g_x]      ; Thread 2: Move 0 into a register.
INC EAX             ; Thread 2: Increment the register to 1.
MOV [g_x], EAX      ; Thread 2: Store 1 back in g_x.

MOV [g_x], EAX      ; Thread 1: Store 1 back in g_x.
```

如果代码按这种形式来运行，`g_x` 中的最后值就不是2，而是你预期的1。这使人感到非常担心，因为你对调度程序的控制能力非常小。实际上，如果有100个线程在执行相同的线程函数，当它们全部退出之后，`g_x` 中的值可能仍然是1。显然，软件开发人员无法在这种环境中工作。我们希望在所有情况下两次递增0产生的结果都是2。另外，不要忘记，编译器生成代码的方法，哪个CPU在执行这个代码，以及主计算机中安装了多少个CPU等因素，决定了产生的结果可能是不同的。这就是该环境的运行情况，我们对此无能为力。但是，Windows确实提供了一些函数，如果正确地使用这些函数，就能确保产生应用程序的代码得到的结果。

为了解决上面的问题，需要某种比较简单的方法。我们需要一种手段来保证值的递增能够以原子操作方式来进行，也就是不中断地进行。互锁的函数家族提供了我们需要的解决方案。互锁的函数尽管用处很大，而且很容易理解，却有些让人望而生畏，大多数软件开发人员用得很少。所有的函数都能以原子操作方式对一个值进行操作。让我们看一看下面这个 `InterlockedExchangeAdd` 函数：

```
LONG InterlockedExchangeAdd(
    PLONG p1Addend,
```

```
LONG lIncrement);
```

这是个最简单的函数了。只需调用这个函数，传递一个长变量地址，并指明将这个值递增多少即可。但是这个函数能够保证值的递增以原子操作方式来完成。因此可以将上面的代码重新编写为下面的形式：

```
// Define a global variable.
long g_x = 0;

DWORD WINAPI ThreadFunc1(PVOID pvParam) {
    InterlockedExchangeAdd(&g_x, 1);
    return(0);
}

DWORD WINAPI ThreadFunc2(PVOID pvParam) {
    InterlockedExchangeAdd(&g_x, 1);
    return(0);
}
```

通过这个小小的修改，g_x就能以原子操作方式来递增，因此可以确保g_x中的值最后是2。这样是不是感到好一些？注意，所有线程都应该设法通过调用这些函数来修改共享的长变量，任何线程都不应该通过调用简单的C语句来修改共享的变量：

```
// The long variable shared by many threads
LONG g_x;
:
:
// Incorrect way to increment the long
g_x++;
:
:
// Correct way to increment the long
InterlockedExchangeAdd(&g_x, 1);
```

互锁函数是如何运行的呢？答案取决于运行的是何种CPU平台。对于x86家族的CPU来说，互锁函数会对总线发出一个硬件信号，防止另一个CPU访问同一个内存地址。在Alpha平台上，互锁函数能够执行下列操作：

- 1) 打开CPU中的一个特殊的位标志，并注明被访问的内存地址。
- 2) 将内存的值读入一个寄存器。
- 3) 修改该寄存器。
- 4) 如果CPU中的特殊位标志是关闭的，则转入第二步。否则，特殊位标志仍然是打开的，寄存器的值重新存入内存。

你也许会问，执行第4步时CPU中的特殊位标志是如何关闭的呢？答案是：如果系统中的另一个CPU试图修改同一个内存地址，那么它能够关闭CPU的特殊位标志，从而导致互锁函数返回第二步。

不必清楚地了解互锁函数是如何工作的。重要的是要知道，无论编译器怎样生成代码，无论计算机中安装了多少个CPU，它们都能保证以原子操作方式来修改一个值。还必须保证传递给这些函数的变量地址正确地对齐，否则这些函数就会运行失败（第13章将介绍数据对齐问题）。

对于互锁函数，需要了解的另一个重要问题是，它们运行的速度极快。调用一个互锁函数

通常会导致执行几个CPU周期（通常小于50），并且不会从用户方式转换为内核方式（通常这需要执行1000个CPU周期）。

当然，可以使用InterlockedExchangeAdd减去一个值——只要为第二个参数传递一个负值。InterlockedExchangeAdd将返回在*plAddend中的原始值。

下面是另外两个互锁函数：

```
LONG InterlockedExchange(
    PLONG p1Target,
    LONG lValue);
```

```
PVOID InterlockedExchangePointer(
    PVOID* ppvTarget,
    PVOID pvValue);
```

InterlockedExchange和InterlockedExchangePointer能够以原子操作方式用第二个参数中传递的值来取代第一个参数中传递的当前值。如果是32位应用程序，两个函数都能用另一个32位值取代一个32位值。但是，如果是个64位应用程序，那么InterlockedExchange能够取代一个32位值，而InterlockedExchangePointer则取代64位值。两个函数都返回原始值。当实现一个循环锁时，InterlockedExchange是非常有用的：

```
// Global variable indicating whether a shared resource is in use or not
BOOL g_fResourceInUse = FALSE;
:
:
void Funcl() {
    // Wait to access the resource.
    while (InterlockedExchange (&g_fResourceInUse, TRUE) == TRUE)
        Sleep(0);

    // Access the resource.
    :
    :

    // We no longer need to access the resource.
    InterlockedExchange(&g_fResourceInUse, FALSE);
}
```

while循环是循环运行的，它将g_fResourceInUse中的值改为TRUE，并查看它的前一个值，以了解它是否是TRUE。如果这个值原先是FALSE，那么该资源并没有在使用，而是调用线程将它设置为在用状态并退出该循环。如果前一个值是TRUE，那么资源正在被另一个线程使用，while循环将继续循环运行。

如果另一个线程要执行类似的代码，它将在while循环中运行，直到g_fResourceInUse重新改为FALSE。调用函数结尾处的InterlockedExchange，可显示应该如何将g_fResourceInUse重新设置为FALSE。

当使用这个方法时必须格外小心，因为循环锁会浪费CPU时间。CPU必须不断地比较两个值，直到一个值由于另一个线程而“奇妙地”改变为止。另外，该代码假定使用循环锁的所有线程都以相同的优先级等级运行。也可以把执行循环锁的线程的优先级提高功能禁用（通过调用SetProcessPriorityBoost或setThreadPriorityBoost函数来实现之）。

此外，应该保证将循环锁变量和循环锁保护的数据维护在不同的高速缓存行中（本章后面部分介绍）。如果循环锁变量与数据共享相同的高速缓存行，那么使用该资源的CPU将与试图访问该资源的任何CPU争用高速缓存行。

应该避免在单个CPU计算机上使用循环锁。如果一个线程正在循环运行，它就会浪费前一个CPU时间，这将防止另一个线程修改该值。我在上面的 while 循环中使用了 Sleep，从而在某种程度上解决了浪费 CPU 时间的问题。如果使用 Sleep，你可能想睡眠一个随机时间量；每次请求访问该资源均被拒绝时，你可能想进一步延长睡眠时间。这可以防止线程浪费 CPU 时间。根据情况，最好是全部删除对 Sleep 的调用。或者使用对 SwitchToThread（Windows 98 中没有这个函数）的调用来取代它。勇于试验和不断纠正错误，是学习的最好方法。

循环锁假定，受保护的资源总是被访问较短的时间。这使它更加有效地循环运行，然后转为内核方式并进入等待状态。许多编程人员循环运行一定的次数（比如 400 次），如果对资源的访问仍然被拒绝，那么该线程就转为内核方式，在这种方式下，它要等待（不消耗 CPU 时间），直到该资源变为可供使用为止。这就是关键部分实现的方法。

循环锁在多处理器计算机上非常有用，因为当一个线程循环运行的时候，另一个线程可以在另一个 CPU 上运行。但是，即使在这种情况下，也必须小心。不应该让线程循环运行太长的时间，也不能浪费更多的 CPU 时间。本章后面将进一步介绍循环锁。第 10 章将介绍如何使用循环锁。

下面是最后两个互锁函数：

```
PVOID InterlockedCompareExchange(  
    PLONG pDestination,  
    LONG lExchange,  
    LONG lComparand);  
  
PVOID InterlockedCompareExchangePointer(  
    PVOID* ppvDestination,  
    PVOID pvExchange,  
    PVOID pvComparand);
```

这两个函数负责执行一个原子测试和设置操作。如果是 32 位应用程序，那么两个函数都在 32 位值上运行，但是，如果是 64 位应用程序，InterlockedCompareExchange 函数在 32 位值上运行，而 InterlockedCompareExchangePointer 函数则在 64 位值上运行。在伪代码中，它的运行情况如下面所示：

```
LONG InterlockedCompareExchange(PLONG pDestination,  
    LONG lExchange, LONG lComparand) {  
  
    LONG lRet = *pDestination; // Original value  
  
    if (*pDestination == lComparand)  
        *pDestination = lExchange;  
    return(lRet);  
}
```

该函数对当前值（pDestination 参数指向的值）与 lComparand 参数中传递的值进行比较。如果两个值相同，那么 *pDestination 改为 lExchange 参数的值。如果 *pDestination 中的值与 lExchange 的值不匹配，*pDestination 保持不变。该函数返回 *pDestination 中的原始值。记住，所有这些操作都是作为一个原子执行单位来进行的。

没有任何互锁函数仅仅负责对值进行读取操作（而不改变这个值），因为这样的函数根本是不需要的。如果线程只是试图读取值的内容，而这个值始终都由互锁函数来修改，那么被读取的值总是一个很好的值。虽然你不知道你读取的是原始值还是更新值，但是你知道它是这两个值中的一个。对于大多数应用程序来说，这一点很重要。此外，当要对共享内存区域（比如

内存映象文件)中的值的访问进行同步时,互锁函数也可以供多进程中的线程使用(第9章中包含了几个示例应用程序,以显示如何正确地使用互锁函数)。

虽然Windows还提供了另外几个互锁函数,但是上面介绍的这些函数能够实现其他函数能做的一切功能,甚至更多。下面是两个其他的函数:

```
LONG InterlockedIncrement(PULONG p1Addend);
```

```
LONG InterlockedDecrement(PULONG p1Addend);
```

InterlockedExchangeAdd函数能够取代这些较老的函数。新函数能够递增或递减任何值,老的函数只能加1或减1。

8.2 高速缓存行

如果想创建一个能够在多处理器计算机上运行的高性能应用程序,必须懂得CPU的高速缓存行。当一个CPU从内存读取一个字节时,它不只是取出一个字节,它要取出足够的字节来填入高速缓存行。高速缓存行由32或64个字节组成(视CPU而定),并且始终在第32个字节或第64个字节的边界上对齐。高速缓存行的作用是为了提高CPU运行的性能。通常情况下,应用程序只能对一组相邻的字节进行处理。如果这些字节在高速缓存中,那么CPU就不必访问内存总线,而访问内存总线需要多得多的时间。

但是,在多处理器环境中,高速缓存行使得内存的更新更加困难,下面这个例子就说明了这一点:

- 1) CPU1读取一个字节,使该字节和它的相邻字节被读入CPU1的高速缓存行。
- 2) CPU2读取同一个字节,使得第一步中的相同的各个字节读入CPU2的高速缓存行。
- 3) CPU1修改内存中的该字节,使得该字节被写入CPU1的高速缓存行。但是该信息尚未写入RAM。
- 4) CPU2再次读取同一个字节。由于该字节已经放入CPU2的高速缓存行,因此它不必访问内存。但是CPU2将看不到内存中该字节的新值。

这种情况会造成严重的后果。当然,芯片设计者非常清楚这个问题,并且设计它们的芯片来处理这个问题。尤其是,当一个CPU修改高速缓存行中的字节时,计算机中的其他CPU会被告知这个情况,它们的高速缓存行将变为无效。因此,在上面的情况下,CPU2的高速缓存在CPU1修改字节的值时变为无效。在第4步中,CPU1必须将它的高速缓存内容迅速转入内存,CPU2必须再次访问内存,重新将数据填入它的高速缓存行。如你所见,高速缓存行能够帮助提高运行的速度,但是它们也可能是多处理器计算机上的一个不利因素。

这一切意味着你应该将高速缓存行存储块中的和高速缓存行边界上的应用程序数据组合在一起。这样做的目的是确保不同的CPU能够访问至少由高速缓存行边界分开的不同的内存地址。还有,应该将只读数据(或不常读的数据)与读写数据分开。同时,应该将同一时间访问的数据组合在一起。

下面是设计得很差的数据结构的例子:

```
struct CUSTINFO {  
    DWORD    dwCustomerID;    // Mostly read-only  
    int      nBalanceDue;     // Read-write  
    char     szName[100];     // Mostly read-only  
    FILETIME ftLastOrderDate; // Read-write  
};
```

下面是该结构的改进版本:

```
// Determine the cache line size for the host CPU.
#ifdef _X86_
#define CACHE_ALIGN 32
#endif
#ifdef _ALPHA_
#define CACHE_ALIGN 64
#endif
#ifdef _IA64_
#define CACHE_ALIGN ??
#endif

#define CACHE_PAD(Name, BytesSoFar) \
    BYTE Name[CACHE_ALIGN - ((BytesSoFar) % CACHE_ALIGN)]

struct CUSTINFO {
    DWORD    dwCustomerID;    // Mostly read-only
    char     szName[100];    // Mostly read-only

    // Force the following members to be in a different cache line.
    CACHE_PAD(bPad1, sizeof(DWORD) + 100);

    int      nBalanceDue;    // Read-write
    FILETIME ftLastOrderDate; // Read-write

    // Force the following structure to be in a different cache line.
    CACHE_PAD(bPad2, sizeof(int) + sizeof(FILETIME));
};
```

上面定义的CACHE_ALIGN宏是不错的，但是并不很好。问题是必须手工将每个成员变量的字节值输入该宏。如果增加、移动或删除数据成员，也必须更新对CACHE_PAD宏的调用。将来，Microsoft的C/C++编译器将支持一种新句法，该句法可以更容易地调整数据成员。它的形式类似__declspec(align(32))。

注意 最好是始终都让单个线程来访问数据（函数参数和局部变量是确保做到这一点的最好方法），或者始终让单个CPU访问这些数据（使用线程亲缘性）。如果采取其中的一种方法，就能够完全避免高速缓存的各种问题。

8.3 高级线程同步

当必须以原子操作方式来修改单个值时，互锁函数家族是相当有用的。你肯定应该先试试它们。但是大多数实际工作中的编程问题要解决的是比单个32位或64位值复杂得多的数据结构。为了以原子操作方式使用更加复杂的数据结构，必须将互锁函数放在一边，使用Windows提供的其他某些特性。

前面强调了不应该在单处理器计算机上使用循环锁，甚至在多处理器计算机上，也应该小心地使用它们。原因是CPU时间非常宝贵，决不应该浪费。因此需要一种机制，使线程在等待访问共享资源时不浪费CPU时间。

当线程想要访问共享资源，或者得到关于某个“特殊事件”的通知时，该线程必须调用一个操作系统函数，给它传递一些参数，以指明该线程正在等待什么。如果操作系统发现资源可供使用，或者该特殊事件已经发生，那么函数就返回，同时该线程保持可调度状态（该线程可以不必立即执行，它处于可调度状态，可以使用前一章介绍的原则将它分配给一个CPU）。

如果资源不能使用，或者特殊事件还没有发生，那么系统便使该线程处于等待状态，使该线程无法调度。这可以防止线程浪费 CPU 时间。当线程处于等待状态时，系统作为一个代理，代表你的线程来执行操作。系统能够记住你的线程需要什么，当资源可供使用的时候，便自动使该线程退出等待状态，该线程的运行将与特殊事件实现同步。

从实际情况来看，大多数线程几乎总是处于等待状态。当系统发现所有线程有若干分钟均处于等待状态时，系统的强大的管理功能就会发挥作用。

要避免使用的一种方法

如果没有同步对象，并且操作系统不能发现各种特殊事件，那么线程就不得不使用下面要介绍的一种方法使自己与特殊事件保持同步。不过，由于操作系统具有支持线程同步的内置特性，因此决不应该使用这种方法。

运用这种方法时，一个线程能够自己与另一个线程中的任务的完成实现同步，方法是不断查询多个线程共享或可以访问的变量的状态。下面的代码段说明了这个情况：

```
volatile BOOL g_fFinishedCalculation = FALSE;

int WINAPI WinMain(...) {
    CreateThread(..., RecalcFunc, ...);
    :
    // Wait for the recalculation to complete.
    while (!g_fFinishedCalculation)
        ;
    :
}

DWORD WINAPI RecalcFunc(PVOID pvParam) {
    // Perform the recalculation.
    :
    g_fFinishedCalculation = TRUE;
    return(0);
}
```

如你所见，当主线程（执行 WinMain）必须使自己与 RecalcFunc 函数的完成运行实现同步时，它并没有使自己进入睡眠状态。由于主线程没有进入睡眠状态，因此操作系统继续为它调度 CPU 时间，这就要占用其他线程的宝贵时间周期。

前面代码段中使用的查询方法存在的另一个问题是，BOOL 变量 g_f FinishedCalculation 从来没有被设置为 TRUE。当主线程的优先级高于执行 RecalcFunc 函数的线程时，就会发生这种情况。在这种情况下，系统决不会将任何时间片分配给 RecalcFunc 线程。如果执行 WinMain 函数的线程被置于睡眠状态，而不是进行查询，那么这就不是已调度的时间。系统可以将时间调度给低优先级的线程，如 RecalcFunc 线程，使它们得以运行。

应该说，有时查询迟早都可以进行，毕竟是循环锁执行的操作。不过有些方法进行这项操作是恰当的，而有些方法是不恰当的。一般来说，应该调用一些函数，使线程进入睡眠状态，直到线程需要的资源可供使用为止。下一节将介绍一种正确的方法。

首先，在前面介绍的代码段的开头，你会发现它使用了 volatile 一词。为了使这个代码段更接近工作状态，必须有一个 volatile 类型的限定词。它告诉编译器，变量可以被应用程序本身以外的某个东西进行修改，这些东西包括操作系统，硬件或同时执行的线程等。尤其是，

volatile限定词会告诉编译器，不要对该变量进行任何优化，并且总是重新加载来自该变量的内存单元的值。比如，编译器为前面的代码段中的while语句生成了下面的伪代码：

```
MOV    Reg0, [g_fFinishedCalculation] ; Copy the value into a register
Label: TEST  Reg0, 0                  ; Is the value 0?
JMP     Reg0 == 0, Label               ; The register is 0, try again
...                                     ; The register is not 0 (end of loop)
```

如果不使布尔变量具备易变性，编译器就能像上面所示的那样优化你的C代码。为了实现这样的优化，编译器只需将BOOL变量的值装入一个CPU寄存器一次。然后，它对该CPU寄存器反复进行测试。这样得出的性能当然要比不断地重复读取内存地址中的值并对它进行重复测试要好，因此，优化编译器能够编写上面所示的那种代码。但是，如果编译器进行这样的操作，线程就会进入一个无限循环，永远无法唤醒。另外，使一个结构具备易变性，可以确保它的所有成员都具有易变性，当它们被引用时，总是可以从内存中读取它们。

你也许会问，循环变量g_fResourceInUse是否应该声明为volatile变量。答案是不必，因为我们将该变量的地址传递给各个不同的互锁函数，而不是传递变量值本身。当将一个变量地址传递给一个函数时，该函数必须从内存读取该值。优化程序不会对它产生任何影响。

8.4 关键代码段

关键代码段是指一个小代码段，在代码能够执行前，它必须独占对某些共享资源的访问权。这是让若干行代码能够“以原子操作方式”来使用资源的一种方法。所谓原子操作方式，是指该代码知道没有别的线程要访问该资源。当然，系统仍然能够抑制你的线程的运行，而抢先安排其他线程的运行。不过，在线程退出关键代码段之前，系统将不给想要访问相同资源的其他任何线程进行调度。

下面是个有问题的代码，它显示了不使用关键代码段会发生什么情况：

```
const int MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];

DWORD WINAPI FirstThread(PVOID pvParam) {

    while (g_nIndex < MAX_TIMES) {
        g_dwTimes[g_nIndex] = GetTickCount();
        g_nIndex++;
    }
    return(0);
}

DWORD WINAPI SecondThread(PVOID pvParam) {

    while (g_nIndex < MAX_TIMES) {
        g_nIndex++;
        g_dwTimes[g_nIndex - 1] = GetTickCount();
    }
    return(0);
}
```

如果分开来看，这两个线程函数将会产生相同的结果，不过每个函数的编码略有不同。如果FirstThread函数自行运行，它将用递增的值填入g_dwTimes数组。如果SecondThread函数也

是自行运行,那么情况也一样。在理想的情况下,我们希望两个线程能够同时运行,并且仍然使g_dwTimes数组能够产生递增的值。但是,上面的代码存在一个问题,那就是 g_dwTimes不会被正确地填入数据,因为两个线程函数要同时访问相同的全局变量。

下面是如何出现这种情况的一个例子。比如说,我们刚刚在只有一个CPU的系统上启动执行两个线程。操作系统首先启动运行 SecondThread(这种情况很可能出现),当SecondThread将g_nIndex递增为1之后,系统就停止该线程的运行,而让FirstThread运行。这时FirstThread将g_dwTimes[1]设置为系统时间,然后系统停止该线程的运行,将CPU时间重新赋予SecondThread线程。然后SecondThread将g_dwTimes[1-1]设置为新的系统时间。由于这个操作发生在较晚的时间,因此新系统时间的值大于放入FirstThread数组中的时间值,另外要注意,g_dwTimes的索引1填在索引0的前面。数组中的数据被破坏了。

应该说明的是,这个例子的设计带有一定的故意性,因为要设计一个实际工作中的例子而不使用好几页的源代码是很难的。不过,通过这个例子,能够看到这个问题在实际工作中有些什么表现。考虑一下管理一个链接对象列表的情况。如果对该链接列表的访问没有取得同步,那么一个线程可以将一个项目添加给这个列表,而另一个线程则试图搜索该列表中的一个项目。如果两个线程同时给这个列表添加项目,那么这种情况会变得更加复杂。通过运用关键代码段,就能够确保在各个线程之间协调对数据结构的访问。

既然已经了解了存在的所有问题,那么下面让我们用关键代码段来修正这个代码:

```
const int MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];
CRITICAL_SECTION g_cs;

DWORD WINAPI FirstThread(PVOID pvParam) {
    while (g_nIndex < MAX_TIMES) {
        EnterCriticalSection(&g_cs);
        g_dwTimes[g_nIndex] = GetTickCount();
        g_nIndex++;
        LeaveCriticalSection(&g_cs);
    }
    return(0);
}

DWORD WINAPI SecondThread(PVOID pvParam) {
    while (g_nIndex < MAX_TIMES) {
        EnterCriticalSection(&g_cs);
        g_nIndex++;
        g_dwTimes[g_nIndex - 1] = GetTickCount();
        LeaveCriticalSection(&g_cs);
    }
    return(0);
}
```

这里指定了一个CRITICAL_SECTION数据结构g_cs,然后在EnterCriticalSection和LeaveCriticalSection函数调用中封装了要接触共享资源(在这个例子中为g_nIndex和g_dwTimes)的任何代码。注意,在对EnterCriticalSection和LeaveCriticalSection的所有调用中,我传递了g_cs的地址。

有一个关键问题必须记住。当拥有一项可供多个线程访问的资源时，应该创建一个 CRITICAL_SECTION 结构。由于我是在飞行旅途上编写这个代码的，让我描绘下面这个模拟情况。CRITICAL_SECTION 就像飞机上的厕所，抽水马桶是你要保护的数据。由于厕所很小，每次只能一个人（线程）进入厕所（关键代码段）使用抽水马桶（受保护的资源）。

如果有多个资源总是被一道使用，可以将它们全部放在一个厕所里，也就是说可以创建一个 CRITICAL_SECTION 结构来保护所有的资源。

如果有多个不是一道使用的资源，比如线程 1 和线程 2 访问一个资源，而线程 1 和线程 3 访问另一个资源，那么应该为每个资源创建一个独立的厕所，即 CRITICAL_SECTION 结构。

现在，无论在何处拥有需要访问资源的代码，都必须调用 EnterCriticalSection 函数，为它传递用于标识该资源的 CRITICAL_SECTION 结构的地址。这就是说，当一个线程需要访问一个资源时，它首先必须检查厕所门上的“有人”标志。CRITICAL_SECTION 结构用于标识线程想要进入哪个厕所，而 EnterCriticalSection 函数则是线程用来检查“有人”标志的函数。

如果 EnterCriticalSection 函数发现厕所中没有任何别的线程（门上的标志显示“无人”），那么调用线程就可以使用该资源。如果 EnterCriticalSection 发现厕所中有另一个线程正在使用，那么调用函数必须在厕所门的外面等待，直到厕所中的另一个线程离开厕所。

当一个线程不再执行需要访问资源的代码时，它应该调用 LeaveCriticalSection 函数。这样，它就告诉系统，它准备离开包含该资源的厕所。如果忘记调用 LeaveCriticalSection，系统将认为该线程仍然在厕所中，因此不允许其他正在等待的线程进入厕所。这就像离开了厕所但没有换上“无人”的标志。

注意 最难忘住的一件事情是，编写的需要使用共享资源的任何代码都必须封装在 EnterCriticalSection 和 LeaveCriticalSection 函数中。如果忘记将代码封装在一个位置，共享资源就可能遭到破坏。例如，如果我删除了 FirstThread 线程对 EnterCriticalSection 和 LeaveCriticalSection 的调用，g_nIndex 和 g_dwTimes 变量就会遭到破坏。即使 SecondThread 线程仍然正确地调用 EnterCriticalSection 和 LeaveCriticalSection，也会出现这种情况。

忘记调用 EnterCriticalSection 和 LeaveCriticalSection 函数就像是不请求允许进入厕所。线程只是想努力挤入厕所并对资源进行操作。可以想象，只要有一个线程表现出这种相当粗暴的行为，资源就会遭到破坏。

当无法用互锁函数来解决同步问题时，你应该试用关键代码段。关键代码段的优点在于它们的使用非常容易，它们在内部使用互锁函数，这样它们就能够迅速运行。关键代码的主要缺点是无法用它们对多个进程中的各个线程进行同步。不过在第 19 章中，我将要创建我自己的同步对象，称为 Optex。这个对象将显示操作系统如何实现关键代码段，它也能用于多个进程中的各个线程。

8.4.1 关键代码段准确的描述

现在你已经从理论上对关键代码段有了一定的了解。已经知道为什么它们非常有用，以及它们是如何实现“以原子操作方式”对共享资源进行访问的。下面让我们更加深入地看一看关键代码段是如何运行的。首先介绍一下 CRITICAL_SECTION 数据结构。如果想查看一下 Platform SDK 文档中关于该结构的说明，也许你会感到无从下手。那么问题究竟何在呢？

并不是 CRITICAL_SECTION 结构没有完整的文档，而是 Microsoft 认为没有必要了解该结

构的全部情况，这是对的。对于我们来说，这个结构是透明的，该结构有文档可查，但是该结构中的成员变量没有文档。当然，由于这只是个数据结构，可以在 Windows 头文件中查找这些信息，可以看到这些数据成员（CRITICAL_SECTION 在 WinNT.h 中定义为 RTL_CRITICAL_SECTION；RTL_CRITICAL_SECTION 结构在 WinBase.h 中作了定义）。但是决不应该编写引用这些成员的代码。

若要使用 CRITICAL_SECTION 结构，可以调用一个 Windows 函数，给它传递该结构的地址。该函数知道如何对该结构的成员进行操作，并保证该结构的状态始终一致。因此下面让我们将注意力转到这些函数上去。

通常情况下，CRITICAL_SECTION 结构可以作为全局变量来分配，这样，进程中的所有线程就能够很容易地按照变量名来引用该结构。但是，CRITICAL_SECTION 结构也可以作为局部变量来分配，或者从堆栈动态地进行分配。它只有两个要求，第一个要求是，需要访问该资源的所有线程都必须知道负责保护资源的 CRITICAL_SECTION 结构的地址，你可以使用你喜欢的任何机制来获得这些线程的这个地址；第二个要求是，CRITICAL_SECTION 结构中的成员应该在任何线程试图访问被保护的资源之前初始化。该结构通过调用下面的函数来进行初始化：

```
VOID InitializeCriticalSection(PCRITICAL_SECTION pcs);
```

该函数用于对（pcs 指向的）CRITICAL_SECTION 结构的各个成员进行初始化。由于该函数只是设置了某些成员变量。因此它的运行不会失败，并且它的原型采用了 VOID 的返回值。该函数必须在任何线程调用 EnterCriticalSection 函数之前被调用。Platform SDK 的文档清楚地说明，如果一个线程试图进入一个未初始化的 CRITICAL_SECTION，那么结果将是很难预计的。

当知道进程的线程不再试图访问共享资源时，应该通过调用下面的函数来清除该 CRITICAL_SECTION 结构：

```
VOID DeleteCriticalSection(PCRITICAL_SECTION pcs);
```

DeleteCriticalSection 函数用于对该结构中的成员变量进行删除。当然，如果有任何线程仍然使用关键代码段，那么不应该删除该代码段。同样，Platform SDK 文档清楚地说明如果删除了关键代码段，其结果就无法知道。当编写要使用共享资源的代码时，必须在该代码的前面放置对下面的函数的调用：

```
VOID EnterCriticalSection(PCRITICAL_SECTION pcs);
```

EnterCriticalSection 函数负责查看该结构中的成员变量。这些变量用于指明当前是哪个变量正在访问该资源。EnterCriticalSection 负责进行下列测试：

- 如果没有线程访问该资源，EnterCriticalSection 便更新成员变量，以指明调用线程已被赋予访问权并立即返回，使该线程能够继续运行（访问该资源）。
- 如果成员变量指明，调用线程已经被赋予对资源的访问权，那么 EnterCriticalSection 便更新这些变量，以指明调用线程多少次被赋予访问权并立即返回，使该线程能够继续运行。这种情况很少出现，并且只有当线程在一行中两次调用 EnterCriticalSection 而不影响对 LeaveCriticalSection 的调用时，才会出现这种情况。
- 如果成员变量指明，一个线程（除了调用线程之外）已被赋予对资源的访问权，那么 EnterCriticalSection 将调用线程置于等待状态。这种情况是极好的，因为等待的线程不会浪费任何 CPU 时间。系统能够记住该线程想要访问该资源并且自动更新 CRITICAL_SECTION 的成员变量，一旦目前访问该资源的线程调用 LeaveCriticalSection

函数，该线程就处于可调度状态。

从内部来讲，EnterCriticalSection函数并不十分复杂。它只是执行一些简单的测试。为什么这个函数是如此有用呢？因为它能够以原子操作方式来执行所有的测试。如果在多处理器计算机上有两个线程在完全相同的时间同时调用 EnterCriticalSection函数，该函数仍然能够正确地起作用，一个线程被赋予对资源的访问权，而另一个线程则进入等待状态。

如果EnterCriticalSection将一个线程置于等待状态，那么该线程在很长时间内就不能再次被调度。实际上，在编写得不好的应用程序中，该线程永远不会再次被赋予 CPU时间。如果出现这种情况，该线程就称为渴求CPU时间的线程。

Windows 2000 在实际操作中，等待关键代码段的线程绝对不会渴求 CPU时间。对 EnterCriticalSection的调用最终将会超时，导致产生一个异常条件。这时可以将一个调试程序附加给应用程序，以确定究竟出了什么问题。超时的时间量是由下面的注册表子关键字中包含的 CriticalSectionTimeout数据值来决定的。

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager
```

这个值以秒为单位，默认为 2 592 000s，即大约30天。不要将这个值设置得太小（比如小于 3s），否则就会对系统中正常等待关键代码段超过 3s的线程和其他应用程序产生不利的影响。

可以使用下面这个函数来代替 EnterCriticalSection：

```
BOOL TryEnterCriticalSection(PCRITICAL_SECTION pcs);
```

TryEnterCriticalSection函数决不允许调用线程进入等待状态。相反，它的返回值能够指明调用线程是否能够获得对资源的访问权。因此，如果 TryEnterCriticalSection发现该资源已经被另一个线程访问，它就返回 FALSE。在其他所有情况下，它均返回 TRUE。

运用这个函数，线程能够迅速查看它是否可以访问某个共享资源，如果不能访问，那么它可以继续执行某些其他操作，而不必进行等待。如果 TryEnterCriticalSection函数确实返回了 TRUE，那么 CRITICAL_SECTION的成员变量已经更新，以便反映出该线程正在访问该资源。因此，对返回 TRUE的 TryEnterCriticalSection函数的每次调用都必须与对 LeaveCriticalSection函数的调用相匹配。

Windows 98 Windows 98没有可以使用的 TryEnterCriticalSection函数的实现代码。调用该函数总是返回 FALSE。

在接触共享资源的代码结尾处，必须调用下面这个函数：

```
VOID LeaveCriticalSection(PCRITICAL_SECTION pcs);
```

LeaveCriticalSection要查看该结构中的成员变量。该函数每次计数时要递减 1，以指明调用线程多少次被赋予对共享资源的访问权。如果该计数大于 0，那么 LeaveCriticalSection不做其他任何操作，只是返回而已。

如果该计数变为 0，它就要查看在调用 EnterCriticalSection中是否有别的线程正在等待。如果至少有一个线程正在等待，它就更新成员变量，并使等待线程中的一个线程（“公正地”选定）再次处于可调度状态。如果没有线程正在等待，LeaveCriticalSection函数就更新成员变量，以指明没有线程正在访问该资源。

与 EnterCriticalSection函数一样，LeaveCriticalSection函数也能以原子操作方式执行所有这些测试和更新。不过，LeaveCriticalSection从来不使线程进入等待状态，它总是立即返回。

8.4.2 关键代码段与循环锁

当线程试图进入另一个线程拥有的关键代码段时，调用线程就立即被置于等待状态。这意味着该线程必须从用户方式转入内核方式（大约 1000个CPU周期）。这种转换是要付出很大代价的。在多处理器计算机上，当前拥有资源的线程可以在不同的处理器上运行，并且能够很快放弃对资源的控制。实际上拥有资源的线程可以在另一个线程完成转入内核方式之前释放资源。如果出现这种情况，就会浪费许多CPU时间。

为了提高关键代码段的运行性能，Microsoft将循环锁纳入了这些代码段。因此，当EnterCriticalSection函数被调用时，它就使用循环锁进行循环，以便设法多次取得该资源。只有当为了取得该资源的每次试图都失败时，该线程才转入内核方式，以便进入等待状态。

若要将循环锁用于关键代码段，应该调用下面的函数，以便对关键代码段进行初始化：

```
BOOL InitializeCriticalSectionAndSpinCount(  
    PCRITICAL_SECTION pcs,  
    DWORD dwSpinCount);
```

与InitializeCriticalSection中的情况一样，InitializeCriticalSectionAndSpinCount的第一个参数是关键代码段结构的地址。但是在第二个参数 dwSpinCount中，传递的是在使线程等待之前它试图获得资源时想要循环锁循环迭代的次数。这个值可以是0至0x00FFFFFF之间的任何数字。如果在单处理器计算机上运行时调用该函数，dwSpinCount参数将被忽略，它的计数始终被置为0。这是对的，因为在单处理器计算机上设置循环次数是毫无用处的，如果另一个线程正在循环运行，那么拥有资源的线程就不能放弃它。

通过调用下面的函数，就能改变关键代码段的循环次数：

```
DWORD SetCriticalSectionSpinCount(  
    PCRITICAL_SECTION pcs,  
    DWORD dwSpinCount);
```

同样，如果主计算机只有一个处理器，那么 dwSpinCount的值将被忽略。我认为，始终都应该将循环锁用于关键代码段，因为这样做有百利而无一害。难就难在确定为 dwSpinCount参数传递什么值。为了实现最佳的性能，只需要调整这些数字，直到对性能结果满意为止。作为一个指导原则，保护对进程的堆栈进行访问的关键代码段使用的循环次数是 4000次。

第10章将要介绍如何实现关键代码段。这种实现将包括循环锁。

8.4.3 关键代码段与错误处理

InitializeCriticalSection函数的运行可能失败（尽管可能性很小）。Microsoft在最初设计该函数时并没有真正想到这个问题，正因为这个原因，该函数的原型才设计为返回 VOID。该函数的运行可能失败，因为它分配了一个内存块以便系统得到一些内部调试信息。如果该内存的分配失败，就会出现一个STATUS_NO_MEMORY异常情况。可以使用结构化异常处理（第23、24和25章介绍）来跟踪代码中的这种异常情况。

使用更新的InitializeCriticalSectionAndSpinCount函数，就能够更加容易地跟踪这个问题。该函数也为调试信息分配了内存块，如果内存无法分配，那么它就返回 FALSE。

当使用关键代码段时还会出现另一个问题。从内部来说，如果两个或多个线程同时争用关键代码段，那么关键代码段将使用一个事件内核对象（第10章介绍Coptex C++类时，我将要说明如何使用该内核对象）。由于争用的情况很少发生，因此，在初次需要之前，系统将不创建事件内核对象。这可以节省大量的系统资源，因为大多数关键代码段从来不被争用。

在内存不足的情况下，关键代码段可能被争用，同时系统可能无法创建必要的事件内核对象。这时EnterCriticalSection函数将会产生一个EXCEPTION_INVALID_HANDLE异常。大多数编程人员忽略了这个潜在的错误，在他们的代码中没有专门的处理方法，因为这个错误非常少见。但是，如果想对这种情况有所准备，可以有两种选择。

可以使用结构化异常处理方法来跟踪错误。当错误发生时，既可以访问关键代码段保护的资源，也可以等待某些内存变成可用状态，然后再次调用EnterCriticalSection函数。

另一种选择是使用InitializeCriticalSectionAndSpinCount函数创建关键代码段，确保设置了dwSpinCount参数的高信息位。当该函数发现高信息位已经设置时，它就创建该事件内核对象，并在初始化时将它与关键代码段关联起来。如果事件无法创建，该函数返回FALSE。可以更加妥善地处理代码中的这个事件。如果事件创建成功，你知道EnterCriticalSection将始终都能运行，并且决不会产生异常情况（如果总是预先分配事件内核对象，就会浪费系统资源。只有当你的代码不能容许EnterCriticalSection运行失败，或者你有把握会出现争用现象，或者你预计进程将在内存非常短缺的环境中运行时，你才能预先分配事件内核对象）。

8.4.4 非常有用的提示和技巧

当使用关键代码段时，有些很好的方法可以使用，而有些方法则应该避免。下面是在使用关键代码段时对你有所帮助的一些提示和技巧。这些技巧也适用于内核对象的同步（下一章介绍）。

1. 每个共享资源使用一个CRITICAL_SECTION变量

如果应用程序中拥有若干个互不相干的数据结构，应该为每个数据结构创建一个CRITICAL_SECTION变量。这比只有单个CRITICAL_SECTION结构来保护对所有共享资源的访问要好，请观察下面这个代码段：

```
int g_nNums[100];           // A shared resource
TCHAR g_cChars[100];        // Another shared resource
CRITICAL_SECTION g_cs;      // Guards both resources
```

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {

    EnterCriticalSection(&g_cs);

    for (int x = 0; x < 100; x++) {
        g_nNums[x] = 0;
        g_cChars[x] = TEXT('X');
    }

    LeaveCriticalSection(&g_cs);
    return(0);

}
```

这个代码使用单个关键代码段，以便在g_nNums数组和g_cChars数组初始化时对它们同时实施保护。但是，这两个数组之间毫无关系。当这个循环运行时，没有一个线程能够访问任何一个数组。如果ThreadFunc函数按下面的形式来实现，那么两个数组将分别被初始化：

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {
```

```

EnterCriticalSection(&g_cs);

for (int x = 0; x < 100; x++)
    g_nNums[x] = 0;

for (x = 0; x < 100; x++)
    g_cChars[x] = TEXT('X');

LeaveCriticalSection(&g_cs);
return(0);
}

```

从理论上讲，当 `g_nNums` 数组初始化后，另一个只需要访问 `g_nNums` 数组而不需要访问 `g_cChars` 数组的线程就可以开始执行，同时 `ThreadFunc` 可以继续对 `g_cChars` 数组进行初始化。但是实际上这是不可能的，因为有一个关键代码段保护着这两个数据结构。为了解决这个问题，可以创建下面两个关键代码段：

```

int g_nNum[100];           // A shared resource
CRITICAL_SECTION g_csNums; // Guards g_nNums

TCHAR g_cChars[100];       // Another shared resource
CRITICAL_SECTION g_csChars; // Guards g_cChars

DWORD WINAPI ThreadFunc(PVOID pvParam) {

    EnterCriticalSection(&g_csNums);

    for (int x = 0; x < 100; x++)
        g_nNums[x] = 0;

    LeaveCriticalSection(&g_csNums);

    EnterCriticalSection(&g_csChars);

    for (x = 0; x < 100; x++)
        g_cChars[x] = TEXT('X');

    LeaveCriticalSection(&g_csChars);
    return(0);
}

```

运用这个实现代码，一旦 `ThreadFunc` 完成对 `g_nNums` 数组的初始化，另一个线程就可以开始使用 `g_nNums` 数组。也可以考虑让一个线程对 `g_nNums` 数组进行初始化，而另一个线程函数对 `g_cChars` 数组进行初始化。

2. 同时访问多个资源

有时需要同时访问两个资源。如果这是 `ThreadFunc` 的要求，可以用下面的代码来实现：

```

DWORD WINAPI ThreadFunc(PVOID pvParam) {

    EnterCriticalSection(&g_csNums);
    EnterCriticalSection(&g_csChars);

    // This loop requires simultaneous access to both resources.
    for (int x = 0; x < 100; x++)
        g_nNums[x] = g_cChars[x];

    LeaveCriticalSection(&g_csChars);
}

```

```

    LeaveCriticalSection(&g_csNums);
    return(0);
}

```

假定下面这个函数的进程中的另一个线程也要求访问这两个数组：

```

DWORD WINAPI OtherThreadFunc(PVOID pvParam) {

    EnterCriticalSection(&g_csChars);
    EnterCriticalSection(&g_csNums);

    for (int x = 0; x < 100; x++)
        g_nNums[x] = g_cChars[x];

    LeaveCriticalSection(&g_csNums);
    LeaveCriticalSection(&g_csChars);
    return(0);
}

```

在上面这个函数中我只是切换了对 EnterCriticalSection 和 LeaveCriticalSection 函数的调用顺序。但是，由于这两个函数是按上面这种方式编写的，因此可能产生一个死锁状态。假定 ThreadFunc 开始执行，并且获得了 g_csNums 关键代码段的所有权，那么执行 OtherThreadFunc 函数的线程就被赋予一定的 CPU 时间，并可获得 g_csChars 关键代码段的所有权。这时就出现了一个死锁状态。当 ThreadFunc 或 OtherThreadFunc 中的任何一个函数试图继续执行时，这两个函数都无法取得对它需要的另一个关键代码段的所有权。

为了解决这个问题，必须始终按照完全相同的顺序请求对资源的访问。注意，当调用 LeaveCriticalSection 函数时，按照什么顺序访问资源是没有关系的，因为该函数决不会使线程进入等待状态。

3. 不要长时间运行关键代码段

当一个关键代码段长时间运行时，其他线程就会进入等待状态，这会降低应用程序的运行性能。下面这个方法可以用来最大限度地减少关键代码段运行所花费的时间。这个代码能够防止其他线程在 WM_SOMEMSG 消息发送到一个窗口之前改变 g_s 的值：

```

SOMESTRUCT g_s;
CRITICAL_SECTION g_cs;

DWORD WINAPI SomeThread(PVOID pvParam) {
    EnterCriticalSection(&g_cs);

    // Send a message to a window.
    SendMessage(hwndSomeWnd, WM_SOMEMSG, &g_s, 0);

    LeaveCriticalSection(&g_cs);
    return(0);
}

```

无法确定窗口过程处理 WM_SOMEMSG 消息时需要花费多长时间，它可能是几个毫秒，也可能需要几年时间。在这个时间内，其他线程都不能访问 g_s 结构。这个代码最好编写成下面的形式：

```

SOMESTRUCT g_s;
CRITICAL_SECTION g_cs;

```

```
DWORD WINAPI SomeThread(PVOID pvParam) {  
  
    EnterCriticalSection(&g_cs);  
    SOMESTRUCT sTemp = g_s;  
    LeaveCriticalSection(&g_cs);  
  
    // Send a message to a window.  
    SendMessage(hwndSomeWnd, WM_SOMEMSG, &sTemp, 0);  
    return(0);  
}
```

这个代码将该值保存在临时变量 `sTemp` 中。也许你能够猜到 CPU 需要多长时间来执行这行代码——只需要几个 CPU 周期。当该临时变量保存后，`LeaveCriticalSection` 函数就立即被调用，因为这个全局结构不再需要保护。上面的第二个实现代码比第一个要好得多，因为其他线程只是在几个 CPU 周期内被停止使用 `g_s` 结构，而不是无限制地停止使用该结构。当然，这个方法的前提是该结构的“瞬态图”应当做到非常好才行，以方便于窗口过程读取。此外，窗口过程不需要改变该结构中的成员。