方亮的专栏

[原]DllMain中不当操作导致死锁问题的分析——DllMain中要谨慎写代码（完结篇）

2012-11-9 阅读2574 评论4

之前几篇文章主要介绍和分析了为什么会在DllMain做出一些不当操作导致死锁的原因。本文将总结以前文章的结论，并介绍些DllMain中还有哪些操作会导致死锁等问题。（转载请指明出于breaksoftware的csdn博客）

DllMain的相关特性

首先列出《DllMain中不当操作导致死锁问题的分析--进程对DllMain函数的调用规律的研究和分析》中论证的11个特性：

Dll的加载不会导致之前创建的线程调用其**DllMain**函数。
线程创建后会调用已经加载了的**DLL**的**DllMain**，且调用原因是**DLL_THREAD_ATTACH**。**(DisableThreadLibraryCalls**会导致该过程不被调用**)**
TerminateThread方式终止线程是不会让该线程去调用该进程中加载的**Dll**的**DllMain**。
线程正常退出时，会调用进程中还没卸载的**DLL**的**DllMain**，且调用原因是**DLL_THREAD_DETACH**。
进程正常退出时，会调用（不一定是主线程）该进程中还没卸载的**DLL**的**DllMain**，且调用原因是**DLL_PROCESS_DETACH**。
加载**DLL**进入进程空间时**(**和哪个线程**LoadLibrary**无关**)**，加载它的线程会调用**DllMain**，且调用原因是**DLL_PROCESS_ATTACH**。
DLL从进程空间中卸载出去前，会被卸载其的线程调用其**DllMain**，且调用原因是**DLL_PROCESS_DETACH**。
TerminateProcess 将导致线程和进程在退出时不对未卸载的**DLL**进行**DllMain**调用。
ExitProcess将导致主线程意外退出，子线程对未卸载的**DLL**进行了**DllMain**调用，且调用原因是**DLL_PROCESS_DETACH**。
ExitThread是最和平的退出方式，它会让线程退出前对未卸载的**DLL**调用**DllMain**。
线程的创建和退出不会对调用了**DisableThreadLibraryCalls**的**DLL**调用**DllMain**。

不要在**DllMain**中做的事情

A 直接或者间接调用**LoadLibrary(Ex)**

假如我们在A.dll中的DllMain收到DLL_PROCESS_ATTACH时，加载了B.dll；而B.dll中的DllMain在收到DLL_PROCESS_ATTACH时又去加载A.dll。则产生了循环依赖。但是注意不要想当然认为这个过程是A.dll的DllMain调用了B.dll的DllMain，B.dll的DllMain再调用了A.dll的DllMain这样的死循环。即使不出现循环依赖，如果出现《DllMain中不当操作导致死锁问题的分析——线程中调用GetModuleFileName、GetModuleHandle等导致死锁》中第三个例子的情况，也会死锁的。

B 使用**CoInitializeEx**

在CoInitializeEx底层会调用LoadLibraryEx，原因同A。

C 使用**CreateProcess**

CreateProcess在底层执行了加载DLL的操作。我用IDA查看Kernel32中的CreateProcess可以发现其底层调用的CreateProcessInternalW中有

```
if ( !v306 )
{
  v24 = LoadLibraryA("advapi32.dll");
  v25 = v24;
  v105 = v24;
  if ( v24 )
  {
    if ( GetProcAddress(v24, "CreateProcessAsUserSecure") )
    {
      v217 = NtQuerySystemInformation(71, &v258, 4, 0);
      if ( !v217 )
        v196 = 1;
    }
    FreeLibrary(v25);
  }
```

D 使用**User32**或**Gdi32**中的函数

User32和Gdi32中部分函数在调用的底层会加载其他DLL。

E 使用托管代码

运行托管代码需要加载其他DLL。

F 与其他线程同步执行

由《DllMain中不当操作导致死锁问题的分析--加载卸载DLL与DllMain死锁的关系》、《DllMain中不当操作导致死锁问题的分析--导致DllMain中死锁的关键隐藏因子》和《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》可知，进程创建和销毁以及DLL的加载都要进入PEB的LoadLock临界区。如果占用了LoaderLock临界区的线程在等待一个需要经过临界区才能结束的线程时，就发生了死锁。以上3篇博文中均有案例。

G 同步对象

如果该同步对象的释放需要获得PEB中的LoaderLock，而占用该临界区的线程又要去等待这个同步对象，则会死锁。其实F中的线程也算是个同步对象。案例详见《DllMain中不当操作导致死锁问题的分析——线程中调用GetModuleFileName、GetModuleHandle等导致死锁》中例子。

H 使用**CreateThread**

理由同F。

I 使用**ExitThread**

理由同F。

J 寄希望于**DisableThreadLibraryCalls**解决死锁问题

由《DllMain中不当操作导致死锁问题的分析--DisableThreadLibraryCalls对DllMain中死锁的影响》可知。DisableThreadLibraryCalls的实现逻辑是：找到PEB结构中用于保存加载器信息的结构体对象Ldr。

```
lkd> dt _PEB
nt!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 SpareBool        : UChar
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS
```

Ldr对象的InMemoryOrderModuleList用户保存已经加载的DLL的链表。

```
lkd> dt _PEB_LDR_DATA
nt!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr32 Void
   +0x00c InLoadOrderModuleList : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY
   +0x024 EntryInProgress  : Ptr32 Void
```

它遍历这个链表，找到调用DisableThreadLibraryCalls的DLL的信息，将该信息中的Flags字段设置或上0x40000。

```
lkd> dt _LDR_DATA_TABLE_ENTRY
nt!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase          : Ptr32 Void
   +0x01c EntryPoint       : Ptr32 Void
   +0x020 SizeOfImage      : Uint4B
   +0x024 FullDllName      : _UNICODE_STRING
   +0x02c BaseDllName      : _UNICODE_STRING
   +0x034 Flags            : Uint4B
   +0x038 LoadCount        : Uint2B
   +0x03a TlsIndex         : Uint2B
   +0x03c HashLinks        : _LIST_ENTRY
   +0x03c SectionPointer   : Ptr32 Void
   +0x040 CheckSum         : Uint4B
   +0x044 TimeDateStamp    : Uint4B
   +0x044 LoadedImports    : Ptr32 Void
   +0x048 EntryPointActivationContext : Ptr32 Void
   +0x04c PatchInformation : Ptr32 Void
```

而创建线程在底层将调用LdrpInitializeThread(详见《DllMain中不当操作导致死锁问题的分析--DisableThreadLibraryCalls对DllMain中死锁的影响》)。该函数一开始便进入了PEB中LoaderLock临界区，在该临界区中根据PEB中LDR的InMemoryOrderModuleList遍历加载的DLL，然后判断该DLL信息的Flags字段是否或上了0x40000。如果或上了，就不调用DllMain。如果没或上，就调用DllMain。这说明DisableThreadLibraryCalls对创建线程时是否进入临界区无关。

在退出线程时底层将调用LdrShutdownThread（详见《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》）。该函数逻辑和LdrpInitializeThread相似，只是在调用DllMain时传的是DLL_THREAD_DETACH。所以DisableThreadLibraryCalls对LdrShutdownThread是否进入临界区也是没有影响的。

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

上一篇 下一篇

发表评论

提 交

查看评论

4楼 Breaksoftware 2014-04-09 11:48
[reply]signed2008[/reply] 以上论点取"非"操作。哈哈。总体来说，就是不会用到锁的操作（直接或者间接的）是可以做的。

3楼 signed2008 2014-04-08 23:52
希望楼主写再写一篇能在dllmain中操作的事情。

2楼 Breaksoftware 2013-03-08 14:22
[reply]HoBoss[/reply] 谢谢，CSDN尽然没有提醒我有留言，我也是刚看到。呵呵。

1楼 HoBoss 2013-02-27 13:21
深入，又易懂。 要是能早三天看到就更好了。 thanks breaksoftware

更多评论（4）

回顶部

移动版 | 桌面版
©1999-2012, CSDN.NET, All Rights Reserved