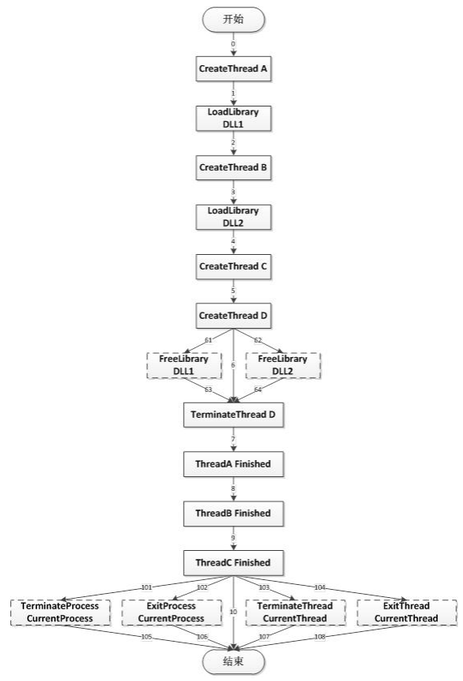
方亮的专栏

[原]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

2012-11-2 阅读2314 评论0

不知道大家是否思考过一个过程：系统试图运行我们写的程序，它是怎么知道程序起始位置的？很多同学想到，我们在编写程序时有个函数，类似Main这样的名字。是的！这就是系统给我们提供的控制程序最开始的地方（注意这儿是提供给我们的，而实际有比这个还要靠前的main）。于是看到DllMain就可以想到它是干嘛的了：Dll的入口点函数。那何时调用这个函数的呢？以及各种调用场景都传给了它什么参数呢？（转载请指明出于breaksoftware的csdn博客）

进程对DLL的载入卸载，以及新线程的创建和退出都会导致对DllMain的调用。于是，我们设计了如下流程



为了尽可能排除一些因素对我们实验的影响，所有线程函数公用一个简单的例程函数

```
static DWORD WINAPI ThreadRoutine(LPVOID lpParam) {
    DWORD dwTID = GetCurrentThreadId();
    PrintLog("Thread%s %u\n", (LPSTR)lpParam, dwTID );
    Sleep(15000);
    PrintLog("\nThread%s Will Exit\n", (LPSTR)lpParam );
    return 0;
}
```

DllMain函数也是非常简单，两个DLL的DllMain函数99.99%是相同的，只是在最后输出所在DLL时列出了各自的DLL名字，以Dll1为例

```
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                         )
{
    string strReason;
    DWORD TID = GetCurrentThreadId();
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:{
            strReason = "DLL_PROCESS_ATTACH";
        }break;
        case DLL_PROCESS_DETACH:{
            strReason = "DLL_PROCESS_DETACH";
        }break;
        case DLL_THREAD_ATTACH:{
            strReason = "DLL_THREAD_ATTACH";
        }break;
        case DLL_THREAD_DETACH:{
            strReason = "DLL_THREAD_DETACH";
        }break;
        default:{
            strReason = "default";
        }break;
    }
    PrintLog("Dll1 TID:%u %s\n", TID, strReason.c_str() );
    return TRUE;
}
```

现在我们说下我设计这个流程的考虑：

0 1 这个过程是为了查看Dll加载后，DllMain被调用是否受之前创建的线程影响。如果受到影响，我们应该能看到Dll1中输出的信息中包含有线程A TID的记录。反之则没有记录。

2 这个过程是为了验证创建新线程，对之前加载的Dll的DllMain调用情况。如果Dll1的DllMain输出了线程B TID记录，那么说明新线程创建会让之前加载Dll的DllMain。反之说明创建新线程不会调用之前加载DLL的DllMain。

3 是为了再次验证0，1这个过程得出的结论。

4 是为了再次验证2这个过程得出的结论。

5 创建的线程是为了之后验证线程正常退出和强制关闭之间的影响。

61，62 是为了验证FreeLibrary是否会对之前对此DLL调用DllMain的线程存在影响。也就是想查看之前在创建线程时对Dll调用DllMain的线程，是否会发现要FreeLibrary了，从而对该Dll再调用DllMain做某些处理(比如清理)。该过程导致DllMain中输出的信息包括那些线程TID的记录，则说明存在影响（其他线程调用DllMain），否则说明不存在影响（其他线程不调用DllMain）。

6 验证通过强制关闭线程对DllMain调用的影响。

7 8 9 验证对不同DLL的DllMain调用情况可能存在不同的线程，在退出时，是否会调用DllMain，以及它们对DllMain的调用规律。

10 101 102 103 104等是通过不同方式验证进程退出对DllMain是否存在调用，以及调用的规律。

我们先在主线程中用 1 2 3 4 5 6 7 8 9 10 这个流程，其结果是

	MainTid:1056	主线程ID是1056
0	CreatThread A ThreadA 3156	A线程ID是3156
1	LoadLibraryA Dll1 Dll1 TID:1056 DLL_PROCESS_ATTACH	Dll1加载了，它是主线程(1056)加载的。调用原因是**DLL_PROCESS_ATTACH**。而它的加载，并不会导致之前创建的**A**线程对其调用**DllMain**。
2	CreatThread B Dll1 TID:4784 DLL_THREAD_ATTACH ThreadB 4784	B线程(4784)在执行到线程函数之前，会去调用之前加载了但还没有卸载的**Dll1**的**DllMain**函数。调用原因是**DLL_THREAD_ATTACH**，而不是之前的DLL_PROCESS_ATTACH。
3	LoadLibraryA Dll2 Dll2 TID:1056 DLL_PROCESS_ATTACH	Dll2加载了，调用其DllMain是主线程。调用原因是**DLL_PROCESS_ATTACH**。加载后，并不会导致线程**A**、**B**去调用其**DllMain**。
4	CreatThread C Dll1 TID:4052 DLL_THREAD_ATTACH Dll2 TID:4052 DLL_THREAD_ATTACH ThreadC 4052	C线程(4052)在执行其线程函数之前，会去调用之前在主线程中加载了但还没有卸载的**DLL**的**DllMain**函数，调用原因是**DLL_THREAD_ATTACH**。
5	CreatThread D Dll1 TID:3440 DLL_THREAD_ATTACH	同上。

	Dll2 TID:3440 DLL_THREAD_ATTACH ThreadD 3440	
6	TerminateThread D	强制关闭线程，不会导致任何**DllMain**的调用。
7	ThreadA Will Exit Dll2 TID:3156 DLL_THREAD_DETACH Dll1 TID:3156 DLL_THREAD_DETACH	线程**A**退出之前，会调用之前加载了但还没有卸载的所有**DLL**的**DllMain**。注意，此处调用是线程A(3156)，而不是主线程(1056)。调用原因是**DLL_THREAD_DETACH**。
8	ThreadB Will Exit Dll2 TID:4784 DLL_THREAD_DETACH Dll1 TID:4784 DLL_THREAD_DETACH	同上。
9	ThreadC Will Exit Dll2 TID:4052 DLL_THREAD_DETACH Dll1 TID:4052 DLL_THREAD_DETACH	同上。
10	Proceess Exit Dll2 TID:1056 DLL_PROCESS_DETACH Dll1 TID:1056 DLL_PROCESS_DETACH	主线程退出前，会调用所有加载了但还没有卸载的**DLL**的**DllMain**。调用原因是**DLL_PROCESS_DETACH**。

为了排除主线程对我们环境的影响我们看下在子线程中执行以上流程的结果（之后我们对流程的修改，都将建立在子线程执行流程的基础之上）

	MainTid:5536	执行的线程ID是5536
0	CreatThread A ThreadA 5684	A线程ID是5684

1	LoadLibraryA Dll1 Dll1 TID:5536 DLL_PROCESS_ATTACH	Dll1加载了，它是执行线程(5536)加载的。调用原因是**DLL_PROCESS_ATTACH**。而它的加载，并不会导致之前创建的**A**线程对其调用**DllMain**。
2	CreatThread B Dll1 TID:4716 DLL_THREAD_ATTACH ThreadB 4716	B线程(4716)在执行到线程函数之前，会去调用之前加载了但还没有卸载的**Dll1**的**DllMain**函数。调用原因是**DLL_THREAD_ATTACH**，而不是之前的DLL_PROCESS_ATTACH。
3	LoadLibraryA Dll2 Dll2 TID:5536 DLL_PROCESS_ATTACH	Dll2加载了，调用其DllMain是执行线程(5536)。调用原因是**DLL_PROCESS_ATTACH**。加载后，并不会导致线程**A**、**B**去调用其**DllMain**。
4	CreatThread C Dll1 TID:2620 DLL_THREAD_ATTACH Dll2 TID:2620 DLL_THREAD_ATTACH ThreadC 2620	C线程(2620)在执行其线程函数之前，会去调用之前在执行线程中加载了但还没有卸载的**DLL**的**DllMain**函数，调用原因是**DLL_THREAD_ATTACH**。
5	CreatThread D Dll1 TID:1016 DLL_THREAD_ATTACH Dll2 TID:1016 DLL_THREAD_ATTACH ThreadD 1016	同上。
6	TerminateThread D	强制关闭线程，不会导致任何DllMain的调用。
7	ThreadA Will Exit Dll2 TID:5684 DLL_THREAD_DETACH Dll1 TID:5684 DLL_THREAD_DETACH	线程**A**退出之前，会调用之前加载了但还没有卸载的所有**DLL**的**DllMain**。注意，此处调用是线程A(5684)，而不是执行线程(5536)。调用原因是**DLL_THREAD_DETACH**。
8	ThreadB Will Exit	同上。

	Dll2 TID:4716 DLL_THREAD_DETACH Dll1 TID:4716 DLL_THREAD_DETACH	
9	ThreadC Will Exit Dll2 TID:2620 DLL_THREAD_DETACH Dll1 TID:2620 DLL_THREAD_DETACH	同上。
10	Dll2 TID:5536 DLL_THREAD_DETACH Dll1 TID:5536 DLL_THREAD_DETACH Proceess Exit Dll2 TID:3904 DLL_PROCESS_DETACH Dll1 TID:3904 DLL_PROCESS_DETACH	执行线程**(5536)**在退出时调用了它加载了但还没有卸载的两个**DLL**的**DllMain**，调用原因是**DLL_THREAD_DETACH**。 主线程退出前，会调用所有之前加载了但还没有卸载的**DLL**的**DllMain**。调用原因是**DLL_PROCESS_DETACH**。

看了如此一串后，我想很多人都会有点晕，现在我总结一下：

一 **Dll**的加载不会导致之前创建的线程调用其**DllMain**函数。

二 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，之后会介绍)

三 **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**。

四 线程正常退出时，会调用进程中已经加载过的的DLL的DllMain，且调用原因是DLL_THREAD_DETACH。(不准确，之后纠正)

五 进程正常退出时，会调用该进程中已经加载过的的DLL的DllMain，且调用原因是DLL_PROCESS_DETACH。(不准确，之后纠正)

六 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**。

我们将过程6替换为过程61，并在子线程中执行，结果大部分相似，我把不一样的地方列出来(执行线程TID是4752)

61	Dll1 TID:4752 DLL_PROCESS_DETACH	执行线程(4752)中卸载了Dll1，则执行线程(4752)调用该DLL的DllMain，且原因是DLL_PROCESS_DETACH。
6	TerminateThread D	
7	ThreadA Will Exit	线程A不会对已经卸载了的Dll1调用其DllMain。

	Dll2 TID:3688 DLL_THREAD_DETACH	
8	ThreadB Will Exit Dll2 TID:1872 DLL_THREAD_DETACH	同上。
9	ThreadC Will Exit Dll2 TID:5600 DLL_THREAD_DETACH	同上。
10	Dll2 TID:4752 DLL_THREAD_DETACH Proceess Exit Dll2 TID:2364 DLL_PROCESS_DETACH	同上。 进程退出时，对尚未卸载的DLL调用其DllMain，且原因是DLL_PROCESS_DETACH。

基于以上结果，我们将以上四五两点结论再严谨点

四 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**。

五 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**。

并得出以下结论

七 **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**。

如果仔细看过我试验结果的同学，应该看到一个现象：线程**A**不会对**Dll1**调用**DllMain**(DLL_THREAD_ATTACH)，而在线程**A**退出时，却会调用**DLL1**的**DllMain**(DLL_THREAD_DETACH)。这种不同步的现象是不是让你内心感觉很疑惑？你说windows为什么要这么设计呢？我不明白。《windows核心编程》也有对该现象的一个描述：虽然当系统将该线程连接到该DLL的时候，不会向该DLL发送DLL_THREAD_ATTACH通知。但是当系统将该线程与DLL解除连接的时候，却会向该DLL发送DLL_THREAD_DETACH通知。由于这个原因，我们在进行与线程相关的清理时必须极其小心。幸运的是，在大多数程序中，调用**Loadlibrary**的线程与调用**Freelibrary**的线程是同一个线程。

现在我们再将过程61换成6，并依次用101(TerminateProcess)、102(ExitProcess)、103(TerminateThread)、104(ExitThread)替换10。我列一下不同点

101	The thread 'Win32 Thread' (0x142c) has exited with code -1 (0xffffffff). The program '[6128] CallDllMain.exe: Native' has exited with code -1 (0xffffffff).	执行线程(0x142c)和进程退出时未对任何加载的DLL调用DllMain。 没有对主线程退出的捕获。
102	The thread 'Win32 Thread' (0x1214) has exited with code -1 (0xffffffff). Dll2 TID:4660 DLL_PROCESS_DETACH Dll1 TID:4660 DLL_PROCESS_DETACH The program '[2576] CallDllMain.exe: Native' has exited with code -1 (0xffffffff).	主进程(0x1214) 提前意外关闭，未对任何加载的DLL调用DllMain。 执行线程(4660)退出时对加载了的DLL调用了其DllMain的DLL_PROCESS_DETACH。
103	The thread 'Win32 Thread' (0x81c) has exited with code -1 (0xffffffff). Proceess Exit Dll2 TID:2356 DLL_PROCESS_DETACH	执行线程(0x81c)退出时未对任何加载的DLL调用DllMain。 主进程(2356)退出时对加载了的DLL调用了其DllMain的DLL_PROCESS_DETACH。

	Dll1 TID:2356 DLL_PROCESS_DETACH The program '[5860] CallDllMain.exe: Native' has exited with code 0 (0x0).	
104	Dll2 TID:5600 DLL_THREAD_DETACH Dll1 TID:5600 DLL_THREAD_DETACH The thread 'Win32 Thread' (0x15e0) has exited with code -1 (0xffffffff). Proceess Exit Dll2 TID:632 DLL_PROCESS_DETACH Dll1 TID:632 DLL_PROCESS_DETACH The program '[284] CallDllMain.exe: Native' has exited with code 0 (0x0).	执行线程(5600)退出时对加载的DLL调用了DllMain，且原因是DLL_THREAD_DETACH。 主进程(632)退出时对加载了的DLL调用了其DllMain的DLL_PROCESS_DETACH。

从以上我们可以看出Terminate(101、103)类型函数比Exit(102、104)类型函数暴力。

102例子中我们看到主线程退出后，子线程还在正常工作的场景，可以想象，可能是ExitProcess是直接TerminateThread主线程了。总结如下：

八 **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。

九 **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**。(《windows核心编程》上是说，调用ExitProcess函数的线程将负责执行DllMain函数的代码。(DLL_PROCESS_DETACH))

十 **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**。

我们再考虑下DisableThreadLibraryCalls函数对DllMain函数的调用的影响。我们在Dll1的DllMain中加入DisableThreadLibraryCalls(hModule);我们观察下结果

	MainTid:7760	
0	CreatThread A ThreadA 7992	
1	LoadLibraryA Dll1 TID:7760 DLL_PROCESS_ATTACH	加载DLL1，执行线程调用其DllMain，原因是DLL_PROCESS_ATTACH。
2	CreatThread B ThreadB 6684	线程B创建不会对DLL1调用DllMain了。因为DLL1中调用了DisableThreadLibraryCalls。
3	LoadLibraryA Dll2 Dll2 TID:7760 DLL_PROCESS_ATTACH	加载DLL2。执行线程调用其DllMain，原因是DLL_PROCESS_ATTACH。
4	CreatThread C	线程C创建不会对DLL1调用DllMain了。但是会对没有调用过DisableThreadLibraryCalls的DLL2调用DllMain。

	Dll2 TID:8168 DLL_THREAD_ATTACH ThreadC 8168	
5	CreatThread D Dll2 TID:1848 DLL_THREAD_ATTACH ThreadD 1848	同上
6	TerminateThread D	
7	ThreadA Will Exit Dll2 TID:7992 DLL_THREAD_DETACH	线程A退出，不会对DLL1调用DllMain了。但是会对没有调用过DisableThreadLibraryCalls的DLL2调用DllMain。
8	ThreadB Will Exit Dll2 TID:6684 DLL_THREAD_DETACH	同上
9	ThreadC Will Exit Dll2 TID:8168 DLL_THREAD_DETACH	同上
10	Dll2 TID:7760 DLL_THREAD_DETACH Proceess Exit Dll2 TID:8096 DLL_PROCESS_DETACH Dll1 TID:8096 DLL_PROCESS_DETACH	执行线程（7760）出前不会对DLL1调用DllMain了。 进程退出前，主线程会对DLL1和DLL2调用DllMain。

通过以上我们可以再得出一个结论

十一 线程的创建和退出不会对调用了**DisableThreadLibraryCalls**的**DLL**调用**DllMain**。

最后，我们考虑下LoadLibrary和Freelibrary对DllMain的影响。我将在两个线程中尝试多次LoadLibrary同一个Dll，多次Freelibrary同一个Dll。

```
PrintLog("LoadLibraryA1\n");
HMODULE hDll1 = LoadLibraryA("DLL1");
WAIT();
PrintLog("LoadLibraryA2\n");
hDll1 = LoadLibraryA("DLL1");
WAIT();
PrintLog("LoadLibraryA3\n");
hDll1 = LoadLibraryA("DLL1");
WAIT();
CreateThread( NULL, NULL, ThreadFun, NULL, 0, NULL );
```

```
Sleep(35000);
PrintLog("FreeLibrary1\n");
FreeLibrary(hDll1);
WAIT();
PrintLog("FreeLibrary2\n");
FreeLibrary(hDll1);
WAIT();
PrintLog("FreeLibrary3\n");
FreeLibrary(hDll1);
WAIT();
```

其结果是

```
LoadLibraryA1
Dll1 TID:4620 DLL_PROCESS_ATTACH
LoadLibraryA2
LoadLibraryA3
Dll1 TID:5560 DLL_THREAD_ATTACH 子线程创建时调用的
MainTid:5560
……
FreeLibrary1
FreeLibrary2
FreeLibrary3
Dll1 TID:4620 DLL_PROCESS_DETACH
```

我们发现第一句LoadLibrary对DllMain产生了调用DLL_PROCESS_ATTACH，而第二三句LoadLibrary不会对DllMain产生任何调用(《windows核心编程》系统不会让进程的主线程用DLL_THREAD_ATTACH值调用DLLMain函数。系统不会让用DLL_PROCESS_ATTACH来调用该DLL的DllMain函数的线程不会得到DLL_THREAD_ATTACH通知)；第一二次FreeLibrary对DllMain没有产生调用，而第三次FreeLibrary对DllMain产生了DLL_PROCESS_DETACH调用。

可以见得，在一个线程中对DLL产生了DllMain调用后，就不会因为Loadlibrary再发生DllMain的调用。

我们前两次FreeLibrary不会对DllMain进行调用，而第三次就是DLL_PROCESS_DETACH。同样这个线程中LoadLibraryA也被调用了三次。可以想象LoadLibraryA和FreeLibrary之间存在一个计数器的关系(LoadLibraryA加计数器，FreeLibrary减计数器)。正如《windows核心编程》上所说：当系统第一次将一个DLL映射到进程的地址空间中时……如果之后一个线程在调用Loadlibrary(Ex)来载入一个已经被映射到进程的地址空间的DLL，那么操作系统只不过是递增该DLL的使用计数，而不会再次用DLL_PROCESS_ATTACH来调用DllMain函数。

本文中介绍了经过几轮实验，得出了11条规律。我们之后研究DllMain导致的死锁，将用到这些规律。

上一篇 下一篇

发表评论

提 交

查看评论

更多评论（0）

回顶部