[秒杀多线程第一篇 多线程笔试面试题汇总](http://blog.csdn.net/morewindows/article/details/7392749)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-04-05 09:35 100487人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7392749#comments)(106) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7392749#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[知名公司](http://www.csdn.net/tag/%e7%9f%a5%e5%90%8d%e5%85%ac%e5%8f%b8)[面试题汇总](http://www.csdn.net/tag/%e9%9d%a2%e8%af%95%e9%a2%98%e6%b1%87%e6%80%bb)

目录[(?)[+]](http://blog.csdn.net/morewindows/article/details/7392749)

    系列前言

    本系列是本人参加微软亚洲研究院，腾讯研究院，迅雷面试时整理的，另外也加入一些其它IT公司如百度，阿里巴巴的笔试面试题目，因此具有很强的针对性。系列中不但会详细讲解多线程同步互斥的各种“招式”，而且会进一步的讲解多线程同步互斥的“内功心法”。有了“招式”和“内功心法”，相信你也能对多线程挥洒自如，在笔试面试中顺利的秒杀多线程试题。

              -------------------------------------华丽的分割线---------------------------------------

**第一篇    多线程笔试面试题汇总**

    多线程在笔试面试中经常出现，下面列出一些公司的多线程笔试面试题。首先是一些概念性的问答题，这些是多线程的基础知识，经常出现在面试中的第一轮面试（我参加2011年腾讯研究院实习生招聘时就被问到了几个概念性题目）。然后是一些选择题，这些一般在笔试时出现，虽然不是太难，但如果在选择题上花费大多时间无疑会对后面的编程题造成影响，因此必须迅速的解决掉。最后是综合题即难一些的问答题或是编程题。这种题目当然是最难解决了，要么会引来面试官的追问，要么就很容易考虑不周全，因此解决这类题目时一定要考虑全面和细致。

    下面就来看看这三类题目吧。

**一．概念性问答题**

第一题：线程的基本概念、线程的基本状态及状态之间的关系？

第二题：线程与进程的区别？

       这个题目问到的概率相当大，计算机专业考研中也常常考到。要想全部答出比较难。

第三题：多线程有几种实现方法，都是什么？

第四题：多线程同步和互斥有几种实现方法，都是什么？

       我在参加2011年迅雷校园招聘时的一面和二面都被问到这个题目，回答的好将会给面试成绩加不少分。

第五题：多线程同步和互斥有何异同，在什么情况下分别使用他们？举例说明。

**二．选择题**

第一题（百度笔试题）：

以下多线程对int型变量x的操作，哪几个不需要进行同步：   
A. x=y;      B. x++;    C. ++x;    D. x=1;

第二题（阿里巴巴笔试题）

多线程中栈与堆是公有的还是私有的

A：栈公有, 堆私有

B：栈公有，堆公有

C：栈私有, 堆公有

D：栈私有，堆私有

**三．综合题**

第一题（台湾某杀毒软件公司面试题）：

在Windows编程中互斥量与临界区比较类似，请分析一下二者的主要区别。

第二题：

一个全局变量tally，两个线程并发执行（代码段都是ThreadProc)，问两个线程都结束后，tally取值范围。

inttally = 0;//glable

voidThreadProc()

{

       for(inti = 1; i <= 50; i++)

              tally += 1;

}

第三题（某培训机构的练习题）：

子线程循环 10 次，接着主线程循环 100 次，接着又回到子线程循环 10 次，接着再回到主线程又循环 100 次，如此循环50次，试写出代码。

第四题（迅雷笔试题）：

编写一个程序，开启3个线程，这3个线程的ID分别为A、B、C，每个线程将自己的ID在屏幕上打印10遍，要求输出结果必须按ABC的顺序显示；如：ABCABC….依次递推。

第五题（Google面试题）

有四个线程1、2、3、4。线程1的功能就是输出1，线程2的功能就是输出2，以此类推.........现在有四个文件ABCD。初始都为空。现要让四个文件呈如下格式：

A：1 2 3 4 1 2....

B：2 3 4 1 2 3....

C：3 4 1 2 3 4....

D：4 1 2 3 4 1....

请设计程序。

下面的第六题与第七题也是在考研中或是程序员和软件设计师认证考试中的热门试题。

第六题

生产者消费者问题

这是一个非常经典的多线程题目，题目大意如下：有一个生产者在生产产品，这些产品将提供给若干个消费者去消费，为了使生产者和消费者能并发执行，在两者之间设置一个有多个缓冲区的缓冲池，生产者将它生产的产品放入一个缓冲区中，消费者可以从缓冲区中取走产品进行消费，所有生产者和消费者都是异步方式运行的，但它们必须保持同步，即不允许消费者到一个空的缓冲区中取产品，也不允许生产者向一个已经装满产品且尚未被取走的缓冲区中投放产品。

第七题

读者写者问题

这也是一个非常经典的多线程题目，题目大意如下：有一个写者很多读者，多个读者可以同时读文件，但写者在写文件时不允许有读者在读文件，同样有读者读时写者也不能写。

多线程相关题目就列举到此，如果各位有多线程方面的笔试面试题，欢迎提供给我，我将及时补上。谢谢大家。

下一篇《[多线程第一次亲密接触 CreateThread与\_beginthreadex本质区别](http://blog.csdn.net/morewindows/article/details/7421759)》将从源代码的层次上讲解创建多线程的二个函数CreateThread与\_beginthreadex到底有什么区别，让你明明白白的完成与多线程第一次亲密接触。

# [秒杀多线程第二篇 多线程第一次亲密接触 CreateThread与\_beginthreadex本质区别](http://blog.csdn.net/morewindows/article/details/7421759)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-04-05 09:36 102111人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7421759#comments)(100) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7421759#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[thread](http://www.csdn.net/tag/thread)[null](http://www.csdn.net/tag/null)[structure](http://www.csdn.net/tag/structure)[attributes](http://www.csdn.net/tag/attributes)[security](http://www.csdn.net/tag/security)

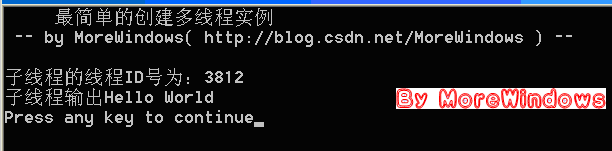
    本文将带领你与多线程作第一次亲密接触，并深入分析CreateThread与\_beginthreadex的本质区别，相信阅读本文后你能轻松的使用多线程并能流畅准确的回答CreateThread与\_beginthreadex到底有什么区别，在实际的编程中到底应该使用CreateThread还是\_beginthreadex？

   使用多线程其实是非常容易的，下面这个程序的主线程会创建了一个子线程并等待其运行完毕，子线程就输出它的线程ID号然后输出一句经典名言——Hello World。整个程序的代码非常简短，只有区区几行。

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7421759)

1. //最简单的创建多线程实例
2. #include <stdio.h>
3. #include <windows.h>
4. //子线程函数
5. **DWORD** WINAPI ThreadFun(**LPVOID** pM)
6. {
7. printf("子线程的线程ID号为：%d\n子线程输出Hello World\n", GetCurrentThreadId());
8. **return** 0;
9. }
10. //主函数，所谓主函数其实就是主线程执行的函数。
11. **int** main()
12. {
13. printf("     最简单的创建多线程实例\n");
14. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
16. **HANDLE** handle = CreateThread(NULL, 0, ThreadFun, NULL, 0, NULL);
17. WaitForSingleObject(handle, INFINITE);
18. **return** 0;
19. }

运行结果如下所示：



下面来细讲下代码中的一些函数

第一个 CreateThread

函数功能：创建线程

函数原型：

HANDLEWINAPICreateThread(

  LPSECURITY\_ATTRIBUTESlpThreadAttributes,

  SIZE\_TdwStackSize,

  LPTHREAD\_START\_ROUTINElpStartAddress,

  LPVOIDlpParameter,

  DWORDdwCreationFlags,

  LPDWORDlpThreadId

);

函数说明：

第一个参数表示线程内核对象的安全属性，一般传入NULL表示使用默认设置。

第二个参数表示线程栈空间大小。传入0表示使用默认大小（1MB）。

第三个参数表示新线程所执行的线程函数地址，多个线程可以使用同一个函数地址。

第四个参数是传给线程函数的参数。

第五个参数指定额外的标志来控制线程的创建，为0表示线程创建之后立即就可以进行调度，如果为CREATE\_SUSPENDED则表示线程创建后暂停运行，这样它就无法调度，直到调用ResumeThread()。

第六个参数将返回线程的ID号，传入NULL表示不需要返回该线程ID号。

函数返回值：

成功返回新线程的句柄，失败返回NULL。

第二个 WaitForSingleObject

函数功能：等待函数 – 使线程进入等待状态，直到指定的内核对象被触发。

函数原形：

DWORDWINAPIWaitForSingleObject(

  HANDLEhHandle,

  DWORDdwMilliseconds

);

函数说明：

第一个参数为要等待的内核对象。

第二个参数为最长等待的时间，以毫秒为单位，如传入5000就表示5秒，传入0就立即返回，传入INFINITE表示无限等待。

因为线程的句柄在线程运行时是未触发的，线程结束运行，句柄处于触发状态。所以可以用WaitForSingleObject()来等待一个线程结束运行。

函数返回值：

在指定的时间内对象被触发，函数返回WAIT\_OBJECT\_0。超过最长等待时间对象仍未被触发返回WAIT\_TIMEOUT。传入参数有错误将返回WAIT\_FAILED

CreateThread()函数是Windows提供的API接口，在C/C++语言另有一个创建线程的函数\_beginthreadex()，在很多书上（包括《Windows核心编程》）提到过尽量使用\_beginthreadex()来代替使用CreateThread()，这是为什么了？下面就来探索与发现它们的区别吧。

       首先要从标准C运行库与多线程的矛盾说起，标准C运行库在1970年被实现了，由于当时没任何一个操作系统提供对多线程的支持。因此编写标准C运行库的程序员根本没考虑多线程程序使用标准C运行库的情况。比如标准C运行库的全局变量errno。很多运行库中的函数在出错时会将错误代号赋值给这个全局变量，这样可以方便调试。但如果有这样的一个代码片段：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7421759)

1. **if** (system("notepad.exe readme.txt") == -1)
2. {
3. **switch**(errno)
4. {
5. ...//错误处理代码
6. }
7. }

假设某个线程A在执行上面的代码，该线程在调用system()之后且尚未调用switch()语句时另外一个线程B启动了，这个线程B也调用了标准C运行库的函数，不幸的是这个函数执行出错了并将错误代号写入全局变量errno中。这样线程A一旦开始执行switch()语句时，它将访问一个被B线程改动了的errno。这种情况必须要加以避免！因为不单单是这一个变量会出问题，其它像strerror()、strtok()、tmpnam()、gmtime()、asctime()等函数也会遇到这种由多个线程访问修改导致的数据覆盖问题。

为了解决这个问题，Windows操作系统提供了这样的一种解决方案——每个线程都将拥有自己专用的一块内存区域来供标准C运行库中所有有需要的函数使用。而且这块内存区域的创建就是由C/C++运行库函数\_beginthreadex()来负责的。下面列出\_beginthreadex()函数的源代码（我在这份代码中增加了一些注释）以便读者更好的理解\_beginthreadex()函数与CreateThread()函数的区别。

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7421759)

1. //\_beginthreadex源码整理By MoreWindows( http://blog.csdn.net/MoreWindows )
2. \_MCRTIMP **uintptr\_t** \_\_cdecl \_beginthreadex(
3. **void** \*security,
4. unsigned stacksize,
5. unsigned (\_\_CLR\_OR\_STD\_CALL \* initialcode) (**void** \*),
6. **void** \* argument,
7. unsigned createflag,
8. unsigned \*thrdaddr
9. )
10. {
11. \_ptiddata ptd;          //pointer to per-thread data 见注1
12. **uintptr\_t** thdl;         //thread handle 线程句柄
13. unsigned **long** err = 0L; //Return from GetLastError()
14. unsigned dummyid;    //dummy returned thread ID 线程ID号
16. // validation section 检查initialcode是否为NULL
17. \_VALIDATE\_RETURN(initialcode != NULL, EINVAL, 0);
19. //Initialize FlsGetValue function pointer
20. \_\_set\_flsgetvalue();
22. //Allocate and initialize a per-thread data structure for the to-be-created thread.
23. //相当于new一个\_tiddata结构，并赋给\_ptiddata指针。
24. **if** ( (ptd = (\_ptiddata)\_calloc\_crt(1, **sizeof**(**struct** \_tiddata))) == NULL )
25. **goto** error\_return;
27. // Initialize the per-thread data
28. //初始化线程的\_tiddata块即CRT数据区域 见注2
29. \_initptd(ptd, \_getptd()->ptlocinfo);
31. //设置\_tiddata结构中的其它数据，这样这块\_tiddata块就与线程联系在一起了。
32. ptd->\_initaddr = (**void** \*) initialcode; //线程函数地址
33. ptd->\_initarg = argument;              //传入的线程参数
34. ptd->\_thandle = (**uintptr\_t**)(-1);
36. #if defined (\_M\_CEE) || defined (MRTDLL)
37. **if**(!\_getdomain(&(ptd->\_\_initDomain))) //见注3
38. {
39. **goto** error\_return;
40. }
41. #endif  // defined (\_M\_CEE) || defined (MRTDLL)
43. // Make sure non-NULL thrdaddr is passed to CreateThread
44. **if** ( thrdaddr == NULL )//判断是否需要返回线程ID号
45. thrdaddr = &dummyid;
47. // Create the new thread using the parameters supplied by the caller.
48. //\_beginthreadex()最终还是会调用CreateThread()来向系统申请创建线程
49. **if** ( (thdl = (**uintptr\_t**)CreateThread(
50. (LPSECURITY\_ATTRIBUTES)security,
51. stacksize,
52. \_threadstartex,
53. (**LPVOID**)ptd,
54. createflag,
55. (**LPDWORD**)thrdaddr))
56. == (**uintptr\_t**)0 )
57. {
58. err = GetLastError();
59. **goto** error\_return;
60. }
62. //Good return
63. **return**(thdl); //线程创建成功,返回新线程的句柄.
65. //Error return
66. error\_return:
67. //Either ptd is NULL, or it points to the no-longer-necessary block
68. //calloc-ed for the \_tiddata struct which should now be freed up.
69. //回收由\_calloc\_crt()申请的\_tiddata块
70. \_free\_crt(ptd);
71. // Map the error, if necessary.
72. // Note: this routine returns 0 for failure, just like the Win32
73. // API CreateThread, but \_beginthread() returns -1 for failure.
74. //校正错误代号(可以调用GetLastError()得到错误代号)
75. **if** ( err != 0L )
76. \_dosmaperr(err);
77. **return**( (**uintptr\_t**)0 ); //返回值为NULL的效句柄
78. }

讲解下部分代码：

注1．\_ptiddataptd;中的\_ptiddata是个结构体指针。在mtdll.h文件被定义：

      typedefstruct\_tiddata \* \_ptiddata

微软对它的注释为Structure for each thread's data。这是一个非常大的结构体，有很多成员。本文由于篇幅所限就不列出来了。

注2．\_initptd(ptd, \_getptd()->ptlocinfo);微软对这一句代码中的getptd()的说明为：

      /\* return address of per-thread CRT data \*/

      \_ptiddata \_\_cdecl\_getptd(void);

对\_initptd()说明如下：

      /\* initialize a per-thread CRT data block \*/

      void\_\_cdecl\_initptd(\_Inout\_ \_ptiddata \_Ptd,\_In\_opt\_ pthreadlocinfo \_Locale);

注释中的CRT （C Runtime Library）即标准C运行库。

注3．if(!\_getdomain(&(ptd->\_\_initDomain)))中的\_getdomain()函数代码可以在thread.c文件中找到，其主要功能是初始化COM环境。

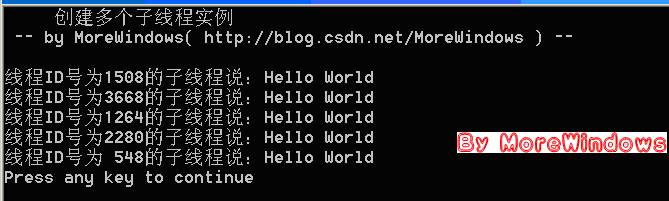
由上面的源代码可知，\_beginthreadex()函数在创建新线程时会分配并初始化一个\_tiddata块。这个\_tiddata块自然是用来存放一些需要线程独享的数据。事实上新线程运行时会首先将\_tiddata块与自己进一步关联起来。然后新线程调用标准C运行库函数如strtok()时就会先取得\_tiddata块的地址再将需要保护的数据存入\_tiddata块中。这样每个线程就只会访问和修改自己的数据而不会去篡改其它线程的数据了。因此，**如果在代码中有使用标准C运行库中的函数时，尽量使用\_beginthreadex()来代替CreateThread()。**相信阅读到这里时，你会对这句简短的话有个非常深刻的印象，如果有面试官问起，你也可以流畅准确的回答了^\_^。

接下来，类似于上面的程序用CreateThread()创建输出“Hello World”的子线程，下面使用\_beginthreadex()来创建多个子线程：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7421759)

1. //创建多子个线程实例
2. #include <stdio.h>
3. #include <process.h>
4. #include <windows.h>
5. //子线程函数
6. unsigned **int** \_\_stdcall ThreadFun(**PVOID** pM)
7. {
8. printf("线程ID号为%4d的子线程说：Hello World\n", GetCurrentThreadId());
9. **return** 0;
10. }
11. //主函数，所谓主函数其实就是主线程执行的函数。
12. **int** main()
13. {
14. printf("     创建多个子线程实例 \n");
15. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
17. **const** **int** THREAD\_NUM = 5;
18. **HANDLE** handle[THREAD\_NUM];
19. **for** (**int** i = 0; i < THREAD\_NUM; i++)
20. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, ThreadFun, NULL, 0, NULL);
21. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
22. **return** 0;
23. }

运行结果如下：

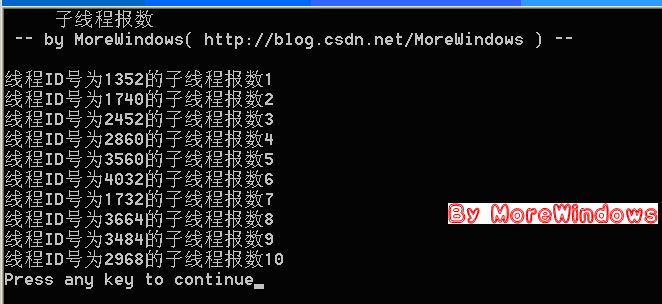


图中每个子线程说的都是同一句话，不太好看。能不能来一个**线程报数**功能，即第一个子线程输出1，第二个子线程输出2，第三个子线程输出3，……。要实现这个功能似乎非常简单——每个子线程对一个全局变量进行递增并输出就可以了。代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7421759)

1. //子线程报数
2. #include <stdio.h>
3. #include <process.h>
4. #include <windows.h>
5. **int** g\_nCount;
6. //子线程函数
7. unsigned **int** \_\_stdcall ThreadFun(**PVOID** pM)
8. {
9. g\_nCount++;
10. printf("线程ID号为%4d的子线程报数%d\n", GetCurrentThreadId(), g\_nCount);
11. **return** 0;
12. }
13. //主函数，所谓主函数其实就是主线程执行的函数。
14. **int** main()
15. {
16. printf("     子线程报数 \n");
17. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
19. **const** **int** THREAD\_NUM = 10;
20. **HANDLE** handle[THREAD\_NUM];
22. g\_nCount = 0;
23. **for** (**int** i = 0; i < THREAD\_NUM; i++)
24. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, ThreadFun, NULL, 0, NULL);
25. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
26. **return** 0;
27. }

对一次运行结果截图如下：



显示结果从1数到10，看起来好象没有问题。

       答案是不对的，虽然这种做法在逻辑上是正确的，但在多线程环境下这样做是会产生严重的问题，下一篇《[秒杀多线程第三篇 原子操作 Interlocked系列函数](http://blog.csdn.net/morewindows/article/details/7429155)》将为你演示错误的结果（可能非常出人意料）并解释产生这个结果的详细原因。

[秒杀多线程第三篇 原子操作 Interlocked系列函数](http://blog.csdn.net/morewindows/article/details/7429155)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-04-09 09:08 59195人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7429155#comments)(122) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7429155#report)

[thread](http://www.csdn.net/tag/thread)[Thread](http://www.csdn.net/tag/Thread)[原子操作](http://www.csdn.net/tag/%e5%8e%9f%e5%ad%90%e6%93%8d%e4%bd%9c)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[汇编](http://www.csdn.net/tag/%e6%b1%87%e7%bc%96)

上一篇《[多线程第一次亲密接触 CreateThread与\_beginthreadex本质区别](http://blog.csdn.net/morewindows/article/details/7421759)》中讲到一个多线程报数功能。为了描述方便和代码简洁起见，我们可以只输出最后的报数结果来观察程序是否运行出错。这也非常类似于统计一个网站每天有多少用户登录，每个用户登录用一个线程模拟，线程运行时会将一个表示计数的变量递增。程序在最后输出计数的值表示有今天多少个用户登录，如果这个值不等于我们启动的线程个数，那显然说明这个程序是有问题的。整个程序代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7429155)

1. #include <stdio.h>
2. #include <process.h>
3. #include <windows.h>
4. **volatile** **long** g\_nLoginCount; //登录次数
5. unsigned **int** \_\_stdcall Fun(**void** \*pPM); //线程函数
6. **const** **int** THREAD\_NUM = 10; //启动线程数
7. unsigned **int** \_\_stdcall ThreadFun(**void** \*pPM)
8. {
9. Sleep(100); //some work should to do
10. g\_nLoginCount++;
11. Sleep(50);
12. **return** 0;
13. }
14. **int** main()
15. {
16. g\_nLoginCount = 0;
18. **HANDLE**  handle[THREAD\_NUM];
19. **for** (**int** i = 0; i < THREAD\_NUM; i++)
20. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, ThreadFun, NULL, 0, NULL);
22. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
23. printf("有%d个用户登录后记录结果是%d\n", THREAD\_NUM, g\_nLoginCount);
24. **return** 0;
25. }

程序中模拟的是10个用户登录，程序将输出结果：

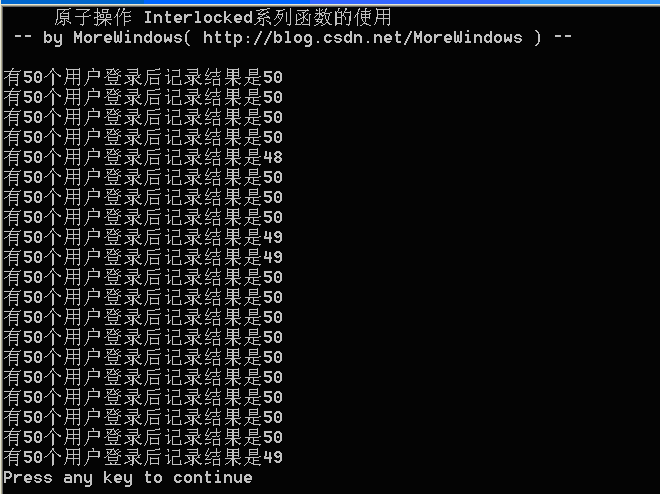


和[上一篇](http://blog.csdn.net/morewindows/article/details/7421759)的线程报数程序一样，程序输出的结果好象并没什么问题。下面我们增加点用户来试试，现在模拟50个用户登录，为了便于观察结果，在程序中将50个用户登录过程重复20次，代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7429155)

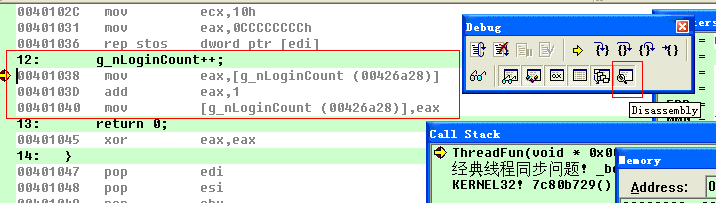
1. #include <stdio.h>
2. #include <windows.h>
3. **volatile** **long** g\_nLoginCount; //登录次数
4. unsigned **int** \_\_stdcall Fun(**void** \*pPM); //线程函数
5. **const** **DWORD** THREAD\_NUM = 50;//启动线程数
6. **DWORD** WINAPI ThreadFun(**void** \*pPM)
7. {
8. Sleep(100); //some work should to do
9. g\_nLoginCount++;
10. Sleep(50);
11. **return** 0;
12. }
13. **int** main()
14. {
15. printf("     原子操作 Interlocked系列函数的使用\n");
16. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
18. //重复20次以便观察多线程访问同一资源时导致的冲突
19. **int** num= 20;
20. **while** (num--)
21. {
22. g\_nLoginCount = 0;
23. **int** i;
24. **HANDLE**  handle[THREAD\_NUM];
25. **for** (i = 0; i < THREAD\_NUM; i++)
26. handle[i] = CreateThread(NULL, 0, ThreadFun, NULL, 0, NULL);
27. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
28. printf("有%d个用户登录后记录结果是%d\n", THREAD\_NUM, g\_nLoginCount);
29. }
30. **return** 0;
31. }

运行结果如下图：



现在结果水落石出，明明有50个线程执行了g\_nLoginCount++;操作，但结果输出是不确定的，有可能为50，但也有可能小于50。

       要解决这个问题，我们就分析下g\_nLoginCount++;操作。在VC6.0编译器对g\_nLoginCount++;这一语句打个断点，再按F5进入调试状态，然后按下Debug工具栏的Disassembly按钮，这样就出现了汇编代码窗口。可以发现在C/C++语言中一条简单的自增语句其实是由三条汇编代码组成的，如下图所示。



讲解下这三条汇编意思：

第一条汇编将g\_nLoginCount的值从内存中读取到寄存器eax中。

第二条汇编将寄存器eax中的值与1相加，计算结果仍存入寄存器eax中。

第三条汇编将寄存器eax中的值写回内存中。

       这样由于线程执行的并发性，很可能线程A执行到第二句时，线程B开始执行，线程B将原来的值又写入寄存器eax中，这样线程A所主要计算的值就被线程B修改了。这样执行下来，结果是不可预知的——可能会出现50，可能小于50。

       因此在多线程环境中对一个变量进行读写时，我们需要有一种方法能够保证对一个值的递增操作是原子操作——即不可打断性，一个线程在执行原子操作时，其它线程必须等待它完成之后才能开始执行该原子操作。这种涉及到硬件的操作会不会很复杂了，幸运的是，Windows系统为我们提供了一些以Interlocked开头的函数来完成这一任务（下文将这些函数称为Interlocked系列函数）。

下面列出一些常用的Interlocked系列函数：

**1.增减操作**

LONG\_\_cdeclInterlockedIncrement(LONG volatile\* Addend);

LONG\_\_cdeclInterlockedDecrement(LONG volatile\* Addend);

返回变量执行增减操作之后的值。

LONG\_\_cdec InterlockedExchangeAdd(LONG volatile\* Addend, LONGValue);

返回运算后的值，注意！加个负数就是减。

**2.赋值操作**

LONG\_\_cdeclInterlockedExchange(LONG volatile\* Target, LONGValue);

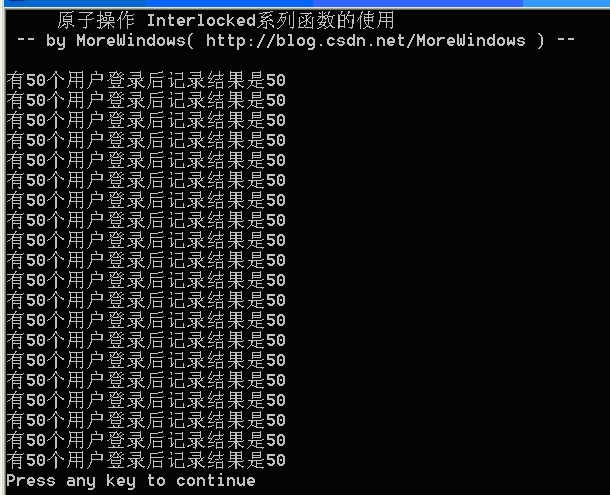
Value就是新值，函数会返回原先的值。

在本例中只要使用InterlockedIncrement()函数就可以了。将线程函数代码改成：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7429155)

1. **DWORD** WINAPI ThreadFun(**void** \*pPM)
2. {
3. Sleep(100);//some work should to do
4. //g\_nLoginCount++;
5. InterlockedIncrement((**LPLONG**)&g\_nLoginCount);
6. Sleep(50);
7. **return** 0;
8. }

再次运行，可以发现结果会是唯一的。



       因此，在多线程环境下，我们对变量的自增自减这些简单的语句也要慎重思考，防止多个线程导致的数据访问出错。更多介绍，请访问MSDN上Synchronization Functions这一章节，地址为 <http://msdn.microsoft.com/zh-cn/library/aa909196.aspx>

看到这里，相信本系列首篇《[秒杀多线程第一篇 多线程笔试面试题汇总](http://blog.csdn.net/morewindows/article/details/7392749)》中选择题第一题（百度笔试题）应该可以秒杀掉了吧（知其然也知其所以然），正确答案是D。另外给个附加问题，程序中是用50个线程模拟用户登录，有兴趣的同学可以试下用100个线程来模拟一下（上机试试绝对会有意外发现^\_^）。

下一篇《秒杀多线程第四篇 一个经典多线程同步问题》将提出一个稍为复杂点但却非常经典的多线程同步互斥问题，这个问题会采用不同的方法来解答，从而让你充分熟练多线程同步互斥的“招式”。更多精彩，欢迎继续参阅。

[秒杀多线程第四篇 一个经典的多线程同步问题](http://blog.csdn.net/morewindows/article/details/7442333)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-04-10 09:57 86558人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7442333#comments)(46) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7442333#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[thread](http://www.csdn.net/tag/thread)[semaphore](http://www.csdn.net/tag/semaphore)[fun](http://www.csdn.net/tag/fun)[null](http://www.csdn.net/tag/null)[windows](http://www.csdn.net/tag/windows)

上一篇《[秒杀多线程第三篇原子操作 Interlocked系列函数](http://blog.csdn.net/morewindows/article/details/7429155)》中介绍了原子操作在多进程中的作用，现在来个复杂点的。这个问题涉及到线程的同步和互斥，是一道非常有代表性的多线程同步问题，如果能将这个问题搞清楚，那么对多线程同步也就打下了良好的基础。

程序描述：

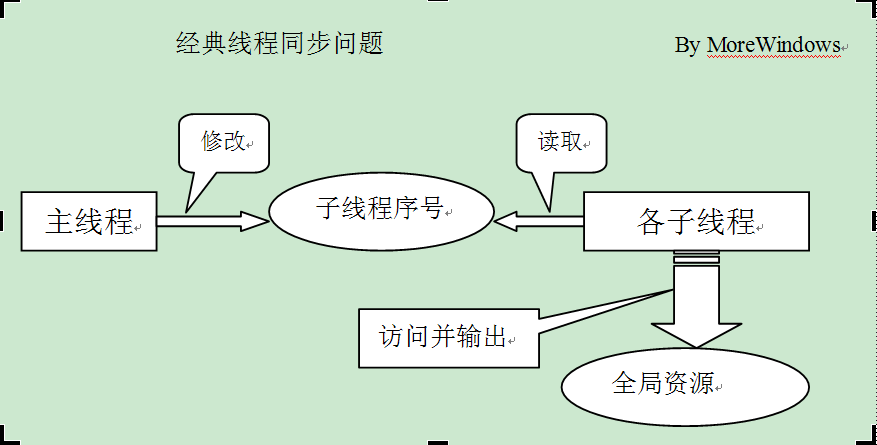
主线程启动10个子线程并将表示子线程序号的变量地址作为参数传递给子线程。子线程接收参数 -> sleep(50) -> 全局变量++ -> sleep(0) -> 输出参数和全局变量。

要求：

1．子线程输出的线程序号不能重复。

2．全局变量的输出必须递增。

下面画了个简单的示意图：



分析下这个问题的考察点，主要考察点有二个：

1．主线程创建子线程并传入一个指向变量地址的指针作参数，由于线程启动须要花费一定的时间，所以在子线程根据这个指针访问并保存数据前，主线程应等待子线程保存完毕后才能改动该参数并启动下一个线程。这涉及到**主线程与子线程之间的同步**。

2．子线程之间会互斥的改动和输出全局变量。要求全局变量的输出必须递增。这涉及到**各子线程间的互斥**。

下面列出这个程序的基本框架，可以在此代码基础上进行修改和验证。

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7442333)

1. //经典线程同步互斥问题
2. #include <stdio.h>
3. #include <process.h>
4. #include <windows.h>
6. **long** g\_nNum; //全局资源
7. unsigned **int** \_\_stdcall Fun(**void** \*pPM); //线程函数
8. **const** **int** THREAD\_NUM = 10; //子线程个数
10. **int** main()
11. {
12. g\_nNum = 0;
13. **HANDLE**  handle[THREAD\_NUM];
15. **int** i = 0;
16. **while** (i < THREAD\_NUM)
17. {
18. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, Fun, &i, 0, NULL);
19. i++;//等子线程接收到参数时主线程可能改变了这个i的值
20. }
21. //保证子线程已全部运行结束
22. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
23. **return** 0;
24. }
26. unsigned **int** \_\_stdcall Fun(**void** \*pPM)
27. {
28. //由于创建线程是要一定的开销的，所以新线程并不能第一时间执行到这来
29. **int** nThreadNum = \*(**int** \*)pPM; //子线程获取参数
30. Sleep(50);//some work should to do
31. g\_nNum++;  //处理全局资源
32. Sleep(0);//some work should to do
33. printf("线程编号为%d  全局资源值为%d\n", nThreadNum, g\_nNum);
34. **return** 0;
35. }

运行结果可以参考下列图示，强烈建议读者亲自试一试。

图1

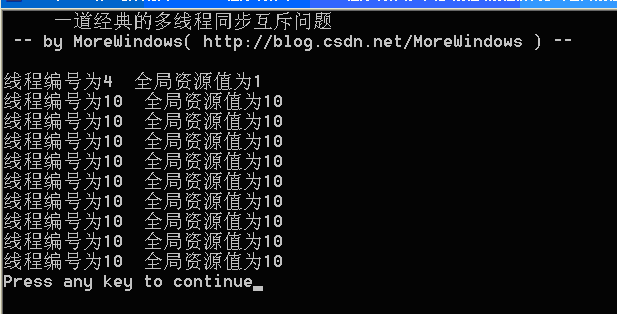


图2

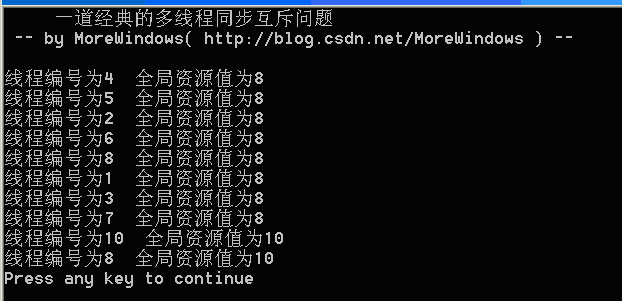


图3



可以看出，运行结果完全是混乱和不可预知的。本系列将会运用Windows平台下各种手段包括关键段，事件，互斥量，信号量等等来解决这个问题并作一份全面的总结，敬请关注。

[秒杀多线程第五篇 经典线程同步 关键段CS](http://blog.csdn.net/morewindows/article/details/7442639)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-04-11 09:06 44097人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7442639#comments)(122) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7442639#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[thread](http://www.csdn.net/tag/thread)[microsoft](http://www.csdn.net/tag/microsoft)[fun](http://www.csdn.net/tag/fun)[null](http://www.csdn.net/tag/null)[struct](http://www.csdn.net/tag/struct)

上一篇《[秒杀多线程第四篇 一个经典的多线程同步问题](http://blog.csdn.net/morewindows/article/details/7442333)》提出了一个经典的多线程同步互斥问题，本篇将用关键段CRITICAL\_SECTION来尝试解决这个问题。

本文首先介绍下如何使用关键段，然后再深层次的分析下关键段的实现机制与原理。

关键段CRITICAL\_SECTION一共就四个函数，使用很是方便。下面是这四个函数的原型和使用说明。

函数功能：初始化

函数原型：

void InitializeCriticalSection(LPCRITICAL\_SECTIONlpCriticalSection);

函数说明：定义关键段变量后必须先初始化。

函数功能：销毁

函数原型：

void DeleteCriticalSection(LPCRITICAL\_SECTIONlpCriticalSection);

函数说明：用完之后记得销毁。

函数功能：进入关键区域

函数原型：

void EnterCriticalSection(LPCRITICAL\_SECTIONlpCriticalSection);

函数说明：系统保证各线程互斥的进入关键区域。

函数功能：离开关关键区域

函数原型：

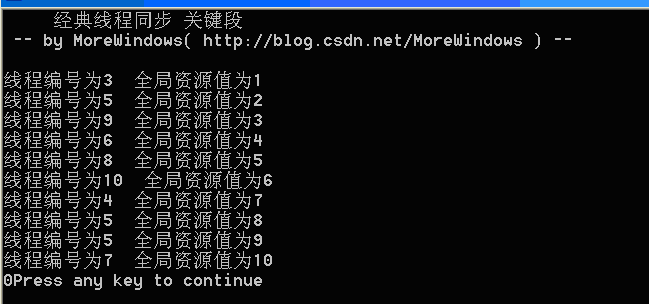
void LeaveCriticalSection(LPCRITICAL\_SECTIONlpCriticalSection);

然后在经典多线程问题中设置二个关键区域。一个是主线程在递增子线程序号时，另一个是各子线程互斥的访问输出全局资源时。详见代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7442639)

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. //关键段变量声明
8. CRITICAL\_SECTION  g\_csThreadParameter, g\_csThreadCode;
9. **int** main()
10. {
11. printf("     经典线程同步 关键段\n");
12. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
14. //关键段初始化
15. InitializeCriticalSection(&g\_csThreadParameter);
16. InitializeCriticalSection(&g\_csThreadCode);
18. **HANDLE**  handle[THREAD\_NUM];
19. g\_nNum = 0;
20. **int** i = 0;
21. **while** (i < THREAD\_NUM)
22. {
23. EnterCriticalSection(&g\_csThreadParameter);//进入子线程序号关键区域
24. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, Fun, &i, 0, NULL);
25. ++i;
26. }
27. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
29. DeleteCriticalSection(&g\_csThreadCode);
30. DeleteCriticalSection(&g\_csThreadParameter);
31. **return** 0;
32. }
33. unsigned **int** \_\_stdcall Fun(**void** \*pPM)
34. {
35. **int** nThreadNum = \*(**int** \*)pPM;
36. LeaveCriticalSection(&g\_csThreadParameter);//离开子线程序号关键区域
38. Sleep(50);//some work should to do
40. EnterCriticalSection(&g\_csThreadCode);//进入各子线程互斥区域
41. g\_nNum++;
42. Sleep(0);//some work should to do
43. printf("线程编号为%d  全局资源值为%d\n", nThreadNum, g\_nNum);
44. LeaveCriticalSection(&g\_csThreadCode);//离开各子线程互斥区域
45. **return** 0;
46. }

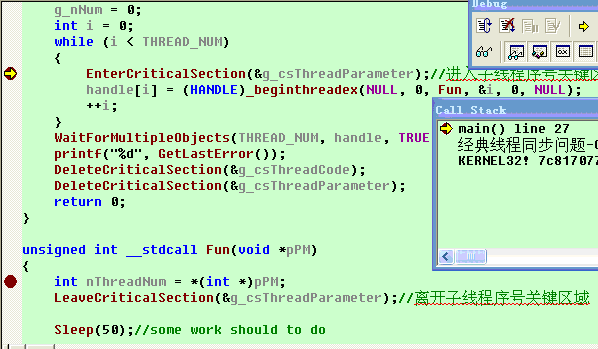
运行结果如下图：



可以看出来，各子线程已经可以互斥的访问与输出全局资源了，但主线程与子线程之间的同步还是有点问题。

       这是为什么了？

要解开这个迷，最直接的方法就是先在程序中加上断点来查看程序的运行流程。断点处置示意如下：



然后按F5进行调试，正常来说这两个断点应该是依次轮流执行，但实际调试时却发现不是如此，主线程可以多次通过第一个断点即

       EnterCriticalSection(&g\_csThreadParameter);//进入子线程序号关键区域

这一语句。这说明主线程能多次进入这个关键区域！找到主线程和子线程没能同步的原因后，下面就来分析下原因的原因吧^\_^

先找到关键段CRITICAL\_SECTION的定义吧，它在WinBase.h中被定义成RTL\_CRITICAL\_SECTION。而RTL\_CRITICAL\_SECTION在WinNT.h中声明，它其实是个结构体：

typedef struct \_RTL\_CRITICAL\_SECTION {

    PRTL\_CRITICAL\_SECTION\_DEBUGDebugInfo;

    LONGLockCount;

    LONGRecursionCount;

    HANDLEOwningThread; // from the thread's ClientId->UniqueThread

    HANDLELockSemaphore;

    DWORDSpinCount;

} RTL\_CRITICAL\_SECTION, \*PRTL\_CRITICAL\_SECTION;

各个参数的解释如下：

第一个参数：PRTL\_CRITICAL\_SECTION\_DEBUGDebugInfo;

调试用的。

第二个参数：LONGLockCount;

初始化为-1，n表示有n个线程在等待。

第三个参数：LONGRecursionCount;

表示该关键段的拥有线程对此资源获得关键段次数，初为0。

第四个参数：HANDLEOwningThread;

即拥有该关键段的线程句柄，微软对其注释为——from the thread's ClientId->UniqueThread

第五个参数：HANDLELockSemaphore;

实际上是一个自复位事件。

第六个参数：DWORDSpinCount;

旋转锁的设置，单CPU下忽略

由这个结构可以知道关键段会记录拥有该关键段的线程句柄即**关键段是有“线程所有权”概念的**。事实上它会用第四个参数OwningThread来记录获准进入关键区域的线程句柄，如果这个线程再次进入，EnterCriticalSection()会更新第三个参数RecursionCount以记录该线程进入的次数并立即返回让该线程进入。其它线程调用EnterCriticalSection()则会被切换到等待状态，一旦拥有线程所有权的线程调用LeaveCriticalSection()使其进入的次数为0时，系统会自动更新关键段并将等待中的线程换回可调度状态。

因此可以将关键段比作旅馆的房卡，调用EnterCriticalSection()即申请房卡，得到房卡后自己当然是可以多次进出房间的，在你调用LeaveCriticalSection()交出房卡之前，别人自然是无法进入该房间。

回到这个经典线程同步问题上，主线程正是由于拥有“线程所有权”即房卡，所以它可以重复进入关键代码区域从而导致子线程在接收参数之前主线程就已经修改了这个参数。所以**关键段可以用于线程间的互斥，但不可以用于同步。**

另外，由于将线程切换到等待状态的开销较大，因此为了提高关键段的性能，Microsoft将旋转锁合并到关键段中，这样EnterCriticalSection()会先用一个旋转锁不断循环，尝试一段时间才会将线程切换到等待状态。下面是配合了旋转锁的关键段初始化函数

函数功能：初始化关键段并设置旋转次数

函数原型：

BOOLInitializeCriticalSectionAndSpinCount(

  LPCRITICAL\_SECTIONlpCriticalSection,

  DWORDdwSpinCount);

函数说明：旋转次数一般设置为4000。

函数功能：修改关键段的旋转次数

函数原型：

DWORDSetCriticalSectionSpinCount(

  LPCRITICAL\_SECTIONlpCriticalSection,

  DWORDdwSpinCount);

《Windows核心编程》第五版的第八章推荐在使用关键段的时候同时使用旋转锁，这样有助于提高性能。值得注意的是如果主机只有一个处理器，那么设置旋转锁是无效的。无法进入关键区域的线程总会被系统将其切换到等待状态。

最后总结下关键段：

**1．关键段共初始化化、销毁、进入和离开关键区域四个函数。**

**2．关键段可以解决线程的互斥问题，但因为具有“线程所有权”，所以无法解决同步问题。**

**3．推荐关键段与旋转锁配合使用。**

下一篇《[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)》将介绍使用事件Event来解决这个经典线程同步问题。

[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)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-04-11 09:06 51490人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7445233#comments)(140) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7445233#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[null](http://www.csdn.net/tag/null)[thread](http://www.csdn.net/tag/thread)[semaphore](http://www.csdn.net/tag/semaphore)[fun](http://www.csdn.net/tag/fun)[access](http://www.csdn.net/tag/access)

阅读本篇之前推荐阅读以下姊妹篇：

《[秒杀多线程第四篇 一个经典的多线程同步问题](http://blog.csdn.net/morewindows/article/details/7442333)》

《[秒杀多线程第五篇 经典线程同步关键段CS》](http://blog.csdn.net/morewindows/article/details/7442639)

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：

HANDLECreateEvent(

 LPSECURITY\_ATTRIBUTESlpEventAttributes,

 BOOLbManualReset,

 BOOLbInitialState,

 LPCTSTRlpName

);

函数说明：

第一个参数表示安全控制，一般直接传入NULL。

第二个参数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。打个小小比方，手动置位事件相当于教室门，教室门一旦打开（被触发），所以有人都可以进入直到老师去关上教室门（事件变成未触发）。自动置位事件就相当于医院里拍X光的房间门，门打开后只能进入一个人，这个人进去后会将门关上，其它人不能进入除非门重新被打开（事件重新被触发）。

第三个参数表示事件的初始状态，传入TRUR表示已触发。

第四个参数表示事件的名称，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：

HANDLEOpenEvent(

 DWORDdwDesiredAccess,

 BOOLbInheritHandle,

 LPCTSTRlpName     //名称

);

函数说明：

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

第二个参数表示事件句柄继承性，一般传入TRUE即可。

第三个参数表示名称，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：BOOLSetEvent(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

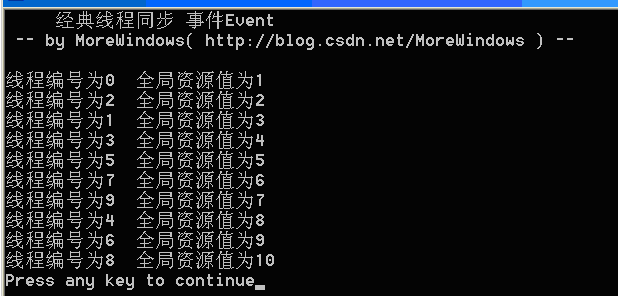
由于事件是内核对象，因此使用CloseHandle()就可以完成清理与销毁了。

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：将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：BOOLPulseEvent(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。

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：

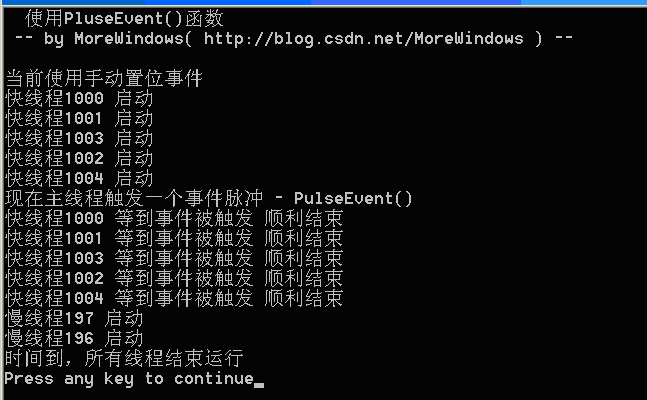
**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7445233)

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。

后面二篇《[秒杀多线程第七篇 经典线程同步 互斥量Mutex](http://blog.csdn.net/morewindows/article/details/7470936)》和《[秒杀多线程第八篇 经典线程同步 信号量Semaphore](http://blog.csdn.net/morewindows/article/details/7481609)》将介绍如何使用互斥量和信号量来解决这个经典线程同步问题。欢迎大家继续秒杀多线程之旅。

[秒杀多线程第七篇 经典线程同步 互斥量Mutex](http://blog.csdn.net/morewindows/article/details/7470936)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-04-18 09:26 53182人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7470936#comments)(95) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7470936#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[semaphore](http://www.csdn.net/tag/semaphore)[null](http://www.csdn.net/tag/null)[thread](http://www.csdn.net/tag/thread)[access](http://www.csdn.net/tag/access)[fun](http://www.csdn.net/tag/fun)

阅读本篇之前推荐阅读以下姊妹篇：

《[秒杀多线程第四篇一个经典的多线程同步问题](http://blog.csdn.net/morewindows/article/details/7442333)》

《[秒杀多线程第五篇经典线程同步关键段CS](http://blog.csdn.net/morewindows/article/details/7442639)》

《[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)》

前面介绍了[关键段CS](http://blog.csdn.net/morewindows/article/details/7442639)、[事件Event](http://blog.csdn.net/morewindows/article/details/7445233)在[经典线程同步问题](http://blog.csdn.net/morewindows/article/details/7442333)中的使用。本篇介绍用互斥量Mutex来解决这个问题。

互斥量也是一个内核对象，它用来确保一个线程独占一个资源的访问。互斥量与关键段的行为非常相似，并且互斥量可以用于不同进程中的线程互斥访问资源。使用互斥量Mutex主要将用到四个函数。下面是这些函数的原型和使用说明。

第一个 CreateMutex

函数功能：创建互斥量（注意与事件Event的创建函数对比）

函数原型：

HANDLECreateMutex(

  LPSECURITY\_ATTRIBUTESlpMutexAttributes,

  BOOLbInitialOwner,

  LPCTSTRlpName

);

函数说明：

第一个参数表示安全控制，一般直接传入NULL。

第二个参数用来确定互斥量的初始拥有者。如果传入TRUE表示互斥量对象内部会记录创建它的线程的线程ID号并将递归计数设置为1，由于该线程ID非零，所以互斥量处于未触发状态。如果传入FALSE，那么互斥量对象内部的线程ID号将设置为NULL，递归计数设置为0，这意味互斥量不为任何线程占用，处于触发状态。

第三个参数用来设置互斥量的名称，在多个进程中的线程就是通过名称来确保它们访问的是同一个互斥量。

函数访问值：

成功返回一个表示互斥量的句柄，失败返回NULL。

第二个打开互斥量

函数原型：

HANDLEOpenMutex(

 DWORDdwDesiredAccess,

 BOOLbInheritHandle,

 LPCTSTRlpName     //名称

);

函数说明：

第一个参数表示访问权限，对互斥量一般传入MUTEX\_ALL\_ACCESS。详细解释可以查看MSDN文档。

第二个参数表示互斥量句柄继承性，一般传入TRUE即可。

第三个参数表示名称。某一个进程中的线程创建互斥量后，其它进程中的线程就可以通过这个函数来找到这个互斥量。

函数访问值：

成功返回一个表示互斥量的句柄，失败返回NULL。

第三个触发互斥量

函数原型：

BOOLReleaseMutex (HANDLEhMutex)

函数说明：

访问互斥资源前应该要调用等待函数，结束访问时就要调用ReleaseMutex()来表示自己已经结束访问，其它线程可以开始访问了。

最后一个清理互斥量

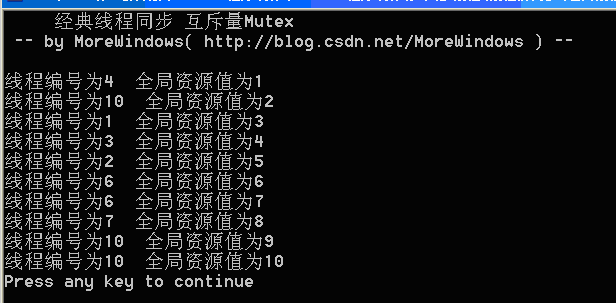
由于互斥量是内核对象，因此使用CloseHandle()就可以（这一点所有内核对象都一样）。

接下来我们就在经典多线程问题用互斥量来保证主线程与子线程之间的同步，由于互斥量的使用函数类似于事件Event，所以可以仿照上一篇的实现来写出代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7470936)

1. //经典线程同步问题 互斥量Mutex
2. #include <stdio.h>
3. #include <process.h>
4. #include <windows.h>
6. **long** g\_nNum;
7. unsigned **int** \_\_stdcall Fun(**void** \*pPM);
8. **const** **int** THREAD\_NUM = 10;
9. //互斥量与关键段
10. **HANDLE**  g\_hThreadParameter;
11. CRITICAL\_SECTION g\_csThreadCode;
13. **int** main()
14. {
15. printf("     经典线程同步 互斥量Mutex\n");
16. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
18. //初始化互斥量与关键段 第二个参数为TRUE表示互斥量为创建线程所有
19. g\_hThreadParameter = CreateMutex(NULL, FALSE, NULL);
20. InitializeCriticalSection(&g\_csThreadCode);
22. **HANDLE**  handle[THREAD\_NUM];
23. g\_nNum = 0;
24. **int** i = 0;
25. **while** (i < THREAD\_NUM)
26. {
27. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, Fun, &i, 0, NULL);
28. WaitForSingleObject(g\_hThreadParameter, INFINITE); //等待互斥量被触发
29. i++;
30. }
31. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
33. //销毁互斥量和关键段
34. CloseHandle(g\_hThreadParameter);
35. DeleteCriticalSection(&g\_csThreadCode);
36. **for** (i = 0; i < THREAD\_NUM; i++)
37. CloseHandle(handle[i]);
38. **return** 0;
39. }
40. unsigned **int** \_\_stdcall Fun(**void** \*pPM)
41. {
42. **int** nThreadNum = \*(**int** \*)pPM;
43. ReleaseMutex(g\_hThreadParameter);//触发互斥量
45. Sleep(50);//some work should to do
47. EnterCriticalSection(&g\_csThreadCode);
48. g\_nNum++;
49. Sleep(0);//some work should to do
50. printf("线程编号为%d  全局资源值为%d\n", nThreadNum, g\_nNum);
51. LeaveCriticalSection(&g\_csThreadCode);
52. **return** 0;
53. }

运行结果如下图：



可以看出，与关键段类似，互斥量也是不能解决线程间的同步问题。

       联想到关键段会记录线程ID即有“线程拥有权”的，而互斥量也记录线程ID，莫非它也有“线程拥有权”这一说法。

       答案确实如此，互斥量也是有“线程拥有权”概念的。“线程拥有权”在[关键段](http://blog.csdn.net/morewindows/article/details/7442639)中有详细的说明，这里就不再赘述了。另外由于互斥量常用于多进程之间的线程互斥，所以它比关键段还多一个很有用的特性——“**遗弃**”情况的处理。比如有一个占用互斥量的线程在调用ReleaseMutex()触发互斥量前就意外终止了（相当于该互斥量被“遗弃”了），那么所有等待这个互斥量的线程是否会由于该互斥量无法被触发而陷入一个无穷的等待过程中了？这显然不合理。因为占用某个互斥量的线程既然终止了那足以证明它不再使用被该互斥量保护的资源，所以这些资源完全并且应当被其它线程来使用。因此在这种“**遗弃**”情况下，系统自动把该互斥量内部的线程ID设置为0，并将它的递归计数器复置为0，表示这个互斥量被触发了。然后系统将“公平地”选定一个等待线程来完成调度（被选中的线程的WaitForSingleObject()会返回WAIT\_ABANDONED\_0）。

下面写二个程序来验证下：

第一个程序创建互斥量并等待用户输入后就触发互斥量。第二个程序先打开互斥量，成功后就等待并根据等待结果作相应的输出。详见代码：

第一个程序：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7470936)

1. #include <stdio.h>
2. #include <conio.h>
3. #include <windows.h>
4. **const** **char** MUTEX\_NAME[] = "Mutex\_MoreWindows";
5. **int** main()
6. {
7. **HANDLE** hMutex = CreateMutex(NULL, TRUE, MUTEX\_NAME); //创建互斥量
8. printf("互斥量已经创建，现在按任意键触发互斥量\n");
9. getch();
10. //exit(0);
11. ReleaseMutex(hMutex);
12. printf("互斥量已经触发\n");
13. CloseHandle(hMutex);
14. **return** 0;
15. }

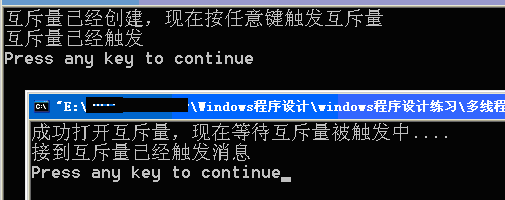
第二个程序：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7470936)

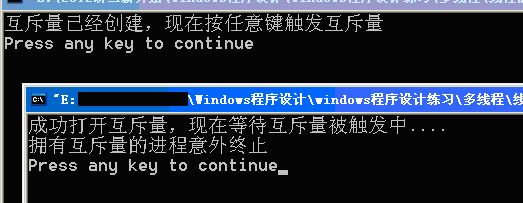
1. #include <stdio.h>
2. #include <windows.h>
3. **const** **char** MUTEX\_NAME[] = "Mutex\_MoreWindows";
4. **int** main()
5. {
6. **HANDLE** hMutex = OpenMutex(MUTEX\_ALL\_ACCESS, TRUE, MUTEX\_NAME); //打开互斥量
7. **if** (hMutex == NULL)
8. {
9. printf("打开互斥量失败\n");
10. **return** 0;
11. }
12. printf("等待中....\n");
13. **DWORD** dwResult = WaitForSingleObject(hMutex, 20 \* 1000); //等待互斥量被触发
14. **switch** (dwResult)
15. {
16. **case** WAIT\_ABANDONED:
17. printf("拥有互斥量的进程意外终止\n");
18. **break**;
20. **case** WAIT\_OBJECT\_0:
21. printf("已经收到信号\n");
22. **break**;
24. **case** WAIT\_TIMEOUT:
25. printf("信号未在规定的时间内送到\n");
26. **break**;
27. }
28. CloseHandle(hMutex);
29. **return** 0;
30. }

运用这二个程序时要先启动程序一再启动程序二。下面展示部分输出结果：

结果一．二个进程顺利执行完毕：



结果二．将程序一中//exit(0);前面的注释符号去掉，这样程序一在触发互斥量之前就会因为执行exit(0);语句而且退出，程序二会收到WAIT\_ABANDONED消息并输出“拥有互斥量的进程意外终止”：



有这个对“**遗弃**”问题的处理，在多进程中的线程同步也可以放心的使用互斥量。

最后总结下互斥量Mutex：

**1．互斥量是内核对象，它与关键段都有“线程所有权”所以不能用于线程的同步。**

**2．互斥量能够用于多个进程之间线程互斥问题，并且能完美的解决某进程意外终止所造成的“遗弃”问题。**

下一篇《[秒杀多线程第八篇 经典线程同步 信号量Semaphore](http://blog.csdn.net/morewindows/article/details/7481609)》将介绍使用信号量Semaphore来解决这个经典线程同步问题。

[秒杀多线程第八篇 经典线程同步 信号量Semaphore](http://blog.csdn.net/morewindows/article/details/7481609)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-05-03 09:30 52017人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7481609#comments)(97) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7481609#report)

[semaphore](http://www.csdn.net/tag/semaphore)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[thread](http://www.csdn.net/tag/thread)[null](http://www.csdn.net/tag/null)[attributes](http://www.csdn.net/tag/attributes)[fun](http://www.csdn.net/tag/fun)

阅读本篇之前推荐阅读以下姊妹篇：

《[秒杀多线程第四篇一个经典的多线程同步问题](http://blog.csdn.net/morewindows/article/details/7442333)》

《[秒杀多线程第五篇经典线程同步关键段CS](http://blog.csdn.net/morewindows/article/details/7442639)》

《[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)》

《[秒杀多线程第七篇经典线程同步互斥量Mutex](http://blog.csdn.net/morewindows/article/details/7470936)》

前面介绍了[关键段CS](http://blog.csdn.net/morewindows/article/details/7442639)、[事件Event](http://blog.csdn.net/morewindows/article/details/7445233)、[互斥量Mutex](http://blog.csdn.net/morewindows/article/details/7470936)在经典线程同步问题中的使用。本篇介绍用信号量Semaphore来解决这个问题。

首先也来看看如何使用信号量，信号量Semaphore常用有三个函数，使用很方便。下面是这几个函数的原型和使用说明。

第一个 CreateSemaphore

函数功能：创建信号量

函数原型：

HANDLE CreateSemaphore(

  LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes,

  LONG lInitialCount,

  LONG lMaximumCount,

  LPCTSTR lpName

);

函数说明：

第一个参数表示安全控制，一般直接传入NULL。

第二个参数表示初始资源数量。

第三个参数表示最大并发数量。

第四个参数表示信号量的名称，传入NULL表示匿名信号量。

第二个 OpenSemaphore

函数功能：打开信号量

函数原型：

HANDLE OpenSemaphore(

  DWORD dwDesiredAccess,

  BOOL bInheritHandle,

  LPCTSTR lpName

);

函数说明：

第一个参数表示访问权限，对一般传入SEMAPHORE\_ALL\_ACCESS。详细解释可以查看MSDN文档。

第二个参数表示信号量句柄继承性，一般传入TRUE即可。

第三个参数表示名称，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信号量。

第三个 ReleaseSemaphore

函数功能：递增信号量的当前资源计数

函数原型：

BOOL ReleaseSemaphore(

  HANDLE hSemaphore,

  LONG lReleaseCount,

  LPLONG lpPreviousCount

);

函数说明：

第一个参数是信号量的句柄。

第二个参数表示增加个数，必须大于0且不超过最大资源数量。

第三个参数可以用来传出先前的资源计数，设为NULL表示不需要传出。

注意：**当前资源数量大于0，表示信号量处于触发，等于0表示资源已经耗尽故信号量处于末触发。**在对信号量调用等待函数时，等待函数会检查信号量的当前资源计数，如果大于0(即信号量处于触发状态)，减1后返回让调用线程继续执行。一个线程可以多次调用等待函数来减小信号量。

最后一个 信号量的清理与销毁

由于信号量是内核对象，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信号量和一个关键段。用信号量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/7481609)

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. //信号量与关键段
8. **HANDLE**            g\_hThreadParameter;
9. CRITICAL\_SECTION  g\_csThreadCode;
10. **int** main()
11. {
12. printf("     经典线程同步 信号量Semaphore\n");
13. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
15. //初始化信号量和关键段
16. g\_hThreadParameter = CreateSemaphore(NULL, 0, 1, NULL);//当前0个资源，最大允许1个同时访问
17. InitializeCriticalSection(&g\_csThreadCode);
19. **HANDLE**  handle[THREAD\_NUM];
20. g\_nNum = 0;
21. **int** i = 0;
22. **while** (i < THREAD\_NUM)
23. {
24. handle[i] = (**HANDLE**)\_beginthreadex(NULL, 0, Fun, &i, 0, NULL);
25. WaitForSingleObject(g\_hThreadParameter, INFINITE);//等待信号量>0
26. ++i;
27. }
28. WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);
30. //销毁信号量和关键段
31. DeleteCriticalSection(&g\_csThreadCode);
32. CloseHandle(g\_hThreadParameter);
33. **for** (i = 0; i < THREAD\_NUM; i++)
34. CloseHandle(handle[i]);
35. **return** 0;
36. }
37. unsigned **int** \_\_stdcall Fun(**void** \*pPM)
38. {
39. **int** nThreadNum = \*(**int** \*)pPM;
40. ReleaseSemaphore(g\_hThreadParameter, 1, NULL);//信号量++
42. Sleep(50);//some work should to do
44. EnterCriticalSection(&g\_csThreadCode);
45. ++g\_nNum;
46. Sleep(0);//some work should to do
47. printf("线程编号为%d  全局资源值为%d\n", nThreadNum, g\_nNum);
48. LeaveCriticalSection(&g\_csThreadCode);
49. **return** 0;
50. }

运行结果如下图：



可以看出来，信号量也可以解决线程之间的同步问题。

由于信号量可以计算资源当前剩余量并根据当前剩余量与零比较来决定信号量是处于触发状态或是未触发状态，因此信号量的应用范围相当广泛。本系列的《秒杀多线程第十篇 生产者消费者问题》将再次使用它来解决线程同步问题，欢迎大家参阅。

至此，经典线程同步问题全部结束了，下一篇《[秒杀多线程第九篇 经典多线程同步问题总结](http://blog.csdn.net/morewindows/article/details/7538247)》将会对其作个总结以梳理各知识点。

# [秒杀多线程第九篇 经典线程同步总结 关键段 事件 互斥量 信号量](http://blog.csdn.net/morewindows/article/details/7538247)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-05-09 10:20 33906人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7538247#comments)(39) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7538247#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[semaphore](http://www.csdn.net/tag/semaphore)[任务](http://www.csdn.net/tag/%e4%bb%bb%e5%8a%a1)

前面《[秒杀多线程第四篇一个经典的多线程同步问题](http://blog.csdn.net/morewindows/article/details/7442333)》提出了一个经典的多线程同步互斥问题，这个问题包括了主线程与子线程的同步，子线程间的互斥，是一道非常经典的多线程同步互斥问题范例，后面分别用了四篇

《[秒杀多线程第五篇经典线程同步关键段CS](http://blog.csdn.net/morewindows/article/details/7442639)》

《[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)》

《[秒杀多线程第七篇经典线程同步互斥量Mutex](http://blog.csdn.net/morewindows/article/details/7470936)》

《[秒杀多线程第八篇经典线程同步信号量Semaphore](http://blog.csdn.net/morewindows/article/details/7481609)》

来详细介绍常用的线程同步互斥机制——关键段、事件、互斥量、信号量。下面对它们作个总结，帮助大家梳理各个知识点。

首先来看下关于线程同步互斥的概念性的知识，相信大家通过前面的文章，已经对线程同步互斥有一定的认识了，也能模糊的说出线程同步互斥的各种概念性知识，下面再列出从《计算机操作系统》一书中选取的一些关于线程同步互斥的描述。相信先有个初步而模糊的印象再看下权威的定义，应该会记忆的特别深刻。

1．线程（进程）同步的主要任务

答：在引入多线程后，由于线程执行的异步性，会给系统造成混乱，特别是在急用临界资源时，如多个线程急用同一台打印机，会使打印结果交织在一起，难于区分。当多个线程急用共享变量，表格，链表时，可能会导致数据处理出错，因此线程同步的主要任务是使并发执行的各线程之间能够有效的共享资源和相互合作，从而使程序的执行具有可再现性。

2．线程（进程）之间的制约关系？

当线程并发执行时，由于资源共享和线程协作，使用线程之间会存在以下两种制约关系。

（1）．间接相互制约。一个系统中的多个线程必然要共享某种系统资源，如共享CPU，共享I/O设备，所谓间接相互制约即源于这种资源共享，打印机就是最好的例子，线程A在使用打印机时，其它线程都要等待。

（2）．直接相互制约。这种制约主要是因为线程之间的合作，如有线程A将计算结果提供给线程B作进一步处理，那么线程B在线程A将数据送达之前都将处于阻塞状态。

间接相互制约可以称为**互斥**，直接相互制约可以称为**同步**，对于互斥可以这样理解，线程A和线程B互斥访问某个资源则它们之间就会产个顺序问题——要么线程A等待线程B操作完毕，要么线程B等待线程操作完毕，这其实就是线程的同步了。因此**同步包括互斥，互斥其实是一种特殊的同步**。

3．临界资源和临界区

在一段时间内只允许一个线程访问的资源就称为临界资源或独占资源，计算机中大多数物理设备，进程中的共享变量等待都是临界资源，它们要求被互斥的访问。每个进程中访问临界资源的代码称为临界区

看完概念性知识，下面用几个表格来帮助大家更好的记忆和运用多线程同步互斥的四个实现方法——关键段、事件、互斥量、信号量。

关键段CS与互斥量Mutex

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 创建或初始化 | 销毁 | 进入互斥区域 | 离开互斥区域 |
| 关键段CS | Initialize-  CriticalSection | Delete-  CriticalSection | Enter-  CriticalSection | Leave-  CriticalSection |
| 互斥量Mutex | CreateMutex | CloseHandle | 等待系列函数如WaitForSingleObject | ReleaseMutex |

关键段与互斥量都有“线程所有权”概念，可以将“线程所有权”理解成旅馆的房卡，在旅馆前台登记名字拥有房卡后是可以多次进出房间的，其它人则无法进入直到你交出房卡。每个线程必须先通过EnterCriticalSection或WaitForSingleObject来尝试获得“线程所有权”才能调用LeaveCriticalSection或ReleaseMutex。否则会调用失败，这就相当于伪造房卡去办理退房手续——由于登记本上没有你的名字所以会被拒绝。

互斥量能很好的处理“遗弃”情况，因此在多进程之间可以放心的使用。

事件Event

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 创建 | 销毁 | 使事件触发 | 使事件未触发 |
| 事件Event | CreateEvent | CloseHandle | SetEvent | ResetEvent |

注意事件的手动置位和自动置位要分清楚，不要混淆了。

信号量Semaphore

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 创建 | 销毁 | 递减计数 | 递增计数 |
| 信号量  Semaphore | Create-  Semaphore | CloseHandle | 等待系列函数如WaitForSingleObject | Release-  Semaphore |

信号量在计数大于0时表示触发状态，调用WaitForSingleObject不会阻塞，等于0表示未触发状态，调用WaitForSingleObject会阻塞直到有其它线程递增了计数。

注意：互斥量，事件，信号量都是内核对象，可以跨进程使用（通过OpenMutex，OpenEvent，OpenSemaphore）。不过为什么只有互斥量能解决“遗弃”情况了，请看《[秒杀多线程第十五篇 关键段,事件,互斥量,信号量的“遗弃”问题](http://blog.csdn.net/morewindows/article/details/7823572)》。

呵呵^\_^，本系列一共使用了六篇文章来讲解了上面三个表格，如果读者能轻松写出这个表格并能解释下各函数的用法，那么对多线程的同步互斥问题也就有了良好的基础。

通过经典线程同步问题的学习，我们已经初步练好了解决多线程同步互斥的各种“招式”，下面再通过学习二个著名的实例《[秒杀多线程第十篇 生产者消费者问题](http://blog.csdn.net/morewindows/article/details/7577591)》和《[秒杀多线程第十一篇 读者写者问题](http://blog.csdn.net/morewindows/article/details/7596034)》来使我们在解决多线程同步时更加熟练。

[秒杀多线程第十篇 生产者消费者问题](http://blog.csdn.net/morewindows/article/details/7577591)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-05-21 10:18 51381人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7577591#comments)(111) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7577591#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[null](http://www.csdn.net/tag/null)[buffer](http://www.csdn.net/tag/buffer)[产品](http://www.csdn.net/tag/%e4%ba%a7%e5%93%81)[semaphore](http://www.csdn.net/tag/semaphore)[output](http://www.csdn.net/tag/output)

    继[经典线程同步问题](http://blog.csdn.net/morewindows/article/details/7442333)之后，我们来看看生产者消费者问题及读者写者问题。生产者消费者问题是一个著名的线程同步问题，该问题描述如下：有一个生产者在生产产品，这些产品将提供给若干个消费者去消费，为了使生产者和消费者能并发执行，在两者之间设置一个具有多个缓冲区的缓冲池，生产者将它生产的产品放入一个缓冲区中，消费者可以从缓冲区中取走产品进行消费，显然生产者和消费者之间必须保持同步，即不允许消费者到一个空的缓冲区中取产品，也不允许生产者向一个已经放入产品的缓冲区中再次投放产品。

    这个生产者消费者题目不仅常用于操作系统的课程设计，也常常在程序员和软件设计师考试中出现。并且在计算机考研的专业课考试中也是一个非常热门的问题。因此现在就针对这个问题进行详细深入的解答。

    首先来简化问题，先假设生产者和消费者都只有一个，且缓冲区也只有一个。这样情况就简便多了。

    第一．从缓冲区取出产品和向缓冲区投放产品必须是互斥进行的。可以用[关键段](http://blog.csdn.net/morewindows/article/details/7442639)和[互斥量](http://blog.csdn.net/morewindows/article/details/7470936)来完成。

    第二．生产者要等待缓冲区为空，这样才可以投放产品，消费者要等待缓冲区不为空，这样才可以取出产品进行消费。并且由于有二个等待过程，所以要用二个[事件](http://blog.csdn.net/morewindows/article/details/7445233)或[信号量](http://blog.csdn.net/morewindows/article/details/7481609)来控制。

    考虑这二点后，代码很容易写出来。另外为了美观起见，将消费者的输出颜色设置为彩色，有关如何在控制台下设置彩色输出请参阅《[VC 控制台颜色设置](http://blog.csdn.net/morewindows/article/details/6789206)》。

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7577591)

1. //1生产者 1消费者 1缓冲区
2. //使用二个事件，一个表示缓冲区空，一个表示缓冲区满。
3. //再使用一个关键段来控制缓冲区的访问
4. #include <stdio.h>
5. #include <process.h>
6. #include <windows.h>
7. //设置控制台输出颜色
8. **BOOL** SetConsoleColor(**WORD** wAttributes)
9. {
10. **HANDLE** hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);
11. **if** (hConsole == INVALID\_HANDLE\_VALUE)
12. **return** FALSE;
13. **return** SetConsoleTextAttribute(hConsole, wAttributes);
14. }
15. **const** **int** END\_PRODUCE\_NUMBER = 10;   //生产产品个数
16. **int** g\_Buffer;                        //缓冲区
17. //事件与关键段
18. CRITICAL\_SECTION g\_cs;
19. **HANDLE** g\_hEventBufferEmpty, g\_hEventBufferFull;
20. //生产者线程函数
21. unsigned **int** \_\_stdcall ProducerThreadFun(**PVOID** pM)
22. {
23. **for** (**int** i = 1; i <= END\_PRODUCE\_NUMBER; i++)
24. {
25. //等待缓冲区为空
26. WaitForSingleObject(g\_hEventBufferEmpty, INFINITE);
28. //互斥的访问缓冲区
29. EnterCriticalSection(&g\_cs);
30. g\_Buffer = i;
31. printf("生产者将数据%d放入缓冲区\n", i);
32. LeaveCriticalSection(&g\_cs);
34. //通知缓冲区有新数据了
35. SetEvent(g\_hEventBufferFull);
36. }
37. **return** 0;
38. }
39. //消费者线程函数
40. unsigned **int** \_\_stdcall ConsumerThreadFun(**PVOID** pM)
41. {
42. **volatile** **bool** flag = **true**;
43. **while** (flag)
44. {
45. //等待缓冲区中有数据
46. WaitForSingleObject(g\_hEventBufferFull, INFINITE);
48. //互斥的访问缓冲区
49. EnterCriticalSection(&g\_cs);
50. SetConsoleColor(FOREGROUND\_GREEN);
51. printf("  消费者从缓冲区中取数据%d\n", g\_Buffer);
52. SetConsoleColor(FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);
53. **if** (g\_Buffer == END\_PRODUCE\_NUMBER)
54. flag = **false**;
55. LeaveCriticalSection(&g\_cs);
57. //通知缓冲区已为空
58. SetEvent(g\_hEventBufferEmpty);
60. Sleep(10); //some other work should to do
61. }
62. **return** 0;
63. }
64. **int** main()
65. {
66. printf("  生产者消费者问题   1生产者 1消费者 1缓冲区\n");
67. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
69. InitializeCriticalSection(&g\_cs);
70. //创建二个自动复位事件，一个表示缓冲区是否为空，另一个表示缓冲区是否已经处理
71. g\_hEventBufferEmpty = CreateEvent(NULL, FALSE, TRUE, NULL);
72. g\_hEventBufferFull = CreateEvent(NULL, FALSE, FALSE, NULL);
74. **const** **int** THREADNUM = 2;
75. **HANDLE** hThread[THREADNUM];
77. hThread[0] = (**HANDLE**)\_beginthreadex(NULL, 0, ProducerThreadFun, NULL, 0, NULL);
78. hThread[1] = (**HANDLE**)\_beginthreadex(NULL, 0, ConsumerThreadFun, NULL, 0, NULL);
79. WaitForMultipleObjects(THREADNUM, hThread, TRUE, INFINITE);
80. CloseHandle(hThread[0]);
81. CloseHandle(hThread[1]);
83. //销毁事件和关键段
84. CloseHandle(g\_hEventBufferEmpty);
85. CloseHandle(g\_hEventBufferFull);
86. DeleteCriticalSection(&g\_cs);
87. **return** 0;
88. }

运行结果如下所示：



可以看出生产者与消费者已经是有序的工作了。

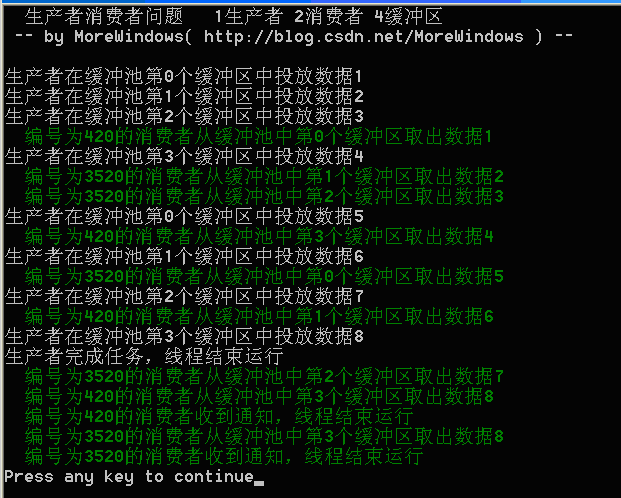
    然后再对这个简单生产者消费者问题加大难度。将消费者改成2个，缓冲池改成拥有4个缓冲区的大缓冲池。

    如何来思考了这个问题了？首先根据上面分析的二点，可以知道生产者和消费者由一个变成多个的影响不大，唯一要注意的是缓冲池变大了，回顾一下《[秒杀多线程第八篇 经典线程同步 信号量Semaphore](http://blog.csdn.net/morewindows/article/details/7481609)》中的信号量，不难得出用二个信号量就可以解决这种缓冲池有多个缓冲区的情况——用一个信号量A来记录为空的缓冲区个数，另一个信号量B记录非空的缓冲区个数，然后生产者等待信号量A，消费者等待信号量B就可以了。因此可以仿照上面的代码来实现复杂生产者消费者问题，示例代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7577591)

1. //1生产者 2消费者 4缓冲区
2. #include <stdio.h>
3. #include <process.h>
4. #include <windows.h>
5. //设置控制台输出颜色
6. **BOOL** SetConsoleColor(**WORD** wAttributes)
7. {
8. **HANDLE** hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);
9. **if** (hConsole == INVALID\_HANDLE\_VALUE)
10. **return** FALSE;
12. **return** SetConsoleTextAttribute(hConsole, wAttributes);
13. }
14. **const** **int** END\_PRODUCE\_NUMBER = 8;  //生产产品个数
15. **const** **int** BUFFER\_SIZE = 4;          //缓冲区个数
16. **int** g\_Buffer[BUFFER\_SIZE];          //缓冲池
17. **int** g\_i, g\_j;
18. //信号量与关键段
19. CRITICAL\_SECTION g\_cs;
20. **HANDLE** g\_hSemaphoreBufferEmpty, g\_hSemaphoreBufferFull;
21. //生产者线程函数
22. unsigned **int** \_\_stdcall ProducerThreadFun(**PVOID** pM)
23. {
24. **for** (**int** i = 1; i <= END\_PRODUCE\_NUMBER; i++)
25. {
26. //等待有空的缓冲区出现
27. WaitForSingleObject(g\_hSemaphoreBufferEmpty, INFINITE);
29. //互斥的访问缓冲区
30. EnterCriticalSection(&g\_cs);
31. g\_Buffer[g\_i] = i;
32. printf("生产者在缓冲池第%d个缓冲区中投放数据%d\n", g\_i, g\_Buffer[g\_i]);
33. g\_i = (g\_i + 1) % BUFFER\_SIZE;
34. LeaveCriticalSection(&g\_cs);
36. //通知消费者有新数据了
37. ReleaseSemaphore(g\_hSemaphoreBufferFull, 1, NULL);
38. }
39. printf("生产者完成任务，线程结束运行\n");
40. **return** 0;
41. }
42. //消费者线程函数
43. unsigned **int** \_\_stdcall ConsumerThreadFun(**PVOID** pM)
44. {
45. **while** (**true**)
46. {
47. //等待非空的缓冲区出现
48. WaitForSingleObject(g\_hSemaphoreBufferFull, INFINITE);
50. //互斥的访问缓冲区
51. EnterCriticalSection(&g\_cs);
52. SetConsoleColor(FOREGROUND\_GREEN);
53. printf("  编号为%d的消费者从缓冲池中第%d个缓冲区取出数据%d\n", GetCurrentThreadId(), g\_j, g\_Buffer[g\_j]);
54. SetConsoleColor(FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);
55. **if** (g\_Buffer[g\_j] == END\_PRODUCE\_NUMBER)//结束标志
56. {
57. LeaveCriticalSection(&g\_cs);
58. //通知其它消费者有新数据了(结束标志)
59. ReleaseSemaphore(g\_hSemaphoreBufferFull, 1, NULL);
60. **break**;
61. }
62. g\_j = (g\_j + 1) % BUFFER\_SIZE;
63. LeaveCriticalSection(&g\_cs);
65. Sleep(50); //some other work to do
67. ReleaseSemaphore(g\_hSemaphoreBufferEmpty, 1, NULL);
68. }
69. SetConsoleColor(FOREGROUND\_GREEN);
70. printf("  编号为%d的消费者收到通知，线程结束运行\n", GetCurrentThreadId());
71. SetConsoleColor(FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);
72. **return** 0;
73. }
74. **int** main()
75. {
76. printf("  生产者消费者问题   1生产者 2消费者 4缓冲区\n");
77. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
79. InitializeCriticalSection(&g\_cs);
80. //初始化信号量,一个记录有产品的缓冲区个数,另一个记录空缓冲区个数.
81. g\_hSemaphoreBufferEmpty = CreateSemaphore(NULL, 4, 4, NULL);
82. g\_hSemaphoreBufferFull  = CreateSemaphore(NULL, 0, 4, NULL);
83. g\_i = 0;
84. g\_j = 0;
85. memset(g\_Buffer, 0, **sizeof**(g\_Buffer));
87. **const** **int** THREADNUM = 3;
88. **HANDLE** hThread[THREADNUM];
89. //生产者线程
90. hThread[0] = (**HANDLE**)\_beginthreadex(NULL, 0, ProducerThreadFun, NULL, 0, NULL);
91. //消费者线程
92. hThread[1] = (**HANDLE**)\_beginthreadex(NULL, 0, ConsumerThreadFun, NULL, 0, NULL);
93. hThread[2] = (**HANDLE**)\_beginthreadex(NULL, 0, ConsumerThreadFun, NULL, 0, NULL);
94. WaitForMultipleObjects(THREADNUM, hThread, TRUE, INFINITE);
95. **for** (**int** i = 0; i < THREADNUM; i++)
96. CloseHandle(hThread[i]);
98. //销毁信号量和关键段
99. CloseHandle(g\_hSemaphoreBufferEmpty);
100. CloseHandle(g\_hSemaphoreBufferFull);
101. DeleteCriticalSection(&g\_cs);
102. **return** 0;
103. }

运行结果如下图所示：



输出结果证明各线程的同步和互斥已经完成了。

至此，生产者消费者问题已经圆满的解决了，下面作个总结：

1．首先要考虑生产者与消费者对缓冲区操作时的互斥。

2．不管生产者与消费者有多少个，缓冲池有多少个缓冲区。都只有二个同步过程——分别是生产者要**等待**有空缓冲区才能投放产品，消费者要**等待**有非空缓冲区才能去取产品。

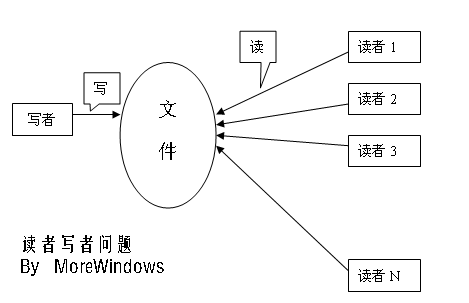
下一篇《秒杀多线程第十一篇读者写者问题》将介绍另一个著名的同步问题——读者写者问题，欢迎大家再来参阅。

[秒杀多线程第十一篇 读者写者问题](http://blog.csdn.net/morewindows/article/details/7596034)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-05-28 09:44 31914人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7596034#comments)(72) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7596034#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[null](http://www.csdn.net/tag/null)[list](http://www.csdn.net/tag/list)[output](http://www.csdn.net/tag/output)[c++](http://www.csdn.net/tag/c%2b%2b)[面试](http://www.csdn.net/tag/%e9%9d%a2%e8%af%95)

与上一篇《[秒杀多线程第十篇 生产者消费者问题](http://blog.csdn.net/morewindows/article/details/7577591)》的生产者消费者问题一样，读者写者也是一个非常著名的同步问题。读者写者问题描述非常简单，有一个写者很多读者，多个读者可以同时读文件，但写者在写文件时不允许有读者在读文件，同样有读者在读文件时写者也不去能写文件。



上面是读者写者问题示意图，类似于[生产者消费者问题](http://blog.csdn.net/morewindows/article/details/7577591)的分析过程，首先来找找哪些是属于“等待”情况。

第一．写者要等到没有读者时才能去写文件。

第二．所有读者要等待写者完成写文件后才能去读文件。

找完“等待”情况后，再看看有没有要互斥访问的资源。由于只有一个写者而读者们是可以共享的读文件，所以按题目要求并没有需要互斥访问的资源。类似于上一篇中美观的彩色输出，我们对生产者输出代码进行了颜色设置（在控制台输出颜色设置参见《[VC 控制台颜色设置](http://blog.csdn.net/morewindows/article/details/6789206)》）。因此在这里要加个互斥访问，不然很有可能在写者线程将控制台颜色设置还原之前，读者线程就已经有输出了。所以要对输出语句作个互斥访问处理，修改后的读者及写者的输出函数如下所示：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7596034)

1. //读者线程输出函数
2. **void** ReaderPrintf(**char** \*pszFormat, ...)
3. {
4. **va\_list**   pArgList;
5. va\_start(pArgList, pszFormat);
6. EnterCriticalSection(&g\_cs);
7. vfprintf(stdout, pszFormat, pArgList);
8. LeaveCriticalSection(&g\_cs);
9. va\_end(pArgList);
10. }
11. //写者线程输出函数
12. **void** WriterPrintf(**char** \*pszStr)
13. {
14. EnterCriticalSection(&g\_cs);
15. SetConsoleColor(FOREGROUND\_GREEN);
16. printf("     %s\n", pszStr);
17. SetConsoleColor(FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);
18. LeaveCriticalSection(&g\_cs);
19. }

读者线程输出函数所使用的可变参数详见《[C,C++中使用可变参数](http://blog.csdn.net/morewindows/article/details/6707662)》。

       解决了互斥输出问题，接下来再考虑如何实现同步问题。可以设置一个变量来记录正在读文件的读者个数，第一个开始读文件的读者要负责将关闭允许写者进入的标志，最后一个结束读文件的读者要负责打开允许写者进入的标志。这样第一种“等待”情况就解决了。第二种“等待”情况是有写者进入时所以读者不能进入，使用一个事件就可以完成这个任务了——所有读者都要等待这个事件而写者负责触发事件和设置事件为未触发。详细见代码中注释：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7596034)

1. //读者与写者问题
2. #include <stdio.h>
3. #include <process.h>
4. #include <windows.h>
5. //设置控制台输出颜色
6. **BOOL** SetConsoleColor(**WORD** wAttributes)
7. {
8. **HANDLE** hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);
9. **if** (hConsole == INVALID\_HANDLE\_VALUE)
10. **return** FALSE;
12. **return** SetConsoleTextAttribute(hConsole, wAttributes);
13. }
14. **const** **int** READER\_NUM = 5;  //读者个数
15. //关键段和事件
16. CRITICAL\_SECTION g\_cs, g\_cs\_writer\_count;
17. **HANDLE** g\_hEventWriter, g\_hEventNoReader;
18. **int** g\_nReaderCount;
19. //读者线程输出函数(变参函数的实现)
20. **void** ReaderPrintf(**char** \*pszFormat, ...)
21. {
22. **va\_list**   pArgList;
24. va\_start(pArgList, pszFormat);
25. EnterCriticalSection(&g\_cs);
26. vfprintf(stdout, pszFormat, pArgList);
27. LeaveCriticalSection(&g\_cs);
28. va\_end(pArgList);
29. }
30. //读者线程函数
31. unsigned **int** \_\_stdcall ReaderThreadFun(**PVOID** pM)
32. {
33. ReaderPrintf("     编号为%d的读者进入等待中...\n", GetCurrentThreadId());
34. //等待写者完成
35. WaitForSingleObject(g\_hEventWriter, INFINITE);
37. //读者个数增加
38. EnterCriticalSection(&g\_cs\_writer\_count);
39. g\_nReaderCount++;
40. **if** (g\_nReaderCount == 1)
41. ResetEvent(g\_hEventNoReader);
42. LeaveCriticalSection(&g\_cs\_writer\_count);
44. //读取文件
45. ReaderPrintf("编号为%d的读者开始读取文件...\n", GetCurrentThreadId());
47. Sleep(rand() % 100);
49. //结束阅读,读者个数减小,空位增加
50. ReaderPrintf(" 编号为%d的读者结束读取文件\n", GetCurrentThreadId());
52. //读者个数减少
53. EnterCriticalSection(&g\_cs\_writer\_count);
54. g\_nReaderCount--;
55. **if** (g\_nReaderCount == 0)
56. SetEvent(g\_hEventNoReader);
57. LeaveCriticalSection(&g\_cs\_writer\_count);
59. **return** 0;
60. }
61. //写者线程输出函数
62. **void** WriterPrintf(**char** \*pszStr)
63. {
64. EnterCriticalSection(&g\_cs);
65. SetConsoleColor(FOREGROUND\_GREEN);
66. printf("     %s\n", pszStr);
67. SetConsoleColor(FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);
68. LeaveCriticalSection(&g\_cs);
69. }
70. //写者线程函数
71. unsigned **int** \_\_stdcall WriterThreadFun(**PVOID** pM)
72. {
73. WriterPrintf("写者线程进入等待中...");
74. //等待读文件的读者为零
75. WaitForSingleObject(g\_hEventNoReader, INFINITE);
76. //标记写者正在写文件
77. ResetEvent(g\_hEventWriter);
79. //写文件
80. WriterPrintf("  写者开始写文件.....");
81. Sleep(rand() % 100);
82. WriterPrintf("  写者结束写文件");
84. //标记写者结束写文件
85. SetEvent(g\_hEventWriter);
86. **return** 0;
87. }
88. **int** main()
89. {
90. printf("  读者写者问题\n");
91. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
93. //初始化事件和信号量
94. InitializeCriticalSection(&g\_cs);
95. InitializeCriticalSection(&g\_cs\_writer\_count);
97. //手动置位,初始已触发
98. g\_hEventWriter = CreateEvent(NULL, TRUE, TRUE, NULL);
99. g\_hEventNoReader  = CreateEvent(NULL, FALSE, TRUE, NULL);
100. g\_nReaderCount = 0;
102. **int** i;
103. **HANDLE** hThread[READER\_NUM + 1];
104. //先启动二个读者线程
105. **for** (i = 1; i <= 2; i++)
106. hThread[i] = (**HANDLE**)\_beginthreadex(NULL, 0, ReaderThreadFun, NULL, 0, NULL);
107. //启动写者线程
108. hThread[0] = (**HANDLE**)\_beginthreadex(NULL, 0, WriterThreadFun, NULL, 0, NULL);
109. Sleep(50);
110. //最后启动其它读者结程
111. **for** ( ; i <= READER\_NUM; i++)
112. hThread[i] = (**HANDLE**)\_beginthreadex(NULL, 0, ReaderThreadFun, NULL, 0, NULL);
113. WaitForMultipleObjects(READER\_NUM + 1, hThread, TRUE, INFINITE);
114. **for** (i = 0; i < READER\_NUM + 1; i++)
115. CloseHandle(hThread[i]);
117. //销毁事件和信号量
118. CloseHandle(g\_hEventWriter);
119. CloseHandle(g\_hEventNoReader);
120. DeleteCriticalSection(&g\_cs);
121. DeleteCriticalSection(&g\_cs\_writer\_count);
122. **return** 0;
123. }

运行结果如下所示：



根据结果可以看出当有读者在读文件时，写者线程会进入等待状态中。当写者线程在写文件时，读者线程也会排队等待，说明读者和写者已经完成了同步。

本系列通过[经典线程同步问题](http://blog.csdn.net/morewindows/article/details/7442333)来列举线程同步手段的[关键段](http://blog.csdn.net/morewindows/article/details/7442639)、[事件](http://blog.csdn.net/morewindows/article/details/7445233)、[互斥量](http://blog.csdn.net/morewindows/article/details/7470936)、[信号量](http://blog.csdn.net/morewindows/article/details/7481609)，并作对这四种方法进行了[总结](http://blog.csdn.net/morewindows/article/details/7538247)。然后又通过二个著名的线程同步实例——[生产者消费者问题](http://blog.csdn.net/morewindows/article/details/7577591)和[读者写者问题](http://blog.csdn.net/morewindows/article/details/7596034)来强化对多线程同步互斥的理解与运用。希望读者们能够熟练掌握，从而在笔试面试中能够顺利的“秒杀”多线程的相关试题，获得自己满意的offer。

从《[秒杀多线程第十篇](http://blog.csdn.net/morewindows/article/details/7577591)生产者消费者问题》到《[秒杀多线程第十一篇](http://blog.csdn.net/morewindows/article/details/7596034)读者写者问题》可以得出多线程问题的关键在于找到所有“等待”情况和判断有无需要互斥访问的资源。那么如何从实际问题中更好更快更全面的找出这些了？请看《[秒杀多线程第十二篇多线程同步内功心法——PV操作上](http://blog.csdn.net/morewindows/article/details/7650470)》和《秒杀多线程第十三篇多线程同步内功心法——PV操作下》这二篇以加强解决多线程同步问题的“内功”。

另外，读者写者问题可以用读写锁SRWLock来解决，请看《[秒杀多线程第十四篇 读者写者问题继 读写锁SRWLock](http://blog.csdn.net/morewindows/article/details/7650574)》

# [秒杀多线程第十二篇 多线程同步内功心法——PV操作上](http://blog.csdn.net/morewindows/article/details/7650470)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-06-11 09:35 23650人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7650470#comments)(30) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7650470#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[semaphore](http://www.csdn.net/tag/semaphore)[apple](http://www.csdn.net/tag/apple)[windows](http://www.csdn.net/tag/windows)[system](http://www.csdn.net/tag/system)[算法](http://www.csdn.net/tag/%e7%ae%97%e6%b3%95)

阅读本篇之前推荐阅读以下姊妹篇：

《[秒杀多线程第四篇一个经典的多线程同步问题](http://blog.csdn.net/morewindows/article/details/7442333)》

《[秒杀多线程第五篇经典线程同步关键段CS](http://blog.csdn.net/morewindows/article/details/7442639)》

《[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)》

《[秒杀多线程第七篇经典线程同步互斥量Mutex](http://blog.csdn.net/morewindows/article/details/7470936)》

《[秒杀多线程第八篇经典线程同步信号量Semaphore](http://blog.csdn.net/morewindows/article/details/7481609)》

《[秒杀多线程第九篇经典线程同步总结关键段事件互斥量信号量](http://blog.csdn.net/morewindows/article/details/7538247)》

《[秒杀多线程第十篇生产者消费者问题](http://blog.csdn.net/morewindows/article/details/7577591)》

《[秒杀多线程第十一篇读者写者问题](http://blog.csdn.net/morewindows/article/details/7596034)》

上面的文章讲解了在Windows系统下实现多线程同步互斥的方法，为了提高在实际问题中分析和思考多个线程之间同步互斥问题的能力，接下来将讲解PV操作，这也是操作系统中的重点和难点。本文将会先简要介绍下PV操作的来源和基本使用方法，然后再通过两道经典的计算机考研真题——放水果和安全岛来示范如何运用PV操作。

先讲讲PV操作的起源和用法。

1962年，荷兰学者Dijksrta在参与X8计算机的开发中设计并实现了具有多道程序运行能力的操作系统——THE Multiprogramming System。为了解决这个操作系统中进程（线程）的同步与互斥问题，他巧妙地利用火车运行控制系统中的“信号灯”（semaphore，或叫“信号量”）概念加以解决。信号量的值大于0时，表示当前可用资源的数量；当它的值小于0时，其绝对值表示等待使用该资源的进程个数。注意，这个信号量的值仅能由PV操作来改变。

PV操作由P操作原语和V操作原语组成（原语也叫原子操作Atomic Operation，是不可中断的过程），对信号量（注意不要和Windows中的[信号量机制](http://blog.csdn.net/morewindows/article/details/7481609)相混淆）进行操作，具体定义如下：

P(S)：

①将信号量S的值减1，即S=S-1；

②如果S>=0，则该进程继续执行；否则该进程置为等待状态。

V(S)：

①将信号量S的值加1，即S=S+1；

②该进程继续执行；如果该信号的等待队列中有等待进程就唤醒一等待进程。

用PV操作实现多线程的同步与互斥是非常简单的，只要考虑逻辑处理上合理严密而不用考虑具体技术细节，因此与写伪代码较为相似。比如有多个进程P1、P2、 ……PN。它们要互斥的访问一个资源。用PV操作来实现就非常方便直观。下面是PV操作代码：

设置信号量为S，初值为1。各进程的操作流程如下：

进程P1              进程P2           ……          进程Pn

P（S）；              P（S）；                           P（S）；

访问资源；         访问资源；                      访问资源；

V（S）；             V（S）；                          V（S）；

**可以看出PV操作会忽略具体的编程细节，让程序员的主要精力放在线程同步互斥的逻辑处理上。因此，通过练习PV操作能快速有效提高程序员对多线程的逻辑思维能力，达到强化“内功”的目的**。

接下来就来几道简单的计算机考研真题。

**第一题 放水果 南京大学计算机考研真题**

桌上有一空盘，允许存放一只水果。爸爸可向盘中放苹果，也可向盘中放桔子，儿子专等吃盘中的桔子，女儿专等吃盘中的苹果。规定当盘空时一次只能放一只水果供吃者取用，请用P、V原语实现爸爸、儿子、女儿三个并发进程的同步。

这个题目涉及的东西非常之多，光人物就有三个再加水果，盘子等等，确实让人感觉好像无从下手。但**不管题目如何变，只要牢牢的抓住同步和互斥来分析问题就必定能迎刃而解。**

下面先考虑同步情况即所有“等待”情况：

第一．爸爸要等待盘子为空。

第二．儿子要等待盘中水果是桔子。

第三．女儿要等待盘中水果是苹果。

接下来来考虑要互斥处理的资源，看起来盘子好像是要作互斥处理的，但由于题目中的爸爸、儿子、女儿均只有一个，并且他们访问盘子的条件都不一样，所以他们根本不会同时去访问盘子，因此盘子也就不用作互斥处理了。分析至些，这个题目已经没有难度了，下面用PV原语给出答案：

先设置三个信号量，信号量Orange表示盘中有桔子，初值为0。信号量Apple表示盘中有苹果，初值为0。信号量EmptyDish表示盘子为空，初值为1。三个人的操作流程如下所示：

1．爸爸

P(EmptyDish)

if (rand()%2==0)

{

    放桔子

    V(Orange)

}

else

{

    放苹果

    V(Apple)

}

2．儿子

P(Orange)

取桔子

V(EmptyDish)

3．女儿

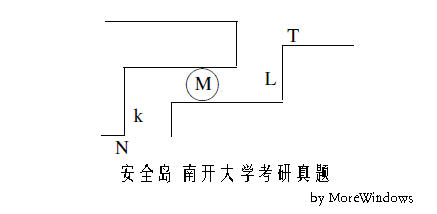
P(Apple)

取苹果

V(EmptyDish)

**第二题 安全岛 南开大学考研真题**

在南开大学至天津大学间有一条弯曲的路，每次只允许一辆自行车通过，但中间有小的安全岛M（同时允许两辆车），可供两辆车在已进入两端小车错车，设计算法并使用P，V实现。



这个问题应该如何考虑了？同样**只要牢牢的抓住同步和互斥来分析问题就必定能迎刃而解。**

考虑所有“等待”情况：

在路口N准备从N到T的人应该什么时候进入了？如果他只判断道路K上有没有人肯定是不行的，因为如果安全岛M上已经有2个人，那么路口N和路口T再各进一人，肯定会造成死锁。因此可以这样——在路口N准备从N到T的人要等待与他同方向的人已经到达T，如果此人已经到达T，且道路K上没有人，他必定可以上路了。同理在路口T准备从T到N的人也应该这样做。

再考虑互斥情况：

路上每次只允许一辆自行车通过，所以道路是需要作互斥处理的。

分析之后，下面就用PV原语给出答案（考研辅导书上的答案）：

设置信号量NT表示在路口N且从N到T方向上允许出发的自行车数量，初值为1。信号量TN表示在路口T且从T到N方向上允许出发的自行车数量，初值为1。信号量K和L表示道路，初值均为1。这样从N到T的车和从T到N的车的行驶流程如下：

**从N到T的车                     从T到N的车**

P(NT)                P(TN)

P(K)                 P(L)

由N到M               由T到M

V(K)                 V(L)

P(L)                 P(K)

由M到T               由M到T

V(L)                 V(K)

V(NT)                V(TN)

这个题目的解法有很多，比如还可以用信号量M来记录安全岛M上空位个数，初值为2。每个进入道路前的人都要先预订安全岛上的空位，订到后再互斥的进入道路。否则就要等待安全岛上有空位。信号量K和L表示道路，初值均为1。然后从N到T的车和从T到N的车的行驶流程如下：

**从N到T的车                     从T到N的车**

P(M)                 P(M)

P(K)                 P(L)

由N到M               由T到M

V(K)                 V(L)

P(L)                 P(K)

V(M)                 V(M)

由M到T               由M到T

V(L)                 V(K)

这种解决方法也是不会造成死锁的。安全岛的解法非常之多，网上还有不少不同的解法，有兴趣的童鞋可以搜索一下。

下一篇《秒杀多线程第十三篇多线程同步内功心法——PV操作下》将讲解更难的一道PV操作题，欢迎大家参阅。

[秒杀多线程第十四篇 读者写者问题继 读写锁SRWLock](http://blog.csdn.net/morewindows/article/details/7650574)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-06-13 09:33 19548人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7650574#comments)(31) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7650574#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[null](http://www.csdn.net/tag/null)[output](http://www.csdn.net/tag/output)[windows](http://www.csdn.net/tag/windows)[server](http://www.csdn.net/tag/server)[c++](http://www.csdn.net/tag/c%2b%2b)

    在《[秒杀多线程第十一篇读者写者问题](http://blog.csdn.net/morewindows/article/details/7596034)》文章中我们使用[事件](http://blog.csdn.net/morewindows/article/details/7445233)和一个记录读者个数的变量来解决读者写者问题。问题虽然得到了解决，但代码有点复杂。本篇将介绍一种新方法——读写锁SRWLock来解决这一问题。读写锁在对资源进行保护的同时，还能区分想要读取资源值的线程（读取者线程）和想要更新资源的线程（写入者线程）。对于读取者线程，读写锁会允许他们并发的执行。当有写入者线程在占有资源时，读写锁会让其它写入者线程和读取者线程等待。因此用读写锁来解决读者写者问题会使代码非常清晰和简洁。

    下面就来看看如何使用读写锁，要注意编译读写锁程序需要VS2008，运行读写锁程序要在Vista或Windows Server2008系统（比这两个更高级的系统也可以）。读写锁的主要函数就五个，分为初始化函数，写入者线程申请和释放函数，读取者线程申请和释放函数，以下是详细的函数使用说明：

第一个 InitializeSRWLock

函数功能：初始化读写锁

函数原型：VOID InitializeSRWLock(PSRWLOCK SRWLock);

函数说明：初始化（没有删除或销毁SRWLOCK的函数，系统会自动清理）

第二个 AcquireSRWLockExclusive

函数功能：写入者线程申请写资源。

函数原型：VOID AcquireSRWLockExclusive(PSRWLOCK SRWLock);

第三个 ReleaseSRWLockExclusive

函数功能：写入者线程写资源完毕，释放对资源的占用。

函数原型：VOID ReleaseSRWLockExclusive(PSRWLOCK SRWLock);

第四个 AcquireSRWLockShared

函数功能：读取者线程申请读资源。

函数原型：VOID AcquireSRWLockShared(PSRWLOCK SRWLock);

第五个 ReleaseSRWLockShared

函数功能：读取者线程结束读取资源，释放对资源的占用。

函数原型：VOID ReleaseSRWLockShared(PSRWLOCK SRWLock);

注意一个线程仅能锁定资源一次，不能多次锁定资源。

使用读写锁精简后的代码如下（代码中变参函数的实现请参阅《[C,C++中使用可变参数](http://blog.csdn.net/morewindows/article/details/6707662)》，控制台颜色设置请参阅《[VC 控制台颜色设置](http://blog.csdn.net/morewindows/article/details/6789206)》）：

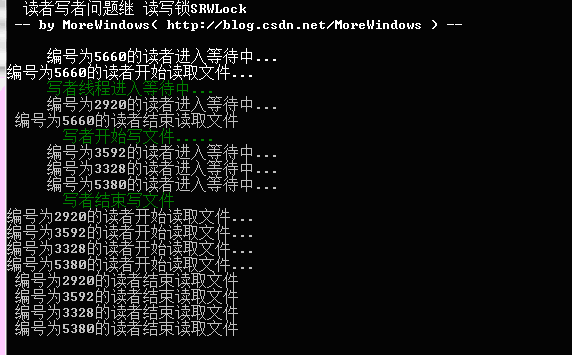
**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7650574)

1. //读者与写者问题继 读写锁SRWLock
2. #include <stdio.h>
3. #include <process.h>
4. #include <windows.h>
5. //设置控制台输出颜色
6. **BOOL** SetConsoleColor(**WORD** wAttributes)
7. {
8. **HANDLE** hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);
9. **if** (hConsole == INVALID\_HANDLE\_VALUE)
10. **return** FALSE;
11. **return** SetConsoleTextAttribute(hConsole, wAttributes);
12. }
13. **const** **int** READER\_NUM = 5;  //读者个数
14. //关键段和事件
15. CRITICAL\_SECTION g\_cs;
16. SRWLOCK          g\_srwLock;
17. //读者线程输出函数(变参函数的实现)
18. **void** ReaderPrintf(**char** \*pszFormat, ...)
19. {
20. **va\_list**   pArgList;
21. va\_start(pArgList, pszFormat);
22. EnterCriticalSection(&g\_cs);
23. vfprintf(stdout, pszFormat, pArgList);
24. LeaveCriticalSection(&g\_cs);
25. va\_end(pArgList);
26. }
27. //读者线程函数
28. unsigned **int** \_\_stdcall ReaderThreadFun(**PVOID** pM)
29. {
30. ReaderPrintf("     编号为%d的读者进入等待中...\n", GetCurrentThreadId());
31. //读者申请读取文件
32. AcquireSRWLockShared(&g\_srwLock);
34. //读取文件
35. ReaderPrintf("编号为%d的读者开始读取文件...\n", GetCurrentThreadId());
36. Sleep(rand() % 100);
37. ReaderPrintf(" 编号为%d的读者结束读取文件\n", GetCurrentThreadId());
39. //读者结束读取文件
40. ReleaseSRWLockShared(&g\_srwLock);
41. **return** 0;
42. }
43. //写者线程输出函数
44. **void** WriterPrintf(**char** \*pszStr)
45. {
46. EnterCriticalSection(&g\_cs);
47. SetConsoleColor(FOREGROUND\_GREEN);
48. printf("     %s\n", pszStr);
49. SetConsoleColor(FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);
50. LeaveCriticalSection(&g\_cs);
51. }
52. //写者线程函数
53. unsigned **int** \_\_stdcall WriterThreadFun(**PVOID** pM)
54. {
55. WriterPrintf("写者线程进入等待中...");
56. //写者申请写文件
57. AcquireSRWLockExclusive(&g\_srwLock);
59. //写文件
60. WriterPrintf("  写者开始写文件.....");
61. Sleep(rand() % 100);
62. WriterPrintf("  写者结束写文件");
64. //标记写者结束写文件
65. ReleaseSRWLockExclusive(&g\_srwLock);
66. **return** 0;
67. }
68. **int** main()
69. {
70. printf("  读者写者问题继 读写锁SRWLock\n");
71. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
73. //初始化读写锁和关键段
74. InitializeCriticalSection(&g\_cs);
75. InitializeSRWLock(&g\_srwLock);
77. **HANDLE** hThread[READER\_NUM + 1];
78. **int** i;
79. //先启动二个读者线程
80. **for** (i = 1; i <= 2; i++)
81. hThread[i] = (**HANDLE**)\_beginthreadex(NULL, 0, ReaderThreadFun, NULL, 0, NULL);
82. //启动写者线程
83. hThread[0] = (**HANDLE**)\_beginthreadex(NULL, 0, WriterThreadFun, NULL, 0, NULL);
84. Sleep(50);
85. //最后启动其它读者结程
86. **for** ( ; i <= READER\_NUM; i++)
87. hThread[i] = (**HANDLE**)\_beginthreadex(NULL, 0, ReaderThreadFun, NULL, 0, NULL);
88. WaitForMultipleObjects(READER\_NUM + 1, hThread, TRUE, INFINITE);
89. **for** (i = 0; i < READER\_NUM + 1; i++)
90. CloseHandle(hThread[i]);
92. //销毁关键段
93. DeleteCriticalSection(&g\_cs);
94. **return** 0;
95. }

对比下《[秒杀多线程第十一篇读者写者问题](http://blog.csdn.net/morewindows/article/details/7596034)》中的代码就可以发现这份代码确实清爽许多了。这个程序用VS2008编译可以通过，但在XP系统下运行会导致报错。



在Win7系统下能够正确的运行，结果如图所示：



最后总结一下读写锁SRWLock

1．读写锁声明后要初始化，但不用销毁，系统会自动清理读写锁。

2．读取者和写入者分别调用不同的申请函数和释放函数。

[秒杀多线程第十五篇 关键段,事件,互斥量,信号量的“遗弃”问题](http://blog.csdn.net/morewindows/article/details/7823572)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2012-08-06 11:16 18095人阅读 [评论](http://blog.csdn.net/morewindows/article/details/7823572#comments)(17) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/7823572#report)

[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[semaphore](http://www.csdn.net/tag/semaphore)[null](http://www.csdn.net/tag/null)[access](http://www.csdn.net/tag/access)[object](http://www.csdn.net/tag/object)[thread](http://www.csdn.net/tag/thread)

目录[(?)[+]](http://blog.csdn.net/morewindows/article/details/7823572)

**秒杀多线程第十五篇 关键段,事件,互斥量,信号量的“遗弃”问题**

在《[秒杀多线程第九篇 经典线程同步总结 关键段 事件 互斥量 信号量](http://blog.csdn.net/morewindows/article/details/7538247)》中对[经典多线程同步互斥问题](http://blog.csdn.net/morewindows/article/details/7442333)进行了回顾和总结，这篇文章对Windows系统下常用的线程同步互斥机制——[关键段](http://blog.csdn.net/morewindows/article/details/7442639)、[事件](http://blog.csdn.net/morewindows/article/details/7445233)、[互斥量](http://blog.csdn.net/morewindows/article/details/7470936)、[信号量](http://blog.csdn.net/morewindows/article/details/7481609)进行了总结。有网友问到互斥量能处理“遗弃”问题，事件和信号量是否也能处理“遗弃”问题。因此本文将对事件和信号量作个试验，看看事件和信号量能否处理“遗弃”问题。

**一.什么是“遗弃”问题**

在《[秒杀多线程第七篇 经典线程同步 互斥量Mutex](http://blog.csdn.net/morewindows/article/details/7470936)》讲到了互斥量能处理“遗弃”问题，下面引用原文：

互斥量常用于多进程之间的线程互斥，所以它比关键段还多一个很有用的特性——“**遗弃**”情况的处理。比如有一个占用互斥量的线程在调用ReleaseMutex()触发互斥量前就意外终止了（相当于该互斥量被“遗弃”了），那么所有等待这个互斥量的线程是否会由于该互斥量无法被触发而陷入一个无穷的等待过程中了？这显然不合理。因为占用某个互斥量的线程既然终止了那足以证明它不再使用被该互斥量保护的资源，所以这些资源完全并且应当被其它线程来使用。因此在这种“**遗弃**”情况下，系统自动把该互斥量内部的线程ID设置为0，并将它的递归计数器复置为0，表示这个互斥量被触发了。然后系统将“公平地”选定一个等待线程来完成调度（被选中的线程的WaitForSingleObject()会返回WAIT\_ABANDONED\_0）。

可见“遗弃”问题就是——**占有某种资源的进程意外终止后，其它等待该资源的进程能否感知。**

**二．关键段的“遗弃”问题**

关键段在这个问题上很简单——由于关键段不能跨进程使用，所以关键段不需要处理“遗弃”问题。

**三．事件，互斥量，信号量的“遗弃”问题**

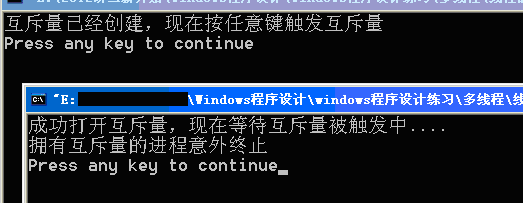
事件，互斥量，信号量都是内核对象，可以跨进程使用。一个进程在创建一个命名的事件后，其它进程可以调用OpenEvent()并传入事件的名称来获得这个事件的句柄。因此事件，互斥量和信号量都会遇到“遗弃”问题。我们已经知道互斥量能够处理“遗弃”问题，接下来就来看看事件和信号量是否能够处理“遗弃”问题。类似于《[秒杀多线程第七篇 经典线程同步互斥量Mutex](http://blog.csdn.net/morewindows/article/details/7470936)》对互斥量所做的试验，下面也对事件和信号量作同样的试验：

1． 创建二个进程。

2． 进程一创建一个初始为未触发的事件，然后等待按键，按下y则触发事件后结束进程，否则直接退出表示进程一已意外终止。

3． 进程二先获得事件的句柄，然后调用WaitForSingleObject()等待这个事件10秒，在这10秒内如果事件已经触发则输出“已收到信号”，否则输出“未在规定的时间内收到信号”。如果在等待的过程中进程一意外终止，则输出“拥有事件的进程意外终止”。信号量的试验方法类似。

为了加强对比效果，将互斥量的试验结果先展示出来（代码请参见《[秒杀多线程第七篇经典线程同步 互斥量Mutex](http://blog.csdn.net/morewindows/article/details/7470936)》）



可以看出在第一个进程在没有触发互斥量就直接退出的情况下，等待这个互斥量的第二个进程是能够感知进程一所发生的意外终止的。

接下来就先完成事件的“遗弃”问题试验代码。

进程一：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7823572)

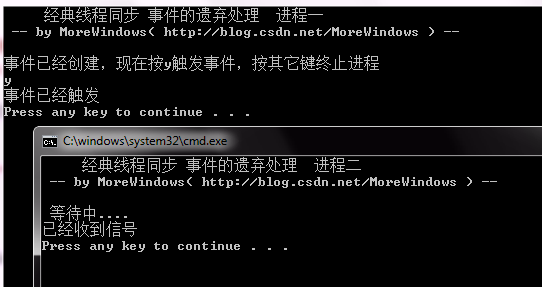
1. #include <stdio.h>
2. #include <conio.h>
3. #include <windows.h>
4. **const** **TCHAR** STR\_EVENT\_NAME[] = TEXT("Event\_MoreWindows");
5. **int** main()
6. {
7. printf("     经典线程同步 事件的遗弃处理  进程一\n");
8. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
9. **HANDLE** hEvent = CreateEvent(NULL, FALSE, FALSE, STR\_EVENT\_NAME);//自动置位 当前未触发
10. printf("事件已经创建，现在按y触发事件，按其它键终止进程\n");
11. **char** ch;
12. scanf("%c", &ch);
13. **if** (ch != 'y')
14. exit(0); //表示进程意外终止
15. SetEvent(hEvent);
16. printf("事件已经触发\n");
17. CloseHandle(hEvent);
18. **return** 0;
19. }

进程二：

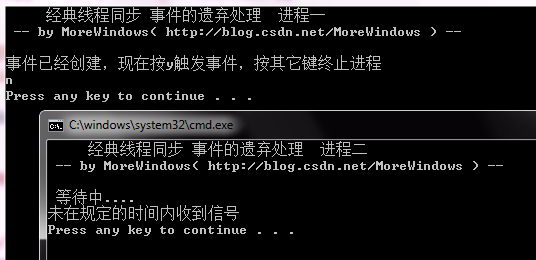
**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7823572)

1. #include <stdio.h>
2. #include <windows.h>
3. **const** **TCHAR** STR\_EVENT\_NAME[] = TEXT("Event\_MoreWindows");
4. **int** main()
5. {
6. printf("     经典线程同步 事件的遗弃处理  进程二\n");
7. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
9. **HANDLE** hEvent = OpenEvent(EVENT\_ALL\_ACCESS, TRUE, STR\_EVENT\_NAME); //打开事件
10. **if** (hEvent == NULL)
11. {
12. printf("打开事件失败\n");
13. **return** 0;
14. }
15. printf(" 等待中....\n");
16. **DWORD** dwResult = WaitForSingleObject(hEvent, 10 \* 1000); //等待事件被触发
17. **switch** (dwResult)
18. {
19. **case** WAIT\_ABANDONED:
20. printf("拥有事件的进程意外终止\n");
21. **break**;
23. **case** WAIT\_OBJECT\_0:
24. printf("已经收到信号\n");
25. **break**;
27. **case** WAIT\_TIMEOUT:
28. printf("未在规定的时间内收到信号\n");
29. **break**;
30. }
31. CloseHandle(hEvent);
32. **return** 0;
33. }

事件Event试验结果1-进程一触发事件后正常结束：



事件Event试验结果2-进程一意外终止：



可以看出进程二没能感知进程一意外终止，说明事件不能处理“遗弃”问题。

下面再来试下信号量。

信号量的“遗弃”问题试验代码：

进程一：

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7823572)

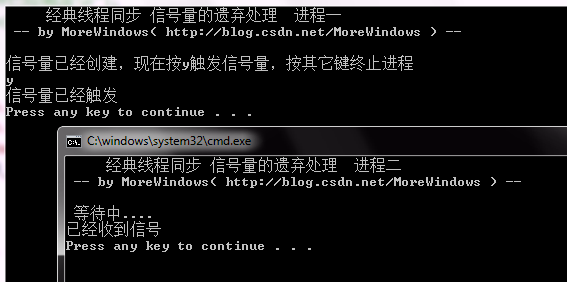
1. #include <stdio.h>
2. #include <conio.h>
3. #include <windows.h>
4. **const** **TCHAR** STR\_SEMAPHORE\_NAME[] = TEXT("Semaphore\_MoreWindows");
5. **int** main()
6. {
7. printf("     经典线程同步 信号量的遗弃处理  进程一\n");
8. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
10. **HANDLE** hSemaphore = CreateSemaphore(NULL, 0, 1, STR\_SEMAPHORE\_NAME);//当前0个资源，最大允许1个同时访问
11. printf("信号量已经创建，现在按y触发信号量，按其它键终止进程\n");
12. **char** ch;
13. scanf("%c", &ch);
14. **if** (ch != 'y')
15. exit(0); //表示进程意外终止
16. ReleaseSemaphore(hSemaphore, 1, NULL);
17. printf("信号量已经触发\n");
18. CloseHandle(hSemaphore);
19. **return** 0;
20. }

进程二：

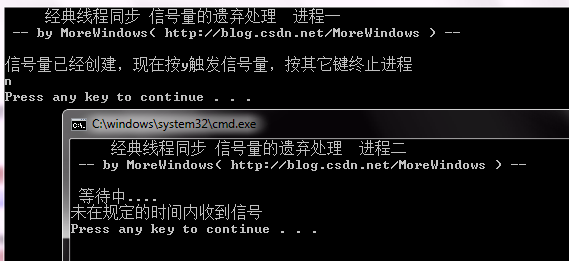
**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/7823572)

1. #include <stdio.h>
2. #include <windows.h>
3. **const** **TCHAR** STR\_SEMAPHORE\_NAME[] = TEXT("Semaphore\_MoreWindows");
4. **int** main()
5. {
6. printf("     经典线程同步 信号量的遗弃处理  进程二\n");
7. printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");
9. **HANDLE** hSemaphore = OpenSemaphore (SEMAPHORE\_ALL\_ACCESS, TRUE, STR\_SEMAPHORE\_NAME); //打开信号量
10. **if** (hSemaphore == NULL)
11. {
12. printf("打开信号量失败\n");
13. **return** 0;
14. }
15. printf(" 等待中....\n");
16. **DWORD** dwResult = WaitForSingleObject(hSemaphore, 10 \* 1000); //等待信号量被触发
17. **switch** (dwResult)
18. {
19. **case** WAIT\_ABANDONED:
20. printf("拥有信号量的进程意外终止\n");
21. **break**;
23. **case** WAIT\_OBJECT\_0:
24. printf("已经收到信号\n");
25. **break**;
27. **case** WAIT\_TIMEOUT:
28. printf("未在规定的时间内收到信号\n");
29. **break**;
30. }
31. CloseHandle(hSemaphore);
32. **return** 0;
33. }

信号量Semaphore试验结果1-进程一触发信号量后正常结束



信号量Semaphore试验结果2-进程一意外终止



可以看出进程二没能感知进程一意外终止，说明信号量与事件一样都不能处理“遗弃”问题。

**四．“遗弃”问题总结**

由本文所做的试验可知，互斥量能够处理“遗弃”情况，事件与信号量都无法解决这一情况。

再思考下互斥量能处理“遗弃”问题的原因，其实正是因为它有“线程所有权”概念。在系统中一旦有线程结束后，系统会判断是否有互斥量被这个线程占有，如果有，系统会将这互斥量对象内部的线程ID号将设置为NULL，递归计数设置为0，这表示该互斥量已经不为任何线程占用，处于触发状态。其它等待这个互斥量的线程就能顺利执行下去了。至于线程如何获取互斥量的“线程所有权”，MSDN上介绍为——A thread obtainsownership of a mutex either by creating it with the *bInitialOwner*parameter set to **TRUE** or by specifying its handle in a call toone of the [wait functions](http://msdn.microsoft.com/en-us/library/windows/desktop/ms687069(v=vs.85).aspx).

[秒杀多线程第十六篇 多线程十大经典案例之一 双线程读写队列数据](http://blog.csdn.net/morewindows/article/details/8646902)

分类： [Windows多线程](http://blog.csdn.net/MoreWindows/article/category/1115271) [Windows编程](http://blog.csdn.net/MoreWindows/article/category/862060)2013-03-13 02:28 32674人阅读 [评论](http://blog.csdn.net/morewindows/article/details/8646902#comments)(43) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/morewindows/article/details/8646902#report)

[Windows编程](http://www.csdn.net/tag/Windows%e7%bc%96%e7%a8%8b)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[多线程同步](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b%e5%90%8c%e6%ad%a5)[十大经典案例](http://www.csdn.net/tag/%e5%8d%81%e5%a4%a7%e7%bb%8f%e5%85%b8%e6%a1%88%e4%be%8b)

目录[(?)[+]](http://blog.csdn.net/morewindows/article/details/8646902)

本文配套程序下载地址为：<http://download.csdn.net/detail/morewindows/5136035>

转载请标明出处，原文地址：<http://blog.csdn.net/morewindows/article/details/8646902>

欢迎关注微博：<http://weibo.com/MoreWindows>

在《[秒杀多线程系列](http://blog.csdn.net/column/details/killthreadseries.html)》的前十五篇中介绍[多线程的相关概念](http://blog.csdn.net/morewindows/article/details/7421759)，多线程同步互斥问题《[秒杀多线程第四篇一个经典的多线程同步问题](http://blog.csdn.net/morewindows/article/details/7442333)》及解决多线程同步互斥的常用方法——[关键段](http://blog.csdn.net/morewindows/article/details/7442639)、[事件](http://blog.csdn.net/morewindows/article/details/7445233)、[互斥量](http://blog.csdn.net/morewindows/article/details/7470936)、[信号量](http://blog.csdn.net/morewindows/article/details/7481609)、[读写锁](http://blog.csdn.net/morewindows/article/details/7650574)。为了让大家更加熟练运用多线程，将会有十篇文章来讲解十个多线程使用案例，相信看完这十篇后会让你能更加游刃有余的使用多线程。

首先来看第一篇——《秒杀多线程第十六篇 多线程十大经典案例之一 双线程读写队列数据》

**《多线程十大经典案例之一双线程读写队列数据》案例描述：**

MFC对话框中一个按钮的响应函数实现两个功能：  
显示数据同时处理数据，因此开两个线程，一个线程显示数据（开了一个定时器，响应WM\_TIMER消息按照一定时间间隔向TeeChart图表添加数据并显示）同时在队列队尾添加数据，另一个线程从该队列队头去数据来处理。

本案例来源于<http://bbs.csdn.net/topics/390383114>，感谢hehening88提供题目，特此鸣谢。

下面就来解决这个案例。先来分析下。

**《多线程十大经典案例之一双线程读写队列数据》案例分析：**

这个案例是一个线程向队列中的队列头部读取数据，一个线程向队列中的队列尾部写入数据。看起来很像读者写者问题（见《[秒杀多线程第十一篇读者写者问题](http://blog.csdn.net/morewindows/article/details/7596034)》和《[秒杀多线程第十四篇读者写者问题继读写锁SRWLock](http://blog.csdn.net/morewindows/article/details/7650574)》），但其实不然，如果将队列看成缓冲区，这个案例明显是个生产者消费者问题（见《[秒杀多线程第十篇生产者消费者问题](http://blog.csdn.net/morewindows/article/details/7577591)》）。因此我们仿照生产者消费者的思路来具体分析下案例中的“等待”情况：

    1.     当队列为空时，读取数据线程必须等待写入数据向队列中写入数据。也就是说**当队列为空时，读取数据线程要等待队列中有数据**。

    2.     当队列满时，写入数据线程必须等待读取数据线程向队列中读取数据。也就是说**当队列满时，写入数据线程要等待队列中有空位**。

在访问队列时，需要互斥吗？**这将依赖于队列的数据结构实现**，如果使用STL中的vector，由于vector会动态增长。因此要做互斥保护。如果使用循环队列，那么读取数据线程拥有读取指针，写入数据线程拥有写入指针，各自将访问队列中不同位置上的数据，因此不用进行互斥保护。

分析完毕后，再来考虑使用什么样的数据结构，同样依照《[秒杀多线程第十篇生产者消费者问题](http://blog.csdn.net/morewindows/article/details/7577591)》中的做法。使用两个信号量，一个来记录循环队列中空位的个数，一个来记录循环队列中产品的个数（非空位个数）。代码非常容易写出，下面给出完整的源代码。

代码中的信号量相关函数可以参考《[秒杀多线程第八篇经典线程同步信号量Semaphore](http://blog.csdn.net/morewindows/article/details/7481609)》，代码中的SetConsoleColor是用来改变控制台的文字颜色，具体可以参考《[VC 控制台颜色设置](http://blog.csdn.net/morewindows/article/details/6789206)》。

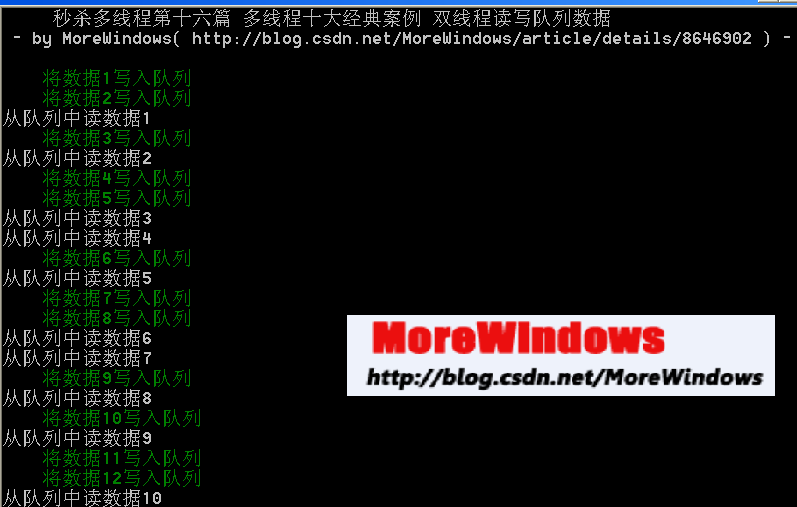
**《多线程十大经典案例之一双线程读写队列数据》完整代码：**

**[cpp]** [view plaincopy](http://blog.csdn.net/morewindows/article/details/8646902)

1. //秒杀多线程第十六篇 多线程十大经典案例之一 双线程读写队列数据
2. //http://blog.csdn.net/MoreWindows/article/details/8646902
3. #include <stdio.h>
4. #include <process.h>
5. #include <windows.h>
6. #include <time.h>
7. **const** **int** QUEUE\_LEN = 5;
8. **int** g\_arrDataQueue[QUEUE\_LEN];
9. **int** g\_i, g\_j, g\_nDataNum;
10. //关键段 用于保证互斥的在屏幕上输出
11. CRITICAL\_SECTION g\_cs;
12. //信号量 g\_hEmpty表示队列中空位 g\_hFull表示队列中非空位
13. **HANDLE**     g\_hEmpty, g\_hFull;
14. //设置控制台输出颜色
15. **BOOL** SetConsoleColor(**WORD** wAttributes)
16. {
17. **HANDLE** hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);
18. **if** (hConsole == INVALID\_HANDLE\_VALUE)
19. **return** FALSE;
20. **return** SetConsoleTextAttribute(hConsole, wAttributes);
21. }
22. //读数据线程函数
23. unsigned **int** \_\_stdcall ReaderThreadFun(**PVOID** pM)
24. {
25. **int** nData = 0;
26. **while** (nData < 20)
27. {
28. WaitForSingleObject(g\_hFull, INFINITE);
29. nData = g\_arrDataQueue[g\_i];
30. g\_i = (g\_i + 1) % QUEUE\_LEN;
31. EnterCriticalSection(&g\_cs);
32. printf("从队列中读数据%d\n", nData);
33. LeaveCriticalSection(&g\_cs);
34. Sleep(rand() % 300);
35. ReleaseSemaphore(g\_hEmpty, 1, NULL);
36. }
37. **return** 0;
38. }
39. //写数据线程函数
40. unsigned **int** \_\_stdcall WriterThreadFun(**PVOID** pM)
41. {
42. **int** nData = 0;
43. **while** (nData < 20)
44. {
45. WaitForSingleObject(g\_hEmpty, INFINITE);
46. g\_arrDataQueue[g\_j] = ++nData;
47. g\_j = (g\_j + 1) % QUEUE\_LEN;
48. EnterCriticalSection(&g\_cs);
49. SetConsoleColor(FOREGROUND\_GREEN);
50. printf("    将数据%d写入队列\n", nData);
51. SetConsoleColor(FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);
52. LeaveCriticalSection(&g\_cs);
53. Sleep(rand() % 300);
54. ReleaseSemaphore(g\_hFull, 1, NULL);
55. }
56. **return** 0;
57. }
58. **int** main()
59. {
60. printf("     秒杀多线程第十六篇 多线程十大经典案例 双线程读写队列数据\n");
61. printf(" - by MoreWindows( http://blog.csdn.net/MoreWindows/article/details/8646902 ) -\n\n");
63. InitializeCriticalSection(&g\_cs);
64. g\_hEmpty = CreateSemaphore(NULL, QUEUE\_LEN, QUEUE\_LEN, NULL);
65. g\_hFull = CreateSemaphore(NULL, 0, QUEUE\_LEN, NULL);
67. srand(time(NULL));
68. g\_i = g\_j = 0;
69. **HANDLE** hThread[2];
70. hThread[0] = (**HANDLE**)\_beginthreadex(NULL, 0, ReaderThreadFun, NULL, 0, NULL);
71. hThread[1] = (**HANDLE**)\_beginthreadex(NULL, 0, WriterThreadFun, NULL, 0, NULL);
73. WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
75. **for** (**int** i = 0; i < 2; i++)
76. CloseHandle(hThread[i]);
77. CloseHandle(g\_hEmpty);
78. CloseHandle(g\_hFull);
79. DeleteCriticalSection(&g\_cs);
80. **return** 0;
81. }

**《多线程十大经典案例之一双线程读写队列数据》运行结果：**

程序运行结果如下：



本文配套程序下载地址为：<http://download.csdn.net/detail/morewindows/5136035>