漫谈兼容内核之十五:
Windows 线程的等待/唤醒机制

毛德操

对于任何一个现代的操作系统，进程间通信都是不可或缺的。

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各自的用户空间，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。所以，共享内存区天然就是一种进程间通信机制。但是这又是很原始的手段，因为这里有个读出方如何知道共享区的内容已经被写入方改变的问题。轮询，或者定期轮询，当然也是个办法，但是一般而言效率毕竟太低。所以，这里需要有个能够对通信双方的活动加以有效协调的机制，这就是“进程间同步”机制。进程间同步本身也是一种进程间通信(因为涉及信息的交换)，当然也是一种原始的进程间通信，但同时又是更高级的进程间通信机制的基石。

所以，在谈论通信机制之前，应该先考察一下进程间同步机制。在 Linux 中，这就是进程的睡眠/唤醒机制，或者说阻塞/解阻塞机制，体现为信息的接收方(进程)在需要读取信息、而发送方(进程)尚未向其发送之时就进入睡眠，到发送方向其发送信息时则加以唤醒。在 Windows 中，这个过程的原理是一样的，只是名称略有不同，称为“等待/唤醒”，表现形式上也有些不同。

在 Windows 中，进程间通信必须凭籍着某个已打开的“对象(Object)”才能发生(其实 Linux 中也是一样，只是没有统一到“对象”这个概念上)。我们不妨把这样的对象想像成某类货品的仓库，信息的接受方试图向这个仓库领货。如果已经到货，那当然可以提了就走，但要是尚未到货就只好等待，到一边歇着去(睡眠)，直至到了货才把它(唤醒)叫回来提货。Windows 专门为此过程提供了两个系统调用，一个是 NtWaitForSingleObject()，另一个是 NtWaitForMultipleObjects()。后者是前者的推广、扩充，使得一个线程可以同时在多个对象上等待。

于是，在 Windows 应用程序中，当一个线程需要从某个对象“提货”、即获取信息时，就通过系统调用 NtWaitForSingleObject()实现在目标对象上的等待，当前线程因此而被“阻塞”、即进入睡眠状态，直至所等待的条件得到满足时才被唤醒。

```
NTSTATUS    STDCALL
NtWaitForSingleObject(IN HANDLE ObjectHandle,
                      IN BOOLEAN Alertable,
                      IN PLARGE_INTEGER TimeOut    OPTIONAL)
{
    . . . . . .
    PreviousMode = ExGetPreviousMode();

    if(TimeOut != NULL && PreviousMode != KernelMode)
    {
      _SEH_TRY
      {
        ProbeForRead(TimeOut, sizeof(LARGE_INTEGER), sizeof(ULONG));
```

```
            /* make a copy on the stack */
            SafeTimeOut = *TimeOut;
            TimeOut = &SafeTimeOut;
        }
        _SEH_HANDLE
        {
            Status = _SEH_GetExceptionCode();
        }
        _SEH_END;

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

    Status = ObReferenceObjectByHandle(ObjectHandle, SYNCHRONIZE, NULL,
                                       PreviousMode, &ObjectPtr, NULL);
    . . . . . .
    if (!KiIsObjectWaitable(ObjectPtr))
        {
            DPRINT1("Waiting for object type '%wZ' is not supported\n",
                &BODY_TO_HEADER(ObjectPtr)->ObjectType->TypeName);
            Status = STATUS_HANDLE_NOT_WAITABLE;
        }
    else
        {
            Status = KeWaitForSingleObject(ObjectPtr, UserRequest,
                                           PreviousMode, Alertable, TimeOut);
        }

    ObDereferenceObject(ObjectPtr);
    return(Status);
}
```

参数 ObjectHandle 和 TimeOut 的作用不言自明。另一个参数 Alertable 是个布尔量，表示是否允许本次等待因用户空间 APC 而中断，或者说被“警醒”。警醒与唤醒是不同的，唤醒是因为所等待的条件得到了满足(仓库到了货)，而警醒是因为别的原因(与仓库无关)。

我们知道，Windows 的系统调用函数既可以从用户空间通过自陷指令 int 0x2e 加以调用，也可以在内核中直接加以调用。如果是从用户空间调用，而且又有以指针形式传递的参数，那就需要从用户空间读取这些指针所指的内容。但是，这些指针所指处的(虚存)页面是否有映射呢？这是没有保证的。如果没有映射，那么在访问时就会发生“页面错误”异常。另一方面，既然读不到调用参数，原定的操作也就无法继续下去了。为此，代码中把对于目标是否可读的测试 ProbeForRead()以及参数内容的复制放在_SEH_TRY{}中，并且设置好“页面

错误”异常处理的向量，使得一旦发生“页面错误”异常就执行_SEH_HANDLE{}中的操作。这是 Windows 的“结构化出错处理”即 SHE 机制的一部分，以后还要有专文介绍。由于篇幅的关系，以后在系统调用的程序中就不再列出这些代码了。

NtWaitForSingleObject()中实质性的操作只有两个。一是 ObReferenceObjectByHandle()，就是通过已打开对象的 Handle 获取指向该目标对象(数据结构)的指针。第二个操作就是 KeWaitForSingleObject()，这是下面要讲的。不过，并非对于所有的对象都可以执行这个函数，有的对象是“可等待”的，有的对象却是“不可等待”的，所以先要通过一个函数 KiIsObjectWaitable()加以检验。这样，一言以蔽之，NtWaitForSingleObject()的作用就是对可等待目标对象的数据结构执行 KeWaitForSingleObject()。

那么什么样的对象才是可等待的呢？看一下这个函数的代码就知道了：

```
BOOL inline FASTCALL    KiIsObjectWaitable(PVOID Object)
{
    POBJECT_HEADER Header;
    Header = BODY_TO_HEADER(Object);

    if (Header->ObjectType == ExEventObjectType ||
       Header->ObjectType == ExIoCompletionType ||
       Header->ObjectType == ExMutantObjectType ||
       Header->ObjectType == ExSemaphoreObjectType ||
       Header->ObjectType == ExTimerType ||
       Header->ObjectType == PsProcessType ||
       Header->ObjectType == PsThreadType ||
       Header->ObjectType == IoFileObjectType) {
          return TRUE;
    } else {
          return FALSE;
    }
}
```

可见，所谓“可等待”的对象包括进程、线程、Timer、文件，以及用于进程间通信的对象 Event、Mutant、Semaphore，还有用于设备驱动的 IoCompletion。这 IoCompletion 属于设备驱动框架，所以 KeWaitForSingleObject()既是进程间通信的重要一环，同时也是设备驱动框架的一个重要组成部分。

注意这里(取自 ReactOS)关于对象数据结构的处理是很容易让人摸不着头脑的，因而需要加一些说明。首先，每个进程的“打开对象表”是由 Handle 表项构成的，是一个 HANDLE_TABLE_ENTRY 结构指针数组。而 HANDLE_TABLE_ENTRY 数据结构中有个指针指向另一个数据结构(而且这个指针的低位又被用于一些标志位)，可是这个数据结构并非具体对象的数据结构，而是一个通用的 OBJECT_HEADER 数据结构：

```
typedef struct _OBJECT_HEADER
/*
 * PURPOSE: Header for every object managed by the object manager
 */
```

```
{
    UNICODE_STRING Name;
    LIST_ENTRY Entry;
    LONG RefCount;
    LONG HandleCount;
    BOOLEAN Permanent;
    BOOLEAN Inherit;
    struct _DIRECTORY_OBJECT* Parent;
    POBJECT_TYPE ObjectType;
    PSECURITY_DESCRIPTOR SecurityDescriptor;

    /*
     * PURPOSE: Object type
     * NOTE: This overlaps the first member of the object body
     */
    CSHORT Type;

    /*
     * PURPOSE: Object size
     * NOTE: This overlaps the second member of the object body
     */
    CSHORT Size;
} OBJECT_HEADER, *POBJECT_HEADER;
```

紧随在 OBJECT_HEADER 后面的才是具体对象的数据结构的正身、即 Body。所以 OBJECT_HEADER 和 Body 合在一起才构成一个对象的完整的数据结构。但是，当传递一个对象的数据结构指针时，所传递的指针却既不是指向其正身，又不是指向其 OBJECT_HEADER，而是指向其 OBJECT_HEADER 结构中的字段 Type。宏定义 HEADER_TO_BODY 说明了这一点：

```
#define HEADER_TO_BODY(objhdr)   \
   (PVOID)((ULONG_PTR)objhdr + sizeof(OBJECT_HEADER) \
                             - sizeof(COMMON_BODY_HEADER))
```

就是说，具体对象数据结构的起点是 objhdr 加上 OBJECT_HEADER 的大小、再减去 COMMON_BODY_HEADER 的大小。而 COMMON_BODY_HEADER 定义为：

```
typedef struct
{
    CSHORT Type;
    CSHORT Size;
} COMMON_BODY_HEADER, *PCOMMON_BODY_HEADER;
```

显然这就是 OBJECT_HEADER 中的最后两个字段。那么具体对象的数据结构又是什么

样的呢？我们以 Semaphore 的数据结构 KSEMAPHORE 为例：

```
typedef struct _KSEMAPHORE {
    DISPATCHER_HEADER Header;
    LONG Limit;
} KSEMAPHORE;
```

它的第一个成分是一个 DISPATCHER_HEADER 数据结构，但是却看不到 Type 和 Size 这两个字段，也看不到 COMMON_BODY_HEADER。我们进一步看 DISPATCHER_HEADER 的定义：

```
typedef struct _DISPATCHER_HEADER {
  UCHAR    Type;
  UCHAR    Absolute;
  UCHAR    Size;
  UCHAR    Inserted;
  LONG    SignalState;
  LIST_ENTRY    WaitListHead;
} DISPATCHER_HEADER, *PDISPATCHER_HEADER;
```

与 COMMON_BODY_HEADER 相比较，我们确实看到这里有 Type 和 Size，但是中间却又夹着别的字段。但是，仔细观察，就可看出在 COMMON_BODY_HEADER 中 Type 的类型是 16 位的 CSHORT，而在这里是 8 位的 UCHAR，而且下面 Absolute 的类型也是 UCHAR。这就清楚了，原来 COMMON_BODY_HEADER 中(以及 OBJECT_HEADER 中)的 Type 虽然是 16 位的，实际上却只用了其低 8 位，而在 DISPATCHER_HEADER 中则将其高 8 位用作 Absolute。编译器在分配空间时是由低到高(地址)分配的，所以 Type 是低 8 位而 Absolute 是高 8 位。同样的道理也适用于 Size 和 Inserted。这样的安排当然使代码的可读性变得很差，笔者尚不明白为什么非得要这么干。另一方面，不同的对象有不同的数据结构，所以代码中有关对象指针的类型一般总是 PVOID，这似乎也合理。但是既然第一个成分总是 DISPATCHER_HEADER，那为什么不用 PDISPATCHER_HEADER 呢？那样至少也可以改善一些可读性。

回到 NtWaitForSingleObject()的代码，我们需要进一步往下看 KeWaitForSingleObject()的代码。不过在此之前先得考察一下有关的数据结构。

首先，执行这个函数的主体是个线程，而所等待的又是通过一个对象传递的信息，就一定要有个数据结构把这二者连系起来，这就是 KWAIT_BLOCK 数据结构：

```
typedef struct _KWAIT_BLOCK
/*
 * PURPOSE: Object describing the wait a thread is currently performing
 */
{
    LIST_ENTRY WaitListEntry;
    struct _KTHREAD* Thread;
    struct _DISPATCHER_HEADER *Object;
```

```
    struct _KWAIT_BLOCK* NextWaitBlock;
    USHORT WaitKey;
    USHORT WaitType;
} KWAIT_BLOCK, *PKWAIT_BLOCK;
```

这里的 Thread 和 Object 都是指针。前者指向一个 KTHREAD 数据结构，代表着正在等待的线程；后者指向一个对象的数据结构，虽然指针的类型是 DISPATCHER_HEADER*，但是如上所述这是不管什么对象的数据结构中的第一个成分，所以指向这个数据结构也就是指向了它所在对象的数据结构。此外，结构中的成分 WaitListEntry 显然是用来把这个数据结构挂入某个(双链)队列的，同时指针 NextWaitBlock 也是用来维持一个(单链)队列。这是因为一个“等待块”即 KWAIT_BLOCK 数据结构可能同时出现在两个队列中。首先，多个线程可能在同一个对象上等待，每个线程为此都有一个等待块，从而形成特定目标对象的等待队列，这就是由 WaitListEntry 维持的队列。这样，对于一个具体的对象而言，其等待队列中的每个等待块都代表着一个线程。同时，一个线程又可能同时在多个对象上等待，因而又可能有多个等待块。对于这个线程而言，每个等待块都代表着一个不同的对象，这些等待块则通过 NextWaitBlock 构成一个队列。其余字段的作用以后就会明白。

既然等待是具体线程的行为，线程数据结构中就得有相应的安排，KTHREAD 结构中与此有关的成分如下：

```
typedef struct _KTHREAD
{
    /* For waiting on thread exit */
    DISPATCHER_HEADER DispatcherHeader;       /* 00 */
    . . . . . .
    LONG                 WaitStatus;          /* 50 */
    KIRQL                WaitIrql;            /* 54 */
    CHAR                 WaitMode;            /* 55 */
    UCHAR                WaitNext;            /* 56 */
    UCHAR                WaitReason;          /* 57 */
    PKWAIT_BLOCK         WaitBlockList;       /* 58 */
    LIST_ENTRY           WaitListEntry;       /* 5C */
    ULONG                WaitTime;            /* 64 */
    CHAR                 BasePriority;        /* 68 */
    UCHAR                DecrementCount;      /* 69 */
    UCHAR                PriorityDecrement;   /* 6A */
    CHAR                 Quantum;             /* 6B */
    KWAIT_BLOCK          WaitBlock[4];        /* 6C */
    PVOID                LegoData;            /* CC */
    . . . . . .

} KTHREAD;
```

首先我们注意到这里有个结构数组 WaitBlock[4]，这就是 KWAIT_BLOCK 数据结构座落所在，需要时就在这里就地取材。之所以是个数组，是因为有时候需要同时在多个对象上

等待，这就是 KeWaitForMultipleObjects()的目的，有点类似于 Linux 中的 select()。此时 KWAIT_BLOCK 指针 WaitBlockList 指向本线程的等待块队列。如前所述，这个队列中的每个等待块都代表着一个对象。WaitStatus 则是状态信息，在结束等待时反映着结束的原因。

下面我们就来看 KeWaitForSingleObject()的代码。

[NtWaitForSingleObject() > KeWaitForSingleObject()]

```
NTSTATUS STDCALL
KeWaitForSingleObject(PVOID Object,
                      KWAIT_REASON WaitReason,
                      KPROCESSOR_MODE WaitMode,
                      BOOLEAN Alertable,
                      PLARGE_INTEGER Timeout)
{
    PDISPATCHER_HEADER CurrentObject;
    PKWAIT_BLOCK WaitBlock;
    PKWAIT_BLOCK TimerWaitBlock;
    PKTIMER ThreadTimer;
    PKTHREAD CurrentThread = KeGetCurrentThread();
    NTSTATUS Status;
    NTSTATUS WaitStatus;

    . . . . . .
    /* Check if the lock is already held */
    if (CurrentThread->WaitNext)    {

        /* Lock is held, disable Wait Next */
        DPRINT("Lock is held\n");
        CurrentThread->WaitNext = FALSE;
    } else {
        /* Lock not held, acquire it */
        DPRINT("Lock is not held, acquiring\n");
        CurrentThread->WaitIrql = KeAcquireDispatcherDatabaseLock();
    }

    /* Start the actual Loop */
    do {

        /* Get the current Wait Status */
        WaitStatus = CurrentThread->WaitStatus;

        /* Append wait block to the KTHREAD wait block list */
        CurrentThread->WaitBlockList = WaitBlock = &CurrentThread->WaitBlock[0];
```

```
/* Get the Current Object */
CurrentObject = (PDISPATCHER_HEADER)Object;

/* FIXME:
* Temporary hack until my Object Manager re-write.. . . . . .
*/
if (CurrentObject->Type == IO_TYPE_FILE) {
      . . . . . .
}

/* Check if the Object is Signaled */
if (KiIsObjectSignaled(CurrentObject, CurrentThread)) {

      /* Just unwait this guy and exit */
      if (CurrentObject->SignalState != MINLONG) {

            /* It has a normal signal state, so unwait it and return */
            KiSatisfyObjectWait(CurrentObject, CurrentThread);
            Status = STATUS_WAIT_0;
            goto WaitDone;

      } else {

            /* Is this a Mutant? */
            if (CurrentObject->Type == MutantObject) {

                  /* According to wasm.ru, we must raise this exception (tested and true) */
                  KeReleaseDispatcherDatabaseLock(CurrentThread->WaitIrql);
                  ExRaiseStatus(STATUS_MUTANT_LIMIT_EXCEEDED);
            }
      }
}

/* Set up the Wait Block */
WaitBlock->Object = CurrentObject;
WaitBlock->Thread = CurrentThread;
WaitBlock->WaitKey = (USHORT)(STATUS_WAIT_0);
WaitBlock->WaitType = WaitAny;
WaitBlock->NextWaitBlock = NULL;

/* Make sure we can satisfy the Alertable request */
KiCheckAlertability(Alertable, CurrentThread, WaitMode, &Status);

/* Set the Wait Status */
```

```
        CurrentThread->WaitStatus = Status;

        /* Enable the Timeout Timer if there was any specified */
        if (Timeout != NULL) {
            . . . . . .
        }

        /* Link the Object to this Wait Block */
        InsertTailList(&CurrentObject->WaitListHead, &WaitBlock->WaitListEntry);

        /* Handle Kernel Queues */
        if (CurrentThread->Queue) {

            DPRINT("Waking Queue\n");
            KiWakeQueue(CurrentThread->Queue);
        }

        /* Block the Thread */
        . . . . . .
        PsBlockThread(&Status, Alertable, WaitMode, (UCHAR)WaitReason);

        /* Check if we were executing an APC */
        if (Status != STATUS_KERNEL_APC) {

            /* Return Status */
            return Status;
        }

        DPRINT("Looping Again\n");
        CurrentThread->WaitIrql = KeAcquireDispatcherDatabaseLock();

    } while (TRUE);

WaitDone:
    /* Release the Lock, we are done */
    . . . . . .
    KeReleaseDispatcherDatabaseLock(CurrentThread->WaitIrql);
    return Status;
}
```

为简单起见，我们略去了代码中对于超时(Timeout)的设置，也略去了原作者所加的一大段注释，说日后还要对代码进行改写云云。此外，还略去了当目标对象的类型为 IO_TYPE_FILE 时的处理，因为我们现在还没有深入到设备驱动中。

代码中首先检查 CurrentThread->WaitNext 是否为 0，以决定在本次调用中是否需要通过

KeAcquireDispatcherDatabaseLock()来“锁定”线程调度，即禁止因别的原因(例如中断)而引起的线程调度。注意这里锁定线程调度是必要的，CurrentThread->WaitNext 为 TRUE 只是说明原来就已经锁定，不应该再来锁定了。有关线程调度及其锁定以后在别的漫谈中还会讲到。

然后就是程序的主体了。这是一个 do{}while(TRUE)无限循环，这循环一共有 3 个出口，其中之一与 Timeout 有关，由于有关的代码已被略去，在这里已经看不见了。剩下的两个出口，一个是条件语句“if (KiIsObjectSignaled(CurrentObject, CurrentThread))”里面的“goto WaitDone”，这说明目标对象已经收到“信号”、或者说此前已经“到货”，所以不需要等待了。比方说，假定调用 KeWaitForSingleObject()的目的是接收一个报文，但是这个报文在此之前就已到达了，那当然就不用再睡眠等待，而可以立即就返回。不过在返回之前要通过 KiSatisfyObjectWait()把账销掉。另一个就是 PsBlockThread()后面条件语句“if (Status != STATUS_KERNEL_APC)”里面的返回语句。PsBlockThread()是 KeWaitForSingleObject()中最本质的操作。正是这个函数，在一般的情况下，将当前线程置于被阻塞(睡眠)状态并启动线程调度，直至当前线程因为所等待的条件得到满足而被唤醒，才从这个函数返回。但是也有例外，就是因为有 APC 请求而被警醒，此时被警醒的线程会执行 APC 函数，执行完以后就立即返回，并把 Status 的返回值设置成 STATUS_KERNEL_APC。在这种情况下当然要再次执行 PsBlockThread()，这就是为什么要把 PsBlockThread()放在一个循环中的原因。

再看阻塞当前进程之前的准备工作。这里有两个重点。一个是对 KiIsObjectSignaled()的调用、以及根据其结果作出的反应，这前面已经讲过了。另一个是对 KiCheckAlertability()的调用。在调用 NtWaitForSingleObject()的时候有个参数 Alertable，表示是否允许本次等待因 APC 请求而中断，或者说“被警醒”。这个参数一路传了下来。

如果一个线程的睡眠/等待状态是“可警醒”的，那么：

- 别的线程可以通过系统调用 NtAlertThread()将其警醒。
- APC 请求的到来也可以将其警醒。

读者也许会想，既然当前线程已经睡眠，这 APC 请求从何而来呢？其实很简单。例如，一个线程可能通过系统调用 NtWriteFile()启动了一次异步的文件操作，由于是异步操作，这个系统调用很快就返回了，然后这线程就因为进程间通信而辗转地进入了 KeWaitForSingleObject()、并因此而被阻塞进入了睡眠。然而，这个进程所启动的异步操作仍在进行，当这异步操作最终完成时，内核会把预定的 APC 函数挂入这个线程的 APC 请求队列。这时候，目标线程的睡眠之是否可警醒，就显然会有不一样的效果。这里 KiCheckAlertability()的作用是结合参数 Alertable 的值检查当前线程是否已经受到警醒，以及是否已经有 APC 请求，并依此进行必要的状态设置。

再看对于 KWAIT_BLOCK 数据结构的设置，其中的字段 WaitType 设置成 WaitAny，与此相对的是 WaitAll。在对于多个对象的等待中，前者表示其中只要有任何一个对象满足条件即可，后者则表示必须全部满足条件。不过 KeWaitForSingleObject()所等待的是单一对象，所以二者其实并无不同，只是按编程的约定采用 WaitAny。另一个字段 WaitKey 实际上就是当前状态，STATUS_WAIT_0 表示正常的等待。

接着就通过 InsertTailList()把已经准备好的 KWAIT_BLOCK 数据结构插入目标对象的 WaitListHead 队列。这样，一旦目标对象的状态发生变化，就可以顺着这个队列找到所有在这个对象上等待的线程。

下面对 CurrentThread->Queue 的检查和对于 KiWakeQueue()的调用与设备驱动有关，而不在我们此刻关心的范围内，所以也把它跳过。

再往下就是调用 PsBlockThread()使当前线程进入睡眠了。

[NtWaitForSingleObject() > KeWaitForSingleObject() > PsBlockThread()]

```
VOID
STDCALL
PsBlockThread(PNTSTATUS Status,
              UCHAR Alertable,
              ULONG WaitMode,
              UCHAR WaitReason)
{
    PKTHREAD Thread = KeGetCurrentThread();
    PKWAIT_BLOCK WaitBlock;

    if (Thread->ApcState.KernelApcPending) {

        DPRINT("Dispatching Thread as ready (APC!)\n");

        /* Remove Waits */
        WaitBlock = Thread->WaitBlockList;
        while (WaitBlock) {
            RemoveEntryList (&WaitBlock->WaitListEntry);
            WaitBlock = WaitBlock->NextWaitBlock;
        }
        Thread->WaitBlockList = NULL;

        /* Dispatch it and return status */
        PsDispatchThreadNoLock (THREAD_STATE_READY);
        if (Status != NULL) *Status = STATUS_KERNEL_APC;

    } else {

        /* Set the Thread Data as Requested */
        DPRINT("Dispatching Thread as blocked\n");
        Thread->Alertable = Alertable;
        Thread->WaitMode = (UCHAR)WaitMode;
        Thread->WaitReason = WaitReason;

        /* Dispatch it and return status */
        PsDispatchThreadNoLock(THREAD_STATE_BLOCKED);
        if (Status != NULL) *Status = Thread->WaitStatus;
    }

    DPRINT("Releasing Dispatcher Lock\n");
    KfLowerIrql(Thread->WaitIrql);
}
```

这个函数没有什么特殊之处，值得一说的是它会检查是否有 APC 请求在等待执行，如果有的话就退出等待，不进入睡眠了。PsDispatchThreadNoLock()的作用是线程调度，作为参数传递给它的 THREAD_STATE_READY 和 THREAD_STATE_BLOCKED 分别表示目标线程的新的状态。THREAD_STATE_READY 显然是“就绪”，也就是不进入睡眠。而 THREAD_STATE_BLOCKED 当然是“阻塞”、即睡眠。当然，如果进入睡眠的话，那就要到以后被唤醒(所等待的条件得到满足或超时)或警醒(APC 请求的到来或受别的线程警醒)才会从这个函数返回。在这里，我们假定当前进程被阻塞。

至此，当前线程、即等待“到货”(进程间信息的到来)的线程已被阻塞进入了睡眠。下面要看的是当进程间信息到来时怎样唤醒这个线程了。

在 Windows 内核中，与 KeWaitForSingleObject()相对的函数是 KiWaitTest()。前者使一个线程在一个(可等待)对象上等待而被阻塞进入睡眠，后者则唤醒在一个对象上等待的所有线程。

```
VOID    FASTCALL
KiWaitTest(PDISPATCHER_HEADER Object, KPRIORITY Increment)
{
    PLIST_ENTRY WaitEntry;
    PLIST_ENTRY WaitList;
    PKWAIT_BLOCK CurrentWaitBlock;
    PKWAIT_BLOCK NextWaitBlock;

    /* Loop the Wait Entries */
    DPRINT("KiWaitTest for Object: %x\n", Object);
    WaitList = &Object->WaitListHead;
    WaitEntry = WaitList->Flink;
    while ((WaitEntry != WaitList) && (Object->SignalState > 0)) {

        /* Get the current wait block */
        CurrentWaitBlock =
                CONTAINING_RECORD(WaitEntry, KWAIT_BLOCK, WaitListEntry);

        /* Check the current Wait Mode */
        if (CurrentWaitBlock->WaitType == WaitAny) {

            /* Easy case, satisfy only this wait */
            DPRINT("Satisfiying a Wait any\n");
            WaitEntry = WaitEntry->Blink;
            KiSatisfyObjectWait(Object, CurrentWaitBlock->Thread);
        } else {

            /* Everything must be satisfied */
            DPRINT("Checking for a Wait All\n");
            NextWaitBlock = CurrentWaitBlock->NextWaitBlock;
```

```
            /* Loop first to make sure they are valid */
            while (NextWaitBlock) {

                /* Check if the object is signaled */
                if (!KiIsObjectSignaled(Object, CurrentWaitBlock->Thread)) {

                    /* It's not, move to the next one */
                    DPRINT1("One of the object is non-signaled, sorry.\n");
                    goto SkipUnwait;
                }

                /* Go to the next Wait block */
                NextWaitBlock = NextWaitBlock->NextWaitBlock;
            }

            /* All the objects are signaled, we can satisfy */
            DPRINT("Satisfiying a Wait All\n");
            WaitEntry = WaitEntry->Blink;
            KiSatisifyMultipleObjectWaits(CurrentWaitBlock);
        }

        /* All waits satisfied, unwait the thread */
        DPRINT("Unwaiting the Thread\n");
        KiAbortWaitThread(CurrentWaitBlock->Thread,
                          CurrentWaitBlock->WaitKey, Increment);

SkipUnwait:
        /* Next entry */
        WaitEntry = WaitEntry->Flink;
    }

    DPRINT("Done\n");
}
```

这个函数是针对一个特定对象的，所以参数就是指向其数据结构的指针(这一次的类型倒是 DISPATCHER_HEADER 指针)。另一个参数 Increment 说明是否、以及怎样、对被唤醒线程的优先级作出调整。

程序的代码基本上就是一个 while 循环，这是对目标对象的等待队列中所有等待块即 KWAIT_BLOCK 数据结构的循环。额外的条件是目标对象数据结构中的 SignalState 字段必须大于 0，其原因和意义我们后面就会看到。如前所述，队列中的每个 KWAIT_BLOCK 数据结构都代表着一个因为在这个对象上等待而被阻塞的线程。另一方面，所涉及的每个线程又都可能同时在多个对象上等待，而等待的类型(方式)则有 WaitAny 和 WaitAll 两种。如果只是在单一的对象上等待，则我们在前面看到所用的是 WaitAny。

如果队列中某个进程的等待方式是 WaitAny，那么就对目标对象执行一次 KiSatisfyObjectWait()，然后就通过 KiAbortWaitThread()将其唤醒、使其结束等待。反之，要是这个线程的等待方式是 WaitAll，则要通过另一个 while 循环扫描该线程的等待块队列，通过 KiIsObjectSignaled()检查其所等待的每一个对象是否接受了“信号”(是否到了货)。只要其中有一个对象没有满足条件，这个线程就不满足被唤醒的条件，因而通过“goto SkipUnwait”语句跳过对此线程的处理。反之，如果所等待的所有对象都收到了“信号”，就对这些对象执行 KiSatisifyMultipleObjectWaits()，那实际上就是对所有这些对象都执行 KiSatisfyObjectWait()。

前面讲过，KiSatisfyObjectWait()的作用是把目标对象上已经接收到的“信号”消耗掉，或者说对已经到货、并且有了“货主”(因而马上就要被领走)的货物进行销账处理。怎么销呢？我们看代码：

[KiWaitTest() > 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--;
    }
}
```

表面上 KiSatisfyObjectWait()似乎是个通用的程序，可以适用于任何可等待对象，但是实际上它的内部还是按不同的可等待对象分别加以处理的。我们以“信号量(Semaphore)”为例来说明其操作方式。对于信号量对象，其数据结构中的 SignalState 表示可用的资源数量，或者可以理解为筹码的数量。每当资源的提供者向此对象提供一个筹码时，就使 SignalState 加 1，而每当资源的使用者消耗一个筹码时就使其减 1。由于此时目标对象已经得到了所要求的资源，SignalState 已经递加了计数，这里使其递减，就表示把所收到的资源消耗掉了。这也解释了前面 KiWaitTest()中的 while 循环为什么有个条件“Object->SignalState > 0”，因为如果 Object->SignalState 不大于 0 就说明资源已经耗尽，再继续循环就入不敷出了。

最后，把已经接收到的资源耗用掉以后，就要通过 KiAbortWaitThread()唤醒正在等待此项资源的线程。读者也许会想，既然接收到的资源已经耗用掉，那正在目标进程上等待的线程还能得到什么呢？其实，这里的所谓耗用一般只是“账面”上的，与此相伴随的还有“真金白银”(例如报文)，这才是等待中的线程所需要的。打个比方，目标对象得到“进货”以后，账面上就有了“库存”，而 KiSatisfyObjectWait()就是把账上的库存销掉，而实际的货物则有待于要货的线程被唤醒以后前去领取。这一点，读者在读了下一篇关于进程间通信的漫谈以后就会更加清楚。

我们接着看 KiAbortWaitThread()的代码。注意这程序名中的“Abort”是指等待状态的 Abort，而不是线程的 Abort。

[KiWaitTest() > KiAbortWaitThread()]

```
VOID    FASTCALL
KiAbortWaitThread(PKTHREAD Thread, NTSTATUS WaitStatus, KPRIORITY Increment)
{
    PKWAIT_BLOCK WaitBlock;

    /* If we are blocked, we must be waiting on something also */
    . . . . . .
    ASSERT((Thread->State == THREAD_STATE_BLOCKED) ==
                                            (Thread->WaitBlockList != NULL));
```

```
    /* Remove the Wait Blocks from the list */
    DPRINT("Removing waits\n");
    WaitBlock = Thread->WaitBlockList;
    while (WaitBlock) {
        /* Remove it */
        DPRINT("Removing Waitblock: %x, %x\n", WaitBlock, WaitBlock->NextWaitBlock);
        RemoveEntryList(&WaitBlock->WaitListEntry);

        /* Go to the next one */
        WaitBlock = WaitBlock->NextWaitBlock;
    };

    /* Check if there's a Thread Timer */
    if (Thread->Timer.Header.Inserted) {
        /* Cancel the Thread Timer with the no-lock fastpath */
        DPRINT("Removing the Thread's Timer\n");
        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 */
    DPRINT("Unblocking the Thread\n");
    PsUnblockThread((PETHREAD)Thread, &WaitStatus, 0);
}
```

这是一段简单直白的代码，如果略去对定时器(Timer)的处理以及对 Thread->Queue 的处理(如前所述，这与设备驱动有关)，那就只剩下两件事。第一件事是通过一个 while 循环将目标线程的所有等待块 WaitBlock 从其所在的队列中脱离出来。之所以如此是因为目标进程可能同时在多个对象上等待，因此通过不同的 WaitBlock 挂入了不同对象的队列。不过，对于因执行 KeWaitForSingleObject()而被阻塞的线程而言，实际上只挂入了一个对象的队列，所以 WaitBlock->NextWaitBlock 为 0。第二件事就是通过 PsUnblockThread()唤醒目标线程，对于熟悉 Linux 内核的读者这已是了无新意的事了。

但是，等待中的线程被唤醒或警醒以后怎么走，倒是值得一说的。为此我们又要回到 PsBlockThread()的代码中。在那里，睡眠中的目标线程被唤醒或警醒后从 PsDispatchThreadNoLock()返回，此时目标线程的 Thread->WaitStatus 记录着返回时的状态，实际上也反映了返回的原因。返回的原因无非就是这么几种：

- 条件得到满足而被唤醒，此时的 Thread->WaitStatus 为表示成功的状态码。
- 超时或出错，此时的 Thread->WaitStatus 为相应的出错代码。
- 别的线程对其执行了 NtAlertThread()系统调用，此时的 Thread->WaitStatus 为 STATUS_ALERTED。
- 因 APC 请求而警醒，此时的 Thread->WaitStatus 为 STATUS_KERNEL_APC。

从代码中可以看出，从 PsBlockThread()返回到 KeWaitForSingleObject()时，通过参数 Status 返回的值就是 Thread->WaitStatus。此外，也可能在 PsBlockThread()中因发现有 APC 请求存在而根本就没有进入睡眠，此时返回的 Status 也是 STATUS_KERNEL_APC。于是，当目标线程从 PsBlockThread()返回到 KeWaitForSingleObject()中时，如果 Status 为 STATUS_KERNEL_APC 就说明这是因为被 APC 请求警醒而返回的，原先的目的并未达到，还须再接再厉，所以才有 KeWaitForSingleObject()中的 do{}while(TRUE)循环。还要说明，由于 APC 机制的特殊安排，目标进程从睡眠时醒来时会先执行 APC 函数，然后才回到原来的轨道而从 PsDispatchThreadNoLock()返回到 PsBlockThread()。

最后，由于本文实际上也涉及警醒，而不仅仅是唤醒，这里也看一下系统调用 NtAlertThread()的代码：

```
NTSTATUS    STDCALL
NtAlertThread (IN HANDLE ThreadHandle)
{
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    PETHREAD Thread;
    NTSTATUS Status;

    /* 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;
}
```

当然，参数 ThreadHandle 代表着已被打开的目标线程。显然，这里实质性的操作是 KeAlertThread()。

[NtAlertThread() > KeAlertThread()]

```
BOOLEAN    STDCALL
KeAlertThread(PKTHREAD Thread, KPROCESSOR_MODE AlertMode)
{
    KIRQL OldIrql;
    BOOLEAN PreviousState;

    /* 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) {

            DPRINT("Aborting Wait\n");
            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;
}
```

这段代码就留给读者自己阅读了。