"一个框架、两个界面"
是 Linux 兼容内核开发的主体

毛德操

我们开发 Linux 兼容内核的目标是让 Windows 应用软件可以直接在这个内核上运行，更确切地说是在以这个内核为核心的操作系统上运行。同时，还要让为 Windows 而开发的一些设备驱动模块也能装入这个内核运行。这一方面是因为针对 Windows 开发的设备驱动模块在数量和品种上都远远多于 Linux；另一方面是因为有愈来愈多的应用软件需要跟专用的设备驱动模块配套运行，缺了这设备驱动模块就运行不起来。只有同时实现了这两个目标，才能说 Linux 的内核兼容了 Windows 内核。那么，为了要达到这两个目标，我们必须对 Linux 内核作些什么修改和扩充呢？本文就是要对此作一分析，并回答这个问题。

首先，一段封闭的程序、如果既不调用库程序、也不作系统调用，是跟操作系统无关的，只要所在机器的 CPU 跟这段程序编译时的目标 CPU 相符就可以了。这是因为，以 PC 机为例，这段程序编译后所生成的所有指令全都是 x86 机器指令，而且所有这些指令全都可以正常运行，它的所有跳转指令和子程序调用指令的目标都不会越出这段程序的边界。其次，即使需要调用库程序，只要不直接或间接进行系统调用，就也跟操作系统无关，因为调用库程序在效果上就相当于扩大了这段程序的地盘、推进了它的边界。所以，只要不作直接或间接的系统调用，应用程序的运行就跟操作系统无关，因而可以在任何操作系统上运行。然而，一般而言，这样的程序是不能作为一个独立的应用、独立的进程运行的，因为独立的进程至少总得要输入/输出，这就离不开系统调用了。

那么应用软件怎样进行系统调用呢？大家都知道，这是通过陷阱指令实现的，在 x86 处理器上是 int 指令。事实上，Linux 内核采用“int 0x80”实现系统调用，而 Windows 内核采用“int 0x2e”。这首先就使得 Windows 应用软件不能在 Linux 内核上运行。试想，Windows 应用软件企图通过“int 0x2e”进行系统调用，而 Linux 内核认为这是非法陷阱指令。由此可见，首先就得使 Linux 内核接受“int 0x2e”作为系统调用的手段之一。我们真该感谢两个内核的设计者，要是他们当初都选择了 0x80、或者都选择了 0x2e，那我们现在就不好办了。幸好这二者互不冲突。

由于所有的系统调用都通过同一条陷阱指令进入内核，就需要通过参数“调用号”来区分所要求的具体操作和功能。于是，在通过什么指令进行系统调用以外，还要考虑一共使用了多少个调用号，每个调用号对应着什么操作和功能，需要一些什么参数，结果是什么、怎么返回，有什么副作用等等。所有这些因素合在一起，就构成了一个系统调用界面。当然，Windows 的系统调用界面与 Linux 的不同。而要使 Windows 软件能在 Linux 内核上直接运行，就必须让 Linux 内核也提供一个 Windows 系统调用界面，否则就只好像 Wine 那样把 Windows 软件“嫁接”到 Linux 系统调用上来。在“嫁接”方面，Wine 已经做得很好了，但是我们知道效果还是不理想。

我们说 Windows 系统调用界面，一方面当然是说它的表现，就是 Windows 应用软件在内核外面所看到的各个具体系统调用的调用方式、参数、效果、返回的结果、以及副作用。

但是，另一方面，更重要的其实是这个界面背后的东西，即这些系统调用的实现，我们在这方面的工作量主要也正在于此。

据“Undocumented Windows 2000 Secrets”介绍，Win2k 共有 248 个常规的系统调用(有逆向工程的结果为证)。其中有的系统调用很简单，或者几乎可以直接用相应的 Linux 系统调用替代，有的却非常复杂。显然，要从零开始开发出这些系统调用是不大现实的(尽管 ReactOS 确实在这样做)，我们要做的是尽量利用 Linux 内核中的低层函数来实现这些 Windows 系统调用。在某种意义上，Wine 是在内核外面将 Windows 应用嫁接到 Linux 系统调用上，而我们的兼容内核则要在内核里面将 Windows 系统调用嫁接到 Linux 的诸多内核函数上。之所以在内核里面嫁接比在内核外面嫁接更好，一方面是因为内核函数的“粒度”较小，内核函数与系统调用之间的关系有点像汇编语言与高级语言之间的关系。另一方面是因为内核空间是统一的，可以提供全局的视野。例如，在内核中可以看到所有进程的进程控制块(PCB)；而一出内核，到了用户空间，进程之间就互相隔离了。所以，凡是要求有全局视野的操作，就应该在内核中实现。

除这些常规的系统调用外，Win2k 还有 639 个图形界面(GUI)系统调用。这等于是把 X11 搬到了内核里面。在 WinNT 4.0 以前，Windows 也像 Linux 一样，由用户空间的图形服务进程来实现对图形和视窗的操作和管理，那样当然效率比较低。从 WinNT 4.0 开始，就把图形操作和视窗管理移到了内核里面，成为一个可安装模块 win32k.sys。这对于提高图形操作的速度很有意义，否则游戏软件的运行就不会有那么流畅(当然，后来又有了 DirectX，进一步提高了图像/图形操作的速度，那是后话)。所以，严格地说，所谓 Windows 系统调用界面也包括这些图形操作调用。不过实现这些调用的“原材料”不再是 Linux 内核函数，而是 X11 中的相关函数。当然，事情有轻重缓急，完全没有必要在一开始就把这六百多个调用也一锅煮。实际上，即使是 248 个常规系统调用的实现，也没有必要一步到位。

过了系统调用界面这道关，许多系统调用就可以实现了，例如创建进程、取系统时间等等这些，在 Linux 内核里面就可以完整地实现。但是，一旦涉及输入/输出，涉及设备驱动，就又可能有问题了。问题不在于常规的设备驱动，例如键盘、鼠标、硬盘之类常用设备的驱动 Linux 内核中本来就有，只要把系统调用嫁接过去就行了。问题在于一些比较特殊的设备，特别是新出现的设备。这些设备的制造商往往只为 Windows 提供设备驱动模块、即.sys 文件。如果不能把这样的.sys 模块装入 Linux 内核并让它们正常运行，对 Windows 应用的兼容就不可能是完整的。

将.sys 模块装入 Linux 内核不难，因为 Linux 的动态安装模块与之并无本质的不同。难的是使其在 Linux 内核中正常运行。要达到这个目标，一是要将装入的模块往上跟有关的系统调用衔接，往下(可能)跟中断响应机制挂上钩；二是要为模块的运行提供一个特定的环境。事实上，.sys 模块之所以可以被装入 Windows 内核并正常运行，正是因为 Windows 内核满足了这两个条件。显然，现有的 Linux 内核不具备这样的条件，它所满足的是 Linux 动态安装模块的运行条件，这跟两个系统调用界面的差别是相似的。所以这也成了兼容内核开发的一个重要组成部分，也代表着相当的工作量。

看看 Windows 内核怎样来满足这两个条件，就知道我们要做些什么了。

首先是怎样使.sys 模块跟有关的系统调用和中断响应挂上钩、接上口。换言之，怎样将.sys 模块纳入内核中的设备驱动框架，包括把具体的模块放在框架中的什么位置上，跟上下左右相邻的模块或部件怎么交互。在 Windows 内核中，这主要是 I/O 子系统的事，具体主要体现在两个方面：一方面是模块装入以后首先要向 I/O 子系统登记，或向上一层的设备驱动登记，那也就是间接地向 I/O 子系统登记。另一方面，当有关的系统调用发生时，正是通过 I/O 子系统转化成对于具体设备驱动的逐层调用和返回。所以，I/O 子系统代表着一个设备驱动框架的主体。不过，I/O 子系统并不是设备驱动框架的全部，.sys 模块可能还要跟中断响应挂上钩。例如，一个.sys 模块中的一部分可能是作为中断服务程序运行的，另一部分可能是作为 bh 函数(在 Windows 中称为 DPC)运行的，还有一部分则受 I/O 子系统的驱动。Windows 的.sys 模块都是针对 Windows 的设备驱动框架开发的，既然我们要把这些.sys 模块装入 Linux 内核运行，就必须在 Linux 内核中构建起这么一个框架。具体地，就是要在 Linux 内核中实现 Windows 的 I/O 子系统，Windows 的中断响应/服务机制，以及有关的一些辅助性的成分。

Windows 设备驱动框架解决了对.sys 模块的装入和使用。如果每个模块都是封闭、独立的，不需要环境的支持，那就够了。但是，事实上几乎不存在这种封闭、独立的设备驱动模块，几乎每个模块的运行都需要得倒环境的支持。举例来说，一个设备驱动模块在运行中可能需要分配缓冲区。可是设备驱动模块本身是不具备这个能力的，因为它并不掌握任何可动态分配的内存资源。于是，就只好请求内核予以分配，这就是对环境的依赖。为此，操作系统内核要“引出(export)”大量的内核函数供动态安装模块调用。这个函数集合的大小、以及每个具体函数的调用条件、参数、作用、返回值、副作用，就构成了具体内核的设备驱动支撑界面。显然，不同的内核有不同的设备驱动支撑界面，就好像不同的内核有不同的系统调用界面。要让 Windows 的.sys 模块在 Linux 内核中正常运行，就必须在 Linux 内核中提供 Windows 的设备驱动支撑界面。与系统调用界面不同，Windows 设备驱动支撑界面的定义是公开的(否则第三方就无法为其开发设备驱动模块了)，定义于 Win2k 或 WinXP 的 DDK、即“设备驱动开发包”。Win2k 的 DDK 中定义了 2000 来个支撑函数，但是常用的也就是几百个、甚至更少。当然，微软公开的只是这个界面的定义，而不包括它的实现。

那么，怎样实现这些支撑函数呢？还是老办法，把它们转化成、或者嫁接到相应的 Linux 设备驱动支撑函数上去。

值得一提的是，Windows 对网络设备的驱动另外定义了一个框架，称为 NDIS；并专门提供了一个 NDIS 设备驱动支撑界面，使 NDIS 设备驱动模块只需要(并且只应该)调用 NDIS 界面上的支撑函数。这样，网络设备的驱动就自成体系，有了一个大框架中的小框架。不过，无论是 NDIS 的框架还是支撑界面都比较小，只能算是 Windows 设备驱动框架和界面的附庸和扩充，而且许多 NDIS 支撑函数实际上就是一些 Windows 设备驱动函数的简单包装，所以我们将 NDIS 归入 Windows 设备驱动的框架和界面。

综上所述，我们开发 Linux 兼容内核，主要就是在 Linux 内核中实现这一个框架和两个界面，就是：Windows 设备驱动框架、Windows 系统调用界面、以及 Windows 设备驱动支

撑界面。这就是 Linux 兼容内核开发工作的主体。当然，还有不少零碎的、辅助性的开发工作要做，但相比之下工作量就比较小了。

不过，说“一个框架、两个界面”是开发工作的主体，这是就兼容内核本身而言，而不是就整个操作系统而言。要让 Windows 应用软件正常运行，在内核外面还需要有一些 DLL、即动态连接库的支持，以及一些配套服务进程的支持。如果单纯从技术的角度看问题，那么确实只要有了兼容内核就行了，因为我们可以把 Windows 上的 DLL 和配套服务程序都拷贝过来。可是这不仅是个技术问题，更是个法律问题，我们不能去侵犯微软的版权。所以，这些 DLL 和配套服务程序也都需要开发，幸好 Wine 已经为我们做了许多这方面的工作，只是需要对其中的四个 DLL、即所谓“四大件”作些修改。须知把 Windows 系统调用嫁接到 Linux 系统调用的正是这四大件：kernel32.dll、gdi32.dll、user32.dll、还有 ntdll.dll，现在则又要使它们回归本来面目，把扭曲了的东西再扭曲回去。

还应看到，Windows 的技术和应用本身也在发展，这些发展更多地体现在 DLL 和服务进程中，例如 OLE、COM/DCOM、.NET 这些技术大多是通过 DLL 和服务进程实现的。Wine 虽然已经做了很多这样的 DLL 和服务程序，但是这方面的开发也许永远不会完，不过那是细水长流的事了。