开发 Linux 兼容内核的策略与路线

毛德操

笔者自从提出 Linux 兼容内核的构想以后，听到了不少反响。其中支持者固然不乏其人，如开源软件推进联盟主席陆首群先生、倪光南院士、还有 OSDL 的平野正信先生，都是支持的。但是无庸讳言，对此不以为然的意见也有不少。有趣的是，这些不以为然的意见往往分成两个极端。一个极端说，Linux 兼容内核要达到的目标已经达到了，因此根本就不用多此一举。说已经达到，是因为在用户空间我们已经有了 Wine，而 NdisWrapper 则已经解决了把 Windows 设备驱动装入 Linux 内核的问题。另一个极端呢，则是说兼容内核的难度太大，根本就不可能成功，所以只是一个“梦”。显然，在谈论开发路线图之前首先应该回答这两种意见。幸好它们正好互相构成回答。

首先，正如笔者在另几篇文章中所说，Wine 只是在逻辑上、功能上基本解决了用 Linux 内核来模拟/仿真 Windows 内核的问题，但是性能上是无法令人满意的，而且说基本上解决其实也很勉强。至于 NdisWrapper，它所解决的是一些网络设备(网卡)的驱动，而不是普遍意义上的设备驱动。更何况 NdisWrapper 设备驱动的上层只能是 Linux 的 Socket，而不能与诸多文件操作的系统调用挂上钩。实际上，退一步说，即使 NdisWrapper 和 Wine 真的已经分别解决了 Windows 设备驱动和应用软件在 Linux 内核上的跨平台运行，也还需要有人把这二者整合起来。有许多公司只是把一些开源软件收集(而不是整合)拢来，做成发行版，不是就号称成了“高科技公司”吗？所以，要使 Windows 的应用软件和设备驱动能在 Linux 上高效地运行，从而真正为广大用户所接受，兼容内核的开发应该说是很有必要，而且也不是那么轻而易举。

至于说难度太大，那么 Wine 和 NdisWrapper 的存在和发展恰恰为兼容内核技术上的可行性提供了参考。诚然，Wine 在某些方面很不理想，但那正是因为要避开内核而导致的后果，许多在用户空间很难解决的问题一到了内核里面就可以豁然开朗。或者可以反过来说，Wine 在用户空间都可以基本解决的问题，到了内核里面就更好解决了。当然，涉及内核的设计和编程比之用户空间在难度(复杂度)上要高得多，但是“难”是个相对的概念，谁能说清到底难到什么程度就根本不能做了呢？再说这也毕竟不是登月、不是哥德巴赫猜想。另一方面，Wine 所提供的许多高层 DLL 为兼容内核的实际使用提供了条件，或者可以说是解决了后顾之忧。如果说 Wine 毕竟不涉及内核，因此还不足为凭，那么 NdisWrapper 可确实为我们在内核中构建设备驱动框架和支撑界面提供了参考，实质上也是一个可行性证明。

认为难度太大的人还有一个顾虑，就是 Windows 的代码是不公开的，藏在黑盒子中，光凭几本书能得到多少信息、如何就能开发出跟 Windows 兼容的内核？这种顾虑当然也有一些道理，但是 ReactOS 又在这方面给我们提供了参考。ReactOS 以零为起点从头开发，Wine 只在内核外面做文章，尚且都能在一定程度上达到设计目标，而我们站在它们的肩膀上，又有 Linux 内核作为原材料，至少条件比他们好多了。

当然，把 Wine、NdisWrapper 和 ReactOS 作为参考意味着我们需要吃透、或者至少基本上理解它们的代码。为此笔者将陆续写一些分析文章在本网站上推出，起个抛转引玉的作用。

总之，兼容内核的开发既不是唾手可得，也不是难于上青天，既不能一蹴而就，也不至于遥遥无期。说起来还是那句老话：战略上藐视困难，战术上重视困难。

后发跟进、逐步逼近

我们开发兼容内核不能采取一步到位、而应采取逐步逼近的策略。以系统调用界面为例，我们完全可以先搁置那些用于 GUI、即 win32k.sys 的扩充系统调用，即便是对于 248 个常

规系统调用也可以分期分批地实现。实际上，我们甚至并无必要追求一个完整的实现。工程上有一个所谓 20/80 原理，说是 20%的工作量往往可以实现 80%的功能，而剩下的 20%功能却往往需要 80%的工作量才能实现。如果我们的兼容内核可以支持 80%的 Windows 应用，剩下的 20%慢慢从长计议也无不可。再说，Windows 本身也在发展，今天还是“完整”的实现，明天就可能是不完整的了。所以，我们可以采取在一定距离后面跟进的策略。只要 Windows 还存在、还在发展，这样的跟进就永远不会完。这种后发跟进、逐步逼近的策略决定了我们的开发必定是一个螺旋式的渐进开发过程。就是说：从不同成分之间的关系看，是螺旋式的发展；从同一成分内部看，则是渐进的发展。

那么，这个螺旋式渐进开发过程的起点是什么呢？起点就是 Linux+Wine。随着开发的进行，Linux 的内核逐渐变成兼容内核，我们姑且以 Linux+表示；而 Wine 则逐渐演变成一个按 Windows 系统调用界面定制并且优化了的 Wine，我们姑且称之为 Wine’。所以整个开发过程就是：

(Linux + Wine) => … => … => (Linux+ + Wine’)

起点 Linux+Wine 显然是可以运行的，开发过程中的每一步都实现一组有限的目标，每一步的结果都应该是一个可以运行的、更逼近 Windows 的、可以发行的版本。

对 Linux 内核的修改原则上以动态安装模块的形式实现，尽可能不改变 Linux 内核原有的代码，必要时当然也可以打一下补丁。

兼容内核开发的主体是一个框架、两个界面。如果按它们在内核中的位置从上到下排序，那就是：系统调用界面，设备驱动框架，以及设备驱动支撑界面。下面分别加以讨论。

系统调用界面的开发

系统调用界面的实现有个“门槛”，那就是内核的进入/退出机制，即系统调用时的空间切换机制，不跨过这道门槛就谈不上系统调用界面。不过这个机制的实现并不复杂，因为我们要实现的本质上是 Linux 内核上的系统调用，从而这实际上就是 Linux 系统调用的空间切换机制，所不同的只是：

- Linux 原有的系统调用都是通过指令“int 0x80”进入内核的；现在需要为 Windows 系统调用添上通过指令“int 0x2e”进入内核。
- Linux 原来有个系统调用跳转表，这是一个以 Linux 系统调用号为下标的函数指针数组；现在需要添上一个 Windows 的系统调用跳转表，就是以 Windows 系统调用号为下标的函数指针数组；以后还要再添上一个用于 GUI 的扩充系统调用跳转表。

可见，这道门槛的实现并不困难。

有了(系统调用时)内核的进入/退出机制，就可以来考虑系统调用界面本身、即系统调用内核函数的集合了。对于 Win2k，这就是“Windows NT/2000 Native API Reference”等参考书中所述的二百多个函数。这些函数的实现实际上构成了 Windows 的文件系统、I/O 子系统、进程管理子系统、内存管理子系统，等等。

这里的工作量可就大了，但是可以(也应该)分期分批地予以实现。

- 248个系统调用，数量上与Linux相当。
- 系统空间的进入/退出可以重用Linux的相应代码。
- 进程/线程管理。需要融合两个系统的进程/线程管理机制。
- 进程间通信(包括LPC)。基本上可以嫁接到Linux的相应机制上。
- 存储管理。需要扩充Linux的内存管理介面。
- 文件系统。基本上可以嫁接到Linux的文件系统上。
- 设备驱动。连接到 WDM 设备驱动框架。

不光是分期分批，就是同一个系统调用也可能需要分几次来完成。有的系统调用有很多可选项，其中有些可选项实际上很少用到。对这样的系统调用，我们就可以先实现其基本功能，然后慢慢完善。

怎样实现这些系统调用内核函数呢？

- 有些系统调用可以嫁接到相应的 Linux 系统调用。
- 有些系统调用可以部分地重用相应 Linux 系统调用的代码。
- 有些系统调用在Linux中没有对应物,需要借助Linux内核中的低层函数予以实现。
- 可以借鉴 Wine 的代码，在一定程度上是把 Wine 的部分代码移入内核并加以优化。

显然，这个系统调用界面上的文件系统应该嫁接到 Linux 的文件系统，I/O 子系统属于我们要实现的 Windows 设备驱动框架，进程管理子系统应该嫁接到 Linux 的进程管理，内存管理子系统应该嫁接到 Linux 的内存管理，如此等等。

Wine 中的底层四大件，即 kernel32.dll、user32.dll、gdi32.dll、以及 ntdll.dll，原来把所有的系统调用都引向 Linux 系统调用。随着开发的进展，每实现一个 Windows 系统调用，就应该在 Wine 这一层上把原先的系统调用“重定向”到这个 Windows 系统调用上来。为此，可以对 DLL 的装入/连接机制加以扩充，以实现“虚拟连接”、即重定向的功能。例如，可以在装入/连接下层DLL时对于需要从下层DLL引入的每一个函数都先检查一个映射文件，看是否需要把这个函数重定向到另一个 DLL 文件中的另一个函数名。这样，每实现一个 Windows 系统调用以后，只要修改这个映射文件，并提供另一组底层 DLL 就可以了。发展到最后，Wine 原有的底层 DLL 就为新的(同名)DLL 所取代。这时候，Wine 就变成了 Wine’。

如前所述，这些系统调用函数可以分期分批实现，其中首先需要实现的是与文件系统和设备驱动有关的系统调用，以便在此基础上搭建 Windows 设备驱动框架。

除常规的系统调用外，还有一个专用于图形界面(GUI)的扩充系统调用界面，这是因为微软把原先在用户空间实现的 GUI 支持(类似于 X11)移到了内核中，成为一个动态安装模块 win32k.sys。这个扩充系统调用界面与常规界面合用同一个进入/退出机制，只是系统调用号全都大于 1000。为此，内核中另外需要一个函数跳转表。至于具体系统调用函数的实现，则基本上就是把 X11 服务进程移植到内核中来，对此我们可以暂时搁置，留待将来再来讨论和实现。

设备驱动框架的开发

基本的设备驱动框架对应着 Win2k 内核中的 I/O 管理,以及电源管理、即插即用等机制，也涉及部分对象管理、系统配置、和安全管理方面的功能。其中最主要的是 WDM 层次式设备驱动机制的实现。这个框架上面与有关文件操作的系统调用(open()，close()，read()，write()，ioctl()等)相衔接，中间实现基于“IO 请求包”IRP（IO Request Packet）的设备驱动机制，下面则融入 Linux 内核的中断响应/服务机制、包括“软中断”即 bh 函数的执行机制。主要包括:

- 设备驱动程序的动态装入和连接。
- IRP 的生成和传递、以及设备驱动程序的启动、同步、和终结。
- 将设备驱动程序的中断服务登记嫁接到 Linux 的中断机制上,将设备驱动程序所关心的 Windows 内核运行状态映射到 Linux 内核的运行状态上。
- 将设备驱动程序的 DPC 请求嫁接到 Linux 的 bh 函数机制上。

基本设备驱动框架的实现基本上是个“有”与“无”的问题，一旦这个框架成了形，这方面剩下的工作就不很多了。所以，设备驱动框架的实现“门槛”比较高，技术上的难度也相对较大。不过，我们有 ReactOS 的代码和 NdisWrapper 的部分代码可资借鉴。

此外，网络设备(网卡)、即 NDIS 设备的驱动既有其特殊性，又应该看作是 Windows 设备驱动的一部分，所以设备驱动框架的实现应该包含 NDIS 的实现。不过这倒是得来全不费功夫，NdisWrapper 的代码基本上可以直接加以利用。

设备驱动框架的开发与系统调用界面的开发并不一定是顺序的。在系统调用界面上有关文件操作的系统调用尚未实现之前，不妨先借用 Linux 的有关系统调用作为对特殊设备或/proc 节点的驱动，就像在大桥尚未造好之前先架一座便桥一样。

基本的设备驱动框架到位以后，如果单纯从技术角度看，所有的.sys 文件(即 Windows 的设备驱动模块)就都可以装入 Linux 内核运行。但是，有些.sys 模块是由微软连同其操作系统捆绑发行的，微软拥有这些.sys 文件的版权，Linux 的用户不能直接拿过来用。这一类.sys 模块基本上都是用于一些标准的、基本的、常用的外部设备，总的来说可以分成几个大类(class)，例如磁盘、USB、图形设备、网络设备等等，它们的调用界面既有公共的部分，又各有其特殊之处。一般而言，Linux 实际上已经具备相应的功能，只是需要将 Linux 内核(包括设备驱动模块)中的这些函数和数据结构与具体.sys 的调用界面之间架起桥梁。这一部分工作在形式上与系统调用界面和/或设备驱动界面的实现有相似之处，并且也像系统调用界面的实现那样可以从 Wine+Linux 开始，例如 Wine 结合内核中 HPFS 和 NTFS 的实现实际上就是把 Linux 的磁盘文件驱动适配到了 Windows 的磁盘文件访问。

但是也可能有一些微软的.sys 模块在 Linux 内核中找不到对应物，那就需要仿制了。不过这方面的工作没有必要在一开始就进行，而可以推迟到实现了基本的设备驱动框架，并且部分地实现了两个界面以后再回过来渐进地开发。

设备驱动支撑界面的开发

根据微软提供的 WinXP DDK，内核中可供设备驱动程序调用的函数(以及全局变量)大约有 1000 个（比 Win2000 多)。在这 1000 个左右的内核函数中，有一些属于“IO 管理器”IoManager 和“对象管理器”ObManager，因而属于设备驱动框架，其余的则属于设备驱动界面。

- 有些资源的调用/引用可以映射(重定向)到 Linux 内核中的对应物上。
- 有些资源的调用/引用可以嫁接(适配)到 Linux 内核中的对应物上。
- 有些资源的调用/引用需要另加实现。

设备驱动支撑界面的实现也有个起码的门槛。例如分配缓冲区的函数以及相当于 spinlock()的函数就是必须的。幸运的是 ReactOS 和 NdisWrapper 已经为我们提供了一个基本的支撑函数集合，只是 ReactOS 的代码不是建立在 Linux 内核基础上的，所以需要进行一些适配和优化的工作。在这个基本集合的基础上，每多实现一组函数，兼容内核的覆盖面就加大了一些。

从功能的角度看，多数设备驱动界面函数(以及数据结构)在 Linux 内核中都有对应物，例如，上述用于分配缓冲区的函数以及 spinlock()就是，需要做的就是把所需的支撑函数和数据结构落实到相应的 Linux 内核函数和数据结构上，这里面当然有些适配的工作。也有些函数在 Linux 内核中没有较为接近的对应物，那就要用 Linux 内核中的各种素材加以搭建。

从开发的时序看，支撑界面的开发应该在设备驱动框架到位以后，或者至少是二者同步开发。这是因为，离开了设备驱动框架，支撑界面的存在就失去了意义，同时也失去了测试的手段。

其它开发

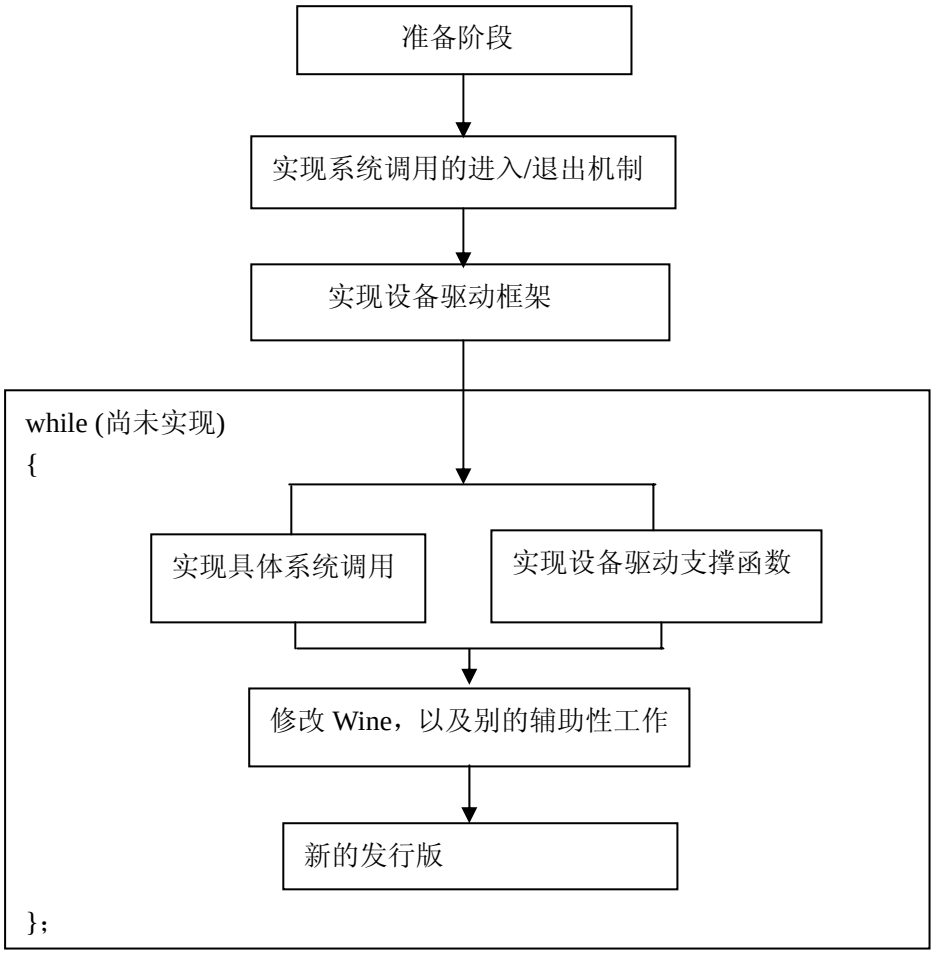
除一个框架、两个界面以外，还有一些辅助性的开发要做，例如：

- 对 Wine 的修改
- 某些.sys 模块的仿制
- 与注册表的交互
- Unicode 在字符串和文件名中的使用
- 从 Windows 风格的文件路径名到 Linux 路径名的映射
- 从 Windows 风格的设备名到 Linux 路径名的映射

还可以举出不少。这些工作量合在一起也不容小看，但是从总体上看、从难度和技术含量看、毕竟不是主体。

开发路线图

综合上述各点，我们的开发路线图将大致上如图所示：



首先是准备阶段。在这个阶段中笔者将陆续写出一批文章，对 Windows 与 Linux 的异同

作一些介绍和比较，对 Wine、NdisWrapper、ReactOS 的代码作一些介绍和分析，旨在引起网友们的参与和讨论，更欢迎大家到我们网站上发表有关的研究成果和见解。这样，经过一段时间的探讨，参与研发的人对于要做的事情应该可以了然于胸。同时，对于开发的团队，无论是专职的还是业余的，也需要有个形成和组织的过程，我们的网站也需要一段时间来逐步完善。此外，我们还要制定出一个编码标准和一套代码管理办法(包括代码的提交、评审、录用、存档、管理、奖励等各方面的办法)，为具体的代码编写做好准备。这么一个准备阶段显然是很有必要的，估计这个阶段将延续到明年 1 季度，我们希望原则上以春节为界。

开始具体的开发以后，第一件事就是要实现系统调用界面的进入/退出机制。这前面已经讲了。接下来就是设备驱动框架。前者是不可分割的，是有或者无的问题。后者稍微好一些，可以先搞个最低限度的基本框架，再慢慢充实。如果把准备阶段作为整个项目的第一阶段，那么这就是第二阶段。

有了这些以后，下面的第三阶段就可以渐进开发了，所以图中把具体系统调用和设备驱动支撑函数的开发放在一个循环体内。如前所述，这里的每一轮循环都实现一组有限的目标，每一轮的结果都应该是一个可以运行的、更逼近 Windows 的、可以发行的版本。这个阶段也许永远不会完，永远要循环下去，其结果是愈来愈逼近 Windows。另一方面，我们的兼容内核是 Linux 兼容内核，而 Linux 也在发展。于是，兼容内核将一边跟随 Linux“水涨船高”地发展，一边又愈来愈准确、愈来愈广泛地兼容 Windows 应用软件。

我们的开发目标是，在一年之内要进入第三阶段并至少完成第一轮循环，即基本实现若干最重要的系统调用和设备驱动支撑函数，取得一些可见的效果。这样，一年内我们就能有一个可以发行的原型。反过来，这个原型又进一步构成整个项目的可行性证明，可以进一步增强开发人员和公众对项目的信心。至于更具体的安排，则有待于准备阶段中进一步的细化。