漫谈 Wine 之二:

Windows 的文件操作

毛德操

对 Wine 的研究不应该孤立地进行，而应该把它放在 Windows(注意，除特别说明者以外，都是指 WinNT)和 Linux 两个操作系统、特别是两个内核的差异的背景下来开展。这里面的道理很简单：如果两个内核没有什么差异，或者二者之间存在着简单的映射，那 Wine 的存在就没有什么意义、没有多大空间了，从而兼容内核也就没有什么必要了。所以，无论是对于 Wine 的研究，还是对于兼容内核的开发， Windows 和 Linux 的对比研究有着特别重要的意义。

进一步，我们着眼于两个系统的差异，又特别应该着眼于那些从用户角度“能见度”较高的差异，即对于系统的功能、性能有较大影响的差异，而对于“能见度”较低的差异就可以先放一放、甚至可以不太关心。举例来说，在内存管理方面，即虚存空间的划分与使用、页面的换入/换出这些问题上，两个内核也存在着差异。但是这些问题一般而言对功能、性能的影响不大，我们就可以把它放到“运动后期”再来考虑。反之如文件操作则“能见度”相当大，就应该优先考虑。其实，如果我们回顾 Wine 项目的开发历程，也正好可以看到这么一种策略。

不过，有兴趣于 Wine 或兼容内核、或者来我们这个网站走走看看的人，对于 Linux 多少都已经有了一些了解和理解，所以我这里将集中谈论 Windows 的一些特点，并与 Linux 中相关的对应物作一些比较；而对 Linux 的这些对应物就不作介绍了，有需要的读者可参考“Linux 内核源代码情景分析”和其它有关著作。

这一篇先讲文件操作。

我这里讲 Window 与 Linux 在文件操作方面的差异，还不是指它们在文件系统的结构与格式方面的差异，而是指应用软件可以对文件作些什么操作、可以怎样来使用文件和文件系统。也就是说，是指应用软件与操作系统之间的界面、即 API。事实上，这才是更为根本、更具结构性、因而更显著更重要的差异。

从表面上看，文件操作不外乎打开文件/关闭文件、读、写、移动指针、冲涮缓冲区，还有就是对文件加锁/解锁这些操作，变也变不到哪儿去。这里面似乎不会有什么根本的、结构性的差异。

然而，事情不是那么简单。读者看了我下面列举的一些差异(尽管并不详尽)就会明白这些差异实在并不简单，从而可以对 Wine 的系统结构和开发兼容内核的主张有更深刻的理解。

一、打开文件号

要对一个文件进行操作，就得先打开这个文件，在这一点上两个系统(内核)是一致的。我们知道，Linux 的“已打开文件号”实际上是用于一个指针数组的下标。与此相似，在 Windows 中虽然不称为“已打开文件号”而称为“Handle”(一般译作“句柄”，但是窃以为不如“把手”更为传神)，实质上也是指针数组的下标。这样看来，在这一点上似乎也是一致的。

然而有着一个重要的区别。

作为 Linux“已打开文件号”的下标，是用于一个“打开文件表”的下标，每个进程有一个“打开文件表”。这个表(数组)中的每一项都代表着一个已经被本进程打开的文件。从

Unix 时代开始，设备被看作文件，因而这或者是文件、或者是设备(也包括“虚拟设备”)。

可是，作为 Windows“句柄”的下标，却是用于一个“打开对象表”的下标。虽然也是每个进程一个“打开对象表”，但是每个表项所代表的却不仅仅是文件或设备了，它也可以代表着内核中的某个数据结构。所谓对象，就是 Object，实际上就是“物件”、“东西”的意思，文件是物件，设备是物件，窗口是物件，数据结构也可以是物件。说“数据结构”，读者可能还不太敏感，我们来点具体的：比方说，进程的“进程控制块”、即 PCB，就可以是个被打开的对象。所以，在 Windows 的系统调用界面上，既有 NtCreateFile()，也有 NtOpenProcess()。前者用于打开或创建一个文件，后者则用于打开某个进程、实际上是这个进程的 PCB，如果是创建进程则另有 NtCreateProcess()。打开某个进程干什么呢？例如，可以进一步通过系统调用 NtTerminateProcess()杀灭这个进程。当然，还可以有别的操作，这留到别的漫谈中再来讨论。

此外，一个进程同时打开的文件数量毕竟有限，而可能要同时打开的“对象”数量可就大了。

这么一比较，二者的差异就明显了。Unix/Linux 将设备看作文件，而 Windows 进一步将文件看作对象。显然后者更为一般化，是前者的推广。

单纯由这个因素引起的功能和性能上的差异也许并不大，但是与别的因素结合在一起所造成的差异就相当可见了。

二、已打开文件的遗传

当一个进程创建一个子进程时，可以将它的一些已打开文件“遗传”给所创建的子进程。这样，子进程从一开始就取得了对这些文件的访问权和上下文。这一点上 Windows 和 Linux 都是如此。

可是，Windows 内核的这种遗传机制却有着与 Linux 不同的特点，可以归纳成下列要素：

1） 在打开一个文件(实际上是更为广义的“对象”，下同)的时候，可以说明是否允许将这个已打开文件(如果成功打开的话)遗传给子进程。

2） 在创建一个子进程的时候，可以规定是否将那些允许遗传的已打开文件遗传给所创建的子进程。显然，如果需要遗传的话，Windows 内核会扫描父进程的已打开对象表，把所有允许遗传的表项复制到子进程的已打开对象表中。

总之，这是有选择的遗传和继承，分两次作出选择。

那么 Linux 怎么样？

在 Linux 中，子进程的创建是分两步走的。第一步是通过系统调用 fork()创建一个线程(共享父进程的用户空间和别的资源)。然后是第二步，通过系统调用 execve()使新创建的线程变成进程，开始独立拥有自己的用户空间和别的资源，并开始走自己的路。另一个系统调用 system()则只是把 fork()和 execve()、还有 wait4()、组装在一起，因此实际上还是分两步走。在第一步中的遗传/继承是全面的，父进程所有的已打开文件都遗传给子进程(线程)。而第二步，其实 Linux 也支持有选择的遗传和继承。Linux 内核的数据结构 struct files_struct 中有个位图 close_on_exec_init，位图中的遗传控制位(标志位)决定着相应的已打开文件在执行 execve()等系统调用时是否关闭、也即遗传与否。应用程序可以通过系统调用 ioctl()设置具体已打开文件的遗传控制位，以决定具体已打开文件的遗传特性。

于是，Windows 的打开文件系统调用可以这样实现：

```
NtOpenFile()
{
    ……
    fd = open();
```

```
    if (允许遗传)
        ioctl(fd, FIONCLEX);            // exec 时不关闭
    else
        ioctl(fd, FIOCLEX);             //// exec 时关闭
}
```

显然，至此为止，这样的实现还是很简洁的。

可是且慢，Windows 允许在创建子进程时再作一次选择。如果选择不遗传，那么所有的已打开文件都不遗传。而 Linux 却不提供这一次选择，这就是一项比较显著的“核内差异”了。

那么我们怎么来“核外补”呢？似乎可以这样：先让它遗传，然后再在子进程中把已经遗传下来的文件关闭掉。可是，这样的实现就谈不上简洁高效了。问题在于子进程并不知道到底有哪些已打开文件被遗传了下来，所以只好来一个格杀勿论：

```
    for(ch=0; ch< OPEN_MAX; ch++)
        close(ch);
```

这里的 OPEN_MAX 是 Linux 允许一个进程拥有已打开文件的最大数量，定义为 256。显然，这个办法不好。

读者也许会想，如果我们在用户空间也提供一个位图、作为内核中本进程的 close_on_exec 位图在用户空间的副本，再让 NtOpenFile()在设置内核中的位图时也设置一下用户空间的副本，最后让子进程根据这个副本位图有针对地关闭文件，这样不就可以提高效率吗？然而问题在于：一般而言，父进程用户空间的内容是不能遗传给子进程的。实际上，作为一条准则，凡是要遗传给子进程的东西，就不能只是保存在父进程的用户空间，而必须保存在别的什么地方，一般是内核中或是另一个进程(例如服务进程)中。

相对而言，一个差强人意的实现是在父进程的用户空间保留一个副本，然后这样：

```
NtCreateProcess()
{
    ……
    if (不遗传)
    {   //根据副本 close_on_exec 位图
        for(每个允许遗传的已打开文件)
            ioctl(fd, FIOCLEX);             //暂时设置成不遗传
    }
    fork()
    if (在父进程中)
    {
        if (不遗传)
        {   //还是根据副本 close_on_exec 位图
            for(每个允许遗传的已打开文件)
                ioctl(fd, FIONCLEX);        //恢复原状
        }
    }
    ……
```

}

如果选择不遗传，就在 fork()之前先把有关的已打开文件暂时改成不让遗传，然后在 fork()以后由父进程把这些已打开文件的遗传控制位复原。这里的问题是允许遗传的已打开文件可能很多，所以可能要调用 ioctl()很多次。

这是个可以差强人意的实现，却还是谈不上简洁和高效。

我以前说过，在用户空间用 Linux 的系统调用来模拟实现 Windows 的系统调用，就好像是用一种高级语言实现另一种高级语言，这就是一个实例。

相比之下，如果我们不死守“核外补”，而在 Linux 内核中设法解决，那就简单多了。至少我们可以在 ioctl()中为 close_on_exec 位图的设置增添一种“批处理”操作，即一次系统调用设置(本进程的)整个位图。可是，既然动了内核，那又何必又要在用户空间保持一个位图副本，又是前后两次调用 ioctl()？为 fork()增添一个参数，把是否需要遗传的信息直接带下去不就行了？然而 fork()的调用界面属于标准，是不容许随便更改的。既然如此，为什么不干脆增加一个新的系统调用呢？进一步，既然增加一个新的系统调用，那就应该与 NtCreateProcess()的界面相同。这样，就转到我们“一个框架、两个界面”中的“Windows 系统调用界面”的思路上来了。

其实就这一项“核内差异”而言，上述的方法虽不能说简洁，但是至少还可行，下面这个问题就更麻烦了。

三、文件访问的跨进程复制

Windows 不仅支持父子进程之间对于已打开文件(对象)有选择的遗传/继承，还支持跨进程的复制。这种跨进程复制可以在没有“血缘”关系的进程之间进行。Windows 为此提供了一个系统调用 NtDuplicateObject()，我们看一下它的调用界面：

```
NTSTATUS
NTAPI
NtDuplicateObject(
   IN HANDLE    SourceProcessHandle,
   IN HANDLE    SourceHandle,
   IN HANDLE    TargetProcessHandle,
   OUT PHANDLE    TargetHandle    OPTIONAL,
   IN ACCESS_MASK    DesiredAccess,
   IN ULONG    Attributes,
   IN ULONG    Options);
```

这里 SourceProcessHandle 和 TargetProcessHandle 是源和目标两个已打开进程的句柄，代表着两个进程。调用这个函数的进程本身可以是其中之一，也可以是个“第三者”。SourceHandle 是源进程的某个已打开文件(对象)的句柄，这个系统调用的目的就是要把源进程对这个已打开文件的访问通道复制给目标进程，或者说把源进程访问这个文件的上下文复制给目标进程。而 TargetHandle，则用来返回复制以后该已打开文件在目标进程中的句柄。不过，调用时也可以通过 TargetHandle 对目标句柄(打开文件号)给出一个建议，所以是 OPTIONAL。至于其余三个参数，则我们在这里并不关心。在典型的应用中有三个进程卷入此项操作，源和目标两个进程、以及执行此项操作的进程、可以是任意进程(只要访问权限

允许)，而不必是父子或兄弟。

我们知道，Linux 有个系统调用 dup()，但那是在同一个进程的内部。已打开文件的遗传与 pipe()、dup()、close()等系统调用结合在一起，就构成了在父进程与子进程之间建立管道连接、以及输入/输出重定向等重要的手段。除父子进程之间以外，进程间的管道连接也可建立于两个兄弟进程之间，但那必须由它们共同的父进程发起和参与。

相比之下，Windows 当然也可以通过句柄复制来实现类似的功能，但是更进了一步，因为那可以发生在任意进程之间。

更重要的是，Linux 进程之间管道连接的建立实际上与 fork()捆绑在一起，只能在创建子进程的过程中建立。一旦子进程业已创建，便无能为力了，因而就子进程而言这是静态的。而 Windows 的句柄复制，却可以在任何时候动态地进行。

Unix/Linux 进程间的管道连接以及输入/输出重定向等手段有着重要的意义。对这些手段的运用甚至导致了“Unix 程序设计环境”的出现和程序设计方法上的革新。这已经是广大 Unix/Linux 程序员所熟知的了。而 Windows 的句柄复制显然又前进了一步，加以巧妙运用就可以发展出一些功能更强、灵活性更好的手段。例如，在一个由许多进程通过管道连接构成的一个应用系统中，可以由一个总控进程动态地改变这些进程间的连接，从而改变整个系统的结构和形态（不过，要达到这个目的也并不是非要有句柄复制这个功能不可，但是句柄复制显然很简洁/高效）。

对于两个内核在这方面的差异，要在核外加以补偿是很难的。显然，要实现跨进程的句柄复制，调用 NtDuplicateObject()的进程就必须能(直接或间接地)访问其它两个进程的打开文件表，而打开文件表在内核中。另一方面，只要 CPU 运行于用户空间，它就只能访问当前进程自己的用户空间，而不能访问别的进程的空间。唯一的例外是通过共享内存，即把一部分页面同时映射到多个进程的空间中。然而用户空间共享内存的使用是不大安全的，因为任何一个应用程序的失误都有可能破坏共享内存中的内容，从而破坏整个系统的运行。可是，即便是共享内存，那也只是用户空间的，根本访问不到内核空间去(至于内核空间，本来就是共享的，但只有进入内核以后才能共享)。由此可见，对于这样的“核内差异”，要“核外补”是不容易的。怎么办呢？

那么，可不可以干脆就不补呢？问题在于，既然 Windows 提供了句柄复制，就会有应用软件来使用它；如果对此不加补偿，就使有些 Windows 应用软件不能在 Linux 上运行。当然，用到了句柄复制的应用软件到底有多少，那倒是可以调查研究的。所以，这取决于你的目标以及对多种因素的综合考虑。

不过“核外补”的办法还是有的。那就是采用文件服务器(文件服务进程)的方法。在这种方法里，所有的文件实际上都是由文件服务进程打开和拥有的，而“客户”进程只是名义上打开并拥有各自的那些已打开文件，“客户”进程对文件的操作都要委托服务进程代理。这实质上就是把对于已打开文件的所有权与享用权区分了开来。于是，(在 Linux 上运行的)所有 Windows 进程的已打开文件全都属于同一个进程、即服务进程所有，“客户”进程并不直接打开任何文件。这样就使各“客户”进程本身在内核中的打开文件表失去了效用(从而无需去访问它们)。而服务进程，则在其用户空间为“客户”进程另行维持各自的幻影式的打开文件表。从效果上看，这是把内核与各进程打开文件表之间的关系和操作往上平移/提升到了用户空间，变成服务进程与各“客户”进程的“幻影”打开文件表之间的关系与操作。

这样一来，应用软件层面的所谓跨进程句柄复制，对于 Linux 内核而言却只是服务进程内部的句柄复制，类似于 dup()。这就实现了对于句柄复制的“核外补”。而且，这一来别的问题也可以随之而得倒类似的解决。例如有条件遗传/继承的问题，就因为不再使用“客户”进程在内核中的打开文件表，而改成使用由服务进程为各个“客户”进程维持的“幻影”打

开文件表，从而变得简单了。

显然，服务进程不失为实现“核外补”的好方法，Wine 就是采用了这个方法。但是也有缺点，那就是效率问题。所以，一般在“正常”的情况下，文件服务进程只在下列条件下使用：

1. 在物理上与应用软件分离的文件服务器中。例如，由文件服务器和无盘工作站构成的应用系统就是这样。这样的系统往往是文件操作不太繁重、对性能要求也不高的。另一方面，如果客户端自身带有磁盘，则文件服务器只是用作辅助或后备，因此对效率的要求也不高。
2. 在一些采用微内核的嵌入式系统中。此类嵌入式系统通常很少有文件操作，一般只是在要改变系统配置或运行策略时才需要读/写文件，因而文件操作的效率无关紧要。
3. 出于别的考虑、例如集中监控，而愿意以牺牲部分的效率为代价。

可是，我们的目的是要在 Linux 内核上高效运行 Windows 软件，采用文件服务进程的方法在所有这三个方面都与我们的目的背道而驰：

1. 我们的应用软件与文件存储器物理上并不分离。明明磁盘就在本地，本可以直接打开、直接操作，却也要舍近求远绕道服务进程。以 Wine 为例，就好像是把文件服务器和无盘工作站合并放在同一台机器上，降低了效率。
2. 在我们的目标应用、即桌面应用中，文件操作是最重要、最频繁的操作，其效率至关重要。
3. 我们不能以牺牲部分的效率作为代价，因为这恰恰成为许多用户不愿意使用 Wine 的一个重要原因。

正是在这个意义上，我说 Wine 作为解决方案有点怪。如果我们追究其原因，则问题出在“核内差异核外补”这个策略上。我认为，只要我们死守“核外补”，就实际上很难在 Wine 的基础上再作显著的改进与提高(当然，局部的、小幅的改进还是可能的)。其实，我在另一篇漫谈中将要讲到，在“核外补”方面，Wine 已经干得很不错了。但是，如果我们转变策略，来一个“核内差异核内补”，那就“退一步海阔天空”，许多问题都可以高效地解决，而那正是兼容内核的目标之一。