漫谈兼容内核之四:
Kernel-win32 的进程管理

毛德操

由于进程管理与对象管理不可分割，我在谈论 Kernel-win32 的对象管理时也谈到了一些有关进程管理的内容，例如对 task_struct 数据结构的扩充，以及对 Linux 内核有关代码所打的补丁。但是这还不够，还需要进一步讨论。

对于任何现代操作系统而言，进程(线程)管理都是一个十分重要的环节。Windows 与 Linux 在这方面恰恰有着相当大的差异，有的是概念上的，有的是实现细节上的：

1. 在 Linux 内核中，线程和进程都是由 task_struct 数据结构作为代表的。一个 task_struct 数据结构所代表的实体，只要是与其父进程共享同一用户空间的就是线程；否则，如果已经“另立门户”、拥有自己的用户空间，那就是进程。或者，如果换一种观点，那就是进程及其“第一个线程”是合一的，是同一回事。在 Linux 内核中，task_struct 数据结构就是进程调度的单位。而在 Windows 中，则进程与线程有不同的数据结构，只有代表着线程的数据结构才是调度的单位，而代表着进程的数据结构是被架空的，没有受调度运行的权利。因此，所谓创建一个 Windows 进程，总是意味着创建一个进程及其“第一个线程”，以两个不同数据结构的组合作为代表。进程与线程是一对多的关系，这在 Linux 中和 Windows 中都一样，但是在 Linux 中这体现为一组 task_struct 数据结构的“家属树”，逻辑上是层次结构，实现上则是网状结构(属于同一进程的同层线程之间也有链接)。而在 Windows 中则体现为一个进程结构和多个线程结构，最自然的当然是让所有的线程排成一个队列，并且都有指针指向其所属进程的数据结构。
2. 抛开在结构形态上的不同，Linux 的 task_struct 结构也并不是简单地把 Windows 的进程结构和线程结构加在一起。有些成分在 Windows 的数据结构中有，而在 task_struct 结构中没有，有些则反过来。
3. 两个系统用于创建进程/线程的系统调用在语义上有很大的区别。在 Linux 中，这首先是父进程的“细胞分裂”，即分裂成两个线程，然后如果子进程另立门户就又变成两个进程。就是说，创建线程是创建进程的必经之途。而在 Windows 中，则创建进程和创建线程是两码事，创建进程的系统调用并不蕴含着同时创建其第一个线程。
4. 进程在两个系统中的地位与权利有很大区别。在 Linux 中每个进程都有相当的独立性，有自己的“隐私”和“私有财产”，而在 Windows 中一个进程甚至可以替另一个进程创建一个线程。
5. 两个系统在资源和权限的遗传/继承方面有很重要的区别。
6. 两个系统在调度策略和优先级的设置方面也有区别。在 Linux 中，由于 task_struct 是调度单位，每个线程都可以有自己的调度策略和优先级。而在 Windows 中，则首先是进程一级的优先级，然后是线程在同一进程中的相对优先级。前者是一种水平的结构，后者是一种层次的结构。
7. 两个系统在进程间通信方面也有区别，有的是名称和实现细节的不同，有的确有实质的区别，例如 Windows 的跨进程复制 Handle，就在 Linux 中没有对应的机制。

显然，要在 Linux 内核上运行 Windows 软件，就必须让 Windows 线程借用 Linux 的 task_struct 数据结构，否则就不能被调度运行(要不然就得大改 Linux 内核中的 schedule()了，这当然是应该避免的)。这样，内核中的 Windows 线程就成为 Linux 进程/线程的一个子集，

或者说特殊的 Linux 进程/线程。为此，为了在内核中弥补上述的种种不同，首先当然要在 task_struct 结构中增加一个指针(Kernel-win32 使用 task_ornament 队列)，使其指向补充性的附加数据结构。同时，由于要在 Linux 内核上运行 Windows 线程，就有个如何确定一个 Linux 进程是否 Windows 线程的问题。当然，只要 task_struct 结构中的附加数据结构指针非 0、或队列非空，就说明是个 Windows 线程。可是，什么时候为其分配附加数据结构并设置这个指针或队列呢？显然这里需要有个依据、有个手段。我们先看 Kernel-win32 所采用的办法。

Kernel-win32 要求所有 Windows 线程在初始化时都执行一个系统调用 Win32Init()，让内核知道当前线程是个 Windows 线程。这个系统调用是 Kernel-win32 加出来的，Windows 并没有这么一个系统调用。我们先看这个 Kernel-win32 系统调用的实现：

```
int InitialiseWin32(struct WineThread *thread, struct WiocInitialiseWin32 *args)
{
    struct WineThreadConsData wtcd;
    ……

    /* allocate a Wine process object */
    probj = AllocObject(&process_objclass,NULL,NULL);
    ……
    /* allocate a Wine thread object */
    wtcd.wtcd_task = current;
    wtcd.wtcd_process = probj;
    throbj = AllocObject(&thread_objclass,NULL,&wtcd);
    ……
    return 0;
} /* end InitialiseWin32() */
```

不妨假定这是个新创建的 Windows 进程，从而当前线程是这个进程中的第一个线程。先为之分配和创建一个进程对象(及其配套的 WineProcess 数据结构)。代码中的数据结构 wtcd 只是个临时用来传递信息的载体，注意其成分 wtcd_task 设置成 current，这就是指向当前 task_struct 数据结构的指针。显然，对于新创建的 Windows 进程，这个结构中的 task_ornament 队列是空的，所以此刻的当前进程(线程)还是个 Linux 进程(线程)。接着再分配和创建一个线程对象(及其配套的 WineThread 数据结构)。我们知道，在创建对象的过程中要调用该类对象的构建函数，对于线程对象就是 ThreadConstructor()，我们再重温一下：

```
static int ThreadConstructor(Object *obj, void *data)
{
    struct WineThreadConsData *wtcd = data;
    ……
    thread->wt_task = wtcd->wtcd_task;
    ……
    add_task_ornament(thread->wt_task,&thread->wt_ornament);
    ……
}
```

```
void add_task_ornament(struct task_struct *tsk, struct task_ornament *orn)
{
    ornget(orn);
    write_lock(&tsk->alloc_lock);
    list_add_tail(&orn->to_list,&tsk->ornaments);
    write_unlock(&tsk->alloc_lock);
} /* end add_task_ornament() */
```

显然，正是 ThreadConstructor()把新进程的第一个线程挂入了当前 task_struct 结构中的 task_ornament 队列，使其变成非空，从而使 Linux 进程(线程)变成了 Windows 线程。至于前面创建的进程对象，那是通过另一个队列跟其所有的线程串在一起的，与 task_struct 结构并没有直接的连系，这以前已经讲过了。而且，由于每个线程都有自己的 task_struct 数据结构，实际上每个 Windows 线程都得在初始化时调用 Win32Init()。Kernel-win32 似乎并没有考虑“龙生龙，凤生风”式的遗传。

以前讲过，其实 task_struct 数据结构的 task_ornament 队列中只有一个线程，只不过属于同一个 Windows 进程的所有线程都通过另一个队列串在一起。第一个线程与后续线程的区别只是：创建第一个线程时要创建新的进程对象(及其线程队列)，同一进程中后来创建的线程则不创建进程对象，而只是找到其所属的已有进程对象。

既然新进程(线程)在创建之初时是 Linux 进程，可想而知新进程(线程)的创建可以通过 Linux 系统调用实现。事实正是如此，Kernel-win32 并没有实现创建进程或线程的 Windows 系统调用，而仍沿用 fork()、execve()等等作为创建进程或线程的手段。Kernel-win32 代码中的一些测试程序清楚地表明了这一点，下面是测试程序 fivemutex.c 中的一些代码。

```
int main()
{
    int loop;

    for (loop=0; loop<5; loop++) {
        switch (fork()) {
            case -1:
            ERR(1,"fork");
            case 0:
            return child(loop);
            default:
            break;
        }
    }
    while (wait(&loop)>0) {}
    return 0;
}

int child(int pid)
{
    HANDLE left, right, first, second;
```

```
    const char *lname, *rname;
    int count = 0;
    int wt;

    Win32Init();
    ……
}
```

这里，测试进程的第一个线程通过 Linux 系统调用 fork()创建出 5 个线程，每个线程都执行 child()。而所创建的每个线程，则都调用 Win32Init()，使其自身变成 Windows 线程。有趣的是这里的第一个线程 main()并没有调用 Win32Init()，这是因为它干的尽是 Linux 的事，所以并不在乎。在这种情况下，fork()出来的第一个线程就成为“Windows 进程”的第一个线程，即负有创建进程对象的责任。

现在可以讨论了。

首先是把对于 Win32Init()的调用放在哪里。当然不能放在 Windows 应用软件中，因为那都是“木已成舟”的二进制可执行映像。比较可行的是放在某个 DLL 中，最大的可能是放在 ntdll.dll 中。

然后是什么时候调用 Win32Init()。读者可能会想，当应用软件向下调用创建 Windows 进程或线程的时候，在 ntdll.dll 中可以先调用 fork()，再调用 Win32Init()。然而这是错的，因为调用 fork()的是父进程(线程)，而需要调用 Win32Init()的是新创建出来的线程，这是两码事。显然，这里需要某种机制，虽然并非不能实现，却也不是很简单。事实上我们在 kernel-win32 的代码中尚未见到相应的实现。

更重要的是，用 fork()加 Win32Init()是否能忠实地实现 Windows 中那些创建进程/线程系统调用的语义？为此，我们看一下两个 Windows 系统调用的函数定义。

先看进程的创建。

```
CreateProcessA(
    IN LPCSTR                       lpApplicationName,
    IN LPSTR                        lpCommandLine,
    IN LPSECURITY_ATTRIBUTES        lpProcessAttributes,
    IN LPSECURITY_ATTRIBUTES        lpThreadAttributes,
    IN BOOL                         bInheritHandles,
    IN DWORD                        dwCreationFlags,
    IN LPVOID                       lpEnvironment,
    IN LPCSTR                       lpCurrentDirectory,
    IN LPSTARTUPINFOA               lpStartupInfo,
    OUT LPPROCESS_INFORMATION       lpProcessInformation
);
```

这是 Win32 API 界面上的函数定义，所以是个 DLL 函数，还不是系统调用。Windows 系统调用的界面是不公开的。不过好在我们已经有了 ReactOS，从 ReactOS 的代码中可以看到这个系统调用的函数定义是：

```
NtCreateProcess(
        OUT PHANDLE             ProcessHandle,
        IN ACCESS_MASK          DesiredAccess,
        IN POBJECT_ATTRIBUTES   ObjectAttributes   OPTIONAL,
        IN HANDLE               ParentProcess,
        IN BOOLEAN              InheritObjectTable,
        IN HANDLE               SectionHandle   OPTIONAL,
        IN HANDLE               DebugPort   OPTIONAL,
        IN HANDLE               ExceptionPort   OPTIONAL
)
```

详细说明这些参数的作用是件颇费篇幅的事，读者可以自己阅读“Windows NT/2000 Native API Reference”第六章中关于 ZwCreateProcess()的说明(ZwCreateProcess()和 NtCreateProcess()是同一个函数的两个名字，有的文献说在用户空间叫 ZwCreateProcess()、在内核中叫 NtCreateProcess())。我们这里只是长话短说，提一下往往会使 Linux 程序员感到惊讶的东西。

首先当然是 InheritObjectTable。这是个布尔量，表示是否要从父进程继承已经打开的对象，但是即使要继承也不是全盘照收，还要看具体对象在打开时是否允许遗传。

另一个参数 ParentProcess 是个已打开进程对象的 Handle，如果是有效 Handle 的话就表示新创建的对象应该“过继”给这个进程、作为它的子进程，而不是作为其创建者即“生父”的子进程。或者，换句话说就是包办、替别的进程创建一个子进程。

而 DesiredAccess，则有下列选项：

```
#define PROCESS_TERMINATE               1
#define PROCESS_CREATE_THREAD           2
#define PROCESS_SET_SESSIONID           4
#define PROCESS_VM_OPERATION            8
#define PROCESS_VM_READ                 16
#define PROCESS_VM_WRITE                32
#define PROCESS_DUP_HANDLE              64
#define PROCESS_CREATE_PROCESS          128
#define PROCESS_SET_QUOTA               256
#define PROCESS_SET_INFORMATION         512
#define PROCESS_QUERY_INFORMATION       1024
#define PROCESS_ALL_ACCESS          \
            (STANDARD_RIGHTS_REQUIRED|SYNCHRONIZE|0xFFF)
```

别的就不说了，光是上面这些，读者就可以看出 NtCreateProcess()与 fork()、execve()等 Linux 系统调用的差距有多大了。显然，以目前的 kernel-win32，要实现与 Windows 的高度兼容是不可能的。

还要注意，这个系统调用只创建进程、而不包括其第一个线程。这跟 Win32 API 函数 CreateProcess()是不一样的，后者实际上先调用 NtCreateProcess()，再调用 NtCreateThread()。

那么是否可以通过在用户空间、即在 DLL 中把 CreateProcess()化解成一个 kernel-win32 和 Linux 系统调用的序列来解决问题呢？有的可以，有的不行。例如上述把所创建的进程过

继给另一个进程就不行，因为过继给另一个进程也意味着从“继父”那里、而不是从“生父”那里、继承已打开对象(如果需要的话)。

再看线程的创建。

```
CreateThread(
    IN LPSECURITY_ATTRIBUTES    lpThreadAttributes,
    IN DWORD                    dwStackSize,
    IN LPTHREAD_START_ROUTINE   lpStartAddress,
    IN LPVOID                   lpParameter,
    IN DWORD                    dwCreationFlags,
    OUT LPDWORD                 lpThreadId
);
```

同样，这只是 API 函数，而相应的系统调用是 NtCreateThread()，下列函数定义取自 ReactOS 的代码：

```
NtCreateThread(
    OUT PHANDLE T               hreadHandle,
    IN ACCESS_MASK              DesiredAccess,
    IN POBJECT_ATTRIBUTES       ObjectAttributes    OPTIONAL,
    IN HANDLE                   ProcessHandle,
    OUT PCLIENT_ID              ClientId,
    IN PCONTEXT                 ThreadContext,
    IN PINITIAL_TEB             InitialTeb,
    IN BOOLEAN                  CreateSuspended
)
```

同样，详细的说明请看“Windows NT/2000 Native API Reference”第五章，这里只是简单地提一下。首先是 ProcessHandle，这是个已打开进程对象的 Handle。这就是说，NtCreateThread()的调用者可以为别的进程创建线程，而不仅仅是为调用者本身所属的进程。再看 CreateSuspended，这是个布尔量，表示新创建的线程是否一出生就先被挂起、等到有别的线程对其执行 NtResumeThread()后才投入运行，抑或一生下来就立即投入运行。还有 ThreadContext，这是个指针，可以指向一个数据结构，里面规定了新线程降生之初各个寄存器的值(真令人难以理解这到底是为什么)。还有参数 InitialTeb，在“Native API”一书中说是 UserStack，用来指定新线程的用户空间堆栈的位置(这倒有道理)。至于 DesiredAccess，则又有下列许多选项：

```
#define THREAD_TERMINATE                (0x0001L)
#define THREAD_SUSPEND_RESUME           (0x0002L)
#define THREAD_ALERT                    (0x0004L)
#define THREAD_GET_CONTEXT              (0x0008L)
#define THREAD_SET_CONTEXT              (0x0010L)
#define THREAD_SET_INFORMATION          (0x0020L)
#define THREAD_QUERY_INFORMATION        (0x0040L)
```

```
#define THREAD_SET_THREAD_TOKEN         (0x0080L)
#define THREAD_IMPERSONATE              (0x0100L)
#define THREAD_DIRECT_IMPERSONATION     (0x0200L)
#define THREAD_ALL_ACCESS               (0x1f03ffL)
```

读者不难看出，这与 fork()的差距可真够大的了。而且，有些差异是不能在用户空间弥补的，例如为别的进程创建线程，还有让新创建的线程进入“挂起”状态等等就是这样。

这还只是 NtCreateProcess()和 NtCreateThread()两个系统调用。与进程/线程管理有关的 Windows 系统调用至少还有下面这些：

NtAlertThread()
NtCreateProcess()
NtCreateThread()
NtDuplecateObject()
NtGetContextThread()
NtImpersonateThread()
NtOpenProcess()
NtOpenThread()
NtQueueApcThread()
NtResumeThread()
NtSetContextThread()
NtSetInformationProcess()
NtSetInformationThread()
NtSetThreadExecutionState()
NtSuspendThread()
NtTerminateProcess()
NtTerminateThread()
NtYieldExecution()

由此可见，要跟 Windows 高度兼容，可真是路慢慢其修远。不过也不要被吓倒，还是那句老话：战略上藐视困难，战术上重视困难。再说也确实并非所有特性都必须加以实现，因为绝大多数应用软件都不会去用那些刁钻古怪的功能。

但是有一点是明白无误的，那就是迄今为止的 Kernel-win32 还只是朝正确的方向走了一小步。而且，其设计方案也有不当之处，想要用 fork()加 Win32Init()实现进程/线程的创建就是。这也是我认为对于兼容内核而言 ReactOS 远比 Kernel-win32 重要的原因。