Linux 兼容内核的三个重要源泉

毛德操

我们要开发、构建的 Linux 兼容内核并非无源之水，也不需要从零开始“重新发明轮子”。正如牛顿所说要“站在巨人肩膀上”一样，我们也应该充分利用别人已经取得的成就、特别是开源社区已经取得的成就。

除 Linux 本身以外，兼容内核主要的源泉有三个，那就是 Wine、NdisWrapper、以及 ReactOS。三者都是在 Sourceforge 立项的开源项目，源代码可从 www.sourceforge.net 下载。

下面对三个源泉分别作一简单的介绍。

Wine

按 Wine 项目自己的说法，Wine 是“Wine Is Not an Emulator”的缩写。这使人不免想起 GNU 的“Gnu is Not Unix”。许多人对此可能一笑了之，觉得这是文字游戏或者幽默。可是笔者却觉得这背后其实自有深意。GNU 说它不是 Unix，意思是说虽然它基本上就是 Unix，或者非常像 Unix，可是你不能按 Unix 的各种标准来要求它，有些地方它与真正的 Unix 是点差别的。这是一种“有言在先”式的声明。而 Wine 为什么说它不是仿真器呢？这反映了 Wine 的设计者生怕别人误以为 Wine 是个仿真器。这主要是因为“仿真器”这个词容易使人误解，以为是对 CPU 机器指令的仿真，那是效率非常低的。所以这实际上是在申辩，说 Wine 的效率不低。当然，Wine 的效率比普通意义上的仿真确实要高得多。

那么 Wine 到底是什么呢？

- 对于 Windows 应用软件而言，Wine 为其提供对 Windows 运行环境的仿真，所以 Wine 也可以理解为“WIN Emulator”。这也正是 Windows 应用软件能够在 Linux 上运行的条件与原因。不过这种仿真并不是对 CPU 指令的仿真，而是对 Win32 API 函数调用的仿真。
- 对于 Linux 及其内核而言，Wine 是内核与 Windows 应用软件之间的一个中间层。它一方面为 Windows 应用软件提供各种动态连接库(DLL)，使应用软件通过 Win32 API 进行的库函数调用得以落实，一方面将应用软件和动态连接库原本对 Windows 内核所作的系统调用翻译成对 Linux 内核的系统调用，把它们转嫁到 Linux 内核上来。
- 对于许多作为 Windows 操作系统组成部分的动态连接库、服务/守护程序、工具程序而言，Wine 是这些软件在 Linux 上的移植、实际上是仿制。在“关于 Linux 兼容内核的知识产权问题”一文的第二部分中，笔者把 Windows 上的软件分成五类，这些软件都属于第五类，即由微软开发，又理应属于操作系统、跟 Windows 相捆绑的那部分软件。

Wine 完全不触及 Linux 内核，所有的操作都是在内核外面进行。有些操作本来应该

在内核中实现，但是因为不触及内核，就只好设法在内核外面、即用户空间中设法补偿。但是，在内核外面用 Linux 系统调用来实现 Windows 系统调用，就好像要用一种高级语言来实现另一种高级语言一样(比方说，用 Cobol 来实现 Fortran)，往往会导致相当笨拙的实现，有些甚至根本就实现不了。这是因为，Linux 或者 Windows 的每一个特定的系统调用就好像高级语言的一种语句，我们固然可以把它看成是个黑盒子，但是要让两个这样的黑盒子在输入参数和条件，计算结果和副作用等等各方面都完全一样是很困难的。诚然，Linux 的系统调用是很丰富、很灵活、“表现力”很强的，有点像是 C 语言，这给通过 Linux 系统调用实现 Windows 系统调用提供了一个良好的基础。但是，即便如此，也还存在不少的困难。下面我们通过一个例子加以说明。

Windows 系统在打开文件等等操作中返回一个 Handle(翻译成“句柄”，笔者觉得不是很好，但又想不出更好的词)，语义上这是用于一个指针数组的下标。每个进程都有这么一个数组，所以 Handle 原则上不是全局的、而是局限于具体进程的。也许读者马上就会联想到打开文件号，二者似乎完全相同，因而应该可以用 Linux 的打开文件号来实现 Windows 的 Handle。不过 Windows 的 Handle 并不是只用于文件，而是用于所有各种“对象(Object)”，这且不说，这里要说的是个更大的麻烦。在创建子进程时，Windows 内核允许有选择地“遗传”Handle。当然，Linux 内核在创建子进程时也可以遗传打开文件号，这似乎又是一样的。可是麻烦来了。在 Windows 系统中，打开一个文件时就通过参数规定，这个打开了的文件以后是否可以遗传；在创建子进程时，则又通过参数规定，是否要遗传那些事先规定可以遗传的已打开文件、即 Handle(见“Advanced Windows”，p17)。应该说，这是个不错的主意。而 Linux 呢？遗传的是 1、2、3 三个打开文件号。对内核有所了解的朋友不难想到，要在 Linux 内核中作一些扩充，以实现 Windows 的这种遗传机制，是不难的，甚至是很容易的。可是，如果要在内核外面实现呢？读者不妨动动脑筋。如果有很简洁的方法，不妨通知一下笔者，让笔者有个表达钦佩和祝贺的机会。

至于设备驱动，既然不触及内核，自然就不可能解决设备驱动的问题。当然，对于 Linux 上已经有了相应设备驱动程序的那些外设(例如硬盘)，可以把 Windows 应用程序对这些外设的操作“重定向”到相应的 Linux 设备上来。可是问题在于那些在 Windows 上有驱动模块(.sys 文件)，而 Linux 上还没有相应驱动模块的那些设备，对这些设备怎么办呢？

那么 Wine 的效率(性能)究竟如何？好在 Wine 的代码是公开的，我们可以作些分析。当然，实验结果具有更高的权威性，但是有些事是那样地明显，以致我们可以无须列举实验结果作为证明。另一方面，分析也可以使我们更好地理解和解释观察到的现象。不过，现在这里不是引用和分析、讲解源代码的地方，笔者将另外撰文分析 Wine 的结构和操作，现在只是粗线条地作一些说明和讨论。

我们以写文件操作为例。在 Linux 上，这是通过系统调用 write()完成的，但是应用软件一般是通过 C 库函数 write()或 fwrite()间接地启动系统调用 write()，这大家都已经很熟悉，无需多说的了。Win32 API 上与 C 库函数 write()相对应的是 WriteFile()，而 Windows 内核与

系统调用 write()相对应的是 NtWriteFile()。从总体上看，这两对对应物之间还是颇为相似的，似乎可以直接用 write()来实现 WriteFile()。那么在 Wine 中是怎么实现的呢？下面是笔者整理出来的 NtWriteFile()伪代码描述：

```
NtWriteFile()
{
    向服务进程发送一个请求并等待答复；    //将 Handle 转换成打开文件号
    write()；                              //实际的写文件操作
    向服务进程发送一个请求并等待答复；    //释放打开文件号
}
```

这里涉及三次系统调用(其中两次为进程间通信、一次为实际的写文件操作)，以及四次进程调度/切换(每次进程间通信都需要正、反两次进程调度和切换)。而在理想的情况下，则只需要一次系统调用(实际的写文件操作)，不需要进程调度和切换。写文件如此，读文件也如此，事实上所有的文件操作都如此。Wine 的效率如何，由此可见一斑。这就难怪 Wine 的开发者那么在意别人以为它是“仿真器”了。为改进 Wine 的效率，有人提议把服务进程搬入系统空间、使其成为内核线程，但是那只是提高了进程切换的速度和免去了系统调用时的空间切换，却并不能改变整个操作的模式、并不能减少进程调度的次数，因而并不能从根本上解决问题。

Wine 之所以要这样做，当然自有其苦衷(读者绝对不应怀疑 Wine 开发人员的智商)。限于本文的篇幅，笔者在这里不作分析，而只是说明：如果是在内核中实现 NtWriteFile()，那就根本不需要有这个服务进程了。

不过，虽然 Wine 本身并不是一个很好的解决方案，它对于兼容内核却有着特殊的重要性。这是因为内核并非一个操作系统的全部，在内核外面还需要许多系统软件、特别是动态连接库的配合。离开 Windows 系统的大量 DLL，应用软件就无法运行。这些“系统 DLL”是微软作为 Windows 的一部分、与内核捆绑在一起销售的。其实这些系统 DLL 大多可以从网上下载，但是那样就会涉及版权问题。而 Wine 恰恰为我们开发了(并且还在继续开发)许多这样的 DLL。所不同的是：几个底层的、直接进行系统调用的 DLL 所进行的是 Linux 系统调用而不是 Windows 系统调用，这是我们需要恢复其本来面目的。另一方面，如上所述，由于 Wine 不涉及内核，许多本来应该放在内核中实现的功能和机制不得不放在服务进程中，从而牺牲了效率。随着兼容内核的开发，这些功能和机制将被移入内核。但是 Wine 的存在为我们提供了一条渐进的开发路线，而许多有关的代码则或可借鉴、或可利用。

所以，随着兼容内核的开发和日益完善，对于 Wine 所提供的资源需要在分析的基础上区别对待：

- 有许多是我们可以直接拿过来用的，主要是大量上层的 DLL。

- Wine 的结构框架大体上可以利用，但是需要作较大的修正。
- 贴近系统调用界面的几个 DLL，特别是底层的、直接进行系统调用的 DLL，则需要加以移植改写，以恢复其 Windows 系统对应物的本来面目。

鉴于 Wine 对兼容内核的重要性，笔者将另外撰写专文介绍和分析 Wine 的代码。

NdisWrapper

在各种设备驱动程序中，网络(网口)的驱动有着许多特殊性。网络驱动的层次非常分明，其高层是“协议栈”，底层才是对接口设备如网卡的驱动。高层的协议栈部分是很固定、很少改变的，而网络接口设备本身则特别五花八门，每一种不同的网络接口设备都需要提供自己的驱动程序，称为“微端口”(Miniport)驱动。而且，网络驱动常常会要求在协议栈和微端口驱动之间再插入中间层，以提供过滤、加密、流量控制、负载均衡等等功能。考虑到网络接口驱动的特殊性，简化微端口驱动和中间层的开发，提高这些模块的可移植性，也为了提高网络处理的效率，微软定义了一个“网络驱动程序接口规格(Network Driver Interface Specification)”，缩写为 NDIS。有了 NDIS 以后，微端口驱动和中间层的开发就不必(也不应)考虑常规的 Window 设备驱动框架和界面，而只要符合 NDIS 的规定就可以了。读者可以把 NDIS 设想成一个虚拟的网络接口设备，这个设备本身并不是物理的网络接口，而仿佛只是一个提供了若干插槽的智能化的背板，具体的网卡则插在插槽上。这些插槽是有层次关系的，要插入一个中间层时只要把它插在网卡前面就可以了。高层软件要驱动某个具体网卡时只需要跟这个智能化背板打交道即可，而具体网卡的驱动程序也只要符合这个智能化背板的规矩就行。在某种意义上，NDIS 也是一种设备驱动框架，不过是 Windows 设备框架中的一个子框架，是对 Windows 设备框架的一个扩充。为此，微软提供了一个动态安装内核模块 ndis.sys，这个模块就好像是提供插槽的背板。对于插在这个“背板”上的微端口驱动和中间层模块而言，ndis.sys 就好像是个机箱、是个包装，所以人们常常称之为“Ndis Wrapper”。这一来，Windows 内核中的网络驱动就实际上涉及两个框架、两种制式。作为 Windows 设备驱动的协议栈模块、例如 tcpip.sys，是在 Windows 设备驱动框架中，遵循 Windows 设备驱动的模式和界面。而作为 NDIS 驱动的微端口驱动，则位于 NDIS 子框架中，并遵循 NDIS 设备驱动的模式和界面。当然，后者比前者简单。不过，这里又有特例：如果一个微端口驱动实际上并不直接驱动目标设备，而需要途经另一个设备才能访问目标设备，例如对于插在 USB 口上的网卡，则因为所途经的设备遵循 Windows 设备驱动的模式和界面，这个微端口驱动的“下沿”也得遵循 Windows 设备驱动的模式和界面，这就又回到了 Windows 设备驱动框架中。

开源软件 NdisWrapper 实际上就是 ndis.sys 的移植，从 Windows 内核到 Linux 内核的移植。

在 Linux 内核中，协议栈是在 socket 内部实现的，而 socket 在形式上不同于一般的设备驱动，有着一套独立的系统调用。所以，NdisWrapper 的上沿与 socket 相衔接，下沿则与

Windows 微端口驱动相衔接，原则上与 Linux 本身的设备驱动框架无关。所以，从本质上说，NdisWrapper 在 Linux 内核中营造了一个 NDIS 设备驱动框架。如上所述，NDIS 设备驱动与 Windows 设备驱动不是一回事。不过，如果目标设备的驱动需要途径别的设备、如 USB，则又要回到 Windows 设备驱动框架，所以 NdisWrapper 的代码中也包含了一些本该属于 Windows 设备驱动框架的函数。但是那并不等于说 NdisWrapper 已经提供了 Windows 设备驱动框架。事实上，Windows 设备驱动框架的主体是 Windows 内核中的“I/O 子系统”。没有这个子系统，下层的设备驱动模块就不能“通顶”、即不能与系统调用界面接上口。在 Windows 内核中，ndis.sys 也不直接通顶，而需要经过 tdi.sys 和 tcpip.sys 才能与系统调用接上口。而在 Linux 内核中，由于 socket 系统调用的存在，NdisWrapper 经过 socket 通顶，但是那显然只适用于网络设备。

笔者与朋友谈论在 Linux 内核中实现 Windows 设备驱动框架时，常常有人以为 NdisWrapper 已经实现了这个目标，这都是因为对其了解不深而引起的误解。

还应该说明一点，NdisWrapper 及其所“包装”的 Windows 微端口驱动是供 Linux 应用软件(而不是 Windows 应用软件)使用的。这当然是我们所需要的，但是还不够。我们还需要让在 Linux 上运行的 Windows 应用软件也能通过 NdisWrapper 使用安装在 Linux 内核中的 Windows 微端口驱动。

那么 NdisWrapper 对于兼容内核的意义在那里呢？显然，作为对 Windows 设备驱动框架的扩充，NDIS 本来就是必须的，否则就无法利用数量众多的 Windows 网卡驱动模块。而 NdisWrapper 的代码只要稍加修改就可以跟 Windows 设备驱动框架接上口。另一方面，它也提供了一些本该属于 Windows 设备驱动框架的函数，可在实现 Windows 设备驱动框架时加以利用或借鉴。但是，就总体而言，这种利用和借鉴的价值远不能与 ReactOS 相比，因为 NdisWrapper 所实现的毕竟只是一个局部。

ReactOS

ReactOS 项目是一群 Windows“发烧友”创立的。他们坚信 Windows 是个比 Linux 更好的操作系统，问题仅在于源代码不公开，所以他们要来开发一个开源的 Windows 内核。令人诧异的是：他们的开发基本上是从零开始的。

Windows 与 Linux 孰优孰劣恐怕是个永远争论不出结论的话题，但是开发出一个开源的 Windows 内核，让用户可以有多一个选择，那无疑是很有意义的，他们的勇气和毅力更是使人起敬。这个项目的进展也很快，现在已经初具规模。不过，考虑到 Windows 内核本身的复杂性，ReactOS 离实用还是有相当大的距离。在笔者看来，认定要开发一个 Windows 内核，却坚持从 0 开始编程、而刻意不去利用现成的 Linux 代码资源，似乎有点意气用事，也不必要地增加了开发的工作量与难度。而我们的兼容内核，则首先是继承 Linux 的资源，再反过来补上 Windows 内核与 Linux 内核之间的差异，而且即使是对于这些差异也要尽量利用和借鉴现有的开源代码资源、包括 ReactOS。

与兼容内核一样，ReacOS 也只是个内核，跟完整的操作系统相比，中间还缺着以 DLL 为主的系统软件这一层。在这个问题上，ReactOS 的策略和我们一样，也是利用 Wine。另外，即使在内核中，ReactOS 也不完全排斥采用别的开源软件，例如其 socket 机制的实现就采用了 Oskit 的有关代码(笔者感到不解的是为什么不采用 Linux 的有关代码)。

对于我们来说，ReacOS 的可贵之处在于为我们提供了一个可资参考的样本。Windows 内核的源代码不公开，ReactOS 的源代码却是公开的。虽然 ReactOS 不等于 Windows，但这至少让我们看到另一群人对 Windows 内核的理解，以及他们的具体实现。

那么我们可以在什么程度上利用 ReactOS 的代码呢？那要看具体情况：

- 对于 Windows 设备驱动框架的实现、包括 I/O 子系统的实现，ReactOS 的有关代码对我们有极其重要的意义，有些代码甚至可以直接加以利用。
- 对于 NDIS 的实现，可以将 NDISWrapper 与 ReactOS 中的相应代码作一比较，一方面是择优录用，一方面是互相补充、互相借鉴。
- 对于系统调用界面，ReactOS 的相应代码可资借鉴。
- 对于进程管理和资源管理，可以参照和比较 ReactOS 和 Wine 服务进程的有关代码，然后在 Linux 内核中加以实现。
- 对于设备驱动界面(环境)，可以根据 DDK 的定义逐一考察 ReactOS 的实现(如果已经实现的话)，再在 Linux 内核中寻找其对应物或近似物。然后加以参照比较，以确定具体应该如何实现。

总的来说，如果某一项功能既已在 ReactOS 中实现，又可以利用 Linux 内核中的某些资源加以实现，那么我们更倾向于利用 Linux 的资源。这是因为一般而言 Linux 的资源已经经受了更多的考验。另一方面，兼容内核毕竟是个 Linux 内核，而不是把两个内核堆积在一起。像对 Wine 一样，笔者将另撰专文介绍和分析 ReacOS 中的有关代码。

最后还要说明，笔者说兼容内核有三个重要的源泉，绝不是说就只有这么三个源泉。海纳百川，有容乃大，只要对兼容内核的开发有用、有助，我们对任何源泉都不应排斥，只是需要择优采用。