# 秒杀多线程第五篇 经典线程同步 关键段CS

原创

MoreWindows 于 2012-04-11 09:06:40 发布 87190 收藏 25
分类专栏： Windows多线程 Windows编程 秒杀多线程面试题系列 Windows C/C++/C# 编程 文章标签： 多线程 thread microsoft fun null struct



Windows多线程 同时被 3 个专栏收录
15 篇文章 35 订阅
订阅专栏


Windows编程
87 篇文章 11 订阅
订阅专栏

秒杀多线程面试题系列
15 篇文章 1113 订阅
订阅专栏

上一篇《秒杀多线程第四篇 一个经典的多线程同步问题》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_SECTION lpCriticalSection);
```

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_SECTION lpCriticalSection);
```

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：

```
#include <stdio.h>
#include <process.h>
#include <windows.h>
long g_nNum;
unsigned int __stdcall Fun(void *pPM);
const int THREAD_NUM = 10;
//关键段变量声明
CRITICAL_SECTION  g_csThreadParameter, g_csThreadCode;
int main()
{
 printf("     经典线程同步 关键段\n");
 printf(" -- by MoreWindows( http://blog.csdn.net/MoreWindows ) --\n\n");

 //关键段初始化
 InitializeCriticalSection(&g_csThreadParameter);
 InitializeCriticalSection(&g_csThreadCode);

 HANDLE  handle[THREAD_NUM];
 g_nNum = 0;
 int i = 0;
 while (i < THREAD_NUM)
 {
  EnterCriticalSection(&g_csThreadParameter);//进入子线程序号关键区域
  handle[i] = (HANDLE)_beginthreadex(NULL, 0, Fun, &i, 0, NULL);
  ++i;
 }
 WaitForMultipleObjects(THREAD_NUM, handle, TRUE, INFINITE);

 DeleteCriticalSection(&g_csThreadCode);
 DeleteCriticalSection(&g_csThreadParameter);
 return 0;
}
unsigned int __stdcall Fun(void *pPM)
{
 int nThreadNum = *(int *)pPM;
 LeaveCriticalSection(&g_csThreadParameter);//离开子线程序号关键区域

 Sleep(50);//some work should to do

 EnterCriticalSection(&g_csThreadCode);//进入各子线程互斥区域
 g_nNum++;
 Sleep(0);//some work should to do
 printf("线程编号为%d  全局资源值为%d\n", nThreadNum, g_nNum);
 LeaveCriticalSection(&g_csThreadCode);//离开各子线程互斥区域
 return 0;
}
```

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》将介绍使用事件Event来解决这个经典线程同步问题。