调试之剑专栏第 8 期

拯救挂死的 PowerPoint

这一期给大家讲一个真实的故事，起源是笔者亲身经历的一次应用程序挂死，主要内容是探寻挂死原因的求索过程，末尾是关于“谁之错”的思考。因为很想把这个故事早点与大家分享，所以我们把上个月开始的托管系列中断一期（抱歉无法一一征求读者诸君同意）。

戛然而止

故事发生在一个星期天的下午，虽然是星期天，但笔者仍伏在电脑面前“做功课”。这天要完成的任务是使用 PowerPoint 准备一份讲义，内容是关于调试的——我最喜欢的话题。启动 PowerPoint，打开惯用的模板，想出一个喜欢的题目——《用户态崩溃和转储分析》，列出简单的提纲，然后选取几年来积累的资料，驾轻就熟，很快就做好了二十几个页面；当然还应该取个中意的文件名，保存一下。如此这般，很快几个小时便过去了，我仍然不觉得疲倦，正如古人所说的：“我自乐此，不为疲也”，眼看着再有几页就做完了。这时需要把一个脚本文件插入到 PPT 中，为的是在演讲时可以打开真实的脚本来讲解。于是，我便把 PowerPoint 程序的窗口调小一些，用鼠标抓起脚本文件，移动到 PowerPoint 窗口的投影页面，鼠标箭头右下方出现预期的加号，松开鼠标释放……但是释放后，我很快感觉到了异常，“货物”没有落在页面中，鼠标箭头右下方的加号还在，这意味着拖动的“货物”没有放下去。点击 PowerPoint 窗口的图标，就像点到木头上，过了几秒钟再点时，窗口标题中出现了熟悉的“（Not Responding）”（图 1），PowerPoint 挂死了，刚才还灵活自如的窗口元素现在都僵硬不动了，喧嚣归于宁静，一切戛然而止……

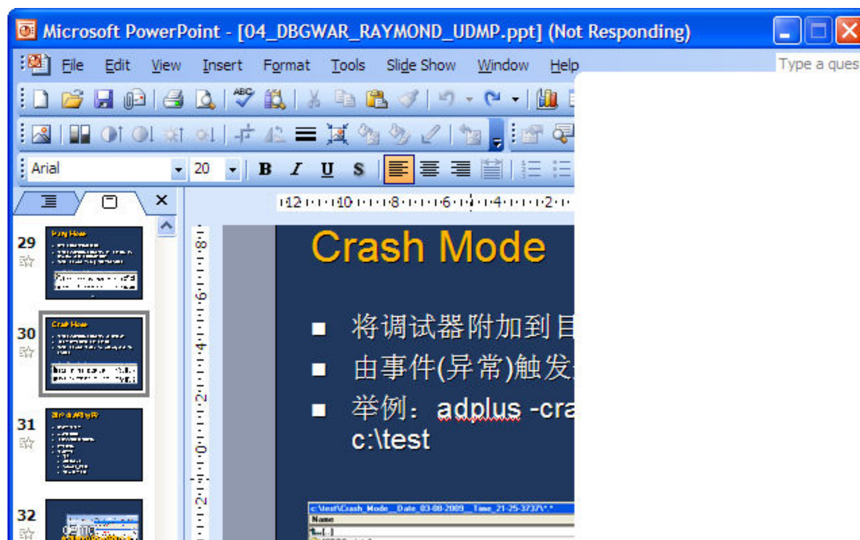

图 1 PowerPoint 戛然而止

捶胸顿足

宁静了几秒钟，我狠狠的一跺脚，“诶呀，还有好几页没有保存呢！”。怎么办？所有按钮和菜单无法操作，想通过简易的方法把文件保存起来几乎不可能了。把进程杀掉？杀掉简单，但是没有保存的东西很可能就永远不见了。虽然，前面曾经保存过，但是最近十几分钟所做

的内容还没有保存；虽然十几分钟的劳动不值几个钱，但是也舍不得白白放弃呀；虽然 Office 程序有文件自动保存和恢复机制，但是没人确保它一定工作呀；虽然……更何况，自己是开发软件的，又是热衷调试的，并且说过不畏软件难题，因此对于这样的问题怎么能轻易放过呢，是可忍孰不可忍？

拍照存档

首先使用 ADPlus 给 PowerPoint 程序和从中拖出文件的 Windows Commander 程序“拍照”，也就是将这两个进程的当前状态转储到文件中。使用的命令如下：

Adplus -pn powerpnt.exe -pn wincmd32.exe -hang -o c:\test

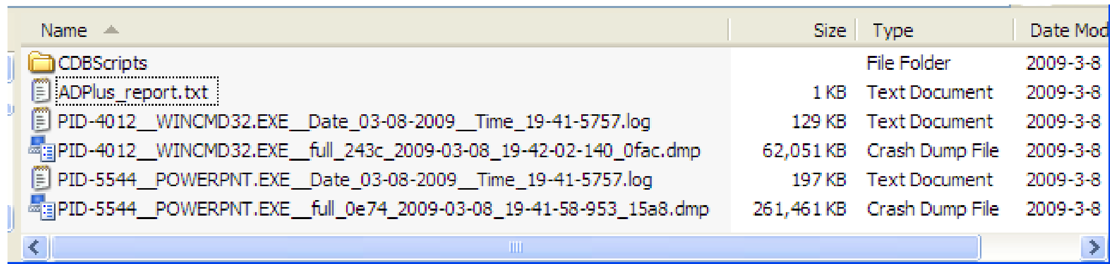

图 2 使用 ADPlus 产生的转储文件和日志报告

命令执行后，便会产生一系列转储文件、日志文件和报告文件（图 2），这样便保存下了永久的现场资料，即使短时间无法找到问题，那么还可以等以后有时间或者时机成熟时继续分析，让问题“躲过初一，躲不了十五”。

上调试器

调试器是解决复杂软件问题的最重要工具（《软件调试》第 6 篇序），检验这句话的时候又到了。启动 WinDBG，附加（File > Attach to a Process…）到执行映像为 POWERPNT.exe 的进程，稍稍停顿后，WinDBG 成功附加到目标，列出进程内的所有模块后，中断下来，等待命令。

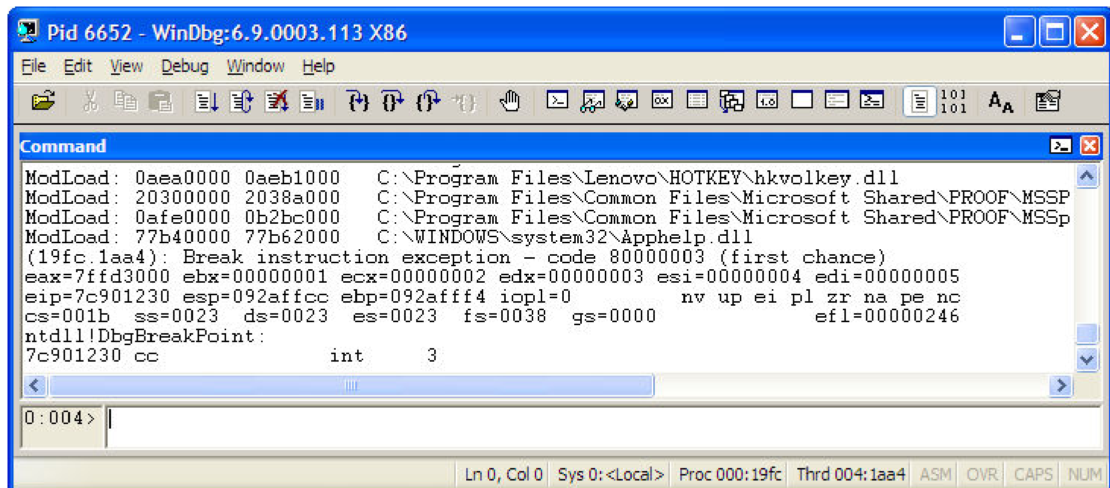

图 3 将 WinDBG 附加到挂死的 PowerPoint 进程

先浏览一下进程内各个模块，看是否有可能是病毒的异常模块侵入。没有发现可疑对象，接下来该如何分析呢？

时光倒流

软件调试中的栈回溯（Stack Backtrace）技术可以根据保存在栈上的信息生成一个线程的函数调用过程，因为是从当前的执行点反向追溯父函数，所以叫栈回溯。对于眼下的问题，界面完全失去响应，这说明负责消息处理的界面更新的 UI 线程阻塞了。对于几乎所有 Windows GUI 程序，编号为 0 的初始线程就是 UI 线程。因此，使用~0s 命令切换到 0 号线程：

```
0:004> ~0s
eax=0b490000 ebx=774e1a3c ecx=00000000 edx=7c90eb94 esi=000003e0 edi=000003e0
eip=7c90eb94 esp=00139d54 ebp=00139d7c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!KiFastSystemCallRet:
7c90eb94 c3              ret
```

在上面显示的寄存器信息中，eip 代表程序指针寄存器，它的值代表着 CPU 将执行的指令，更确切的说，是 CPU 下次执行这个线程时要执行的指令。它目前值所对应的符号是：ntdll!KiFastSystemCallRet，也就是一条 ret 指令（函数返回）。这意味着，当前线程已经进入到了内核态执行，用户态的上下文中保存的是从内核态返回后将要执行的指令和状态。

执行 kn 100 命令显示栈回溯，100 表示要显示的深度：

```
0:000> kn 100
 # ChildEBP RetAddr
00 00139d50 7e4194ae ntdll!KiFastSystemCallRet
01 00139d7c 775c00be USER32!NtUserMessageCall+0xc
02 00139d9c 775c24ce ole32!wInitiate+0x3e
03 00139db8 775bda92 ole32!CDdeObject::CProxyManagerImpl::Connect+0x5f
04 0013a1f4 775bdd00 ole32!CDdeObject::DocumentLevelConnect+0x3f
05 0013a638 775f9904 ole32!DdeBindToObject+0xde
06 0013a8c0 7757c79c ole32!CPackagerMoniker::BindToObject+0xc7
07 0013a8ec 775c9ad3 ole32!BindMoniker+0x60
08 0013a908 775c9edb ole32!wCreateFromFileEx+0x1f
09 0013a988 775ca0d0 ole32!OleCreateFromFileEx+0xfb
0a 0013ac24 7757becb ole32!wCreatePackageEx+0x198
0b 0013ac68 7757bf1f ole32!OleCreateFromDataEx+0xea
0c 0013aca4 3028ad21 ole32!OleCreateFromData+0x42
WARNING: Stack unwind information not available. Following frames may be wrong.
0d 0013acec 302c419a POWERPNT+0x28ad21
0e 0013ad28 303ddf42 POWERPNT+0x2c419a
[为节约篇幅略去多行]
24 0013dcec 7e418806 USER32!InternalCallWinProc+0x28
25 0013dd54 7e4189bd USER32!UserCallWinProcCheckWow+0x150
26 0013ddb4 7e418a00 USER32!DispatchMessageWorker+0x306
27 0013ddc4 30034f03 USER32!DispatchMessageW+0xf
28 0013dde8 30034eca POWERPNT+0x34f03
29 0013ddf8 30034cf5 POWERPNT+0x34eca
2a 0013de40 30004a7b POWERPNT+0x34cf5
2b 0013ff18 30004a2c POWERPNT+0x4a7b
```

```
2c 0013ffc0 7c816ff7 POWERPNT+0x4a2c
2d 0013fff0 00000000 kernel32!BaseProcessStart+0x23
```

扫描一眼这个栈回溯结果，最下方是 kernel32!BaseProcessStart 函数，即初始线程正式开始执行时的起点，其上的几行（#28~#2c）都显示为 POWERPNT 加一个很大的偏移值，这意味着没有找到 POWERPNT 模块的符号，所以只好以模块的起始地址为参照物，微软的符号服务器包含了大多数 Windows 系统模块的公开符号文件，但是没有包含 Office 程序的符号文件，因此这样显示也属正常。栈帧#0c~#09 中的函数与 OLE（对象链接与嵌入）有关，栈帧#05~#03 中都包含 DDE（动态数据交换）字样，这些信息与挂死前的文件拖动操作很吻合。栈帧#01 表示发起一个系统服务调用 NtUserMessageCall，栈帧#0 表示以快速系统调用方式进入到内核态。

从栈回溯可以推测出初始线程因为 NtUserMessageCall 这个系统调用而进入到内核态执行，“至今”尚未返回。事实上，大多数应用程序挂死也都是“挂”在内核态。

顺藤摸瓜

要想了解挂死的更多原因，就要进一步分析挂死前的执行过程，也就是继续从上面的栈回溯中挖掘线索。很多函数都使用栈来传递参数，因此可以尝试通过观察参数取值来搜集更多资料。要观察参数，最好知道参数类型，这又需要知道函数原型。因为我们没有系统模块的私有符号文件，所以没有办法让 WinDBG 直接显示函数原型和参数（kPL 命令）。怎么办呢？因为 SDK 中公开的函数在 MSDN 中有原型公布，所以可以从公开的 API 下手，纵观栈帧中的各个函数，OleCreateFromData、OleCreateFromDataEx 和 OleCreateFromFileEx 都是公开的 API，查看 MSDN，OleCreateFromFileEx 的第二个参数是字符串类型：

```
HRESULT OleCreateFromFileEx(
  REFCLSID rclsid,          //Reserved; must be CLSID_NULL
  LPCOLESTR lpszFileName,   //Pointer to name of file to initialize
```

于是可以根据对应栈帧的基地址（0013a988）得到第二参数的地址（EBP+C），然后用 dU 命令来观察这个参数的值：

```
0:000> dU poi(0013a988+c)
001cc458  "c:\dumps\dde\Hang_Mode__Date_03-"
001cc498  "08-2009__Time_16-22-1818\CDBScri"
001cc4d8  "pts\PID-1076__INETINFO.EXE_IIS_I"
001cc518  "n-Process_Applications.cfg"
```

这恰恰是当时拖动的那个文件的完整路径，看来 PowerPoint 正在接收这个文件，但意外挂死了。

Google 一下或者看一下 SendMessageW 的反汇编，就可以知道 SendMessage API 内部会调用 NtUserMessageCall 这个内核服务，而且二者的函数原型也相似的。显示有关栈帧的参数：

```
0:000> kb 2
ChildEBP RetAddr  Args to Child
00139d50 7e4194ae 7e441396 00010014 000003e0 ntdll!KiFastSystemCallRet
00139d7c 775c00be ffffffff 000003e0 001d06dc USER32!NtUserMessageCall+0xc
```

NtUserMessageCall 的第一个参数是要发送消息的目标窗口句柄，而 0xffffffff 具有特殊含义，是用来广播消息的（HWND_BROADCAST）。第二个参数是要发送的消息，因为 SDK 中并没有把所有 Windows 消息常量定义在一起，所以寻找 0x3e0 所代表的消息名称不是特别简单，我们不妨先放下这条线索。

分析到这里，可以推测出，PowerPoint 的初始线程正在做接收拖动过来的.cfg 文件的工作，在完成这个任务的过程中，它调用了 OleCreateFromFileEx API，后者内部调用了 DDE 有关的内部类和方法，而 DDE 的方法出于某种目的而调用内核服务 NtUserMessageCall 开始广播 0x3e0 消息。而这个消息广播调用一去不回。

上下求索

既然 UI 线程进入内核态执行迟迟不归而导致用户界面失去响应，那么很自然地想到去看看这个线程在内核态干什么？

于是再运行一个 WinDBG 实例，开始一个本地内核调试会话（File > Kernel Debug > Local）。然后执行下面的命令找到 PowerPoint 进程：

lkd> !process 0 0 powerpnt.exe

PROCESS 88bfbb80 SessionId: 0 Cid: 19fc Peb: 7ffd3000 ParentCid: 03c4

DirBase: 18900e00 ObjectTable: e44e0090 HandleCount: 268.

Image: POWERPNT.EXE

然后列出这个进程的各个线程结构（以下输出信息的格式做过调整）：

lkd> !PROCESS 88bfbb80 2

PROCESS 88bfbb80 SessionId: 0 Cid: 19fc Peb: 7ffd3000 ParentCid: 03c4

DirBase: 18900e00 ObjectTable: e44e0090 HandleCount: 268.

Image: POWERPNT.EXE

THREAD 891c4020 Cid 19fc.1dbc Teb: 7ffdf000 Win32Thread: e2456b00 WAIT: (Suspended) KernelMode Non-Alertable SuspendCount 1 FreezeCount 1

然后执行!THREAD 891c4020 命令显示第一个线程（UI 线程）的详细信息（图 4）。

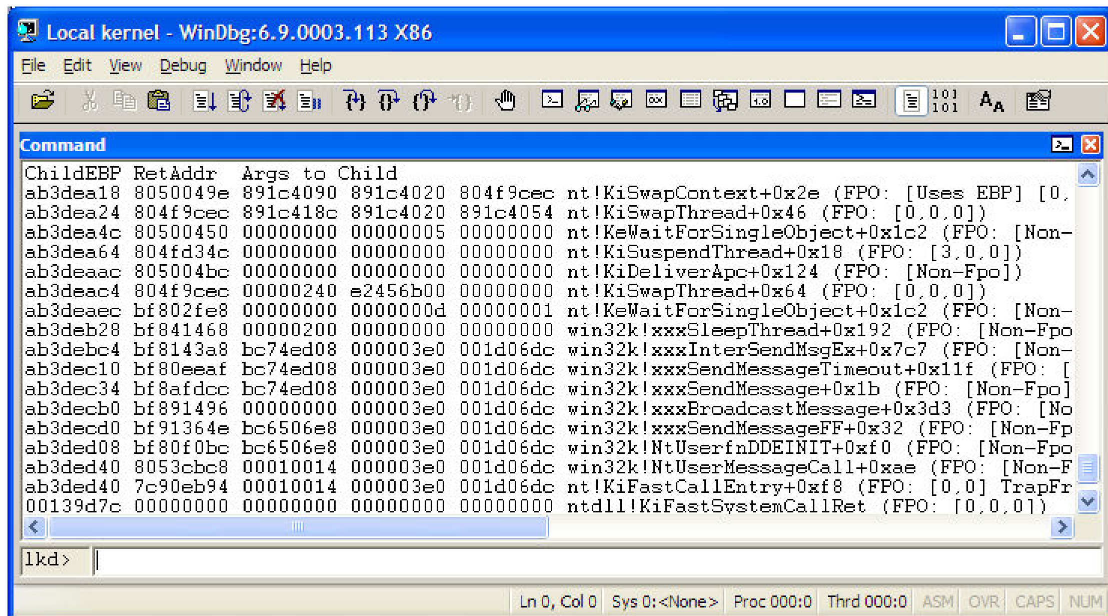

图 4 UI 线程在内核态执行的过程

观察图 4 所示的内核态栈回溯，从很多函数名中包含的 SendMessage 字样就可以可以知道这个线程正在内核态执行发送消息的任务，这与前面的分析是一致的。

现在的关键是找到向哪个窗口发送消息时停滞不前了。这还是需要分析参数，可以看到第二参数那一列中有多行都是 3e0，说明是在发送 3e0 这个消息。再看 xxxSendMessage、xxxSendMessageTimeout 和 xxxInterSendMsgEx 这个几个函数的第一个参数，它们的第一个

参数都是 bc74ed08，这个值很可能与发送消息的目标窗口有关。或许是包含窗口句柄的指针，使用 DD 命令观察它：

```
lkd> dd bc74ed08 l1
bc74ed08  001506c4
```

如何验证 001506c4 是不是一个窗口句柄呢？有多种方法，可以使用 Spy++工具来查找，也可以使用 Skywing 编写的 sdbgext 扩展模块中的 hwnd 扩展命令来观察：

```
0:000> !sdbgext.hwnd 001506c4
Window    001506c4
Name
Class     WindowsForms10.Window.0.app.0.3ce0bb8
WndProc   00000000
Style     WS_OVERLAPPED
ExStyle   WS_EX_WINDOWEDGE WS_EX_LEFT WS_EX_LTRREADING WS_EX_RIGHTSCROLLBAR
HInstance 00400000
ParentWnd 00000000
Id        00000000
UserData  00000000
Unicode   TRUE
ThreadId  000016d8
ProcessId 00001c68
```

果真是窗口句柄，该窗口所属的进程 ID 和线程 ID 分别是 00001c68 和 000016d8，转化成十进制（使用.formats 命令），然后到任务管理器中查找，就可以看到进程的名称了。

幕后黑手

分析到这里，我们定位到了阻碍 PowerPoint 的 UI 线程的窗口句柄和所属进程，很可能是这个进程不回复 PowerPoint 发过来的消息而导致 PowerPoint 阻塞在那里。那么这个进程为什么不回复消息呢？运行第三个 WinDBG 实例，附加到这个进程，使用~*命令列出所有线程，然后切换到线程号为 000016d8 的线程（~0s），观察栈回溯：

```
0:000> kn 100
 # ChildEBP RetAddr
00 0012f0a0 7c90e288 ntdll!KiFastSystemCallRet
01 0012f0a4 7c801875 ntdll!NtReadFile+0xc
02 0012f10c 77deb3cb KERNEL32!ReadFile+0x16c
03 0012f138 77deb25f ADVAPI32!ScGetPipeInput+0x2a
04 0012f1ac 77deb568 ADVAPI32!ScDispatcherLoop+0x3f
05 0012f40c 00a6a762 ADVAPI32!StartServiceCtrlDispatcherW+0xe3
WARNING: Frame IP not in any known module. Following frames may be wrong.
06 0012f428 67a241d2 0xa6a762
07 00000000 00000000 System_ServiceProcess_ni+0x41d2
```

从上面的栈回溯可以看出，这是一个使用.Net 语言编写的系统服务，0xa6a762 是即时编译后产生的代码所在的位置。从栈帧#05~#00 可以看出这个线程进入了等待系统服务控制命令（暂停、停止服务等）的循环，在等待服务管理器发送给它的命令。也就是说它根本没有等待和处理窗口消息。概括一下，这个进程创建了一个顶层的窗口，但是没有合适的窗口循环，

因为没有对系统发送给它的消息作出合适的分发和处理。

起死回生

找到了导致问题的进程和原因后，在系统服务管理器中停止这个服务，然后将附加到PowerPoint进程的WinDBG分离开（Debug > Detach Debuggee），这时会发现，PowerPoint起死回生了，又焕发了活力，所有功能也都是正常的。

环境问题？

回过头来看这个问题，到底是谁的错呢？如果说是PowerPoint，似乎它的实现也没有明显的错误，它按部就班地调用API，也不知道发送消息会得不到回复呀，事实上，其它在UI线程这样做的程序也有这个问题。如果说是那个系统服务程序的问题，它也可以争辩“我为什么一定要处理你的消息？”“不处理，你就应该死在那么？”，听着也有道理，事实上，如果用调试器将某个窗口程序中断下来，也会造成类似的影响。推而广之，很多挂死的问题都是因为双方或者多方协作时出了问题。前面提到过的那个3e0消息是WM_DDE_INITIATE消息，这是16位的Windows所定义的消息，当时整个系统的任务调度机制都是协作方式的，要求系统中的每个任务谦恭礼让，有绅士风度，但是…。扯远了，就此打住，下一期再见！

本期问题：
除了本期讨论的情况，你知道还有哪些原因导致应用程序挂死么？