漫谈兼容内核之十二:
Windows 的 APC 机制

毛德操

前两篇漫谈中讲到，除 ntdll.dll 外，在启动一个新进程运行时，PE 格式 DLL 映像的装入和动态连接是由 ntdll.dll 中的函数 LdrInitializeThunk()作为 APC 函数执行而完成的。这就牵涉到了 Windows 的 APC 机制，APC 是“异步过程调用(Asyncroneus Procedure Call)”的缩写。从大体上说，Windows 的 APC 机制相当于 Linux 的 Signal 机制，实质上是一种对于应用软件(线程)的“软件中断”机制。但是读者将会看到，APC 机制至少在形式上与软件中断机制还是有相当的区别，而称之为“异步过程调用”确实更为贴切。

APC 与系统调用是密切连系在一起的，在这个意义上 APC 是系统调用界面的一部分。然而 APC 又与设备驱动有着很密切的关系。例如，ntddk.h 中提供“写文件”系统调用 ZwWriteFile()、即 NtWriteFile()的调用界面为：

```
NTSYSAPI
NTSTATUS
NTAPI
ZwWriteFile(
    IN HANDLE FileHandle,
    IN HANDLE Event OPTIONAL,
    IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
    IN PVOID ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PVOID Buffer,
    IN ULONG Length,
    IN PLARGE_INTEGER ByteOffset OPTIONAL,
    IN PULONG Key OPTIONAL
    );
```

这里有个参数 ApcRoutine，这是一个函数指针。什么时候要用到这个指针呢？原来，文件操作有“同步”和“异步”之分。普通的写文件操作是同步写，启动这种操作的线程在内核进行写文件操作期间被“阻塞(blocked)”而进入“睡眠”，直到设备驱动完成了操作以后才又将该线程“唤醒”而从系统调用返回。但是，如果目标文件是按异步操作打开的，即在通过 W32 的 API 函数 CreateFile()打开目标文件时把调用参数 dwFlagsAndAttributes 设置成 FILE_FLAG_OVERLAPPED，那么调用者就不会被阻塞，而是把事情交给内核、不等实际的操作完成就返回了。但是此时要把 ApcRoutine 设置成指向某个 APC 函数。这样，当设备驱动完成实际的操作时，就会使调用者线程执行这个 APC 函数，就像是发生了一次中断。执行该 APC 函数时的调用界面为：

```
typedef
VOID
(NTAPI *PIO_APC_ROUTINE) (IN PVOID ApcContext,
```

IN PIO_STATUS_BLOCK IoStatusBlock, IN ULONG Reserved);

这里的指针 ApcContext 就是 NtWriteFile()调用界面上传下来的，至于作什么解释、起什么作用，那是包括 APC 函数在内的用户软件自己的事，内核只是把它传递给 APC 函数。

在这个过程中，把 ApcRoutine 设置成指向 APC 函数相当于登记了一个中断服务程序，而设备驱动在完成实际的文件操作后就向调用者线程发出相当于中断请求的“APC 请求”，使其执行这个 APC 函数。

从这个角度说，APC 机制又应该说是设备驱动框架的一部分。事实上，读者以后还会看到，APC 机制与设备驱动的关系比这里所见的还要更加密切。此外，APC 机制与异常处理的关系也很密切。

不仅内核可以向一个线程发出 APC 请求，别的线程、乃至目标线程自身也可以发出这样的请求。Windows 为应用程序提供了一个函数 QueueUserAPC()，就是用于此项目的，下面是 ReactOS 中这个函数的代码：

```
DWORD STDCALL
QueueUserAPC(PAPCFUNC pfnAPC, HANDLE hThread, ULONG_PTR dwData)
{
  NTSTATUS Status;

  Status = NtQueueApcThread(hThread, IntCallUserApc,
                                    pfnAPC, (PVOID)dwData, NULL);
  if (Status)
    SetLastErrorByStatus(Status);

  return NT_SUCCESS(Status);
}
```

参数 pfnAPC 是函数指针，这就是 APC 函数。另一个参数 hThread 是指向目标线程对象(已打开)的 Handle，这可以是当前线程本身，也可以是同一进程中别的线程，还可以是别的进程中的某个线程。值得注意的是：如果目标线程在另一个进程中，那么 pfnAPC 必须是这个函数在目标线程所在用户空间的地址，而不是这个函数在本线程所在空间的地址。最后一个参数 dwData 则是需要传递给 APC 函数的参数。

这里的 NtQueueApcThread()是个系统调用。“Native API”书中有关于 NtQueueApcThread()的一些说明。这个系统调用把一个“用户 APC 请求”挂入目标线程的 APC 队列(更确切地说，是把一个带有函数指针的数据结构挂入队列)。注意其第二个参数是需要执行的 APC 函数指针，本该是 pfnAPC，这里却换成了函数 IntCallUserApc()，而 pfnAPC 倒变成了第三个参数，成了需要传递给 IntCallUserApc()的参数之一。IntCallUserApc()是 kernel32.dll 内部的一个函数，但是并未引出，所以不能从外部直接加以调用。

APC 是针对具体线程、要求由具体线程加以执行的，所以每个线程都有自己的 APC 队列。内核中代表着线程的数据结构是 ETHREAD，而 ETHREAD 中的第一个成分 Tcb 是 KTHREAD 数据结构，线程的 APC 队列就在 KTHREAD 里面：

```
typedef struct _KTHREAD
```

```
{
    . . . . . .
    /* Thread state (one of THREAD_STATE_xxx constants below) */
    UCHAR                State;                  /* 2D */
    BOOLEAN              Alerted[2];             /* 2E */
    . . . . . .
    KAPC_STATE           ApcState;               /* 34 */
    ULONG                ContextSwitches;        /* 4C */
    . . . . . .
    ULONG                KernelApcDisable;       /* D0 */
    . . . . . .
    PKQUEUE              Queue;                  /* E0 */
    KSPIN_LOCK           ApcQueueLock;           /* E4 */
    . . . . . .
    PKAPC_STATE          ApcStatePointer[2];     /* 12C */
    . . . . . .
    KAPC_STATE           SavedApcState;          /* 140 */
    UCHAR                Alertable;              /* 158 */
    UCHAR                ApcStateIndex;          /* 159 */
    UCHAR                ApcQueueable;           /* 15A */
    . . . . . .
    KAPC                 SuspendApc;             /* 160 */
    . . . . . .
} KTHREAD;
```

Microsoft 并不公开这个数据结构的定义，所以 ReactOS 代码中对这个数据结构的定义带有逆向工程的痕迹，每一行后面的十六进制数值就是相应结构成分在数据结构中的位移。这里我们最关心的是 ApcState，这又是一个数据结构、即 KAPC_STATE。可以看出，KAPC_STATE 的大小是 0x18 字节。其定义如下：

```
typedef struct _KAPC_STATE {
    LIST_ENTRY    ApcListHead[2];
    PKPROCESS     Process;
    BOOLEAN       KernelApcInProgress;
    BOOLEAN       KernelApcPending;
    BOOLEAN       UserApcPending;
} KAPC_STATE, *PKAPC_STATE, *__restrict PRKAPC_STATE;
```

显然，这里的 ApcListHead 就是 APC 队列头。不过这是个大小为 2 的数组，说明实际上(每个线程)有两个 APC 队列。这是因为 APC 函数分为用户 APC 和内核 APC 两种，各有各的队列。所谓用户 APC，是指相应的 APC 函数位于用户空间、在用户空间执行；而内核 APC，则相应的 APC 函数为内核函数。

读者也许已经注意到，KTHREAD 结构中除 ApcState 外还有 SavedApcState 也是 KAPC_STATE 数据结构。此外还有 ApcStatePointer[2]和 ApcStateIndex 两个结构成分。这是

干什么用的呢？原来，在 Windows 的内核中，一个线程可以暂时“挂靠(Attach)”到另一个进程的地址空间。比方说，线程 T 本来是属于进程 A 的，当这个线程在内核中运行时，如果其活动与用户空间有关(APC 就是与用户空间有关)，那么当时的用户空间应该就是进程 A 的用户空间。但是 Windows 内核允许一些跨进程的操作(例如将 ntdll.dll 的映像装入新创进程 B 的用户空间并对其进行操作)，所以有时候需要把当时的用户空间切换到别的进程(例如 B) 的用户空间，这就称为“挂靠(Attach)”，对此我将另行撰文介绍。在当前线程挂靠在另一个进程的期间，既然用户空间是别的进程的用户空间，挂在队列中的 APC 请求就变成“牛头不对马嘴”了，所以此时要把这些队列转移到别的地方，以免乱套，然后在回到原进程的用户空间时再于恢复。那么转移到什么地方呢？就是 SavedApcState。当然，还要有状态信息说明本线程当前是处于“原始环境”还是“挂靠环境”，这就是 ApcStateIndex 的作用。代码中为 SavedApcState 的值定义了一种枚举类型：

```
typedef enum _KAPC_ENVIRONMENT
{
   OriginalApcEnvironment,
   AttachedApcEnvironment,
   CurrentApcEnvironment
} KAPC_ENVIRONMENT;
```

实际可用于 ApcStateIndex 的只是 OriginalApcEnvironment 和 AttachedApcEnvironment，即 0 和 1。读者也许又要问，在挂靠环境下原来的 APC 队列确实不适用了，但不去用它就是，何必要把它转移呢？再说，APC 队列转移以后，ApcState 不是空下来不用了吗？问题在于，在挂靠环境下也可能会有(针对所挂靠进程的)APC 请求(不过当然不是来自用户空间)，所以需要有用于两种不同环境的 APC 队列，于是便有了 ApcState 和 SavedApcState。进一步，为了提供操作上的灵活性，又增加了一个 KAPC_STATE 指针数组 ApcStatePointer[2]，就用 ApcStateIndex 的当前值作为下标，而数组中的指针则根据情况可以分别指向两个 APC_STATE 数据结构中的一个。

这样，以 ApcStateIndex 的当前数值为下标，从指针数组 ApcStatePointer[2]中就可以得到指向 ApcState 或 SavedApcState 的指针，而要求把一个 APC 请求挂入队列时则可以指定是要挂入哪一个环境的队列。实际上，当 ApcStateIndex 的值为 OriginalApcEnvironment、即 0 时，使用的是 ApcState；为 AttachedApcEnvironment、即 1 时，则用的是 SavedApcState。

每当要求挂入一个 APC 函数时，不管是用户 APC 还是内核 APC，内核都要为之准备好一个 KAPC 数据结构，并将其挂入相应的队列。

```
typedef struct _KAPC
{
    CSHORT Type;
    CSHORT Size;
    ULONG Spare0;
    struct _KTHREAD* Thread;
    LIST_ENTRY ApcListEntry;
    PKKERNEL_ROUTINE KernelRoutine;
    PKRUNDOWN_ROUTINE RundownRoutine;
    PKNORMAL_ROUTINE NormalRoutine;
```

```
    PVOID NormalContext;
    PVOID SystemArgument1;
    PVOID SystemArgument2;
    CCHAR ApcStateIndex;
    KPROCESSOR_MODE ApcMode;
    BOOLEAN Inserted;
} KAPC, *PKAPC;
```

结构中的 ApcListEntry 就是用来将 KAPC 结构挂入队列的。注意这个数据结构中有三个函数指针，即 KernelRoutine、RundownRoutine、NormalRoutine。其中只有 NormalRoutine 才指向(执行)APC 函数的请求者所提供的函数，其余两个都是辅助性的。以 NtQueueApcThread() 为例，其请求者(调用者)QueueUserAPC() 所提供的函数是 IntCallUserApc()，所以 NormalRoutine 应该指向这个函数。注意真正的请求者其实是 QueueUserAPC()的调用者，真正的目标 APC 函数也并非 IntCallUserApc()，而是前面的函数指针 pfnAPC 所指向的函数，而 IntCallUserApc()起着类似于“门户”的作用。

现在我们可以往下看系统调用 NtQueueApcThread()的实现了。

```
NTSTATUS
STDCALL
NtQueueApcThread(HANDLE ThreadHandle, PKNORMAL_ROUTINE ApcRoutine,
            PVOID NormalContext, PVOID SystemArgument1, PVOID SystemArgument2)
{
    PKAPC Apc;
    PETHREAD Thread;
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    NTSTATUS Status;

    /* Get ETHREAD from Handle */
    Status = ObReferenceObjectByHandle(ThreadHandle, THREAD_SET_CONTEXT,
                                PsThreadType, PreviousMode, (PVOID)&Thread, NULL);
    . . . . . .
    /* Allocate an APC */
    Apc = ExAllocatePoolWithTag(NonPagedPool, sizeof(KAPC), TAG('P', 's', 'a', 'p'));
    . . . . . .
    /* Initialize and Queue a user mode apc (always!) */
    KeInitializeApc(Apc, &Thread->Tcb, OriginalApcEnvironment,
                        KiFreeApcRoutine, NULL, ApcRoutine, UserMode, NormalContext);

    if (!KeInsertQueueApc(Apc, SystemArgument1, SystemArgument2,
                                                        IO_NO_INCREMENT))
    {
        Status = STATUS_UNSUCCESSFUL;
```

```
    } else {
        Status = STATUS_SUCCESS;
    }

    /* Dereference Thread and Return */
    ObDereferenceObject(Thread);
    return Status;
}
```

先看调用参数。第一个参数是代表着某个已打开线程的 Handle，这说明所要求的 APC 函数的执行者、即目标线程、可以是另一个线程，而不必是请求者线程本身。第二个参数不言自明。第三个参数 NormalContext，以及后面的两个参数，则是准备传递给 APC 函数的参数，至于怎样解释和使用这几个参数是 APC 函数的事。看一下前面 QueueUserAPC()的代码，就可以知道这里的 APC 函数是 IntCallUserApc()，而准备传给它的参数分别为 pfnAPC、dwData、和 NULL，前者是真正的目标 APC 函数指针，后两者是要传给它的参数。

根据 Handle 找到目标线程的 ETHREAD 数据结构以后，就为 APC 函数分配一个 KAPC 数据结构，并通过 KeInitializeApc()加以初始化。

[NtQueueApcThread() > KeInitializeApc()]

```
VOID
STDCALL
KeInitializeApc(IN PKAPC Apc,
                IN PKTHREAD Thread,
                IN KAPC_ENVIRONMENT TargetEnvironment,
                IN PKKERNEL_ROUTINE KernelRoutine,
                IN PKRUNDOWN_ROUTINE RundownRoutine OPTIONAL,
                IN PKNORMAL_ROUTINE NormalRoutine,
                IN KPROCESSOR_MODE Mode,
                IN PVOID Context)
{
    . . . . . .

    /* Set up the basic APC Structure Data */
    RtlZeroMemory(Apc, sizeof(KAPC));
    Apc->Type = ApcObject;
    Apc->Size = sizeof(KAPC);

    /* Set the Environment */
    if (TargetEnvironment == CurrentApcEnvironment) {

        Apc->ApcStateIndex = Thread->ApcStateIndex;

    } else {
```

```
        Apc->ApcStateIndex = TargetEnvironment;
    }

    /* Set the Thread and Routines */
    Apc->Thread = Thread;
    Apc->KernelRoutine = KernelRoutine;
    Apc->RundownRoutine = RundownRoutine;
    Apc->NormalRoutine = NormalRoutine;

    /* Check if this is a Special APC, in which case we use KernelMode and no Context */
    if (ARGUMENT_PRESENT(NormalRoutine)) {

        Apc->ApcMode = Mode;
        Apc->NormalContext = Context;

    } else {

        Apc->ApcMode = KernelMode;
    }
}
```

这段代码本身很简单，但是有几个问题需要结合前面 NtQueueApcThread()的代码再作些说明。

首先，从 NtQueueApcThread()传下来的 KernelRoutine 是 KiFreeApcRoutine()，顾名思义这是在为将来释放 PKAPC 数据结构做好准备，而 RundownRoutine 是 NULL。

其次，参数 TargetEnvironment 说明要求挂入哪一种环境下的 APC 队列。实际传下来的值是 OriginalApcEnvironment，表示是针对原始环境、即当前线程所属(而不是所挂靠)进程的。注意代码中所设置的是 Apc->ApcStateIndex、即 PKAPC 数据结构中的 ApcStateIndex 字段，而不是 KTHREAD 结构中的 ApcStateIndex 字段。另一方面，ApcStateIndex 的值只能是 OriginalApcEnvironment 或 AttachedApcEnvironment，如果所要求的是 CurrentApcEnvironment 就要从 Thread->ApcStateIndex 获取当前的环境值。

最后，APC 请求的模式 Mode 是 UserMode。但是有个例外，那就是：如果指针 NormalRoutine 为 0，那么实际的模式变成了 KernelMode。这是因为在这种情况下没有用户空间 APC 函数可以执行，唯一将得到执行的是 KernelRoutine，在这里是 KiFreeApcRoutine()。这里的宏操作 ARGUMENT_PRESENT 定义为：

```
#define ARGUMENT_PRESENT(ArgumentPointer) \
  ((BOOLEAN) ((PVOID)ArgumentPointer != (PVOID)NULL))
```

回到 NtQueueApcThread()代码中，下一步就是根据 Apc->ApcStateIndex、Apc->Thread、和 Apc->ApcMode 把准备好的 KAPC 结构挂入相应的队列。根据 APC 请求的具体情况，有时候要插在队列的前头，一般则挂在队列的尾部。限于篇幅，我们在这里就不看 KeInsertQueueApc()的代码了；虽然这段代码中有一些特殊的处理，但都不是我们此刻所特

别关心的。

如果跟 Linux 的 Signal 机制作一类比，那么 NtQueueApcThread()相当于设置 Signal 处理函数(或中断服务程序)。在 Linux 里面，Signal 处理函数的执行需要受到某种触发，例如收到了别的线程或某个内核成分发来的信号；而执行 Signal 处理函数的时机则是在 CPU 从内核返回目标线程的用户空间程序的前夕。可是 Windows 的 APC 机制与此有所不同，一般来说，只要把 APC 请求挂入了队列，就不再需要触发，而只是等待执行的时机。对于用户 APC 请求，这时机同样也是在 CPU 从内核返回目标线程用户空间程序的前夕(对于内核 APC 则有所不同)。所以，在某种意义上，把一个 APC 请求挂入队列，就同时意味着受到了触发。对于系统调用 NtQueueApcThread()，我们可以理解为是把 APC 函数的设置与触发合在了一起。而对于异步的文件读写，则 APC 函数的设置与触发是分开的，内核先把 APC 函数记录在别的数据结构中，等实际的文件读写完成以后才把 APC 请求挂入队列，此时实际上只是触发其运行。不过那已是属于设备驱动框架的事了。所以，一旦把 APC 请求挂入队列，就只是等待执行时机的问题了。从这个意义上说，“异步过程调用”还真不失为贴切的称呼。

下面就来看执行 APC 的时机，那是在(系统调用、中断、或异常处理之后)从内核返回用户空间的途中。

_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

/* Raise IRQL to APC_LEVEL */
movl $1, %ecx
call @KfRaiseIrql@4

/* Save old IRQL */
pushl %eax

/* Deliver APCs */
sti
pushl %ebx
```

```
    pushl $0
    pushl $UserMode
    call _KiDeliverApc@12
    cli

    /* Return to old IRQL */
    popl %ecx
    call @KfLowerIrql@4
    . . . . . .
```

这是内核中处理系统调用返回和中断/异常返回的代码。在返回前夕，这里先通过%fs:KPCR_CURRENT_THREAD 取得指向当前线程的 ETHREAD(从而 KTHREAD)的指针，然后依次检查：

- 即将返回的是否用户空间。
- 是否有用户 APC 请求正在等待执行(KTHREAD_PENDING_USER_APC 是 ApcState.KernelApcPending 在 KTHREAD 数据结构中的位移)。

要是通过了这两项检查，执行针对当前线程的 APC 请求的时机就到了，于是就调用 KiDeliverApc()去“投递”APC 函数，这跟 Linux 中对 Signal 的处理又是十分相似的。注意在调用这个函数的前后还分别调用了 KfRaiseIrql()和 KfLowerIrql()，这是为了在执行 KiDeliverApc()期间让内核的“中断请求级别”处于 APC_LEVEL，执行完以后再予恢复。我们现在暂时不关心“中断请求级别”，以后会回到这个问题上。

前面讲过，KTHREAD 中有两个 KAPC_STATE 数据结构，一个是 ApcState，另一个是 SavedApcState，二者都有 APC 队列，但是要投递的只是 ApcState 中的队列。

注意在 call 指令前面压入堆栈的三个参数，特别是首先压入堆栈的%ebx，它指向(系统空间)堆栈上的“中断现场”、或称“框架”，即 CPU 进入本次中断或系统调用时各寄存器的值，这就是下面 KiDeliverApc()的调用参数 TrapFrame。

下面我们看 KiDeliverApc()的代码。

[KiDeliverApc()]

```
VOID
STDCALL
KiDeliverApc(KPROCESSOR_MODE DeliveryMode,
             PVOID Reserved,
             PKTRAP_FRAME TrapFrame)
{
  PKTHREAD Thread = KeGetCurrentThread();
  . . . . . .

  ASSERT_IRQL_EQUAL(APC_LEVEL);

  /* Lock the APC Queue and Raise IRQL to Synch */
  KeAcquireSpinLock(&Thread->ApcQueueLock, &OldIrql);
```

```
/* Clear APC Pending */
Thread->ApcState.KernelApcPending = FALSE;

/* Do the Kernel APCs first */
while (!IsListEmpty(&Thread->ApcState.ApcListHead[KernelMode])) {
   /* Get the next Entry */
   ApcListEntry = Thread->ApcState.ApcListHead[KernelMode].Flink;
   Apc = CONTAINING_RECORD(ApcListEntry, KAPC, ApcListEntry);

   /* Save Parameters so that it's safe to free the Object in Kernel Routine*/
   NormalRoutine     = Apc->NormalRoutine;
   KernelRoutine     = Apc->KernelRoutine;
   NormalContext     = Apc->NormalContext;
   SystemArgument1 = Apc->SystemArgument1;
   SystemArgument2 = Apc->SystemArgument2;

   /* Special APC */
   if (NormalRoutine == NULL) {
      /* Remove the APC from the list */
      Apc->Inserted = FALSE;
      RemoveEntryList(ApcListEntry);

      /* Go back to APC_LEVEL */
      KeReleaseSpinLock(&Thread->ApcQueueLock, OldIrql);

      /* Call the Special APC */
      DPRINT("Delivering a Special APC: %x\n", Apc);
      KernelRoutine(Apc, &NormalRoutine, &NormalContext,
                              &SystemArgument1, &SystemArgument2);

      /* Raise IRQL and Lock again */
      KeAcquireSpinLock(&Thread->ApcQueueLock, &OldIrql);

   } else {

      /* Normal Kernel APC */
      if (Thread->ApcState.KernelApcInProgress || Thread->KernelApcDisable)
      {
          /*
          * DeliveryMode must be KernelMode in this case, since one may not
          * return to umode while being inside a critical section or while
          * a regular kmode apc is running (the latter should be impossible btw).
          * -Gunnar
          */
```

```
            ASSERT(DeliveryMode == KernelMode);

            KeReleaseSpinLock(&Thread->ApcQueueLock, OldIrql);
            return;
        }

        /* Dequeue the APC */
        RemoveEntryList(ApcListEntry);
        Apc->Inserted = FALSE;

        /* Go back to APC_LEVEL */
        KeReleaseSpinLock(&Thread->ApcQueueLock, OldIrql);

        /* Call the Kernel APC */
        DPRINT("Delivering a Normal APC: %x\n", Apc);
        KernelRoutine(Apc,
                      &NormalRoutine,
                      &NormalContext,
                      &SystemArgument1,
                      &SystemArgument2);

        /* If There still is a Normal Routine, then we need to call this at PASSIVE_LEVEL */
        if (NormalRoutine != NULL) {

            /* At Passive Level, this APC can be prempted by a Special APC */
            Thread->ApcState.KernelApcInProgress = TRUE;
            KeLowerIrql(PASSIVE_LEVEL);

            /* Call and Raise IRQ back to APC_LEVEL */
            DPRINT("Calling the Normal Routine for a Normal APC: %x\n", Apc);
            NormalRoutine(&NormalContext, &SystemArgument1, &SystemArgument2);
            KeRaiseIrql(APC_LEVEL, &OldIrql);
        }

        /* Raise IRQL and Lock again */
        KeAcquireSpinLock(&Thread->ApcQueueLock, &OldIrql);
        Thread->ApcState.KernelApcInProgress = FALSE;
    }
}   //end while
```

参数 DeliveryMode 表示需要“投递”哪一种 APC，可以是 UserMode，也可以是 KernelMode。不过，KernelMode 确实表示只要求执行内核 APC，而 UserMode 却表示在执行内核 APC 之外再执行用户 APC。这里所谓“执行内核 APC”是执行内核 APC 队列中的

所有请求，而“执行用户 APC”却只是执行用户 APC 队列中的一项。

所以首先检查内核模式 APC 队列，只要非空就通过一个 while 循环处理其所有的 APC 请求。队列中的每一项(如果队列非空的话)、即每一个 APC 请求都是 KAPC 结构，结构中有三个函数指针，但是这里只涉及其中的两个。一个是 NormalRoutine，若为非 0 就是指向一个实质性的内核 APC 函数。另一个是 KernelRoutine，指向一个辅助性的内核 APC 函数，这个指针不会是 0，否则这个 KAPC 结构就不会在队列中了(注意 KernelRoutine 与内核模式 NormalRoutine 的区别)。NormalRoutine 为 0 是一种特殊的情况，在这种情况下 KernelRoutine 所指的内核函数无条件地得到调用。但是，如果 NormalRoutine 非 0，那么首先得到调用的是 KernelRoutine，而指针 NormalRoutine 的地址是作为参数传下去的。KernelRoutine 的执行有可能改变这个指针的值。这样，如果执行 KernelRoutine 以后 NormalRoutine 仍为非 0，那就说明需要加以执行，所以通过这个函数指针予以调用。不过，内核 APC 函数的执行是在 PASSIVE_LEVEL 级别上执行的，所以对 NormalRoutine 的调用前有 KeLowerIrql()、后有 KeRaiseIrql()，前者将 CPU 的运行级别调整为 PASSIVE_LEVEL，后者则将其恢复为 APC_LEVEL。

执行完内核 APC 队列中的所有请求以后，如果调用参数 DeliveryMode 为 UserMode 的话，就轮到用户 APC 了。我们继续往下看：

[KiDeliverApc()]

```
  /* Now we do the User APCs */
  if ((!IsListEmpty(&Thread->ApcState.ApcListHead[UserMode])) &&
        (DeliveryMode == UserMode) && (Thread->ApcState.UserApcPending == TRUE)) {

     /* It's not pending anymore */
     Thread->ApcState.UserApcPending = FALSE;

     /* Get the APC Object */
     ApcListEntry = Thread->ApcState.ApcListHead[UserMode].Flink;
     Apc = CONTAINING_RECORD(ApcListEntry, KAPC, ApcListEntry);

     /* Save Parameters so that it's safe to free the Object in Kernel Routine*/
     NormalRoutine     = Apc->NormalRoutine;
     KernelRoutine     = Apc->KernelRoutine;
     NormalContext     = Apc->NormalContext;
     SystemArgument1 = Apc->SystemArgument1;
     SystemArgument2 = Apc->SystemArgument2;

     /* Remove the APC from Queue, restore IRQL and call the APC */
     RemoveEntryList(ApcListEntry);
     Apc->Inserted = FALSE;

     KeReleaseSpinLock(&Thread->ApcQueueLock, OldIrql);
     DPRINT("Calling the Kernel Routine for for a User APC: %x\n", Apc);
     KernelRoutine(Apc,
```

```
                        &NormalRoutine,
                        &NormalContext,
                        &SystemArgument1,
                        &SystemArgument2);

      if (NormalRoutine == NULL) {
        /* Check if more User APCs are Pending */
        KeTestAlertThread(UserMode);
      }else {
        /* Set up the Trap Frame and prepare for Execution in NTDLL.DLL */
        DPRINT("Delivering a User APC: %x\n", Apc);
        KiInitializeUserApc(Reserved,
                            TrapFrame,
                            NormalRoutine,
                            NormalContext,
                            SystemArgument1,
                            SystemArgument2);
      }

    } else {

      /* Go back to APC_LEVEL */
      KeReleaseSpinLock(&Thread->ApcQueueLock, OldIrql);
    }
}
```

当然，执行用户 APC 是有条件的。首先自然是用户 APC 队列非空，同时调用参数 DeliveryMode 必须是 UserMode；并且 ApcState 中的 UserApcPending 为 TRUE，表示队列中的请求确实是要求尽快加以执行的。

读者也许已经注意到，比之内核 APC 队列，对用户 APC 队列的处理有个显著的不同，那就是对用户 APC 队列并不是通过一个 while 循环处理队列中的所有请求，而是每次进入 KiDeliverApc()只处理用户 APC 队列中的第一个请求。同样，这里也是只涉及两个函数指针，即 NormalRoutine 和 KernelRoutine，也是先执行 KernelRoutine，并且 KernelRoutine 可以对指针 NormalRoutine 作出修正。但是再往下就不同了。

首先，如果执行完 KernelRoutine(所指的函数)以后指针 NormalRoutine 为 0，这里要执行 KeTestAlertThread()。这又是跟设备驱动有关的事(Windows 术语中的 Alert 相当于 Linux 术语中的“唤醒”)，我们在这里暂不关心。

反之，如果指针 NormalRoutine 仍为非 0，那么这里执行的是 KiInitializeUserApc()，而不是直接调用 NormalRoutine 所指的函数，因为 NormalRoutine 所指的函数是在用户空间，要等 CPU 回到用户空间才能执行，这里只是为其作好安排和准备。

[KiDeliverApc() > KiInitializeUserApc()]

```
VOID
```

```
STDCALL
KiInitializeUserApc(IN PVOID Reserved,
                    IN PKTRAP_FRAME TrapFrame,
                    IN PKNORMAL_ROUTINE NormalRoutine,
                    IN PVOID NormalContext,
                    IN PVOID SystemArgument1,
                    IN PVOID SystemArgument2)
{
    PCONTEXT Context;
    PULONG Esp;

    . . . . . .
    /*
     * Save the thread's current context (in other words the registers
     * that will be restored when it returns to user mode) so the
     * APC dispatcher can restore them later
     */
    Context = (PCONTEXT)(((PUCHAR)TrapFrame->Esp) - sizeof(CONTEXT));
    RtlZeroMemory(Context, sizeof(CONTEXT));
    Context->ContextFlags = CONTEXT_FULL;
    Context->SegGs = TrapFrame->Gs;
    Context->SegFs = TrapFrame->Fs;
    Context->SegEs = TrapFrame->Es;
    Context->SegDs = TrapFrame->Ds;
    Context->Edi = TrapFrame->Edi;
    Context->Esi = TrapFrame->Esi;
    Context->Ebx = TrapFrame->Ebx;
    Context->Edx = TrapFrame->Edx;
    Context->Ecx = TrapFrame->Ecx;
    Context->Eax = TrapFrame->Eax;
    Context->Ebp = TrapFrame->Ebp;
    Context->Eip = TrapFrame->Eip;
    Context->SegCs = TrapFrame->Cs;
    Context->EFlags = TrapFrame->Eflags;
    Context->Esp = TrapFrame->Esp;
    Context->SegSs = TrapFrame->Ss;

    /*
     * Setup the trap frame so the thread will start executing at the
     * APC Dispatcher when it returns to user-mode
     */
    Esp = (PULONG)(((PUCHAR)TrapFrame->Esp) -
                                (sizeof(CONTEXT) + (6 * sizeof(ULONG))));
    Esp[0] = 0xdeadbeef;
```

```
    Esp[1] = (ULONG)NormalRoutine;
    Esp[2] = (ULONG)NormalContext;
    Esp[3] = (ULONG)SystemArgument1;
    Esp[4] = (ULONG)SystemArgument2;
    Esp[5] = (ULONG)Context;
    TrapFrame->Eip = (ULONG)LdrpGetSystemDllApcDispatcher();
    TrapFrame->Esp = (ULONG)Esp;
}
```

这个函数的名字取得不好，很容易让人把它跟前面的 KeInitializeApc()相连系，实际上却完全是两码事。参数 TrapFrame 是由 KiDeliverApc()传下来的一个指针，指向用户空间堆栈上的“中断现场”。这里要做的事情就是在原有现场的基础上“注水”，伪造出一个新的现场，使得 CPU 返回用户空间时误认为中断(或系统调用)发生于进入 APC 函数的前夕，从而转向 APC 函数。

怎么伪造呢？首先使用户空间的堆栈指针 Esp 下移一个 CONTEXT 数据结构的大小，外加 6 个 32 位整数的位置(注意堆栈是由上向下伸展的)。换言之就是在用户空间堆栈上扩充出一个 CONTEXT 数据结构和 6 个 32 位整数。注意，TrapFrame 是在系统空间堆栈上，而 TrapFrame->Esp 的值是用户空间的堆栈指针，所指向的是用户空间堆栈。所以这里扩充的是用户空间堆栈。这样，原先的用户堆栈下方是 CONTEXT 数据结构 Context，再往下就是那 6 个 32 位整数。然后把 TrapFrame 的内容保存在这个 CONTEXT 数据结构中，并设置好 6 个 32 位整数，那是要作为调用参数传递的。接着就把保存在 TrapFrame 中的 Eip 映像改成指向用户空间的一个特殊函数，具体的地址通过 LdrpGetSystemDllApcDispatcher()获取。这样，当 CPU 返回到用户空间时，就会从这个特殊函数“继续”执行。当然，也要调整 TrapFrame 中的用户空间堆栈指针 Esp。

LdrpGetSystemDllApcDispatcher()只是返回一个(内核)全局量 SystemDllApcDispatcher 的值，这个值是个函数指针，指向 ntdll.dll 中的一个函数，是在映射 ntdll.dll 映像时设置好的。

```
PVOID LdrpGetSystemDllApcDispatcher(VOID)
{
   return(SystemDllApcDispatcher);
}
```

与全局变量 SystemDllApcDispatcher 相似的函数指针有：

- SystemDllEntryPoint，指向 LdrInitializeThunk()。
- SystemDllApcDispatcher，指向 KiUserApcDispatcher()。
- SystemDllExceptionDispatcher，指向 KiUserExceptionDispatcher()。
- SystemDllCallbackDispatcher，指向 KiUserCallbackDispatcher()。
- SystemDllRaiseExceptionDispatcher，指向 KiRaiseUserExceptionDispatcher()。

这些指针都是在 LdrpMapSystemDll()中得到设置的。给定一个函数名的字符串，就可以通过一个函数 LdrGetProcedureAddress()从(已经映射的)DLL 映像中获取这个函数的地址(如果这个函数被引出的话)。

于是，CPU 从 KiDeliverApc()回到_KiServiceExit 以后会继续完成其返回用户空间的行程，只是一到用户空间就栽进了圈套，那就是 KiUserApcDispatcher()，而不是回到原先的断点上。关于原先断点的现场信息保存在用户空间堆栈上、并形成一个 CONTEXT 数据结构，

但是“深埋”在 6 个 32 位整数的后面。而这 6 个 32 位整数的作用则为：

- Esp[0]的值为 0xdeadbeef，用来模拟 KiUserApcDispatcher()的返回地址。当然，这个地址是无效的，所以 KiUserApcDispatcher()实际上是不会返回的。
- Esp[1]的值为 NormalRoutine，在我们这个情景中指向“门户”函数 IntCallUserApc()。
- Esp[2]的值为 NormalContext，在我们这个情景中是指向实际 APC 函数的指针。
- 余类推。其中 Esp[5]指向(用户)堆栈上的 CONTEXT 数据结构。

总之，用户堆栈上的这 6 个 32 位整数模拟了一次 CPU 在进入 KiUserApcDispatcher()还没有来得及执行其第一条指令之前就发生了中断的假象，使得 CPU 在结束了 KiDeliverApc()的执行、回到_KiServiceExit 中继续前行、并最终回到用户空间时就进入 KiUserApcDispatcher()执行其第一条指令。

另一方面，对于该线程原来的上下文而言，则又好像是刚回到用户空间就发生了中断，而 KiUserApcDispatcher()则相当于中断相应程序。

```
VOID STDCALL
KiUserApcDispatcher(PIO_APC_ROUTINE ApcRoutine, PVOID ApcContext,
                    PIO_STATUS_BLOCK Iosb, ULONG Reserved, PCONTEXT Context)
{
    /* Call the APC */
    ApcRoutine(ApcContext, Iosb, Reserved);
    /* Switch back to the interrupted context */
    NtContinue(Context, 1);
}
```

这里的第一个参数 ApcRoutine 指向 IntCallUserApc()，第二个参数 ApcContext 指向真正的(目标)APC 函数。

[KiUserApcDispatcher() > IntCallUserApc()]

```
static void CALLBACK
IntCallUserApc(PVOID Function, PVOID dwData, PVOID Argument3)
{
    PAPCFUNC pfnAPC = (PAPCFUNC)Function;
    pfnAPC((ULONG_PTR)dwData);
}
```

可见，IntCallUserApc()其实并无必要，在 KiUserApcDispatcher()中直接调用目标 APC 函数也无不可，这样做只是为将来可能的修改扩充提供一些方便和灵活性。从 IntCallUserApc()回到 KiUserApcDispatcher()，下面紧接着是系统调用 NtContinue()。

KiUserApcDispatcher()是不返回的。它之所以不返回，是因为对 NtContinue()的调用不返回。正如代码中的注释所述，NtContinue()的作用是切换回被中断了的上下文，不过其实还不止于此，下面读者就会看到它还起着循环执行整个用户 APC 请求队列的作用。

[KiUserApcDispatcher() > NtContinue()]

```
NTSTATUS STDCALL
NtContinue (IN PCONTEXT Context,    IN BOOLEAN TestAlert)
{
    PKTHREAD Thread = KeGetCurrentThread();
    PKTRAP_FRAME TrapFrame = Thread->TrapFrame;
    PKTRAP_FRAME PrevTrapFrame = (PKTRAP_FRAME)TrapFrame->Edx;
    PFX_SAVE_AREA FxSaveArea;
    KIRQL oldIrql;

    DPRINT("NtContinue: Context: Eip=0x%x, Esp=0x%x\n", Context->Eip, Context->Esp );
    PULONG Frame = 0;
    __asm__("mov %%ebp, %%ebx" : "=b" (Frame) : );
    . . . . . .

    /*
    * Copy the supplied context over the register information that was saved
    * on entry to kernel mode, it will then be restored on exit
    * FIXME: Validate the context
    */
    KeContextToTrapFrame ( Context, TrapFrame );

    /* Put the floating point context into the thread's FX_SAVE_AREA
    * and make sure it is reloaded when needed.
    */
    FxSaveArea = (PFX_SAVE_AREA)((ULONG_PTR)Thread->InitialStack –
                                                    sizeof(FX_SAVE_AREA));
    if (KiContextToFxSaveArea(FxSaveArea, Context))
    {
        Thread->NpxState = NPX_STATE_VALID;
        KeRaiseIrql(DISPATCH_LEVEL, &oldIrql);
        if (KeGetCurrentPrcb()->NpxThread == Thread)
        {
            KeGetCurrentPrcb()->NpxThread = NULL;
            Ke386SetCr0(Ke386GetCr0() | X86_CR0_TS);
        }
        else
        {
            ASSERT((Ke386GetCr0() & X86_CR0_TS) == X86_CR0_TS);
        }
        KeLowerIrql(oldIrql);
    }

    /* Restore the user context */
```

```
    Thread->TrapFrame = PrevTrapFrame;
    __asm__("mov %%ebx, %%esp;\n" "jmp _KiServiceExit": : "b" (TrapFrame));

    return STATUS_SUCCESS; /* this doesn't actually happen */
}
```

注意从 KiUserApcDispatcher()到 NtContinue()并不是普通的函数调用，而是系统调用，这中间经历了空间的切换，也从用户空间堆栈切换到了系统空间堆栈。CPU 进入系统调用空间后，在_KiSystemServicex 下面的代码中把指向中断现场的框架指针保存在当前线程的 KTHREAD 数据结构的 TrapFrame 字段中。这样，很容易就可以找到系统空间堆栈上的调用框架。当然，现在的框架是因为系统调用而产生的框架；而要想回到当初、即在执行用户空间 APC 函数之前的断点，就得先恢复当初的框架。那么当初的框架在哪里呢？它保存在用户空间的堆栈上，就是前面 KiInitializeUserApc()保存的 CONTEXT 数据结构中。所以，这里通过 KeContextToTrapFrame()把当初保存的信息拷贝回来，从而恢复了当初的框架。

下面的 KiContextToFxSaveArea()等语句与浮点处理器有关，我们在这里并不关心。

最后，汇编指令“jmp _KiServiceExit”使 CPU 跳转到了返回用户空间途中的_KiServiceExit 处(见前面的代码)。在这里，CPU 又会检查 APC 请求队列中是否有 APC 请求等着要执行，如果有的话又会进入 KiDeliverApc()。前面讲过，每次进入 KiDeliverApc()只会执行一个用户 APC 请求，所以如果用户 APC 队列的长度大于 1 的话就得循环着多次走过上述的路线，即：

1. 从系统调用、中断、或异常返回途径_KiServiceExit，如果 APC 队列中有等待执行的 APC 请求，就调用 KiDeliverApc()。
2. KiDeliverApc()，从用户 APC 队列中摘下一个 APC 请求。
3. 在 KiInitializeUserApc()中保存当前框架，并伪造新的框架。
4. 回到用户空间。
5. 在 KiUserApcDispatcher()中调用目标 APC 函数。
6. 通过系统调用 NtContinue()进入系统空间。
7. 在 NtContinue()中恢复当初保存的框架。
8. 从 NtContinue()返回、途径_KiServiceExit 时，如果 APC 队列中还有等待执行的 APC 请求，就调用 KiDeliverApc()。于是转回上面的第二步。

这个过程一直要循环到 APC 队列中不再有需要执行的请求。注意这里每一次循环中保存和恢复的都是同一个框架，就是原始的、开始处理 APC 队列之前的那个框架，代表着原始的用户空间程序断点。一旦 APC 队列中不再有等待执行的 APC 请求，在_KiServiceExit 下面就不再调用 KiDeliverApc()，于是就直接返回用户空间，这次是返回到原始的程序断点了。所以，系统调用 neContinue()的作用不仅仅是切换回到被中断了的上下文，还包括执行用户 APC 队列中的下一个 APC 请求。

对于 KiUserApcDispatcher()而言，它对 NtContinue()的调用是不返回的。因为在 NtContinue()中 CPU 不是“返回”到对于 KiUserApcDispatcher()的另一次调用、从而对另一个 APC 函数的调用；就是返回到原始的用户空间程序断点，这个断点既可能是因为中断或异常而形成的，也可能是因为系统调用而形成的。

理解了常规的 APC 请求和执行机制，我们不妨再看看启动执行 PE 目标映像时函数的动态连接。以前讲过，PE 格式 EXE 映像与(除 ntdll.dll 外的)DLL 的动态连接、包括这些 DLL 的装入，是由 ntdll.dll 中的一个函数 LdrInitializeThunk()作为 APC 函数执行而完成的，所以

这也是对 APC 机制的一种变通使用。

要启动一个 EXE 映像运行时，首先要创建进程，再把目标 EXE 映像和 ntdll.dll 的映像都映射到新进程的用户空间，然后通过系统调用 NtCreateThread()创建这个进程的第一个线程、或称“主线程”。而 LdrInitializeThunk()作为 APC 函数的执行，就是在 NtCreateThread()中安排好的。

```
NtCreateThread(OUT PHANDLE ThreadHandle, IN ACCESS_MASK DesiredAccess,
                    IN POBJECT_ATTRIBUTES ObjectAttributes    OPTIONAL,
                    IN HANDLE ProcessHandle, OUT PCLIENT_ID ClientId,
                    IN PCONTEXT ThreadContext, IN PINITIAL_TEB InitialTeb,
                    IN BOOLEAN CreateSuspended)
{
  HANDLE hThread;

  . . . . . .
  . . . . . .
  /*
   * Queue an APC to the thread that will execute the ntdll startup
   * routine.
   */
  LdrInitApc = ExAllocatePool(NonPagedPool, sizeof(KAPC));
  KeInitializeApc(LdrInitApc, &Thread->Tcb, OriginalApcEnvironment,
                                    LdrInitApcKernelRoutine,
                                    LdrInitApcRundownRoutine,
                                    LdrpGetSystemDllEntryPoint(), UserMode, NULL);
  KeInsertQueueApc(LdrInitApc, NULL, NULL, IO_NO_INCREMENT);

  /*
   * The thread is non-alertable, so the APC we added did not set UserApcPending to TRUE.
   * We must do this manually. Do NOT attempt to set the Thread to Alertable before the call,
   * doing so is a blatant and erronous hack.
   */
  Thread->Tcb.ApcState.UserApcPending = TRUE;
  Thread->Tcb.Alerted[KernelMode] = TRUE;
  . . . . . .
  . . . . . .
}
```

NeCreateThread()要做的事当然很多，但是其中很重要的一项就是安排好 APC 函数的执行。这里的 KeInitializeApc()和 KeInsertQueueApc 读者都已经熟悉了，所以我们只关心调用参数中的三个函数指针，特别是其中的 KernelRoutine 和 NormalRoutine。前者十分简单：

```
VOID STDCALL
LdrInitApcKernelRoutine(PKAPC Apc, PKNORMAL_ROUTINE* NormalRoutine,
```

```
                PVOID* NormalContext, PVOID* SystemArgument1, PVOID* SystemArgument2)
{
  ExFreePool(Apc);
}
```

而 NormalRoutine，这里是通过 LdrpGetSystemDllEntryPoint()获取的，它只是返回全局量 SystemDllEntryPoint 的值：

```
PVOID LdrpGetSystemDllEntryPoint(VOID)
{
   return(SystemDllEntryPoint);
}
```

前面已经讲到，全局量 SystemDllEntryPoint 是在 LdrpMapSystemDll()时得到设置的，指向已经映射到用户空间的 ntdll.dll 映像中的 LdrInitializeThunk()。注意这 APC 请求是挂在新线程的队列中，而不是当前进程的队列中。事实上，新线程和当前进程处于不同的进程，因而不在同一个用户空间中。还要注意，这里的 NormalRoutine 直接就是 LdrInitializeThunk()，而不像前面通过 QueueUserAPC()发出的 APC 请求那样中间还有一层 IntCallUserApc()。至于 KiUserApcDispatcher()，那是由 KeInitializeApc()强制加上的，正是这个函数保证了对 NtContinue()的调用。

此后的流程本来无需细说了，但是由于情景的特殊性还是需要加一些简要的说明。由 NtCreateProcess()创建的进程并非一个可以调度运行的实体，而 NtCreateThread()创建的线程却是。所以，在 NtCreateProcess()返回的前夕，系统中已经多了一个线程。这个新增线程的“框架”是伪造的，目的在于让这个线程一开始在用户空间运行就进入预定的程序入口。从 NtCreateProcess()返回是回到当前线程、而不是新增线程，而刚才的 APC 请求是挂在新增线程的队列中，所以在从 NtCreateThread()返回的途中不会去执行这个 APC 请求。可是，当新增线程受调度运行时，首先就是按伪造的框架和堆栈模拟一个从系统调用返回的过程，所以也要途径_KiServiceExit。这时候，这个 APC 请求就要得到执行了(由 KiUserApcDispatcher()调用 LdrInitializeThunk())。然后，在用户空间执行完 APC 函数 LdrInitializeThunk()以后，同样也是通过 NtContinue()回到内核中，然后又按原先的伪造框架“返回”到用户空间，这才真正开始了新线程在用户空间的执行。

最后，我们不妨比较一下 APC 机制和 Unix/Linux 的 Signal 机制。

Unix/Linux 的 Signal 机制基本上是对硬件中断机制的软件模拟，具体表现在以下几个方面：

1） 现代的硬件中断机制一般都是“向量中断”机制，而 Signal 机制中的 Signal 序号(例如 SIG_KILL)就是对中断向量序号的模拟。

2） 作为操作系统对硬件中断机制的支持，一般都会提供“设置中断向量”一类的内核函数，使特定序号的中断向量指向某个中断服务程序。而系统调用 signal()就相当于是这一类的函数。只不过前者在内核中、一般只是供其它内核函数调用，而后者是系统调用、供用户空间的程序调用。

3） 在硬件中断机制中，“中断向量”的设置只是为某类异步事件、及中断的发生做好了准备，但是并不意味着某个特定时间的发生。如果一直没有中断请求，那么所设置的中断向量就一直得不到执行，而中断的发生只是触发了中断服务程

序的执行。在 Signal 机制中，向某个进程发出“信号”、即 Signal、就相当于中断请求。

相比之下，APC 机制就不能说是对于硬件中断机制的模拟了。首先，通过 NtQueueApcThread()设置一个 APC 函数跟通过 signal()设置一个“中断向量”有所不同。将一个 APC 函数挂入 APC 队列中时，对于这个函数的得到执行、以及大约在什么时候得到执行，实际上是预知的，只是这得到执行的条件要过一回儿才会成熟。而“中断”则不同，中断向量的设置只是说如果发生某种中断则如何如何，但是对于其究竟是否会发生、何时发生则常常是无法预测的。所以，从这个意义上说，APC 函数只是一种推迟执行、异步执行的函数调用，因此称之为“异步过程调用”确实更为贴切。

还有，signal 机制的 signal()所设置的“中断服务程序”都是用户空间的程序，而 APC 机制中挂入 APC 队列的函数却可以是内核函数。

但是，尽管如此，它们的(某些方面的)实质还是一样的。“中断”本来就是一种异步执行的机制。再说，(用户)APC 与 Signal 的执行流程几乎完全一样，都是在从内核返回用户空间的前夕检查是否有这样的函数需要加以执行，如果是就临时修改堆栈，偏离原来的执行路线，使得返回用户空间后进入 APC 函数，并且在执行完了这个函数以后仍进入内核，然后恢复原来的堆栈，再次返回用户空间原来的断点。这样，对于原来的流程而言，就相当于受到了中断。