漫谈兼容内核之五:
Kernel-win32 的系统调用机制

毛德操

正如许多网友所言，要在 Linux 内核中实现 Windows 系统调用(或别的系统调用)，最简单的办法莫过于把这些系统调用“搭载”在 Linux 系统调用上。具体又有几种不同的方法：

1. 为 Linux 系统调用 ioctl()增加一些“命令码”，每个新的命令码都代表着一个 Windows 系统调用。
2. 为 Linux 增加一个新的系统调用、例如 win32_syscall()、作为总的入口和载体，然后定义一些类似于 ioctl()中所用那样的命令码。
3. 在 Linux 系统中定义一种虚拟的特殊文件，然后把 Windows 系统调用搭载在某个文件操作的系统调用上，例如 ioctl()、read()等等都可以用于这个目的。作为一种特例，在/proc 下面增加一个节点，就可以用于这个目的。又如 socket 也可以看作是这样的特殊文件。
4. 其它。例如也可以采取类似于“远程过程调用”、即 RPC 的形式，但是让“服务端”成为内核线程，或者直接在调用者的上下文中执行(这实际上是第 3 种方法的变种)。

其中又以第 1、2 两种方法更为简单易行。事实上 Kernel-win32 正是这样做的。

Kernel-win32 原先在这方面提供两种选项。一种是利用 Linux 的 ioctl()系统调用；另一种是为 Linux 增添一个 win32()系统调用，然后在这个新添系统调用的内部采用类似于 ioctl()那样的实现。但是在后来的版本中已经放弃了采用 ioctl()的选项(在相应的代码中加上了“#if 0”)，所以现在已经只采用上述的第二种方法，即为 Linux 增加一个系统调用作为载体。下面我们看它的代码。

首先，Kernel_win32 为 Linux 定义了一个新的系统调用号：

```
/* Linux Win32 emulation syscall */
#define   __NR_win32   249
```

新的调用号__NR_win32 定义为 249。这个调用号在当时无疑是空闲的，但是在 Linux 内核的 2.6.14 版中分配使用的系统调用号已经达到了 288，而调用号为 249 的系统调用是 io_cancel()。所以如果要在 2.6.14 版内核上使用 Kernel-win32 肯定得要修改这个调用号的定义。其实，这也反映出此种方法的缺点：只要是没有被正式纳入 Lunux 内核代码的系统调用号，都是靠不住的。此外，注释中说这是用来实现(Windows)系统调用“仿真(emulation)”的，但是我认为这只能说是“模拟(simulation)”，因为只是逻辑上相同、而形式上是不同的。

这个 Linux 系统调用只是个载体，而实际的 Windows 系统调用号(更确切地说是

Kernel_win32 系统调用号)，则另有定义：

```
/* Win32 system call numbers */
typedef enum {
    WINESERVER_INITIALISE_WIN32,
    WINESERVER_UNINITIALISE_WIN32,
    WINESERVER_CLOSE_HANDLE,
    WINESERVER_WAIT_FOR_MULTIPLE_OBJECTS,
    WINESERVER_CREATE_MUTEX,
    ……
    WINESERVER_MAP_VIEW_OF_FILE,
    WINESERVER_UNMAP_VIEW_OF_FILE,
    WINESERVER__LAST
} WineSyscallNum;
```

一共是 30 个调用号(最后的 WINESERVER__LAST 并非有效的调用号)。注意这些调用号跟真正的 Windows 系统调用号是完全不同的，例如 NtCloseHandle()的调用号是 24，而在这里是 2。而且，所定义的许多系统调用在 Windows 中并没有对应物，例如开头两个就是这样。至于 Windows 中有、而在这里没有定义的系统调用，那就更多了(Win2k 有 248 个系统调用)。所以这实际上不能说是 Windows 系统调用。代码中称为 WineSyscallNum，意思大概是把 Wine 的一些 RPC 函数转化成了系统调用。

此外，作为可安装模块的 Kernel-win32 还要在初始化时“登记”用来实现这个系统调用的函数。

```
static int __init wineserver_init_module(void)
{
#ifdef USE_WIN32SYSCALL
    int tmp;
#endif

    ……
#ifdef USE_WIN32SYSCALL
    /* register the syscall */
    tmp = register_win32_syscall(__NR_win32,win32syscall, &wineserver_ornament_ops);
    if (tmp<0) {
        remove_proc_entry("wineserver",NULL);
        return tmp;
    }
#endif

    return 0;
```

```
} /* end wineserver_init_module() */
```

要登记的函数是 win32syscall()，而指针&wineserver_ornament_ops 是要传递给这个函数的参数。我们往下看 register_win32_syscall()的代码。

[wineserver_init_module() > register_win32_syscall()]

```
int register_win32_syscall(int syscall, win32syscall_func func,
                           const struct task_ornament_operations *ornament_type)
{
    ……
    if (!w32handler) {
        if (cmpxchg(&sys_call_table[syscall],
                    (long)sys_ni_syscall,
                    (long)sys_win32
                    )==(long)sys_ni_syscall
                ) {
            /* we installed it successfully */
            w32syscall = syscall;
            w32handler = func;
            w32ornament_type = ornament_type;
        }
        ret = 0;
    }
    ……
    return ret;
} /* end register_win32_syscall() */
```

显然，实际填写到系统调用(跳转)表 sys_call_table[ ]中的函数指针是 sys_win32()。

应用软件在需要进行 Windows 系统调用(更确切地说是 Kernel_win32 系统调用)时通过一个库函数 win32()实施调用。例如要调用 CreateFile()时就这样调用：

```
int i = win32(WINESERVER_CREATE_FILE, &args);
```

参数 WINESERVER_CREATE_FILE 就是系统调用号，或称“命令码”。而真正用于 CreateFile()的参数，则都组装在一个数据结构中，&args 就是这个数据结构的地址。

库函数 win32()的代码很简单：

```
static __inline__ int win32(int cmd, void *args)
{
#ifdef USE_WIN32SYSCALL
    return syscall(__NR_win32, cmd, args);
#else
```

```
#error must use Win32 Syscall
#endif
}
```

这里的 syscall()是 C 库中的一个函数，其代码在 glibc 的一个源文件 syscall.s 中，有兴趣的读者可以自行阅读。其作用则不言自明：__NR_win32 是 Linux 系统调用号；cmd 是命令码、即 Kernel_win32 系统调用号；args 是指向参数结构的指针。后面两项都是对于 Linux 系统调用 win32()的参数。

进入内核以后，CPU 根据系统调用号和内核中的系统调用(跳转)表进入内核函数 sys_win32()：

```
asmlinkage int sys_win32(unsigned int cmd, void *args)
{
    struct task_ornament *orn;
    win32syscall_func fnx;
    int error;

    ……
    fnx = w32handler;
    ……
    /* find the ornament on the current task (does ornget if successful) */
    orn = task_ornament_find(current, w32ornament_type);
    ……
    /* invoke the handler */
    error = fnx(orn,cmd,args);
    ……
    return error;

} /* end sys_win32() */
```

从代码中可以看出，这个函数只是中转，真正的目的是要通过前面登记的函数指针 w32handler 调用 win32syscall()。那么为什么不直接把 win32syscall()放在系统调用表中，从而直接进入 win32syscall()呢？比较一下两个函数的调用参数就可以知道，后者要求以指向当前线程的 task_ornament 结构指针作为参数，而内核根据系统调用表中的函数指针进行调用时是不能带额外参数的。之所以需要这个指针，显然是因为有关 Windows 线程的补充信息都在这个数据结构中，或者从这个数据结构开始才能找到。

为了要得到当前线程的这个 task_ornament 结构指针，这里通过 task_ornament_find()在当前 task_struct 的 ornament 队列中寻找。这里的指针 w32ornament_type 指向数据结构 wineserver_ornament_ops，这是前面登记系统调用函数指针时设置好的。

[sys_win32() > task_ornament_find()]

```
struct task_ornament *task_ornament_find(struct task_struct *tsk,
                                struct task_ornament_operations *type)
{
    struct task_ornament *orn;
    struct list_head *ptr;

    read_lock(&tsk->alloc_lock);
    for (ptr=tsk->ornaments.next; ptr!=&tsk->ornaments; ptr=ptr->next) {
        orn = list_entry(ptr,struct task_ornament,to_list);
        if (orn->to_ops==type)
            goto found;
    }

    read_unlock(&tsk->alloc_lock);
    return NULL;

 found:
    ornget(orn);
    read_unlock(&tsk->alloc_lock);
    return orn;
} /* end task_ornament_find() */
```

这段程序扫描给定 task_struct 结构的 ornament 队列，从中寻找类型为 type、实际上是指向 wineserver_ornament_ops 的 task_ornament 数据结构。其实，正如在“对象管理”那篇漫谈所述，这个队列中一般只有一个数据结构，而且其类型也正是 wineserver_ornament_ops。所以是否真的需要如此大动干戈是值得推敲的。

找到了这个补充性的数据结构，就可以调用 win32syscall()了。

[sys_win32() > win32syscall()]

```
#ifdef USE_WIN32SYSCALL
int win32syscall(struct task_ornament *orn, unsigned int syscall, void *uargs)
{
    struct WineThread *thread;
    void *args;
    int err;

    ……
    if (copy_from_user(args, uargs, ioctl_cmds[syscall].ic_argsize)) {
        err = -EFAULT;
        goto cleanup;
    }
```

```
    /* invoke the syscall handler */
    if (orn)
        thread = orn_entry(orn,struct WineThread,wt_ornament);
    else
        thread = NULL;
    ……
    err = ioctl_cmds[syscall].ic_handler(thread,args,uargs);
    ……
    return err;
} /* end win32syscall() */
#endif
```

先从用户空间把实际的调用参数(组装在一个数据结构中)复制到系统空间的一个缓冲区。下面的 orn_entry 是个宏操作，目的是把 task_ornament 结构指针换算成 WineThread 结构指针。因为前者是后者内部的一个成分，所以可以进行换算。

关键的操作就是根据 Kernel-win32 系统调用号 syscall 从系统调用表 ioctl_cmds[ ]中取得目标函数指针并加以调用。这个数组之所以叫 ioctl_cmds，是因为原先 Kernel-win32 是通过 ioctl()进行调用的。

```
static const struct _ioctl_cmd ioctl_cmds[] =
{
    _WIN32(InitialiseWin32),
    _WIN32(UninitialiseWin32),
    _WIN32(CloseHandle),
    _……
    _WIN32(CreateFileA),
    _WIN32(ReadFile),
    _WIN32(WriteFile),
    ……
    _WIN32(CreateFileMappingA),
    _WIN32(MapViewOfFile),
    _WIN32(UnmapViewOfFile),
    { 0, NULL }
};
```

这里宏操作_WIN32 的定义为：

```
#define _WIN32(X) { sizeof(struct Wioc##X), X }
```

以数组中的元素_WIN32(CreateFileA)为例，经过编译以后就成为：

```
{sizeof(struct WiocCreateFileA), CreateFileA}
```

所以前面引用的 ioctl_cmds[syscall].ic_argsize 和 ioctl_cmds[syscall].ic_handler 分别为参数结构的大小和函数指针。再往下的事就不用说了。

把 Windows 系统调用(或 Kernel-win32 系统调用)搭载在 Linux 系统调用上的做法简单易行，但是也有缺点。

首先是降低了效率。从上述的过程一步一步下来，可以看出系统的开销还是不小的。这里面有的是因为 Kernel-win32 具体的设计所引起，实际上还可优化；有的却是这种方法所固有的，这跟 CPU 的“间接寻址”与“直接寻址”的区别有些相似。这一点开销，对于本身较大、较费时的系统调用而言固然可以忽略不计；但是对于本身较小、特别是需要频繁调用的系统调用而言却是不可忽略的了。

更重要的是，由于是搭载在 Linux 系统调用上，返回用户空间时也必定跟 Linux 系统调用走同一条路线。然而在这方面两个系统是有区别的。在返回的过程中，Linux 要检查是否有 Signal，如果有就要在用户空间加以执行(类似于对用户空间程序的中断)，而 Windows 则要检查和处理 APC。这二者原理相似，但是具体的实现还是有差别的(至少有待研究)。显然，最好是能够各走各的路，否则对于要达到高度兼容的目的是不利的。

另一方面，这种方法对于用户空间 DLL 的实现、特别是 ntdll.dll 的实现有了特殊要求。所以这种方法只能说是“模拟”而不是“仿真”。为了达到高度兼容的目标，在测试时最好能够把我们的全套 DLL、包括 ntdll、安装到 Windows 上去，再运行 Windows 应用软件，看其表现和效果是否与使用“原装”DLL 时相同。反过来，也最好能把 Windows 的原装 DLL 和应用软件安装到 Linux 上(如果这样做不构成侵犯版权的话)，以检验 Linux 兼容内核对 Windows 系统调用的支持是否正确与完整。这就要求二者采用相同的机制与手段，例如都采用 int 0x2e，发生 0x2e 自陷时有相同的堆栈内容，采用相同的系统调用号，等等。而且，有些特殊的应用软件甚至可能绕过 Win32 API，而直接通过 int 0x2e 进行系统调用，对于这样的应用显然就无法在 DLL 中拦截其系统调用并加以转换。更何况 Windows 的系统调用还不完全限于 int 0x2e 这一种手段，还有 0x2b、0x2c、0x2d 也是特殊的系统调用手段，而且那些调用更有可能绕过 Win32 API。当然，那些调用的实现是将来的事，甚至可能不会去实现，但作为设计方案也应该加以考虑、或留下余地。

所以，对于兼容内核，我们应该考虑采用 int 0x2e 作为系统调用的手段，并实现一套跟 Windows 尽可能一致的机制。具体地，就是要把 ReactOS 的系统调用机制与 Linux 的系统调用机制揉合在一起，使其在系统空间与用户空间的分界线上呈现出跟 Windows 尽量一致、甚至完全一致的特性。其实这也并不像有些人想像的那么难，实际上我们已经调通了这样的一个雏型，春节之后整理一下就可以把源码公开出来。