【转帖】Windows异常处理流程 - 看雪软件安全论坛

原创

导读：z

作者：SoBelt
出处：http://www.xfocus.net/articles/200412/761.html
日期：2005-01-06

先来说说异常和中断的区别。中断可在任何时候发生，与CPU正在执行什么指令无关，中断主要由I/O设备、处理器时钟或定时器等硬件引发，可以被允许或取消。而异常是由于CPU执行了某些指令引起的，可以包括存储器存取违规、除0或者特定调试指令等，内核也将系统服务视为异常。中断和异常更底层的区别是当广义上的中断(包括异常和硬件中断)发生时如果没有设置在服务寄存器(用命令号0xb向8259-1中断控制器0x20端口读出在服务寄存器1，用0xb向8259-2中断控制器的0xa0端口读出在服务寄存器2)相关的在服务位(每个在服务寄存器有8位，共对应IRQ 0-15)则为CPU的异常，否则为硬件中断。

下面是WINDOWS2000根据INTEL x86处理器的定义，将IDT中的前几项注册为对应的异常处理程序(不同的操作系统对此的实现标准是不一样的，这里给出的和其它一些资料不一样是因为这是windows的具体实现)：

中断号　名字　　原因
0x0　除法错误　　1、DIV和IDIV指令除0
　　2、除法结果溢出
0x1　调试陷阱　　1、EFLAG的TF位置位
　　2、执行到调试寄存器(DR0-DR4)设置的断点
　　3、执行INT 1指令
0x2　NMI中断　　将CPU的NMI输入引脚置位(该异常为硬件发生非屏蔽中断而保留)
0x3　断点　　执行INT 3指令
0x4　整数溢出　　执行INTO指令且OF位置位
0x5　BOUND边界检查错误　BOUND指令比较的值在给定范围外
0x6　无效操作码　指令无法识别
0x7　协处理器不可用　1、CR0的EM位置位时执行任何协处理器指令
　　2、协处理器工作时执行了环境切换
0x8　双重异常　　处理异常时发生另一个异常
0x9　协处理器段超限　浮点指令引用内存超过段尾
0xA　无效任务段　任务段包含的描述符无效(windows不使用TSS进行环境切换，所以发生该异常说明有其它问题)
0xB　段不存在　　被引用的段被换出内存
0xC　堆栈错误　　1、被引用内存超出堆栈段限制
　　2、加载入SS寄存器的描述符的present位置0
0xD　一般保护性错误　所有其它异常处理例程无法处理的异常
0xE　页面错误　　1、访问的地址未被换入内存
　　2、访问操作违反页保护规则
0x10　协处理器出错　CR0的EM位置位时执行WAIT或ESCape指令
0x11　对齐检查错误　对齐检查开启时(EFLAG对齐位置位)访问未对齐数据

其它异常还包括获取系统启动时间服务int 0x2a、用户回调int 0x2b、系统服务int 0x2e、调试服务int 0x2d等系统用来实现自己功能的部分，都是通过异常的机制，触发方式就是执行相应的int指令。

这里给出几个异常处理中重要的结构：

陷阱帧TrapFrame结构(后面提到的异常帧ExceptionFrame结构其实也是一个KTRAP_FRAME结构)：

```
typedef struct _KTRAP_FRAME {
    ULONG   DbgEbp;
```

```
    ULONG   DbgEip;
    ULONG   DbgArgMark;
    ULONG   DbgArgPointer;
    ULONG   TempSegCs;
    ULONG   TempEsp;
    ULONG   Dr0;
    ULONG   Dr1;
    ULONG   Dr2;
    ULONG   Dr3;
    ULONG   Dr6;
    ULONG   Dr7;
    ULONG   SegGs;
    ULONG   SegEs;
    ULONG   SegDs;
    ULONG   Edx;
    ULONG   Ecx;
    ULONG   Eax;
    ULONG   PreviousPreviousMode;
    PEXCEPTION_REGISTRATION_RECORD ExceptionList;
    ULONG   SegFs;
    ULONG   Edi;
    ULONG   Esi;
    ULONG   Ebx;
    ULONG   Ebp;
    ULONG   ErrCode;
    ULONG   Eip;
    ULONG   SegCs;
    ULONG   EFlags;
    ULONG   HardwareEsp;
    ULONG   HardwareSegSs;
    ULONG   V86Es;
    ULONG   V86Ds;
    ULONG   V86Fs;
    ULONG   V86Gs;
} KTRAP_FRAME;
```

环境Context结构：

```
typedef struct _CONTEXT {
    ULONG ContextFlags;
    ULONG   Dr0;
    ULONG   Dr1;
    ULONG   Dr2;
    ULONG   Dr3;
    ULONG   Dr6;
    ULONG   Dr7;
    FLOATING_SAVE_AREA FloatSave;
    ULONG   SegGs;
    ULONG   SegFs;
    ULONG   SegEs;
    ULONG   SegDs;
    ULONG   Edi;
    ULONG   Esi;
    ULONG   Ebx;
    ULONG   Edx;
    ULONG   Ecx;
    ULONG   Eax;
    ULONG   Ebp;
    ULONG   Eip;
    ULONG   SegCs;
    ULONG   EFlags;
    ULONG   Esp;
    ULONG   SegSs;
    UCHAR   ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
} CONTEXT;
```

异常记录ExceptionRecord结构：

```
typedef struct _EXCEPTION_RECORD {
    NTSTATUS ExceptionCode;
    ULONG ExceptionFlags;
    struct _EXCEPTION_RECORD *ExceptionRecord;
    PVOID ExceptionAddress;
    ULONG NumberParameters;
    ULONG_PTR ExceptionInformatio[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

当发生异常后，CPU记录当前各寄存器状态并在内核堆栈中建立陷阱帧TrapFrame，然后将控制交给对应异常的陷阱处理程序。当陷阱处理程序能处理异常时，比如缺页时通过调页程序MmAccessFault将页换入物理内存后通过iret返回发生异常的地方。但大多数无法处理异常，这时先是调用CommonDispatchException在内核堆栈中建立异常记录ExceptionRecord和异常帧ExceptionFrame。ExceptionRecord很重要，它记录了异常代码、异常地址以及一些其它附加的参数。然后调用KiDispatchException进行异常的分派。这个函数是WINDOWS下异常处理的核心函数，负责异常的分派处理。

KiDispatchException的处理流程(每当异常被某个例程处理时处理的例程将返回TRUE到上一个例程，未处理则返回FALSE。当任何一个例程处理了异常返回TRUE时，则KiDispatchException正常返回)：

在进行用户态内核态的异常的分派前，先判断异常是否来自用户模式，是的话将Context.ContextFlags(这时候Context结构还刚初始化完，还未赋初值) or上CONEXT_FLOATING_POINT，意味着对来自用户模式的异常总是尝试分派浮点状态，这样可以允许异常处理程序或调试器检查和修改协处理器的状态。然后从陷阱帧中取出寄存器值填入Context结构，并判断是否是断点异常(int 0x3和int 0x2d)，如果是的话先将Context.Eip减一使它指向int 0x3指令(无论是由int 0x3还是由int 0x2d引起的异常，因为前面的陷阱处理程序里已经改变过TrapFrame里面的Eip了)。然后判断异常是发生于内核模式还是用户模式，根据不同模式而采取不同处理过程。

如果异常发生于内核模式，会给予内核调试器第一次机会和第二次机会处理异常。当异常被处理后就将设置好陷阱帧并返回到陷阱处理程序，在那里iret返回发生异常的地方继续执行。

内核模式异常处理流程为：

(第一次机会)判断KiDebugRoutine是否为空，不为空就将Context、陷阱帧、异常记录、异常帧、发生异常的模式等压入栈并将控制交给KiDebugRoutine。

|

|

|

若KiDebugRoutine为空(正常的系统这里不为空。正常启动的系统KiDebugRoutine为KdpStub，在Boot.ini里加上/DEBUG启动的系统的KiDebugRoutine为KdpTrap。如果这里为空的话会因为处理不了DbgPrint这类int 0x2d产生的异常而导致系统崩溃)或者KiDebugRoutine未处理异常，则将Context结构和异常记录ExceptionRecord压栈并调用内核模式的RtlDispatchException在内核堆栈中查找基于帧的异常处理例程。

|

|

|

RtlDispatchException调用RtlpGetRegistrationHead从fs:[0](0xffdff000)处获取当前线程异常处理链表指针，并调用RtlpGetStackLimits从0xffdff004和0xffdff008取出当前线程堆栈底和顶。然后开始由异常处理链表指针遍历链表查找异常处理例程(若在XP和2003下先处理VEH再处理SEH)，其实这就是SEH，只是和用户态有一点不同是既没有顶层异常处理例程(TOP LEVEL SEH)也没有默认异常处理例程。然后对每个当前异常处理链表指针检查判断堆栈是否有效(是否超出了堆栈范围或者未对齐)及堆栈是否是DPC堆栈。若0xffdff80c处DpcRoutineActive为TRUE且堆栈顶和底在0xffdff81c处取出的DpcStack到DpcStack-0x3000(一个内核堆栈大小)，若是则更新堆栈顶和底为DpcStack和DpcStack-0x3000并继续处理，否则将异常记录结构里的异常标志ExceptionRecord.ExceptionFlags设置EXCEPTION_STACK_INVALID表示为无效堆栈并返回FALSE。

|

|

|

|

调用异常处理链表上的异常处理例程之前会在异常处理例程链表上插入一个新的节点，对应的异常处理例程是用来处理嵌套异常，也就是在处理异常时发生另一个异常。处理后RtlDispatchException判断异常处理例程的返回值：

若为ExceptionContinueExecution，若异常标志ExceptionRecord.ExceptionFlags未设置EXCEPTION_NONCONTINUABLE不可恢复执行，则返回TRUE到上一层，否则在做了一些工作后调用RtlRaiseException进入到KiDispatchException的第二次机会处理部分。

若为ExceptionContinueSearch，则继续查找异常处理例程。

若为ExceptionNestedException，嵌套异常。保留当前异常处理链表指针为内层异常处理链表并继续查找异常处理例程。当发现当前异常处理链表地址大于保留的内层异常处理链表时，表示当前的异常处理链表比保留的更内层(因为堆栈是由高向低扩展的，地址越高则入栈越早，表示更内层)，则将其值赋予内层异常处理链表指针，除了第一次赋初值外发生修改保留的内层异常处理链表指针这种情况表示嵌套了不止一次的异常。当搜索到新的异常处理链表指针和保留的内层异常处理链表指针相同，则清除ExceptionRecord.ExceptionFlags的嵌套异常位(&(~EXCEPTION_NESTED_CALL))，表示嵌套的异常已经处理完。

其它的返回值都视为无效，调用RtlRaiseException回到KiDispatchException的第二次机会处理部分。

当异常处理链表指针为0xffffffff时，异常处理例程链表已到头。

|
|
|
若RtlDispatchException无法处理异常，(第二次机会)判断KiDebugRoutine是否为空，不为空则交给KiDebugRoutine处理。
|
|
|
当所有例程都无法处理异常时，调用KeBugCheckEx蓝屏，错误代码为KMODE_EXCEPTION_NOT_HANDLED，表示谁也没有处理异常，系统视这个异常为无法恢复的致命异常。至此内核模式下异常处理完毕。

若异常发生在用户模式，同样给予调试器第一次机会和第二次机会调试。只不过因为调试器是用户态调试器，所以通过LPC发送消息给会话管理器smss.exe，再由会话管理器将消息转发给调试器。

用户模式异常处理流程：

若KiDebugRoutine不为空，则不为空就将Context、陷阱帧、异常记录、异常帧、发生异常的模式等压入栈并将控制交给KiDebugRoutine。当处理完毕用Context设置陷阱帧并返回到上一级例程。(第一次机会)否则把异常记录压栈并调用DbgkForwardException，在DbgkForwardException里判断当前线程ETHREAD结构的HideFromDebugger成员如果为FALSE(为TRUE表示该异常对用户调试器不可见)则向当前进程的调试端口(DebugPort)发送LPC消息。
|
|
|
当上一步无法处理异常时将Context结构拷贝到用户堆栈，在堆栈中设置一个陷阱帧，陷阱帧的Eip为Ke(i)UserExceptionDisptcher((i)表示这个函数的Ke和Ki打头的符号其实是一回事)，接着返回陷阱处理程序，由陷阱处理程序iret返回用户态执行Ke(i)UserExceptionDispatcher(这个函数虽然是Ke(Ki)打头，却不是内核里的函数。同样性质特殊的还有Ke(i)RaiseUserExceptionDispatcher、Ke(i)UserCallbackDispatcher、Ke(i)UserApcDispatcher。它们的共同特点就是不是被调用的，而是由内核例程设置了陷阱帧TrapFrame.Eip为该函数后iret执行到这里的)。Ke(i)UserExceptionDisptcher调用RtlDispatchException(用户态下的)寻找堆栈中基于帧的异常处理例程(若在XP和2003下先处理VEH再处理SEH)，这个流程大家应该很熟了，就是搜索SEH链表，若都不处理就调用顶层异常处理(TOP LEVEL SEH)例程。当再无法处理时就调用默认异常处理例程终止进程(有VC时这里就换成了VC)。有点不同的是用户态下的RtlDispatchException只判断返回值是ExceptionContinueExecution还是ExceptionContinueSearch。若RtlDispatchException找到异常处理例程能够处理异常，则调用ZwContinue按照设置好的Context结构继续执行，否则调用ZwRaiseException，并且把第三个布尔参数设为FALSE，表示进入第二次机会处理。
|
|
|
ZwRaiseException经过一系列调用最后直接调用KiDispatchException，由于把布尔值FirstChance设置为FALSE，在KiDispatchException里直接进入第二次机会处理。
|
|
|
(第二次机会)向进程的DebugPort发消息，若无法处理，则改向进程的ExceptionPort发消息(这里同样如果该异常对用户调试器不可见，则只会发送到ExceptionPort)。DebugPort和ExceptionPort的区别在于，若向ExceptionPort发消息，先停止目标进程所有线程的执行，直到收到回应消息后线程才恢复执行，而向DebugPort发消息则不需要停止线程运行。还有DebugPort是向会话管理器发消息，而ExceptionPort是向Win32子系统发消息，当向ExceptionPort发消息时，已经不给用户态调试器任何机会了:)。
|
|
|
当异常还是无法处理，则终止当前线程，并调用KeBugCheckEx蓝屏，错误代码为KMODE_EXCEPTION_NOT_HANDLED。至此用户模式下异常处理完毕。

有一点要说明的是，这里只是列出了操作系统的流程。如果现在去看比方说驱动或者一个应用程序里，加入了__try/__except代码，却没发现SEH上的节点对应的异常处理例程和自己写的有啥关系。其中的原因就是因为M$的编译器(比方说VC++、DDK)在系统内部封装了SEH的机制。在异常处理例程链表节点上的异常处理例程实际上是_except_handler3，这个函数自己在内部又实现了一套类似SEH的机制，就是通过对每个函数建立一个表，包括了该函数中所有__try块对应的过滤异常例程指针(__try()括号里的对应函数，如果是括号里是EXCEPTION_EXECUTE_HANDLER之类的则该过滤异常例程很简单地把eax赋为相应的1、0或-1然后返回，对应了EXCEPTION_EXECUTE_HANDLER、EXCEPTION_CONTINUE_SEARCH、EXCEPTION_CONTINUE_EXECUTE)和处理例程指针(__except{}和__finally{}里面代码的地址)。所以对应每个被调用到的函数都在SEH链表上注册一个节点，并建立一张表。象诸如EXCEPTION_EXECUTE_HANDLER、EXCEPTION_CONTINUE_SEARCH、EXCEPTION_CONTINUE_EXECUTE实际上是返回给_except_handler3的返回值，而不是返回给RtlDispatchException的。

看得有点晕吧，呵呵。总结一下流程(指异常未被处理的完整流程，若异常在任一个环节被处理都从该环节上退出继续原来正常程序代码的执行)：

内核模式下：
KiDispatchException->(第一次机会)KiDebugRoutine->RtlDispatchException在内核堆栈寻找SEH(VEH)->(第二次机

会)KiDebugRoutine->KeBugCheckEx

用户模式下：
KiDispatchException->KiDebugRoutine->(第一次机会)发送消息到进程调试端口->RtlDispatchException在用户堆栈寻找SEH(VEH)->ZwRaiseException回到KiDispatchException->(第二次机会)发送消息到进程调试或异常端口->KeBugCheckEx。

就这么简单，呵呵。举个例子来说明异常的处理。

我们程序中通过加入int 0x3来对程序进行调试，那么int 0x3产生的异常是怎么处理的呢？

若int 0x3发生在内核模式下(驱动程序里)，又分为内核调试器是否加载。内核调试器未加载时，KiDebugRoutine(KdpStub)不处理int 0x3异常，则在堆栈中搜索由驱动程序编写者注册的SEH，若也没有对int 0x3异常的处理，则只好调用KeBugCheckEx蓝屏，错误代码是KMODE_EXCEPTION_NOT_HANDLED(谁都不干活，系统只好悲愤地自我了结了)。若有内核调试器加载，则KiDebugRoutine(KdpTrap)可以处理int 0x3异常，异常处理到这里被正常返回。处理方法是将当前的处理器状态(比如各寄存器等重要信息)发送给通过串口相连的主机上的内核调试器，并一直在等待内核调试器的回应，系统这时在KdpTrap里调用一个Kd函数KdpSendWaitContinue循环等待来自串口的数据(内核调试器和被调试系统通过串口联系)，直到内核调试器下达继续执行的命令，系统可以正常从int 0x3后面一条指令执行。

若int 0x3发生在用户模式下，也分为用户调试器是否加载。用户调试器未加载时，KiDebugRoutine不处理int 0x3异常，且用户进程无DebugPort，将记录异常的相关结构拷入用户堆栈交由用户模式的RtlDispatchException搜索用户堆栈中的SEH。若也没有对int 0x3异常的处理，则调用默认异常处理例程，默认情况下是将发生异常的进程终止。若有用户调试器加载，则向进程的DebugPort或ExceptionPort发送有关异常的LPC消息并判断发送函数的返回状态，若用户调试器继续执行，则返回STATUS_SUCCESS，内核视为异常已解决继续执行。

这里也说明了一点，就是相同的错误，发生在?核态下比发生在用户态下致命得多。其它异常的处理流程基本也一样。少数不一样的异常象DebugPrint、DebugPrompt、加载和卸载symbol等是通过调用DebugService的(这实际上是通过产生一个异常int 0x2d)。可以在KdpStub中就被处理(处理很简单，只是把Context结构的Eip加一略过当前int 0x2d后那条int 0x3指令)，若在KdpTrap中被处理则是和内核调试器更进一步的交互。关于KdpStub、KdpTrap和DebugService更详细的介绍参见我的另一篇文章《windows内核调试器原理浅析》。

后记：
前几天翻硬盘时找出了这篇半年前写了大半的文章。当时因为还有些不明白的东西，所以没写完，结果一放就放了半年。现在翻出来写完后弄明白了不少东西，也把心得拿出来供大家参考，就当是抛砖引玉吧:)

QQ：27324838
EMAIL：kinvis@hotmail.com
本文转自
http://bbs.pediy.com/showthread.php?t=16628