漫谈兼容内核之十九:
Windows 线程间的强相互作用

毛德操

在现代的计算机系统中，一项作业(Job)往往需要多个进程或线程的协作，而操作系统则要为进程或线程间的协作提供基础设施和机制上的支持。操作系统、特别是内核，提供什么样的设施和手段，系统中的进程之间和线程之间就会有什么样的相互作用。如果把一个系统比作一个社会，那么系统中的进程和线程就好像是社会中的成员。成员的行为和成员之间的关系有其“社会性”的一面、即互相影响的一面。例如一个线程的调度优先级就有社会性，因为这个线程的优先级高了就意味着其它线程的优先级相对降低了。再如一个线程睡眠就使其它线程得到了更多的运行机会。而进程间的通信和同步，则更是体现着进程间的相互控制和协调的一面。

以前说过，Linux 就像是一个比较自由化、各个成员比较有独立自主性的社会。Linux 进程(线程)之间直接作用的手段基本上就是包括 Signal 在内的进程间通信，而进程间通信基本上是在双方都自愿、至少是知情的条件下进行的温和行为。通过信号值为 SIGKILL 的 Signal“杀死”对方是唯一的例外。而别的剧烈作用，例如使另一个进程“挂起”即暂停其运行，将一个文件映射到另一个进程的用户空间等等，则根本就不提供这样的手段。所以 Linux 进程基本上不具备直接控制、支配另一个进程的能力。所以说，Linux 进程(线程)之间的作用是“弱相互作用”而不是“强相互作用”。

而 Windows 就不同了。Windows 允许进程/线程之间的“强相互作用”，并为此提供手段、即系统调用。下面介绍 Windows 所提供的线程间控制和监视手段，其中有些就是“强作用”。至于以前讲到的跨进程操作，那就不止是“强作用”、而已经是“粗暴作用”了。

Windows 为进程/线程间(包括对自身)的控制和信息获取提供了不少系统调用，我们考察其中比较重要的几个。

首先是系统调用 NtQueryInformationProcess()，用来获取一个已打开进程对象的各种信息。这大致上相当于 Linux 在目录/proc 下面提供的信息，但是种类更多。从 PROCESSINFOCLASS 类型的定义可以看出这些信息的种类之多(不过并非都已实现)：

```
typedef enum _PROCESSINFOCLASS {
    ProcessBasicInformation,
    ProcessQuotaLimits,
    ProcessIoCounters,
    ProcessVmCounters,
    ProcessTimes,
    ProcessBasePriority,
    ProcessRaisePriority,
    ProcessDebugPort,
    ProcessExceptionPort,
    ProcessAccessToken,
    ProcessLdtInformation,
    ProcessLdtSize,
```

```
    ProcessDefaultHardErrorMode,
    ProcessIoPortHandlers,
    ProcessPooledUsageAndLimits,
    ProcessWorkingSetWatch,
    ProcessUserModeIOPL,
    ProcessEnableAlignmentFaultFixup,
    ProcessPriorityClass,
    ProcessWx86Information,
    ProcessHandleCount,
    ProcessAffinityMask,
    ProcessPriorityBoost,
    ProcessDeviceMap,
    ProcessSessionInformation,
    ProcessForegroundInformation,
    ProcessWow64Information,
    ProcessImageFileName,
    ProcessLUIDDeviceMapsEnabled,
    ProcessBreakOnTermination,
    ProcessDebugObjectHandle,
    ProcessDebugFlags,
    ProcessHandleTracing,
    ProcessUnknown33,
    ProcessUnknown34,
    ProcessUnknown35,
    ProcessCookie,
    MaxProcessInfoClass
} PROCESSINFOCLASS;
```

其中许多种类都有相应的数据结构。每次调用 NtQueryInformationProcess()时都要指定一个种类，所返回的则是按相应数据结构组织的数据。这里有几个种类需要作些说明。首先是 ProcessBasicInformation，其相应的数据结构为 PROCESS_BASIC_INFORMATION：

```
typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS    ExitStatus;
    PPEB    PebBaseAddress;
    KAFFINITY    AffinityMask;
    KPRIORITY    BasePriority;
    HANDLE    UniqueProcessId;
    HANDLE    InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION;
```

这里指针 PebBaseAddress 的意义自明；AffinityMask 用于多处理器系统结构，表示该进程可以在那些处理器上运行；UniqueProcessId 和 InheritedFromUniqueProcessId 分别是本进程和父进程对象的 Handle。BasePriority 是进程的基本调度优先级。在 Windows 中受调度运

行的是线程，而线程的调度优先级由两部分构成，一部分是其所在进程的调度优先级，另一部分是线程在进程中的相对优先级。而进程的调度优先级又有基本优先级(BasePriority)和提升优先级(RaisePriority)，后者是为克服“优先级倒置”问题而临时提高了的优先级。此外，进程的优先级还因进程的性质和状态而分成不同的类，例如“空转”、“常规”、“实时”等等。所以 Windows 进程/线程的调度优先级是个相当复杂的话题。

类型 ProcessQuotaLimits 与“时间片”大小有关，是为“时间片”设置的上限。

类型 ProcessAccessToken 与进程的访问权限有关。每一个进程在创建时都被授予一个“证章(Token)”，实际上是个 PROCESS_ACCESS_TOKEN 数据结构，其下面是包括 TOKEN 在内的一系列数据结构，里面记载着该进程对各种对象的访问权限，进程的 EPROCESS 结构中则有个 PROCESS_ACCESS_TOKEN 指针。这要留到将来在谈到 Windows 的安全管理时再作介绍。

类型 ProcessCookie 就是指 EPROCESS 结构中的字段 Cookie，这是个由时间等等信息杂凑形成的特征值，其数值有一定的随机性。

其余的讲不胜讲。有些看看名称就可明白；有些现在暂不明白，以后随着对 Windows 的了解逐步深入自会慢慢明白。

现在我们看代码：

```
NTSTATUS STDCALL
NtQueryInformationProcess(IN    HANDLE ProcessHandle,
                  IN    PROCESSINFOCLASS ProcessInformationClass,
                  OUT PVOID ProcessInformation,
                  IN    ULONG ProcessInformationLength,
                  OUT PULONG ReturnLength    OPTIONAL)
{
    . . . . . .

    PreviousMode = ExGetPreviousMode();
    DefaultQueryInfoBufferCheck(ProcessInformationClass,
                                    PsProcessInfoClass,
                                    ProcessInformation,
                                    ProcessInformationLength,
                                    ReturnLength,
                                    PreviousMode,
                                    &Status);
    . . . . . .
    if(ProcessInformationClass != ProcessCookie)
    {
      Status = ObReferenceObjectByHandle(ProcessHandle,
                          PROCESS_QUERY_INFORMATION, PsProcessType,
                          PreviousMode, (PVOID*)&Process, NULL);
      . . . . . .
    }
    else if(ProcessHandle != NtCurrentProcess())
    {
```

```
   /* retreiving the process cookie is only allowed for the calling process
      itself! XP only allowes NtCurrentProcess() as process handles even if a
      real handle actually represents the current process. */
   return STATUS_INVALID_PARAMETER;
}

switch (ProcessInformationClass)
   {
    case ProcessBasicInformation:
    {
      PPROCESS_BASIC_INFORMATION ProcessBasicInformationP =
                              (PPROCESS_BASIC_INFORMATION)ProcessInformation;

      _SEH_TRY
      {
        ProcessBasicInformationP->ExitStatus = Process->ExitStatus;
        ProcessBasicInformationP->PebBaseAddress = Process->Peb;
        ProcessBasicInformationP->AffinityMask = Process->Pcb.Affinity;
        ProcessBasicInformationP->UniqueProcessId = Process->UniqueProcessId;
        . . . . . .
        if (ReturnLength)
        {
          *ReturnLength = sizeof(PROCESS_BASIC_INFORMATION);
        }
      }
      . . . . . .
      break;
    }
    case ProcessQuotaLimits:
    case ProcessIoCounters:
    . . . . . .
    case ProcessLdtInformation:
    case ProcessWorkingSetWatch:
    case ProcessWx86Information:
       Status = STATUS_NOT_IMPLEMENTED;
       break;
    case ProcessHandleCount:
    . . . . . .
    case ProcessWow64Information:
      Status = STATUS_NOT_IMPLEMENTED;
      break;
    case ProcessVmCounters:
    . . . . . .
     /* Note: The following 10 information classes are verified to not be
```

```
            implemented on NT, and do indeed return STATUS_INVALID_INFO_CLASS; */
        case ProcessBasePriority:
        case ProcessRaisePriority:
        case ProcessExceptionPort:
        case ProcessAccessToken:
        case ProcessLdtSize:
        case ProcessIoPortHandlers:
        case ProcessUserModeIOPL:
        case ProcessEnableAlignmentFaultFixup:
        case ProcessAffinityMask:
        case ProcessForegroundInformation:
        default:
                Status = STATUS_INVALID_INFO_CLASS;
    }
    if(ProcessInformationClass != ProcessCookie)
    {
      ObDereferenceObject(Process);
    }
    return Status;
}
```

ProcessInformation 是用来返回结果的缓冲区，参数 ProcessInformationClass 指定什么信息类型，这缓冲区就被用作什么数据结构。这里先由宏操作 DefaultQueryInfoBufferCheck()根据信息类型对缓冲区的大小与映射作一检查。然后根据目标进程的 Handle 获取指向其 EPROCESS 结构的指针。这里有个特例，就是 Process->Cookie 只允许由本进程读取，别的都可以跨进程读取。下面就是对具体信息的获取了，这些信息大多来自目标进程的 EPROCESS 结构，所以这里只列出对 ProcessBasicInformation 类信息的读出，别的就省略了。还有些类型的信息是不允许读出的，有的则尚未实现。

系统调用 NtQueryInformationProcess()只是用于获取信息，与其相对应的另一个系统调用 NtSetInformationProcess()就是用来设置数据的了。但是，设置一个进程对象的某些数据，实际上就改变了它某些方面的性状和行为特性。例如设置其 ProcessPriorityClass 就改变了目标进程、从而其所有线程的调度优先级。又如设置其 ProcessAccessToken 就等于是给它换发了一个证章。这样的操作，如果是由父进程对子进程实施，那倒还无可非语。事实上读者已经看到，在 CreateProcessW()中就是通过 NtSetInformationProcess()设置子进程的优先级类型的。但是，如果是发生在没有父子关系的两个进程之间，那就属于进程间的“强作用”了。至于这个函数的代码，因为简单，我们就不看了。

进程间如此，线程间也是如此，Windows 也提供了 NtQueryInformationThread()和 NtSetInformationThread()两个系统调用，操作的目标是已经打开的线程对象。为此先要知道目标线程的“客户 ID”、即 CID，然后通过 NtOpenThread()打开其线程对象，再把由此得到的 Handle 作为这两个系统调用的参数。

如果说这已经属于进程间和线程间的“强作用”，那系统调用 NtSuspendThread()就更加了。这个系统调用的作用是使另一个正在运行中的线程被“挂起”、即暂停运行。为此，同样先要打开目标线程对像，并且在打开的时候就说明要求允许对其实行挂起/恢复的操作，

然后把目标线程对像的 Handle 用于 NtSuspendThread()的调用。

```
NTSTATUS STDCALL
NtSuspendThread(IN HANDLE ThreadHandle,
                IN PULONG PreviousSuspendCount    OPTIONAL)
{
   PETHREAD Thread;
   NTSTATUS Status;
   ULONG Count;

   PAGED_CODE();

   Status = ObReferenceObjectByHandle(ThreadHandle, THREAD_SUSPEND_RESUME,
                                      PsThreadType, UserMode, (PVOID*)&Thread, NULL);
   . . . . . .
   Status = PsSuspendThread(Thread, &Count);
   . . . . . .
   if (PreviousSuspendCount != NULL)
   {
        *PreviousSuspendCount = Count;
   }
   ObDereferenceObject ((PVOID)Thread);
   return STATUS_SUCCESS;
}
```

注意在 ObReferenceObjectByHandle()时使用的参数 THREAD_SUSPEND_RESUME，当初打开这个线程对象时必须说明允许此类操作。

线程的 KTHREAD 数据结构中有个字段 SuspendCount，正常情况下其数值为 0，如果大于 0 就说明该线程已被挂起；要是大于 1 就说明已经有不止一次的挂起尚未恢复。参数 PreviousSuspendCount 就是用来返回本次操作之前的 SuspendCount。

显然，具体的操作是由 PsSuspendThread()实现的：

[NtSuspendThread() > PsSuspendThread()]

```
NTSTATUS    PsSuspendThread(PETHREAD Thread, PULONG PreviousSuspendCount)
{
   ULONG OldValue;

   ExAcquireFastMutex(&SuspendMutex);
   OldValue = Thread->Tcb.SuspendCount;
   Thread->Tcb.SuspendCount++;
   if (!Thread->Tcb.SuspendApc.Inserted)
   {
        if (!KeInsertQueueApc(&Thread->Tcb.SuspendApc,
```

```
                                        NULL, NULL, IO_NO_INCREMENT))
        {
          Thread->Tcb.SuspendCount--;
          ExReleaseFastMutex(&SuspendMutex);
          return(STATUS_THREAD_IS_TERMINATING);
        }
    }
    ExReleaseFastMutex(&SuspendMutex);
    if (PreviousSuspendCount != NULL)
    {
        *PreviousSuspendCount = OldValue;
    }
    return(STATUS_SUCCESS);
}
```

所谓“挂起”一个线程实际包括两方面的操作。一是递增 KTHREAD 结构中的 SuspendCount 字段，这个字段的值表示目标线程已经有了几次尚未恢复的“挂起”操作。二是为此线程挂入一个 APC 请求。这两项操作都应该在临界区中排它地进行，所以内核中提供了一个互斥门 SuspendMutex，用它来构成用于挂起/恢复操作的临界区。注意这里的 Thread 代表着目标线程、而不是当前进程(除非当前进程就是目标线程)。

这里的 Thread->Tcb.SuspendApc 是创建线程时设置好的，我们不妨回顾一下线程对象的初始化：

```
KeInitializeThread(PKPROCESS Process, PKTHREAD Thread, BOOLEAN First)
{
    . . . . . .
    /* Initialize the Suspend APC */
    KeInitializeApc(&Thread->SuspendApc, Thread, OriginalApcEnvironment,
                    PiSuspendThreadKernelRoutine,
                    PiSuspendThreadRundownRoutine,
                    PiSuspendThreadNormalRoutine,
                    KernelMode, NULL);
    /* Initialize the Suspend Semaphore */
    KeInitializeSemaphore(&Thread->SuspendSemaphore, 0, 128);
    . . . . . .
}
```

其中 PiSuspendThreadKernelRoutine()和 PiSuspendThreadRundownRoutine()都是空函数，但 PiSuspendThreadNormalRoutine()不是：

```
VOID STDCALL
PiSuspendThreadNormalRoutine(PVOID NormalContext,
                             PVOID SystemArgument1, PVOID SystemArgument2)
{
```

```
    PETHREAD CurrentThread = PsGetCurrentThread();
    while (CurrentThread->Tcb.SuspendCount > 0)
    {
        KeWaitForSingleObject(&CurrentThread->Tcb.SuspendSemaphore,
                                                    0, UserMode, TRUE, NULL);
    }
}
```

就是说，当目标进程下一次被调度运行时，就会先在内核中执行这个 APC 函数，从而在其自身的“信号量”SuspendSemaphore 上执行一次 P 操作。但是这个信号量的初值是 0，所以这个线程势必会进入睡眠，这就是被“挂起”了。那么什么时候才能恢复运行呢？首先得要从 KeWaitForSingleObject()中被唤醒，就是必须有谁对这个“信号量”执行一次 V 操作。同时，其 KTHREAD 结构中的 SuspendCount 又必须为 0，否则即使唤醒了还会调头又睡。实际上，如前所述，Windows 提供的信号量是变型的，因为它不取负值。而计数值 SuspendCount 和这里的“信号量”合在一起就相当于“原型”的信号量。注意这里的当前线程 CurrentThread 实际上就是目标线程，因为是目标线程在执行这个 APC 函数。

这里还有个问题，所谓挂起“正在运行的”目标线程是什么意思呢？显然，真正“正在运行”的线程就是当前线程、即调用并执行 NtSuspendThread()的线程，而不是目标进程。但是目标进程之所以并不真的在运行是因为被线程调度暂时剥夺了运行权，例如因为当前这个进程的优先级更高，或者用完了本次运行的时间配额，但是它的状态仍是“就绪”状态，宏观地看这就算是正在运行了。那么要是目标线程已经睡眠会怎样呢？如果已经睡眠，那就不会被调度运行，因而就不会执行这个 APC 函数。但是，一旦原来的睡眠被唤醒，就早晚会被调度运行，从而就会因执行 APC 函数而又进入睡眠，这一次就是被挂起了。

与 NtSuspendThread()相对应的系统调用是 NtResumeThread()，其作用是使被挂起的目标线程恢复运行。

```
NTSTATUS STDCALL
NtResumeThread(IN HANDLE ThreadHandle, IN PULONG SuspendCount    OPTIONAL)
{
    . . . . . .

    Status = ObReferenceObjectByHandle (ThreadHandle, THREAD_SUSPEND_RESUME,
                                        PsThreadType, UserMode, (PVOID*)&Thread, NULL);
    . . . . . .
    Status = PsResumeThread (Thread, &Count);
    . . . . . .
    ObDereferenceObject ((PVOID)Thread);

    return STATUS_SUCCESS;
}
```

同样，具体的操作是由 PsResumeThread()完成的：

[NtResumeThread() > PsResumeThread()]

```
NTSTATUS PsResumeThread (PETHREAD Thread, PULONG SuspendCount)
{
  ExAcquireFastMutex (&SuspendMutex);
  if (SuspendCount != NULL)
  {
      *SuspendCount = Thread->Tcb.SuspendCount;
  }
  if (Thread->Tcb.SuspendCount > 0)
  {
      Thread->Tcb.SuspendCount--;
      if (Thread->Tcb.SuspendCount == 0)
      {
        KeReleaseSemaphore (&Thread->Tcb.SuspendSemaphore,
                                          IO_NO_INCREMENT, 1, FALSE);
      }
  }
  ExReleaseFastMutex (&SuspendMutex);
  return STATUS_SUCCESS;
}
```

首先是递减目标线程的 SuspendCount。当递减到 0 的时候，目标进程的所有“挂起”都已被撤销，可以恢复运行了。此时对其信号量 SuspendSemaphore 执行一次 V 操作，从而将其唤醒。注意在前面 PiSuspendThreadNormalRoutine()代码中的 KeWaitForSingleObject()是放在一个 while 循环中，目标线程醒来以后又要检查 SuspendCount 是否大于 0，如果大于 0 则又要睡眠等待。这是因为虽然此刻的 SuspendCount 已经是 0，但是目标线程被唤醒以后未必立刻就被调度运行，此前可能会有别的线程先有机会运行，而这些线程有可能又对目标线程实行“挂起”操作。

读者应该十分明确一个概念，那就是：在一个系统中，除当前线程以外的所有线程都不在运行，而不在运行的线程肯定都进了内核。为什么呢？因为线程的“运行权”都是在内核中被放弃或被剥夺的。因系统调用进入内核而被阻塞就不必说了；即使是在用户空间好好运行的线程，既然被暂时剥夺了运行权，那就说明发生了线程调度，而强制的(剥夺性的)线程调度只发生于 CPU 从内核返回用户空间的途中。那是怎么进的内核呢？除系统调用之外就只有两种可能，即异常和中断。所以，在用户空间好好运行的线程突然被剥夺了运行，最大的可能就是发生了中断、例如时钟中断，从而导致了线程调度。所不同的是有的线程进入了睡眠，有的则没有、而只是在就绪队列中等待再次被调度运行。这一点，在 Linux 上是这样，在 Windows 上也是一样。

既然除当前进程之外所有的线程都在内核中，这些线程的内核堆栈中就留有它们进入内核时所保留的“现场”，又称“上下文(Context)”，实际上就是它们进入内核时各个寄存器的内容，所以这是在用户空间的上下文。Windows 提供了读/写这些上下文的系统调用，这就是 NtGetContextThread()和 NtSetContextThread()，前者用于获取一个线程的上下文，后者用于改变一个线程的上下文。当然，这里的目标线程必须事先已经打开。

如果说获取一个线程的上下文只是窥探隐私，那么改变一个线程的上下文可就不简单了。比方说，改变目标线程上下文中的 Eip 映像、即寄存器 EIP 的内容，就可以使目标线程受调度运行而返回用户空间时脱离原来的轨道，“返回”到别的地方去。在“跨进程操作”那篇漫谈中我们看到一个进程可以在另一个进程的用户空间分配存储区间、把一段程序拷贝过去，再在目标进程中创建一个线程并令其执行。其实，连创建线程都没有必要，只要改变其中一个线程的上下文，使其返回到“栽赃”进去的程序入口就行了。显然，线程间如此这般的作用是强作用而不是弱作用。事实上，这两个系统调用是供程序调试工具做 Debug 用的，调试程序的时候就得使被调试的线程指哪去哪，这就是通过改变目标线程的上下文实现的。但是，调试工具也好，恶意代码也罢，一旦打开了目标线程，就可以对其为所欲为了。由此也可以看出，Windows 系统中对于打开进程对象和线程对象的安全保护措施是何等重要。

NtGetContextThread()和 NtSetContextThread()两个系统调用的代码基本上是对称的，所以我们在这里只看后者的代码。当然，目标线程必须已被打开，并且已经准备好了一个伪造的上下文，一般是先通过 NtGetContextThread()获取目标线程的上下文，再加以篡改而成。

```
NTSTATUS STDCALL
NtSetContextThread(IN HANDLE ThreadHandle, IN PCONTEXT ThreadContext)
{
  . . . . . .

  . . . . . .
  PreviousMode = ExGetPreviousMode();
  if(PreviousMode != KernelMode)
  {
    _SEH_TRY . . . _SEH_END;
    . . . . . .
  }

  Status = ObReferenceObjectByHandle(ThreadHandle, THREAD_SET_CONTEXT,
                                     PsThreadType, PreviousMode, (PVOID*)&Thread, NULL);
  if(NT_SUCCESS(Status))
  {
    if (Thread == PsGetCurrentThread())
    {
      /* I don't know if trying to set your own context makes much
         sense but we can handle it more efficently. */
      KeContextToTrapFrame(&Context, Thread->Tcb.TrapFrame);
    }
    else
    {
      KeInitializeEvent(&Event, NotificationEvent, FALSE);
      KeInitializeApc(&Apc, &Thread->Tcb, OriginalApcEnvironment,
                      KeSetContextKernelRoutine,
                      KeGetSetContextRundownRoutine,
```

```
                                NULL, KernelMode, (PVOID)&Context);
        if (!KeInsertQueueApc(&Apc,
                                (PVOID)&Event, (PVOID)&Status, IO_NO_INCREMENT))
        {
          Status = STATUS_THREAD_IS_TERMINATING;
        }
        else
        {
          Status = KeWaitForSingleObject(&Event, 0, KernelMode, FALSE, NULL);
        }
      }
      ObDereferenceObject(Thread);
    }
    return Status;
}
```

首先要根据 Handle 取得目标线程数据结构的指针，而 ObReferenceObjectByHandle()在这里设置了一道防线。参数 THREAD_SET_CONTEXT 是个标志位，表示访问目标对象的目的是设置上下文，这必须符合打开目标线程时所申明的操作范围。而在打开目标对象时的安全保护机制则把所要求的操作范围作为判断是否允许打开的依据之一。

具体的操作是由内核 APC 函数实现的。这里只是为 APC 函数的执行作了些准备，并提出 APC 请求，然后就睡眠等待具体操作的完成，而事件 Event 的作用就是作为等待/唤醒机制的载体。此外，由于参数 ThreadContext 所指的上下文数据结构是在用户空间，所以先要把它复制到内核中，就是这里的 Context。注意调用 KeInitializeApc()时的第二个参数 &Thread->Tcb，这个参数是个 KTHREAD 指针。但是这里的 Thread 指向目标线程、而不是当前线程的 ETHREAD 数据结构。所以，这里的 APC 请求是对目标线程的 APC 请求，而不是对当前进程的 APC 请求。

于是，当前进程就在 KeWaitForSingleObject()中进入了睡眠，内核将调度别的线程运行。

当目标线程被调度运行，准备返回用户空间的前夕，就到了它检查 APC 请求和执行 APC 函数的时候。由于已经有 APC 请求在队列中等候，目标线程就会执行 APC 函数 KeSetContextKernelRoutine()。

```
VOID STDCALL
KeSetContextKernelRoutine(PKAPC Apc, PKNORMAL_ROUTINE* NormalRoutine,
        PVOID* NormalContext, PVOID* SystemArgument1, PVOID* SystemArgument2)
{
  PKEVENT Event;
  PCONTEXT Context;
  PNTSTATUS Status;

  Context = (PCONTEXT)(*NormalContext);
  Event = (PKEVENT)(*SystemArgument1);
  Status = (PNTSTATUS)(*SystemArgument2);
  KeContextToTrapFrame(Context, KeGetCurrentThread()->TrapFrame);
```

```
   *Status = STATUS_SUCCESS;
   KeSetEvent(Event, IO_NO_INCREMENT, FALSE);
}
```

注意此时的“当前线程”就是目标线程，所以 KeGetCurrentThread()所返回的就是目标线程的 KTHREAD 结构指针，结构中的 TrapFrame 则指向其内核堆栈中的“陷阱框架”，那就是这个线程进入内核时所保留的“现场”，也即“上下文”。

[KeSetContextKernelRoutine() > KeContextToTrapFrame()]

```
VOID KeContextToTrapFrame(PCONTEXT Context, PKTRAP_FRAME TrapFrame)
{
    if ((Context->ContextFlags & CONTEXT_CONTROL) == CONTEXT_CONTROL)
    {
       TrapFrame->Esp = Context->Esp;
       TrapFrame->Ss = Context->SegSs;
       TrapFrame->Cs = Context->SegCs;
       TrapFrame->Eip = Context->Eip;
       TrapFrame->Eflags = Context->EFlags;
       TrapFrame->Ebp = Context->Ebp;
    }
    if ((Context->ContextFlags & CONTEXT_INTEGER) == CONTEXT_INTEGER)
    {
       TrapFrame->Eax = Context->Eax;
       TrapFrame->Ebx = Context->Ebx;
       TrapFrame->Ecx = Context->Ecx;
       TrapFrame->Edx = Context->Edx;
       TrapFrame->Esi = Context->Esi;
       TrapFrame->Edi = Context->Edi;
    }
    if ((Context->ContextFlags & CONTEXT_SEGMENTS) == CONTEXT_SEGMENTS)
    {
       TrapFrame->Ds = Context->SegDs;
       TrapFrame->Es = Context->SegEs;
       TrapFrame->Fs = Context->SegFs;
       TrapFrame->Gs = Context->SegGs;
    }
    if ((Context->ContextFlags & CONTEXT_FLOATING_POINT) ==
                                                    CONTEXT_FLOATING_POINT)
    {
      /* Not handled.      This should be handled separately I think.    - blight */
    }
    if ((Context->ContextFlags & CONTEXT_DEBUG_REGISTERS) ==
```

```
                                                    CONTEXT_DEBUG_REGISTERS)
    {
      /* Not handled */
    }
}
```

这个函数的作用就是把 CONTEXT 数据结构中的内容复制到当前进程的内核堆栈中去，只是作为 KTRAP_FRAME 数据结构的“陷阱框架”与 CONTEXT 数据结构略有不同。后者有个字段 ContextFlags 是一些标志位，用来控制把哪一些内容复制到陷阱框架中，例如 CONTEXT_CONTROL 就表示要把 Eip、Esp 等等控制着程序执行的寄存器内容复制过去。当然，这些标志位是在调用 NtSetContextThread()之前就设置好了的。

复制完了以后，回到 KeSetContextKernelRoutine()，下一步就是通过 KeSetEvent()唤醒 NtSetContextThread()的调用者了。

前面提出 APC 请求时还有个函数是 KeGetSetContextRundownRoutine()。所谓 RundownRoutine 都是为要结束生命的线程准备的，目的是对于跟 APC 请求有关的事务作个了断，因为此时再执行常规的 APC 函数已经没有意义了。当一个线程要结束生命退出运行的时候，就要检查是否有 APC 请求存在，以及 APC 请求中是否有这样的 RundownRoutine 函数存在，如果有的话就要加以执行。不过在 0.2.6 版 ReactOS 的代码中还没有实现这一点，在 0.2.9 版中倒是已经有了。那么这些 RundownRoutine 函数到底干些什么，作些什么样的了断呢？看一下这里 KeGetSetContextRundownRoutine()就可以明白：

```
VOID STDCALL
KeGetSetContextRundownRoutine(PKAPC Apc)
{
  PKEVENT Event;
  PNTSTATUS Status;

  Event = (PKEVENT)Apc->SystemArgument1;
  Status = (PNTSTATUS)Apc->SystemArgument2;
  (*Status) = STATUS_THREAD_IS_TERMINATING;
  KeSetEvent(Event, IO_NO_INCREMENT, FALSE);
}
```

显而易见，就 NtSetContextThread()的 APC 请求而言，这就是唤醒正在睡眠等待前述事件发生的线程。要不然，NtSetContextThread()的调用者可就长眠不起了。

读者也许会问，改变目标线程的上下文，为什么要采取这么曲折的方法呢？把线程调度锁住，然后根据目标线程数据结构中的指针 TrapFrame 找到其内核堆栈上的“陷阱框架”，不是也能修改其上下文吗？这里有个问题，就是当目标线程被唤醒的时侯很可能还要在内核中运行一阵，在这个过程中可能会改变其(用户空间)上下文中某些寄存器的值，这样就可能会把所设置的值给覆盖了。而 APC 函数是在返回用户空间的途中执行的，此时上下文中各寄存器的值已是板上钉钉，一经改变就是最后版本了。所以，用这样显得有些曲折的方法来实现对目标线程上下文的修改，确实还是很有道理的。

此外，Windows 还提供了将另一个线程从睡眠等待中“惊醒(Alert)”过来的手段，这就

是系统调用 NtAlertThread()。所谓“惊醒”，是说目标线程本来在睡眠等待某些条件得到满足，如果得不到满足就会一直睡眠等待下去，但是现在另一个线程要它醒过来别等了。笼统地说，这也是唤醒，但是“唤醒”一般用于条件得到满足的时侯，所以就另称之为“惊醒”，以示区别。其实，“超时(Timeout)”和惊醒是很相似的，只不过前者是由内核发起，而后者是由另一个线程发起的。不过读者很快就会看到，并非所有的睡眠等待都可以被惊醒。

下面我们看 NtAlertThread()的代码。

```
NTSTATUS
STDCALL
NtAlertThread (IN HANDLE ThreadHandle)
{
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    . . . . . .

    /* Reference the Object */
    Status = ObReferenceObjectByHandle(ThreadHandle, THREAD_SUSPEND_RESUME,
                                       PsThreadType, PreviousMode, (PVOID*)&Thread, NULL);
    /* Check for Success */
    if (NT_SUCCESS(Status)) {
        /*
         * Do an alert depending on the processor mode. If some kmode code wants to
         * enforce a umode alert it should call KeAlertThread() directly. If kmode
         * code wants to do a kmode alert it's sufficient to call it with Zw or just
         * use KeAlertThread() directly
         */
        KeAlertThread(&Thread->Tcb, PreviousMode);
        /* Dereference Object */
        ObDereferenceObject(Thread);
    }
    /* Return status */
    return Status;
}
```

这段代码就不需要什么解释了，我们往下看 KeAlertThread()：

[NtAlertThread() > KeAlertThread()]

```
BOOLEAN STDCALL
KeAlertThread(PKTHREAD Thread, KPROCESSOR_MODE AlertMode)
{
    . . . . . .

    /* Acquire the Dispatcher Database Lock */
    OldIrql = KeAcquireDispatcherDatabaseLock();
```

```
    /* Save the Previous State */
    PreviousState = Thread->Alerted[AlertMode];
    /* Return if Thread is already alerted. */
    if (PreviousState == FALSE) {
        /* If it's Blocked, unblock if it we should */
        if (Thread->State == THREAD_STATE_BLOCKED &&
            (AlertMode == KernelMode || Thread->WaitMode == AlertMode) &&
            Thread->Alertable)
        {
            KiAbortWaitThread(Thread, STATUS_ALERTED,
                                        THREAD_ALERT_INCREMENT);
        } else {
            /* If not, simply Alert it */
            Thread->Alerted[AlertMode] = TRUE;
        }
    }

    /* Release the Dispatcher Lock */
    KeReleaseDispatcherDatabaseLock(OldIrql);
    /* Return the old state */
    return PreviousState;
}
```

如前所述，线程的有些睡眠是可惊醒的，有些是不可惊醒的。对此我们不仿看一下KeWaitForSingleObject()的调用界面：

```
NTSTATUS KeWaitForSingleObject(
  IN PVOID    Object,
  IN KWAIT_REASON    WaitReason,
  IN KPROCESSOR_MODE    WaitMode,
  IN BOOLEAN    Alertable,
  IN PLARGE_INTEGER    Timeout    OPTIONAL);
```

这里的参数 Alertable 就规定了本次睡眠是否可惊醒。

而 WaitMode，则说明本次睡眠是用户模式、出于用户程序的要求，还是内核模式。其取值范围为 KernelMode 和 UserMode。用户模式的睡眠既可以被用户模式的要求惊醒，也可以被内核模式的要求惊醒。但是内核模式的睡眠不能被用户模式的要求惊醒。注意不要混淆 WaitMode 和 WaitType，后者的取值范围为 WaitAll 和 WaitAny。

所以，KTHREAD 结构中的 Alertable 字段就表明了当前的睡眠是否可惊醒。而 hread->WaitMode 则记录着本次睡眠的模式。如果各种条件都允许惊醒，就通过 KiAbortWaitThread()结束目标线程的等待状态，否则就只是记录下已经有过惊醒的要求。

由于具体的惊醒涉及将目标线程挂入就绪队列，这部分操作只能放在禁止线程调度的条件下进行，所以这里先通过 KeAcquireDispatcherDatabaseLock()将运行级别提高到 DISPATHER 级，并锁定线程调度队列，待操作完成以后再予恢复。

实际的操作则是由 KiAbortWaitThread()完成的：

[NtAlertThread() > KeAlertThread() > KiAbortWaitThread()]

```
VOID FASTCALL
KiAbortWaitThread(PKTHREAD Thread, NTSTATUS WaitStatus, KPRIORITY Increment)
{
    . . . . . .

    . . . . . .
    /* Remove the Wait Blocks from the list */
    WaitBlock = Thread->WaitBlockList;
    while (WaitBlock) {
        RemoveEntryList(&WaitBlock->WaitListEntry);
        WaitBlock = WaitBlock->NextWaitBlock;
    };
    /* Check if there's a Thread Timer */
    if (Thread->Timer.Header.Inserted) {

        /* Cancel the Thread Timer with the no-lock fastpath */
        Thread->Timer.Header.Inserted = FALSE;
        RemoveEntryList(&Thread->Timer.TimerListEntry);
    }
    /* Increment the Queue's active threads */
    if (Thread->Queue) {
        DPRINT("Incrementing Queue's active threads\n");
        Thread->Queue->CurrentCount++;
    }
    /* Reschedule the Thread */
    PsUnblockThread((PETHREAD)Thread, &WaitStatus, 0);
}
```

处于睡眠等待状态的线程通过其 WaitBlockList 中的 WaitBlock 与所等待的对象挂勾，每一个 WaitBlock 代表着一个对象，所以现在要把它们全部解脱出来。此外，如果进入睡眠等待时规定了超时，那就也要和定时器脱钩。最后则通过 PsUnblockThread()解除目标线程的阻塞，这个函数的代码我们以前已经看到过了。

最后，还有个系统调用 NtTerminateThread()，是用来结束一个已打开线程的生命。这个已打开线程当然可以是当前线程本身，但是更重要的是它也可以是别的线程。

总之，Windows 线程之间的作用是强作用，一个线程只要能打开另一个线程，不管是否属于同一个进程，就有了控制、支配这个被打开线程的权力。另一方面，读者在“跨进程操作”那篇漫谈中已经看到，一旦打开了另一个进程，也就取得了控制、支配这个被打开进程、包括其用户空间的权力。相比之下，Linux 的进程之间、线程之间是比较平等、独立的。就像“权力导致腐败”一样，Windows 线程之间的这种强作用也会带来问题、特别是安全问题。

为此，Windows 在对象的打开和操作两个环节上采取了安全保护措施：

- 在要求打开对象时，尤其是要求打开进程对象和线程对象时，要根据要求者的身份及其证章所示的权限、目标对象的性质和访问控制表、还有所要求的操作范围进行综合的判断，如果条件不合就拒绝打开。
- 在要求对已打开的对象进行操作时，要检查是否与打开该对象时所申明的操作范围相符，如果不符就拒绝操作。

但是，就像对文件的访问权限常常会设置不当一样，对进程、线程、以及其它对象的访问控制也常常会管理不当，这就带来了安全问题。这二者本质上是同一回事，但是后者往往更容易被忽略，因而更容易发生。