漫谈兼容内核之二：
关于 kernel-win32 的对象管理

毛德操

近来屡有网友提到一个旨在将 Wineserver 移入内核的开源项目 kernel-win32；有问及其本身，希望能对其代码作一些分析、讲解的，也有问及兼容内核与此项目之间关系的。所以从这篇漫谈开始就来谈谈 kernel-win32。

首先，兼容内核项目应当从所有(能找到的)相关开源项目吸取营养，有时候甚至就采取“拿来主义”，反正都是开源项目，只要遵守有关的规定就行。从这个意义上说，我们对于 kernel-win32 肯定要借鉴，也可能要“拿来”一些。但是这种借鉴和拿来的取舍必须以客观的分析为基础，必须与我们的终极目标相一致。相信读者在看完从本文开始的几篇漫谈以后就会明白我为什么把 Wine、ReactOS、和 NDISwrapper 列为兼容内核的三个主要源泉，而没有把 kernel-win32 也列为主要源泉之一。

从总体上说，kernel-win32 把原来由 Wine 服务进程提供的某些功能和机制移入了 Linux 内核，具体(就目前所见版本而言)有这么一些：

1. 文件操作。
2. Semaphore 操作。
3. Mutex 操作。
4. Event 操作。
5. 作为同步手段的 WaitForMultipleObjects()系统调用。

所有这些机制和功能的实现都有个共同的基础，那就是各种内核“对象(Object)”及其 Handle 的实现。由于已打开的对象是属于进程的资源，又由同一进程中的所有线程所共享，所以又跟进程、线程的实现和管理有关。

此外，kernel-win32 也提供了比 Wine 更高效的 RPC 机制，用以提高应用进程与 Wine 服务进程通信的效率。

但是 kernel-win32 的实现并不完整，甚至并不构成局部的完整性，而且已经实现的部分恰恰是相对而言难度并不高的部分，所采取的方案也还值得推敲。

特别地，kernel-win32 的目标只在于提高 Wine 的效率，所以并不涉及设备驱动。与兼容内核的终极目标相比，二者只是在一小段路程上有“同路”的关系。以我们的眼光看，kernel-win32 无疑是朝正确的方向上在走，但是走的毕竟太近。

Kernel-win32 的代码大体上分成三个部分。第一部分是对于 Linux 内核代码的补丁，在它的 kernel 目录下。第二部分是它本身的代码，是要作为 module 动态安装到内核中去的，具体的代码文件都在它的根目录下。第三部分是一些测试/演示程序，这是作为应用软件在 Wine 上运行，或者部分地绕过 Wine、与 Wine 并行的，特别是其中还包括一个用于启动系统调用的库程序 win32.c，这些程序都在它的 test 目录下。从逻辑上说，库程序 win32.c 属于 kernel-win32，而测试/演示程序则不是；就好像 Linux 的内核与 libc 都属于 Linux，而用来测试/演示其功能的程序却不算一样。此外，为了帮助调试，代码中的 strace 目录下还包括了对 Linux 的一个调试工具 strace 的补丁与扩充。这个工具可以用来跟踪应用软件所作的 Linux 系统调用，实时地显示所跟踪的应用软件进行具体系统调用时的参数和内核的返回值。这对于调试当然大有帮助，但在逻辑上也并不是 kernel-win32 的一部分。

下面我从 kernel-win32 的代码入手，对它的方方面面作一简要的介绍和分析，本篇先说 kernel-win32 的对象管理，即 Object 和 Handle 的实现，实际上也必然会牵涉到进程和线程。

Object 和 Handle 的实现

我曾经讲到，Linux 把设备看成文件，所以“文件”是个广义的概念；而 Windows 又进一步把文件看成“对象(Object)”，“对象”是个更广义的概念。而标志着已打开对象的“句柄(Handle)”，则虽然在物理意义上与“打开文件号”相似(实质上都是下标)，却有着许多不同的特性，因而不能把二者混为一谈。为了把 Linux(内核)的文件系统机制“嫁接”到 Windows 的系统调用界面，必须为每个运行着 Windows 应用的进程(下称“Windows 进程”或“Wine 进程”)准备下一个地方，用来维持一个类似于“打开文件表”的“打开对象表”。另一方面，Windows 的“进程”和“线程”都与它们的 Linux 对应物有所不同，从而有着不同的数据结构，因此需要为每个 Windows 进程提供附加的数据结构，作为对 Linux“进程控制块”、即 task_struct 数据结构的补充，并且在二者之间建立起某种连接，例如在 task_struct 结构中增添一个指针等等。

Kernel-win32 本质上正是这样做的，我们不妨看一下它对 task_struct 结构所打的补丁(为便于阅读，已经作了一些整理)：

```
struct task_struct {
      . . . . . .
-     spinlock_t          alloc_lock;
+     rwlock_t            alloc_lock;
+     struct list_head    ornaments;
 };
```

本来 alloc_lock 是这个数据结构中的最后一个成分，现在把它的类型从 spinlock_t 改成了 rwlock_t，这是因为原来的加锁只是针对多处理器结构、防止不同处理器之间互相冲突的，现在则范围有所扩大。而所增添的成分，则是一个双链队列头，称为“ornaments”。Ornament 这个词原本是“装饰、饰物”的意思，在这里则引伸成“附件、补充”的意思。

那么准备要链入这个队列的数据结构是什么呢？这是 task_ornament 数据结构：

```
struct task_ornament {
      atomic_t                                to_count;
      struct list_head                        to_list;
      const struct task_ornament_operations   *to_ops;
};
```

其中的 to_count 显然是个使用计数，计数为 0 时表示该数据结构不再有“用户”，从而可以撤销了。队列头 to_list 显然就是用来将此数据结构挂入 ornaments 队列的，所以这儿实质性的成分就是指针 to_ops，它指向一个 task_ornament_operations 数据结构，里面主要是一些函数指针。目前 kernel-win32 只定义了一种 task_ornament_operations 数据结构，即 wineserver_ornament_ops，后面还要讲到。

而 task_ornament 数据结构，则又可以是另一个数据结构 WineThread 中的一个成分。所以挂入 ornaments 队列的(除特殊情况外)实际上都是 WineThread 数据结构，此时 task_ornament 数据结构起着“连接件”的作用。

```
struct WineThread {
#ifdef WINE_THREAD_MAGIC
    unsigned                wt_magic;          /* magic number */
#endif
    struct task_ornament    wt_ornament;       /* Linux task attachment */
    struct task_struct      *wt_task;          /* Linux task */
    Object                  *wt_obj;           /* thread object */
    struct WineProcess      *wt_process;       /* Wine process record */
    struct list_head        wt_list;           /* process's thread list */
    enum WineThreadState    wt_state;          /* thread state */
    unsigned                wt_exit_status;    /* thread exit status */
    pid_t                   wt_tid;            /* thread ID */
};
```

当然，每个 WineThread 数据结构代表着一个 Wine 线程、即 Windows 线程。

在 Linux 内核中，task_struct 数据结构代表着一个进程或线程，是内核调度运行的对象。Wine 线程既要受调度运行，就必须落实到一个 task_struct 数据结构、即 Linux 线程或进程上。反过来，作为一个受调度运行单位的 task_struct 数据结构(如果代表着 Wine 线程的话)，也不能代表多个 Wine 线程，否则这几个 Wine 线程就合并成一个调度单位了。所以这二者之间应该是一一对应的关系。既然是一一对应的关系，就应该使用指针、而不是队列来建立互相的连系。而既然采用了队列，队列中又只能有一个 Wine 线程，那么队列中别的成员(如果有的话)，就必定是别的什么东西了。从逻辑上说，同一个队列中的诸多成员之间有着平等的关系，可是有什么东西可以和线程处于平等的地位呢？所以这是值得推敲的，后面我还要谈这个问题。

WineThread 结构中的几个成分需要加以说明：

指针 wt_obj 指向一个 Object 指针。在 Windows 中线程、进程都是“对象”，都要有一个 Object 数据结构作为代表。对于 Object 数据结构等一下再作说明。

指针 wt_task 指向当前进程(线程)的 task_struct 数据结构，这样就可以在一个 Wine 线程与其所落实的 Linux 线程之间建立起双向的连系(另一个方向就是顺着 Linux 线程的 ornaments 队列)。另一个指针 wt_process 指向一个 WineProcess 数据结构。显然，WineProcess 数据结构代表着 Widows 进程。

在 Linux 内核中，进程并没有独立于线程的数据结构，都由 task_struct 作为代表。一个进程初创(通过 execve()等调用与其父进程决裂)时就成为进程，同时也可以说是该进程中的第一个线程。以后，该进程通过 fork()等调用创建子进程。子进程在创建之初都是与父进程共享空间的，因而都是线程，后来才通过 execve()等调用另立门户，有了自己的空间而成为进程。然而 Windows 不是这样。在 Windows 中。一个进程与该进程的第一个线程(以及别的线程)是两个不同的概念，有不同的数据结构。大体上说，进程代表着资源，特别是代表着一片用户空间；而线程则代表着上下文。打个比方，进程就像是舞台和剧本，而线程是演员及其表演的过程。在 Windows 内核中，这两方面的信息分置于不同的数据结构中，而在 Linux 内核中则全都存放在 task_struct 结构中。但是 Windows 进程和线程的有些信息是 task_struct 结构中所没有或者不同的，这也正是需要在 task_struct 结构之外加以“装饰”、补充的原因。

WineProcess 数据结构的定义如下：

```
struct WineProcess {
    int                     wp_magic;       /* magic number */
    struct nls_table        *wp_nls;        /* unicode-ascii translation */
    pid_t                   wp_pid;         /* Linux task ID */
    enum WineProcessState   wp_state;       /* process state */
    struct list_head        wp_threads;     /* thread list */
    rwlock_t                wp_lock;
    struct Object           *wp_obj;        /* process object */
    struct Object           *wp_handles[0]; /* handle map */
};
```

其中的队列头 wp_threads 与上面 WineThread 结构中的 wt_list 相对应，用来构成一个 Windows 进程与其所含的所有 Windows 线程之间的双链队列。如前所述，Windows 进程在概念上并不落实到某个具体的 Linux 进程或线程，所以这个数据结构中并没有指向 task_struct 结构的指针。不过实质上当然还是有连系的，因为在 Linux 中一个“进程”和它的“第一个线程”是一回事。这里的 wp_pid 既然说是 Linux 的“task ID”(就是 Linux 的 pid)，实际上还是一样，还不如改成 task_struct 结构指针更好。

顺便提一下，Windows 进程与 Linux 进程在优先级设置方面有很大的不同，而 Linux 内核是根据 task_struct 数据结构中记载的优先级进行调度的，这里面有个如何换算的问题。WineProcess 数据结构中没有关于进程优先级的记载，显然 kernel-win32 的作者还没有考虑这个问题。

在 Windows 中进程也是“对象”，所以这里也有个 Object 结构指针 wp_obj。这与 WineThread 结构中的指针 wt_obj 是一样的，只不过两个对象的类型不同，一个代表着进程，一个代表着线程。

指针数组 wp_handles[ ]才是这个数据结构的实质所在，这就是一个 Windows 进程的“打开对象表”。数组中的每个有效(非 0)指针都指向一个 Object 结构；对象的类型不同，相应 Object 结构所代表的目标也就不同。例如有的代表着文件，有的代表着“事件”，有的代表着进程，等等。而具体指针在数组中的下标(严格地说是经过换算的下标，见后所述)，则就是打开该对象后的 Handle。在代码中，这个数组的大小定义为 0，这是因为其大小取决于为 WineProcess 数据结构分配的存储空间的大小。目前，kernel-win32 总是为 WineProcess 数据结构分配一个 4KB 的物理页面，扣除这个数据结构的头部以后就都用于这个数组，其大小的计算如下所示：

```
#define MAXHANDLES ((PAGE_SIZE-sizeof(struct WineProcess))/sizeof(struct Object*))
```

所以“打开对象表”的大小大约是 1020。这与 Windows 相比当然差距很大(在理论上，Windows 打开对象表的大小几乎是无限的)，但是实际上也够了。

前面我一直说 Handle 就是下标，其实这只是就其逻辑意义而言，严格地说是经过换算的下标。这是因为：首先，0 不是一个合法的 handle 数值。另一方面，Handle 的数值都是 4 的倍数，所反映的是以字节为单位的位移。如果 index 是真正意义上的下标，那么 handle 的数值就是(index+1)* sizeof(Object*)。

总之，每个进程都有一个打开对象表，为该进程所含的诸多线程所共享，表中的每个有效指针都指向一个 Object 数据结构。内核中的对象就好像磁盘上的文件一样，都有个从创建到打开、到关闭、最后被删除的“生命周期”。每个对象都由一个 Object 数据结构作为代

表，其定义为：

```
/*
 * object definition
 * - object namespace is indexed by name and class
 */
typedef struct Object {
    struct list_head        o_objlist;          /* obj list (must be 1st) */
#ifdef OBJECT_MAGIC
    int                     o_magic;            /* magic number (debugging) */
#endif
    atomic_t                o_count;            /* usage count */
    wait_queue_head_t       o_wait;             /* waiting process list */
    struct ObjectClass      *o_class;           /* object class */
    struct oname            o_name;             /* name of object */
    void                    *o_private;         /* type-specific data */
} Object;
```

对象的类型是以其数据结构中的 o_class 为标志的，这个指针指向哪一个 ObjectClass 数据结构，这个对象就是什么类型。此外，每个对象都可以有个对象名，就好像每个文件都有个文件名一样，o_name 就是用来保持对象名的数据结构。

显然，内核中对象的数量可以很大，这里有个如何寻找某个特定对象的问题。所以首先要按对象的类型划分，然后为每个类别都安排若干个以对象名的 hash 值划分的队列。具体对象的 Object 数据结构就按其对象名的 hash 值链入所属类别的某个队列中，结构中的队列头 o_objlist 就是用于这个目的。

读者不难看出，Object 数据结构中各个成分所反映的基本上都是作为某类对象的共性，而并没有反映出具体对象的个性。所以这个结构中有个无类型的指针 o_private，用来指向一个描述具体对象的数据结构。当对象的类型为线程时，这就是个 WineThread 数据结构；当对象的类型为文件时，这就是个 WineFile 数据结构；如此等等。

如上所述，Object 数据结构中的指针 o_class 标志着对象的类型。这是个指向某个 ObjectClass 结构的指针，每个 ObjectClass 结构代表着一种对象类型，其定义如下。

```
struct ObjectClass {
    struct list_head        oc_next;
    const char              oc_type[6];     /* type name (5 chars + NUL) */
    int                     oc_flags;
#define OCF_DONT_NAME_ANON   0x00000001 /* don't name anonymous objects */

    int (*constructor)(Object *, void *);
    int (*reconstructor)(Object *, void *);
    void (*destructor)(Object *);
    int (*describe)(Object *, struct wineserver_read_buf *);
    int (*poll)(struct wait_table_entry *, struct WineThread *);
    void (*detach)(Object *, struct WineProcess *);
```

```
    /* lock governing access to object lists */
    rwlock_t            oc_lock;

    /* named object hash */
    struct list_head    oc_nobjs[OBJCLASSNOBJSSIZE];

    /* anonymous object list */
    struct list_head    oc_aobjs;
};
```

结构中的第一个队列头oc_next用来构成一个对象类型、即ObjectClass数据结构的队列。而队列头数组 oc_nobjs[OBJCLASSNOBJSSIZE]，则用来构成该对象类的hash队列数组，具体的对象根据其对象名的 hash 值决定链入其中的哪一个队列。数组的大小OBJCLASSNOBJSSIZE 定义为16。对象也可以是无名的，无名的对象都链入所属类别的无名队列、即 oc_aobjs 队列。

ObjectClass 数据结构中最具实质性的成分是一组函数指针，特别是其中的“构造”函数指针 constructor，因为它决定了如何构造出一个具体类型的对象。目前 keenel-win32 定义了 event_objclass、file_objclass、mutex_objclass、semaphore_objclass、process_objclass、thread_objclass、rpc_client_objclass、rpc_server_objclass、rpc_service_objclass、section_objclass等 11 种对象类型的 ObjectClass 数据结构。

我们不妨以比较简单的“信号量(semaphore)”类型为例来说明对象的创建。

函数 CreateSemaphoreA()是 kernel_win32 对同名系统调用在内核中的实现，其目的是为当前进程创建并打开一个(内核中的)带有对象名的信号量。我们先跳过系统调用如何进入内核这一步，从这个函数开始看有关的代码。

```
int CreateSemaphoreA(struct WineThread *filp, struct WiocCreateSemaphoreA *args)
{
    HANDLE hSemaphore;
    Object *obj;

    obj = CreateObject(filp,&semaphore_objclass,args->lpName,args, &hSemaphore);
    . . . . . .
    return (int) hSemaphore;
} /* end CreateSemaphoreA() */
```

调用参数 filp 指向当前线程的 WineThread 数据结构，这是由 kernel_win32 所实现的系统调用机制所提供的。另一个参数 args，则是指向一个 WiocCreateSemaphoreA 数据结构的指针。Kernel_win32 所实现的系统调用机制把 Windows 系统调用的参数都组装在一个数据结构中，再把这个结构的起始地址作为参数传给内核。为此，Kernel_win32 为其所实现的每个 Windows 系统调用都定义了一个数据结构，WiocCreateSemaphoreA 就是为系统调用CreateSemaphoreA()的参数而定义的数据结构。

CreateSemaphoreA()的主体就是对函数 CreateObject()的调用。由于要创建的对象是信号量，就把此种对象类型的数据结构 semaphore_objclass 的起始地址也作为参数传了下去。参

数&hSemaphore 则是用来返回 Handle 的。

[CreateSemaphoreA() > CreateObject()]

```
Object *CreateObject(struct WineThread *thread, struct ObjectClass *clss,
                     const char *name, void *data, HANDLE *hObject)
{
    struct WineProcess *process;
    struct oname oname;
    Object *obj, **ppobj, **epobj;
    int err;

    *hObject = NULL;

    /* retrieve the name */
    err = fetch_oname(&oname,name);
    if (err<0)
        return ERR_PTR(err);

    /* allocate an object */
    obj = _AllocObject(clss,&oname,data);
    if (oname.name) putname(oname.name);
    if (IS_ERR(obj))
        return obj;

    /* find a handle slot */
    process = GetWineProcess(thread);
    epobj = &process->wp_handles[MAXHANDLES];
    write_lock(&process->wp_lock);
    for (ppobj=process->wp_handles; ppobj<epobj; ppobj++)
        if (!*ppobj)    goto found_handle;

    write_unlock(&process->wp_lock);
    objput(obj);
    return ERR_PTR(-EMFILE);

found_handle:
    /* make link to object */
    objget(obj);
    *ppobj = obj;
    write_unlock(&process->wp_lock);
    ppobj++; /* don't use the NULL handle */
    *hObject = (HANDLE) ((char*)ppobj - (char*)process->wp_handles);
    return obj;
```

```
} /* end CreateObject() */
```

这个函数的操作可以分成两大部分。第一部分是对_AllocObject()的调用，旨在创建具体的对象。第二部分是将指向所创建对象的指针“安装”在当前进程的“打开对象表”中，并将相应的下标转换成 Handle。为便于阅读讨论，我们先假定第一部分的操作业已完成，_AllocObject()已经返回所创建的 Object 结构的指针，先看看对于“打开对象表”的操作，这是从注释行“/* find a handle slot */”开始的。至于 putname()、objget ()、objput()一类的函数，那只是递增或递减数据结构中的引用计数(减到 0 就要释放其占用的存储空间)，并不影响对于实质性操作的讨论。

“打开对象表”在 WineProcess 数据结构中，而从上面传下来的只是个 WineThread 指针。所以这里要通过 GetWineProcess()找到当前线程所属的 Wine 进程，这其实只是从 WineThread 数据结构中获取其 wt_process 指针而已。

找到了所属进程的 WineProcess 数据结构以后，就通过一个 for 循环扫描其“打开对象表”,旨在找到一个空闲的位置,指针为 0 就表示空闲。这也说明了为什么 0 不能被用作 handle 的值。找到以后，就把新创建对象的 obj 指针填写到这个位置上。而 handle 数值的计算，则可以看出基本上是该指针在数组中的(字节)位移量加 4，实际上就是下标加 1 后再乘 4。注意 handle 的值是通过调用参数 hObject 返回的。

再回到第一部分，即对象的创建，这是由_AllocObject()完成的。

[CreateSemaphoreA() > CreateObject() > _AllocObject()]

```
static Object *_AllocObject(struct ObjectClass *clss, struct oname *name, void *data)
{
    Object *obj;

    . . . . . .
    /* create and initialise an object */
    obj = (Object *) kmalloc(sizeof(Object),GFP_KERNEL);
    . . . . . .
    atomic_set(&obj->o_count,1);
    init_waitqueue_head(&obj->o_wait);
    . . . . . .
    /* name anonymous objects as "class:objaddr" if so requested */
    if (!name->name && ~clss->oc_flags&OCF_DONT_NAME_ANON) {
        . . . . . .
    }
    /* cut'n'paste the name from the caller's name buffer */
    else {
        obj->o_name.name = name->name;
        obj->o_name.nhash = name->nhash;
        name->name = NULL;
    }

    /* attach to appropriate object class list */
```

```
    obj->o_class = clss;
    if (obj->o_name.name)
        list_add(&obj->o_objlist,
            &clss->oc_nobjs[obj->o_name.nhash&OBJCLASSNOBJSMASK]);
    else
        list_add(&obj->o_objlist,&clss->oc_aobjs);
    . . . . . .
    err = clss->constructor(obj,data); /* call the object constructor */
    if (err==0)    goto cleanup_1;

    . . . . . .
 cleanup_1:
    write_unlock(&clss->oc_lock);
 cleanup_0:
    return obj;
} /* end _AllocObject() */
```

首先是由 kmalloc()为所创建的对象分配存储空间。然后是 Object 结构的初始化，包括把对象名(及其 hash 值)拷贝到 Object 结构中的 o_name 里面。

接着，如果有对象名，就根据其 hash 值把所创建的 Object 结构挂入所属对象类别的相应 hash 队列中，否则就挂入该类别的无名对象队列中。

下面就是实质性的操作了，这是通过所属类别提供的 constructor 函数完成的。对于“信号量”而言，该类对象的类型数据结构是 semaphore_objclass。

```
struct ObjectClass semaphore_objclass = {
    oc_type:  "SEMA ",
    constructor:    SemaphoreConstructor,
    reconstructor:  SemaphoreReconstructor,
    destructor:     SemaphoreDestructor,
    poll:       SemaphorePoll,
    describe:  SemaphoreDescribe
};
```

显然，其 constructor 函数是 SemaphoreConstructor()，所以实际调用的就是这个函数。

[CreateSemaphoreA() > CreateObject() > _AllocObject() > SemaphoreConstructor()]

```
static int SemaphoreConstructor(Object *obj, void *data)
{
    struct WiocCreateSemaphoreA *args = data;
    struct WineSemaphore *semaphore;

    . . . . . .
    semaphore =
```

```
        (struct WineSemaphore *) kmalloc(sizeof(struct WineSemaphore), GFP_KERNEL);
    . . . . . .
    obj->o_private = semaphore;
    semaphore->ws_count = args->lInitialCount;
    semaphore->ws_max   = args->lMaximumCount;
    return 0;
} /* end SemaphoreConstructor() */
```

程序很简单，先分配一个 WineSemaphore 数据结构所需的空间，这个数据结构才是真正意义上的具体的“对象”、一个信号量。当然，还要使 Object 结构中的指针 o_private 指向这个 WineSemaphore 数据结构。而对于这个信号量的初始化，则只不过是把作为参数传下来的 lInitialCount 和 lMaximumCount 填写进去。

显然，kernel-win32 另行实现了一个信号量机制，而不是把 Windows 应用程序的信号量操作“嫁接”到 Linux 已有的信号量机制上。对于相对而言比较简单的信号量机制，这样当然也是可以的(也还值得推敲)。而对于比较复杂的机制、特别是文件操作，那就只能走嫁接这“华山一条路”了，以后我们还要通过文件操作看 kernel-win32 是如何实现这种嫁接的。

此外，前面曾经提到，task_ornament 数据结构中有个指针 to_ops，指向某个 task_ornament_operations 数据结构。这个数据结构的主体是一组函数指针，说明内核在 close、exit、signal、execve、fork 等操作时应该对上述种种附加的数据结构和对象做些什么。目前 kernel-win32 只定义了一种 task_ornament_operations 数据结构：

```
static const struct task_ornament_operations wineserver_ornament_ops = {
    name:       "wineserver",
    owner:      THIS_MODULE,
    close:      ThreadOrnamentClose,
    exit:       ThreadOrnamentExit,
    signal:     ThreadOrnamentSignal,
    execve:     ThreadOrnamentExecve,
    fork:       ThreadOrnamentFork
};
```

那么怎样使用这个数据结构呢？与这些函数指针基本对应，kernel-win32 的代码中还有下列几个函数：

task_ornament_notify_exit()、
task_ornament_notify_signal()、
task_ornament_notify_execve()、
task_ornament_notify_fork()、

这些函数会根据相应的函数指针调用有关的程序。对这几个函数的调用则出现在 kernel-win32 对 Linux 内核所打的补丁中。以对于 fork.c 所打的补丁为例：

```
do_fork(. . . . . .)
{
    . . . . . .
```

```
        p->p_cptr = NULL;
        init_waitqueue_head(&p->wait_chldexit);
        p->vfork_sem = NULL;
-       spin_lock_init(&p->alloc_lock);
+       rwlock_init(&p->alloc_lock);
        . . . . . .
        current->counter >>= 1;
        if (!current->counter)
                current->need_resched = 1;
+       /*
+        * tell any ornaments to duplicate themselves
+        */
+       INIT_LIST_HEAD(&p->ornaments);
+       task_ornament_notify_fork(current,p,clone_flags);
        . . . . . .
}
```

这里作的第一个修改是把 spin_lock_init()换成 rwlock_init()。下面实质性的修改则是将子进程(线程)的 ornaments 队列头加以初始化，然后调用 task_ornament_notify_fork()。调用的目的，按注释所述，是复制当前进程(线程)的 ornaments 队列中的各个附件。

[do_fork() > task_ornament_notify_fork()]

```
static __inline__ void task_ornament_notify_fork(struct task_struct *tsk,
                                        struct task_struct *child, unsigned long clone_flags)
{
        /* only iterate through the list if there _is_ a list */
        if (!list_empty(&tsk->ornaments))
                __task_ornament_notify_fork(tsk,child,clone_flags);
}
```

实际的操作由另一个函数__task_ornament_notify_fork()完成。

[do_fork() > task_ornament_notify_fork() > __task_ornament_notify_fork()]

```
void __task_ornament_notify_fork(struct task_struct *tsk,
                                struct task_struct *child, unsigned long clone_flags)
{
        struct task_ornament buoy[2], *orn;
        struct list_head *ptr;

        atomic_set(&buoy[0].to_count, 0x3fffffff);
        INIT_LIST_HEAD(&buoy[0].to_list);
        buoy[0].to_ops = NULL;
```

```
        atomic_set(&buoy[1].to_count, 0x3fffffff);
        INIT_LIST_HEAD(&buoy[1].to_list);
        buoy[1].to_ops = NULL;

        /* loop through all task ornaments, but be careful in case the
         * list is rearranged. The buoy is used as a marker between the current
         * position and the next
         */
        write_lock(&tsk->alloc_lock);
        list_add_tail(&buoy[1].to_list,&tsk->ornaments);
        ptr = tsk->ornaments.next;

        for (;;) {
                /* skip over buoys from other processors */
                for (;;) {
                        if (ptr==&buoy[1].to_list)
                                goto end_of_list_reached;

                        orn = list_entry(ptr,struct task_ornament,to_list);
                        if (orn->to_ops)
                                break;
                        ptr = ptr->next;
                }

                /* we've found a real ornament */
                ornget(orn);

                /* stuff a buoy in the queue after it */
                list_add(&buoy[0].to_list, ptr);
                write_unlock(&tsk->alloc_lock);

                /* call the operation */
                orn->to_ops->fork(orn,tsk,child,clone_flags);
                ornput(tsk,orn);

                /* remove the buoy */
                write_lock(&tsk->alloc_lock);
                ptr = buoy[0].to_list.next;
                list_del(&buoy[0].to_list);
        }
end_of_list_reached:
        list_del(&buoy[1].to_list);
        write_unlock(&tsk->alloc_lock);
} /* end __task_ornament_notify_fork() */
```

前面讲过，挂在 ornaments 队列中的数据结构是 task_ornament，但 task_ornament 又是 WineThread 数据结构中的一个成分，所以一般实际挂入队列的是 WineThread 数据结构。但是，这并不排除把独立的 task_ornament 结构挂入 ornaments 队列。这里的两个局部量 buoy[2]就是如此。这个词的原意是“浮标”，在这里就是作为分隔标志使用的。

首先是对两个浮标的初始化，浮标的特点是它的 to_ops 指针为 0。然后把浮标 buoy[1]挂入父进程(线程)的 ornaments 队列末尾。这样，在浮标 buoy[1]之前的所有 task_ornament 数据结构都是需要复制的。而若此后再有新的 task_ornament 数据结构挂入这个队列，就不在应该复制之列了。接着使指针 ptr 指向队列中的第一个 task_ornament 数据结构，下面就是对队列中所有成员的 for 循环了。

这里有两个嵌套的 for 循环。外层 for 循环是对于队列中所有有效成员(浮标除外)的循环，而内层 for 循环则有两个目的。其一是在队列中碰到浮标 buoy[1]时便跳转到 end_of_list_reached:，结束整个复制过程。其二是跳过队列中别的浮标，即由别的线程所设置的浮标(特点是其 to_ops 指针为 0)。

对于队列中的每个有效成员，即每个 WineThread 数据结构，先在其后面再加上一个浮标 buoy[0]作为分隔，接着就调用由相应 task_ornament_operations 数据结构提供的函数进行处理，即 orn->to_ops->fork()。显然，这就是 ThreadOrnamentFork()。

```
[do_fork() > task_ornament_notify_fork() > __task_ornament_notify_fork()
  > ThreadOrnamentFork()]

/*
 * notification that fork/clone has set up the new process and
 * is just about to dispatch it
 * - no ornaments will have been copied by default
 */
static void ThreadOrnamentFork(struct task_ornament *ornament, struct task_struct *parent,
                               struct task_struct *child, unsigned long clone_flags)
{
    ktrace("%s(%p,%p,%p,%08lx)\n",
           __FUNCTION__,ornament,parent,child,clone_flags);
} /* end ThreadOrnamentFork() */
```

出乎意外的是，这里只是调用了一下 ktrace()、即 printk()。函数代码前面的注释说“默认的操作是不复制 ornaments”。我想，合理的解释之一是开发还在进行中，还没有来得及实现。但是，在 fork()的时候是否真的应该为子进程(线程)复制父进程(线程)的 ornaments 队列呢？如果是，那么 Wine 线程与其所落实的 task_struct 数据结构之间还是不是一一对应的关系呢？如果不是，那么从__task_ornament_notify_fork()开始的一系列操作岂不是无的放矢？

实际上 ThreadOrnamentExecve()、ThreadOrnamentSignal()也是没有实质性的操作。

还有个问题，就是一个 Wine 线程到底有几个 WineThread 数据结构？如果只有一个的话，那么应该出现在谁的 ornaments 队列里？我们不妨带着这些问题到代码中找找答案。

首先，task_ornament 结构是由 add_task_ornament()挂入 ornaments 队列的：

```
+/*
+ * add an ornament to a task
+ */
+void add_task_ornament(struct task_struct *tsk,
+                       struct task_ornament *orn)
+{
+    ornget(orn);
+    write_lock(&amp;tsk-&gt;alloc_lock);
+    list_add_tail(&amp;orn-&gt;to_list,&amp;tsk-&gt;ornaments);
+    write_unlock(&amp;tsk-&gt;alloc_lock);
+} /* end add_task_ornament() */
```

如前所述，task_ornament 结构是 WineThread 数据结构中的一个成分，是“连接件”。所以说的是 add_task_ornament()，实际上加入队列的是 WineThread 数据结构(除前述的“浮标”以外)。那么是谁在调用这个函数呢？搜索的结果是唯一的，那就是 ThreadConstructor()：

```
static int ThreadConstructor(Object *obj, void *data)
{
    struct WineThreadConsData *wtcd = data;
    struct WineProcess *process;
    struct WineThread *thread;
    . . . . . .
    process = (struct WineProcess *) wtcd->wtcd_process->o_private;
    . . . . . .
    thread = (struct WineThread *) kmalloc(sizeof(struct WineThread), GFP_KERNEL);
    . . . . . .
    obj->o_private = thread;
    . . . . . .
    thread->wt_task = wtcd->wtcd_task;
    . . . . . .
    list_add(&thread->wt_list,&process->wp_threads);
    . . . . . .
    add_task_ornament(thread->wt_task,&thread->wt_ornament);
    . . . . . .
    return 0;
} /* end ThreadConstructor() */
```

这就是线程对象的构造函数。这个函数分配、构造了一个 WineThread 数据结构，并将它挂入两个队列。一个是该线程所属 Wine 进程的 wp_threads 队列，把属于同一个 Wine 进程的线程都串在一起。另一个队列就是某个 task_struct 结构的 ornaments 队列。谁的 task_struct 结构？这是作为参数传下来的。如果我们向上“顺藤摸瓜”，就可以发现来源于 InitialiseWin32()，而在那里这个 task_struct 结构指针的源头是 current，就是当前进程(线程)的 task_struct 结构。

InitialiseWin32()是 kernel-win32 用来实现系统调用 Win32Init()的内核函数(其实 Windows

并没有这么个系统调用，这是 kernel-win32 增添出来的)，其用意是让每个线程一开始时就调用一下这个系统调用。这在测试程序 test/semaphore.c 中可以看得很清楚，那里的 main()一下子 fork()出 5 个线程，都执行 child()，而 child()的第一个语句就是 Win32Init()。由此可见，每个 WineThread 数据结构实际上只是挂入其对应 task_struct 结构的 ornaments 队列。

再看另一个佐证。前面 CreateSemaphoreA()的第一个调用参数是个 WineThread 结构指针，这是后面的处理所要求的，这一点事实上绝大多数系统调用都是一样。显然，这应该是当前线程的 WineThread 结构。然而 Linux 内核怎么找到这个 WineThread 结构呢？kernel-win32 为此提供了一个函数 task_ornament_find()，根据给定的 task_struct 结构找到其相应的 WineThread 结构。这个函数是在 sys_win32()中调用的(在另一篇漫谈中还要讨论)，调用时的参数是 current、即当前进程(线程)的 task_struct 结构指针。显然这是在当前进程(线程)的 ornaments 队列中寻找：

```
/*
 * find an ornament of a particular type attached to a specific task
 */
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() */
```

这里的 for 循环搜索的确实是 task_struct 中的 ornaments 队列，而且队列中第一个符合条件的 task_ornament 数据结构、从而相应的 WineThread 数据结构，就是所要的结果。那么条件是什么呢？条件是结构中的 task_ornament_operations 指针相符。可是 kernel-win32 一共才定义了一种 task_ornament_operations 结构，那就是 wineserver_ornament_ops。而且实际上也看不出再定义别的此类数据结构的必要。所以，所找到的必然就是由 Win32Init()挂入该队列的那个 WineThread 数据结构，而且是队列中唯一的 WineThread 数据结构。

既然如此，那__task_ornament_notify_fork()又何必弄得那么复杂呢？再说，要是一个线

程、例如 child()、调用了 Win32Init()两次，那又会怎样呢？既然实际上只有、也只需要一个 WineThread 数据结构，在 task_struct 结构中放上一个指针不是更好吗？

所以，我看这不仅仅是具体的实现已经做到了哪一步的问题，实际上也反映了作者对整个方案的构思和设计是比较凌乱的。

下一篇将介绍 kernel-win32 的文件操作。