漫谈兼容内核之二十:
Windows 线程的系统空间堆栈

毛德操

在计算机技术的发展史上，堆栈的发明有着划时代的意义。从那以后，实际上已经不再存在可以脱离堆栈而运行的程序。我们从堆栈的用途和内容可以看出其重要性：

- 记录子程序调用的轨迹，使嵌套的(多层的)子程序调用成为可能。
- 通过堆栈传递子程序调用参数，使程序设计得以简化。要是不能通过堆栈传递参数，实际上就不会有现代的程序设计。
- 在堆栈上为子程序内部的“局部变量”分配空间，这既使“模块化程序设计”和“结构化程序设计”成为可能，也为线程(进程)概念的产生提供了条件。
- 程序执行的轨迹和局部变量结合在一起成为“脉络”、即“上下文”，由此产生了线程(进程)的概念，使多线程(多进程)系统及其调度/切换成为可能。

可见堆栈与“程序的执行”有着不可分割的关系；而程序的执行又恰恰是线程的最重要、最本质的特征。在线程的概念和应用出现之前，人们常说“进程是程序的执行”；后来有了线程的概念，人们在相当长一段时期中都不很重视线程和进程在概念上的区分，还是常说“进程是程序的执行”；现在则应该说“线程是程序的执行”了。总之，没有堆栈就无所谓线程，线程是离不开堆栈的，而且线程最根本的“私有财产”就是它的堆栈(以及以后要讲到的“堆栈本地存储(TLS)”)，别的都是属于进程、而不是属于个别线程的“集体财产”。

像 Unix/Linux 一样，除“内核线程”外，每个 Windows 线程都有两个堆栈，一个是用户空间堆栈，一个是系统空间堆栈。由于“内核线程”只在内核中执行而没有用户空间，所以就没有用户空间堆栈。

在系统完成初始化以后，当 CPU 运行于用户空间时，它就一定是在执行某个线程、即“当前线程”的程序，因而就使用这个线程的用户空间堆栈。当 CPU 运行于系统空间时，也一定有个“当前线程”，具体取决于当时处于运行状态的是哪一个线程，因而就使用这个线程的系统空间堆栈。或者也可以反过来说，当前正在使用哪一个线程的系统空间堆栈，这个线程就是当前线程。不过，CPU 运行于系统空间时却不一定是在执行当前线程的程序、或者说不一定是在为当前线程而执行程序(内核中的程序是由所有线程公有的)，而有可能是在为它人做嫁衣裳。例如，中断处理就未必是为当前线程而执行的。所以 Windows 的内核又分成上、下两大层，其中上面的一层称为“执行层(Executive)”，执行层中的程序肯定是为当前线程而执行的，执行层下面就不一定了(不过也并非肯定不是)。其实 Linux 内核也是这样，只不过没有“执行层”这么个说法。

这样，不管 CPU 是在用户空间还是系统空间，不管是在做些什么事情、为谁做，它一定是在使用当前线程的堆栈，具体使用哪一个则取决于 CPU 当时运行于哪一个空间。

线程的用户空间堆栈比较容易为人们所理解。因为一来 CPU 在用户空间的活动基本上是由应用软件的程序员们编排的，人们比较熟悉；二来 CPU 在用户空间是专一的，总是在为当前线程而劳碌，不会“开小差”。

而系统空间堆栈就不同了，人们对此往往会有似懂非懂的感觉，所以本文对线程的系统空间堆栈作一些说明，具体包括：

- 系统空间堆栈的分配和建立。
- 系统空间堆栈在各种情况下的消长变化。

- 系统空间堆栈上的陷阱框架和调用框架。

不难想像，线程的系统空间堆栈是在创建线程的时侯分配和建立的。在系统调用NtCreateThread()内部，线程初始化的一部分就是为其系统空间堆栈分配存储区间。

[NtCreateThread() > PsInitializeThread() > KeInitializeThread()]

```
VOID
STDCALL
KeInitializeThread(PKPROCESS Process, PKTHREAD Thread, BOOLEAN First)
{
    PVOID KernelStack;
    . . . . . .

    . . . . . .
    /* If this is isn't the first thread, allocate the Kernel Stack */
    if (!First) {
        PFN_TYPE Page[MM_STACK_SIZE / PAGE_SIZE];
        KernelStack = NULL;
        MmLockAddressSpace(MmGetKernelAddressSpace());
        Status = MmCreateMemoryArea(NULL, MmGetKernelAddressSpace(),
                                    MEMORY_AREA_KERNEL_STACK, &KernelStack,
                                    MM_STACK_SIZE, 0, &StackArea, FALSE, FALSE,
                                    BoundaryAddressMultiple);
        MmUnlockAddressSpace(MmGetKernelAddressSpace());
        . . . . . .
        /* Mark the Stack */
        for (i = 0; i < (MM_STACK_SIZE / PAGE_SIZE); i++) {
            Status = MmRequestPageMemoryConsumer(MC_NPPOOL, TRUE, &Page[i]);
            . . . . . .
        }
        /* Create a Virtual Mapping for it */
        Status = MmCreateVirtualMapping(NULL, KernelStack, PAGE_READWRITE,
                                                Page, MM_STACK_SIZE / PAGE_SIZE);
        . . . . . .
        /* Set the Kernel Stack */
        Thread->InitialStack = (PCHAR)KernelStack + MM_STACK_SIZE;
        Thread->StackBase    = (PCHAR)KernelStack + MM_STACK_SIZE;
        Thread->StackLimit   = (ULONG_PTR)KernelStack;
        Thread->KernelStack  = (PCHAR)KernelStack + MM_STACK_SIZE;
    } else {
        /* Use the Initial Stack */
        Thread->InitialStack = (PCHAR)init_stack_top;
        Thread->StackBase = (PCHAR)init_stack_top;
```

```
        Thread->StackLimit = (ULONG_PTR)init_stack;
        Thread->KernelStack = (PCHAR)init_stack_top;
    }
    /*
     * Establish the pde's for the new stack and the thread structure within the
     * address space of the new process. They are accessed while taskswitching or
     * while handling page faults. At this point it isn't possible to call the
     * page fault handler for the missing pde's.
     */
    MmUpdatePageDir((PEPROCESS)Process, (PVOID)Thread->StackLimit,
                                                    MM_STACK_SIZE);
    MmUpdatePageDir((PEPROCESS)Process, (PVOID)Thread, sizeof(ETHREAD));
    . . . . . .
    Thread->KernelStackResident = 1;
    . . . . . .
    Thread->EnableStackSwap = 0;
    . . . . . .
}
```

参数 First 表明目标线程是否整个系统中(初始化以后)的第一个线程，也就是第一个进程中的第一个线程。系统中的第一个线程用 init_stack 作为它的系统空间堆栈，这是一块固定的缓冲区，所以就无需分配了。当然，我们此刻关心的绝不是系统中的第一个线程。

只要不是系统中的第一个线程，就通过 MmCreateMemoryArea()在系统空间分配一块虚拟地址区间。由于实际参数 KernelStack 事先设置成 NULL，所以对具体的位置并无要求，而由内存管理自由分配。区间的大小为 MM_STACK_SIZE、实际上是 3 个 4KB 的页面。再通过 MmRequestPageMemoryConsumer()分配相应数量的物理页面，然后通过 MmCreateVirtualMapping()建立虚存页面和物理页面之间的映射。与 Linux 内核不同，Windows 内核中并非所有页面都不受换出/换入，但是各线程的系统空间堆栈所占页面一般是不受换出/换入的。

为新建线程分配了系统空间堆栈之后，还要在这堆栈上虚构出一个“陷阱框架(Trap Frame)”，使堆栈的内容看起来就像这个线程是从用户空间通过系统调用进入内核、而尚未返回一样，为其受调度运行后在内核中的活动以及“返回”用户空间埋下伏笔。如果是新建进程中的第一个线程则还要为 APC 函数 LdrInitializeThunk()的执行作好安排。有关情况在以前的几篇漫谈中已有述及，后面也还要提到。

关于“框架(Frame)”，此刻只要把它理解为当 CPU 在某个特定函数中执行时堆栈上与其相关的内容。一般而言，框架是因为函数调用而形成的，每调用一个函数就会引起堆栈的扩张，堆栈上就多出一个框架；返回时则堆栈收缩，堆栈上消失一个框架。除一般的函数调用以外，系统调用、中断、以及异常也会在系统空间堆栈上形成一个框架，不同的是此时所形成的框架有可能是跨空间的、因而是跨堆栈的。而且，如果跨空间，就总是原来在用户空间，而新的框架则形成在系统空间堆栈上。这是因为系统调用、中断、和异常都使 CPU 进入系统空间，并使用系统空间堆栈。“陷阱框架”这个词最初可能是因系统调用而来，因为系统调用一般是通过自陷指令实现的；但是后来因中断和异常所引起的框架也都称为陷阱框架了。

系统空间堆栈属于线程所有、归线程使用，因而其位置自然就要记录在线程的

KTHREAD 数据结构中。注意代码中变量 KernelStack 所指向的是区间的(物理的)起点、即这个区间中地址最低的字节，而堆栈是从上向下伸展的，因此所记录的(逻辑的)堆栈起点是(PCHAR)KernelStack+MM_STACK_SIZE，这里 MM_STACK_SIZE 是以字节为单位的区间长度，而 StackLimit、即其(逻辑的)终点倒是 KernelStack。

这里把KTHREAD数据结构中的InitialStack、StackBase、和KernelStack三个字段都设置成指向系统空间堆栈区间的终点、即上部边界。显然这只是初值相同，三个字段应该有不同的意义和用途。其实，KTHREAD结构中与堆栈有关的字段还不止上面所看到的四个：

```
typedef struct _KTHREAD
{
    . . . . . .
    PVOID                  InitialStack;          /* 18 */
    ULONG_PTR              StackLimit;            /* 1C */

    /* Pointer to the thread's environment block in user memory */
    PTEB                   Teb;                   /* 20 */

    /* Pointer to the thread's TLS array */
    PVOID                  TlsArray;              /* 24 */
    PVOID                  KernelStack;           /* 28 */
    . . . . . .
    UCHAR                  KernelStackResident;   /* 11E */
    UCHAR                  NextProcessor;         /* 11F */
    PVOID                  CallbackStack;          /* 120 */
    . . . . . .
    UCHAR                  EnableStackSwap;       /* 134 */
    UCHAR                  LargeStack;            /* 135 */
    . . . . . .
    PVOID                  StackBase;             /* 15C */
    . . . . . .
} KTHREAD;
```

设立EnableStackSwap字段的用意显然是想有控制地允许换出/换入系统空间堆栈所占的页面。如果允许换出/换入的话，KernelStackResident想必是用来表明这些页面当前是否在内存中。可是为什么要允许换出/换入系统空间堆栈呢？我们在前面看到，每个线程的系统空间堆栈的大小只是区区3个4KB的页面，似乎也并不是很大一笔资源。可是另一个字段LargeStack给了我们一些线索，看来设计者的意图是也可以采用“大堆栈”，而在采用大堆栈的条件下换出/换入其存储页面就有意义了。不过，就目前ReactOS的代码而言，这几个字段尚无实际的意义。

另一个字段CallbackStack跟从系统空间“回调(callback)”用户空间的函数有关，在回调用户空间函数时会暂时使用另一个堆栈，所以才需要这个字段；另一方面这也解释了为什么要有InitialStack这个字段。不过我们此刻对此并不关心。

对于线程的运行有实质性意义的是 KernelStack 这个字段。这个字段在刚为系统空间堆栈分配存储区间时指向区间的终点、即上部边界，但是随着进一步的初始化、以及为新建

线程虚构陷阱框架的过程、就有了变化。事实上，完成了对新建线程基本的初始化以后，NtCreateThread()的代码中还有个函数调用 KiArchInitThreadWithContext()，这实际上是个宏操作，对于 x86处理器定义为 Ke386InitThreadWithContext()，这就是对系统空间堆栈的进一步初始化。读者在以前讲述线程创建的漫谈中见到过这个函数，但是现在需要从堆栈使用的角度加以深入考察：

[NtCreateThread() > Ke386InitThreadWithContext()]

```
NTSTATUS Ke386InitThreadWithContext(PKTHREAD Thread, PCONTEXT Context)
{
  PULONG KernelStack;
  ULONG InitSize;
  PKTRAP_FRAME TrapFrame;
  PFX_SAVE_AREA FxSaveArea;

  /* Setup a stack frame for exit from the task switching routine */
  InitSize = 6 * sizeof(DWORD) + sizeof(DWORD) + 6 * sizeof(DWORD) +
                                + sizeof(KTRAP_FRAME) + sizeof (FX_SAVE_AREA);
  KernelStack = (PULONG)((char*)Thread->KernelStack - InitSize);

  /* Set up the initial frame for the return from the dispatcher. */
  KernelStack[0] = (ULONG)Thread->InitialStack-sizeof(FX_SAVE_AREA); /* TSS->Esp0 */
  KernelStack[1] = 0;          /* EDI */
  KernelStack[2] = 0;          /* ESI */
  KernelStack[3] = 0;          /* EBX */
  KernelStack[4] = 0;          /* EBP */
  KernelStack[5] = (ULONG)&PsBeginThreadWithContextInternal;     /* EIP */

  /* Save the context flags. */
  KernelStack[6] = Context->ContextFlags;

  /* Set up the initial values of the debugging registers. */
  KernelStack[7] = Context->Dr0;
  . . . . . .
  KernelStack[12] = Context->Dr7;

  /* Set up a trap frame from the context. */
  TrapFrame = (PKTRAP_FRAME)(&KernelStack[13]);
  TrapFrame->DebugEbp = (PVOID)Context->Ebp;
  . . . . . .
  TrapFrame->Eax = Context->Eax;
  TrapFrame->PreviousMode = UserMode;
  TrapFrame->ExceptionList = (PVOID)0xFFFFFFFF;
  TrapFrame->Fs = TEB_SELECTOR;
```

```
  TrapFrame->Edi = Context->Edi;
  . . . . . .
  TrapFrame->Eip = Context->Eip;
  TrapFrame->Eflags = Context->EFlags | X86_EFLAGS_IF;
  TrapFrame->Eflags &= ~(X86_EFLAGS_VM | X86_EFLAGS_NT | X86_EFLAGS_IOPL);
  TrapFrame->Esp = Context->Esp;
  TrapFrame->Ss = (USHORT)Context->SegSs;
  /* FIXME: Should check for a v86 mode context here. */

  /* Set up the initial floating point state. */
  /* FIXME: Do we have to zero the FxSaveArea or is it already? */
  FxSaveArea = (PFX_SAVE_AREA)
                        ((ULONG_PTR)KernelStack + InitSize - sizeof(FX_SAVE_AREA));
  if (KiContextToFxSaveArea(FxSaveArea, Context))
  {
      Thread->NpxState = NPX_STATE_VALID;
  }
  else
  {
      Thread->NpxState = NPX_STATE_INVALID;
  }

  /* Save back the new value of the kernel stack. */
  Thread->KernelStack = (PVOID)KernelStack;

  return(STATUS_SUCCESS);
}
```

调用参数 Context 指向一个 CONTEXT 数据结构，这是作为系统调用 NtCreateThread() 的参数传下来的。这个数据结构给定了当新建线程开始在用户空间运行时的初始上下文，例如 Context->Eip 就是新建线程在用户空间的程序入口，Context->Esp 就是新建线程开始在用户空间运行时的堆栈指针，等等。

我们看这里的指针 KernelStack，注意最后 KTHREAD 结构中的 KernelStack 字段就是被设置成了这个指针的值，从而与前述 InitialStack 和 StackBase 两个字段拉开了距离。那么指针 KernelStack 的值是什么呢？从代码中可以看出，是从原来的区间终点开始下调，调整的距离是 InitSize，这段距离逻辑上分成三个部分、用于三个目的：

一、首先是一个 FX_SAVE_AREA 数据结构。在有浮点处理器的系统中，这是用来保存浮点处理器状态的，后面的指针 FxSaveArea 就是指向这个数据结构的起点。这个数据结构其实不属于系统空间堆栈，只是借用它一块宝地而已。在这下面才是真正意义上的堆栈，而这也就是系统空间的原点。当系统空间堆栈为空时，堆栈指针就指向这里。特别地，当 CPU 运行于用户空间时，当前线程的系统空间堆栈总是空的。

二、然后是一个 KTRAP_FRAME 数据结构，这就是一个陷阱框架。事实上，只要 CPU 运行于系统空间，当前线程系统空间堆栈的(区间)顶部一定是一个陷阱框架。但是系统调用的陷阱框架和中断的陷阱框架不一样，而这里要构筑的是系统调用的陷阱框架，要

为新建线程制造出一个处于系统调用过程中的假象。这样，当 CPU 从实现具体系统调用的函数返回、到达_KiServiceExit 的时侯，堆栈指针应该恰好指向这个数据结构的起点、即陷阱框架的下部边界。KTRAP_FRAME 数据结构的定义如下：

```
typedef struct _KTRAP_FRAME
{
    PVOID DebugEbp;
    PVOID DebugEip;
    PVOID DebugArgMark;
    PVOID DebugPointer;
    PVOID TempCs;
    PVOID TempEip;
    ULONG Dr0;
    ULONG Dr1;
    ULONG Dr2;
    ULONG Dr3;
    ULONG Dr6;
    ULONG Dr7;
    USHORT Gs;
    USHORT Reserved1;
    USHORT Es;
    USHORT Reserved2;
    USHORT Ds;
    USHORT Reserved3;
    ULONG Edx;
    ULONG Ecx;
    ULONG Eax;
    ULONG PreviousMode;
    PVOID ExceptionList;
    USHORT Fs;
    USHORT Reserved4;
    ULONG Edi;
    ULONG Esi;
    ULONG Ebx;
    ULONG Ebp;
    ULONG ErrorCode;
    ULONG Eip;
    ULONG Cs;
    ULONG Eflags;
    ULONG Esp;
    USHORT Ss;
    USHORT Reserved5;
    USHORT V86_Es;
    USHORT Reserved6;
```

```
    USHORT V86_Ds;
    USHORT Reserved7;
    USHORT V86_Fs;
    USHORT Reserved8;
    USHORT V86_Gs;
    USHORT Reserved9;
} KTRAP_FRAME, *PKTRAP_FRAME;
```

这里要注意：

- 这数据结构中各个字段的次序与实际压入堆栈的次序正好相反，因为堆栈是从上向下伸展的。所以，例如 Ds 就比 Es 先压入堆栈。
- 结构中有些字段的类型是 16 位的 USHORT，但是 CPU 的堆栈操作都是 32 位的，所以这些 16 位字段实际上都是合二为一的。例如字段 Gs 和 Reserved1 其实是合在一起的，余可类推。之所以如此，是因为段寄存器都是 16 位的。
- 在用户空间通过 int 指令进行系统调用时，CPU 自动压入堆栈的第一项数据是用户空间堆栈段寄存器 SS 的值，就是这里字段 Ss 和 Reserved5 的组合。按理说这已经是堆栈区间的顶部，再往上没有别的数据了。可是这里的数据结构定义中在这上面还有 V86_Es 等 4 个栈项，这是因为如果在进入系统空间之前 CPU 运行于 V86 模式的话就还有 4 项额外的数据，即在 V86 模式下段寄存器 ES、DS、FS、GS 的值。所以，数据结构 KTRAP_FRAME 实质上是一个 Union。但是这并不妨碍通过 KTRAP_FRAME 指针访问普通陷阱框架中的数据，只是在一般情况下框架顶部到 Ss 和 Reserved5 为止，也就是说一般情况下的系统调用框架比这里的短一点。

当新建线程受调度运行而“返回”用户空间时，CPU 逐步恢复保存在陷阱框架中的上下文，通过 iret 指令返回用户空间的时侯，从系统空间堆栈中取出的最后一项数据是(用户空间)段寄存器 SS 的值，所以堆栈指针就停留在了指向安排用于 V86_Es 的位置。对于不运行于 V86 模式的线程，其系统空间堆栈指针不会再往上跑了，这不意味着系统空间堆栈不为空了吗？应该说，物理意义上确实是这样，但是逻辑意义上仍然是空的，而且这只发生于新建线程的第一次返回用户空间。这是因为当 CPU 下一次因系统调用、中断、异常而再次进入系统空间、切换到当前线程的系统空间堆栈时，是从其原点开始的。此时所形成的陷阱框架大小则与当时是否运行于 V86 模式有关。这样，以后的框架消长就总是平衡的，CPU 每次回到用户空间时其系统空间堆栈指针就总是真正地回到原点。

三、在陷阱框架下面还有 13 个栈项，即 KernelStack[0]至 KernelStack[12]。其中 KernelStack[7]至 KernelStack[12]用于调试寄存器 Dr0、Dr1 等等。而 KernelStack[5]是 Eip 的映像，本来应该是函数调用的返回地址，这里则指向一段汇编代码。这就相当于在陷阱框架下面又开了一个函数调用框架。事实上，这段汇编代码最后是通过 jmp 指令跳转到 _KiServiceExit 的，所以新建线程就相当于是身处一个在_KiServiceExit 前面调用的子程序中。KernelStack[1]至 KernelStack[4]则用于为新建线程准备下几个寄存器的初值(都是 0)。

下面就是上述的汇编代码_PsBeginThreadWithContextInternal，读者可以结合、对照上面 Ke386InitThreadWithContext()的代码阅读：

```
_PsBeginThreadWithContextInternal:
   /* This isn't really a function, we are called as the return address of a context switch */
```

```
  /* Do the necessary prolog before the context switch */
  call    _PiBeforeBeginThread

  /* Load the context flags. */
  popl    %ebx

  /* Load the debugging registers */
  testl   $(CONTEXT_DEBUG_REGISTERS & ~CONTEXT_i386), %ebx
  jz    .L1
  popl    %eax            /* Dr0 */
  movl    %eax, %dr0
  popl    %eax            /* Dr1 */
  movl    %eax, %dr1
  popl    %eax            /* Dr2 */
  movl    %eax, %dr2
  popl    %eax            /* Dr3 */
  movl    %eax, %dr3
  popl    %eax            /* Dr6 */
  movl    %eax, %dr6
  popl    %eax            /* Dr7 */
  movl    %eax, %dr7
  jmp   .L3
.L1:
  addl    $24, %esp
.L3:
  /* Load the rest of the thread's user mode context. */
  movl    $0, %eax
  jmp   _KiServiceExit
```

开始时对于 PiBeforeBeginThread()的调用是为了改变代码的运行级别，我们在此不必关心。

这里 call 指令以后对堆栈的操作消去了堆栈上对应于前面代码中从 KernelStack[6]至 KernelStack[12]的 7 个表项，其中各调试寄存器的内容根据当前实际上是否在调试而或者装入相应的寄存器，或者通过调整堆栈指针予以跳过、丢弃。这样，当最后跳转到 _KiServiceExit 时，堆栈指针恰好指向陷阱框架的(区间)起点。所以，对于 KernelStack[5]的设置实际上是为线程的调度/切换准备的，目的是为新建线程提供一个虚构的程序执行“断点”，仿佛原先就是在这里被剥夺了运行，下次受调度运行时就从这一点上恢复。

那么 KernelStack[0]又是干什么用的呢？注释中说了这就是 TSS->Esp0，这跟线程切换的机制与过程有关，这里先简单说一下。当新建线程受调度运行时，会将这个数值写入“任务状态段”TSS 中的 ESP0 字段，而每当 CPU 从用户空间进入系统空间、需要切换到系统空间堆栈时，就总是把 TSS 中的这个数值装入 ESP，所以这就是系统空间堆栈的原点。从代码中可以看到，这就是 Thread->InitialStack-sizeof(FX_SAVE_AREA)。

最后又要回到对 Thread->KernelStack 的设置。KTHREAD 结构中这个字段的用途是在线程切换的时侯保存堆栈指针。就是说，其它寄存器的内容都保存在堆栈上，而堆栈指针

则保留在 KTHREAD 结构中。当一个线程暂时放弃运行、或被剥夺运行时(一定发生于系统空间)，就把它当时的堆栈指针保存在这儿；到下一次又被调度运行时则从这儿恢复其堆栈指针。

当新建线程受调度运行并跳转到_KiServiceExit 处时，其系统空间堆栈上已经只剩下陷阱框架了，而随后的“恢复现场”以及最后 iret 指令的执行则使陷阱框架消失，逻辑意义上系统空间堆栈的大小收缩到 0、堆栈指针回到原点；同时 CPU 返回到用户空间、切换到用户空间堆栈。以后每次从系统调用、中断、或异常返回用户空间时，则物理意义上也都是如此。显然，这一点是很重要的，要不然系统空间堆栈就早晚会被耗尽。

但是，如果需要执行 APC 函数的话，那就还有个不小的插曲和变化。我们先看_KiServiceExit 处对 APC 函数的处理：

```
_KiServiceExit:
    /* Get the Current Thread */
    cli
    movl %fs:KPCR_CURRENT_THREAD, %esi

    /* Deliver APCs only if we were called from user mode */
    testb $1, KTRAP_FRAME_CS(%esp)
    je KiRosTrapReturn

    /* And only if any are actually pending */
    cmpb $0, KTHREAD_PENDING_USER_APC(%esi)
    je KiRosTrapReturn

    /* Save pointer to Trap Frame */
    movl %esp, %ebx
    . . . . . .
    /* Deliver APCs */
    sti
    pushl   %ebx
    pushl   $0
    pushl   $UserMode
    call    _KiDeliverApc@12
    cli
```

如果是要返回到用户空间(本次系统调用是从系统空间启动)，并且有 APC 请求存在，就要调用 KiDeliverApc()，并把此时堆栈指针的内容作为参数之一传下去，因为这也就是指向陷阱框架即 KTRAP_FRAME 结构的指针。这个函数干些什么呢？就我们此刻所关心的角度而言，它干了两件事：

(1)、根据执行 APC 函数的安排和要求修改陷阱框架，使得 CPU 返回到用户空间时不是返回到原来启动系统调用的地方，而是“返回”到执行 APC 函数的地方。不过陷阱框架的大小和结构并不改变，所以回到用户空间时其系统空间堆栈同样为空。

(2)、把原来的陷阱框架保存在用户空间堆栈上，留待执行完 APC 函数以后加以恢复(并在恢复后再次返回用户空间)，因为要不然就回不到当初启动系统调用的地方去了。为此，需要修改当前线程的用户空间堆栈，在上面增添一个 CONTEXT 数据结构以及为执行 APC 函数所需的函数框架。

我们不妨想想，可以把原来的陷阱框架保存在哪里呢？无非是两种现实的可能。一种是保存在当前线程的系统空间堆栈上，即保留原来的陷阱框架不动，下面再嵌套一个新的陷阱框架。另一种是保存在用户空间堆栈上，而且本来就需要在原来的框架下面嵌套一个 APC 函数框架。比较下来，还是以保存在用户空间堆栈上为好。不过并不是原封不动地保存，而是另外定义了一个略有不同的 CONTEXT 数据结构：

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

之所以略有不同，是因为跟浮点运算有关的信息也需要保存，但是却不在陷阱框架中，这前面已经讲过了。从陷阱框架到 CONTEXT 结构的转换在 KiInitializeUserApc()里面完成，反过来则由 KeContextToTrapFrame()和 KiContextToFxSaveArea()实现转换。

从 KiDeliverApc()返回到_KiServiceExit 下面以后，CPU 继续其返回用户空间的行程。最后，当执行 iret 指令的时侯，陷阱框架就从系统空间堆栈上消失了，系统空间的堆栈指针又回到了原点。所不同的只是“返回”到了用户空间执行 APC 函数的地方，而用户空间

堆栈上则已经为此准备好了执行 APC 函数的框架。

但是，CPU 最终还得要回到用户空间原先的地方，我们知道那是通过系统调用 NtContinue()实现的，而在启动 NtContinue()时则又以堆栈上的 CONTEXT 结构作为参数带回内核，使内核得以恢复原先的陷阱框架。注意 NtContinue()本身的执行也会产生一个陷阱框架，但是这个框架被根据 CONTEX 数据结构恢复过来的框架所覆盖。因为这个框架本来就没有什么意义，NtContinue()是不返回其调用地的。

现在，假定新建线程已经在用户空间正常运行(不在执行 APC 函数)，我们又要从用户空间开始，考察系统调用时其系统空间堆栈的消长，特别是陷阱框架和其它框架的形成和消失。

当 CPU 在用户空间因执行 int 指令而切换到系统空间时，会从“任务状态段”TSS 获取当前线程系统空间堆栈的原点，自动把下列内容依次压入这个堆栈：

- 用户空间堆栈段寄存器 SS 的内容。
- 用户空间堆栈指针 ESP 的当前内容。此时 ESP 指向用户空间堆栈上最后压入的调用参数。由于将调用参数压入堆栈的次序与它们出现在代码中的次序相反，ESP 所指向的是第一个调用参数在用户空间堆栈上的位置。根据这个指针，内核是可以找到用户空间堆栈上的各个调用参数的，但是为方便起见 ReactOS 让寄存器 EDX 也指向这个位置，这在“ReactOS 怎样实现系统调用”一文中已经讲过。当然，这意味着在执行系统调用之前要先把 EDX 原来的内容先保存在用户空间堆栈上。
- 然后是“标志寄存器”EFLAGS 的当前内容。这个寄存器记录着许多有关 CPU 运行模式和状态的信息，值得特别一提的是其中的中断控制位决定着是否允许中断。实际上当 CPU 运行于用户空间时总是允许中断的，也没有关闭中断的手段，而标志寄存器内容的(自动)保存和恢复保证了这一点。
- 接着是用户空间代码段寄存器 CS 的内容。
- 最后是指令计数器的当前内容，这实际上就是用户空间的返回地址。注意这返回地址是压在系统空间堆栈上，而不是在用户空间堆栈上。

所以，当 CPU 从用户空间进入系统空间的时侯，当前线程的系统堆栈上已经有了 5 项数据，都是由 CPU 自动压入的。剩下来的就是软件的事了。换言之，本次系统调用的陷阱框架已经开始形成，但是尚未完成。系统调用在内核中的入口是_KiSystemService，我们重点看堆栈的消长以及陷阱框架的形成：

_KiSystemService:

```
/*
 * Construct a trap frame on the stack.
 * The following are already on the stack.
 */
// SS                                                    + 0x0
// ESP                                                   + 0x4
// EFLAGS                                                + 0x8
// CS                                                    + 0xC
// EIP                                                   + 0x10
pushl   $0                                              // + 0x14
```

```
pushl   %ebp                                                    // + 0x18
pushl   %ebx                                                    // + 0x1C
pushl   %esi                                                    // + 0x20
pushl   %edi                                                    // + 0x24
pushl   %fs                                                     // + 0x28

/* Load PCR Selector into fs */
movw    $PCR_SELECTOR, %bx
movw    %bx, %fs

/* Save the previous exception list */
pushl   %fs:KPCR_EXCEPTION_LIST                                 // + 0x2C

/* Set the exception handler chain terminator */
movl    $0xffffffff, %fs:KPCR_EXCEPTION_LIST

/* Get a pointer to the current thread */
movl    %fs:KPCR_CURRENT_THREAD, %esi

/* Save the old previous mode */
pushl   %ss:KTHREAD_PREVIOUS_MODE(%esi)                         // + 0x30

/* Set the new previous mode based on the saved CS selector */
movl    0x24(%esp), %ebx
andl    $1, %ebx
movb    %bl, %ss:KTHREAD_PREVIOUS_MODE(%esi)
```

代码中注释了各寄存器的内容映像在堆栈上的位置，不过注意这是按压入堆栈的次序相对于堆栈原点的位移，是从上往下计算的。前面的 5 项数据就是 CPU 自动压入的，而软件压入堆栈的第一项数据是数值 0，这是将来返回给用户空间程序的出错代码，暂时设为 0。把段寄存器 FS 的内容压入堆栈是因为要把 FS 设置成指向数据结构 KPCR、即“处理器控制区”。这个段寄存器在用户空间指向当前线程的 TEB，在内核中则指向 KPCR。FS 指向 KPCR 以后就可以从中获取指向当前线程的“异常处理队列”的指针，并把这个指针也保存在堆栈上，并把 KPCR 中的这个指针设置成-1，这是与“结构化异常处理”、即 SEH 有关的操作，并非我们此刻所关心的话题。KPCR 中还有个指针指向当前线程的 ETHRAD 数据结构，这里让寄存器 ESI 指向这个数据结构。

所谓“先前模式(Previous Mode)”，是说 CPU 在进入本次系统调用之前的运行模式，即用户模式或内核模式。这是 Windows 内核所特有的，原因是 Windows 内核允许在内核中进行系统调用，这就需要知道本次系统调用是在哪一个空间启动，因为这牵涉到应该从何处获取调用参数的问题。这个信息可以从先前的段寄存器 CS 中获取，段选择项的最低两位是运行级别 RPL，内核为 0 而用户空间为 3，所以检查其最低位即可获取这个信息。系统调用是通过 int 指令启动的，所以即便是在系统空间启动，CPU 也会自动把当时的 CS 随同 EIP 一起压入堆栈，因而这个寄存器的映像一定在陷阱框架中。而 ETHREAD 数据结构中(实际上是 KTHREAD 结构中)则有个 PreviousMode 字段，这里就把这信息记录在这个

字段中。然而，既然允许在内核中启动系统调用，那就有可能发生嵌套的系统调用，每次都需要记录先前模式，可是当前线程的 PreviousMode 字段只有一个。怎么办呢？这里就把前一次(上一层)系统调用的先前模式记录在本次系统调用的陷阱框架中。这样，对于嵌套的系统调用，就会形成一个先前模式的链、实际上是先前模式的堆栈。显然，前面对于“异常处理队列”指针的处理也与此相似。

我们继续往下看。

```
    /* Save other registers */
    pushl   %eax                                        // + 0x34
    pushl   %ecx                                        // + 0x38
    pushl   %edx                                        // + 0x3C
    pushl   %ds                                         // + 0x40
    pushl   %es                                         // + 0x44
    pushl   %gs                                         // + 0x48
    sub   $0x28, %esp                                   // + 0x70

#ifdef DBG
    /* Trick gdb 6 into backtracing over the system call */
    mov   0x6c(%esp), %ebx
    pushl   4(%ebx)         /* DebugEIP */              // + 0x74
#else
    pushl   0x60(%esp)   /* DebugEIP */                 // + 0x74
#endif
    pushl   %ebp            /* DebugEBP */              // + 0x78

    /* Load the segment registers */
    sti
    movw   $KERNEL_DS, %bx
    movw   %bx, %ds
    movw   %bx, %es

    /* Save the old trap frame pointer where EDX would be saved */
    movl   KTHREAD_TRAP_FRAME(%esi), %ebx
    movl   %ebx, KTRAP_FRAME_EDX(%esp)
    /* Allocate new Kernel stack frame */
    movl   %esp, %ebp

    /* Save a pointer to the trap frame in the TCB */
    movl   %ebp, KTHREAD_TRAP_FRAME(%esi)
```

这里对其余一些寄存器内容的保存是不言自明的。接着在堆栈指针上减去了 0x28，使堆栈指针下移了 40 个字节，在堆栈上跳过了陷阱框架中从 DebugArgMark 到 Dr7 的一共 10 个栈项，因为这些数据与常规的系统调用无关，而只与 Debug 有关。陷阱框架中的最后两项数据显然也是与 Debug 有关的。

至此，从代码中可以看出，程序的设计者认为陷阱框架已经形成了，此时堆栈指针 ESP 所指向的地方就被认为是陷阱框架所占区间的起点。所以，在将段寄存器 DS 和 ES 的值设置成 KERNEL_DS 以后，先将 KTHREAD 结构中 TrapFrame 字段的内容保存在陷阱框架中本来用于寄存器 EDX 映像的单元中，再通过寄存器 EBP 将其地址写入当前进程 KTHREAD 结构中的 TrapFrame 字段。另一方面，这样寄存器 EBP 也指向了陷阱框架的起点。KTHREAD 结构中的 TrapFrame 字段是个_KTRAP_FRAME 结构指针，此时堆栈上的陷阱框架已经与这个数据结构的定义相符(除与 VM86 有关的几个字段外)。

考虑到系统调用可能嵌套，显然应该把 TrapFrame 字段原来的值保存在堆栈上，但是这里利用了本来用于寄存器 EDX 映像的栈项，以求节约一点空间。在系统调用的时候，寄存器 EDX 用来传递用户空间堆栈上调用参数所在位置的，所以在系统调用完成以后就失去了意义。其实用户空间堆栈上的调用参数位置也并不非得要传递下来，因为用户空间的堆栈指针就保存在当前的陷井框架中固定的位置上，而知道了当时的用户空间堆栈指针，当然也就知道了参数所在的位置，所以通过 edx 传递参数所在位置只是使得进入内核以后的处理更方便一些。所以，利用这个栈项保存 TrapFrame 字段原来的值是可行的。

下面的一些与陷阱框架无关的操作就不是我们在这里所关心的了，况且在“ReactOS 怎样实现系统调用”一文中也已谈及，所以我们跳过这些操作、只关心与堆栈有关的操作，直至对目标函数的调用。

```
. . . . . .
/* Allocate space on our stack */
subl %ecx, %esp
/* Get pointer to function */
movl (%edi), %edi
movl (%edi, %eax, 4), %eax

/* Copy the arguments from the user stack to our stack */
shr $2, %ecx
movl %esp, %edi
cld
rep movsd
/* Do the System Call */
call    *%eax
movl    %eax, KTRAP_FRAME_EAX(%ebp)

/* Deallocate the kernel stack frame    */
movl %ebp, %esp
```

注意这里又通过 sub 指令对堆栈指针进行了调整，在堆栈上分配了一些空间。这里寄存器 ECX 持有调用参数所占的字节数，每个系统调用的参数所占的字节数都是预定的，从堆栈指针上减去这么多字节，就在堆栈上为这些参数分配了空间，然后就通过重复的 movsd 指令把调用参数从用户空间堆栈复制到系统空间堆栈上，重复的次数为 ECX 的数值除于 4，即把字节数换算成长字数。注意这并不意味着对用户空间的访问已经就此完成，因为此时复制过来的往往是指针，所以进入目标函数以后可能还需要把这些指针所指的数据复制到系统空间中来。

由于前面的一些我们已经略过的操作，这里寄存器 EAX 持有来自系统调用跳转表的函数指针，对具体系统调用目标函数的调用就是由这里的 call 指令实现的。

这 call 指令的执行使堆栈区间中当前框架的下方开始形成一个新的函数调用框架，因为至少这 call 指令要将返回地址压入堆栈。进入目标函数以后，堆栈指针 ESP 所指的就是由 CPU 压入堆栈的返回地址，就是代码中 movl 指令所在的地址。在这上面是调用参数，再上面就是陷阱框架。从道理上说这些调用参数也在陷阱框架之中，但是上面所说的“陷阱框架”是狭义的，只是指定义于 KTRAP_FRAME 数据结构中的那部分内容。系统调用实质上是跨地址空间的函数调用，如果把系统调用在内核中的入口_KiSystemService 看作一个函数的起点，那就应该有一个函数调用(执行)框架，这个函数框架的主体部分就是狭义的陷阱框架，而整个函数框架则可看成一个广义的陷阱框架。这样，可以认为每个返回地址所在的单元就是一个框架的起点。对于嵌套的函数调用，所形成的框架也是嵌套的，低层的框架嵌在高层的框架之中。不过人们往往是在比较宽松的语境下谈论堆栈框架，而并非总是基于严格的定义；作者也只是就概念而言，而并不涉及对于框架的严格定义。

进入目标函数之后，典型的开始几条指令一般总是类似于这样：

```
pushl     %ebp
movl      %esp, %ebp
subl      $12, %esp
```

在 x86 系统结构的程序中，寄存器 EBP 一般都用作“框架指针”，指向当前框架的起点(从上往下算)。不过按理说框架的起点是返回地址，因而第一条指令就应该是“movl %esp, %ebp”，但是这里有个保存 EBP 原有内容(这是上一层框架的指针)的问题，先得有一条“pushl %ebp”才行，所以“框架指针”指向的并非物理意义上的框架起点。这么一来，EBP 所指向的是当前框架中它本身老的映像，而 4(%ebp)、即 EBP 加 4 处的内容才是返回地址，8(%ebp)则是第一个调用参数，余类推。

然后，如果需要的话，会有一条类似于“subl $12, %esp”的指令，目的是在堆栈上为当前框架、即当前函数中的局部变量分配空间，具体的大小则取决于所减去的数值。这样，就可以借助框架指针访问局部变量了。例如-4(%ebp)就可能是其中的一个局部变量，具体的指派则由编译/汇编工具决定。到要返回的时侯，只要执行一条“movl %ebp, %esp”指令，就把堆栈指针拨回了框架的逻辑起点。再执行一条“popl %ebp”，就一方面恢复了 EBP 原有的内容、即上一层的框架指针，同时也使堆栈指针真正回到了框架的起点，为执行 ret 指令作好了准备。将堆栈指针拨回框架起点，特别是恢复了 EBP 的原值，就意味着丢弃了框架中的内容，例如当前函数的局部变量就不复存在了，所以这几条指令一定是放在最后(紧挨着 ret 指令)才执行的。一执行 ret 指令，这个框架就不存在了。

这只是就大多数的、典型的情况而言，有些函数的汇编代码中也许根本就不使用框架指针，但那并不意味着框架就不存在了，而只是情况特殊而已。

回到前面的代码，当 CPU 从目标函数返回时，寄存器 EAX 的内容就是函数的返回值、实际上是出错代码，这里把它写入框架中保存 EAX 映像的单元中，代码中的位移量 KTRAP_FRAME_EAX 定义为 0x44。注意这里对 EBP 的使用是特殊的，从前面代码中可以看出，它并不指向框架的起点，而是指向陷阱框架地址的起点，那就是堆栈上由上往下数，序号(从 0 开始)为 0x78 的单元所在处。于是 0x78-0x44=0x34，就是序号为 0x34 的所在。

至于堆栈上的调用参数，则有两种常用的方法可以使之消去。一种是在目标函数中使用带有调整堆栈指针功能的 ret 指令，因而使 CPU 在返回的过程中自动调整了堆栈指针，

所以返回后调用参数已经不在堆栈上了。那 CPU 怎么知道需要调整多少个字节呢？这是由汇编工具根据函数的参数表定义计算出来、编码在 ret 指令中的。另一种就是由调用者在返回以后通过 add 指令调整堆栈指针，使其跳过调用参数。而在上面的代码中则把 EBP 的值赋给 ESP，使其跳过陷阱框架之下的所有单元，调用参数当然也就随之而去了。

再往下就是从系统调用返回、系统空间堆栈逐步收缩、最后使陷阱框架消失的过程了。这基本上只是前述过程的逆操作，所以这里就不再详述，读者可以自己阅读 ReactOS 的有关源码。

注意寄存器 EAX 和 EDX 的内容虽然保存了，返回时也予以恢复，但是实际上它们原来的值在回到用户空间后已经不再使用，因为 EAX 是用来传递系统调用号的，系统调用完成以后当然失去了意义。而 EDX 是用来传递用户空间堆栈上调用参数所在位置的，所以在系统调用完成以后也失去了意义。不过，EAX 原来的值固然是失去了意义，但是这个寄存器是用来返回系统调用的函数值、一般是出错代码的，所以在返回时表面上是通过 pop 指令“恢复”其内容，其实却是被设置成需要返回的数值。

再看中断。同样，中断时陷阱框架的前一部分是由硬件形成的，后一部分则由中断响应入口处的汇编指令完成。其中由硬件形成的部分因发生中断时 CPU 的运行模式而有所不同。如果发生中断时 CPU 运行于用户空间，则为：

- SS
- ESP
- EFLAGS
- CS
- EIP

显然，这与从用户空间启动系统调用时相同。

而若发生中断时 CPU 运行于系统空间，则没有前面两项，而只有 EFLAGS、CS、和 EIP 三项。

内核中对于每个中断号都有个程序入口，不同中断号的程序基本上都一样，只是作为参数压入堆栈的“中断向量”各不相同。下面是 3 号中断程序入口_irq_handler_3 的代码：

```
_irq_handler_3:
    cld
    pusha
    pushl   %ds
    pushl   %es
    pushl   %fs
    pushl   %gs
    movl    $0xceafbeef,%eax
    pushl   %eax
    movw    $KERNEL_DS,%ax
    movw    %ax,%ds
    movw    %ax,%es
    movw    %ax,%gs
    movl    $PCR_SELECTOR, %eax
    movl    %eax, %fs
    pushl   %esp
```

```
pushl       $(IRQ_BASE + 3)
call  _KiInterruptDispatch
popl %eax
popl %eax
popl %eax
popl %gs
popl %fs
popl %es
popl %ds
popa
iret
```

这里的指令 pusha 是“push 所有通用寄存器”，popa 则相反。

以 call 指令为中心，这些代码可以分成三个部分。在 call 指令之前、除最后的两条 push 指令以外、是中断框架的形成阶段；之后则是中断框架的消亡阶段；而 call 指令对 KiInterruptDispatch()的调用是实质性操作的阶段。注意 call 指令前面的最后两条 push 指令把两个调用参数压入堆栈。其中之一是当时的堆栈指针(不过没有使用 EBP)，由于此时框架已经形成，堆栈指针实际上指向框架的起点地址、即地址最低点。另一个参数(IRQ_BASE+3)是“中断向量”、实际上就是绝对中断号。

对于因为中断而形成的框架，ReactOS 为其另外定义了一个数据结构：

```
typedef struct _KIRQ_TRAPFRAME
{
    ULONG Magic;
    ULONG Gs;
    ULONG Fs;
    ULONG Es;
    ULONG Ds;
    ULONG Eax;
    ULONG Ecx;
    ULONG Edx;
    ULONG Ebx;
    ULONG Esp;
    ULONG Ebp;
    ULONG Esi;
    ULONG Edi;
    ULONG Eip;
    ULONG Cs;
    ULONG Eflags;
} KIRQ_TRAPFRAME, *PKIRQ_TRAPFRAME;
```

显然，这个框架与前面因系统调用而形成的框架有明显的不同，而且也小得多。这意味着中断的进入与返回不能跟系统调用的进入与返回共用相同的代码，这我们已经看到了。相比之下，在 Linux 内核中它们基本上是共用相同代码的。

代码的作者称系统调用的框架为 KTRAP_FRAME、即“陷阱框架”，而称中断的框架为 KIRQ_TRAPFRAME、即“中断请求陷阱框架”。但是其实中断与陷阱并不是一回事，说起来又拗口，有时候还容易混淆，还不如称为“中断框架”更好。

代码中对段寄存器 DS、ES、GS、FS 的设置这里就不多说了，注意这里并不在意“先前模式”等等，所以比系统调用时形成的陷阱框架要简单一些。常数 0xceafbeef 就是数据结构中的 Magic。注意这个数据结构中并不包括当中断发生于用户空间时自动压入堆栈的 SS 和 ESP 两项数据，所以实际上并不完整。不过是否存在于这个数据结构中只是形式，是否存在于堆栈上才是实质；再说这些数据的恢复是由 CPU 在执行 ret 指令时自动完成的，所以跟编程关系不大。

下面可以看 KiInterruptDispatch()的代码了，我们还是把重点放在堆栈和框架。

```
VOID
KiInterruptDispatch (ULONG vector, PKIRQ_TRAPFRAME Trapframe)
{
    KIRQL old_level;
    KTRAP_FRAME KernelTrapFrame;
    PKTHREAD CurrentThread;
    PKTRAP_FRAME OldTrapFrame=NULL;

    . . . . . .
      /* Actually call the ISR. */
      KiInterruptDispatch2(vector, old_level);
    . . . . . .

    if (old_level==PASSIVE_LEVEL && Trapframe->Cs != KERNEL_CS)
    {
        CurrentThread = KeGetCurrentThread();
        if (CurrentThread!=NULL && CurrentThread->Alerted[1])
        {
            . . . . . .
            if (CurrentThread->TrapFrame == NULL)
            {
                OldTrapFrame = CurrentThread->TrapFrame;
                KeIRQTrapFrameToTrapFrame(Trapframe, &KernelTrapFrame);
                CurrentThread->TrapFrame = &KernelTrapFrame;
            }
            Ke386EnableInterrupts();
            KiDeliverApc(KernelMode, NULL, NULL);
            Ke386DisableInterrupts();

            if (CurrentThread->TrapFrame == &KernelTrapFrame)
            {
                    KeTrapFrameToIRQTrapFrame(&KernelTrapFrame, Trapframe);
                    CurrentThread->TrapFrame = OldTrapFrame;
```

```
            }
        }
    }
}
```

这里的 KiInterruptDispatch2()处理实际的中断服务。它的调用参数只有两个，一个是“中断向量”vector；另一个是中断发生前的 CPU 运行级别 old_level，这是由软件实现的级别，现在可以暂不深究。由此可以看出，这个函数的执行并不涉及陷阱框架。

本来，执行完 KiInterruptDispatch2()，就可以原路返回了，但是这里有个 APC 函数的问题。即使 APC 请求存在，在中断返回时执行 APC 函数也是有条件的，条件是中断发生于用户空间(Trapframe->Cs != KERNEL_CS)、并且不是发生于执行 APC 函数的过程中(old_level==PASSIVE_LEVEL)。我们在前面看到，启动 APC 函数在用户空间的执行要将原来的陷阱框架转换成 CONTEXT 结构、并保存到用户空间堆栈上，但那是对“正宗”的陷阱框架 KTRAP_FRAME 而言，而现在我们面对的是中断框架，所以要先通过 KeIRQTrapFrameToTrapFrame()将其转换成一个正宗陷阱框架，放在一个临时的数据结构 KernelTrapFrame 中，并使当前线程的 KTHREAD 结构中的指针 Trapframe 指向这个数据结构，以便把它转化成 CONTEXT 结构并保存到用户空间堆栈上。事后则再通过 KeTrapFrameToIRQTrapFrame() 转换回来。那么，使 CurrentThread->TrapFrame 指向 KernelTrapFrame 的时侯把它的原值保存在哪里呢？这一次不打扰陷阱框架上的 EDX 映像了。从代码中可以看到，这一次保存在一个局部变量 OldTrapFrame 中。还是在堆栈上，可是不在陷阱框架中，而在陷阱框架下面的函数调用框架中。还有个问题，要是中断嵌套怎么办？如果中断嵌套，那么新的中断就发生于系统空间，CPU 就不从 TSS 装入系统空间堆栈指针、而只是继续使用系统空间堆栈指针。于是，在前述函数调用框架的下方就又会形成一个新的陷阱框架，而陷阱框架的下方又会有函数调用框架，新的函数调用框架中又会有局部变量 OldTrapFrame。可是，这两个 OldTrapFrame 虽然都在堆栈上，却分属不同的框架，互相井水不犯河水。当然，是否允许中断嵌套则又是另一个问题了。

最后再看异常。异常的框架与系统调用的框架相同，都是 KTRAP_FRAME。

但是这里有个不同之处。在系统调用时寄存器 EDX 用来传递调用参数在用户空间堆栈上的位置，为此用户空间的程序在 int 0x2e 指令之前要先把 EDX 原有的内容保存在用户空间堆栈上，到返回用户空间后再予恢复。这样，在系统调用时可以把 KTHREAD 结构中的指针 Trapframe 保存在框架中 EDX 所在的单元中。然而异常就不同了。异常可以发生在任何时候，根本就不可能事先保存 EDX 的内容。

还有个不同之处，那就是从异常返回时不考虑 APC 函数，那倒好办。

我们以 14 号异常的程序入口_KiTrap14 为例来看如何解决保存 EDX 内容的问题。

```
_KiTrap14:
    pushl   %ebp
    pushl   %ebx
    pushl   %esi
    movl    $14, %esi
    jmp  _KiTrapProlog
```

_KiTrapProlog:

```
        pushl   %edi
        pushl   %fs
        . . . . . .
        /* Load the PCR selector into fs */
        movl    $PCR_SELECTOR, %ebx
        movl    %ebx, %fs

        /* Save the old exception list */
        movl    %fs:KPCR_EXCEPTION_LIST, %ebx
        pushl   %ebx

        /* Get a pointer to the current thread */
        movl    %fs:KPCR_CURRENT_THREAD, %edi
        . . . . . .
        /* Save the old previous mode */
        movl    $0, %ebx
        movb    %ss:KTHREAD_PREVIOUS_MODE(%edi), %bl
        pushl   %ebx
        . . . . . .

        /* Save other registers */
        pushl   %eax
        pushl   %ecx
        pushl   %edx
        pushl   %ds
        pushl   %es
        pushl   %gs
        movl    %dr7, %eax
        pushl   %eax            /* Dr7 */
        . . . . . .

        movl    %esp, %ebx
        movl    %esp, %ebp

        /* Save the old trap frame. */
        cmpl    $0, %edi
        je      .L7
        movl    %ss:KTHREAD_TRAP_FRAME(%edi), %edx
        pushl   %edx
        jmp     .L8
.L7:
        pushl   $0
.L8:
```

```
    /* Save a pointer to the trap frame in the current KTHREAD */
    cmpl        $0, %edi
    je          .L6
    movl        %ebx, %ss:KTHREAD_TRAP_FRAME(%edi)
.L6:

    /* Call the C exception handler */
    pushl       %esi
    pushl       %ebx
    call        _KiTrapHandler
    addl        $4, %esp
    addl        $4, %esp

    /* Get a pointer to the current thread */
    movl        %fs:KPCR_CURRENT_THREAD, %esi

        /* Restore the old trap frame pointer */
    popl        %ebx
    movl        %ebx, KTHREAD_TRAP_FRAME(%esi)

    /* Return to the caller */
    jmp         _KiTrapEpilog
```

从 14 号异常的入口_KiTrap14 开始，如果对照着 KTRAP_FRAME 数据结构的定义跟踪所有的 push 指令，就可以看到二者完全相符。而“movl %esp, %ebx”和“movl %esp, %ebp”这两条指令使寄存器 EBX 和 EBP 都指向了陷阱框架的地址起点，此后的 push 指令对堆栈的操作就不在 KTRAP_FRAME 数据结构的范围之内了。

这里 EDI 指向当前进程的 KTHREAD 数据结构，KTHREAD_TRAP_FRAME(%edi)则为该数据结构中的 Trapframe 字段。代码中先由一条 mov 指令将其装入寄存器 EDX，再把它压入堆栈。这样，就在堆栈上为其保存了一个副本，这与把它保存在局部变量中实质上是一样的。在此过程中虽然用到了 EDX，却没有涉及陷阱框架中的 EDX 映像。而在执行了 KiTrapHandler()以后，则由后面的 pop 指令和 mov 指令将所保存的副本恢复到 KTHREAD 数据结构的 Trapframe 字段中。