[秒杀多线程第六篇 经典线程同步 事件Event](http://blog.csdn.net/morewindows/article/details/7445233)

上一篇中使用[关键段](http://blog.csdn.net/morewindows/article/details/7442639)来解决经典的多线程同步互斥问题，由于[关键段](http://blog.csdn.net/morewindows/article/details/7442639)的“线程所有权”特性所以关键段只能用于线程的互斥而不能用于同步。本篇介绍用事件Event来尝试解决这个线程同步问题。

首先介绍下如何使用事件。事件Event实际上是个内核对象，它的使用非常方便。下面列出一些常用的函数。

第一个 CreateEvent

函数功能：创建事件

函数原型：

HANDLE CreateEvent(

LPSECURITY\_ATTRIBUTES lpEventAttributes,

 BOOL bManualReset,

 BOOL bInitialState,

 LPCTSTR lpName

);

函数说明：

第一个参数 LPSECURITY\_ATTRIBUTES lpEventAttributes表示安全控制，一般直接传入NULL。

第二个参数BOOL bManualReset确定事件是手动置位还是自动置位，传入TRUE表示手动置位，传入FALSE表示自动置位。如果为自动置位，则对该事件调用WaitForSingleObject()后会自动调用ResetEvent()使事件变成未触发状态。第三个参数BOOL bInitialState表示事件的初始状态，传入TRUR表示已触发。

第四个参数LPCTSTR lpName表示事件的名称，传入NULL表示匿名事件。

第二个 OpenEvent

函数功能：根据名称获得一个事件句柄。

函数原型：

HANDLE OpenEvent(

 DWORD dwDesiredAccess,

 BOOL bInheritHandle,

 LPCTSTR lpName     //名称

);

函数说明：

第一个参数DWORD dwDesiredAccess表示访问权限，对事件一般传入EVENT\_ALL\_ACCESS。详细解释可以查看MSDN文档。

第二个参数BOOL bInheritHandle表示事件句柄继承性，一般传入TRUE即可。

第三个参数LPCTSTR lpName表示名称，不同进程中的各线程可以通过名称来确保它们访问同一个事件。

第三个SetEvent

函数功能：触发事件

函数原型：BOOL SetEvent(HANDLEhEvent);

函数说明：每次触发后，必有一个或多个处于等待状态下的线程变成可调度状态。

第四个ResetEvent

函数功能：将事件设为末触发

函数原型：BOOLResetEvent(HANDLEhEvent);

最后一个事件的清理与销毁

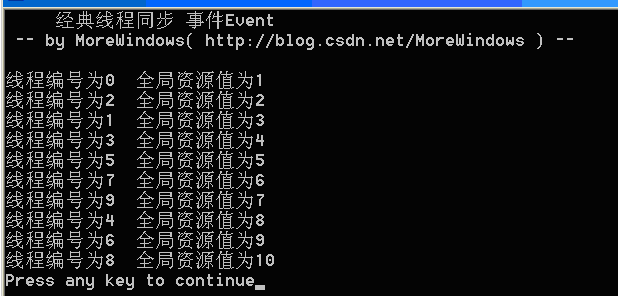
由于事件是内核对象，因此使用CloseHandle()就可以完成清理与销毁了。

在经典多线程问题中设置一个事件和一个关键段。用事件处理主线程与子线程的同步，用关键段来处理各子线程间的互斥。详见代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7445233)

1. #include <stdio.h>
2. #include <process.h>
3. #include <windows.h>
4. **long** g\_nNum;
5. unsigned **int** \_\_stdcall Fun(**void** \*pPM);
6. **const** **int** THREAD\_NUM = 10;
7. //事件与关键段
8. **HANDLE**  g\_hThreadEvent;
9. CRITICAL\_SECTION g\_csThreadCode;
10. **int** main()
11. {
12. printf("     经典线程同步 事件Event\n");
13. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
14. //初始化事件和关键段 自动置位,初始无触发的匿名事件
15. g\_hThreadEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
16. InitializeCriticalSection(&g\_csThreadCode);
18. **HANDLE**  handle[THREAD\_NUM];
19. g\_nNum = 0;
20. **int** i = 0;
21. **while** (i < THREAD\_NUM)
22. {
23. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, Fun, &i, 0, NULL);
24. WaitForSingleObject(g\_hThreadEvent, INFINITE); //等待事件被触发
25. i++;
26. }
27. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
29. //销毁事件和关键段
30. CloseHandle(g\_hThreadEvent);
31. DeleteCriticalSection(&g\_csThreadCode);
32. **return** 0;
33. }
34. unsigned **int** \_\_stdcall Fun(**void** \*pPM)
35. {
36. **int** nThreadNum = \*(**int** \*)pPM;
37. SetEvent(g\_hThreadEvent); //触发事件
39. Sleep(50);//some work should to do
41. EnterCriticalSection(&g\_csThreadCode);
42. g\_nNum++;
43. Sleep(0);//some work should to do
44. printf("线程编号为%d  全局资源值为%d\n", nThreadNum, g\_nNum);
45. LeaveCriticalSection(&g\_csThreadCode);
46. **return** 0;
47. }

运行结果如下图：



可以看出来，经典线程同步问题已经圆满的解决了——线程编号的输出没有重复，说明主线程与子线程达到了同步。全局资源的输出是递增的，说明各子线程已经互斥的访问和输出该全局资源。

现在我们知道了如何使用事件，但学习就应该要深入的学习，何况微软给事件还提供了PulseEvent()函数，所以接下来再继续深挖下事件Event，看看它还有什么秘密没。

先来看看这个函数的原形：

第五个PulseEvent

函数功能：在事件触发后立即将事件设置为未触发，相当于触发一个事件脉冲。

函数原型：BOOL PulseEvent(HANDLEhEvent);

函数说明：这是一个不常用的事件函数，此函数相当于SetEvent()后立即调用ResetEvent();此时情况可以分为两种：

1.对于手动置位事件，所有正处于等待状态下的线程都变成可调度状态。

2.对于自动置位事件，所有正处于等待状态下线程只有一个变成可调度状态。（因为自动置位中某一线程获得资源后会将事件置为未触发）

此后事件是末触发的。该函数不稳定，因为无法预知在调用PulseEvent ()时哪些线程正处于等待状态。

       下面对这个触发一个事件脉冲PulseEvent ()写一个例子，主线程启动7个子线程，其中有5个线程Sleep(10)后对一事件调用等待函数（称为快线程），另有2个线程Sleep(100)后也对该事件调用等待函数（称为慢线程）。主线程启动所有子线程后再Sleep(50)保证有5个快线程都正处于等待状态中。此时若主线程触发一个事件脉冲，那么对于手动置位事件，这5个线程都将顺利执行下去。对于自动置位事件，这5个线程中会有中一个顺利执行下去。而不论手动置位事件还是自动置位事件，那2个慢线程由于Sleep(100)所以会错过事件脉冲，因此慢线程都会进入等待状态而无法顺利执行下去。

代码如下：

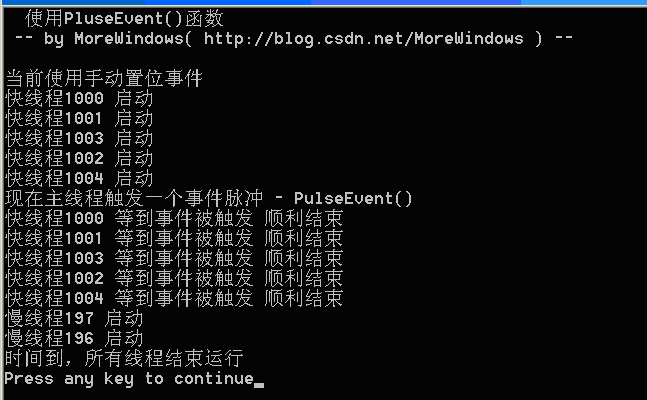
**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7445233)

1. //使用PluseEvent()函数
2. #include <stdio.h>
3. #include <conio.h>
4. #include <process.h>
5. #include <windows.h>
6. **HANDLE**  g\_hThreadEvent;
7. //快线程
8. unsigned **int** \_\_stdcall FastThreadFun(**void** \*pPM)
9. {
10. Sleep(10); //用这个来保证各线程调用等待函数的次序有一定的随机性
11. printf("%s 启动\n", (**PSTR**)pPM);
12. WaitForSingleObject(g\_hThreadEvent, INFINITE);
13. printf("%s 等到事件被触发 顺利结束\n", (**PSTR**)pPM);
14. **return** 0;
15. }
16. //慢线程
17. unsigned **int** \_\_stdcall SlowThreadFun(**void** \*pPM)
18. {
19. Sleep(100);
20. printf("%s 启动\n", (**PSTR**)pPM);
21. WaitForSingleObject(g\_hThreadEvent, INFINITE);
22. printf("%s 等到事件被触发 顺利结束\n", (**PSTR**)pPM);
23. **return** 0;
24. }
25. **int** main()
26. {
27. printf("  使用PluseEvent()函数\n");
28. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
30. **BOOL** bManualReset = FALSE;
31. //创建事件 第二个参数手动置位TRUE，自动置位FALSE
32. g\_hThreadEvent = CreateEvent(NULL, bManualReset, FALSE, NULL);
33. **if** (bManualReset == TRUE)
34. printf("当前使用手动置位事件\n");
35. **else**
36. printf("当前使用自动置位事件\n");
38. **char** szFastThreadName[5][30] = {"快线程1000", "快线程1001", "快线程1002", "快线程1003", "快线程1004"};
39. **char** szSlowThreadName[2][30] = {"慢线程196", "慢线程197"};
41. **int** i;
42. **for** (i = 0; i < 5; i++)
43. \_beginthreadex(NULL, 0, FastThreadFun, szFastThreadName[i], 0, NULL);
44. **for** (i = 0; i < 2; i++)
45. \_beginthreadex(NULL, 0, SlowThreadFun, szSlowThreadName[i], 0, NULL);
47. Sleep(50); //保证快线程已经全部启动
48. printf("现在主线程触发一个事件脉冲 - PulseEvent()\n");
49. PulseEvent(g\_hThreadEvent);//调用PulseEvent()就相当于同时调用下面二句
50. //SetEvent(g\_hThreadEvent);
51. //ResetEvent(g\_hThreadEvent);
53. Sleep(3000);
54. printf("时间到，主线程结束运行\n");
55. CloseHandle(g\_hThreadEvent);
56. **return** 0;
57. }

对自动置位事件，运行结果如下：



对手动置位事件，运行结果如下：



最后总结下事件Event

1．事件是内核对象，事件分为**手动置位事件**和**自动置位事件**。事件Event内部它包含一个使用计数（所有内核对象都有），一个布尔值表示是手动置位事件还是自动置位事件，另一个布尔值用来表示事件有无触发。

2．事件可以由SetEvent()来触发，由ResetEvent()来设成未触发。还可以由PulseEvent()来发出一个事件脉冲。

3．事件可以解决线程间同步问题，因此也能解决互斥问题。

1. //初始为触发
2. g\_eventC=CreateEvent(NULL,FALSE,TRUE,NULL);
4. unsigned **int** \_stdcall Thread\_Fun(**LPVOID** lp){
5. **int** i=\*(**int** \*)(lp);
7. //触发匿名事件
8. SetEvent(g\_eventI);
10. Sleep(100);
12. //用Event代替关键段
13. WaitForSingleObject(g\_eventC,INFINITE);
14. g\_count++;
15. Sleep(10);
16. printf("线程编号：%d  线程输出全局变量:%d \n",i,g\_count);
17. //触发 让其它等待的线程进行挣用
18. SetEvent(g\_eventC);
20. **return** 0;
21. }

既然这篇学了事件，就用事件代替了这里的关键段实现线程互斥访问公有资源。这么写有没有问题呢？ 运行结果是对的。

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7445233)

1. **BOOL** g\_blFlg = FALSE;
2. **while** (i < THREAD\_NUM)
3. {
4. **if**(!g\_blFlg)
5. {
6. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, Fun, &i, 0, NULL);
7. }
8. **while** (!g\_blFlg);//等待线程参数传递结束之后再i++
10. //WaitForSingleObject(g\_hThreadEvent, INFINITE); //等待事件被触发
11. i++;
12. g\_blFlg = FALSE;//准备创建新线程
13. }
15. unsigned **int** \_\_stdcall Fun(**void** \*pPM)
16. {
17. **int** nThreadNum = \*(**int** \*)pPM;
18. //SetEvent(g\_hThreadEvent); //触发事件
19. g\_blFlg = TRUE;//全局flag设为TRUE代表事件被触发
21. Sleep(50);//some work should to do
23. EnterCriticalSection(&g\_csThreadCode);
24. g\_nNum++;
25. Sleep(0);//some work should to do
26. printf("线程编号为%d  全局资源值为%d\n", nThreadNum, g\_nNum);
27. LeaveCriticalSection(&g\_csThreadCode);
28. **return** 0;
29. }

本人用了一个全局Flag，没有使用事件和信号量，实现了同样的功能，不知道我这种方法算不算符合题意要求，应该也算是并发执行的吧

[](http://blog.csdn.net/z82100)

楼主你好，第一段代码我将线程数改到65之后就vs2013会报错了，这是为什么？  
0x77598E19 (ntdll.dll) (Thread\_learning.exe 中)处有未经处理的异常: 0xC0000005: 写入位置 0x00000014 时发生访问冲突。  
中断位置在 EnterCriticalSection(&g\_csThreadCode);

回复z82100：windows下用事件同步时，线程最大数目为64

回复z82100：是因为最大等待数被设置为64了吗？

[http://avatar.csdn.net/A/0/A/3_kewencommon.jpg](http://blog.csdn.net/kewencommon)

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7445233)

1. 用事件来解决互斥问题，以下的例子有问题么？lz和看见的各位麻烦回答一下

4. #include "stdafx.h"
5. #include <stdio.h>
6. #include<windows.h>
7. #include<process.h>
9. **const** **int** THREAD\_NUM = 10;//子线程数
10. **int** g\_nNum;//全局变量
11. unsigned **int** \_\_stdcall Fun(**void** \*pSeq);
12. **HANDLE** EventHandle;//事件句柄
14. **int** \_tmain(**int** argc, \_TCHAR\* argv[])
15. {
16. **HANDLE** handle[THREAD\_NUM];
17. **int** i = 0;
18. g\_nNum = 0;
19. EventHandle = CreateEvent(NULL,FALSE,FALSE,NULL);
21. **while**(i<THREAD\_NUM)
22. {
23. handle[i] =(**HANDLE**)\_beginthreadex(NULL,0,Fun,NULL,0,NULL);
24. Sleep(10);
25. i++;
26. SetEvent(EventHandle);
27. //      Sleep(0);
28. }
29. system("pause");
30. **return** 0;
31. }
33. unsigned **int** \_\_stdcall Fun(**void** \*pSeq)
34. {
35. **int** gThreadCount = g\_nNum;
36. g\_nNum ++;
37. WaitForSingleObject(EventHandle,INFINITE);
38. Sleep(50);//some work to do  放到WaitForSingleObject前就不行了
39. printf("全局变量[%d]\n",gThreadCount);
40. **return** 0;
41. }

Re: [FENGYEJINGXIANG](http://blog.csdn.net/FENGYEJINGXIANG) 2014-10-16 15:10发表 [[回复]](http://blog.csdn.net/morewindows/article/details/7445233#reply)

[](http://blog.csdn.net/FENGYEJINGXIANG)

回复kewencommon：这样写是有问题的  
因为主函数的Sleep时间比Fun函数中的Sleep时间短，SetEvent(EventHandle);会在主函数中连续运行很多次，然后进入任意一个Fun函数后，遇到WaitForSingleObject后EventHandle就被复位了（自动复位），多个等待的Fun函数只有一个运行了。  
也就是说这样最后显示出的数据比预期要少---发现把Fun函数中的Sleep函数放到WaitForSingleObject上该回复就是正确的。

Re: [TCSS001](http://blog.csdn.net/K616358281) 2015-04-02 23:00发表 [[回复]](http://blog.csdn.net/morewindows/article/details/7445233#reply)

[](http://blog.csdn.net/K616358281)

回复FENGYEJINGXIANG：主线程虽然sleep时间短，但是子线程是操作完成以后才开始sleep的啊，也就意味着最终肯定能打印出来。  
大致顺序是：启动子线程i->主线程sleep，同时子线程开始读取全局变量并进行相应操作并进入等待->主线程醒来，触发事件->子线程开始sleep->主线程开启下一个线程。  
后续打印出来的结果顺序就不一定正确了，没法保证打印出来的顺序是同步的，但是可以保证操作的过程是同步的。

[](http://blog.csdn.net/yanziguilai)

int nThreadNum = \*(int \*)pPM;   
SetEvent(g\_hThreadEvent); //触发事件  
把上面的两句代码换一下就会出现错误，？？？？？  
SetEvent(g\_hThreadEvent); //触发事件  
int nThreadNum = \*(int \*)pPM;   
楼主解释一下

回复yanziguilai：主线程传入 i 的地址.  
你把位置换一下, 主线程没有等子线程取得 i 的值就把 i++ 了.

------我：在这里使用事件就是为了对数据的读写进行同步，把SetEvent放到读数据之前就起不到同步的作用了。

[](http://blog.csdn.net/springontime)

引用“MoreWindows”的评论：回复cohn：不是的，手动置位表示等待事件的线程不会去修改事件。而且对自动...  
楼主，其实我觉得你还没有详细回答他的问题，我也有同样的疑问，我觉得这个要从pulseEvent产生的复位脉冲的时间长度上来解释，假设我们已手动复位的事件为前提，如果pulseEvent产生的这个脉冲极短那么任何一个线程都无法被触发，因为置位后会马上复位；如果时间较短，那么可能只够1,2个线程得到复位信号，只有这个脉冲足够长才能够保证5个快线程全部得到复位信号，从楼主的代码运行结构上看，这个脉冲信号是足够长的，足以保证5个快线程在pulseEvent将事件复位前得到事件置位的消息。楼主觉得我说的有道理吗

回复springontime：pulseEvent从长短解释不现实，感觉应该是对于自动复位的，所有线程一起抢，抢到了就顺手复位了，可以理解成进教室的例子。而对于手动复位的，不存在抢的问题，就是在那等信号就OK了，有信号来了就往下走，不考虑关门的问题。

[](http://blog.csdn.net/liuyunjay66)

有几个疑问：  
1.windows下的多线程是真的多线程吗？还是看上去是“并发的”，跟处理的单核，多核是什么个关系啊？忘博主指点..

回复liuyunjay66：多线程肯定是真的多线程，哪有假的多线程？但是并行可以是真的也可以是“看起来”的，单核CPU的并行肯定是“看起来”的，多核CPU如果把不同的线程放到不同的核里去执行，那就是真的并行。

[](http://blog.csdn.net/christiana10247)

我有一个地方不太清楚，就是为什么在自动配置的时候，只显示一个事件被触发顺利结束？是因为其他事件还没等到被触发，就CloseHandle了么

回复christiana10247：自动复位的话，某一个线程得到触发后，马上就调用WaitForSingleObject（）。之后 该事件就被复位为未触发了，其他还没有来得及触发的线程，就只有继续等待了..

[](http://blog.csdn.net/yaxiya)

引用“Unix\_Architect”的评论：经过测试。  
int nThreadNum = \*(int \*)pPM;   
SetEve...  
我感觉博主的意思是说事件可以处理线程同步;如果把fun()函数中的setEvent()放到最后一行,那就彻底的同步了:主线程创建子线程,然后子线程一直执行到最后一行的setEvent();这样一来,主线程与子线程就彻底的同步,就如同只有一个线程在执行;可是这好像又没有多大意义;毕竟博主想让大家明白多线程的学习;

回复yaxiya：将setEvent 放到后面。这样就实现了线程号和资源号的对应，但是这样会不会降低同步的效率

回复zyz913614263：这样好像就不是多线程并发了,应该降低了效率;

[](http://blog.csdn.net/woshinia)

事件确实可以严格控制每个线程的执行顺序，但大多数时候，每个线程都是一个单独的任务副本，即不存在说一定要先让线程A操作公共资源再让线程B操作，B先A后也可以，只要保证公共资源不被同时改写，能正确的读写就行了。因此大多数时候临界区就可以了，事件的开销要大很多。

回复woshinia：这个是的，用户级线程同步互斥对象和内核级线程同步对象所消费的资源相差还是有点大。

回复woshinia：事件和信号量一般在线程间通信，即有数据交互的时候才会用。

楼主只是说了怎样使用事件。但对于什么情况下最适合用事件介绍的太少，望楼主能添加一些。

[](http://blog.csdn.net/Unix_Architect)

经过测试。  
int nThreadNum = \*(int \*)pPM;   
SetEvent(g\_hThreadEvent); //触发事件   
这行代码的作用就是使 线程的标号同步了，如果没有这句的话由于传递的是地址，所有标号最后都是9号。  
LZ不会为了标号同步使用了一个核心对象吧。。。速度可比CRITICAL（临界区）慢100倍呢。

回复MoreWindows： CRITICAL确实实现了变量之间的互斥效果。书上把CRITICAL列入了同步机制的范畴。

回复Unix\_Architect：呵呵，从广义上讲，线程互斥也可以认为是线程同步的一种，这个在《计算机操作系统》（汤子赢）书上有解释。  
因此线程同步的常用工具可以说有关键段，事件，互斥量，信号量这四种，但其实关键段与互斥量只能用于互斥，事件与信号量才既能用于同步，也能用于互斥。

所以我觉得线程的根本是，再不涉及到共享数据结构的时候，就是随便执行，什么同步不同步的，只有涉及到了共享数据，然后大家再同步。你觉得呢？

回复Unix\_Architect："同步"是"互斥"的同时还有对顺序的要求,是更严格的"互斥",

[http://avatar.csdn.net/8/0/2/3_titer1.jpg](http://blog.csdn.net/titer1)

最近看了 windows系列的同步变量。  
有关键区，  
有event ,mutex,semaphore,  
其中event和semaphore有线程所有权的性质，有点乱。  
  
现在以event为例子，汇总下自己的调查结果，大家看对不对？  
  
  
汇总如下：  
1）关键段有“线程所有权”特性，所以关键段只能用于线程的互斥而不能用于同步。//from morewindows  
  
2）只有持有该event的线程可以释放它，其他线程调用释放没有效果（而且还可能出错）。  
3）如果持有该event的线程意外退出，该event将会被遗弃，被系统回收。

[](http://blog.csdn.net/Aselan)

楼主写得很好，让我对多线程的理解更加深刻了。  
另外，我对楼主的代码改动一下，不用临界段，只使用事件对象，并且移动了线程函数的位置。从运行结果上来看，实现了多线程的同步。不知道算不算是逻辑上有问题，楼主的看法呢？

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7445233)

1. unsigned **int** \_\_stdcall Fun(**void** \*pPM)
2. {
3. **int** nThreadNum = \*(**int** \*)pPM;
5. Sleep(50);
7. //EnterCriticalSection(&g\_csThreadCode);
8. g\_nNum++;
9. Sleep(0);
10. printf("线程编号为%d 全局资源值为%d\n",nThreadNum,g\_nNum);
11. //LeaveCriticalSection(&g\_csThreadCode);
12. SetEvent(g\_hThreadEvent);   //触发事件
13. **return** 0;
14. }

运行结果：  
线程编号为0 全局资源值为1  
线程编号为1 全局资源值为2  
线程编号为2 全局资源值为3  
线程编号为3 全局资源值为4  
线程编号为4 全局资源值为5  
线程编号为5 全局资源值为6  
线程编号为6 全局资源值为7  
线程编号为7 全局资源值为8  
线程编号为8 全局资源值为9  
线程编号为9 全局资源值为10

回复Aselan：这已经不是多线程并发了，而是多个线程依次运行。每一个新线程开启，都要等上一个线程完事。

回复Aselan：你这样的话，每个线程都是等到之前的线程运行结束才开始下一个线程的运行，这个时候多线程就没有意义了

回复Aselan：哥们，你这样改的话，执行过程就是创建一个线程后。等待这个线程执行结束。才创建下一个，所以才出现这种情况，是吗？

是的。正如楼主所说的一样，这样丧失了多线程并发的优势。要发挥多线程并发的优势，应该只在需要同步的地方进行同步才行。回复aini201：

回复aini201：他这样做，每个子线程要等前一个子线程执行完毕才能启动，失去了多线程并发的优势了。

[](http://blog.csdn.net/caigen1988)

楼主，我有个问题想请教下您。  
首先说明下我没有学过windows下的多线程编程，学的是Linux下的，但我想原理应该想通吧 。  
问题是：在您第一个代码的第24行是以下代码：  
WaitForSingleObject(g\_hThreadEvent, INFINITE); //等待事件被触发   
我们以创建第一个线程时为例，如果第一个线程刚创建时系统就把运行权给了第一个线程，即开始执行Fun函数内的语句。当Fun函数中运行的语句SetEvent运行结束后系统才将控制权转给主线程，即系统开始执行WaitForSingleObject 语句。 也就是说如果SetEvent语句在WaitForSingleObject语句执行之前，那么系统会不会出现阻塞等待？   
不知道解释清楚了没，期待您的答疑，谢谢呵呵。

回复caigen1988：“如果第一个线程刚创建时系统就把运行权给了第一个线程”，这个在Windows下不是这样的，在Windows下，父线程在创建子线程后，父子线程会同时执行，二个线程都有运行权的。

回复caigen1988：“不知道解释清楚了没，期待您的答疑，谢谢呵呵。”  
--我尝试回答一下。呵呵  
--我跟caigen1988有同样的疑问。  
“就是说如果SetEvent语句在WaitForSingleObject语句执行之前，那么系统会不会出现阻塞等待？ ”  
--我认为：系统会出现阻塞等待，并且一直等待下去，直到遇到下个SetEvent给的信号。如果SetEvent只执行一次，并且在WaitForSingleObject前执行，那么WaitForSingleObject永远得不到触发，所以会一直阻塞下去。  
--以上想法，请大家判明。

回复excelbeginner：你们这么说是不对的，只要执行了SetEvent(g\_hThreadEvent)，事件就会把触发了，然后，WaitForSingleObject(g\_hThreadEvent, INFINITE)就会立刻返回。不会出现无穷等的情况

回复linrulei11：正解，WaitForSingleObject(g\_hThreadEvent, INFINITE)只判断g\_hThreadEvent是否触发，未触发就等待，已经触发就直接返回。

回复MoreWindows：貌似时间有点长了，不过看到好的讨论，还是想回复一下。我没有专门研究过Linux的线程，但是我是在Windows下做Qt的，在Qt里面，如果不是调用系统的API而是调用Qt的库函数，类似这个WaitForSingleObject(g\_hThreadEvent, INFINITE)的机制，如果在等待之前g\_hThreadEvent已经被置位了，它是不知道的，会一直等下去，但是Windows原生的不会一直等下去。 Qt里面的很多机制就是Linux的机制，所以我猜这也是Linux和Windows的不同。

[](http://blog.csdn.net/down_login)

请教两个小白问题。1.线程有一个入口函数，若是在入口函数中有引用了另个函数，那这个函数是不是也要考虑到线程同步互斥呢？  
2.线程被触发是否就是指线程由不可运行状态转换到可运行状态？在这里，WaitForMultipleObjects（）函数的作用就是阻塞了主线程，直到其他的对象全部什么了什么了之后才重新唤醒主线程，也就是说这个函数阻塞当前线程，直到由他创建的线程全部执行完之后才被唤醒。

回复down\_login：第一个问题，线程同步问题要具体情况具体分析。你说的要引用了别一个函数，那要看看这个函数的功能是什么。  
第二个问题，在http://blog.csdn.net/morewindows/article/details/7421759这里有讲解。

9楼 [matianyu00](http://blog.csdn.net/matianyu00) 2012-04-11 21:51发表 [[回复]](http://blog.csdn.net/morewindows/article/details/7445233#reply)

[http://avatar.csdn.net/D/3/0/3_matianyu00.jpg](http://blog.csdn.net/matianyu00)

楼主你好。看了你的帖子真是学到了很多东西阿。这里我想问几个问题。  
1.线程同步和线程互斥在定义上和概念有什么区别呢？  
2.在什么场合使用线程同步，在什么场合使用线程互斥呢？

回复matianyu00：个人理解：互斥主要指多个线程不能同时访问一个资源，如打印机就是互斥资源。同步是指多个线程要按一定的次序访问，如上餐馆吃饭，只有菜上桌后你才能吃。互斥其实是一种特殊的同步。详细的定义可以参考操作系统方面的书籍。

[http://avatar.csdn.net/A/D/A/3_scorpiuseol.jpg](http://blog.csdn.net/scorpiuseol)

Sleep()到底做了什么？主线程sleep了，子线程也跟着sleep么？

回复scorpiuseol：调用Sleep()就相当于执行态变成阻塞态，通俗的说就是线程休眠一段时间。线程在执行上肯定是独立的，一个线程调用Sleep()不会影响其它线程。