漫谈兼容内核之二十六:
Windows 的结构化异常处理(三)

毛德操

前两篇漫谈介绍了在内核中怎样利用 SEH 机制和宏操作_SEH_TRY、_SEH_HANDLE、以及_SEH_END 为可能引起异常的代码提供保护，以及在异常果真发生时的处理过程。读者现在对于 SEH 机制的原理应该已经有了一些了解。前面也曾提到，Windows 操作系统在用户空间也提供了同样的功能，而且实现的方式也基本相同。但是，在前两篇漫谈中读者所看到的是当异常发生于系统空间时的情景，而并没有看到当异常发生于用户空间时怎样经过内核被提交到用户空间、并在那里得到处理的过程。

我们知道，异常就像中断，不管是什么原因(“软异常”除外)所引起，一旦发生首先进入的是内核中的异常响应/处理程序的入口，这就是类似于 KiTrap0()那样的底层内核函数，只是因为引起异常的原因不同而进入不同的入口，就像对于不同的中断向量有不同的入口一样。在内核中，仍以页面异常为例，正如读者已经看到，CPU 会从 KiTrap14()进入函数 KiPageFaultHandler()。在那儿，如果所发生的并非如“缺页”或“写时复制(Copy-On-Write)”那样的“正常”异常，就要根据 CPU 在发生异常时所处的空间而分别调用 KiKernelTrapHandler()或 KiUserTrapHandler()。如果调用的是 KiKernelTrapHandler()，就会顺着 KPCR 数据结构中的“异常(处理)队列”、即 ExceptionList，依次让各个节点认领。如果被认领，就会通过 SEHLongJmp()长程跳转到当初通过_SEH_HANDLE{}给定的代码中。这读者已经见到了。

但是，如果异常发生于用户空间，受到调用的就是 KiUserTrapHandler()。

[_KiTrap14() > KiPageFaultHandler() > KiUserTrapHandler()]

```
ULONG NTAPI
KiUserTrapHandler(PKTRAP_FRAME Tf, ULONG ExceptionNr, PVOID Cr2)
{
  EXCEPTION_RECORD Er;

  if (ExceptionNr == 0)
    {
      Er.ExceptionCode = STATUS_INTEGER_DIVIDE_BY_ZERO;
    }
  else if (ExceptionNr == 1)
    {
      Er.ExceptionCode = STATUS_SINGLE_STEP;
    }
  else if (ExceptionNr == 3)
    {
      Er.ExceptionCode = STATUS_BREAKPOINT;
    }
  else if (ExceptionNr == 4)
```

```
    {
      Er.ExceptionCode = STATUS_INTEGER_OVERFLOW;
    }
  else if (ExceptionNr == 5)
    {
      Er.ExceptionCode = STATUS_ARRAY_BOUNDS_EXCEEDED;
    }
  else if (ExceptionNr == 6)
    {
      Er.ExceptionCode = STATUS_ILLEGAL_INSTRUCTION;
    }
  else
    {
      Er.ExceptionCode = STATUS_ACCESS_VIOLATION;
    }
  Er.ExceptionFlags = 0;
  Er.ExceptionRecord = NULL;
  Er.ExceptionAddress = (PVOID)Tf->Eip;
  if (ExceptionNr == 14)
    {
      Er.NumberParameters = 2;
      Er.ExceptionInformation[0] = Tf->ErrCode & 0x1;
      Er.ExceptionInformation[1] = (ULONG)Cr2;
    }
  else
    {
      Er.NumberParameters = 0;
    }

  /* FIXME: Which exceptions are noncontinuable? */
  Er.ExceptionFlags = 0;

  KiDispatchException(&Er, 0, Tf, UserMode, TRUE);
  return(0);
}
```

显然，这个函数不是仅为 14 号异常所用的。只要是发生于用户空间，别的异常也会进入这个函数。同样，这里也是先在堆栈上准备好一个“异常记录块”，然后调用 KiDispatchException()，只不过这一次的第四个实际参数是 UserMode。读者在上一篇漫谈中看到，KiKernelTrapHandler()也调用这同一个函数，但是那里的第四个实际参数是 KernelMode。

所以，KiDispatchException()是个十分重要的函数，现在我们又要回过头来看这个函数的代码。当然，这一次观察的角度不同了。

[_KiTrap14() > KiPageFaultHandler() > KiUserTrapHandler() > KiDispatchException()]

```
VOID NTAPI
KiDispatchException(PEXCEPTION_RECORD ExceptionRecord,
                    PKEXCEPTION_FRAME ExceptionFrame,
                    PKTRAP_FRAME TrapFrame,
                    KPROCESSOR_MODE PreviousMode,
                    BOOLEAN FirstChance)
{
   . . . . . .

   /* Set the context flags */
   Context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS;

   /* Check if User Mode */
   if (PreviousMode == UserMode)
   {
        /* Add the FPU Flag */
        Context.ContextFlags |= CONTEXT_FLOATING_POINT;
        if (KeI386FxsrPresent) Context.ContextFlags |= CONTEXT_EXTENDED_REGISTERS;
   }

   /* Get a Context */
   KeTrapFrameToContext(TrapFrame, ExceptionFrame, &Context);

   /* Handle kernel-mode first, it's simpler */
   if (PreviousMode == KernelMode)
   {
         . . . . . .
   }
   else
   {
        /* User mode exception, was it first-chance? */
        if (FirstChance)
        {
             /* Enter Debugger if available */
             Action = KdpEnterDebuggerException(ExceptionRecord, PreviousMode,
                                                &Context, TrapFrame, TRUE, TRUE);
             /* Exit if we're continuing */
             if (Action == kdContinue) goto Handled;

             /* FIXME: Forward exception to user mode debugger */

             /* Set up the user-stack */
```

```
        _SEH_TRY
        {
            /* Align context size and get stack pointer */
            Size = (sizeof(CONTEXT) + 3) & ~3;
            Stack = (Context.Esp & ~3) - Size;
            DPRINT("Stack: %lx\n", Stack);

            /* Probe stack and copy Context */
            ProbeForWrite((PVOID)Stack, Size, sizeof(ULONG));
            RtlCopyMemory((PVOID)Stack, &Context, sizeof(CONTEXT));

            /* Align exception record size and get stack pointer */
            Size = (sizeof(EXCEPTION_RECORD) -
                    (EXCEPTION_MAXIMUM_PARAMETERS -
                    ExceptionRecord->NumberParameters) * sizeof(ULONG) + 3) & ~3;
            NewStack = Stack - Size;
            DPRINT("NewStack: %lx\n", NewStack);

            /* Probe stack and copy exception record. Don't forget to add the two params */
            ProbeForWrite((PVOID)(NewStack - 2 * sizeof(ULONG_PTR)),
                          Size +    2 * sizeof(ULONG_PTR),
                          sizeof(ULONG));
            RtlCopyMemory((PVOID)NewStack, ExceptionRecord, Size);

            /* Now write the two params for the user-mode dispatcher */
            *(PULONG_PTR)(NewStack - 1 * sizeof(ULONG_PTR)) = Stack;
            *(PULONG_PTR)(NewStack - 2 * sizeof(ULONG_PTR)) = NewStack;

            /* Set new Stack Pointer */
            KiEspToTrapFrame(TrapFrame, NewStack - 2 * sizeof(ULONG_PTR));

            /* Set EIP to the User-mode Dispathcer */
            TrapFrame->Eip = (ULONG)KeUserExceptionDispatcher;
            UserDispatch = TRUE;
            _SEH_LEAVE;
        }
        _SEH_HANDLE
        {
            /* Do second-chance */
        }
        _SEH_END;
    }

    /* If we dispatch to user, return now */
```

```
        if (UserDispatch) return;

        /* FIXME: Forward the exception to the debugger for 2nd chance */

        /* 3rd strike, kill the thread */
        DPRINT1("Unhandled UserMode exception, terminating thread\n");
        ZwTerminateThread(NtCurrentThread(), ExceptionRecord->ExceptionCode);
        KEBUGCHECKWITHTF(KMODE_EXCEPTION_NOT_HANDLED,
                         ExceptionRecord->ExceptionCode,
                         (ULONG_PTR)ExceptionRecord->ExceptionAddress,
                         ExceptionRecord->ExceptionInformation[0],
                         ExceptionRecord->ExceptionInformation[1],
                         TrapFrame);
    }

Handled:
    /* Convert the context back into Trap/Exception Frames */
    KeContextToTrapFrame(&Context, NULL, TrapFrame, Context.ContextFlags, PreviousMode);
        return;
}
```

首先通过 KeTrapFrameToContext()从堆栈上的异常框架整理出一个上下文数据结构来。不过，对于用户空间的异常处理上下文中需要有更全面的信息，所以在调用这个函数之前把上下文结构中的 CONTEXT_FLOATING_POINT 等标志位设成 1。这些标志位实质上就是对 KeTrapFrameToContext()的指令。

这一次我们把注意集中在 PreviousMode 为 UserMode 的分支上。

读者不妨想想，对于发生于用户空间的异常，这里应该做些什么。显然，用户空间的异常不应靠内核里面的程序处理，应用软件理应为此作好了准备。前面讲过，Windows 的 SEH 机制并不是仅为内核而设计的，用户空间的程序同样可以使用类似于_SEH_TRY{} _SEH_HANDLE{} _SEH_END 那样的手段为应用程序提供保护。事实上，在通过 NtCreateThread()创建的线程首次被调度运行时，整个线程的执行都是作为一个 SEH 域而受到保护的：

```
VOID STDCALL
BaseProcessStartup(PPROCESS_START_ROUTINE lpStartAddress)
{
    UINT uExitCode = 0;

    _SEH_TRY
    {
        /* Set our Start Address */
        NtSetInformationThread(NtCurrentThread(),ThreadQuerySetWin32StartAddress,
                               &lpStartAddress, sizeof(PPROCESS_START_ROUTINE));
        /* Call the Start Routine */
```

```
            uExitCode = (lpStartAddress)();
        }
        _SEH_EXCEPT(BaseExceptionFilter)
        {
            /* Get the SEH Error */
            uExitCode = _SEH_GetExceptionCode();
        }
        _SEH_END;

        /* Exit the Process with our error */
        ExitProcess(uExitCode);
}
```

这里 BaseProcessStartup()是所有线程在用户空间的总入口，而 lpStartAddress 是具体线程的代码入口。这里引用的宏操作之一是_SEH_EXCEPT，而不是_SEH_HANDLE，因而可以提供一个过滤函数。这个过滤函数是 BaseExceptionFilter()，它又通过一个函数指针调用实际的过滤函数，默认为 UnhandledExceptionFilter()。而 UnhandledExceptionFilter()在一般情况下都返回 EXCEPTION_EXECUTE_HANDLER。不过，应用程序可以通过一个函数 SetUnhandledExceptionFilter()将其替换成自己想要的过滤函数。

与此相应，用户空间的每个线程都有一个 ExceptionList，只不过这个队列在每个线程的 TEB 中，而不是在 KPCR 中。既然内核中的 ExceptionList 是由 KiDispatchException()加以处理的，用户空间就应该有个类似于 KiDispatchException()的函数。事实上，动态连接库 ntdll.dll 中的 KiUserExceptionDispatcher()就是用户空间 SEH 处理的总入口。

可是，尽管是发生于用户空间的异常，对异常的初期响应和处理毕竟是在内核中，现在的目的就是要从内核中的 KiDispatchException()启动用户空间这个函数的执行。

对于内核中的 KiDispatchException()，这就是针对用户空间异常的主要操作。不过具体的实现还要再复杂一些，就像针对系统空间异常一样，内核中涉及用户空间异常的处理也分三步：

第一步、参数 FirstChance 为 1 时，先通过 KdpEnterDebuggerException()交由内核调试程序(Kernel Debugger)处理。如果内核调试程序解决了问题、或者认为无需提交用户空间，则返回值就是 kdContinue，这就行了。否则就要把异常提交给用户空间，由用户空间的程序加以处理。代码中的_SEH_TRY{}里面就是启动用户空间异常处理的过程。对于绝大多数的用户空间异常，这就可以了，因为用户空间的 ExceptionList 中应该有节点可以认领和处理本次异常，例如通过预先的安排实施用户空间的长程跳转。

第二步、然而，万一用户空间处理不了，例如 ExceptionList 中没有安排下可以认领、处理本次异常的节点，就会通过 RtlRaiseException()、从而通过系统调用 ZwRaiseException()发起一次“软异常”(见后)，把问题交还内核。此时 CPU 再次进入 KiDispatchException()，但是此时的实际参数 FirstChance 为 0，所以直接进入第二步措施。在 Windows 内核中，这第二次努力是通过进程间通信向用户空间的调试程序(Debugger)发送一个报文、将其唤醒，由调试程序作进一步的处理。例如，对于由用户空间调试程序设置的断点(INT3)，就只能由用户空间调试程序加以处理。不过，在 ReactOS 0.3.0 版的代码中这一步尚未实现，所以这里有个注释说：“FIXME: Forward the exception to the debugger for

2nd chance”。

第三步、如果用户空间调试程序不存在，或者也不能解决，那就属于不可恢复的问题了。于是就有第三步措施，那就是通过 ZwTerminateThread()结束当前线程的运行。正常情况下针对当前线程本身的 ZwTerminateThread()是不返回的；而倘若竟然返回了，那对于整个系统都是不可恢复的问题了，所以通过宏操作 KEBUGCHECKWITHTF()显示出错信息、转储(Dump)当时的内存映像，并进入一个 Ke386HaltProcessor()的无限循环。换言之，整个系统就“死”了。

显然，这里最关键的一步、也是最有希望的一步，是把异常提交给用户空间。怎么提交呢？首先要把上下文数据结构 Context 和异常纪录块 ExceptionRecord 拷贝到用户空间堆栈上去，再在用户空间堆栈上安上两个指针，分别指向这两个数据结构的用户空间副本，并相应调整异常框架中的用户空间堆栈指针。下面就会看到，这两个指针将被用作用户空间的函数调用参数。最后、也是最关键的，则是把异常框架中的用户空间返回地址设置成函数指针 KeUserExceptionDispatcher 所指向的函数。顺利完成了这些准备以后，就把局部量 UserDispatch 设成 1，因此紧接着就从本次异常处理返回了。当然，这是返回到了指针 KeUserExceptionDispatcher 所指向的函数中。已经熟悉 APC 机制的读者应该很容易由此联想到对用户空间 APC 函数的调用。事实上也确实非常相似，如果说 APC 相当于对用户空间软件的中断机制，则异常的提交就相当于对用户空间软件的异常机制。

但是，将两个数据结构复制到用户空间堆栈的过程本身又是有可能会引起异常的，所以这里又用_SEH_TRY{} _SEH_HANDLE{} _SEH_END 将这段代码保护起来，为可能发生的异常作好准备。倘若果真在此过程中发生(系统空间)异常，就直接进入上述的第二步努力。

再看函数指针 KeUserExceptionDispatcher。这是内核中的一个全局量，实际上提供了 Ntdll.dll 映像中的 KiUserExceptionDispatcher()在当前进程内的地址。就像对于 LdrInitializeThunk()和 KiUserApcDispatcher()以及其它几个函数一样，这也是在内核的初始化过程中，初次装入 Ntdll.dll 的映像的时候从其映像中获取的：

```
[KiSystemStartup() > ExpInitializeExecutive() > PsLocateSystemDll()
> PspLookupKernelUserEntryPoints()]

NTSTATUS STDCALL INIT_FUNCTION
PspLookupKernelUserEntryPoints(VOID)
{
    . . . . . .
    /* Retrieve ntdll's startup address */
    RtlInitAnsiString(&ProcedureName, "LdrInitializeThunk");
    Status = LdrGetProcedureAddress((PVOID)PspSystemDllBase, &ProcedureName,
                                    0, &PspSystemDllEntryPoint);
    . . . . . .
    /* Get User APC Dispatcher */
    RtlInitAnsiString(&ProcedureName, "KiUserApcDispatcher");
    Status = LdrGetProcedureAddress((PVOID)PspSystemDllBase, &ProcedureName,
                                    0, &KeUserApcDispatcher);
    . . . . . .
    DPRINT("Getting Entrypoint\n");
    RtlInitAnsiString(&ProcedureName, "KiUserExceptionDispatcher");
```

```
    Status = LdrGetProcedureAddress((PVOID)PspSystemDllBase, &ProcedureName,
                                    0, &KeUserExceptionDispatcher);
    . . . . . .
    /* Get Callback Dispatcher */
    RtlInitAnsiString(&ProcedureName, "KiUserCallbackDispatcher");
    Status = LdrGetProcedureAddress((PVOID)PspSystemDllBase, &ProcedureName,
                                    0, &KeUserCallbackDispatcher);
    . . . . . .
    /* Get Raise Exception Dispatcher */
    RtlInitAnsiString(&ProcedureName, "KiRaiseUserExceptionDispatcher");
    Status = LdrGetProcedureAddress((PVOID)PspSystemDllBase, &ProcedureName,
                                    0, &KeRaiseUserExceptionDispatcher);
    . . . . . .
    return(STATUS_SUCCESS);
}
```

顺便提一下，这里的另一个指针 KeRaiseUserExceptionDispatcher 提供了 Ntdll.dll 映像中函数 KiRaiseUserExceptionDispatcher()的地址，但是似乎只在 NtClose()中用到。

于是，CPU 从本次异常返回，回到用户空间时就进入了 KiUserExceptionDispatcher()，那就是用户空间的异常响应/处理程序的入口。与内核中有_KiTrap0()、_KiTrap14()等等入口不同，用户空间就只有这么一个总的入口。至于发生异常的原因与性质，则由异常纪录块中的 ExceptionCode 字段加以界定。

```
VOID NTAPI
KiUserExceptionDispatcher(PEXCEPTION_RECORD ExceptionRecord,
                                                PCONTEXT Context)
{
    EXCEPTION_RECORD NestedExceptionRecord;
    NTSTATUS Status;

    /* call the vectored exception handlers */
    if(RtlpExecuteVectoredExceptionHandlers(ExceptionRecord, Context)
                                                != ExceptionContinueExecution)
    {
        goto ContinueExecution;
    }
    else
    {
        /* Dispatch the exception and check the result */
        if(RtlDispatchException(ExceptionRecord, Context))
        {
ContinueExecution:
            /* Continue executing */
            Status = NtContinue(Context, FALSE);
```

```
        }
        else
        {
            /* Raise an exception */
            Status = NtRaiseException(ExceptionRecord, Context, FALSE);
        }
    }

    /* Setup the Exception record */
    NestedExceptionRecord.ExceptionCode = Status;
    NestedExceptionRecord.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
    NestedExceptionRecord.ExceptionRecord = ExceptionRecord;
    NestedExceptionRecord.NumberParameters = Status;

    /* Raise the exception */
    RtlRaiseException(&NestedExceptionRecord);
}
```

注意这里的异常纪录块和上下文结构都是从内核复制到用户空间堆栈上的。

用户空间的异常机制是对于系统空间的异常机制的模拟。在内核中，并非所有的异常都是一来就进入“基于 SEH 框架(Frame-Based)”的异常处理，而是先进入_KiTrap14()等等类似于向量中断的入口，在那里可以被拦截进行一些优先的处理，例如页面换入和对 Copy-On-Write 页面的处理等等。这些处理是全局性质的，而不属于某个 SEH 域。这相当于是一层全局性的过滤。只有不属于这个层次的异常才会进入基于 SEH 框架的异常处理。

为此，用户空间的每个进程还有一个“向量式异常处理”程序入口的队列 RtlpVectoredExceptionHead，队列中的每个节点都指向一个具体的异常处理函数。就像内核中“登记”中断处理程序一样，在用户空间也可以通过 RtlAddVectoredExceptionHandler()登记向量式的异常处理程序，这样就可以在异常的源头上加以拦截(不过在 Windows 的资料中并未见到有用户空间向量式异常处理的存在，但是在 Wine 的代码中倒是有)。这里 RtlpExecuteVectoredExceptionHandlers()的作用就是先扫描这个队列，让队列中的各个节点依次认领，如果被认领并且得到了处理，就不再进入基于 SEH 框架、即 ExceptionList 的处理了(代码中 if 语句的条件“!=”疑为“==”之误，待考)，所以处理完之后就通过系统调用 NtContinue()返回内核，而进入内核以后又会返回用户空间当初因异常而被中断的地方。所以，系统调用 NtContinue()实质上就是对“中断返回”的模拟。在讲述 APC 机制的时候，读者已经看到用户空间的 APC 函数也是通过 NtContinue()实行“中断返回”的(相当于 Linux 的系统调用 sigreturn())。

要是在向量式异常处理队列中没有得到处理，那就要通过 RtlDispatchException()进行基于 SEH 框架的异常处理了。

读者在上一篇漫谈中已经看到过 RtlDispatchException()的代码，大致如下：

[KiUserExceptionDispatcher() > RtlDispatchException()]

```
RtlDispatchException(IN PEXCEPTION_RECORD ExceptionRecord,
                                                    IN PCONTEXT Context)
```

```
{
    . . . . . .

    /* Get the current stack limits and registration frame */
    RtlpGetStackLimits(&StackLow, &StackHigh);
    RegistrationFrame = RtlpGetExceptionList();

    /* Now loop every frame */
    while (RegistrationFrame != EXCEPTION_CHAIN_END)
    {
        . . . . . .     /*  扫描 ExceptionList，直至有节点认领/处理本次异常  */
    }
    /* Unhandled, return false */
    return FALSE;
}
```

但是，那是在内核中，作为异常处理的第一步措施(FirstChance)，由 KiDispatchException()针对发生于系统空间的异常而加以调用的。而现在，则显然是在用户空间调用这个函数。那莫非在用户空间就可以直接调用内核中的函数吗？不是的，这个 RtlDispatchException()并非那个 RtlDispatchException()，这只是“代码重用”而已。这个 RtlDispatchException()编译以后连接在 Ntdll.dll 的映像中(不过并不导出)，并且所处理的 ExceptionList 是在用户空间，所使用的堆栈也是用户空间堆栈。

ExceptionList 是通过 RtlpGetExceptionList()获取的。

```
_RtlpGetExceptionList@0:

    /* Return the exception list */
    mov eax, [fs:TEB_EXCEPTION_LIST]
    ret
```

常数 TEB_EXCEPTION_LIST 定义为 0，所以总之就是[fs:0]。当 CPU 运行于系统空间时，其段寄存器 FS 指向 KPCR；而当运行于用户空间时则指向 TEB。所以，不管是在系统空间还是用户空间，RtlpGetExceptionList()总能取到各自的 ExceptionList 指针。

RtlpGetStackLimits()也是一样：

```
_RtlpGetStackLimits@8:
    /* Get the stack limits */
    mov eax, [fs:TEB_STACK_LIMIT]
    mov ecx, [fs:TEB_STACK_BASE]
    /* Return them */
    mov edx, [esp+4]
    mov [edx], eax
    mov edx, [esp+8]
    mov [edx], ecx
```

```
/* return */
ret 8
```

常数 TEB_STACK_LIMIT 定义为 8，TEB_STACK_BASE 定义为 4，分别是 StackLimit 和 StackBase 两个字段相对于 TEB 起点的位移。而在 KPCR 数据结构中同样也有这两个字段，而且也在相同的位置上。当然，前者用于用户空间堆栈，而后者用于系统空间堆栈。由此可见，这些数据结构确实是经过精心设计的。另一方面，在处理 ExceptionList 的过程中所涉及的各种数据结构也都既用于系统空间又用于用户空间。

这样，同一个函数 RtlDispatchException()的代码就既可用于内核，也可用于用户空间的程序库。既然如此，我们也就不必再看一遍 RtlDispatchException()的代码了。

回到 KiUserExceptionDispatcher()，执行 RtlDispatchException()的结果主要有三种可能。

一种是当前异常为 ExceptionList 中的某个节点所认领、即顺利通过了其过滤函数的过滤，并执行了长程跳转。显然，在这种情况下 RtlDispatchException()不会返回，包括 KiUserExceptionDispatcher()在内的函数调用框架均因长程跳转而被跨越和丢弃。不仅如此，复制到用户空间堆栈上的两个数据结构也因为长程跳转而被丢弃。

第二种是被某个节点认领了，也作了某些处理，也许还执行了这个节点的善后函数，但是并未执行长程跳转，此时 RtlDispatchException()返回 TRUE，于是便通过系统调用 NtContinue()完成“中断返回”。由于堆栈上的上下文数据结构含有当初因异常而进入系统空间时所保留的现场，最后就返回到了(用户空间中)当初因异常而被中断了的地方。

第三种是没有任何节点认领，RtlDispatchException()中的 while 循环穷尽了所有的节点，所以返回 FALSE。这一定是出了什么问题，例如 ExceptionList 被损坏了，因为在正常情况下这是不应该发生的。前面已经提及，在用户空间，整个线程的代码都是放在一个 SEH 域里执行的，所以 ExceptionList 一定是非空，并且其最后一个节点(最早进入的)就是前面在 BaseProcessStartup()中进入的那个 SEH 域。这个 SEH 域的过滤函数一般都返回 _SEH_EXECUTE_HANDLER，所以是来者不拒。因此，既然穷尽了 ExceptionList，就一定是发生了严重的问题，所以要通过系统调用 NtRaiseException()引起一次软异常，以进入针对当前异常的第二步措施。注意这里调用 NtRaiseException()时的第三个参数为 FALSE，表示这已经不是第一次尝试。

此外还有一种可能，就是在 RtlDispatchException()内部就已调用了 RtlRaiseException()，例如针对某个节点的 RtlpExecuteHandlerForException()返回 ExceptionContinueExecution，而异常纪录块中 ExceptionFlags 的标志位 EXCEPTION_NONCONTINUABLE 却又是 1，此时就要通过 RtlRaiseException()引起原因为 STATUS_NONCONTINUABLE_EXCEPTION 的软异常。

在正常的情况下，对于调用点而言，系统调用 NtContinue()和 NtRaiseException()是不返回的，返回就说明系统调用本身(而不是对异常的处理)出了错，例如参数有问题。因此，要是果真从这两个系统调用返回，那么正常的处理已经山穷水尽，只能又求助于软异常了，这就是下面对于 RtlRaiseException()的调用。

调用 RtlRaiseException()的目的在于模拟一次异常，这“异常”已经不是原来所要处理的异常，而是对原来的异常处理不下去了。此时需要解决的已经不是引起原来那个异常的问题，而是为什么处理不下去的问题。所以原来的“异常代码”ExceptionCode 可能是 STATUS_ACCESS_VIOLATION，而现在的 ExceptionCode 则是从 NtRaiseException()或 NtContinue()返回的出错代码 Status。同样的道理，通过 RtlDispatchException()发起的则是 STATUS_NONCONTINUABLE_EXCEPTION 软异常。正因为现在要发起的异常并非原来要处理的异常，所以要为 RtlRaiseException()准备一个新的异常纪录块。

另一方面，之所以通过 RtlRaiseException()发起各种不同类型的软异常，是建立在一个前提上的，那就是相信这个异常最终总会得到处理。这包括两个方面，首先是 ExceptionList 中应该有能够认领/处理此类异常的节点，即已经准备好了应对此类问题所需的程序。其次，即使 ExceptionList 中没有这样的节点，终归也有应对的办法，那就是前面讲的三个步骤，包括 Debug、结束当前线程、甚至“死机”在内。注意“死机”也有受控和失控之分，使 CPU 进入停机状态是有控制的死机，那也比任由 CPU 在程序中乱跳一气要好。

然而，尽管现在要发起的是与原来不同的异常，但是这毕竟是在处理原来异常的过程中出的问题，与原来的异常是有关系的，应该让认领新发起异常的处理者知道这一点。所以异常纪录块中有个指针 ExceptionRecord，遇到像这样的情况就让新的异常纪录块通过这个指针指向原来的异常纪录块。所以，软异常的纪录块可以成串，每个纪录块都可以指向本次异常的祸端，只有硬异常的纪录块不会指向别的纪录块。

显然，RtlRaiseException()是个重要的函数。同样，这个函数的代码也是内核和用户空间两栖的。注意下列的调用路线只是许多情景中的一种，实际上调用这个函数的地方很多。

[KiUserExceptionDispatcher() > RtlRaiseException()]

```
VOID NTAPI RtlRaiseException(PEXCEPTION_RECORD ExceptionRecord)
{
    CONTEXT Context;
    . . . . . .

    /* Capture the context */
    RtlCaptureContext(&Context);
    /* Save the exception address */
    ExceptionRecord->ExceptionAddress = RtlpGetExceptionAddress();
    /* Write the context flag */
    Context.ContextFlags = CONTEXT_FULL;

    /* Check if we're being debugged (user-mode only) */
    if (!RtlpCheckForActiveDebugger(TRUE))
    {
        /* Raise an exception immediately */
        Status = ZwRaiseException(ExceptionRecord, &Context, TRUE);
    }
    else
    {
        /* Dispatch the exception and check if we should continue */
        if (RtlDispatchException(ExceptionRecord, &Context))
        {
            /* Raise the exception */
            Status = ZwRaiseException(ExceptionRecord, &Context, FALSE);
        }
        else
```

```
        {
            /* Continue, go back to previous context */
            Status = ZwContinue(&Context, FALSE);
        }
    }

    /* If we returned, raise a status   */
    RtlRaiseStatus(Status);
}
```

调用参数 ExceptionRecord 指向一个异常纪录块。如上所述，这个纪录块中记载着异常的性质，并且一般都通过指针指向另一个异常纪录块。

首先通过 RtlCaptureContext()获取当时的上下文，并通过 RtlpGetExceptionAddress()获取 RtlRaiseException()的返回地址，以此作为发生本次异常的地址。

具体的操作视 RtlpCheckForActiveDebugger()的结果而异。这个函数有系统空间和用户空间两个不同的版本。目前系统空间的版本只是返回调用参数，所以在这里总是返回 TRYE。而用户空间的版本则返回 NtCurrentPeb()->BeingDebugged，所以只在受调试时返回 TRUE。

所以，如果是在用户空间调用 RtlRaiseException()，而又不是在受调试，就启动系统调用 ZwRaiseException()。注意此时的第三个参数为 TRUE，表示这是第一次努力。可见，在这种情况下 RtlRaiseException()是通过系统调用 ZwRaiseException()实现的。而且所用的函数名是 ZwRaiseException()，在用户空间和系统空间都可以调用。

反之如果是在系统空间，或者虽在用户空间但是在受调试，那就先直接调用 RtlDispatchException()，看看本空间的 ExceptionList 中是否有节点可以认领和处理。如果有、并且实施了长程跳转，那就不返回了。而如果返回的话，则视返回值为 TRUE 或 FALSE 而分别启动系统调用 ZwRaiseException()或 ZwContinue()。如果 RtlDispatchException()返回 TRUE，就说明已经得到 ExceptionList 中某个节点的认领、但是并未执行长程跳转而返回 ExceptionContinueExecution，此时通过 ZwRaiseException()进行第二次努力，注意这里调用 ZwRaiseException()时的第 3 个参数为 FALSE，表示对 ExceptionList 的搜索已经失败，下面该采取第二步措施了。而如果 RtlDispatchException()返回 FALSE，则说明 ExceptionList 中根本就没有节点可以认领这次异常，所以就直接通过 ZwContinue()返回到刚才获取的那个上下文中，那就是 RtlRaiseException()本次被调用的地方。

在我们现在所考察的情景中，RtlRaiseException()是在 KiUserExceptionDispatcher()中受到调用的，并且在此之前已经有过对 ZwRaiseException()或 ZwContinue()的调用。那么现在又来调用 ZwRaiseException()或 ZwContinue()是否重复呢？不重复，因为使用的是不同的异常纪录块，不同的异常代码，这是为不同原因而发起的软异常。

还有，为什么在受调试的情况下反倒要直接调用 RtlDispatchException()，避开调试处理呢？因为调试工具所安排的都是针对正常条件下的异常，例如页面访问异常、除数为 0 等等，而现在发生的是在处理异常的过程中本来不应该发生的问题，是属于系统的问题，那不是属于应该由调试工具来处理的问题。而如果直接通过 ZwRaiseException()发起一次软中断，则必将首先进入调试程序。

如果这一次对 ZwRaiseException()或 ZwContinue()的调用又失败了(居然返回了)，那么问题就又更严重了，所以此时再以 ZwRaiseException()或 ZwContinue()返回的出错代码为参数调用 RtlRaiseStatus()。

[KiUserExceptionDispatcher() > RtlRaiseException() > RtlRaiseStatus()]

```
VOID NTAPI RtlRaiseStatus(NTSTATUS Status)
{
    EXCEPTION_RECORD ExceptionRecord;
    CONTEXT Context;
    DPRINT1("RtlRaiseStatus(Status 0x%.08lx)\n", Status);

     /* Capture the context */
    RtlCaptureContext(&Context);
    /* Add one argument to ESP */
    Context.Esp += sizeof(PVOID);

    /* Create an exception record */
    ExceptionRecord.ExceptionAddress = RtlpGetExceptionAddress();
    ExceptionRecord.ExceptionCode = Status;
    ExceptionRecord.ExceptionRecord = NULL;
    ExceptionRecord.NumberParameters = 0;
    ExceptionRecord.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
    /* Write the context flag */
    Context.ContextFlags = CONTEXT_FULL;
    /* Check if we're being debugged (user-mode only) */
    if (!RtlpCheckForActiveDebugger(TRUE))
    {
        /* Raise an exception immediately */
        ZwRaiseException(&ExceptionRecord, &Context, TRUE);
    }
    else
    {
        /* Dispatch the exception */
        RtlDispatchException(&ExceptionRecord, &Context);
        /* Raise exception if we got here */
        Status = ZwRaiseException(&ExceptionRecord, &Context, FALSE);
    }
    /* If we returned, raise a status */
    RtlRaiseStatus(Status);
}
```

这个函数的代码与 RtlRaiseException()其实很相似，只是新的异常纪录块创建在本函数框架内部，并且排除了对 ZwContinue()的调用。特别值得注意的是，这个函数有可能递归调用其自身，如果对 ZwRaiseException()又返回了，就又要调用 RtlRaiseStatus()。显然，这里的希望寄托在每次对 RtlRaiseStatus()的调用参数、即 ZwRaiseException()的失败原因不同，而总有一次得到了妥善的处理，就不再从 ZwRaiseException()返回了。注意这里所谓“妥善的处理”也包括在异常处理的第三步中结束线程的运行或者停机。如果 ZwRaiseException()

老是返回，那就说明连这也出了问题。

读者也许因而对异常处理的结局产生一种黯淡的印象，怎么老是失败又失败？但是须知这是在应付最坏的情况，实际的情况不会有这么糟糕。因为如前所述每个线程的运行都是在一个 SEH 域中进行，并且这个 SEH 域的过滤函数实际上是来者不拒。再说，只要没有很特殊的情况，即使基于 ExceptionList 的处理失败，后面也还有第二步、第三步措施。在正常的条件下，对用户空间异常的最后一手就是通过 ZwTerminateThread()结束线程的运行，对系统空间异常的最后一手则是停机。

如前所述，ZwRaiseException()就是系统调用 NtRaiseException()。一般在用户空间程序中使用 NtRaiseException()，而在内核中使用 ZwRaiseException()。但是在用户空间也可以用 ZwRaiseException()，这就是为像 RtlRaiseException()这样“两栖”的代码而设的，在 Ntdll.dll 所导出的函数表中，NtRaiseException()和 ZwRaiseException()实际上指向同一段代码。

这个系统调用的作用就是模拟一次异常。在用户空间调用就模拟发生于用户空间的异常，在系统空间调用就模拟发生于系统空间的异常。读者也许会问，我们前面在 KiUserExceptionDispatcher()中看到对 RtlDispatchException()的调用，带着一个新的异常纪录块扫描 ExceptionList，这不也是在模拟一次异常码？这里面的区别还是不小的。扫描 ExceptionList 只是异常处理的几个步骤之一，整个异常处理的过程可以包含三次努力。另一方面，无论是真实发生的硬异常、还是由这个系统调用模拟的软异常，都会在系统空间堆栈上形成一个异常框架(由系统调用形成的陷阱框架与此相同)。而直接调用 RtlDispatchException()则并不形成新的异常框架。

系统调用 NtRaiseException()的调用界面如下：

```
NTSYSCALLAPI NTSTATUS NTAPI
NtRaiseException(IN PEXCEPTION_RECORD ExceptionRecord,
                 IN PCONTEXT Context, IN BOOLEAN SearchFrames);
```

前两个参数分别是指向异常纪录块和上下文结构的指针。第三个参数表示是否需要搜索 ExceptionList，其实就是以前见到过的 FirstChance，如果为 FALSE 就表示处理该次异常的第一次努力已经失败。

异常纪录块中的字段 ExceptionCode 说明本次异常的性质、即种类。这是一个 32 位无符号异常代码，其意义有如中断向量，例如 STATUS_INTEGER_DIVIDE_BY_ZERO、STATUS_SINGLE_STEP 等等都是。但是，由于不受硬件限制，异常代码的取值范围比中断向量要广泛得多，就其本质而言这只是异常的发起者与处理者(过滤函数)之间的约定。所以，除微软已经定义的异常代码、即状态代码之外，应用软件的开发者也可以按需要自行定义新的代码。微软为此定下了规则：

- Bit30-31 表示严重性： 0 = 成功， 1 = 提示， 2 = 警告，3 = 出错。
- Bit29 表示定义/使用者：0 = 由微软定义， 1 = 非微软定义。
- Bit28 保留不用，必须为 0。
- Bit16-27 表示类型。
- Bit0-15 为异常编号、即产生异常的具体原因。

例如，STATUS_NETWORK_SESSION_EXPIRED 定义为 0xC000035C，那就是：严重性为“出错”，由微软定义，类型代码为 3，具体原因的代码为 0x5C。

所以，在异常纪录块中，异常代码表示异常的性质和原因。而 ExceptionFlags，则以标志位的形式给出有关的状态信息。当然，对于不同编码的异常，对这些标志位可以作不同的

解释。

至于上下文 Context，则是为 NtContinue()准备的。

不管是 NtRaiseException()还是 ZwRaiseException()，内核中实现这个系统调用的函数总是 NtRaiseException()。在 0.3.0 版 ReactOS 的代码中，这是一段汇编程序：

[NtRaiseException()]

```
_NtRaiseException@12:
    /* NOTE: We -must- be called by Zw* to have the right frame! */
    /* Push the stack frame */
    push ebp

    /* Get the current thread and restore its trap frame */
    mov ebx, [fs:KPCR_CURRENT_THREAD]
    mov edx, [ebp+KTRAP_FRAME_EDX]
    mov [ebx+KTHREAD_TRAP_FRAME], edx

    /* Set up stack frame */
    mov ebp, esp

    /* Get the Trap Frame in EBX */
    mov ebx, [ebp+0]

    /* Get the exception list and restore */
    mov eax, [ebx+KTRAP_FRAME_EXCEPTION_LIST]
    mov [fs:KPCR_EXCEPTION_LIST], eax

    /* Get the parameters */
    mov edx, [ebp+16] /* Search frames */
    mov ecx, [ebp+12] /* Context */
    mov eax, [ebp+8]    /* Exception Record */

    /* Raise the exception */
    push edx
    push ebx
    push 0
    push ecx
    push eax
    call _KiRaiseException@20

    /* Restore trap frame in EBP */
    pop ebp
    mov esp, ebp
```

```
    /* Check the result */
    or eax, eax
    jz _KiServiceExit2

    /* Restore debug registers too */
    jmp _KiServiceExit
```

在进入 NtRaiseException()之前，在系统空间堆栈上已经因为系统调用而形成了一个陷阱框架，“框架指针”EBP 就指向这个框架。这里先通过 KPCR 结构获取指向当前线程 KTHREAD 数据结构的指针，然后使这 KTHREAD 结构中的指针 TrapFrame 指向当前的异常框架。这样，以后通过 ExGetPreviousMode()就能获取正确的“先前模式”。

但是这里有个问题值得一说。常数 KPCR_CURRENT_THREAD 定义为 0x124，而 KPCR 结构的大小只是 0x54，所以指向当前线程 KTHREAD 结构的指针显然不在 KPCR 中。原来，除 KPCR 以外，每个处理器还有个 KPRCB 数据结构(Processor Region Control Block)。KPCR 结构中有个指针 Prcb，指向与其配对的 KPRCB 数据结构。可是，KPRCB 与 KPCR 两个数据结构起点之间的距离却是固定的，那就是 0x120。而 KPRCB 结构中的第二个长字，就是指针 CurrentThread。所以，KPCR 的位置一经确定，相应 KPRCB 的位置也就定了。既然如此，为什么不干脆把两个数据结构合二为一呢？这就不得而知了，也许是历史遗留的问题。

实际的处理是由 KiRaiseException()完成的。处理完以后，如果返回的话，就根据返回值是否为 STATUS_SUCCESS、即 0 而分别经由_KiServiceExit2 或_KiServiceExit 从异常返回。

[NtRaiseException() > KiRaiseException()]

```
NTSTATUS NTAPI
KiRaiseException(PEXCEPTION_RECORD ExceptionRecord, PCONTEXT Context,
                 PKEXCEPTION_FRAME ExceptionFrame,
                 PKTRAP_FRAME TrapFrame, BOOLEAN SearchFrames)
{
    KPROCESSOR_MODE PreviousMode = KeGetPreviousMode();
    CONTEXT LocalContext;
    EXCEPTION_RECORD LocalExceptionRecord;
    ULONG ParameterCount, Size;
    NTSTATUS Status = STATUS_SUCCESS;
    DPRINT1("KiRaiseException\n");

    /* Set up SEH */
    _SEH_TRY
    {
        /* Check the previous mode */
        if (PreviousMode != KernelMode)
        {
            /* Validate the maximum parameters */
            if ((ParameterCount = ExceptionRecord->NumberParameters) >
                                  EXCEPTION_MAXIMUM_PARAMETERS)
```

```
            {
                /* Too large */
                Status = STATUS_INVALID_PARAMETER;
                _SEH_LEAVE;
            }

            /* Probe the entire parameters now*/
            Size = (sizeof(EXCEPTION_RECORD) -
        ((EXCEPTION_MAXIMUM_PARAMETERS - ParameterCount) * sizeof(ULONG)));
            ProbeForRead(ExceptionRecord, Size, sizeof(ULONG));

            /* Now make copies in the stack */
            RtlMoveMemory(&LocalContext, Context, sizeof(CONTEXT));
            RtlMoveMemory(&LocalExceptionRecord, ExceptionRecord, Size);
            Context = &LocalContext;
            ExceptionRecord = &LocalExceptionRecord;

            /* Update the parameter count */
            ExceptionRecord->NumberParameters = ParameterCount;
        }
    }
    _SEH_HANDLE
    {
        Status = _SEH_GetExceptionCode();
    }
    _SEH_END;

    if (NT_SUCCESS(Status))
    {
        /* Convert the context record */
        KeContextToTrapFrame(Context,
                             ExceptionFrame,
                             TrapFrame,
                             Context->ContextFlags,
                             PreviousMode);

        /* Dispatch the exception */
        KiDispatchException(ExceptionRecord,
                            ExceptionFrame,
                            TrapFrame,
                            PreviousMode,
                            SearchFrames);
    }
```

```
    /* Return the status */
    return Status;
}
```

参数 ExceptionFrame 是个指向 KEXCEPTION_FRAME 结构的指针，仅用于 PowerPC 处理器，所以从上面传下来的实参是 0；另一个参数 TrapFrame 则是指向堆栈上陷阱框架的指针。其余参数不言自明。

如果本次系统调用来自用户空间，那么作为参数的异常纪录块和上下文数据结构都在用户空间，所以需要先通过 RtlMoveMemory()把它们复制到系统空间的临时缓冲区里来。然而这恰恰又是可能引起异常的，所以又要有个 SEH 域将其保护起来。

异常纪录块和上下文数据结构都存在于系统空间以后，就可以调用 KiDispatchException()了，参数 SearchFrames 变成了 KiDispatchException()的参数 FirstChance。

KiDispatchException()的代码当然不用再看了。总之，就是前述的三步措施、三次努力，主要就是搜索 ExceptionList。但是搜索的是哪一个 ExceptionList，就取决于“先前模式”，即调用 ZwRaiseException()之时所处的空间：

- 如果“先前模式”是系统模式，那主要就是搜索系统空间的 ExceptionList，在正常的情况下会作长程跳转而不再返回，因此自然也就不会从本次系统调用返回了。如果返回，那就是返回到调用 ZwRaiseException()的地方。
- 如果“先前模式”是用户模式，那主要就是修改堆栈上异常框架中的返回地址，然后返回，使得一旦返回到用户空间就进入 KiUserExceptionDispatcher()，并在那里扫描用户空间的 ExceptionList。在正常的情况下也会因长程跳转而不再返回到调用 ZwRaiseException()的地方。或者，倘若不作长程跳转而要回到调用 ZwRaiseException()的地方，则可以通过系统调用 ZwContinue()实现。

从 KiRaiseException()返回后，NtRaiseException()根据返回值是否为 0 而分别经由 _KiServiceExit2 或_KiServiceExit：

```
_KiServiceExit:
    /* Disable interrupts */
    cli
    /* Check for, and deliver, User-Mode APCs if needed */
    CHECK_FOR_APC_DELIVER 1
    . . . . . .
    /* Exit and cleanup */
    TRAP_EPILOG FromSystemCall, DoRestorePreviousMode, \
                        DoNotRestoreSegments, DoNotRestoreVolatiles, DoRestoreEverything

_KiServiceExit2:
    /* Disable interrupts */
    cli
    /* Check for, and deliver, User-Mode APCs if needed */
    CHECK_FOR_APC_DELIVER 0
    /* Exit and cleanup */
    TRAP_EPILOG NotFromSystemCall, DoRestorePreviousMode, \
                        DoRestoreSegments, DoRestoreVolatiles, DoNotRestoreEverything
```

由于是在.S 汇编代码文件中，定义和引用宏操作的格式有所不同。这里的 FromSystemCall 和 NotFromSystemCall 都是常数，前者定义为 1，后者定义为 0，余类推。

而 TRAP_EPILOG，则显然是与 TRAP_PROLOG 相对应，类似于中断返回，具体的代码就留给读者自己去看了。