漫谈兼容内核之十：
Windows 的进程创建和映像装入

毛德操

关于 Windows 的进程创建和映像装入的过程，“Microsoft Windows Internals 4e”一书的第六章中有颇为详细的说明。本文就以此为依据，夹译、夹叙、夹议地作一介绍。书中说，创建进程的过程分成六个阶段，发生于操作系统的三个部分中，那就是：Windows 客户端即某个应用进程的包括 Kernel32.dll 在内的动态连接库，Windows 的“执行体”、即内核(确切地说是内核的上层)，以及 Windows 子系统的服务进程 Csrss 中。这六个阶段是：

1. 打开目标映像文件。
2. 创建 Windows 的“执行体进程对象”，也就是内核中的“进程控制块”数据结构。
3. 创建该进程的初始(第一个)线程，包括其堆栈、上下文、以及“执行体线程对象”，即内核中的“线程控制块”数据结构。
4. 将新建进程通知 Windows 子系统。
5. 启动初始线程地运行(除非因为参数中的 CREATE_SUSPENDED 标志位为 1 而一创建便被挂起)。
6. 在新进程和线程的上下文中完成用户空间的初始化，包括装入所需的 DLL，然后开始目标程序的运行。

下面分阶段叙述。

第一阶段：打开目标映像文件

在 Win32 位 API 中，创建进程是由 CreateProcess()完成的。这实际上是个宏定义，根据不同的情况定义成 CreateProcessA()或 CreateProcessW()之一，这两个函数都在 kernel32.dll 中(可以用工具 depends 观察)。两个函数的区别仅在于字符串的表达，前者采用 ASCII 字符，而后者采用“宽字符”、即 Unicode。实际上 Windows 的内部都采用宽字符，所以前者只是把字符串转换成宽字符格式，然后调用后者。

可以在 Windows 上运行的可执行软件有好几类，处理的方法自然就不一样：

- Windows 的 32 位.exe 映像，直接运行。
- Windows 的 16 位.exe 映像，启动 ntvdm.exe，以原有命令行作为参数。
- DOS 的.exe、.com、或.pif 映像，启动 ntvdm.exe，以原有命令行作为参数。
- DOS 的.bat 或.cmd 批命令文件(脚本)，启动 cmd.exe，以原有命令行作为参数。
- POSIX 可执行映像，启动 posix.exe，以原有命令行作为参数。
- OS/2 可执行映像，启动 os2.exe，以原有命令行作为参数。

这里面最重要的当然是 32 位的.exe 映像，而最后两类现在已经很少见了。从对于除 32 位.exe 以外的各种映像的处理，读者不妨对比一下 Wine 对.exe 映像的处理，看看这里有着什么样的相似性。

当然，我们在这里只关心 32 位.exe 映像。对于这一类映像，CreateProcess()首先打开映像文件，再为其(分配)创建一个“Section”、即内存区间。创建内存区间的目的当然是要把映像文件影射到这个区间，不过此时还不忙着映射，还要看看。看什么呢？首先是看已经打开的目标文件是否一个合格的.exe 映像(万一是 DLL 映像？)。还要看的事就有点出乎读者意外了，看的是在“注册表”中的这个路径：

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options

用 depends 可以看到，ntdll.dll 中有个函数 LdrQueryImageFileExecutionOptions()，就是专门干这个事的。

如果上述路径下有以目标映像文件的文件名和扩展名为“键”的表项，例如“image.exe”，而表项中又有名为“Debugger”的值，那么这个值(一个字符串)就替换了原来的目标文件名，变成新的目标映像名，并重新执行上述的第一阶段操作。这样做的目的当然是为调试程序提供方便，但是我们不妨设想：如果黑客或某个木马程序设法在注册表中加上了一个表项？这时候用户以为是启动了程序 A，而实际启动的却是 B！。

第二阶段：创建内核中的进程对象

我们知道，Linux 上的每个进程(线程)都有一个“进程控制块”、即 task_struct 数据结构，与具体进程/线程有关的绝大部分信息都集中存储在这个数据结构中。而 Windows 则有所不同。首先，Windows 的进程和线程各有不同的“对象”、即数据结构，从概念上把线程和进程分离开来。线程是具体的(执行)上下文，是 CPU 调度的单位和目标，而进程只是若干共享地址空间和特性(如调度优先级)的线程的集合。于是，进程有进程的数据结构，线程有线程的数据结构。这就好像是对一组 task_struct 数据结构“提取公因子”所形成的结果，这个举措是很好理解的。进一步，Windows 又把本可集中存储的的进程数据结构也拆分成好几个对象，有的在内核中，有的则在用户空间。

内核中与进程有关的对象有：

- EPROCESS。即 struct _EPROCESS，在“Internals”书中也称为“Process Block”。它代表着 Windows 的一个进程，‘E’表示“Executive”，微软把 Windows 内核中的上层称为“Executive”、以区别于下层的设备驱动和内存管理等成分、一般翻译成“执行体”。“Executive”也有“管理”、“运行”的意思(所以 CEO 就是“总裁”)。
- KPROCESS。这是 EPROCESS 内部的一个成分，其名称就叫“Pcb”。
- W32PROCESS。下面将要讲到，在用户空间有个“Windows 子系统”的服务进程 csrss。这个服务进程为系统中的每个 Windows 应用进程都维持着一个数据结构，其中包含了一些与窗口和图形界面有关的信息。而对于窗口和图形界面的操作原来也是由 csrss 在“客户”进程的请求下实现的。但是，为了提高效率，后来把这部分功能移到了内核中。与此相应，有关数据结构的一部分也需要移到内核中，就成了 W32PROCESS。

既然 KPROCESS 是 EPROCESS 一部分，实际上内核中与进程有关的对象实际上只有两种，就是 EPROCESS 和 W32PROCESS。不过这里没有包括“打开对象表”，那也是每个进程都有的(Linux 内核中的“打开文件表”也在进程控制块的外面)。

用户空间与进程有关的对象有：

- 如上所述，把 W32PROCESS 数据结构移入内核以后，csrss 仍需要为每个 Windows 进程保持一些别的信息，所以 csrss 内部仍有按进程的相应数据结构。
- PEB(Process Environment Block)、即“进程环境块”。PEB 中记录着进程的运行参数、映像装入地址等等信息。PEB 在用户空间中的位置是固定的，总是在 0x7ffdf000。在 Windows 中，用户空间和系统空间的分界线是 2GB、即 0x80000000，所以 PEB 在靠近用户空间顶端的地方。

“Internals”书中并未给出有关数据结构的定义，但是通过 Debug 手段给出了 EPROCESS 的内部结构：

```
+0x000      Pcb                 : _KPROCESS
+0x06c      ProcessLock         : _EX_PUSH_LOCK
+0x070      CreateTime          : _LARGE_INTEGER
+0x078      ExitTime            : _LARGE_INTEGER
+0x080      RundownProtect      : _EX_RUNDOWN_REF
+0x084      UniqueProcessId     : Ptr32Void
+0x088      ActiveProcessLinks   : _LIST_ENTRY
+0x090      QuotaUsage          : [3]  Uint4B
+0x09c      QuotaPeak           : [3]  Uint4B
+0x0a8      CommitCharge        : Uint4B
+0x0ac      PeakVirtualSize     : Uint4B
+0x0b0      VirtualSize         : Uint4B
+0x0b4      SessionProcessLinks : _LIST_ENTRY
+0x0bc      DebugPort           : Ptr32Void
+0x0c0      ExceptionPort       : Ptr32Void
+0x0c4      ObjectTable         : Ptr32_HANDLE_TABLE
+0x0c8      Token               : _EX_FAST_REF
+0x0cc      WorkingSetLock      : _FAST_MUTEX
+0x0ec      WorkingSetPage      : Uint4B
+0x0f0      AddressCreationLock : _FAST_MUTEX
+0x110      HyperSpaceLock      : Uint4B
+0x114      ForkInProgress      : Ptr32_ETHREAD
+0x118      HardwareTrigger     : Uint4B
```

可见，EPROCESS 的第一个成分是 Pcb，其类型是_KPROCESS、即 KPROCESS，这是一个大小为 0x6c 的数据结构。书中也给出了它的内部结构。

“Undocumented Windows 2000 Secrets”一书也以 Debug 手段给出了这个数据结构的内部结构，但是列出的结构与此有很大的不同，也许是因为版本的关系。从所列的内容看，似乎“Secrets”一书倒是正确的，因为那里所列的 EPROCESS 结构中有关于虚存的成分 Vm，是一个大小为 0x50 的数据结构，而这里没有，但是虚存(地址空间)显然是进程的主要资源，所以 EPROCESS 数据结构中理应有它的位置。由此看来，“Secrets”一书所述更接近于桌面和服务器系统的现实，而“Internals”书中所列可能更接近于不带 MMU 的嵌入式系统。而且，“Secrets”一书还在附录 C 中给出了通过逆向工程手段得到的 EPROCESS 和 PEB 等数据结构的定义(代码)，这当然是很有价值的。

那么，如果确有不同版本的 EPROCESS，这会有什么影响呢？首先，用户空间的应用程序不能直接访问内核中的 EPROCESS 数据结构，所以具体的 EPROCESS 数据结构属于内核的内部实现，只要内核中的各种成分、各个环节都配套成龙，“自圆其说”，就没有什么问题，这跟 Linux 内核中一些条件编译和裁剪的效果是类似的。可是，另一方面，对于可以动态装入的.sys 模块，如果在模块中需要访问这些数据结构，那就可能有问题了，因为.sys 模块都是以二进制映像的形式提供的，不像在 Linux 中那样可以由源代码重新编译。怎么办呢？我们可以到 Windows 的 DDK 中去找找答案。

在 DDK 的.h 文件中，有函数 IoGetCurrentProcess()的申明：

```
NTKERNELAPI
```

```
PEPROCESS
IoGetCurrentProcess(
    VOID
    );
```

这个函数是内核为.sys 模块提供的支撑函数，相当于由 Linux 内核导出的函数。其返回值类型是 PEPROCESS，就是指向 EPROCESS 数据结构的指针。显然，这跟 Linux 内核中的 current 相似，调用的目的是获取当前进程的 EPROCESS 数据结构(指针)。但是，DDK 的.h 文件中却并未给出 EPROCESS 数据结构的定义，所以调用这个函数所得到的仅仅是个指针，实际上与 void*并无区别。这意味着在.sys 模块中是不允许直接访问其内部成分的。那么，.sys 模块如何使用这个指针呢？下面就是一个例子，还是在 DDK 中：

```
NTKERNELAPI
VOID
MmProbeAndLockProcessPages (
    IN OUT PMDL MemoryDescriptorList,
    IN PEPROCESS Process,
    IN KPROCESSOR_MODE AccessMode,
    IN LOCK_OPERATION Operation
    );
```

这个函数的作用是锁定某个进程的某些存储页面(不让换出)，其输入参数之一就是指向该进程的 EPROCESS 结构的指针。当然，这个函数也是由内核提供的(属于我们所说的设备驱动界面)。所以指针的提供者和使用者都是内核，只要这二者配套即可，.sys 模块在这里只不过是传递了一下，所以也不会有问题。

假定 proc 是指向进程控制块的指针，并且进程控制块中有个成份 X，是个整数，那么在 Linux 的动态安装模块中可以直接用“proc->X”访问这个成分，但是在 Windows 的.sys 模块中则只能通过类似于 get_X()、set_X()一类的支撑函数访问这个成分。将数据结构的内容跟对于这些内容的操作(method)相分离，正是“对象”与“数据结构”的区别所在。而将数据结构的内容“封装”起来，也正是微软所需要的，因为它不愿意公开这些数据结构。

对于兼容内核的开发，这意味着我们不必拘泥于采用与 Windows 完全一致的 EPROCESS 数据结构(尽管“Secrets”的附录 C 已经给出了它的定义)，一些内部的操作和处理也不必完全与之相同，而只要与 DDK 所规定的界面相符就可以了。

了解了有关的进程对象，我们可以言归正传了。

所谓创建内核中的进程对象，实际上就是创建以 EPROCESS 为核心、为基础的相关数据结构，这就是系统调用 NtCreateProcess()要做的事情，主要包括：

- 分配并设置 EPROCESS 数据结构。
- 其他相关的数据结构的设置，例如“打开对象表”。
- 为目标进程创建初始的地址空间。
- 对目标进程的“内核进程块”KPROCESS 进行初始化。
- 将系统 DLL 的映像映射到目标进程的(用户)地址空间。
- 将目标进程的映像映射到其自身的用户空间。
- 设置好目标进程的“进程环境块”PEB。

- 映射其他需要映射到用户空间的数据结构，例如与“当地语言支持”、即 NLS 有关的数据结构。
- 完成 EPROCESS 创建，将其挂入进程队列(注意受调度的是线程队列而不是进程队列)。

这里将系统 DLL、实际上是 ntdll.dll、映射到目标进程的用户空间是很关键的。这是因为，除别的、主流的功能和作用外，ntdll.dll 同时也起着相当于 Linux 中 ELF“解释器”的作用，也担负着为目标映像建立动态连接的任务。

值得注意的是，NtCreateProcess()与 CreateProcess()不同。CreateProcess()创建一个进程并使其(初始线程)运行，除非在创建时就指定要将其挂起。而 NtCreateProcess()，则只是在内核中创建该进程的 EPROCESS 数据结构并为其建立起地址空间。这只是个空壳架子，因为没有线程就谈不上运行，调度的目标是线程而不是线程。而且，对 NtCreateProcess()的调用还有个条件，那就是目标映像已经被映射到一个存储区间(Section)中。

第三阶段：创建初始线程

如上所述，进程只是个空架子，实际的运行实体是里面的线程。所以下一步就是创建目标进程的初始线程，即其第一个线程。

与 EPROCESS 相对应，线程的数据结构是 ETHREAD，并且其第一个成分是数据结构 KTHREAD，称为 TCB。同样，“Internals”和“Secrets”两本书中所列的 ETHREAD 内部结构有所不同，后者的附录 C 中给出了通过逆向工程得到的 ETHREAD 数据结构定义。

同样，从 Windows DDK 中申明的一些函数也可以看出，.sys 模块只是传递 ETHREAD 指针或 KTHREAD 指针(由于 KTHREAD 是 ETHREAD 中的第一个成分，二者实际上是一回事)，而不会直接访问它的具体成分。

PKTHREAD NTAPI **KeGetCurrentThread**();

NTKERNELAPI KPRIORITY
KeQueryPriorityThread (IN PKTHREAD Thread);

NTKERNELAPI LONG
KeSetBasePriorityThread (IN PKTHREAD Thread, IN LONG Increment);

NTKERNELAPI PDEVICE_OBJECT
IoGetDeviceToVerify(IN PETHREAD Thread);

此外，就像进程有“进程环境块”PEB 一样，线程也有“线程环境块”TEB，KTHREAD 结构中有个指针指向其存在于用户空间的 TEB。前面讲过，PEB 在用户空间的位置是固定的，PEB 下方就是 TEB，进程中有几个线程就有几个 TEB，每个 TEB 占一个 4KB 的页面。

这个阶段的操作是通过系统调用 NtCreateThread()完成的，主要包括：

- 创建和设置目标线程的 ETHREAD 数据结构，并处理好与 EPROCESS 的关系(例如进程块中的线程计数等等)。
- 在目标进程的用户空间创建并设置目标线程的 TEB。
- 将目标线程在用户空间的起始地址设置成指向 Kernel32.dll 中的 BaseProcessStart()

或 BaseThreadStart()，前者用于进程中的第一个线程，后者用于随后的线程。用户程序在调用 NtCreateThread()时也要提供一个用户级的起始函数(地址)，BaseProcessStart()和 BaseThreadStart()在完成初始化时会调用这个起始函数。ETHREAD 数据结构中有两个成份，分别用来存放这两个地址。

- 设置目标线程的 KTHREAD 数据结构并为其分配堆栈。特别地，将其上下文中的断点(返回点)设置成指向内核中的一段程序 KiThreadStartup，使得该线程一旦被调度运行时就从这里开始执行。
- 系统中可能登记了一些每当创建线程时就应加以调用的“通知”函数，调用这些函数。

第四阶段：通知 Windows 子系统

Windows、确切地说是 Windows NT、当初的设计目标是支持三种不同系统的应用软件。第一种是 Windows 本身的应用软件，即所谓“Native”Windows 软件，这是微软开发 Windows NT 的真正目的。第二种是 OS/2 的应用软件，这是因为当时微软与 IBM 还有合作关系。第三种是与 Unix 应用软件相似、符合 POSIX 标准的软件，那是因为当时美国的军方采购有这样的要求。不过实际上微软对后两种应用的支持从一开始就是半心半意的，后来翅膀长硬了，就更不必勉为其难了。但是，尽管如此，当初在设计的时候还是考虑了对不同“平台”的支持，即在同一个内核的基础上配以不同的外围软件，形成不同的应用软件运行环境，微软称之为“子系统（Subsystem）”。于是，在 Windows 内核上就有了所谓“Windows 子系统”、“OS/2 子系统”、和“POSIX 子系统”。当然，时至今日，实际上只剩下 Windows 子系统了。

那么，所谓子系统是怎样构成的呢？“Internals”书中阐明了 Windows 子系统的构成，说这是由下列几个要素构成的。

一、子系统进程 csrss.exe。包括了对下列成分和功能的支持：

- 控制台(字符型)窗口的操作。面向控制台/终端的应用本身不支持窗口操作(例如窗口的移动、大化/小化、遮盖等等)，但是又需要在窗口中运行，所以需要有额外的支持。
- 进程和线程的管理。例如弹出一个对话窗，说某个进程没有响应，让使用者选择是否结束该进程的运行，等等。每个 Windows 进程/线程再创建/退出时都要向 csrss.exe 进程发出通知。
- DOS 软件和 16 位 Windows 软件在(32 位)Windows 上的运行。
- 其它。包括对当地语言(输入法)的支持。

这个进程之所以叫 csrss，是“C/S Run-time SubSystem”的意思，csrss 是 Windows 子系统的服务进程。其实三个子系统都是 C/S 结构，但是 OS/2 子系统的服务进程称为 os2ss，POSIX 子系统的服务进程称为 Psxss。之所以如此，据“Internals”说，是因为最初时三个子系统的服务进程是合在一起的，就叫 csrss，后来才把那两个子系统移了出来另立门户，但剩下的还继续叫 csrss。

二、内核中的图形设备驱动、即 Win32k.sys 模块。其功能包括：

- 视窗管理，控制着窗口的显示和各种屏幕输出(例如光标)，还担负着从键盘、鼠标等设备接收输入并将它们分发给各个具体应用的任务。
- 为应用软件提供一个图形函数库。

三、若干“系统 DLL”，如 Kernel32.dll、Advapi32.dll、User32.dll、以及 Gdi32.dll。

上述的第二个要素 Win32k.sys 原先也是和 csrss.exe 合在一起的，这部分功能也由服务进程在用户空间提供。应用进程通过进程间通信向 csrss 发出图形操作请求，由 csrss 完成有

关的图形操作。但是后来发现频繁的进程间通信和调度成了瓶颈，所以就把这一部分功能剥离出来，移进了内核，这就是 Win32k.sys。这一来，对于一般的 32 位 Windows 应用而言，留给 csrss、或者说必须要通过 csrss 办的事就很少了。但是，尽管如此，在创建 WIndows 进程时还是要通知 csrss，因为它担负着对所有 WIndows 进程的管理。另一方面，csrss 在接到通知以后就会在屏幕上显示那个沙漏状的光标，如果这是个有窗口的进程的话。

注意这里向 csrss 发出通知的是父进程、即调用 CreateProcess()的进程，而不是新创建出来的进程，它还没有开始运行。

至此 CreateProcess()的操作已经完成，从 CreateProcess()返回就退出了 kernel32.dll，回到了应用程序或更高层的 DLL 中。这四个阶段都是立足于父进程的用户空间，在整个过程中进行了多次系统调用，每次系统调用完成后都回到用户空间中。例如，在第二阶段中就调用了 NtCreateProcess()，第三阶段中就调用了 NtCreateThread()，而整个创建进程的过程包括了许多次系统调用(有些系统调用属于细节，所以上面并未提及)。

其实 Linux 的进程创建也不是一次系统调用就可完成的，典型的过程就包括 fock()、execve()等系统调用，但是在 Windows 上就更多了。这跟 Windows 的整个系统调用界面的设计有关。以用户空间的内存分配为例，Linux 的系统调用 brk()只有一个参数，那就是区间的长度，但是 Windows 的系统调用 NtAllocateVirtualMemory()却有 6 个参数，其第一个参数是 ProcessHandle，这是标志着一个已打开进程对象的 Handle。这说明什么呢？这说明 Linux 进程只能为自己分配空间，而 Windows 进程却可以为别的进程分配空间。或者说，在存储空间的分配上 Linux 进程是“自力更生”的，而 Windows 进程却可以“包办代替”。

这对于系统设计的影响可能远超读者的想像。就拿为子进程的第一个线程分配用户空间堆栈而言，既然 Linux 进程(线程)只能为自己分配空间，而用户空间堆栈又必须在进入用户空间运行之前就已存在，那就只好在内核中完成用户空间堆栈的分配。相比之下，Windows 进程可以为别的进程分配空间，于是就可以由父进程在用户空间中为子进程完成这些操作。这样，有些事情 Linux 只能在内核中做，而 Windows 可以在用户空间做。有些人称 Windows 为“微内核”，这或许也是个原因。而 Windows 的 CreateProcess()中包含着更多的系统调用，也就很好理解了。

现在，虽然父进程已经从库函数 CreateProcess()中返回了，子进程的运行却还未开始，它的运行还要经历下面的第五和第六两个阶段。

第五阶段：启动初始线程

新创建的线程未必是可以被立即调度运行的，因为用户可能在创建时把标志位 CREATE_ SUSPENDED 设成了 1。如果那样的话，就需要等待别的进程通过系统调用恢复其运行资格以后才可以被调度运行。否则现在已经可以被调度运行了。至于什么时候才会被调度运行，则就要看优先级等等条件了。而一旦受调度运行，那就是以新建进程的身份在运行、与 CreateProcess()的调用者无关了。

如前所述，当进程的第一个线程首次受调度运行时，由于线程(系统空间)堆栈的设置，首先执行的是 KiThreadStartup。这段程序把目标线程的 IRQL 从 DPC 级降低到 APC 级，然后调用内核函数 PspUserThreadStartup()。

最后，PspUserThreadStartup()将用户空间 ntdll.dll 中的函数 LdrInitializeThunk()作为 APC 函数挂入 APC 队列，再企图“返回到”用户空间。Windows 的 APC 跟 Linux 的 signal 机制颇为相似，相当于用户空间的“中断服务”。所以，在返回用户空间的前夕，就会检查 APC 函数的存在并加以执行(如果存在的话)。

于是，此时的 CPU 将两次进入用户空间。第一次是因为 APC 请求的存在而进入用户空

间，执行 APC 函数 LdrInitializeThunk()，执行完毕以后仍回到系统空间。然后，第二次进入用户空间才是“返回”用户空间。返回到用户空间的什么地方呢？前面已经讲到，返回到 Kernel32.dll 中的 BaseProcessStart()或 BaseThreadStart()，对于进程中的第一个线程是 BaseProcessStart()。至于用户程序所提供的(线程)入口，则是作为参数(函数指针)提供给 BaseProcessStart()或 BaseThreadStart()的，这两个函数都会使用这指针调用由用户提供的入口函数。

第六阶段：用户空间的初始化和 DLL 的连接

用户空间的初始化和 DLL 的连接是由 LdrInitializeThunk()作为 APC 函数的执行来完成的。

在应用软件与动态连接库的连接这一点上，我们已经看到，不管是 Linux、Windows、还是 Wine，都是一致的，那就是在用户空间完成：

- Linux 的.so 模块连接由“解释器”在用户空间完成。“解释器”相当于一个不需要事先连接的动态库，因为它的入口是固定的。“解释器”的映像是由内核装入用户空间的。
- Windows 的 DLL 连接由 ntdll.dll 中的 LdrInitializeThunk()在用户空间完成。在此之前 ntdll.dll 与应用软件尚未连接，但是已经被映射到了用户空间。函数 LdrInitializeThunk()在映像中的位置是系统初始化时就预先确定并记录在案的，所以在进入这个函数之前也不需要连接。
- Wine 的动态库连接分两种情况。一种是 ELF 格式的.so 模块，另一种是 PE 格式的 DLL。二者的连接都是在用户空间完成的，前者仍由 ELF 解释器 ld-linux.so.2 完成，后者则由工具软件 wine-kthread 完成。后者的具体调用路径是：

  main() > wine_init() > __wine_process_init() > __wine_kernel_init() >
  wine_switch_to_stack() > start_process() > LdrInitializeThunk()

  这在“Wine 的二进制映像装入和启动”那篇漫谈中已经讲到过了。注意这里最终完成 DLL 连接的函数也叫 LdrInitializeThunk()，显然 Wine 的作者对于 Windows 的这一套是清楚的。

通过以上的叙述，我们可以看到 Windows 的进程创建过程与 Linux 有较大的不同，但是装入 PE 映像和实现 DLL 连接的过程却与 Linux 的对应过程相似，只是把“解释器”集成到了“系统 DLL”里面，并且是作为 APC 函数执行的，其他就没有太大的区别了。但是，如果跟 Wine 的 PE 映像装入过程相比，则显然 Wine 的过程(见“Wine 的二进制映像装入和启动”)是比较复杂、效率也比较低的。