[原]DllMain中不当操作导致死锁问题的分析--线程退出时产生了死锁

2012-11-8 阅读1832 评论0

我们回顾下之前举得例子（转载请指明出于breaksoftware的csdn博客）

```
case DLL_PROCESS_ATTACH: {
          printf("DLL DllWithoutDisableThreadLibraryCalls_A:\tProcess attach (tid = %d)\n", tid);
          HANDLE hThread = CreateThread(NULL, 0, ThreadCreateInDllMain, NULL, 0, NULL);
          WaitForSingleObject(hThread, INFINITE);
          CloseHandle(hThread);
      }break;
```

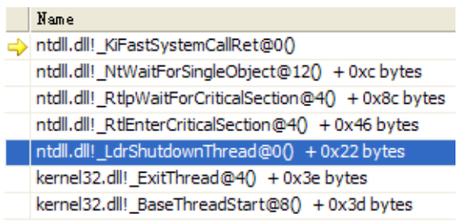
可以想象下这么写代码同学的思路：我要在DLL第一次被映射到进程地址空间时创建一个线程，该线程完成一些可能是初始化的操作后马上结束。然后wait到这个线程结束，我们在DllMain中继续做些操作。

是否想过，如果我们这儿创建一个线程去做事，然后去等待该线程结束。这样就是同步操作了，如此操作不如将线程函数内容放在DllMain中直接执行，何必再去启动一个线程呢？现实中更多的操作可能是：在DLL第一次被映射入进程地址空间时创建一个线程，在卸载出进程空间时将这个线程关闭。

```
HANDLE g_thread_handle = NULL;
HANDLE g_hEvent = NULL;

static DWORD WINAPI ThreadCreateInDllMain( LPVOID p )
{
    WaitForSingleObject( g_hEvent, INFINITE );
    return 0;
}

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                       )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        {
```

```
            g_hEvent = ::CreateEvent( NULL, FALSE, FALSE, NULL );
            g_thread_handle = ::CreateThread( NULL, 0, ThreadCreateInDllMain,NULL, 0, NULL ) ;
        }break;
    case DLL_PROCESS_DETACH:
        {
            ::SetEvent(g_hEvent);
            ::WaitForSingleObject(g_thread_handle, INFINITE );

            ::CloseHandle(g_thread_handle);
            g_thread_handle = NULL ;
            ::CloseHandle(g_hEvent);
            g_hEvent=NULL;
        } break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
        break;
    }
    return TRUE;
}
```

很不幸，这个程序也会死锁。稍微敏感的同学应该可以猜到第25行是死锁的一个因素。是的！那另一个呢？必然是线程了。DllMain中SetEvent之后，工作线程从挂起状态复活，并执行完了return 0。那么另一个死锁因素是出现在线程退出的逻辑中。我们查看堆栈



我们看到是在ExitThread中调用了LdrShutDownThread。我用IDA看了下LdrShutDownThread函数，并和网传的win2K源码做了比较。没发现明显的不一样之处，于是我这儿用更便于阅读的win2K的版本代码

```
VOID
LdrShutdownThread (
    VOID
    )
/*++
Routine Description:
    This function is called by a thread that is terminating cleanly.
    It's purpose is to call all of the processes DLLs to notify them
    that the thread is detaching.
```

```
Arguments:
    None
Return Value:
    None.
--*/
{
    PPEB Peb;
    PLDR_DATA_TABLE_ENTRY LdrDataTableEntry;
    PDLL_INIT_ROUTINE InitRoutine;
    PLIST_ENTRY Next;

    Peb = NtCurrentPeb();

    RtlEnterCriticalSection(&LoaderLock);

    try {
        //
        // Go in reverse order initialization order and build
        // the unload list
        //

Next = Peb->Ldr->InInitializationOrderModuleList.Blink;
while ( Next != &Peb->Ldr->InInitializationOrderModuleList) {
    LdrDataTableEntry
        = (PLDR_DATA_TABLE_ENTRY)
          (CONTAINING_RECORD(Next,LDR_DATA_TABLE_ENTRY,InInitializationOrderLinks));

    Next = Next->Blink;

    //
    // Walk through the entire list looking for
    // entries. For each entry, that has an init
    // routine, call it.
    //

    if (Peb->ImageBaseAddress != LdrDataTableEntry->DllBase) {
        if ( !(LdrDataTableEntry->Flags & LDRP_DONT_CALL_FOR_THREADS)) {
            InitRoutine = (PDLL_INIT_ROUTINE)LdrDataTableEntry->EntryPoint;
            if (InitRoutine && (LdrDataTableEntry->Flags & LDRP_PROCESS_ATTACH_CALLED) ) {
                if (LdrDataTableEntry->Flags & LDRP_IMAGE_DLL) {
```

```
                            if ( LdrDataTableEntry->TlsIndex ) {
                                LdrpCallTlsInitializers(LdrDataTableEntry->DllBase,DLL_THREAD_DETACH);
                                }

#if defined (WX86)
                            if (!Wx86ProcessInit ||
                                LdrpRunWx86DllEntryPoint(InitRoutine,
                                                         NULL,
                                                         LdrDataTableEntry->DllBase,
                                                         DLL_THREAD_DETACH,
                                                         NULL
                                                         ) ==  STATUS_IMAGE_MACHINE_TYPE_MISMATCH)
#endif
                               {
                                LdrpCallInitRoutine(InitRoutine,
                                                    LdrDataTableEntry->DllBase,
                                                    DLL_THREAD_DETACH,
                                                    NULL);
                                }
                            }
                                }
                            }
                        }

        //
        // If the image has tls than call its initializers
        //

        if ( LdrpImageHasTls ) {
            LdrpCallTlsInitializers(NtCurrentPeb()->ImageBaseAddress,DLL_THREAD_DETACH);
            }
        LdrpFreeTls();

    } finally {

        RtlLeaveCriticalSection(&LoaderLock);
    }
}
```

我们看第23行，发现该函数一开始便进入了临界区，也就是说不管该线程是否需要对某DLL调用DllMain都要进入临界区，也就是说DisableThreadLibraryCalls对线程退出时是否进入临界区是没

有影响的。因为主线程正在调用DllMain，所以它先进入了临界区，并一直占用了它。而工作线程退出前也要进入这个临界区做点事，所以它一直进不去，并被系统挂起。而此时占用临界区的主线程要一直等到工作线程退出才肯往下继续执行以退出临界区。这便产生了死锁。

最后附上实验中的例子和《Best Practices for Creating DLLs》

上一篇 下一篇