漫谈兼容内核之十六:
Windows 的进程间通信

毛德操

对于任何一个现代的操作系统，进程间通信都是其系统结构的一个重要组成部分。而说到 Windows 的进程(线程)间通信，那就要看是在什么意义上说了。因为正如“Windows 的跨进程操作”那篇漫谈中所述，在 Windows 上一个进程甚至可以“打开”另一个进程，并在对方的用户空间分配内存、再把程序或数据拷贝过去，最后还可以在目标进程中创建一个线程、让它为所欲为。显然，这已经不只是进程间的“通信”，而是进程间“操纵”了。但是这毕竟属于另类，我们在这里要谈论的是“正规”的进程间通信。

不管是两个甚么样的实体，凡是要通信就得满足一个必要条件，那就是存在双方都可以访问的介质。顾名思义，进程间通信就是在不同进程之间传播或交换信息，那么不同进程之间存在着什么双方都可以访问的介质呢？进程的用户空间是互相独立的，一般而言是不能互相访问的，唯一的例外是共享内存区。但是，系统空间却是“公共场所”，所以内核显然可以提供这样的条件。除此以外，那就是双方都可以访问的外设了。在这个意义上，两个进程当然也可以通过磁盘上的普通文件交换信息，或者通过“注册表”或其它数据库中的某些表项和记录交换信息。广义上这也是进程间通信的手段，但是一般都不把这算作“进程间通信”。因为那些通信手段的效率太低了，而人们对进程间通信的要求是要有一定的实时性。

但是，对于实际的应用而言，光有信息传播的实时性往往还不够。不妨以共享内存区(Section)为例来说明这个问题。共享内存区显然可以用作进程间通信的手段，两个进程把同一组物理内存页面分别映射到自己的用户空间，然后一个进程往里面写，另一个进程就可以读到所写入的内容。从信息传播的角度看，这个过程是“即时”的，有着很高的实时性，但是读取者怎么知道写入者已经写入了一些数据呢？要是共享内存区的物理页面能产生中断请求就好了，可是它不能。让读取者轮询、或者定时轮询、那当然也可以，但是效率就降下来了。所以，这里还需要有通信双方行为上的协调、或称进程间的“同步”。注意所谓“同步”并不是说双方应该同时读或同时写，而是让双方的行为得以有序、紧凑地进行。

综上所述，一般所说的“进程间通信”其实是狭义的、带限制条件的。总的来说，对于进程间通信有三方面的要求：

- 具有不同进程之间传播或交换信息的手段
- 进程间传播或交换信息的手段应具有一定程度的实时性
- 具有进程间的协调(同步)机制。

此外，“进程间通信”一般是指同一台机器上的进程间通信。通过网络或通信链路进行的跨主机的通信一般不归入进程间通信的范畴，虽然这种通信通常也确实是发生于进程之间。不过网络通信往往也可以作用于本机的不同进程之间，这里并没有明确的界线。这样一来范围就广了，所以本文在介绍 Windows 的进程间通信时以其内核是否专门为具体的机制提供了系统调用为准。这样，例如用于网络通信的 Winsock 机制是作为设备驱动实现的，内核并没有为此提供专门的系统调用，所以本文就不把它算作进程间通信。

先看上面三方面要求的第一项，即同一机器上的不同进程之间传播或交换信息的手段，这无非就是几种可能：

- 通过用户空间的共享内存区。
- 通过内核中的变量、数据结构、或缓冲区。
- 通过外设的存储效应。但是一般所讲操作系统内核的“进程间通信”机制都把这排

除在外。

由于通过外设进行的进程间通信一般而言实时性不是很好，所以考虑到上述第二方面的要求就把它给排除掉了。

再看进程间的同步机制。如前所述，进程间同步的目的是要让通信的双方(或多方) 行为得以有序、紧凑地进行。所以本质上就是双方(或多方)之间的等待(睡眠)/唤醒机制，这就是为什么要在上一篇漫谈中先介绍等待/唤醒机制的原因。注意这里的“等待”意味着主动进入睡眠，一般而言，所谓“进程间同步”就是建立在(主动)睡眠/唤醒机制基础上的同步。不主动进入睡眠的同步也是有的，例如“空转锁(Spinlock)”就是，但是那样太浪费 CPU 资源了。再说，在单 CPU 的系统中，如果是在调度禁区中使用 Spinlock，还会引起死锁。所以，一般不把 Spinlock 算作进程间同步手段。在操作系统理论中，“信号量(Semaphore)”是基本的进程间同步机制，别的大都是在此基础上派生出来的。

另一方面，进程间同步的实现本身就需要有进程间的信息传递作为基础，例如“唤醒”这个动作就蕴含着信息的传递。所以，进程间同步其实也是进程间通信，只不过是信息量比较小、或者很小的进程间通信。换言之，带有进程间同步的进程间通信，实际上就是先以少量信息的传递使双方的行为得到协调，再在此基础上交换比较大量的信息。如果需要传递的信息量本来就很小，那么这里的第二步也就不需要了。所以，进程间同步就是(特殊的)进程间通信。

注意这里所说的进程间通信实际上是线程间通信，特别是分属于不同进程的线程之间的通信。因为在 Windows 中线程才是运行的实体，而进程不是。但是上述的原理同样适用于同一进程内部的线程间通信。属于同一进程的线程共享同一个用户空间，所以整个用户空间都成了共享内存区。如果两个线程都访问同一个变量或数据结构，那么实际上就构成了线程间通信(或许是在不知不觉间)。这里仍有双方如何同步的问题，但是既然是共享用户空间，就有可能在用户空间构筑这样的同步机制。所以，一般而言，进程间通信需要内核的支持，而同一进程中的线程间通信则也可以在用户空间(例如在 DLL 中)实现。在下面的叙述中，“进程间通信”和“线程间通信”这两个词常常是混用的，读者应注意领会。

Windows 内核所支持的进程间通信手段有：

- 共享内存区(Section)。
- 信号量(Semaphore)。
- 互斥门(Mutant)。
- 事件(Event)。
- 特殊文件“命名管道(Named Pipe)”和“信箱(Mail Slot)”。

此外，本地过程调用、即 LPC，虽然并非以进程间通信机制的面貌出现，实际上却是建立在进程间通信的基础上，并且本身就是一种独特的进程间通信机制。还有，Windows 的 Win32K 模块提供了一种线程之间的报文传递机制，一般用作“窗口”之间的通信手段，显然也应算作进程间通信，只不过这是由 Win32K 的“扩充系统调用”支持的，而并非由基本的系统调用所支持。所以，还应增加以下两项。

- 端口(Port)和本地过程调用(LPC)。
- 报文(Message)。

注：“Undocumented Windows 2000 Secrets”书中还列出了另一组用于“通道(Channel)”的系统调用，例如 NtOpenChannel()、NtListenChannel()、NtSendWaitReplyChannel()等等，从这些系统调用函数名看来，这应该也是一种进程间通信机制，但是“Windows NT/2000 Native API Reference”书中说这些函数均未实现，调用后只是返回出错代码“STATUS_NOT_IMPLEMENTED”，“Microsoft Windows Internals”书中则并未提及。在

ReactOS 的代码中也未见实现。

下面逐一作些介绍。

1. 共享内存区(Section)

如前所述，共享内存区是可以用于进程间通信的。但是，离开进程间同步机制，它的效率就不会高，所以共享内存区单独使用并不是一种有效的进程间通信机制。

使用的方法是：先以双方约定的名字创建一个 Section 对象，各自加以打开，再各自将其映射到自己的用户空间，然后就可以通过常规的内存读写(例如通过指针)进行通信了。

要通过共享内存区进行通信时，首先要通过 NtCreateSection()创建一个共享内存区对象。从程序的结构看，几乎所有对象的创建、即所有形似 NtCreateXYZ()的函数的代码都是基本相同的，所以下面列出 NtCreateSection()的代码，以后对类似的代码就不再列出了。

```
NTSTATUS STDCALL
NtCreateSection (OUT PHANDLE SectionHandle,
                 IN ACCESS_MASK DesiredAccess,
                 IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
                 IN PLARGE_INTEGER MaximumSize OPTIONAL,
                 IN ULONG SectionPageProtection OPTIONAL,
                 IN ULONG AllocationAttributes,
                 IN HANDLE FileHandle OPTIONAL)
{
   . . . . . .
   PreviousMode = ExGetPreviousMode();

   if(MaximumSize != NULL && PreviousMode != KernelMode)
   {
      _SEH_TRY
      {
         ProbeForRead(MaximumSize,
                      sizeof(LARGE_INTEGER),
                      sizeof(ULONG));
         /* make a copy on the stack */
         SafeMaximumSize = *MaximumSize;
         MaximumSize = &SafeMaximumSize;
      }
      _SEH_HANDLE
      {
         Status = _SEH_GetExceptionCode();
      }
      _SEH_END;

      if(!NT_SUCCESS(Status))
      {
         return Status;
```

```
        }
    }

    /*
     * Check the protection
     */
    if ((SectionPageProtection & PAGE_FLAGS_VALID_FROM_USER_MODE) !=
            SectionPageProtection)
    {
        return(STATUS_INVALID_PAGE_PROTECTION);
    }

    Status = MmCreateSection(&SectionObject, DesiredAccess, ObjectAttributes,
                                        MaximumSize, SectionPageProtection,
                                        AllocationAttributes, FileHandle, NULL);
    if (NT_SUCCESS(Status))
    {
        Status = ObInsertObject ((PVOID)SectionObject, NULL,
                                            DesiredAccess, 0, NULL, SectionHandle);
        ObDereferenceObject(SectionObject);
    }

    return Status;
}
```

虽然名曰“Create”，实际上却是“创建并打开”，参数 SectionHandle 就是用来返回打开后的 Handle。参数 DesiredAccess 说明所创建的对象允许什么样的访问，例如读、写等等。ObjectAttributes 则说明对象的名称，打开以后是否允许遗传，以及与对象保护有关的特性、例如访问权限等等。这几个参数对于任何对象的创建都一样，而其余几个参数就是专为共享内存区的特殊需要而设的了。其中 MaximumSize 当然是共享内存区大小的上限，而 SectionPageProtection 与页面的保护有关。AllocationAttributes 通过一些标志位说明共享区的性质和用途，例如可执行映像或数据文件。最后，共享缓冲区往往都是以磁盘文件作为后盾的，为此需要先创建或打开相应的文件，然后把 FileHandle 作为参数传给 NtCreateSection()。不过用于进程间通信的共享内存区是空白页面，其内容并非来自某个文件，所以 FileHandle 为 NULL。

显然，创建共享内存区的实质性操作是由 MmCreateSection()完成的。对于其它的对象，往往也都有类似的函数。我们看一下 MmCreateSection()的代码：

[NtCreateSection() > MmCreateSection()]

```
NTSTATUS STDCALL
MmCreateSection (OUT PSECTION_OBJECT     * SectionObject,
                 IN ACCESS_MASK           DesiredAccess,
                 IN POBJECT_ATTRIBUTES    ObjectAttributes     OPTIONAL,
```

```
                    IN PLARGE_INTEGER          MaximumSize,
                    IN ULONG                    SectionPageProtection,
                    IN ULONG                    AllocationAttributes,
                    IN HANDLE                   FileHandle      OPTIONAL,
                    IN PFILE_OBJECT             File      OPTIONAL)
{
    if (AllocationAttributes & SEC_IMAGE)
    {
        return(MmCreateImageSection(SectionObject, DesiredAccess, ObjectAttributes,
                                    MaximumSize, SectionPageProtection,
                                    AllocationAttributes, FileHandle));
    }

    if (FileHandle != NULL)
    {
        return(MmCreateDataFileSection(SectionObject, DesiredAccess, ObjectAttributes,
                                       MaximumSize, SectionPageProtection,
                                       AllocationAttributes, FileHandle));
    }

    return(MmCreatePageFileSection(SectionObject, DesiredAccess, ObjectAttributes,
                                   MaximumSize, SectionPageProtection, AllocationAttributes));
}
```

参数 AllocationAttributes 中的 SEC_IMAGE 标志位为 1 表示共享内存区的内容是可执行映像(因而必需符合可执行映像的头部结构)。而 FileHandle 为 1 表示共享内存区的内容来自文件，既然不是可执行映像那就是数据文件了；否则就并非来自文件，那就是用于进程间通信的空白页面了。最后一个参数 File 的用途不明，似乎并无必要。我们现在关心的是空白页面的共享内存区，具体的对象是由 MmCreatePageFileSection()创建的，我们就不往下看了。注意这里还不涉及共享内存区的地址，因为尚未映射。

参与通信的双方通过同一个共享内存区进行通信，所以不能各建各的共享内存区，至少有一方需要打开已经创建的共享内存区，这是通过 NtOpenSection()完成的：

```
NTSTATUS STDCALL
NtOpenSection(PHANDLE      SectionHandle,
              ACCESS_MASK    DesiredAccess,
              POBJECT_ATTRIBUTES ObjectAttributes)
{
    HANDLE hSection;
    KPROCESSOR_MODE PreviousMode;
    NTSTATUS Status = STATUS_SUCCESS;

    PreviousMode = ExGetPreviousMode();
```

```
    if(PreviousMode != KernelMode)
    {
      _SEH_TRY. . . . . _SEH_END;
    }
    Status = ObOpenObjectByName(ObjectAttributes, MmSectionObjectType,
                                NULL, PreviousMode, DesiredAccess,
                                NULL, &hSection);
    if(NT_SUCCESS(Status))
    {
      _SEH_TRY
      {
        *SectionHandle = hSection;
      }
      _SEH_HANDLE
      {
        Status = _SEH_GetExceptionCode();
      }
      _SEH_END;
    }

    return(Status);
}
```

这里实质性的操作是 ObOpenObjectByName()，读者想必已经熟悉。

双方都打开了一个共享内存区对象以后，就可以各自通过 NtMapViewOfSection()将其映射到自己的用户空间。

```
NTSTATUS STDCALL
NtMapViewOfSection(IN HANDLE SectionHandle,
                   IN HANDLE ProcessHandle,
                   IN OUT PVOID* BaseAddress    OPTIONAL,
                   IN ULONG ZeroBits    OPTIONAL,
                   IN ULONG CommitSize,
                   IN OUT PLARGE_INTEGER SectionOffset    OPTIONAL,
                   IN OUT PULONG ViewSize,
                   IN SECTION_INHERIT InheritDisposition,
                   IN ULONG AllocationType    OPTIONAL,
                   IN ULONG Protect)
{
    . . . . . .
    PreviousMode = ExGetPreviousMode();
    if(PreviousMode != KernelMode)
    {
      SafeBaseAddress = NULL;
```

```
        SafeSectionOffset.QuadPart = 0;
        SafeViewSize = 0;
        _SEH_TRY. . . . . _SEH_END;
        . . . . . .
    }
    else
    {
        SafeBaseAddress = (BaseAddress != NULL ? *BaseAddress : NULL);
        SafeSectionOffset.QuadPart = (SectionOffset != NULL ? SectionOffset->QuadPart : 0);
        SafeViewSize = (ViewSize != NULL ? *ViewSize : 0);
    }

    Status = ObReferenceObjectByHandle(ProcessHandle,
                                       PROCESS_VM_OPERATION,
                                       PsProcessType,
                                       PreviousMode,
                                       (PVOID*)(PVOID)&Process,
                                       NULL);
    . . . . . .

    AddressSpace = &Process->AddressSpace;

    Status = ObReferenceObjectByHandle(SectionHandle,
                                       SECTION_MAP_READ,
                                       MmSectionObjectType,
                                       PreviousMode,
                                       (PVOID*)(PVOID)&Section,
                                       NULL);
    . . . . . .

    Status = MmMapViewOfSection(Section, Process,
                                (BaseAddress != NULL ? &SafeBaseAddress : NULL),
                                ZeroBits, CommitSize,
                                (SectionOffset != NULL ? &SafeSectionOffset : NULL),
                                (ViewSize != NULL ? &SafeViewSize : NULL),
                                InheritDisposition, AllocationType, Protect);
    ObDereferenceObject(Section);
    ObDereferenceObject(Process);
    . . . . . .
    return(Status);
}
```

以前讲过，NtMapViewOfSection()并不是专为当前进程的，而是可以用于任何已打开的进程，所以参数中既有共享内存区对象的 Handle，又有进程对象的 Handle。从这个意义上

说，两个进程之间通过共享内存区的通信完全可以由第三方撮合、包办而成，但是一般还是由通信双方自己加以映射的。显然，这里实质性的操作是 MmMapViewOfSection()，那是存储管理底层的事了，我们就不再追究下去了。

一旦把同一个共享内存区映射到了通信双方的用户空间(可能在不同的地址上)，出现在这双方用户空间的特定地址区间就落实到了同一组物理页面。这样，一方往里面写的内容就可以为另一方所见，于是就可以通过普通的内存访问(例如通过指针)实现进程间通信了。

但是，如前所述，通过共享内存区实现的只是原始的、低效的进程间通信，因为共享内存区只满足了前述三个条件中的两个，只是提供了信息的(实时)传输，但是却缺乏进程间的同步。所以实际使用时需要或者结合别的进程间通信(同步)手段，或者由应用程序自己设法实现同步(例如定时查询)。

2. 信号量(Semaphore)

学过操作系统原理的读者想必知道“临界区”和“信号量”、以及二者之间的关系。如果没有学过，那也不要紧，不妨把临界区想像成一个凭通行证入内的工作场所，作为对象存在的“信号量”则是发放通行证的“票务处”。但是，通行证的数量是有限的，一般只有寥寥几张。一旦发完，想要领票的进程(线程)就只好睡眠等待，直到已经在里面干活的进程(线程)完成了操作以后退出临界区并交还通行证，才会被唤醒并领到通行证进入临界区。之所以称之为(翻译为)“信号量”，是因为“票务处”必须维持一个数值，表明当前手头还有几张通行证，这就是作为数值的“信号量”。所以，“信号量”是一个对象，对象中有个数值，而信号量这个名称就是因这个数值而来。那么，为什么要到临界区里面去进行某些操作呢？一般是因为这些操作不允许受到干扰，必须排它地进行，或者说有互斥要求。

在操作系统理论中，“领票”操作称为 P 操作，具体的操作如下：

- 递减信号量的数值。
- 如果递减后大于或等于 0 就说明领到了通行证，因而就进入了临界区，可以接着进行想要做的操作了。
- 反之如果递减后小于 0，则说明通行证已经发完，当前线程只好(主动)在临界区的大门口睡眠，直至有通行证可发时才被唤醒。
- 如果信号量的数值小于 0，那么其绝对值表明正在睡眠等待的线程个数。

领到票进入了临界区的线程，在完成了操作以后应从临界区退出、并交还通行证，这个操作称为 V 操作，具体的操作如下：

- 递增信号量的数值。
- 如果递增以后的数值小于等于 0，就说明有进程正在等待，因而需要加以唤醒。

这里，执行了 P 操作的进程稍后就会执行 V 操作，但是也可以把这两种操作分开来，让有的进程光执行 V 操作，另一些进程则光执行 P 操作。于是，这两种进程就成了供应者/消费者的关系，前者是供应者，后者是消费者，而信号量的 P/V 操作正好可以被用作进程间的睡眠/唤醒机制。例如，假定最初时“通行证”的数量为 0，并把它理解为筹码。每当写入方在共享内存区中写入数据以后就对信号量执行一次 V 操作，而每当读出方想要读出新的数据时就先执行一次 P 操作。所以，许多别的进程间同步机制其实只是“信号量”的变种，是由“信号量”派生、演变而来的。

注意在条件不具备时进入睡眠、以及在条件具备时加以唤醒，是 P 操作和 V 操作固有的一部分，否则就退化成对于标志位或数值的测试和设置了。当然，“测试和设置”也是进程间同步的手段之一，但是那只相当于轮询，一般而言效率是很低的。所以，离开(睡眠)等待和唤醒，就不足于构成高效的进程间通信机制。但是也有例外，那就是如果能肯定所等待

的条件在一个很短的时间内一定会得到满足，那就不妨不断地反复测试直至成功，这就是“空转锁(SpinLock)”的来历。不过空转锁一般只是在内核中使用，而不提供给用户空间。

信号量对象的创建和打开是由 NtCreateSemaphore()和 NtOpenSemaphore()实现的，我们看一下 NtCreateSemaphore()：

```
NTSTATUS
STDCALL
NtCreateSemaphore(OUT PHANDLE SemaphoreHandle,
                  IN ACCESS_MASK DesiredAccess,
                  IN POBJECT_ATTRIBUTES ObjectAttributes    OPTIONAL,
                  IN LONG InitialCount,
                  IN LONG MaximumCount)
{
    . . . . . .
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    . . . . . .
    if(PreviousMode != KernelMode) {
        _SEH_TRY . . . . . ._SEH_END;
        if(!NT_SUCCESS(Status)) return Status;
    }
    . . . . . .
    /* Create the Semaphore Object */
    Status = ObCreateObject(PreviousMode, ExSemaphoreObjectType,
                            ObjectAttributes, PreviousMode, NULL,
                            sizeof(KSEMAPHORE), 0, 0, (PVOID*)&Semaphore);
    /* Check for Success */
    if (NT_SUCCESS(Status)) {
        /* Initialize it */
        KeInitializeSemaphore(Semaphore, InitialCount, MaximumCount);
        /* Insert it into the Object Tree */
        Status = ObInsertObject((PVOID)Semaphore, NULL,
                                DesiredAccess, 0, NULL, &hSemaphore);
        ObDereferenceObject(Semaphore);
        /* Check for success and return handle */
        if(NT_SUCCESS(Status)) {
            _SEH_TRY . . . . . ._SEH_END;
        }
    }
    /* Return Status */
    return Status;
}
```

这个函数可以说是对象创建函数的样板(Template)。一般而言，创建对象的过程总是涉及三步主要操作：

1. 通过 ObCreateObject()创建对象，对象的类型代码决定了具体的对象种类。对于信号量，这个类型代码是 ExSemaphoreObjectType。所创建的对象被挂入内核中该种类型的对象队列(类似于文件系统的目录)，以备别的进程打开。这个函数返回一个指向具体对象数据结构的指针，数据结构的类型取决于对象的类型代码。
2. 类似于 KeInitializeSemaphore()那样的初始化函数，对所创建对象的数据结构进行初始化。
3. 通过 ObInsertObject()将所创建对象的数据结构指针填入当前进程的打开对象表，并返回相应的 Handle。所以，创建对象实际上是创建并打开一个对象。

对于信号量，ObCreateObject()返回的是 KSEMAPHORE 数据结构指针：

```
typedef struct _KSEMAPHORE {
    DISPATCHER_HEADER Header;
    LONG Limit;
} KSEMAPHORE, *PKSEMAPHORE, *RESTRICTED_POINTER PRKSEMAPHORE;
```

这里的 Limit 是信号量数值的上限，来自前面的调用参数 MaximumCount。而参数 InitialCount 的数值、即信号量的初值，则记录在 DISPATCHER_HEADER 内的 SignalState 字段中。所以，信号量对象将其头部的 SignalState 字段用作了“信号量”。

NtOpenSemaphore()的代码就不看了，所有的打开对象操作都是一样的，基本上就是通过内核函数 ObOpenObjectByName()根据对象名(路径名)找到目标对象，然后将它的数据结构指针填入本进程的打开对象表，并返回相应的 Handle。

读者可能已经在急切想要知道信号量的 P/V 操作是怎么实现的。也许会使读者感到意外，Windows 并没有专为信号量的 P 操作而设的系统调用，信号量的 P 操作就是通过 NtWaitForSingleObject()或 NtWaitForMultipleObjects()实现的。事实上，所有与 P 操作类似、会使调用者阻塞的操作都是由这两个函数实现的。读者已经在上一篇漫谈中看过 NtWaitForSingleObject()的代码，这里就不重复了。

信号量的 V 操作倒是有专门的系统调用，那就是 NtReleaseSemaphore()。

```
NTSTATUS
STDCALL
NtReleaseSemaphore(IN HANDLE SemaphoreHandle,
                   IN LONG ReleaseCount,
                   OUT PLONG PreviousCount    OPTIONAL)
{
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    . . . . . .
    if(PreviousCount != NULL && PreviousMode == UserMode) {
        _SEH_TRY . . . . . ._SEH_END;
        if(!NT_SUCCESS(Status)) return Status;
    }

    . . . . . .
```

```
    /* Get the Object */
    Status = ObReferenceObjectByHandle(SemaphoreHandle,
                                       SEMAPHORE_MODIFY_STATE,
                                       ExSemaphoreObjectType, PreviousMode,
                                       (PVOID*)&Semaphore, NULL);

    /* Check for success */
    if (NT_SUCCESS(Status)) {

        /* Release the semaphore */
        LONG PrevCount = KeReleaseSemaphore(Semaphore, IO_NO_INCREMENT,
                                            ReleaseCount, FALSE);
        ObDereferenceObject(Semaphore);

        /* Return it */
        if(PreviousCount) {
            _SEH_TRY . . . . . _SEH_END;
        }
    }
    /* Return Status */
    return Status;
}
```

此类函数都是先调用 ObReferenceObjectByHandle()，以取得指向目标对象数据结构的指针，然后就对此数据结构执行具体对象类型的具体操作，在这里是 KeReleaseSemaphore()。

常规的 V 操作只使信号量加 1，可以理解为只提供一张通行证，而 KeReleaseSemaphore() 则对此作了推广，可以使信号量加 N，即同时提供好几张通行证，参数 ReleaseCount 就是这个增量，而 PreviousCount 则用来返回原来(V 操作之前)的信号量数值。这里的常数 IO_NO_INCREMENT 定义为 0，表示不需要提高被唤醒进程的调度优先级。

[NtReleaseSemaphore() > KeReleaseSemaphore()]

```
LONG    STDCALL
KeReleaseSemaphore(PKSEMAPHORE Semaphore,
                   KPRIORITY Increment,
                   LONG Adjustment,
                   BOOLEAN Wait)

{
    ULONG InitialState;
    KIRQL OldIrql;
    PKTHREAD CurrentThread;
```

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

    /* Save the Old State */
    InitialState = Semaphore->Header.SignalState;
    /* Check if the Limit was exceeded */
    if (Semaphore->Limit < (LONG) InitialState + Adjustment ||
                                                InitialState > InitialState + Adjustment) {
        /* Raise an error if it was exceeded */
        KeReleaseDispatcherDatabaseLock(OldIrql);
        ExRaiseStatus(STATUS_SEMAPHORE_LIMIT_EXCEEDED);
    }

    /* Now set the new state */
    Semaphore->Header.SignalState += Adjustment;
    /* Check if we should wake it */
    if (InitialState == 0 && !IsListEmpty(&Semaphore->Header.WaitListHead)) {
        /* Wake the Semaphore */
        KiWaitTest(&Semaphore->Header, Increment);
    }

    /* If the Wait is true, then return with a Wait and don't unlock the Dispatcher Database */
    if (Wait == FALSE) {
        /* Release the Lock */
        KeReleaseDispatcherDatabaseLock(OldIrql);
    } else {
        /* Set a wait */
        CurrentThread = KeGetCurrentThread();
        CurrentThread->WaitNext = TRUE;
        CurrentThread->WaitIrql = OldIrql;
    }
    /* Return the previous state */
    return InitialState;
}
```

参数 Adjustment 就是信号量数值的增量，另一个参数 Increment 如为非 0 则表示要为被唤醒的线程暂时增加一些调度优先级，使其尽快得到运行的机会。还有个参数 Wait 的作用下面就会讲到。

程序中 KeAcquireDispatcherDatabaseLock()的作用是提升程序的运行级别(称为 IRQL，以后在别的漫谈中会讲到这个问题)，以禁止线程调度，直至执行与之配对的函数 KeReleaseDispatcherDatabaseLock()为止。这样，在这两个函数调用之间就形成了一个“调度禁区”。可是我们从代码中看到，KeReleaseDispatcherDatabaseLock()之是否执行实际上取决于参数 Wait。这是为什么呢？我在上一篇漫谈中讲到，在 Windows 中，当一个线程要在某

个或某几个对象上等待某些事态的发生时，有两个系统调用可资调用，一个是NtWaitForSingleObject()，另一个是 NtWaitForMultipleObjects()。可是其实还有一个，就是NtSignalAndWaitForSingleObject()，只是这个系统调用有些特殊。正如其函数名所示，这个系统调用一方面是“Signal”一个对象，就是对其执行类似于 KeReleaseSemaphore()这样的操作；另一方面自己又立即在另一个对象上等待，类似于执行 NtWaitForSingleObject()；而且这二者应该是一气呵成的。这样就来了问题，如果在 KeReleaseSemaphore()一类的函数中一律调用 KeReleaseDispatcherDatabaseLock()，然后在 NtWaitForSingleObject()中又调用KeAcquireDispatcherDatabaseLock()，那么在此二者之间就有个间隙，在此间隙中是可以发生线程调度的。再说，从程序效率的角度，那样不必要地(且不说有害)来回折腾，也是不可取的，理应加以优化。像现在这样有条件地执行 KeReleaseDispatcherDatabaseLock()，就避免了这个问题。

对信号量本身的操作倒很简单，就是改变 Semaphore->Header.SignalState 的数值。同时，如果有线程在睡眠等待(队列非空)，并且此前信号量的数值是 0，那么既然现在退还了若干张通行证(增加了信号量的数值)，就可以放几个正在等待的进程进入临界区了，所以通过KiWaitTest()唤醒等待中的进程。至于 KiWaitTest()，读者在上一篇漫谈中已经看过它的代码了。注意这里对于睡眠/唤醒的处理与传统的 P/V 操作略有些不同。在传统的 P 操作中，每执行一次 P 操作、不管能否进入临界区、都使信号量的值递减，所以信号量可以有负值，而且此时其绝对值就是正在睡眠等待的进程的数量。另一方面，当事进程之能否进入临界区也是按递减了以后的信号量数值判定的。而在 NtWaitForSingleObject()、KiIsObjectSignaled()、KiSatisfyObjectWait()、以及 KiWaitTest()的代码中，则当事进程只有在信号量大于 0 时才能获准进入临界区，这样的 P 操作才使信号量的值递减，否则当事进程就被挂入等待队列并进入睡眠，因此信号量不会有负值。所以，NtWaitForSingleObject()是变了形的 P 操作。

如前所述，信号量既可以用来实现临界区，也可以使进程(线程)之间形成供应者/消费者的关系和互动。所以，虽然从表面上看信号量操作本身并不携带数据，但是它为高效的进程间通信提供了同步手段。另一方面，进程间同步也蕴含着信息的交换，也属于进程间通信的范畴，所以信号量同时又是一种进程间通信机制。

3. 互斥门(Mutant)

互斥门(Mutant，又称 Mutex，实现于内核中称 Mutant，实现于用户空间称 Mutex)是“信号量”的一个特例和变种。在信号量机制中，如果把信号量的最大值和初始值都设置成 1，就成了互斥门。把信号量的最大值和初始值都设置成 1，就相当于一共只有一张通行证，自然就只能有一个线程可以进入临界区；在它退出临界区之前，别的线程想要进入临界区就只好在大门口睡眠等候。所谓“互斥”，就是因此而来。我的朋友胡希明老师曾把这样的临界区比作列车上的厕所(当时他常坐火车出差，想必屡屡为此所苦)，二十多年过去了，当年的学生聚在一起还会因此事津津乐道。

不过，倘若纯粹就是两个参数的事，那就没有必要另搞一套了。事实上互斥门机制有一些特殊性，下面读者就会看到。

为互斥门的创建和打开提供了 NtCreateMutant()和 NtOpenMutant()两个系统调用，代码就不用看了。下面是互斥门对象的数据结构：

```
typedef struct _KMUTANT {
   DISPATCHER_HEADER       Header;
   LIST_ENTRY              MutantListEntry;
   struct _KTHREAD         *RESTRICTED_POINTER OwnerThread;
```

```
    BOOLEAN                 Abandoned;
    UCHAR                   ApcDisable;
} KMUTANT, *PKMUTANT, KMUTEX, *PKMUTEX;
```

这个数据结构的定义见之于 Windows NT 的 DDK，所以是“正宗”的。可见，这数据结构就与信号量对象的不同。

跟信号量机制一样，请求(试图)通过互斥门进入临界区的操作就是系统调用 NtWaitForSingleObject()或 NtWaitForMultipleObjects()。不过，在 NtWaitForSingleObject()内部，特别是在判定能否进入临界区时所调用的函数 KiIsObjectSignaled()中，其实是按不同的对象类型分别处置的。我们不妨看一下。

[NtWaitForSingleObject() > KeWaitForSingleObject() > KiIsObjectSignaled()]

```
BOOLEAN inline FASTCALL
KiIsObjectSignaled(PDISPATCHER_HEADER Object, PKTHREAD Thread)
{
    /* Mutants are...well...mutants! */
   if (Object->Type == MutantObject) {
        /*
         * Because Cutler hates mutants, they are actually signaled if the Signal State is <= 0
         * Well, only if they are recursivly acquired (i.e if we own it right now).
         * Of course, they are also signaled if their signal state is 1.
         */
        if ((Object->SignalState <= 0 && ((PKMUTANT)Object)->OwnerThread == Thread) ||
            (Object->SignalState == 1)) {
            /* Signaled Mutant */
            return (TRUE);
        } else {
            /* Unsignaled Mutant */
            return (FALSE);
        }
    }

    /* Any other object is not a mutated freak, so let's use logic */
   return (!Object->SignalState <= 0);
}
```

可见，互斥门在这里是作为一种特殊情况处理的，使 KiIsObjectSignaled()返回 TRUE、从而允许当事进程进入临界区的条件之一是 SignalState 为 1。另一个条件表明，只要是互斥门对象当前的“业主(Owner)”，就不受这个限制，即使没有通行证也可以进入。那么谁是互斥门对象当前的业主呢？那就是当前已经在此临界区中的线程，这一点读者看了下面的代码就会清楚。可是既然是已经在临界区中的线程，怎么又会企图通过同一个互斥门进入同一个临界区呢？这意味着一个线程可能递归地多次通过同一个互斥门。这个问题先搁一下，等一下再来探讨。

在上一篇漫谈中，我们看了 KeWaitForSingleObject()的代码，这是 NtWaitForSingleObject()的主体，正是这个函数调用了 KiIsObjectSignaled()。如果 KiIsObjectSignaled()返回 TRUE，那就说明当前进程可以领到通行证而进入临界区，此时需要执行 KiSatisfyObjectWait()，一方面是进行“账面”上的处理，一方面也还有一些附加的操作需要进行，而这些附加的操作是因具体的对象而异的。我们再重温一下这个函数的代码。

[NtWaitForSingleObject() > KeWaitForSingleObject() > KiSatisfyObjectWait()]

```
VOID    FASTCALL
KiSatisfyObjectWait(PDISPATCHER_HEADER Object, PKTHREAD Thread)
{
    /* Special case for Mutants */
    if (Object->Type == MutantObject) {
        /* Decrease the Signal State */
        Object->SignalState--;
        /* Check if it's now non-signaled */
        if (Object->SignalState == 0) {
            /* Set the Owner Thread */
            ((PKMUTANT)Object)->OwnerThread = Thread;
            /* Disable APCs if needed */
            Thread->KernelApcDisable -= ((PKMUTANT)Object)->ApcDisable;
            /* Check if it's abandoned */
            if (((PKMUTANT)Object)->Abandoned) {
                /* Unabandon it */
                ((PKMUTANT)Object)->Abandoned = FALSE;
                /* Return Status */
                Thread->WaitStatus = STATUS_ABANDONED;
            }
            /* Insert it into the Mutant List */
            InsertHeadList(&Thread->MutantListHead,
                                    &((PKMUTANT)Object)->MutantListEntry);
        }
    } else if ((Object->Type & TIMER_OR_EVENT_TYPE) == EventSynchronizationObject) {
        /* These guys (Syncronization Timers and Events) just get un-signaled */
        Object->SignalState = 0;
    } else if (Object->Type == SemaphoreObject) {
        /* These ones can have multiple signalings, so we only decrease it */
        Object->SignalState--;
    }
}
```

我们只看对于互斥门对象的处理。首先是递减 SignalState，这就是所谓“账面”上的处理，也是 P 操作的一部分。由于前面已经通过 KiIsObjectSignaled()进行过试探，如果当时的 SignalState 数值为 1，或者说如果当时的临界区是空的，那么现在的 SignalState 数值必定变

成了 0。所以，下面 if 语句中的代码是在一个线程首次进入一个互斥门时执行的。这里说的“首次进入”并不是指退出以后又进去那样的反复进出中的首次，而是指嵌套多次进入中的首次。当一个线程首次顺利进入互斥门时，它就成了这个互斥门当前的业主，直至退出；所以把互斥门数据结构中的 OwnerThread 字段设置成指向当前线程的 KTHREAD 数据结构。此外，如果 Abandoned 字段显示这个互斥门行将被丢弃，则暂时将其改成继续使用(因为又有线程进来了)，但是把这情况记录在当前线程的 KTHREAD 数据结构中。互斥门数据结构中的 ApcDisable 字段表明通过互斥门进入临界区的线程是否需要关闭 APC 请求，现在当前进程通过了互斥门，所以要把这信息记录在它的数据结构中。注意这里是从 Thread->KernelApcDisable 的数值中减去互斥门的 ApcDisable 的值，结果为非 0(负数)表示关闭 APC 请求，而 ApcDisable 的值则非 1 即 0。举例言之，假定 Thread->KernelApcDisable 原来是 0，而 ApcDisable 为 1，则相减以后的结果为-1，表示关闭 APC 请求。最后，当前进程既已成为这个互斥门的主人，二者之间就有了连系，所以通过队列把它们结合起来，这是因为一个线程有可能同时存在于几个临界区中。

应该说这里别的都还好理解，成为问题的是为什么要允许嵌套进入互斥门。据“Programming the Microsoft Windows Driver Model”书中说，互斥门的特点之一就是允许嵌套进入，而优点之一则是可以防止死锁。书中并没有明确讲这二者之间是否存在因果关系，所以我们只能分析和猜测。首先，如过互斥门不允许嵌套进入(在前面的代码中取消允许当前业主进入的条件)，而已经通过互斥门进入临界区的线程又对同一个互斥门进行 P 操作，那么肯定是会引起死锁的。这个线程会因为在 P 操作中不能通过互斥门而进入睡眠，能唤醒其睡眠的是已经在这个临界区中的线程(如果它执行 V 操作的话)，可是这正是已经在睡眠等待的那个线程本身，所以就永远不会被唤醒。反之，有了前面 KiIsObjectSignaled()中那样的安排，即允许互斥门当前的业主递归通过，那确实就可以避免由此而导致的死锁。

可是，为什么要企图递归通过同一个互斥门呢？既然已经通过这个具体的互斥门进入了临界区，为什么还要再一次试图进入同一个互斥门呢？应该说，在精心设计和实现的软件中是不应该有这种情况出现的。可是，考虑到应用软件的可重用(reuse)，有时候也许会有这种情况。例如，一个线程在临界区内可能调用某个软件模块所提供的操作，而这个软件模块可能需要通过 NtWaitForMultipleObjects()进入由多个互斥门保护的复合临界区，可是其中之一就是已经进入的那个互斥门。在这种情况下，对于已经进入的那个互斥门而言，就构成了递归进入。当然，我们可以通过修改那个软件模块来避免此种递归，但这可能又不是很现实。在这样的条件下，允许递归进入不失为一个简单的解决方案。

还要说明，允许递归通过互斥门固然可以防止此种特定形式的死锁，却并不是对所有的死锁都有效。真要防止死锁，还是得遵守有关的准则，精心设计，精心实现。

读者也许会问：这里所引的代码出自 ReactOS，所反映的是 ReactOS 的作者们对 Windows 互斥门的理解，但是他们的理解是否正确呢？确实，Windows 的代码是不公开的，所以也无从对比。可是，虽然 Windows 的代码不公开，它的一些数据结构的定义却是公开的，这里面就包括 KMUTANT，所以前面特地说明了这是来自 Windows DDK(其实 ReactOS 的许多数据结构都可以在 DDK 中找到)。既然我们知道互斥门允许递归进入，又看到 KMUTANT 中确有 OwnerThread 这个指针，那么我们就有理由相信 ReactOS 的这些代码离“真相”不会太远。当然，我们还可以、也应该、设计出一些实验来加以对比、验证。

再看从临界区退出并交还“通行证”的操作、即 V 操作，这就是系统调用 NtReleaseMutant()。

```
NTSTATUS    STDCALL
```

```
NtReleaseMutant(IN HANDLE MutantHandle, IN PLONG PreviousCount    OPTIONAL)
{
    PKMUTANT Mutant;
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    NTSTATUS Status = STATUS_SUCCESS;

    . . . . . .
    if(PreviousMode == UserMode && PreviousCount) {
        _SEH_TRY . . . . . _SEH_END;
        if(!NT_SUCCESS(Status)) return Status;
    }
    /* Open the Object */
    Status = ObReferenceObjectByHandle(MutantHandle, MUTANT_QUERY_STATE,
                                ExMutantObjectType, PreviousMode, (PVOID*)&Mutant, NULL);
    /* Check for Success and release if such */
    if(NT_SUCCESS(Status)) {

        LONG Prev;

        /* Save the Old State */
        DPRINT("Releasing Mutant\n");
        Prev = KeReleaseMutant(Mutant, MUTANT_INCREMENT, FALSE, FALSE);
        ObDereferenceObject(Mutant);

        /* Return it */
        if(PreviousCount) {
            _SEH_TRY . . . . . _SEH_END;
        }
    }
    /* Return Status */
    return Status;
}
```

显然，这里实质性的操作是 KeReleaseMutant()，我们顺着往下看。

[NtReleaseMutant() > KeReleaseMutant()]

```
LONG
STDCALL
KeReleaseMutant(IN PKMUTANT Mutant, IN KPRIORITY Increment,
                    IN BOOLEAN Abandon, IN BOOLEAN Wait)
{
    KIRQL OldIrql;
    LONG PreviousState;
```

```
PKTHREAD CurrentThread = KeGetCurrentThread();

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

/* Save the Previous State */
PreviousState = Mutant->Header.SignalState;

/* Check if it is to be abandonned */
if (Abandon == FALSE) {
      /* Make sure that the Owner Thread is the current Thread */
      if (Mutant->OwnerThread != CurrentThread) {
            DPRINT1("Trying to touch a Mutant that the caller doesn't own!\n");
            ExRaiseStatus(STATUS_MUTANT_NOT_OWNED);
      }
      /* If the thread owns it, then increase the signal state */
      Mutant->Header.SignalState++;
} else    {
      /* It's going to be abandonned */
      DPRINT("Abandonning the Mutant\n");
      Mutant->Header.SignalState = 1;
      Mutant->Abandoned = TRUE;
}

/* Check if the signal state is only single */
if (Mutant->Header.SignalState == 1) {
      if (PreviousState <= 0) {
            DPRINT("Removing Mutant\n");
            RemoveEntryList(&Mutant->MutantListEntry);
      }
      /* Remove the Owning Thread and wake it */
      Mutant->OwnerThread = NULL;
      /* Check if the Wait List isn't empty */
      DPRINT("Checking whether to wake the Mutant\n");
      if (!IsListEmpty(&Mutant->Header.WaitListHead)) {
            /* Wake the Mutant */
            DPRINT("Waking the Mutant\n");
            KiWaitTest(&Mutant->Header, Increment);
      }
}

/* If the Wait is true, then return with a Wait and don't unlock the Dispatcher Database */
if (Wait == FALSE) {
      /* Release the Lock */
```

```
            KeReleaseDispatcherDatabaseLock(OldIrql);
        } else {
            /* Set a wait */
            CurrentThread->WaitNext = TRUE;
            CurrentThread->WaitIrql = OldIrql;
        }
        /* Return the previous state */
        return PreviousState;
}
```

参数 Abandon 表示在退出临界区以后是否要废弃这个互斥门。我们从 NtReleaseMutant()的代码中可以看出，实际传下来的参数值是 FALSE。那么什么情况下这个参数会是 TRUE 呢？据“Native API”书中说，这发生于互斥门的业主、就是已经通过这个互斥门进入了临界区的线程突然要结束其生命的时候。

既然临界区中的线程要退出，这个互斥门就变成无主的了，所以把 Mutant->OwnerThread 设置成 NULL。其余的代码就留给读者自己去理解了。

4. 事件(Event)

信号量机制的另一个变种是“事件”，这是通过事件对象实现的。Windows 为事件对象的创建和打开提供了 NtCreateEvent()和 NtOpenEvent()两个系统调用。由于所有此类函数的相似性，这两个系统调用的代码就不用看了，只要知道内核中代表着事件对象的数据结构是 KEVENT 就可以了：

```
typedef struct _KEVENT {
   DISPATCHER_HEADER    Header;
} KEVENT, *PKEVENT, *RESTRICTED_POINTER PRKEVENT;
```

这就是说，除 DISPATCHER_HEADER 以外，KEVENT 就不需要有别的什么字段了。

Windows 定义和提供了两种不同类型的事件，每个事件对象也因此而分成两种类型，这就是：

```
typedef enum _EVENT_TYPE {
   NotificationEvent,
   SynchronizationEvent
} EVENT_TYPE;
```

事件对象的类型是在创建的时候(通过参数)设定的，记录在对象头部的 Type 字段中，设定以后就不能改变。为不同的应用和目的需要使用不同类型的事件对象。

类型为 NotificationEvent 的事件代表着“通知”。通知是广播式的，其作用就是通知公众某个事件业已发生(Header 中的 SignalState 为 1)，一个观看者看了这个布告并不影响别的观看者继续观看。所以，在通知型事件对象上的 P 操作并不消耗资源，也就是不改变其数值。在这一点上它就像是一个全局(跨进程)的变量。但是，如果事件尚未发生(SignalState 为 0)，则所有的观看者、即对此对象执行 P 操作的线程全都被阻塞而进入睡眠，直到该事件发

生，在这一点上又不太像“通知”，而反倒是起着同步的作用了(设想你去看高考发榜，但是还没贴出来，你就被“套住”等在那儿了)。读者也许会想到，既然 P 操作不改变 SignalState 的值，那岂不是一旦 SignalState 变成 1 就永远是 1、从而事件对象只能一次性使用了？这确实是个问题，所以 Windows 又专门提供了一个系统调用 NtResetEvent()，用来“重启(Reset)”一个事件对象、即将其 SignalState 清 0。

类型为 SynchronizationEvent 的事件对象则用于同步，这就相当于初值为 0、最大值为 1 的信号量。对于同步型的事件对象，一次 P 操作相当于消耗一个筹码(通行证)，而 V 操作则相当于提供一个筹码。

回顾一下前面 KiIsObjectSignaled()的代码，这是 P 操作中用来判断是否可以(拿到筹码)进入临界区的函数。这个函数对于除互斥门以外的所有对象都返回(!Object->SignalState <= 0)。这就是说，不管是同步型还是通知型的事件对象，执行 P 操作的线程能拿到筹码或看到通知的条件都是 SignalState 为 1，否则就要睡眠等待。

再回顾一下 KiSatisfyObjectWait()的代码，这是 P 操作中拿到筹码以后的操作：

```
KiSatisfyObjectWait(PDISPATCHER_HEADER Object, PKTHREAD Thread)
{
    /* Special case for Mutants */
    if (Object->Type == MutantObject) {
        . . . . . .
    } else if ((Object->Type & TIMER_OR_EVENT_TYPE) == EventSynchronizationObject) {
        /* These guys (Syncronization Timers and Events) just get un-signaled */
        Object->SignalState = 0;
    } else if (Object->Type == SemaphoreObject) {
        /* These ones can have multiple signalings, so we only decrease it */
        Object->SignalState--;
    }
}
```

这里 Object->Type 的最低 3 位记录着对象的类型，如果是 EventSynchronizationObject 就说明是同步型的事件对象，此时把 Object->SignalState 置 0，表示把筹码消耗掉了。由于事件对象的 SignalState 只有两个值 0 或非 0，因而在 SignalState 为 1 的条件下将其设置成 0 跟使之递减是等价的。可是，如果是通知型的事件对象，那就没有任何操作，所以并没有把通知“消耗”掉。所以，这又是变相的 P 操作。

跟信号量和互斥门一样，对事件对象的 P 操作就是系统调用 NtWaitForSingleObject()或 NtWaitForMultipleObjects()，或者(如果从内核中调用)也可以是 KeWaitForSingleObject()，而 KiIsObjectSignaled()和 KiSatisfyObjectWait()都是在 P 操作内部调用的函数。

事件对象的 V 操作是系统调用 NtSetEvent()，意思是把事件对象的 SignalState 设置成 1。就像 NtReleaseMutant()的主体是 KeReleaseMutant()一样，NtSetEvent()的主体是 KeSetEvent()。我们跳过 NtSetEvent()这一层，直接看 KeSetEvent()的代码。

[NtSetEvent() > KeSetEvent()]

```
LONG    STDCALL
```

```
KeSetEvent(PKEVENT Event, KPRIORITY Increment, BOOLEAN Wait)
{
    . . . . . .
    /* Lock the Dispathcer Database */
    OldIrql = KeAcquireDispatcherDatabaseLock();
    /* Save the Previous State */
    PreviousState = Event->Header.SignalState;

    /* Check if we have stuff in the Wait Queue */
    if (IsListEmpty(&Event->Header.WaitListHead)) {
        /* Set the Event to Signaled */
        DPRINT("Empty Wait Queue, Signal the Event\n");
        Event->Header.SignalState = 1;
    } else {
        /* Get the Wait Block */
        WaitBlock = CONTAINING_RECORD(Event->Header.WaitListHead.Flink,
                                                    KWAIT_BLOCK, WaitListEntry);
        /* Check the type of event */
        if (Event->Header.Type == NotificationEvent || WaitBlock->WaitType == WaitAll) {
            if (PreviousState == 0) {
                /* We must do a full wait satisfaction */
                DPRINT("Notification Event or WaitAll, Wait on the Event and Signal\n");
                Event->Header.SignalState = 1;
                KiWaitTest(&Event->Header, Increment);
            }
        } else {
            /* We can satisfy wait simply by waking the thread, since our signal state is 0 now */
            DPRINT("WaitAny or Sync Event, just unwait the thread\n");
            KiAbortWaitThread(WaitBlock->Thread, WaitBlock->WaitKey, Increment);
        }
    }

    /* Check what wait state was requested */
    if (Wait == FALSE) {
        /* Wait not requested, release Dispatcher Database and return */
        KeReleaseDispatcherDatabaseLock(OldIrql);
    } else {
        /* Return Locked and with a Wait */
        KTHREAD *Thread = KeGetCurrentThread();
        Thread->WaitNext = TRUE;
        Thread->WaitIrql = OldIrql;
    }
    /* Return the previous State */
    return PreviousState;
```

}

参数 Increment 和 Wait 所起的作用与前面 KeReleaseSemaphore()中的相同。所谓“设置事件”其实就是一种特殊的、变通的 V 操作。具体的操作分两种情况：

1. 如果 IsListEmpty()为真，即等待队列是空的、没有线程在睡眠等待，就把 Event->Header.SignalState 设置成 1。这样，如果此后有线程对此事件对象执行 P 操作、即 NtWaitForSingleObject()，就因此而不必睡眠等待。这对于通知型和同步型的事件对象都是一样。
2. 已经有线程在这个对象上睡眠等待，那就要从中唤醒一个或所有线程。这时候的处理取决于事件对象的类型以及等待的方式：
   - 对于通知型的事件对象，或者等待者的等待方式是 WaitAll，而且此前 SignalState 为 0，就将 SignalState 置 1，并通过 KiWaitTest()唤醒这个线程，以及等待队列中所有符合条件的线程。可是，要是 SignalState 本来就已经是 1，则没有任何影响。
   - 否则，对于同步型的事件对象，并且等待者的等待方式是 WaitAny，就通过 KiAbortWaitThread()唤醒等待队列中的第一个线程。此时并不改变 SignalState 的值。因为既然唤醒了一个线程，就已经把这筹码消耗掉了。

这里 KiWaitTest()和 KiAbortWaitThread()的区别在于：KiWaitTest()是在一个 while 循环中对等待队列中的所有进程执行 KiAbortWaitThread()，条件是 SignalState 大于 0。对于信号量，由于每唤醒一个线程就使 SignalState 减 1，这循环很快就停止了，一般是只唤醒一个线程。但是如前所述，通知型事件对象在唤醒一个线程的时候不改变 SignalState 的值。于是，这个 while 循环就会唤醒等待队列中的所有进程。不过这里也有例外，如果其中的某个线程是在多个“可等待对象”上等待，而且等待方式是 WaitAll，那就还要看是否别的条件也满足了，不然就只好把它跳过，这也是 KiWaitTest()的代码中按排好了的。

前面讲过，一旦将通知型事件对象的 SignalState 设置成 1，它就一直保持为 1，P 操作不会改变它的值。即使再对其执行一次 KeSetEvent()，也不会改变它的值，因为本来就已经是 1 了。为了使其变成 0，以便再次使用这个事件对象，就需要对其执行另一个系统调用 NtResetEvent()。同样，NtResetEvent()的主体是 KeResetEvent()。

[NtSetEvent() > KeResetEvent()]

```
LONG    STDCALL
KeResetEvent(PKEVENT Event)
{
    KIRQL OldIrql;
    LONG PreviousState;

    DPRINT("KeResetEvent(Event %x)\n",Event);

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

    /* Save the Previous State */
```

```
    PreviousState = Event->Header.SignalState;

    /* Set it to zero */
    Event->Header.SignalState = 0;

    /* Release Dispatcher Database and return previous state */
    KeReleaseDispatcherDatabaseLock(OldIrql);
    return PreviousState;
}
```

解释就没有必要了。注意调用 NtResetEvent()的不必就是 NtSetEvent()的调用者，而可以是别的线程或内核模块。不过有些内核模块只能调用 KeResetEvent()，而不是 NtResetEvent()。

Windows 还有个系统调用 NtPulseEvent()，这是把 NtSetEvent()和 NtResetEvent()组合在了一起，相当于先 NtSetEvent()、然后马上就 NtResetEvent()。这样，其效果就是唤醒已经在通知型事件对象上等待的所有线程，但是下不为例。而对于同步型事件对象则大致等同于 NtSetEvent()。

可见，虽然“事件”实质上是“信号量”的一种特例和变种，但是在使用上却有着明显的差别。信号量的“正宗”的用途是构筑临界区。在这种应用中，一个线程得以通过 P 操作进入临界区的原因可能是有另一个线程执行了 V 操作，但是既然进了临界区就总有从临界区退出而执行 V 操作的时候。这样，一个线程在 P 操作以后总是有个 V 操作。从总体上看，每个线程的 P 操作和 V 操作是平衡的、即数量相等的。但是“事件”则不同，“事件”并不是用来构筑临界区、而纯粹是用于线程间同步的。在这里，等待事件发生的一方总是执行 P 操作，而发出事件通知的一方则总是执行 V 操作。在前面对于“信号量”的比喻中把 P 操作比作领取通行证，把 V 操作比作交还通行证。相比之下，对于“事件”则相当于领取的通行证从来不交还，而另有供应者在不时地提供新的通行证。而且，特别有意义的是，发出事件通知的一方还不必非得是一个线程，也可以是内核中的某些子系统，例如设备驱动，所以也可以用于线程与内核之间的同步，特别是广泛地应用于设备驱动。当然，发出事件通知的一方更不必局限于某一个特定的线程，而是任何一个线程都可以。

为了帮助读者加深对事件机制的理解，下面是一个内核线程 DebugLogThreadMain 的代码：

```
VOID STDCALL
DebugLogThreadMain(PVOID Context)
{
  KIRQL oldIrql;
  IO_STATUS_BLOCK Iosb;
  static CHAR Buffer[256];
  ULONG WLen;

  for (;;)
    {
      LARGE_INTEGER TimeOut;
      TimeOut.QuadPart = -5000000; /* Half a second. */
      KeWaitForSingleObject(&DebugLogEvent, 0, KernelMode, FALSE, &TimeOut);
```

```
        KeAcquireSpinLock(&DebugLogLock, &oldIrql);
        while (DebugLogCount > 0)
        {
          if (DebugLogStart > DebugLogEnd)
          {
               WLen = min(256, DEBUGLOG_SIZE - DebugLogStart);
               memcpy(Buffer, &DebugLog[DebugLogStart], WLen);
               Buffer[WLen + 1] = '\n';
               DebugLogStart =    (DebugLogStart + WLen) % DEBUGLOG_SIZE;
               DebugLogCount = DebugLogCount - WLen;
               KeReleaseSpinLock(&DebugLogLock, oldIrql);
               NtWriteFile(DebugLogFile, NULL, NULL, NULL, &Iosb, Buffer, WLen + 1,
                              NULL, NULL);
          }
          else
          {
               WLen = min(255, DebugLogEnd - DebugLogStart);
               memcpy(Buffer, &DebugLog[DebugLogStart], WLen);
               DebugLogStart =
                        (DebugLogStart + WLen) % DEBUGLOG_SIZE;
               DebugLogCount = DebugLogCount - WLen;
               KeReleaseSpinLock(&DebugLogLock, oldIrql);
               NtWriteFile(DebugLogFile, NULL, NULL, NULL, &Iosb, Buffer, WLen,
                              NULL, NULL);
            }
          KeAcquireSpinLock(&DebugLogLock, &oldIrql);
        }
        KeResetEvent(&DebugLogEvent);
        KeReleaseSpinLock(&DebugLogLock, oldIrql);
      }
}
```

这个内核线程是为内核调试日志(Log)服务的。内核中有个环形缓冲区 DebugLog[]，以及用作该数组下标的变量 DebugLogStart 和 DebugLogEnd，还有表示环形缓冲区中数据长度的变量 DebugLogCount。不管是哪一个线程，只要是进入了内核，如果需要在日志中写上一笔，就可以把字符串拷贝到这个环形缓冲区中，然后要求这个内核线程把内容写到一个日志文件中。为此当然需要同步，这是通过一个(同步型)事件对象 DebugLogEvent 达成的。由于是在内核中，所以这里对事件对象的操作都直接调用其内核版本，例如 KeResetEvent()、而不是 NtResetEvent()。此外，对于环形缓冲区的使用当然还需要互锁，这是通过“空转锁” DebugLogLock 实现的，不过那不是我们此刻所关心的。

每当需要生成一项日志时，可以调用 DebugLogWrite()：

```
VOID
DebugLogWrite(PCH String)
```

```
{
   KIRQL oldIrql;

    . . . . . .
    KeAcquireSpinLock(&DebugLogLock, &oldIrql);

    if (DebugLogCount == DEBUGLOG_SIZE)
    {
        DebugLogOverflow++;
        KeReleaseSpinLock(&DebugLogLock, oldIrql);
        if (oldIrql < DISPATCH_LEVEL)
        {
           KeSetEvent(&DebugLogEvent, IO_NO_INCREMENT, FALSE);
        }
           return;
    }

    while ((*String) != 0)
    {
        DebugLog[DebugLogEnd] = *String;
        String++;
        DebugLogCount++;

        if (DebugLogCount == DEBUGLOG_SIZE)
        {
           DebugLogOverflow++;
           KeReleaseSpinLock(&DebugLogLock, oldIrql);
           if (oldIrql < DISPATCH_LEVEL)
           {
               KeSetEvent(&DebugLogEvent, IO_NO_INCREMENT, FALSE);
           }
           return;
        }
        DebugLogEnd = (DebugLogEnd + 1) % DEBUGLOG_SIZE;
    }

     KeReleaseSpinLock(&DebugLogLock, oldIrql);

     if (oldIrql < DISPATCH_LEVEL)
     {
        KeSetEvent(&DebugLogEvent, IO_NO_INCREMENT, FALSE);
     }
}
```

对于这段代码，以及对于 DebugLogWrite()和 DebugLogThreadMain()之间怎样互动，这里就不作解释了。只是要指出：DebugLogWrite()的每次执行可能都在不同线程的上下文里、代表着不同的线程，因为任何线程都可以调用 DebugLogWrite()。另外，想必读者已经注意到，KeResetEvent()是由 DebugLogThreadMain()自己调用、而不是由别的线程调用的。

介绍完事件对象，还应该提一下，Windows 还有一种特殊的“事件对(EventPair)”对象。与此有关的系统调用有这么一些：

NtCreateEventPair()
NtOpenEventPair()
NtWaitHighEventPair()
NtWaitLowEventPair()
NtSetHighWaitLowEventPair()
NtSetLowWaitHighEventPair()
NtSetHighEventPair()
NtSetLowEventPair()

顾名思义，“事件对”就是把两个事件对象紧密地组合在一起。事实上也正是如此，一个事件对由“高”、“低”两个事件对象组合构成，其设计意图是用于“点对点”的双向进程间通信。实际上这是为提高 Windows 进程与服务进程 Csrss 之间的通信效率而设置的(Csrss 是 Windows 子系统的管理/服务进程)。早期的 csrss 承担着许多操作，Windows 进程与 Csrss 之间的通信非常频繁，所以其效率至关重要。这种进程间通信的典型情景就是一方唤醒另一方、自身却又进入睡眠，反过来等待被对方唤醒，就像打乒乓球一样，为此就专门设计了 NtSetHighWaitLowEventPair()和 NtSetLowWaitHighEventPair()两个系统调用。不仅如此，为了尽可能地提高效率(在这种情况下的优化甚至是以 CPU 的时钟周期数计算的)，还专门单独分配了两个中断向量 0x2B 和 0x2C，而不跟别的系统调用合用 0x2E。不过，后来 Csrss 的许多操作被移到了内核中，不再需要那么频繁的进程间通信了，因而在效率上的容忍度也宽松了一些，所以现在又回到了 0x2E，而不再使用 0x2B 和 0x2C 这两个中断向量。

5. 命名管道(Named Pipe)和信箱(Mail Slot)

前面提到，如果从字面上理解，那么进程间通信也可以通过磁盘文件而实现。但是，把信息写入某个磁盘文件，再由另一个进程从磁盘文件读出，在速度上是很慢的。固然，由于文件缓冲区(Cache)的存在，对磁盘文件的写和读未必都经过磁盘，但是那并没有保证。再说，普通的文件操作也没有提供进程间同步的手段。所以通过普通的磁盘文件实现进程间通信是不太现实的。但是这也提示我们，如果能实现一种特殊文件，使得对文件的读写只在缓冲区中进行(而不写入磁盘)，并且实现进程间的同步，那倒是个不坏的主意。命名管道就是这样一种特殊文件。实际上，命名管道还不仅是这样的特殊文件，它还是一种网络通信的机制，只是当通信的双方存在于同一台机器上时，才落入本文所说的进程间通信的范畴。

既然命名管道是一种特殊文件，它的创建、打开、读写等等操作就基本上都可以利用文件系统中的有关资源加以实现。当然，这毕竟是一种特殊文件，对于使用者来说，最大的特殊之处在于这是一个“先进先出”的字节流，不能对其执行 lseek()一类的操作。

先看命名管道的创建，Windows 的 Win32 API 上提供了一对库函数 CreateNamedPipeA()和 CreateNamedPipeW()，前者用于 ASCII 码字符串，后者用于“宽字符”即 Unicode 的字符串，实际上前者只是把 8 位字符转换成 Unicode 以后再调用后者。对 CreateNamedPipeW()的调用大致如下：

```
Handle = CreateNamedPipeW(L"\\\\.\\pipe\\MyControlPipe",
                          PIPE_ACCESS_DUPLEX,
                          PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
                          . . . . . .);
```

这里的管道名实际上是“\\.\pipe\MyControlPipe”，前面的“\\.”表示本机，而“\pipe\MyControlPipe”则是路径名。如果把“.”换成主机名，那就可以是在另一台机器上的对象。参数 PIPE_ACCESS_DUPLEX 表示要建立的是“双工”、即可以双向通信的管道。另一个参数是一些标志位，表示要建立的管道采用“报文(message)”的方式、而不是字节流的方式读写，并且是阻塞方式、即有等待的读写。

CreateNamedPipeW()内部通过系统调用 NtCreateNamedPipeFile()完成命名管道的创建，这是 Windows 内核为命名管道提供的唯一的系统调用：

```
NTSTATUS STDCALL
NtCreateNamedPipeFile(PHANDLE FileHandle, ACCESS_MASK DesiredAccess,
            POBJECT_ATTRIBUTES ObjectAttributes, PIO_STATUS_BLOCK IoStatusBlock,
            ULONG ShareAccess, ULONG CreateDisposition, ULONG CreateOptions,
            ULONG NamedPipeType, ULONG ReadMode, ULONG CompletionMode,
            ULONG MaximumInstances, ULONG InboundQuota, ULONG OutboundQuota,
            PLARGE_INTEGER DefaultTimeout)
{
  NAMED_PIPE_CREATE_PARAMETERS Buffer;

  . . . . . .
  if (DefaultTimeout != NULL)
  {
      Buffer.DefaultTimeout.QuadPart = DefaultTimeout->QuadPart;
      Buffer.TimeoutSpecified = TRUE;
  }
  else
  {
      Buffer.TimeoutSpecified = FALSE;
  }
  Buffer.NamedPipeType = NamedPipeType;
  Buffer.ReadMode = ReadMode;
  Buffer.CompletionMode = CompletionMode;
  Buffer.MaximumInstances = MaximumInstances;
  Buffer.InboundQuota = InboundQuota;
  Buffer.OutboundQuota = OutboundQuota;

  return IoCreateFile(FileHandle, DesiredAccess, ObjectAttributes, IoStatusBlock,
                      NULL, FILE_ATTRIBUTE_NORMAL, ShareAccess, CreateDisposition,
                      CreateOptions, NULL, 0, CreateFileTypeNamedPipe, (PVOID)&Buffer,
```

```
                0);
}
```

在 Windows 中，文件系统是 I/O 子系统的一部分，文件的创建是由 IoCreateFile()完成的，参数 CreateFileTypeNamedPipe 是个文件类型代码，它决定了所创建的是作为特殊文件的命名管道。至于 IoCreateFile()如何创建此种特殊文件，那已经不在本文要讨论的范围内。

一般，命名管道是由“服务端”线程创建的，创建以后还要对其调用一个 Win32 API 库函数 ConnectNamedPipe()，以等待“客户端”线程打开这个命名管道、从而使双方建立起点对点的连接(注意这里的 Connect 与 Socket 机制中的 Connect 意思完全不同)。

```
BOOL STDCALL
ConnectNamedPipe(HANDLE hNamedPipe,
          LPOVERLAPPED lpOverlapped)
{
  PIO_STATUS_BLOCK IoStatusBlock;
  IO_STATUS_BLOCK Iosb;
  HANDLE hEvent;
  NTSTATUS Status;

  if (lpOverlapped != NULL)
  {
      lpOverlapped->Internal = STATUS_PENDING;
      hEvent = lpOverlapped->hEvent;
      IoStatusBlock = (PIO_STATUS_BLOCK)lpOverlapped;
  }
  else
  {
      IoStatusBlock = &Iosb;
      hEvent = NULL;
  }

  Status = NtFsControlFile(hNamedPipe, hEvent, NULL, NULL, IoStatusBlock,
                                  FSCTL_PIPE_LISTEN, NULL, 0, NULL, 0);
  if ((lpOverlapped != NULL) && (Status == STATUS_PENDING))
    return TRUE;

  if ((lpOverlapped == NULL) && (Status == STATUS_PENDING))
  {
      Status = NtWaitForSingleObject(hNamedPipe, FALSE, NULL);
      if (!NT_SUCCESS(Status))
      {
        SetLastErrorByStatus(Status);
        return FALSE;
      }
```

```
        Status = Iosb.Status;
    }

    if ((!NT_SUCCESS(Status) && Status != STATUS_PIPE_CONNECTED) ||
        (Status == STATUS_PENDING))
    {
        SetLastErrorByStatus(Status);
        return FALSE;
    }
    return TRUE;
}
```

实际的操作是由 NtFsControlFile()完成的，而 NtFsControlFile()有点像 Linux 中的 fcntl()，常常用来实现除标准的“读”、“写”等等以外的许多不同操作。

“客户端”线程通过打开文件建立与“服务端”线程的连接，所打开的就是作为特殊文件的命名管道。Windows 没有特别为此提供系统调用，而是直接利用 NtOpenFile()，例如：

```
    Status = NtOpenFile(&FileHandle, FILE_GENERIC_READ, &ObjectAttributes, &Iosb,
                        FILE_SHARE_READ | FILE_SHARE_WRITE,
                        FILE_SYNCHRONOUS_IO_NONALERT);
```

这里使用的参数 FILE_SYNCHRONOUS_IO_NONALERT 表示对文件的访问将是同步的、也即阻塞的。要打开的文件名在数据结构 ObjectAttributes 中，所以这里看不见。当然，这文件名必须与“服务端”所使用的相同。由此可见，“客户端”线程不必知道这是一个命名管道，甚至不必知道这是特殊文件，而只把它当成一般的文件看待。

在最简单的情况下，这就已经够了，下面就可以通过已经建立的连接进行通信了。具体的操作在形式上就跟普通文件的读/写一样。在读普通文件时，当事线程也会被阻塞(如果要读的内容不是已经在缓冲区中)，此时它所等待的主要是磁盘完成其物理的读出过程，而磁盘完成其读出过程以后的中断则会解除它的阻塞，相当于执行了一次 V 操作。相比之下，对于命名管道，启动读操作的线程也会被阻塞(如果尚无已经到达的数据或报文)，而此时等待的是管道的对方线程往管道中写数据。对方往管道中写数据，就解除了等待方线程的阻塞，实质上构成一次 V 操作。所以，普通文件的读/写同样也包含了同步机制，只不过那一般是进程与外部设备(或外部物理过程)之间的同步，而命名管道所需的是进程与进程之间的同步，但是这并没有本质上的不同。事实上，读者以后会看倒，在设备驱动程序中常常要用到 NtWaitForSingleObject()之类的等待机制，而且这正是设备驱动得以实现的基础。

一般而言，如果在管道中尚无数据的时候启动读操作，当事线程就会被阻塞。但是，为避免被阻塞，也可以先通过 PeekNamedPipe()查询一下是否已经有报文到达。

```
BOOL STDCALL
PeekNamedPipe(HANDLE hNamedPipe, LPVOID lpBuffer,
              DWORD nBufferSize, LPDWORD lpBytesRead,
              LPDWORD lpTotalBytesAvail, LPDWORD lpBytesLeftThisMessage)
{
    PFILE_PIPE_PEEK_BUFFER Buffer;
```

```
    IO_STATUS_BLOCK Iosb;
    ULONG BufferSize;
    NTSTATUS Status;

    BufferSize = nBufferSize + sizeof(FILE_PIPE_PEEK_BUFFER);
    Buffer = RtlAllocateHeap(RtlGetProcessHeap(),0, BufferSize);

    Status = NtFsControlFile(hNamedPipe, NULL, NULL, NULL, &Iosb,
                                    FSCTL_PIPE_PEEK, NULL, 0, Buffer, BufferSize);
    if (Status == STATUS_PENDING)
    {
        Status = NtWaitForSingleObject(hNamedPipe, FALSE, NULL);
        if (NT_SUCCESS(Status)) Status = Iosb.Status;
    }
    if (Status == STATUS_BUFFER_OVERFLOW)
    {
        Status = STATUS_SUCCESS;
    }

    . . . . . .

    if (lpTotalBytesAvail != NULL)
    {
        *lpTotalBytesAvail = Buffer->ReadDataAvailable;
    }
    if (lpBytesRead != NULL)
    {
        *lpBytesRead = Iosb.Information - sizeof(FILE_PIPE_PEEK_BUFFER);
    }
    if (lpBytesLeftThisMessage != NULL)
    {
        *lpBytesLeftThisMessage = Buffer->MessageLength -
                    (Iosb.Information - sizeof(FILE_PIPE_PEEK_BUFFER));
    }
    if (lpBuffer != NULL)
    {
        memcpy(lpBuffer, Buffer->Data,
                    min(nBufferSize, Iosb.Information - sizeof(FILE_PIPE_PEEK_BUFFER)));
    }
    RtlFreeHeap(RtlGetProcessHeap(),0, Buffer);

    return(TRUE);
}
```

同样，实际的查询是由 NtFsControlFile()完成的，只是用了另一个“命令码”。

除命名管道外，还有“信槽(MailSlot)”也是类似的特殊文件，只是信槽所提供的是无连接的通信机制。一般把这样无连接的通信称为“数据报”机制，而命名管道所提供的有连接通信则称为“虚电路”机制。“数据报”机制所提供的通信是“尽力传递”而不是“保证传递”，所以是不可靠的。此外，信槽所提供的通信不一定是点对点、也可以是广播式的。和命名管道一样，Windows 为信槽也只提供了一个系统调用，即 NtCreateMailslotFile()。而且它的内部也只是调用 IoCreateFile()，不同的只是作为参数的文件类型为 CreateFileTypeMailslot。既然这么相似，这里就不多说了。

6. 本地过程调用(LPC)

“本地过程调用”是建立在“端口(Port)”基础上的一种 Client/Server 结构的跨进程过程调用机制。而 Port 则是类似于 Unix 域 Socket 那样的进程间通信机制。离开了进程间通信，Client/Server 结构就无从谈起。所以 Port 机制是基础，而 LPC 是建立在此种基础上的应用。

实际上，前面所讲的各种机制都是单项的手段，严格说来都不构成高效的进程间通信。例如共享内存区提供了传递数据的手段，却不具备进程间同步的功能；而信号量、互斥门、事件三者都是进程间同步的手段，却又缺少了传递数据的功能。所以，真要有高效的进程间通信，就得把这两方面结合起来。而“端口(Port)”恰恰就是把数据传递和进程间同步结合起来、集成在一起而形成的高效率进程间通信机制。

按说 Port 才是高效的进程间通信机制，可是 Microsoft 却又偏偏不把它归入进程间通信，也不作为进程间通信的手段在 Win32 API 上向应用软件提供，而是使它的应用局限于跨进程的过程调用。当然，本地过程调用本身确实不是进程间通信，而是进程间通信的一种应用。不过 Windows 内核自身倒是在利用 Port 作为进程间通信的手段。例如每个进程实际上都有一个 Debug 端口和一个 Exception 端口，在程序执行的某些点上、或者在发生异常时，往往会通过这两个端口给有关的进程发送信息。

由于 LPC 机制的复杂性，笔者将另撰专文予以介绍，这里就不多说了。

7. 报文(Message)

报文(Message)是一种用于“视窗(Window)”的线程间通信机制。在早期的 Windows 系统中，Message 是实现于用户空间的“应用层”进程间通信机制。后来用户空间的许多基础设施都移到了内核中，成为模块 Win32k.sys，并为此而对 Windows 的系统调用界面进行了扩充。这一扩充可真有点喧宾夺主，增加了六百多个“扩展系统调用”，而原来“正宗”的只不过是二百多个。这六百多个新增的系统调用大部分都是用于图形操作，但是也有用于 Message 机制的系统调用，包括 NtUserGetMessage()、NtUserWaitMessage()、NtUserSendMessage()、NtUserPostMessage()等等。相应地，在 Win32 API 界面上则提供了 GetMessage()、WaitMessage()、SendMessage()、PostMessage()等等库函数。这些库函数都在 user32.dll 中。

以前说过，每个线程在内核中都有个 ETHREAD 数据结构，此外还有个因 Win32k 而来的 W32THREAD 数据结构，其定义为：

```
typedef struct _W32THREAD
{
  PVOID MessageQueue;
  FAST_MUTEX WindowListLock;
  LIST_ENTRY WindowListHead;
```

```
    LIST_ENTRY W32CallbackListHead;
    struct _KBDTABLES* KeyboardLayout;
    struct _DESKTOP_OBJECT* Desktop;
    HANDLE hDesktop;
    DWORD MessagePumpHookValue;
    BOOLEAN IsExiting;
} W32THREAD, *PW32THREAD;
```

这个数据结构的第一个成分就是 MessageQueue，就是用于 Message 传递的。

其实，要说 Windows 的各种进程间/线程间通信机制的“知名度”，恐怕莫过于这 Message 机制了。这是因为 Windows 应用程序的核心就是一个接收 Message、处理 Message 的循环。典型 Windows 应用程序的基本结构如下：

```
int WINAPI
WinMain(HINSTANCE hInst, HINSTANCE hPrevInstance,
                        LPSTR lpszCmdLine, int nCmdShow)
{
    . . . . . .
    RegisterClassEx(&wndclass);

    /* Create main window */
    hwndMain = CreateWindow(szFrameClass, szAppTitle, dwStyle, xPos, yPos,
                                    xWidth, yHeight, 0, 0, hInstance, NULL);
    ShowWindow(hwndMain, nCmdShow);
    UpdateWindow(hwndMain);

    while (GetMessage(&msg, NULL, 0, 0) != FALSE)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return(msg.wParam);
}
```

这里的 GetMessage()就是用于 Message 接收的库函数，其内部就是通过扩展系统调用 NtUserGetMessage()实现的。而 Windows 应用程序的核心就是这个 while 循环。

不过，Message 机制是个不小的话题，也需要有专文介绍，所以这里就不多说了。