漫谈兼容内核之十八:
Windows 的 LPC 机制

毛德操

LPC 是“本地过程调用(Local Procedure Call)”的缩写。所谓“本地过程调用”是与“远程过程调用”即 RPC 相对而言的。其实 RPC 是广义的，RPC 可以发生在不同的主机之间，也可以发生在同一台主机上，发生在同一台主机上就是 LPC。所以在 Unix 语境下就没有 LPC 这一说，即使发生在同一台主机上也称为 RPC。在历史上，RPC 是“开放软件基金会(OSF)”设计和提出的一种用以实现“Unix 分布计算环境(Unix DCE)”的标准。实际上，微软的 DCOM 技术，就是建立在 RPC 基础上的。Win2000 的 RPC 可以采用 TCP/IP、SPX、NetBIOS、命名管道、以及“本地”作为底层的通信手段，这“本地”就是 LPC。另一方面，Windows 是一个带有许多微内核系统特征的操作系统(尽管它的内核不是微内核)，系统中有不少“系统级”的服务进程，例如大家已经熟知的 csrss、管理用户登录的“本地安全认证服务”进程 LSASS 等等，用户进程以及微软提供的系统工具软件经常需要调用由这些服务进程提供的服务，这里 LPC 就起着重要的作用。

LPC 的基础是一种称为“端口(Port)”的进程间通信机制，类似于本地的(Unix 域的)Socket。这种 Port 机制提供了面向报文传递(message passing)的进程间通信，而 LPC 则是建立在这个基础上的高层机制，目的是提供跨进程的过程调用。注意这里所谓“跨进程的过程调用”不同于以前所说的“跨进程操作”。前者是双方有约定、遵循一定规程的、有控制的服务提供，被调用者在向外提供一些什么服务、即提供哪些函数调用方面是自主的，而后者则可以是在不知不觉之间的被利用、被操纵。前者是良性的，而后者可以是恶性的。

“Microsoft Windows Internals”书中说 LPC 是“用于快速报文传递的进程间通信机制”。其实这是误导的，应该说 Port 才是这样的进程间通信机制，而 LPC 是建立在这上面的应用。然而这种说法已经被广泛接受和采纳，都把 LPC 和 Port 混淆起来了，所以本文也只好跟着说“Windows 的 LPC 机制”，而实际上要说的则主要是 Windows 的 Port 机制。在下面的叙述中，凡说到 LPC 的地方往往实际上是在说 Port，读者要注意区分。

端口是一种面向连接的通信机制，通信的双方需要先建立起“连接”。这种连接一般建立在用户进程之间。在建立了连接的双方之间有几种交换报文的方法：

- 不带数据的纯报文。
- 不大于 256 字节的短报文。
- 如果是大于 256 字节的长报文，就要在双方之间建立两个共享内存区(Section)。双方通过共享内存区交换数据，但通过报文进行协调和同步。

大块数据之所以要通过共享内存区交换，一来是因为这样就为用于 Port 机制的缓冲区设置了一个上限，便于内存管理。而更重要的是提高了效率，因为否则便要在发送端把大块数据搬入内核空间，又在接收端把大块数据搬到用户空间。

Windows 内核为基于端口的进程间通信机制提供了不少系统调用，包括(但并不限于)：

- NtCreatePort()
- NtCreateWaitablePort()
- NtListenPort()
- NtConnectPort()
- NtAcceptConnectPort()

- NtCompleteConnectPort()
- NtRequestPort()
- NtRequestWaiReplyPort()
- NtReplyPort()
- NtReplyWaitReceivePort()
- NtReplyWaitReceivePortEx()。同上，但是带有超时控制
- NtReadRequestData()
- NtWriteRequestData()
- NtQueryInformationPort()

这么多的系统调用(由此也可见 LPC 在 Windows 操作系统中的份量)，当然不可能在这里一一加以介绍。本文只是从中拣几个关键而典型的作些介绍。另一方面，由于 Port 与 Socket 的相似性，对于兼容内核的开发而言应该比较容易把它嫁接到 Socket 机制上去。

值得一提的是，Port 在 Win32 API 界面上是不可见的(实际上甚至 LPC 也不是直接可见的)，而 Windows 的系统调用界面又不公开。这说明 Port 只是供微软“内部使用”的。读者后面就会看到，Port 是一种既包括进程间的数据传输，又包括进程间的同步、数据量又可大可小的综合性的进程间通信机制。这样，由微软自己开发的软件、特别是一些系统工具，当然可以使用这些系统调用、也即利用 Port 这种功能比较强的进程间通信机制，而第三方开发的软件可就用不上了。

端口分“连接端口(connection port)”和“通信端口(communication port)”两种，各自扮演着不同的角色。连接端口用于连接的建立，通信端口才真正用于双方的通信。只有服务进程才需要有连接端口，但是通信双方都需要有通信端口。

虽然 LPC 一般发生在进程之间，但是实际参与通信的总是具体的线程，所以在下面的叙述中都以线程作为通信的两端。

典型的建立连接和通信的过程如下：

- 需要建立 LPC 通信时，其中提供服务的一方、即服务线程首先要通过 NtCreatePort()创建一个命名的连接端口、即 Port 对象。这个对象名应为请求服务的一方、即客户线程所知。
- 建立了上述连接端口以后，服务线程应通过 NtListenPort()等待接收来自客户线程的连接请求(服务线程被阻塞)。
- 客户线程通过 NtConnectPort()创建一个客户方的无名通信端口，并向上述命名的连接端口发出连接请求(客户线程被阻塞)。
- 服务线程收到连接请求(因而被唤醒)以后，如果同意建立连接就通过 NtAcceptConnectPort()创建一个服务方的无名通信端口、接受连接、并返回该无名通信端口的 Handle。然后再通过 NtCompleteConnectPort()唤醒客户线程。
- 客户线程被唤醒，并返回所创建的无名通信端口的 Handle。
- 服务线程另创建一个新的线程，负责为客户线程提供 LPC 服务。该线程因企图从上述通信端口接收报文、等待来自客户端的服务请求而被阻塞。所以，新创建的线程时 LPC 服务线程，而原来的服务线程是端口服务进程。
- 端口服务线程再次调用 NtListenPort()，等待来自其它客户的连接请求。
- 客户线程通过 NtRequestWaiReplyPort()向对方发送报文，请求得到 LPC 服务，并因等待回答而被阻塞。
- 服务端的相应线程、即 LPC 服务线程因接收到报文而被唤醒，并根据报文内容提

供相应的 LPC 服务。

- LPC 服务线程通过 NtReplyPort()向客户方发送回答报文(一般是计算结果)。客户线程解除阻塞。

如果回顾一下 Wine 进程与服务进程 wineserver 之间的通信，就可以明白 Wine 是用命名管道和 Socket 在模仿 Windows 的 LPC 通信，只不过那是在用户空间的模仿。另一方面，熟悉 Socket 通信的读者可以看到，Port 与 Socket 是很相像的。

先看 Port 的创建。我们看系统调用 NtCreatePort()的代码：

```
NTSTATUS STDCALL
NtCreatePort (OUT    PHANDLE                 PortHandle,
              IN     POBJECT_ATTRIBUTES      ObjectAttributes,
              IN     ULONG                   MaxConnectInfoLength,
              IN     ULONG                   MaxDataLength,
              IN     ULONG                   MaxPoolUsage)
{
  PEPORT          Port;
  NTSTATUS Status;

  DPRINT("NtCreatePort() Name %x\n", ObjectAttributes->ObjectName->Buffer);

  /* Verify parameters */
  Status = LpcpVerifyCreateParameters (PortHandle, ObjectAttributes,
                          MaxConnectInfoLength, MaxDataLength, MaxPoolUsage);
  . . . . . .

  /* Ask Ob to create the object */
  Status = ObCreateObject (ExGetPreviousMode(), LpcPortObjectType,
                    ObjectAttributes, ExGetPreviousMode(),
                    NULL, sizeof(EPORT), 0, 0, (PVOID*)&Port);
  . . . . . .

  Status = ObInsertObject ((PVOID)Port, NULL, PORT_ALL_ACCESS,
                                                0, NULL, PortHandle);
  . . . . . .

  Status = LpcpInitializePort (Port, EPORT_TYPE_SERVER_RQST_PORT, NULL);
  Port->MaxConnectInfoLength = PORT_MAX_DATA_LENGTH;
  Port->MaxDataLength = PORT_MAX_MESSAGE_LENGTH;
  Port->MaxPoolUsage = MaxPoolUsage;

  ObDereferenceObject (Port);

  return (Status);
```

}

参数 ObjectAttributes、即 OBJECT_ATTRIBUTES 结构中有个 Unicode 字符串，那就是对象名，需要在调用 NtCreatePort()之前加以设置，这跟创建/打开文件时的文件名设置是一样的。当然，除对象名以外，ObjectAttributes 中还有别的信息，那就不是我们此刻所关心的了。其余参数的作用则不言自明。代码中的 LpcpVerifyCreateParameters()对参数进行合理性检查，ObCreateObject()和 ObInsertObject()就无需多说了，而 LpcpInitializePort()主要是对代表着端口的 EPORT 数据结构进行初始化。EPORT 数据结构的定义如下：

```
typedef struct _EPORT
{
   KSPIN_LOCK        Lock;
   KSEMAPHORE        Semaphore;
   USHORT            Type;
   USHORT            State;
   struct _EPORT     *RequestPort;
   struct _EPORT     *OtherPort;
   ULONG             QueueLength;
   LIST_ENTRY        QueueListHead;
   ULONG             ConnectQueueLength;
   LIST_ENTRY        ConnectQueueListHead;
   ULONG             MaxDataLength;
   ULONG             MaxConnectInfoLength;
   ULONG             MaxPoolUsage;          /* size of NP zone */
} EPORT, * PEPORT;
```

结构中的 QueueListHead 就是用来接收报文的队列。ConnectQueueListHead 则是用来缓存连接请求的队列，这是因为一个 Port 可能会一下子接收到好几个连接请求。字段 Type 用来纪录端口的类型，一共有三种类型：

```
#define EPORT_TYPE_SERVER_RQST_PORT      (0)
#define EPORT_TYPE_SERVER_COMM_PORT      (1)
#define EPORT_TYPE_CLIENT_COMM_PORT      (2)
```

从上面的代码中可以看出，NtCreatePort()所创建的是“请求端口”，即类型为 EPORT_TYPE_SERVER_RQST_PORT 的端口，也就是“连接端口”。

字段 state 说明端口的状态和性质，例如 EPORT_WAIT_FOR_CONNECT、EPORT_CONNECTED_CLIENT 等等。

注意每个 EPORT 结构中都嵌有一个“信号量”结构 Semaphore，这就是通信双方用来实现同步的手段。所以说，Port 是集成了数据交换和行为同步的综合性的进程间通信机制。

还有个字段 OtherPort 也值得一说，这是个指针，互相指向已经建立了连接的对方端口。

LpcpInitializePort()的代码就不看了。只是要说明一下，端口对象的初始化也包括了对其“信号量”数据结构的初始化，并且信号量的初值是 0，而最大值则为最大整数 LONG_MAX，所以实际上没有限制。

创建了连接端口以后，服务进程就通过 NtListenPort()等待连接请求，并因此而被阻塞进入睡眠。

```
NTSTATUS STDCALL
NtListenPort (IN HANDLE    PortHandle,    IN PLPC_MESSAGE    ConnectMsg)
{
   NTSTATUS Status;

   /* Wait forever for a connection request. */
   for (;;)
   {
      Status = NtReplyWaitReceivePort(PortHandle, NULL, NULL, ConnectMsg);
      /* Accept only LPC_CONNECTION_REQUEST requests. Drop any other message. */
      if (!NT_SUCCESS(Status) ||
                  LPC_CONNECTION_REQUEST == ConnectMsg->MessageType)
      {
         DPRINT("Got message (type %x)\n", LPC_CONNECTION_REQUEST);
         break;
      }
      DPRINT("Got message (type %x)\n", ConnectMsg->MessageType);
   }

   return (Status);
}
```

所谓“收听(Listen)”，就是在一个 for 循环中反复调用 NtReplyWaitReceivePort()，等待接收来自客户方的报文，直至接收到报文、并且所收到报文的类型为“连接请求”、即 LPC_CONNECTION_REQUEST 时为止。如果接收到的报文不是连接请求就回过去再等，所以才把它放在无限 for 循环中。

NtReplyWaitReceivePort()本来的作用是“发送一个应答报文并等待接收”，但是这里的应答报文指针为 NULL，也就是无应答报文可发，这样就成为只是等待来自客户方的请求了。另一方面，这个函数是不带超时(Timeout)的，只要没有收到客户方的请求就一直等待下去。

[NtListenPort() > NtReplyWaitReceivePort()]

```
NTSTATUS STDCALL
NtReplyWaitReceivePort (IN HANDLE PortHandle, OUT PULONG PortId,
        IN PLPC_MESSAGE LpcReply, OUT PLPC_MESSAGE LpcMessage)
{
   return(NtReplyWaitReceivePortEx (PortHandle, PortId, LpcReply, LpcMessage, NULL));
}
```

NtReplyWaitReceivePort()是不带超时的，另一个系统调用 NtReplyWaitReceivePortEx()

则有超时功能，所以前者是通过后者实现的，只是把(最后那个)参数 Timeout 设成 NULL。

[NtListenPort() > NtReplyWaitReceivePort() > NtReplyWaitReceivePortEx()]

```
NTSTATUS STDCALL
NtReplyWaitReceivePortEx(IN HANDLE PortHandle, OUT PULONG PortId,
            IN PLPC_MESSAGE LpcReply, OUT PLPC_MESSAGE LpcMessage,
            IN PLARGE_INTEGERTimeout)
{
    . . . . . .

    if( Port->State == EPORT_DISCONNECTED )
       {
          /* If the port is disconnected, force the timeout to be 0
             so we don't wait for new messages, because there won't be
             any, only try to remove any existing messages */
          Disconnected = TRUE;
          to.QuadPart = 0;
          Timeout = &to;
       }
    else Disconnected = FALSE;

    Status = ObReferenceObjectByHandle(PortHandle, PORT_ALL_ACCESS,
                                LpcPortObjectType, UserMode, (PVOID*)&Port, NULL);
    . . . . . .
    /* Send the reply, only if port is connected */
    if (LpcReply != NULL && !Disconnected)
    {
        Status = EiReplyOrRequestPort(Port->OtherPort, LpcReply, LPC_REPLY, Port);
        KeReleaseSemaphore(&Port->OtherPort->Semaphore,
                                    IO_NO_INCREMENT, 1, FALSE);
        . . . . . .
    }

    /* Want for a message to be received */
    Status = KeWaitForSingleObject(&Port->Semaphore, UserRequest,
                                                UserMode, FALSE, Timeout);
    if( Status == STATUS_TIMEOUT )
    {
          . . . . . .
    }
    . . . . . .
    /* Dequeue the message */
    KeAcquireSpinLock(&Port->Lock, &oldIrql);
```

```
    Request = EiDequeueMessagePort(Port);
    KeReleaseSpinLock(&Port->Lock, oldIrql);

    if (Request->Message.MessageType == LPC_CONNECTION_REQUEST)
    {
        LPC_MESSAGE Header;
        PEPORT_CONNECT_REQUEST_MESSAGE CRequest;

        CRequest = (PEPORT_CONNECT_REQUEST_MESSAGE)&Request->Message;
        memcpy(&Header, &Request->Message, sizeof(LPC_MESSAGE));
        Header.DataSize = CRequest->ConnectDataLength;
        Header.MessageSize = Header.DataSize + sizeof(LPC_MESSAGE);
        Status = MmCopyToCaller(LpcMessage, &Header, sizeof(LPC_MESSAGE));
        if (NT_SUCCESS(Status))
        {
           Status = MmCopyToCaller((PVOID)(LpcMessage + 1),
                                   CRequest->ConnectData, CRequest->ConnectDataLength);
        }
    }
    else
    {
        Status = MmCopyToCaller(LpcMessage, &Request->Message,
                                            Request->Message.MessageSize);
    }
    . . . . . .
    if (Request->Message.MessageType == LPC_CONNECTION_REQUEST)
    {
        KeAcquireSpinLock(&Port->Lock, &oldIrql);
        EiEnqueueConnectMessagePort(Port, Request);
        KeReleaseSpinLock(&Port->Lock, oldIrql);
    }
    else
    {
        ExFreePool(Request);
    }

    /* Dereference the port */
    ObDereferenceObject(Port);
    return(STATUS_SUCCESS);
}
```

在 Port 机制的实现中，NtReplyWaitReceivePortEx()是个经常用到的函数。刚才我们看到的是从 NtListenPort()逐层调用下来的，所以没有应答报文需要发送，直接就进入等待接收报文的阶段了，但在多数情况下是有应答报文要发送的，服务线程运行的典型情景就是“接

收-处理-应答-再接收”，所以把应答和再接收合并在一起。因此，我们也应看一下在有应答报文要发送时的操作，就是代码中 if (LpcReply != NULL && !Disconnected)语句里面的操作。

当然，如果端口已经处于断开状态、即 Disconnected 为 TRUE，那就不发送应答报文了。从代码中可以看出，应答报文的发送是通过 EiReplyOrRequestPort()完成的。完成以后还要向对方端口的“信号量”执行一次 V 操作，即增量为 1 的 KeReleaseSemaphore()操作，如果对方线程正在睡眠等待就把它唤醒。后面这一步属于线程间同步，读者想必已经熟悉了，这里看一下 EiReplyOrRequestPort()的代码：

[NtReplyWaitReceivePort() > NtReplyWaitReceivePortEx()> EiReplyOrRequestPort()]

```
NTSTATUS STDCALL
EiReplyOrRequestPort (IN PEPORT    Port,
                      IN PLPC_MESSAGE    LpcReply,
                      IN ULONG    MessageType,
                      IN PEPORT    Sender)
{
    . . . . . .

    MessageReply = ExAllocatePoolWithTag(NonPagedPool, sizeof(QUEUEDMESSAGE),
                                                        TAG_LPC_MESSAGE);
    MessageReply->Sender = Sender;

    if (LpcReply != NULL)
    {
        memcpy(&MessageReply->Message, LpcReply, LpcReply->MessageSize);
    }
    MessageReply->Message.ClientId.UniqueProcess = PsGetCurrentProcessId();
    MessageReply->Message.ClientId.UniqueThread = PsGetCurrentThreadId();
    MessageReply->Message.MessageType = MessageType;
    MessageReply->Message.MessageId = InterlockedIncrementUL(&LpcpNextMessageId);

    KeAcquireSpinLock(&Port->Lock, &oldIrql);
    EiEnqueueMessagePort(Port, MessageReply);
    KeReleaseSpinLock(&Port->Lock, oldIrql);

    return(STATUS_SUCCESS);
}
```

参数 LpcReply 是从上面传下来的指针，指向一个 LPC_MESSAGE 数据结构，这就是待发送的报文。下面是 LPC_MESSAGE 数据结构的格式定义：

```
typedef struct _LPC_MESSAGE {
    USHORT    DataSize;
    USHORT    MessageSize;
```

```
    USHORT    MessageType;
    USHORT    VirtualRangesOffset;
    CLIENT_ID    ClientId;
    ULONG    MessageId;
    ULONG    SectionSize;
    UCHAR    Data[ANYSIZE_ARRAY];
} LPC_MESSAGE, *PLPC_MESSAGE;
```

实际上这只是报文的头部，后面的不定长数组 Data[ANYSIZE_ARRAY]才是具体的报文。这里 ANYSIZE_ARRAY 定义为 1，其实定义成 0 也可以，只是表示从这儿开始才是具体的报文内容。具体报文的数据结构可以由使用者自行定义，接收方根据头部的 MessageType 就可以知道收到的是什么报文。不过 Port 机制定义了两种特殊的报文用于建立连接的过程：

```
typedef struct _EPORT_CONNECT_REQUEST_MESSAGE
{
   LPC_MESSAGE MessageHeader;
   PEPROCESS ConnectingProcess;
   struct _SECTION_OBJECT* SendSectionObject;
   LARGE_INTEGER SendSectionOffset;
   ULONG SendViewSize;
   ULONG ConnectDataLength;
   UCHAR ConnectData[0];
} EPORT_CONNECT_REQUEST_MESSAGE;
```

这就是“连接请求”报文，其第一个成分是一个 LPC_MESSAGE 数据结构，那就是报文的头部。下面的“连接应答”报文也是一样。前面的参数 LpcReply 表面上是 LPC_MESSAGE 指针，但是实际上却可以是具体报文的指针。注意最后的 ConnectData[0]表示在这个数据结构的后面还可以有不定长的数据，报文头部的 DataSize 就反映了这部分数据的大小。所以，报文头部的 Data[ANYSIZE_ARRAY]实际上是具体报文的正身，而具体报文如“连接请求”中的 ConnectData[0]则是随同具体报文发送的数据。不过这是额外的数据，因为建立连接所必需的信息已经体现在报文正身的各个字段中了。

```
typedef struct _EPORT_CONNECT_REPLY_MESSAGE
{
   LPC_MESSAGE MessageHeader;
   PVOID SendServerViewBase;
   ULONG ReceiveClientViewSize;
   PVOID ReceiveClientViewBase;
   ULONG MaximumMessageSize;
   ULONG ConnectDataLength;
   UCHAR ConnectData[0];
} EPORT_CONNECT_REPLY_MESSAGE;
```

最后，包括额外数据在内的报文总长度是有限的(256 字节)，如果是大块数据就要通过共享缓冲区发送。

再看上面 EiReplyOrRequestPort()的代码。它先在内核空间分配一个 QUEUEDMESSAGE 数据结构作为报文的“容器”，再把需要发送的报文拷贝到这个数据结构中。注意这里的 LpcReply 有可能是从用户空间传下来的，相应的缓冲区也在用户空间，所以需要把它搬到内核空间的缓冲区中。然后，真正的“发送”操作其实只是把这个数据结构挂入目标端口的接收报文队列中，这是由 EiEnqueueMessagePort()完成的。注意这里的 LpcpNextMessageId 是报文的序号，每次递增。还要说明，拷贝到 QUEUEDMESSAGE 数据结构中的只是报文本身，而不包括通过共享内存区传递的数据，这正是为什么要使用共享内存区的原因。

回到 NtReplyWaitReceivePortEx()的代码。由于没有应答报文要发送，这里直接就通过 KeWaitForSingleObject()进入了睡眠等待。当这个线程接收到了报文而被唤醒(在这里我们忽略超时的可能)时，端口的报文队列里已经有了报文，所以通过 EiDequeueMessagePort()从队列中摘下一个报文的数据结构。如果这是个“连接请求”报文，就把它复制到用户提供的缓冲区中，代码中的两次调用 MmCopyToCaller()就是分别复制报文的正身及其所附带的数据。其实不是“连接请求”报文也要复制，只不过此时只复制报文的正身。

然后，对于“连接请求”报文，这里还通过 EiEnqueueConnectMessagePort()把它挂入端口的 ConnectQueueListHead 队列，这是为随后的“接受连接”操作、即系统调用 NtAcceptConnectPort()留下一个参考，后面我们就会看到其作用。

不过，我们这儿讲的只是如果接收到了连接请求就要做些什么操作，但是实际上此刻还没有客户方提出连接请求，所以服务方线程只是睡眠等待。

至此，服务方已经作好了准备，就等着客户方的连接请求了。如前所述，客户方通过系统调用 NtConnectPort()向命名的连接端口请求连接。注意这里连接的目标就是一个连接端口，而并不指定某个具体的进程。哪一个进程在这个连接端口上执行了 NtListenPort()并因此而受到阻塞正在等待，这请求就实际上是发给那个进程(线程)的。

请求连接的一方对于将来要发送多大的数据量应该是心里有数的。如果数据量不大，那就可以作为附加信息随同报文(在同一个缓冲区中)一起发送。但是，要是数据量比较大(报文总长大于 256 字节)，那就要通过共享内存区“发送”，因为否则的话一来传输效率低下，二来报文缓冲区的大小也不好静态地安排确定。所以，在期望的发送数据量比较大时要准备好一个共享内存区(Section)用于数据发送。这是要由请求连接的一方、即客户方做好准备，通过调用参数传给 NtConnectPort()的。为此客户方要准备好两个数据结构，就是 LPC_SECTION_WRITE 和 LPC_SECTION_READ，把有关共享内存区的信息填写在前一个数据结构中，再把指向这两个数据结构的指针作为参数传给 NtConnectPort()。

这两个数据结构的定义为：

```
typedef struct _LPC_SECTION_WRITE {
    ULONG    Length;
    HANDLE    SectionHandle;
    ULONG    SectionOffset;
    ULONG    ViewSize;
    PVOID    ViewBase;
    PVOID    TargetViewBase;
} LPC_SECTION_WRITE;
```

```
typedef struct _LPC_SECTION_READ {
    ULONG    Length;
    ULONG    ViewSize;
    PVOID    ViewBase;
} LPC_SECTION_READ;
```

注意客户方只提供(和映射)用于它写入(发送)的共享内存区，而反方向的共享内存区则由服务方提供(和映射)。所以 LPC_SECTION_READ 数据结构只是用来获取映射后的结果。

下面看 NtConnectPort()的代码。

```
NTSTATUS STDCALL
NtConnectPort (PHANDLE                  UnsafeConnectedPortHandle,
               PUNICODE_STRING              PortName,
               PSECURITY_QUALITY_OF_SERVICE    Qos,
               PLPC_SECTION_WRITE           UnsafeWriteMap,
               PLPC_SECTION_READ       UnsafeReadMap,
               PULONG                  UnsafeMaximumMessageSize,
               PVOID                   UnsafeConnectData,
               PULONG                  UnsafeConnectDataLength)
{
  . . . . . .

  /* Copy in write map and partially validate. */
  . . . . . .
  /* Handle connection data. */
  . . . . . .
  /* Reference the named port. */
  Status = ObReferenceObjectByName (PortName, 0, NULL, PORT_ALL_ACCESS,
                                    LpcPortObjectType, UserMode, NULL, (PVOID*)&NamedPort);
  . . . . . .
  /* Reference the send section object. */
  if (WriteMap.SectionHandle != INVALID_HANDLE_VALUE)
  {
      Status = ObReferenceObjectByHandle(WriteMap.SectionHandle,
                    SECTION_MAP_READ | SECTION_MAP_WRITE,
                    MmSectionObjectType, UserMode, (PVOID*)&SectionObject, NULL);
      . . . . . .
  }
  else
  {
      SectionObject = NULL;
  }
```

```
    /* Do the connection establishment. */
    Status = EiConnectPort(&ConnectedPort, NamedPort, SectionObject, SectionOffset,
                    WriteMap.ViewSize, &WriteMap.ViewBase, &WriteMap.TargetViewBase,
                    &ReadMap.ViewSize, &ReadMap.ViewBase, &MaximumMessageSize,
                    ConnectData, &ConnectDataLength);
    . . . . . .

    /* Do some initial cleanup. */
    if (SectionObject != NULL)
    {
        ObDereferenceObject(SectionObject);
        SectionObject = NULL;
    }
    ObDereferenceObject(NamedPort);
    NamedPort = NULL;

    /* Copy the data back to the caller. */
    . . . . . .

    Status = ObInsertObject(ConnectedPort, NULL, PORT_ALL_ACCESS,
                                            0, NULL, &ConnectedPortHandle);
    . . . . . .
    Status = MmCopyToCaller(UnsafeConnectedPortHandle,
                                &ConnectedPortHandle, sizeof(HANDLE));
    . . . . . .
    if (UnsafeWriteMap != NULL)
    {
        Status = MmCopyToCaller(UnsafeWriteMap,
                                &WriteMap, sizeof(LPC_SECTION_WRITE));
        . . . . . .
    }
    if (UnsafeReadMap != NULL)
    {
        Status = MmCopyToCaller(UnsafeReadMap,
                                &ReadMap, sizeof(LPC_SECTION_READ));
        . . . . . .
    }
    . . . . . .
    return(STATUS_SUCCESS);
}
```

先看调用参数。其中 UnsafeConnectedPortHandle 是用来返回建立连接以后的通信端口 Handle 的。之所以说是“Unsafe”是为了强调这可能是在用户空间，因而可能需要把它复制到内核空间的副本 ConnectedPortHandle 中。其实这也并非特殊，多数系统调用都是这样的。

参数 PortName 是个 Unicode 字符串，就是目标端口的对象名。另一个参数 UnsafeMaximumMessageSize 为允许通过本连接(如果成功的话)发送的报文长度设置了一个上限。此外，在请求建立连接的过程中也可以附带着向服务线程发送一些数据，而服务线程在接受或拒绝连接请求时也可以发回来一些数据，参数 UnsafeConnectData 就是指向为此而设的缓冲区，而 UnsafeConnectDataLength 则为数据长度。

指针 UnsafeWriteMap 和 UnsafeReadMap 就是与共享内存区有关的参数。如果所有需要传递的报文都是短报文，那么这两个指针也可以是 NULL。

参数 Qos 就要费些口舌了。这是个指针，指向一个 SECURITY_QUALITY_OF_SERVICE 数据结构。QoS 字面上的意义是服务质量保证，具体则与一种称为“临时身份(Impersonate)”的机制有关。Impersonate 在字典上的解释是“人格化、扮演”，实际上是“身份、代表、授权”的意思。在 LPC 机制中，服务进程应客户进程的请求而执行某些操作。换言之，服务进程是代表着客户进程、以客户进程的名义在执行这些操作，因而服务进程在执行这些操作的过程中所具有的各种权限理应与具体的客户进程相同，这既不是服务进程本身的权限，也不是某种一成不变的权限。不妨拿日常生活中当事人与律师的关系作个对比：律师有律师的法定权限，不管当事人是谁，律师的权限都是一样的，并且一般都是高于当事人所具有的权限，例如律师可以查阅法庭的卷宗，而当事人显然不可以，所以律师并不是以当事人的身份在工作，这样才能维护社会公正。但是，LPC 则与此相反，服务进程应该以客户进程的身份操作，否则就有可能被权限较低的进程利用来达到本来不能达到的目的，或者本身权限较高的客户却不能达到应该可以达到的目的。操作系统要保证的恰恰是一套等级森严的体制，而不是社会公正。所以，服务进程在为客户提供本地过程调用时就应该临时转换到客户的身份，不过这是可以选择的。显然，这与访问权限有关，因而与系统的安全机制有关。这方面一则说来话长，二则目前的 ReactOS 才刚开始触及安全机制的实现，所以我们以后再来专题讨论。在调用 NtConnectPort()的时候，指针 Qos 也可以是 NULL，那就表示采用默认的方式，就是服务方不采用客户的身份。

继续看 NtConnectPort()其余的调用参数。如前所述，通过 NtConnectPort()建立端口连接时，还可以随同连接请求发送一些数据，而对方也可以回复一些数据。这里的 UnsafeConnectData 就是个指针，而 UnsafeConnectDataLength 则是数据的长度。请求连接的进程在 NtListenPort()中受阻而进入睡眠，直到对方接受(或拒绝)了连接请求时才被唤醒并且再次受调度运行。这样，当客户进程从 NtConnectPort()返回时，缓冲区中的内容已经是从服务进程返回的数据。

NtConnectPort()的开头一段代码都是为了把用户空间的参数复制到内核中，这里就略去了，我们把注意力集中在实质性的操作。显然，这里的核心是 EiConnectPort()。注意这里把 WriteMap 中 ViewBase 和 TargetViewBase 字段的地址，以及 ReadMap 中 ViewSize 和 ViewBase 字段的地址传了下去，这说明 EiConnectPort()有可能修改这些字段的数值。

[NtConnectPort() > EiConnectPort()]

```
NTSTATUS STDCALL
EiConnectPort(IN PEPORT* ConnectedPort,
              IN PEPORT NamedPort,
              IN PSECTION_OBJECT Section,
              IN LARGE_INTEGER SectionOffset,
              IN ULONG ViewSize,
```

```
                OUT PVOID* ClientSendViewBase,
                OUT PVOID* ServerSendViewBase,
                OUT PULONG ReceiveViewSize,
                OUT PVOID* ReceiveViewBase,
                OUT PULONG MaximumMessageSize,
                IN OUT PVOID ConnectData,
                IN OUT PULONG ConnectDataLength)
{
  . . . . . .

  /* Create a port to represent our side of the connection */
  Status = ObCreateObject (KernelMode, LpcPortObjectType, NULL,
                           KernelMode, NULL, sizeof(EPORT), 0, 0, (PVOID*)&OurPort);
  . . . . . .
  LpcpInitializePort(OurPort, EPORT_TYPE_CLIENT_COMM_PORT, NamedPort);

  /* Allocate a request message. */
  RequestMessage = ExAllocatePool(NonPagedPool,
                        sizeof(EPORT_CONNECT_REQUEST_MESSAGE) +
                        RequestConnectDataLength);
  . . . . . .

  /* Initialize the request message. */
  RequestMessage->MessageHeader.DataSize =
     sizeof(EPORT_CONNECT_REQUEST_MESSAGE) + RequestConnectDataLength -
     sizeof(LPC_MESSAGE);
  RequestMessage->MessageHeader.MessageSize =
     sizeof(EPORT_CONNECT_REQUEST_MESSAGE) + RequestConnectDataLength;
  . . . . . .
  RequestMessage->MessageHeader.SectionSize = 0;
  RequestMessage->ConnectingProcess = PsGetCurrentProcess();
  ObReferenceObjectByPointer(RequestMessage->ConnectingProcess,
                             PROCESS_VM_OPERATION, NULL, KernelMode);
  RequestMessage->SendSectionObject = (struct _SECTION_OBJECT*)Section;
  RequestMessage->SendSectionOffset = SectionOffset;
  RequestMessage->SendViewSize = ViewSize;
  RequestMessage->ConnectDataLength = RequestConnectDataLength;
  if (RequestConnectDataLength > 0)
  {
      memcpy(RequestMessage->ConnectData, ConnectData, RequestConnectDataLength);
  }

  /* Queue the message to the named port */
  EiReplyOrRequestPort(NamedPort, &RequestMessage->MessageHeader,
```

```
                                        LPC_CONNECTION_REQUEST, OurPort);
KeReleaseSemaphore(&NamedPort->Semaphore, IO_NO_INCREMENT, 1, FALSE);
ExFreePool(RequestMessage);

/* Wait for them to accept our connection */
KeWaitForSingleObject(&OurPort->Semaphore, UserRequest, UserMode,
                                                        FALSE, NULL);

/* Dequeue the response */
KeAcquireSpinLock (&OurPort->Lock, &oldIrql);
Reply = EiDequeueMessagePort(OurPort);
KeReleaseSpinLock (&OurPort->Lock, oldIrql);
CReply = (PEPORT_CONNECT_REPLY_MESSAGE)&Reply->Message;

/* Do some initial cleanup. */
ObDereferenceObject(PsGetCurrentProcess());

/* Check for connection refusal. */
if (CReply->MessageHeader.MessageType == LPC_CONNECTION_REFUSED)
{
      . . . . . .
      return(STATUS_PORT_CONNECTION_REFUSED);
}

/* Otherwise we are connected. Copy data back to the client. */
*ServerSendViewBase = CReply->SendServerViewBase;
*ReceiveViewSize = CReply->ReceiveClientViewSize;
*ReceiveViewBase = CReply->ReceiveClientViewBase;
*MaximumMessageSize = CReply->MaximumMessageSize;
if (ConnectDataLength != NULL)
{
      *ConnectDataLength = CReply->ConnectDataLength;
      memcpy(ConnectData, CReply->ConnectData, CReply->ConnectDataLength);
}

/* Create our view of the send section object. */
if (Section != NULL)
{
      *ClientSendViewBase = 0;
      Status = MmMapViewOfSection(Section, PsGetCurrentProcess(),
                  ClientSendViewBase, 0, ViewSize, &SectionOffset,
                  &ViewSize, ViewUnmap, 0, PAGE_READWRITE);
      . . . . . .
}
```

```
    /* Do the final initialization of our port. */
    OurPort->State = EPORT_CONNECTED_CLIENT;

    /* Cleanup. */
    ExFreePool(Reply);
    *ConnectedPort = OurPort;
    return(STATUS_SUCCESS);
}
```

参数 ConnectedPort 的方向说是 IN，看来倒应该是 OUT，用来返回新创建的通信端口(见下)的 Handle。参数 Section 是上面传下来的 SECTION_OBJECT 结构指针。至于其余参数的作用，读者不妨对比一下前面调用这个函数时所使用的实参。

这个函数的执行大致上可以分成四个阶段。

第一个阶段为将来(连接请求被接受以后)的通信创建一个“通信端口”，这就是 ObCreateObject()和随后的 LpcpInitializePort()所做的事。其结果就是代码中的指针 OurPort。

第二个阶段先准备好“连接请求”报文的数据结构，包括通过参数 ConnectData 传递下来的附加数据，然后就通过 EiReplyOrRequestPort()把报文的容器挂入目标端口的接收队列。注意这里的目标端口是个“连接端口”而不是“通信端口”。接着，还要通过 KeReleaseSemaphore()对目标端口的“信号量”实行一次 V 操作，把正在 NtListenPort()中睡眠等待的服务线程唤醒。

第三个阶段就是通过 KeWaitForSingleObject()睡眠等待对方是否接受连接请求的答复了。

第四个阶段，得到了服务方线程的答复以后，客户线程从睡眠中被唤醒，通过 EiDequeueMessagePort()从其通信端口的队列中获取对方的应答报文。如果对方接受连接请求的话，就把原先准备下用于发送数据的共享内存区映射到自己的用户空间，并返回映射的地址。

注意这里映射的只是供客户方写入的区间。还有，这里的指针 ClientSendViewBase 就是前面的&WriteMap.ViewBase。反向的区间、即供客户方读出的区间是由服务方代为映射的，实际映射的地址通过应答报文中的 ReceiveClientViewBase 字段返回过来，这里代码中的指针 ReceiveViewBase 就是前面的&ReadMap.ViewBase。应答报文中的 SendServerViewBase 字段是服务方读出区间的映射地址。为什么需要知道对方读出区间的映射地址地址呢？这是因为在发往对方的数据中可能会有起着指针作用的数据，对这些数据的值需要在发送出去之前根据共享内存区在对方的映射位置进行“重定位”。那为什么客户方用于读出的区间要由对方代为映射呢？客户方自己当然也可以映射，但是那样的话又得把映射的地址告知对方，那就又得再发送一个报文了。

再看服务方线程。它在 NtListenPort()中被唤醒，以后就从其连接端口的接收队列中获取由客户方发来的报文。如果这不是“连接请求”报文就又回过去睡眠等待，直至接收到“连接请求”报文才从 NtListenPort()返回。服务方线程在接收到连接请求以后要作出决定，并通过系统调用 NtAcceptConnectPort()加以接受或拒绝。

```
NTSTATUS STDCALL
NtAcceptConnectPort (PHANDLE    ServerPortHandle,    HANDLE    NamedPortHandle,
```

```
                PLPC_MESSAGE    LpcMessage,    BOOLEAN    AcceptIt,
                PLPC_SECTION_WRITE    WriteMap,    PLPC_SECTION_READ    ReadMap)
{
  . . . . . .

  Size = sizeof(EPORT_CONNECT_REPLY_MESSAGE);
  if (LpcMessage)
  {
      Size += LpcMessage->DataSize;
  }

  CReply = ExAllocatePool(NonPagedPool, Size);
  if (CReply == NULL)
     {
       return(STATUS_NO_MEMORY);
     }

  Status = ObReferenceObjectByHandle(NamedPortHandle, PORT_ALL_ACCESS,
                          LpcPortObjectType, UserMode, (PVOID*)&NamedPort, NULL);
  . . . . . .

  /* Create a port object for our side of the connection */
  if (AcceptIt)
  {
       Status = ObCreateObject(ExGetPreviousMode(), LpcPortObjectType, NULL,
                          ExGetPreviousMode(),NULL, sizeof(EPORT), 0, 0, (PVOID*)&OurPort);
       if (!NT_SUCCESS(Status))
       {
          ExFreePool(CReply);
          ObDereferenceObject(NamedPort);
          return(Status);
       }

       Status = ObInsertObject ((PVOID)OurPort, NULL,
                                    PORT_ALL_ACCESS, 0, NULL, ServerPortHandle);
       . . . . . .
       LpcpInitializePort(OurPort, EPORT_TYPE_SERVER_COMM_PORT, NamedPort);
  }

  /* Dequeue the connection request */
  KeAcquireSpinLock(&NamedPort->Lock, &oldIrql);
  ConnectionRequest = EiDequeueConnectMessagePort(NamedPort);
  KeReleaseSpinLock(&NamedPort->Lock, oldIrql);
  CRequest =
```

```
        (PEPORT_CONNECT_REQUEST_MESSAGE)(&ConnectionRequest->Message);

/* Prepare the reply. */
if (LpcMessage != NULL)
{
      memcpy(&CReply->MessageHeader, LpcMessage, sizeof(LPC_MESSAGE));
      memcpy(&CReply->ConnectData, (PVOID)(LpcMessage + 1),
                                                      LpcMessage->DataSize);
      CReply->MessageHeader.MessageSize =
            sizeof(EPORT_CONNECT_REPLY_MESSAGE) + LpcMessage->DataSize;
      CReply->MessageHeader.DataSize = CReply->MessageHeader.MessageSize -
                  CReply->MessageHeader.MessageSize - sizeof(LPC_MESSAGE);
      CReply->ConnectDataLength = LpcMessage->DataSize;
}
else
{
      CReply->MessageHeader.MessageSize =
                                          sizeof(EPORT_CONNECT_REPLY_MESSAGE);
      CReply->MessageHeader.DataSize =
                sizeof(EPORT_CONNECT_REPLY_MESSAGE) - sizeof(LPC_MESSAGE);
      CReply->ConnectDataLength = 0;
}
if (!AcceptIt)
{
      EiReplyOrRequestPort(ConnectionRequest->Sender,
                  &CReply->MessageHeader,
                  LPC_CONNECTION_REFUSED,
                  NamedPort);
      KeReleaseSemaphore(&ConnectionRequest->Sender->Semaphore,
                IO_NO_INCREMENT,
                1,
                FALSE);
      ObDereferenceObject(ConnectionRequest->Sender);
      ExFreePool(ConnectionRequest);
      ExFreePool(CReply);
      ObDereferenceObject(NamedPort);
      return (STATUS_SUCCESS);
}

/* Prepare the connection. */
if (WriteMap != NULL)
{
      PSECTION_OBJECT SectionObject;
      LARGE_INTEGER SectionOffset;
```

```
        Status = ObReferenceObjectByHandle(WriteMap->SectionHandle,
                        SECTION_MAP_READ | SECTION_MAP_WRITE,
                        MmSectionObjectType, UserMode, (PVOID*)&SectionObject, NULL);
        . . . . . .

        SectionOffset.QuadPart = WriteMap->SectionOffset;
        WriteMap->TargetViewBase = 0;
        CReply->ReceiveClientViewSize = WriteMap->ViewSize;
        Status = MmMapViewOfSection(SectionObject, CRequest->ConnectingProcess,
                        &WriteMap->TargetViewBase, 0, CReply->ReceiveClientViewSize,
                        &SectionOffset, &CReply->ReceiveClientViewSize,
                        ViewUnmap, 0, PAGE_READWRITE);
        . . . . . .

        WriteMap->ViewBase = 0;
        Status = MmMapViewOfSection(SectionObject, PsGetCurrentProcess(),
                        &WriteMap->ViewBase, 0, WriteMap->ViewSize,
                        &SectionOffset, &WriteMap->ViewSize,
                        ViewUnmap, 0, PAGE_READWRITE);
        . . . . . .

        ObDereferenceObject(SectionObject);
    }
    if (ReadMap != NULL && CRequest->SendSectionObject != NULL)
    {
        LARGE_INTEGER SectionOffset;

        SectionOffset = CRequest->SendSectionOffset;
        ReadMap->ViewSize = CRequest->SendViewSize;
        ReadMap->ViewBase = 0;
        Status = MmMapViewOfSection(
                        CRequest->SendSectionObject, PsGetCurrentProcess(),
                        &ReadMap->ViewBase, 0, CRequest->SendViewSize,
                        &SectionOffset, &CRequest->SendViewSize,
                        ViewUnmap, 0, PAGE_READWRITE);
        . . . . . .
    }

    /* Finish the reply. */
    if (ReadMap != NULL)
    {
        CReply->SendServerViewBase = ReadMap->ViewBase;
    }
```

```
    else
    {
        CReply->SendServerViewBase = 0;
    }
    if (WriteMap != NULL)
    {
        CReply->ReceiveClientViewBase = WriteMap->TargetViewBase;
    }
    CReply->MaximumMessageSize = PORT_MAX_MESSAGE_LENGTH;

    /* Connect the two ports */
    OurPort->OtherPort = ConnectionRequest->Sender;
    OurPort->OtherPort->OtherPort = OurPort;
    EiReplyOrRequestPort(ConnectionRequest->Sender,
                         (PLPC_MESSAGE)CReply, LPC_REPLY, OurPort);
    ExFreePool(ConnectionRequest);
    ExFreePool(CReply);

    ObDereferenceObject(OurPort);
    ObDereferenceObject(NamedPort);

    return (STATUS_SUCCESS);
}
```

如果接受连接请求，那么服务方也要创建一个通信端口，因为原来的连接端口是专门用来接收连接请求的。第一个参数 ServerPortHandle 就是用来返回新建通信端口的 Handle。而 NamedPortHandle 当然就是连接端口的 Handle，这是本次操作的目标对象。

参数 AcceptIt 表示是否接受连接请求。

参数 WriteMap 和 ReadMap 与 NtConnectPort()中所用者相同。同样，如果预期需要发送的数据量较大的话，服务方也要为此提供一个共享内存区。

先看不接受连接请求时的情况，因为这比较简单。这就是条件语句 if (!AcceptIt)里面的操作：先将一个“拒绝连接”报文、即类型为 LPC_CONNECTION_REFUSED 的报文、通过 EiReplyOrRequestPort()挂入对方端口的报文队列，然后在对方端口的“信号量”上执行一次 V 操作，以唤醒正在等待的对方线程。这样就行了。

接受连接请求时的情况就比较复杂一点：

1. 先创建一个通信端口，就是类型为 EPORT_TYPE_SERVER_COMM_PORT 的端口。
2. 然后为应答报文 LpcMessage 准备好一个内核版本、就是类型为 EPORT_CONNECT_REPLY_MESSAGE 的数据结构 Creply。
3. 处理共享内存区的映射。注意这里做了三次映射：
   - 把由服务方提供的共享内存区映射到客户进程的用户空间，这是客户方的接收区。“连接请求”报文中的 ConnectingProcess 提供了指向客户进程的 EPROCESS 数据结构的指针。
   - 把由服务方提供的共享内存区映射到服务方自己的用户空间，这是服务方进程的写入区。

- 把由客户方提供的共享内存区映射到服务方的用户空间，这是服务方进程的读出区。

  注意这里在调用 MmMapViewOfSection()时所给定的地址都是 0，表示听从分配。所分配的地址要通过 WriteMap 和 ReadMap 返回到用户空间，特别是替客户方进程代为映射的地址要通过应答报文发送给对方。

4. 使服务方通信端口和客户方通信端口的指针 OtherPort 互相指向对方，即建立连接。
5. 通过 EiReplyOrRequestPort()将应答报文挂入客户方端口的报文队列，但是并不唤醒客户方线程。

在完成了 NtAcceptConnectPort()以后，服务方线程还需要对新创建的通信端口执行一下另一个系统调用 NtCompleteConnectPort()。目的在于唤醒客户方线程。注意此时的操作对象已经是新创建的通信端口，而不再是连接端口。

```
NTSTATUS STDCALL
NtCompleteConnectPort (HANDLE hServerSideCommPort)
{
  NTSTATUS Status;
  PEPORT    ReplyPort;

  . . . . . .
  /* Ask Ob to translate the port handle to EPORT */
  Status = ObReferenceObjectByHandle (hServerSideCommPort, PORT_ALL_ACCESS,
                                      LpcPortObjectType, UserMode, (PVOID*)&ReplyPort, NULL);
  . . . . . .
  /* Verify EPORT type is a server-side reply port; otherwise tell the caller
     the port handle is not valid. */
  if (ReplyPort->Type != EPORT_TYPE_SERVER_COMM_PORT)
     {
        ObDereferenceObject (ReplyPort);
        return STATUS_INVALID_PORT_HANDLE;
     }

  ReplyPort->State = EPORT_CONNECTED_SERVER;
  /* Wake up the client thread that issued NtConnectPort. */
  KeReleaseSemaphore(&ReplyPort->OtherPort->Semaphore,
                                      IO_NO_INCREMENT, 1, FALSE);
  /* Tell Ob we are no more interested in ReplyPort */
  ObDereferenceObject (ReplyPort);
  return (STATUS_SUCCESS);
}
```

前面，在 NtAcceptConnectPort()的代码中，虽然已经将应答报文挂入了客户方端口的接收队列，却并未唤醒客户方线程。现在就通过对其信号量的 V 操作将其唤醒。

这样，就建立起了客户方与服务方的一对通信端口的连接。以后就可以通过这个连接通

信了。一般总是服务方线程先通过 NtReplyWaitReceivePort()或 NtReplyWaitReceivePortEx()等待对方发来报文，由于 Port 机制实际上只用于 LPC，客户方发往服务方的一般都是服务请求报文，而服务方则根据具体的请求提供服务，然后发回应答报文、一般是返回结果。不过，也并没有规定必须是服务方等待客户方的报文，反过来也并无不可。

不管是那一方，需要向对方发送一个报文时可以通过系统调用 NtRequestPort()发送。

```
NTSTATUS STDCALL
NtRequestPort (IN HANDLE PortHandle, IN PLPC_MESSAGE LpcMessage)
{
    . . . . . .

    Status = ObReferenceObjectByHandle(PortHandle, PORT_ALL_ACCESS,
                                LpcPortObjectType, UserMode, (PVOID*)&Port, NULL);
    . . . . . .
    Status = LpcRequestPort(Port->OtherPort, LpcMessage);
    ObDereferenceObject(Port);
    return(Status);
}
```

显然，具体的操作是由 LpcRequestPort()完成的。区别在于 LpcRequestPort()要求使用指向 EPORT 数据结构的指针，而传给 NtRequestPort()的是 Handle，需要加以转换。Handle 本质上是数组下标，所以从 Handle 到结构指针的转换开销并不大。

[NtRequestPort() > LpcRequestPort()]

```
NTSTATUS STDCALL LpcRequestPort (IN PEPORT    Port,
                                 IN PLPC_MESSAGE    LpcMessage)
{
    NTSTATUS Status;

    DPRINT("LpcRequestPort(PortHandle %08x, LpcMessage %08x)\n", Port, LpcMessage);

#ifdef __USE_NT_LPC__
    /* Check the message's type */
    if (LPC_NEW_MESSAGE == LpcMessage->MessageType)
    {
        LpcMessage->MessageType = LPC_DATAGRAM;
    }
    else if (LPC_DATAGRAM == LpcMessage->MessageType)
    {
        return STATUS_INVALID_PARAMETER;
    }
    else if (LpcMessage->MessageType > LPC_CLIENT_DIED)
    {
```

```
        return STATUS_INVALID_PARAMETER;
    }
    /* Check the range offset */
    if (0 != LpcMessage->VirtualRangesOffset)
    {
        return STATUS_INVALID_PARAMETER;
    }
#endif

    Status = EiReplyOrRequestPort(Port, LpcMessage, LPC_DATAGRAM, Port);
    KeReleaseSemaphore(&Port->Semaphore, IO_NO_INCREMENT, 1, FALSE );
    return(Status);
}
```

可见，NtRequestPort()只是发送，而并不等待对方的回应。如果需要等待回应的话可以采用另一个系统调用 NtRequestWaitReplyPort()。

需要向对方发送应答报文时可以用 NtReplyPort()。

```
NTSTATUS STDCALL
NtReplyPort (IN HANDLE PortHandle, IN PLPC_MESSAGE LpcReply)
{
    NTSTATUS Status;
    PEPORT Port;

    DPRINT("NtReplyPort(PortHandle %x, LpcReply %x)\n", PortHandle, LpcReply);

    Status = ObReferenceObjectByHandle(PortHandle, PORT_ALL_ACCESS,
                                       LpcPortObjectType, UserMode, (PVOID*)&Port, NULL);
    . . . . . .
    Status = EiReplyOrRequestPort(Port->OtherPort, LpcReply, LPC_REPLY, Port);
    KeReleaseSemaphore(&Port->OtherPort->Semaphore, IO_NO_INCREMENT, 1, FALSE);

    ObDereferenceObject(Port);

    return(Status);
}
```

当然，这是纯粹的发送应答报文，如果是发送应答报文并且等待下一个请求，那就要用 NtReplyWaitReceivePort()，这读者已经在前面看到了。

可见，Port 是一种功能相当强、相当齐全、结构又相当完整的综合性的进程间通信机制，这样的机制理应提供给应用软件的开发者，或者在 Win32 API 上提供相应的库函数，或是把有关的系统调用公诸于世。但是微软却并不这么干，倒是一方面讳莫如深，一方面供自己的软件内部使用。这样，如果都来开发应用软件，那别的公司如何能与微软公平竞争呢？正因

为如此，美国一直有人在呼吁甚至提起诉讼，要把操作系统和应用软件的开发分拆开来，不能让同一家公司既做操作系统又做应用软件。另一方面，这也可以解释为什么总是有这许多人热衷于探究 Windows 和相关产品的“Undocumented…”、“…Internals”、“Inside…”。

最后还要提一下，有些资料中还提到 Windows 有一种“快捷 LPC（QuickLPC）”机制。这就是建立在上一篇漫谈中讲到的“事件对”基础上的 LPC。早期 Windows 上的 csrss 通信太频繁了，需要有一种非常轻快的进程间通信手段，所以才有了 QuickLPC。现在，一方面是 csrss 的功能大都移到了内核中，一方面是处理器的速度也有了量级的提高，QuickLPC 就变得不那么重要了。