漫谈兼容内核之三:
Kernel-win32 的文件操作

毛德操

上一篇漫谈中分析/介绍了 Kernel-win32 的对象管理机制。在各种对象类型中，文件显然是最重要、最复杂(就其操作而言)的类型。如果说像“信号量”之类的简单对象还有可能独立地加以实现，那么像文件这样的复杂对象这就不太现实，并且实际上也不合适了。所以，本文的目的就是通过几个典型文件操作的实现介绍 kernel-win32 是怎样把 Windows 用于文件操作的系统调用“嫁接”到 Linux 内核中的有关成分上。

文件的创建

如前文所述，所有的对象都有个 Object 数据结构，里面有个 ObjectClass 指针，它指向什么 ObjectClass 数据结构，这个对象就是什么类型。“文件”就是一种对象类型，它的数据结构是 file_objclass：

```
struct ObjectClass file_objclass = {
        oc_type:  "FILE ",
        constructor:     FileConstructor,
        reconstructor:  FileReconstructor,
        destructor:      FileDestructor,
        poll:        FilePoll,
        describe:  FileDescribe,
        oc_flags: OCF_DONT_NAME_ANON /* names shared from FCTRL class */
};
```

需要创建一个文件时、或者说需要创建一个类型为文件的对象时，函数_AllocObject()将调用这个结构所提供的函数 FileConstructor()，以构造出代表着这个具体对象的数据结构，并且创建并/或打开具体的文件。

另一方面，Object 数据结构中还有个指针 o_private，指向由上述构造函数生成、代表着具体类别对象的数据结构。对于文件类对象，这个数据结构是 WineFile：

```
/*
 * process-access file object definition
 * - can't be shared, since they have to have separate attributes for the file
 *     access
 */
struct WineFile {
        struct list_head            wf_ctllist;          /* file control list next */
        WineFileControl          *wf_control;       /* the file control object */
        struct file                 *wf_file;            /* the Linux file */
        __u32                       wf_access;         /* file access mode */
        __u32                       wf_sharing;         /* sharing mode */
```

```
    __u32                   wf_attrs;           /* file open attributes */
};
```

这个数据结构代表着目标文件的一次“打开”。也即一个文件访问上下文。至于没有被打开的文件，则只是存在于磁盘上、或本来就是个外设，在内存中一般不存在相应的数据结构。

注意这里的指针 wf_file，这是一个 struct file 指针。我们知道，struct file 就是 Linux 内核中代表着已打开文件的数据结构，存在于每个进程的“打开文件表”中的就是 struct file 指针。由于 struct file 数据结构所代表的是目标文件的一次打开，所以可以断定 WineFile 所代表的也是一次打开、即一个上下文(例如可以通过 lseek()取得的当前读/写位置)，而不是目标文件本身。

显然，对于已经打开的文件，除了代表着文件访问上下文的数据结构以外，还应该有代表着目标文件本身的数据结构。这就是“文件控制类”的对象，其数据结构是 WineFileControl：

```
struct ObjectClass file_control_objclass = {
    oc_type:  "FCTRL",
    constructor:    FileControlConstructor,
    reconstructor:  FileControlReconstructor,
    destructor:     FileControlDestructor,
    poll:       FileControlPoll,
    describe:  FileControlDescribe
};
```

文件控制类对象的 o_private 数据结构则是 WineFileControl：

```
/*
 * file-control object definition
 * - links together WineFile objects that use the same file
 */
struct WineFileControl {
    struct list_head        wfc_accessors;      /* who's accessing this file */
    spinlock_t              wfc_lock;           /* govern access to list */
    Object                  *wfc_myself;        /* my own object */
};
```

这里的 wfc_myself 指向其本身的 Object 数据结构。队列头 wfc_accessors 用来维持一个“访问者”队列，这是一个 WineFile 数据结构的队列。WineFile 数据结构通过其队列头 wf_ctllist 挂入这个队列。WineFileControl 与 WineFile 之间是一对多的关系。只要一个文件被打开，内存中就得有一个(并且只有一个)代表着这个文件的 WineFileControl 数据结构，而 WineFile 数据结构的数量则取决于对此文件同时共存的上下文个数。在某种意义上，WineFile 是对(Linux 的)struct file 数据结构的包装和扩充，而 WineFileControl 是对 struct dentry 数据结构的的补充。但是，应该在 WineFileControl 中加以补充的 Windows 文件所独有(有别于 Linux 文件属性)的属性有很多，而这里却什么也没有。所以目前这个数据结构的定义只是提供了一个空的骨架，还有待充实。

还有值得一提的是，打开目标文件以后的 struct file 指针 wf_file 保存在 WineFile 数据结构中，而不是保存在 Linux 进程(线程)的打开文件表中。这意味着设计者的意图是另起炉灶、完全由自己来管理 Windows 进程所打开的文件，而与 Linux 的有关机制分道扬镳。可是这是有可能带来问题的。举个例子来说明这一点：假定一个 Windows 进程在运行中发生了问题，或者被别的进程 kill，从而执行了 do_exit()。在退出运行的过程中，do_exit()会扫描该进程的打开文件表，根据表中的 struct file 指针关闭所有已打开文件、并释放资源。可是现在 Windows 进程的 struct file 指针不在打开文件表中，因而就享受不到这个待遇。设计者显然意识到这一点，所以在有关的内核代码上打了补丁，让它调用一个外挂函数(见前一篇漫谈)。然而，我们却看到有关的外挂函数尚未实现。对于诸如此类的问题，是尽量利用 Linux 内核原有的代码较好？还是另起炉灶较好？应该说是各有利弊，具体的情况需要具体分析，总之是值得推敲的。

介绍了这些背景以后，我们可以阅读 kernel-win32 所实现的代码了。这里看三个系统调用的代码，分别是 CreateFileA()、ReadFile()、和 CloseHandle()。了解了这三种操作，对于 kernel-win32 文件操作的实现就知道了个大概，再进一步阅读其它有关的代码也就不难了。这三个系统调用的函数名与一般 Windows 文献中所见有所不同，但是这没有什么关系，因为这些函数是内核中不向外界开放(导出)的函数，叫什么都可以。

先看 CreateFileA()，这个函数根据给定的路径名打开一个文件，如果这个文件尚不存在就加以创建、并且打开。这跟 Linux 的 sys_open()基本上是一致的，只是函数名 sys_open()突出了“打开”而 CreateFileA()突出了“创建”。不过，除 CreateFile()外，Windows 另有一个系统调用 OpenFile()，这很容易让人误以为前者是只创建不打开，而后者是只打开不创建。

还有，函数名 CreateFileA()中的“A”表示 ASCII，意思是所涉及的字符串(例如文件名)是 8 位 ASCII 码。与之相对的是“W”，表示 16 位“宽”字符，或 Unicode。

下面就来看代码。

```
/*
 * open a file object, maybe creating if non-existent
 */
int CreateFileA(struct WineThread *thread, struct WiocCreateFileA *args)
{
    HANDLE hFile;
    Object *obj;

    /* must have a name... */
    if (!args->lpFilename)
        return -EINVAL;

    /* ...but create the object _without_ a name and attach one later
     * (need a separate file-access object for each CreateFileA
     */
    obj = CreateObject(thread, &file_objclass, NULL, args, &hFile);
    if (IS_ERR(obj))
        return PTR_ERR(obj);
```

```
    ktrace("*** [%d] CreateFileA(%p) = %p\n",current->pid,obj,hFile);

    objput(obj);
    return (int) hFile;

} /* end CreateFileA() */
```

Kernel-win32 所实现的系统调用通过数据结构传递参数，这是与 Windows 不同、而带有 Linux 风格的做法；原因就是 Kernel-win32 通过 Linux 系统调用来“搭载”实现 Windows 系统调用。对于 CreateFileA()，这个数据结构是 struct WiocCreateFileA。

```
struct WiocCreateFileA {
    LPCSTR              __pad__ lpFilename;
    DWORD               __pad__ dwDesiredAccess;
    DWORD               __pad__ dwShareMode;
    LPSECURITY_ATTRIBUTES   __pad__   lpSecurityAttributes;
    DWORD                   __pad__   dwCreationDisposition;
    DWORD                   __pad__   dwFlagsAndAttributes;
    HANDLE                  __pad__   hTemplateFile;
};
```

可见与 open()的参数有相当大的不同，这里限于篇幅不作介绍了，读者可参阅有关的 Windows 文献和资料。

程序的主体就是 CreateObject()。如我在上一篇漫谈中所示，这个函数先调用 _AllocObject()创建具体的 Object 数据结构(并完成实际目标的创建和/或打开操作)，然后把指向该数据结构的指针安装在所属进程的“打开对象表”中，并返回实质上是数组下标的 Handle。而_AllocObject()则进一步细化，通过由 file_objclass 数据结构所提供的函数指针调用 FileConstructor()，这才是实质性的操作。

[CreateFileA() > CreateObject() > _AllocObject() > FileConstructor()]

```
/*
 * construct a file access object (allocate its private data)
 * - called by CreateObject if the file does not already exists
 * - called with the object class lock held
 */
static int FileConstructor(Object *obj, void *data)
{
    struct WiocCreateFileA *args = data;
    struct WineFile *wf;
    struct Object *fco;
    int err, flags, mode;
```

```
ktrace("FileConstructor(%p)\n",obj);

/* construct the file information record */
wf = (struct WineFile *) kmalloc(sizeof(struct WineFile),GFP_KERNEL);
if (!wf)
	return -ENOMEM;

obj->o_private = wf;

wf->wf_access = args->dwDesiredAccess;
wf->wf_sharing = args->dwShareMode;
wf->wf_attrs = args->dwFlagsAndAttributes;

/* gain access to the file control object */
fco = AllocObject(&file_control_objclass, args->lpFilename, data);
if (IS_ERR(fco)) {
	err = PTR_ERR(fco);
	goto cleanup_priv;
}

wf->wf_control = fco->o_private;

spin_lock(&wf->wf_control->wfc_lock);

/* make use of the name stored in the file control object */
obj->o_name.name = fco->o_name.name;

/* check that we can share access */
err = FileCheckSharing(args, wf->wf_control);
if (err<0)
	goto cleanup_fco;

/* determine Linux file open parameters */
switch (args->dwCreationDisposition) {
	case CREATE_NEW:		flags = O_CREAT | O_EXCL;	break;
	case CREATE_ALWAYS:		flags = O_CREAT | O_TRUNC;	break;
	case OPEN_ALWAYS:		flags = O_CREAT;		break;
	case TRUNCATE_EXISTING:	flags = O_TRUNC;		break;
	case OPEN_EXISTING:		flags = 0;			break;
	default:
			err = -EINVAL;
	goto cleanup_fco;
}
```

```
switch (args->dwDesiredAccess & (GENERIC_READ|GENERIC_WRITE)) {
        case 0:                                 break;
        case GENERIC_READ:              flags |= O_RDONLY;     break;
        case GENERIC_WRITE:             flags |= O_WRONLY;     break;
        case GENERIC_READ|GENERIC_WRITE:        flags |= O_RDWR; break;
}
mode = (args->dwFlagsAndAttributes & FILE_ATTRIBUTE_READONLY)? 0444 : 0666;

/* open the Linux file */
wf->wf_file = filp_open(obj->o_name.name, flags, mode);
if (IS_ERR(wf->wf_file)) {
        err = PTR_ERR(wf->wf_file);
        goto cleanup_fco;
}

/* don't permit a directory to be opened */
if (S_ISDIR(wf->wf_file->f_dentry->d_inode->i_mode)) {
        err = -EACCES;
        goto cleanup_file;
}

list_add(&wf->wf_ctllist, &wf->wf_control->wfc_accessors);
spin_unlock(&wf->wf_control->wfc_lock);

kdebug("FileConstructor: f_count=%d i_count=%d i_wc=%d\n",
            atomic_read(&wf->wf_file->f_count),
            atomic_read(&wf->wf_file->f_dentry->d_inode->i_count),
            atomic_read(&wf->wf_file->f_dentry->d_inode->i_writecount));

return 0;

/* clean up on error */
cleanup_file:
fput(wf->wf_file);

cleanup_fco:
obj->o_name.name = NULL;
spin_unlock(&wf->wf_control->wfc_lock);
objput(wf->wf_control->wfc_myself);

cleanup_priv:
poison(wf,sizeof(struct WineFile));
kfree(wf);
return err;
```

```
} /* end FileConstructor() */
```

简而言之，这个函数的操作主要有这么一些：

1. 通过 kmalloc()分配一块空间用于 WineFile 数据结构。
2. 将调用参数纪录在 WineFile 数据结构中。
3. 通过 AllocObject()找到或创建代表着目标文件的“文件控制”对象。
4. 通过 FileCheckSharing()检查是否允许调用参数所要求的访问模式。
5. 将调用参数所要求的操作模式(CREATE_NEW、CREATE_ALWAYS、OPEN_ALWAYS 等等)和访问模式(GENERIC_READ、GENERIC_WRITE 等)换算成 Linux 的相应标志位。
6. 调用 Linux 内核函数 filp_open()，以打开目标文件，并将已打开文件的 struct file 结构指针填写在 WineFile 数据结构中。
7. 通过 list_add()将前面创建的 WineFile 数据结构挂入相应 WineFileControl 数据结构的 wfc_accessors 队列中。

上面 AllocObject()的主体就是对_AllocObject()的调用。而_AllocObject()，则先试图根据类型(在这里是 file_control_objclass)和对象名找到目标对象，找不到才加以创建。所以“文件控制”对象与“打开文件”对象之间是一对多的关系。如果目标文件尚未被打开，因此需要创建“文件控制”对象的话，就会调用他的构建函数 FileControlConstructor()：

[CreateFileA() > CreateObject() > AllocObject() > _AllocObject() > FileControlConstructor()]

```
/*
 * construct a file control object (allocate its private data)
 * - called by AllocObject if the file does not already exists
 * - called with the object class lock held
 */
static int    FileControlConstructor(Object *obj, void *data)
{
    struct WineFileControl *wfc;

    ktrace("FileControlConstructor(%p)\n",obj);

    /* construct the file information record */
    wfc = (struct WineFileControl *) kmalloc(sizeof(struct WineFileControl),GFP_KERNEL);
    if (!wfc)
        return -ENOMEM;

    obj->o_private = wfc;

    INIT_LIST_HEAD(&wfc->wfc_accessors);
    spin_lock_init(&wfc->wfc_lock);
    wfc->wfc_myself = obj;

    return 0;
```

```
} /* end FileControlConstructor() */
```

显然，这个函数毫无特殊之处，而且比前面的 FileConstructor()简单多了。这当然是因为许多功能和特性尚未实现的缘故，而并不是本来就应该这么简单。

回到 FileConstructor()的代码中，还有 FileCheckSharing()是值得一提的。这个函数根据 CreateFileA()的调用参数 dwDesiredAccess 和 dwShareMode 检查所要求的访问模式(例如 GENERIC_READ 和 GENERIC_WRITE)和共享模式(例如 FILE_SHARE_READ)是否与目标文件已有的各个“打开”相容。具体地说，就是扫描目标文件的 WineFileControl 结构中的 wfc_accessors 队列，与队列中各个 WineFile 结构中记载的访问模式和共享模式进行比较，以确定是否相容。

创建并/或打开了文件以后，就可以对文件进行各种操作了，最重要的当然是读/写。所以我们就来看看读文件操作的实现。在 Windows 上，读文件是通过系统调用 NtReadFile()进行的。在 Kernel_win32 的代码中，这个系统调用在内核中的实现就是 ReadFile()。同样，调用参数都包装在一个数据结构中：

```
struct WiocReadFile {
    HANDLE          __pad__     hFile;
    LPVOID          __pad__     lpBuffer;
    DWORD           __pad__     nNumberOfBytesToRead;
    LPDWORD         __pad__     lpNumberOfBytesRead;
    LPOVERLAPPED    __pad__     lpOverlapped;
};
```

前面三个参数的用途是不言自明的。参数 lpNumberOfBytesRead 则是个指针，用来返回实际读出的字节数。显然这与 Linux 的 sys_read()不同，因为后者是把它作为函数值返回的。另一个参数 lpOverlapped 也是个指针，用于异步的读文件操作。所谓异步，是指调用者并不(睡眠)等待操作的完成，而是把访问的要求交给内核以后就先返回干别的事，内核在完成操作以后再予以“回叫(callback)”。但是 Kernel-win32 尚未实现此项功能。

下面我们看 ReadFile()的代码。

```
int ReadFile(struct WineThread *thread, struct WiocReadFile *args)
{
    struct WineFile *wf;
    struct file *file;
    Object *obj = NULL;
    int ret;

    kdebug("ReadFile(%p,%p)\n",thread,args->hFile);

    ret = 0;
    ret = put_user(ret,args->lpNumberOfBytesRead);
    if (ret<0)
        goto cleanup_nobj;
```

```
obj = GetObject(thread,args->hFile,&file_objclass);
if (IS_ERR(obj))
	return PTR_ERR(obj);

wf = obj->o_private;
file = wf->wf_file;

ktrace("*** [%d] ReadFile(%p,%p,%u)\n",
		current->pid,obj,
		args->lpBuffer,
		args->nNumberOfBytesToRead
		);

/* check the file can actually be read */
ret = -EACCES;
if (!(wf->wf_access & GENERIC_READ))
	goto cleanup;

ret = -EBADF;
if (!(file->f_mode & FMODE_READ))
	goto cleanup;

ret = -EINVAL;
if (!file->f_op || !file->f_op->read)
	goto cleanup;

/* check that the area isn't mandatorally locked */
ret = locks_verify_area(FLOCK_VERIFY_READ, file->f_dentry->d_inode,
			file, file->f_pos, args->nNumberOfBytesToRead);
if (ret<0)
	goto cleanup;

/* try and read */
ret = file->f_op->read(file, args->lpBuffer, args->nNumberOfBytesToRead, &file->f_pos);
if (ret<0)
	goto cleanup;

ktrace("*** [%d] ReadFile(%p) read %d bytes\n",current->pid,obj,ret);

put_user(ret,args->lpNumberOfBytesRead);

ret = 0;
cleanup:
```

```
    objput(obj);

    cleanup_nobj:
    ktrace("*** [%d] ReadFile(%p) = %d\n",current->pid,obj,ret);
    return ret;
} /* end ReadFile() */
```

这个函数的逻辑是很简单的：先根据 Handle 通过当前线程的打开对象表找到目标对象，并进一步找到其 WieFile 数据结构，指向目标文件的 struct file 结构的指针就保存在 WieFile 数据结构中；然后在 Wine 和 Linux 文件系统两个层面上检查访问权限；最后通过 Linux 文件系统实施具体的读出。这个过程清楚地表明 Kernel-win32 是怎样把 Windows 的文件操作嫁接到 Linux 文件操作上的。读操作如此，写操作也是大同小异。

再看“关闭文件”的操作。Windows 并没有专门针对文件的“关闭文件”操作，而只有“关闭 Handle”操作，因为文件只是“对象”中的一种。

```
int CloseHandle(struct WineThread *thread, struct WiocCloseHandle *args)
{
    struct WineProcess *process;
    Object **ppobj, **epobj, *obj;
    int last;

    ktrace("*** [%d] CloseHandle(%p,%p)\n",
            current->pid,thread,args->hObject);

    /* validate the handle */
    if (args->hObject<MINHANDLE ||
        args->hObject>=MAXHANDLE ||
        ((__u32)args->hObject & (sizeof(Object*)-1))
        )
        return -EINVAL;

    process = GetWineProcess(thread);

    write_lock(&process->wp_lock);
    ppobj = (Object**)
        ((char*)&process->wp_handles + (__u32)args->hObject – sizeof(Object*));
    obj = *ppobj;
    *ppobj = NULL;

    /* see if this was the last attachment from this "process" */
    epobj = &process->wp_handles[MAXHANDLES];
    last = 1;
    if (obj && obj->o_class->detach) {
        for (ppobj=process->wp_handles; ppobj<epobj; ppobj++) {
```

```
                if (*ppobj==obj) {
                    last = 0; /* yes */
                    break;
                }
            }
        }

        write_unlock(&process->wp_lock);

        if (!obj)
            return -EBADF;

        if (last && obj->o_class->detach)
            obj->o_class->detach(obj,process); /* last attachment gone */

        objput(obj);
        return 0;
} /* end CloseHandle() */
```

开始时在所属进程的打开对象表中搜索，根据 Handle 找到目标对象的过程跟前面 ReadFile()中调用的 GetObject()基本相同，不同的是这里要把打开对象表中的相应表项写成 NULL，以断开跟目标对象的连系。接下去的意图很明显，就是检查这是否当前进程对目标对象的最后一个 Handle，即是否对目标对象的最后一个打开，如果是就调用该类对象的 detach 操作(注意代码中的注释“yes”意思反了)。可是，看一下文件类对象的数据结构 file_objclass，就可以发现它并没有提供 detach 函数，所以实际上并不会执行。看来，在 kernel-win32 的设计中，关闭文件类对象时并不需要 detach 操作。然而在关闭对象时总得对目标对象做点什么啊，否则其数据结构所占的空间何时才能释放？这就是下面 objput()要做的事。

[CloseHandle() > objput()]

```
/*
 * decrement an object's usage count
 * - when usage count reaches zero, the object is destroyed
 * - uses the class's rwlock to govern access to the count
 */
void objput(Object *obj)
{
        ktrace("objput(%p)\n",obj);

        if (!obj) return;

#ifdef OBJECT_MAGIC
        if (obj->o_magic!=OBJECT_MAGIC)
```

```
            panic("bad object magic\n");
#endif

        write_lock(&obj->o_class->oc_lock);
        if (!atomic_dec_and_test(&obj->o_count))
            goto still_in_use;

#ifdef OBJECT_MAGIC
        obj->o_magic = 0x01010101;
#endif

        list_del(&obj->o_objlist);
        obj->o_class->destructor(obj);

        write_unlock(&obj->o_class->oc_lock);

        /* quick insanity check */
        if (waitqueue_active(&obj->o_wait)) {
            printk(KERN_ALERT "wineserver:"
                    " object being deleted is still being waited upon\n");
            return;
        }

        if (obj->o_name.name) putname(obj->o_name.name);
        poison(obj,sizeof(Object));
        kfree(obj);
        return;

        still_in_use:
        write_unlock(&obj->o_class->oc_lock);
} /* end objput() */
```

首先通过 atomic_dec_and_test()递减目标对象的引用计数、并测试其结果。如果尚未达到 0，就说明目标对象还有用户，所以就不需要有进一步的处理。否则，要是计数达到了 0，那就要把它从对象队列中删除，并调用该类对象的 destructor 函数。文件类对象的 destructor 函数是 FileDestructor()。

[CloseHandle() > objput() > FileDestructor()]

```
/*
  * destroy a file (discard its private data)
  */
static void FileDestructor(Object *obj)
{
```

```
    struct WineFile *wf = obj->o_private;

    kdebug("FileDestructor: f_count=%d i_count=%d i_wc=%d\n",
            atomic_read(&wf->wf_file->f_count),
            atomic_read(&wf->wf_file->f_dentry->d_inode->i_count),
            atomic_read(&wf->wf_file->f_dentry->d_inode->i_writecount)
            );

    ktrace("FileDestructor(%p)\n",obj);

    /* discard my association with the Linux file */
    fput(wf->wf_file);

    /* unlink from the controlling object */
    list_del(&wf->wf_ctllist);
    obj->o_name.name = NULL;
    objput(wf->wf_control->wfc_myself);

    poison(obj->o_private,sizeof(struct WineFile));
    kfree(obj->o_private);

} /* end FileDestructor() */
```

先通过 fput()切断与 Linux 文件系统中目标文件 struct file 结构的连系，然后将 WineFile 数据结构从目标文件控制对象的队列中删除，再对目标文件控制对象也实施 objput()，最后释放 WineFile 数据结构。在释放前还调用了一个函数 poison()，那只是将 WineFile 数据结构中的内容破坏掉。注意前面 objput()中也调用了 poison()和 kfree()，那是释放目标对象的 Object 数据结构，而这里释放的是与之配套的 WineFile 数据结构。

从上述创建文件、读文件、和关闭文件这三个文件操作的代码中，我们可以体会到：“内核差异内核补”、把 Windows 的文件操作嫁接到 Linux 文件操作上面、其逻辑和思路是多么简单明了，这与 Wine 试图在核外解决问题的做法形成了鲜明的对比。虽然本文没有涉及写文件操作的代码，但是读者想必明白那是大同小异。

更重要的是，与 Wine 相比，逻辑上和操作上的这种简化和优化必将大大提高文件操作的效率。

但是，另一方面，我们也应看到，Kernel-win32 只是朝正确的方向走了一小步，迄今所实现的只是系统调用界面的一个很粗糙、很不完整的骨架，离实用还相距甚远，而且许多具体的设计和实现也值得推敲。

在 CreateFileA()的过程中，我们在前面 FileConstructor()的代码中可以看到调用参数之一 lpSecurityAttributes 根本就没有被引用；另一个参数 dwFlagsAndAttributes 虽然被引用了，但只是检查了其中的“只读”标志位。然而这两个调用参数恰恰是很重要的。例如 lpSecurityAttributes 是个指针，应该指向一个 SECURITY_ATTRIBUTES 数据结构，Windows DDK 中有这个数据结构的定义：

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD nLength;
    LPVOID lpSecurityDescriptor;
    BOOL bInheritHandle;
} SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES, *LPSECURITY_ATTRIBUTES;
```

可见，这个数据结构中不光有关于(所创建)目标文件安全特性的描述，还规定了这个文件本次打开后的上下文是否可遗传。尽管许多应用程序在调用 CreateFileA()时会对此给出一个 NULL 指针，表示采用默认的行为特征，但是有些应用程序却会利用这个参数。忽略这个参数无疑有可能会使那样的应用程序产生一些令人莫名其妙的结果。

再看 dwFlagsAndAttributes，这个参数也绝不只是一个 FILE_ATTRIBUTE_READONLY 标志位那么简单。例如其中有个标志位是 FILE_FLAG_DELETE_ON_CLOSE，表示在关闭这个文件时应将其删除。再例如 FILE_ATTRIBUTE_COMPRESSED 也是个标志位，其作用不言自明。显然，Kernel-win32 只考虑了其中的 FILE_ATTRIBUTE_READONLY，而留下了大量其它的标志位未作处理，甚至在数据结构中也没有为保存这些成分留下空间。实际上，实现那些标志位的工作量可能远远超过 Kernel-win32 迄今已有的实现。

此外，ReadFile()的参数 lpOverlapped 也没有被用到，这是用来实现异步操作的。Linux 内核也支持异步文件操作，但是显然这里还没有考虑把它嫁接过去。

还有特别值得一提的是已打开对象的遗传问题。显然 Kernel-win32 对此还没有考虑，因为否则在 Object 结构中(或者别的什么数据结构中)总得有个地方来保存有关的信息。

最后还要提一下跨进程的 Handle 复制。我在以前的漫谈中也曾讲到，只有在内核中才能有效率地实现这个功能，但是我们并没有看到 Kernel-win32 在做这件事的迹象。

不过，尽管差距还很大，由于 Kernel-win32 毕竟是在内核中弥补这些差距，所以原则上并不存在大的障碍，只是还没有来得及做、或者是否准备做的问题。

实际上，这正是所谓“20/80 原理”的一个例证。就文件操作而言，有了打开/创建、读、写、移动读写位置这些基本功能，许多应用程序就可以运行了。实现这些基本功能的工作量和复杂性可能只是 20%，而受惠的应用软件可能是 80%。但是要让剩下来的那 20%也能运行起来，所要求的工作量和复杂性则可能有 80%。所以前面那 20%的工作当然应该优先。然而，想要达到“高度兼容”的目的，是绝不能置那 20%于不顾的。就像艺术表现的真实性往往在于细节一样，兼容程度的高低也往往深受细节的影响。而且，有些“细节”考虑甚至可能会导致推翻原来的设计方案。例如关于跨进程复制 Handle 的考虑、关于遗传方式的考虑、就使得 Wine 实际上不可能达到高度兼容，至少是不可能达到比较有效率的高度兼容(更何况还有设备驱动的问题)。而 Kernel-win32，说是对 Wine 的优化，实质上就是推翻了 Wine 原来的设计方案。好在 Kernel-win32 的工作已经做到了内核中，从大的方面来说，总体的设计方案已经不太会因为某些细节不好实现而被推翻了，但是一些具体的设计、具体的实现、则仍是可以推敲和改变的。

对于兼容内核，Kernel-win32 要实现的相当于它的一个子集，即其系统调用界面。因此兼容内核理应从 Kernel-win32 吸取营养，甚至直接“拿来”。不过由于 Kernel-win32 还很很粗糙、很不完整，一些具体的设计和实现也值得推敲，所以能直接“拿来”的大概不多。倒是它对于各种成分的轻重缓急和实现次序方面的考虑，还是值得、也应该借鉴的。