漫谈 Wine 之四：

内核差异核内补

毛德操

明眼人一看便知，我的前几篇漫谈都是在为“内核差异核内补”这个话题作铺垫。本来，我的计划是还要再铺垫上几篇，然后才转入这个话题，可是现在已经有网友在我们论坛上摩拳擦掌，说要动手干了。在这样的形势下，我觉得有必要调整一下计划，先讲讲“内核差异核内补”、以及怎样“核内补”的话题。而别的内容，则可以留到后面慢慢来谈。

之所以要“内核差异核内补”，是因为内核的差异往往很难在核外高效地加以补偿，有些甚至可能是无法妥善补偿的。Wine 之所以性能不佳，主要就是因为走的是“核外补”的技术路线。我常常想，以参与 Wine 这个项目的人员的素质和才华，如果早一点就转向“核内补”，说不定现在桌面 Linux 的竞争力早已有了长足的进步。

我在漫谈之三中引用了“写文件”这个操作来说明 Wine 对此操作的实现效率如何的低，以及为什么这么低，并且说明了只要是“核外补”就只能这么低(小幅的改进是可能的，但是大幅的、根本性的改进则不可能)。现在，作为对比，我们仍以写文件操作为例，来看看如果是“核内补”的话可以怎样来实现，以及可以达到甚么样的效率。

首先，我们要为 Linux 内核增添一个与 NtWriteFile()等价的系统调用。至于以什么方式来实现新增的系统调用是另一回事，有的网友建议用 ioctl()，有的网友建议用 int 0x2e，那都属于实现细节，我们回头再来讨论。之所以要增添这么一个系统调用，是为了要把 NtWriteFile()所有的参数都带进内核，并且在内核中正确地使用这些参数，达到所规定的效果。

如果以 Linux 的系统调用为参照，那么最大的差异就是 NtWriteFile()带下来的是 Handle 而不是打开文件号。但是现在就不需要由服务进程进行从 Handle 到打开文件号的映射了。与进程的“打开文件表”相对应，我们应该仿照 WinNT（实际上是 ReactOS）为每个进程配备一个“Handle 表”。为此，我们可以在 Linux 的进程控制块中增加一个指针，例如：

```
struct task_struct {
    ……
    /* open file information */
    struct files_struct *files;
#ifdef UNIFIED_KERNEL
    struct handle_struct *handles;
#endif
    ……
}
```

这里的指针 files 指向本进程的“打开文件表”，而新增加的指针 handles 则指向本进程的“Handle 表”。为此，我们需要对源代码打上一个补丁。我们兼容内核的程序主体当然会做成可动态安装的模块，但是适当地打一些(少量)补丁也是必要的。不过指针 handles 也可以不放在进程控制块中，而放在 struct files_struct 数据结构中，那也是属于实现细节，例如：

```
/*
```

```
 * Open file table structure
 */
struct files_struct {
	atomic_t count;
	spinlock_t file_lock;	/* Protects all the below members. Nests inside tsk->alloc_lock */
	int max_fds;
	int max_fdset;
	int next_fd;
	struct file ** fd;	/* current fd array */
	fd_set *close_on_exec;
	fd_set *open_fds;
	fd_set close_on_exec_init;
	fd_set open_fds_init;
#ifdef UNIFIED_KERNEL
	struct handle_struct *handles;
#endif
	struct file * fd_array[NR_OPEN_DEFAULT];
};
```

至于“Handle 表”的内容，当然有 Handle 到打开文件号的映射。但是 Windows 的 Handle 不完全是用于已打开文件，也可以标志着一个进程，还可以是别的“对象”。所以，每个表项中还应该包含关于对象类型的描述，以及由此而来的具体映射目标的数据类型。例如，要是一个 Handle 标志着一个已打开文件，那么映射的目标可以是一个整数、即打开文件号，而若标志着一个进程，则映射的目标应该是个 struct task_struct 指针。

那怎么知道传下来的是 Windows 的 Handle、还是 Linux 的打开文件号呢？这很简单，从 NtWriteFile()下来的是 Handle，而从 write()下来的是打开文件号。

我在前两篇漫谈中说过，Wine 之所以要由服务进程来集中管理所有进程的已打开文件，是因为否则就不便、甚至不能实现 Windows 文件操作的一些特殊性。那么，如果我们按上述的方案来实现文件操作，是否就能够实现这些特殊的文件操作呢？回答是肯定的。

先看创建子进程时的遗传。我在漫谈之二中提到过，Linux 内核在各进程的“打开文件表”、即数据结构 struct files_struct 中也记录了各个已打开文件是否需要遗传的信息，这就是位图 close_on_exec_init，而指针 close_on_exec 则指向这个位图。问题在于 Linux 内核没有在创建子进程时再提供一次是否确实要遗传的选项，系统调用 execve()的参数表中也不包含这样的信息。可是，如果我们增添一个与 NtCreateProcess()等价的系统调用，那么它的一个参数、即布尔量 InheritObjectTable，就携带着这个信息。

在 Linux 内核的代码中，子进程在 fork()时全盘继承下来的已打开的文件是通过 flush_old_files()有选择地加以关闭的，其根据就是 close_on_exec 位图。

```
static inline void flush_old_files(struct files_struct * files)
{
	long j = -1;

	spin_lock(&files->file_lock);
	for (;;) {
```

```
            unsigned long set, i;

            j++;
            i = j * __NFDBITS;
            if (i >= files->max_fds || i >= files->max_fdset)
                  break;
            set = files->close_on_exec->fds_bits[j];
            if (!set)
                  continue;
            files->close_on_exec->fds_bits[j] = 0;
            spin_unlock(&files->file_lock);
            for ( ; set ; i++,set >>= 1) {
                  if (set & 1) {
                        sys_close(i);
                  }
            }
            spin_lock(&files->file_lock);

      }
      spin_unlock(&files->file_lock);
}
```

现在，只要改成有条件地、根据 InheritObjectTable 的值确定是否调用这个 inline 函数就行了。

事实上，对于 flush_old_files()的调用本来就是在具体格式的可执行文件装入程序中调用的，例如 ELF 格式的装入程序 load_elf_binary()、a.out 格式的 load_aout_binary()等等。而对于 Windows 的可执行程序，我们本来就需要有个针对 PE 格式的 load_pe_binary()，只要在这个程序中直接或间接实现对 flush_old_files()的有条件调用就可以了。例如：

```
#ifdef UNIFIED_KERNEL
      if (!InheritObjectTable)
#endif
            flush_old_files();
```

再看跨进程的 Handle 复制。Linux 内核本来就有本进程的打开文件号复制，即 sys_dup()。现在需要的是跨进程的 Handle 复制。在内核里面这显然不是一件困难的事，因为在内核中我们可以访问所有进程的进程控制块，当然也就可以进而访问所有进程的“Handle 表”。至于具体的程序，则一可以参考 sys_dup()的实现，二可以参考 Wine（在用户空间）的实现，三可以参考 ReactOS 中的实现，我这里就不多费口舌了。

回到前面的进程控制块数据结构中，从本质上说，指针 handles 所直接和间接指向的数据结构就是用来弥补 Linux 内核在文件系统方面的数据结构与其 WinNT 内核对应物之间的差异。而程序上的修改，则用来弥补二者在操作(包括算法)和性状上的差异。一般而言，我们只需要、也只应该补上二者之差。如果以 W 表示 WinNT 内核、L 表示 Linux 内核，那就

是(W - L)。我曾经解释，兼容内核决不是把两个内核(例如 ReactOS 跟 Linux)堆积在一起，那样的话就变成(W + L)了。

文件操作是两个内核之间的重要差异之一，而对进程/线程的管理又是另一个重要差异，对此我将在以后的漫谈中加以讨论，这里先说“核内补”的事。为了弥补二者在这方面的差异，在 Linux 的进程控制块 struct task_struct 数据结构中还应再增添一个指针，例如：

```
struct w32_task_struct    *w32_task;
```

这里的数据结构 struct w32_task_struct 就是对 struct task_struct 的补充，使得两个(或更多)数据结构之和能够实现它们在 WinNT 内核中的对应物。当然，对原有的程序也要作相应的修改和补充。这样，如果 Linux 进程控制块中的这个指针为 0，就说明本进程是个 Linux 进程(线程)，否则就是个 Windows 进程(线程)。相应地，代码中对于程序的修改和扩充，则一般都可以用指针 current->w32_task 非 0 作为执行的条件、或者条件之一。至于具体如何修改和扩充，那是另一个问题，我们以后还要讨论。

上述的两个指针，即 handles 和 w32_task，当然也可以合并成一个，就是把 handles 移到 struct w32_task_struct 中。这一般只是具体实现上的细小差别，并不影响到整个方案的实施。

两个内核在其它方面的差异，原则上也可以仿此办理。

倘若我们按这样的方案、在这样的背景下实现 NtWriteFile()，其运行效率与 Wine 在核外的实现显然不可同日而语。另一方面，我们也看到，这与简单的“把 Wine(服务进程)搬进内核”也不是一回事。

当然，作为一个中间步骤，先简单地把 Wine 的服务进程基本上原封不动地搬入内核，让它作为一个内核线程在内核中运行，从而提高一些效率，让用户尽快见到一些效果，那也未尝不可，但是我们长远的目标则必须更加远大。这里还要指出，把 Wine 服务进程搬入内核成为内核线程，所节省下来的只是切换空间(进程切换、以及进/出系统调用时)所需的时间，这毕竟是很有限的。所以，这样固然可以提高一些效率，期望值却不能太高。另一方面，把 Wine 服务进程移入内核也实际上不可能原封不动、也得作一些修改，例如必须在某些点上主动调用 schedule()进行进程调度。再说，Wine 服务进程的有些操作是因为身处用户空间、不得已而为之，搬进内核以后就没有理由继续存在，而且修改也比较容易，这样必然就有人会去作一些“顺手牵羊”式的优化。慢慢地，如果能不断“精益求精”的话，我看最终还是会发展成上面所说哪样的实现。不过先把 Wine 的服务进程搬入内核毕竟简单。作为第一步也是好的。所以，从这个意义上说，兼容内核的开发几乎不可能完全失败，因为我们可以走一条完全渐进的道路，从把 Wine 搬入内核开始，再逐步向前推进、逐步替换。这样，每向前推进一步，就可以得倒一个性能更好、兼容程度更高的版本。

实际上，对于兼容内核来说，情况还要更复杂一点。这是因为除上面所说的实现以外还需要考虑设备驱动的问题，但是那可以留待稍后再来讨论和考虑。

最后，我们再回到系统调用界面，考虑一下以甚么方式实现系统调用、即应用进程进行系统调用时怎样进/出内核的问题。对此，网友们提出了几种不同的建议。归纳起来，大致上有这么几种方式：

1. 把新的系统调用“搭载”在对一个虚拟设备的 ioctl()系统调用上。这种方式的好处是简单易行，也不会导致不兼容的问题。但是也有缺点，那就是多少要降低一些运行效率，因为每次系统调用都要到 sys_ioctl()中去转上一圈。
2. 为 Linux 系统调用界面增加一个总的 w32call()系统调用，把所有的 Windows 系统调用搭载在这上面。这样的做法不乏先例，socket 操作的所有系统调用就都搭载在

socketcall()上，所有系统 5 的 IPC 操作就都搭载在 ipc()系统调用上。这种方法也很简单易行，问题是需要取得 OSDL 的同意，否则日后可能发生系统调用号的冲突和不兼容。

3. 通过 int 0x2e 实现一套新的、独立的系统调用。缺点当然是(相对而言)不那么简单易行，但是好处也不少。

首先，从整个兼容内核的实现形式上考虑，不管采用这三种方式中的哪一种，总归都是模块加补丁。当然，如果能有无须打补丁、纯粹由模块实现的形式，那是再好不过的事，但是实际上这是不现实的，因为我们不主张将两个内核简单地“堆积”在一起。事实上，即便是堆积，也还避免不了要对现有的 Linux 内核代码打一些补丁。为说明这一点，我们先考虑对进程控制块的补丁。如上所述，我们可以把指针 handles(以及其它类似的指针)移到 struct w32_task_struct 中，从而使需要增添到进程控制块中的指针减少到只剩一个。那么是否可以连一个也不要、从而可以避免对进程控制块打补丁呢？如果单纯从功能上考虑是可以的，例如我们可以在 struct w32_task_struct 中放上进程号 pid，这样根据当前进程的 pid 就可以找到对应的 struct w32_task_struct 数据结构。但是这显然会降低性能，所以并不合适。再考虑对于文件的操作，想要充分利用现有的代码，就难免要在程序中加上一些条件语句和#ifdef，这就免不了要打补丁了。既然补丁不可避免，上述三种方式在这方面就没有多大区别。

再看实现的难易，第一种方式固然很容易，但是第三种方式、即采用 int 0x2e 的方式也并不很困难。为此，我们可以回顾一下 Linux 内核中的有关代码(参看“情景分析”，以下代码取自 Linux 内核代码 2.6 版，所以略有不同)。

```
void __init trap_init(void)
{
    ……
    set_system_gate(SYSCALL_VECTOR, &system_call);
    ……
}
```

首先在 trap_init()中设置好 int 0x80 的向量，使其指向汇编程序 system_call，这跟设置中断向量没有什么区别，这里 SYSCALL_VECTOR 的定义就是 0x80。设置好向量以后，当应用程序在用户空间执行指令“int 0x80”时，CPU 就会进入内核空间并转入汇编程序 system_call。不过，在一般的应用程序中是见不到“int 0x80”这条指令的，因为它深埋在 C 程序库的 read()、write()等等函数中。我们接下去看 system_call 的代码：

```
ENTRY(system_call)
    pushl %eax                # save orig_eax
    SAVE_ALL
    GET_THREAD_INFO(%ebp)
    cmpl $(nr_syscalls), %eax
    jae syscall_badsys
      # system call tracing in operation
    testb $_TIF_SYSCALL_TRACE,TI_FLAGS(%ebp)
    jnz syscall_trace_entry
syscall_call:
```

```
        call *sys_call_table(,%eax,4)
        movl %eax,EAX(%esp)         # store the return value
syscall_exit:
        cli                         # make sure we don't miss an interrupt
                                    # setting need_resched or sigpending
                                    # between sampling and the iret
        movl TI_FLAGS(%ebp), %ecx
        testw $_TIF_ALLWORK_MASK, %cx    # current->work
        jne syscall_exit_work
restore_all:
        RESTORE_ALL
```

代码中的宏操作 SAVE_ALL 和 RESTORE_ALL 分别为保存现场和恢复现场的操作。从用户空间带下来的“系统调用号”在寄存器%eax 中。将%eax 和 nr_syscalls 相比较，就是在检查系统调用号是否越界。如果系统调用号在合理的范围中，就调用 sys_call_table()，在那里 CPU 会根据系统调用号在一个跳转表(指针数组)中取得相应的地址、并进入具体系统调用的服务程序。然后，当 CPU 从 sys_call_table()返回时，具体系统调用的程序已经执行完了，如果没有别的事要干就恢复现场并返回用户空间。否则就先执行一些附加的系统操作，例如进程调度、本进程对信号(signal)的处理等，然后再返回。至于代码中的 syscall_trace_entry，则用于对系统调用的跟踪。

这段代码既不长，也不复杂，我们完全可以仿照着写一段汇编代码 w32_syscall，并让它与 int 0x2e 挂上勾：

```
#ifdef UNIFIED_KERNEL
        set_system_gate(0x2e, &w32_syscall);
#endif
```

汇编代码 w32_syscall 基本上可以照抄 system_call，只是一些具体的变量和参数需要加以修改。例如代码中的 nr_syscalls 应改成 nr_w32calls，即 Win2k 的系统调用总数，实际上是 248；而对 sys_call_table()的调用则应改成 w32_call_table()。当然，还要增加一个变量 nr_w32calls，还要有个函数 w32_call_table()。如有必要的话，我还将在以后的漫谈中进一步讨论有关细节。

显然，这样的实现并不难。

至于运行效率，采用 int 0x2e 的方式显然是最佳的，尽管去 sys_ioctl()转一圈的开销也不大，但毕竟要受一些影响。为说明这个问题，我们不妨看一下 sys_ioctl()的代码：

```
asmlinkage long sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
        struct file * filp;
        unsigned int flag;
        int on, error = -EBADF;

        filp = fget(fd);
        if (!filp)
```

```
            goto out;
        error = 0;
        lock_kernel();
        switch (cmd) {
            case FIOCLEX:
                set_close_on_exec(fd, 1);
                break;

            ……

            default:
                error = -ENOTTY;
                if (S_ISREG(filp->f_dentry->d_inode->i_mode))
                    error = file_ioctl(filp, cmd, arg);
                else if (filp->f_op && filp->f_op->ioctl)
                    error = filp->f_op->ioctl(filp->f_dentry->d_inode, filp, cmd, arg);
        }
        unlock_kernel();
        fput(filp);

out:
        return error;
}
```

CPU 经过前述的 sys_call_table()进入系统调用 ioctl()的处理程序 sys_ioctl()以后，首先比对“命令”码 cmd 是否 FIOCLEX、FIONCLEX、FIONBIO、FIOASYNC 等标准的命令码。然后才通过 default 下面的代码从目标文件的 struct file_operations 数据结构中获取具体设备文件的 ioctl 函数指针，去处理“自定义”的命令码、例如 FIOW32CALL。这一圈兜下来，虽不能说长，但每次系统调用都要无谓地牺牲掉这么一些，毕竟也不值得。

还有个因素也值得考虑，那就是：如果采用 int 0x2e 的方式，就有希望能使我们的系统调用界面达到跟 Windows 最大程度上的一致，从而有可能让微软的 ntdll.dll 等“系统 DLL”直接在兼容内核上运行。虽然那不是我们的目的(因为牵涉到版权问题)，但是在调试中偶而拿来试一下，跟我们自己的(基本上是 Wine 的)DLL 作一对比，也可能是很有帮助的。

综合上述这些因素，我倾向于采用 int 0x2e。