版权声明

作者简介

灰狐，又名 grayfox、nokyo 等，马甲众多，自 2005 年至 2009 年就读于成都信息工程学院，喜编程、擅灌水，以结识志同道合者为好，常混迹于各大 BBS，潜水居多。

MSN：peiyaoqiang@msn.com
Gtalk：peiyaoqiang@gmail.com
Email：peiyaoqiang@126.com
QQ 群：醉爱编程 [iCoodle]（群号：78298479）
个人主页：http://nokyo.blogbus.com

相关链接

灰狐's Blog ：http://nokyo.blogbus.com
Adly's Blog ：http://adly.blog.sohu.com
泡面代码社区 ：http://www.icoodle.org
邪恶八进制 ：http://www.eviloctal.com

关于 iCoodle 的一点说明

在您访问 iCoodle 的时候，可能还没有正式开放，因为我们苦于没有空间支持，免费的空间又总有这样那样的缺陷，希望能有好心的读者给予赞助一点服务器空间，我们只需要放置一个主页，发表原创性质的文章，仅需要您有 Access 数据库和支持 ASP。

同时希望各位有心一起进步的会员加入到 iCoodle 来，我们共创一个灿烂的明天。

前言

一开始，我只是想把自己的学习笔记、心得发出来，承蒙风泽大哥的厚爱，建议我把这些内容进行整理，制作成电子书，以方便他人的阅读、学习。

本书所涉及到的内容，大多都已经网络上广为流传，我本人是通过《Windows 驱动开发技术详解》和楚狂人的《Windows 驱动编程基础教程》学习的，因此本书中的内容大多是来自这两本教程中的一些总结和概括，不敢擅称“原创”。

在学习 Windows 驱动开发的过程中，道路不可谓曲折，从一开始学习直到半年后还是不得其门而入，对驱动学习的畏惧感骤然上升。当然，这也与我在学习的过程中心境不够清醒有关，往往感到浮躁，越是想快速入门，越是倍受挫折。

《Windows 驱动开发技术详解》，这本书的确不错，由浅入深，逐步推进地对驱动开发的相关概念都进行了详细的讲解，非常适合新手入门。

学习驱动，亲自动手编写程序是必须的，而且要大量地编写，只有在亲自把代码编写完毕、调试成功之后，我们才能真正理解理论上的概念。

在我学习的过程中，非常感谢 adly 的指导，每当我遇到了一些难题，总会在他那里得到一针见血的答案。在 adly 的指点下，我也算是简单地入门了，不过想到自己在学习过程中的艰辛，还是感觉有必要把自己的学习过程记录下来，给后来的新手们一个指导。

由于我自己对驱动还不是很熟悉，在本文中难免会出现一些错误的概念或结论，如果您看到了这些错误，衷心地恳请您给予指导。

需要说明的是，本书并不是一本详尽的入门书籍，它只是根据我本人的学习过程进行的经验总结，适合作为专门教程的辅助读物。

目录

注：本书所述内容均来自互联网，例如楚狂人的《Windows 驱动编程基础教程》和张帆的《Windows 驱动开发技术详解》以及其他一些参考资料。本书属于学习笔记系列文章汇总，并不敢称为原创，凡因此书引发的版权问题，作者本人不负任何责任。

本书所有代码均经过作者的初步测试，所有章节提及到的代码均在附书源码的对应“.\Src\第 x 章”目录中有完整源码工程。

如果您在本书中发现错误，请及时给作者（http://nokyo.blogbus.com）留言说明，我也很愿意与您一起讨论相关的问题。

第一章 驱动开发环境的搭建

工欲善其事，必先利其器。同样地，我们要想顺利地在驱动开发的道路上启程，准备一款趁手的工具是必须的。

1.1 关于DDK

开发驱动程序必备的一个东西就是 DDK（Device Development Kit，设备驱动开发包），它跟我们在 ring3 常听到的 SDK 差不多，只不过它们分别支持开发不同的程序而已。DDK 和微软其他的产品一样，具有良好的向后兼容性，比如你用 DDK 2000 开发的驱动在 DDK XP 里面同样可以编译，但反之却不能保证。

DDK 常见的版本有 DDK 2000、DDK XP 等（别提 Win 98，那已经是古董了），不过现在微软推出的驱动开发包已经不叫 DDK 了，而是 WDK（Windows Driver Kit，Windows 驱动开发包）。同时您还可能听说过 DriverStudio 之类的驱动开发工具，其实那只是对 DDK 的简单封装，跟 SDK 与 MFC 的关系差不多，不过 DriverStudio 不仅仅是对 DDK 的封装，而是个完整的开发工具包，它提供了很多有用的工具用于驱动程序的开发和调试，不过这些工具我们可以单独提取出来使用。

DDK 可以在微软的官方网站下载，当然也可以在 Google 搜索到很多链接，推荐您至少使用 DDK XP 或更高的版本，下载到本地后直接双击安装就可以了。

1.2 关于驱动程序的编译

前面我们说了，DDK 相当于在开发普通的 ring3 应用程序所使用的 SDK，那么我们是否有与开发 ring3 应用程序对应的 IDE 呢，比如 VC 6.0、VC.NET 2003、Delphi 等。

很遗憾，除了 DriverStudio，关于驱动开发的 IDE 我知道的不多，多数情况下我们都是使用 DDK 提供的 builder.exe 在命令行下直接编译连接生成“.sys”文件，同时还需要自己编写 makefile 和 sources 文件。

对于使用 VC 开始学习编程的人们来说，makefile 概念有点可能有点陌生，而且很多习惯了使用 IDE 的人们往往很讨厌麻烦的命令行编译。实际上我们要知道，VC 本身只是个框架，“编译”这个工作还是由一个名为“cl.exe”的命令行工具执行的，VC 不过是通过一个良好的界面帮我们完成了一些参数设置的工作。

知道了这一点，我们就应该明白，使用 VC 来编译驱动程序是可行的，实际上已经有很多文章详细介绍如何使用 VC 环境开发驱动程序，版本从经典的 VC 6.0 到最新的 VS 2008 都没有被遗漏。

这里我们介绍一个简单的方法，即使用 EasySys 这个小工具。这是一个很实用的小工具，它可以通过简单的设置生成一个完整的“.dsw”工程，我们使用 VC 6.0 打开该工程文件就可以方便地编写代码了，然后直接按“F7”完成编译连接这个过程。

EasySys 是开源的程序，所有人都可以通过修改代码来定制自己的 EasySys，但在我们对驱动开发比较熟悉之前，不建议这样做，如果我们生成的代码不合理很容易造成蓝屏。网上有很多大牛发布有自己修改过的版本，我们直接下载使用就可以了。

它通过一个批处理文件来设置编译参数，但我们在开始学习时可以不理会它是如何工作的，只要能使用就是好东西。

我们先来简单看看 EasySys 的界面和使用方法，如图 1-1 所示：

图 1-1 EasySys 运行界面

第一个编译框我们要填写的是需要建立的工程名，不用废话了；第二个编辑框是这个工程要存在的目录，也不废话了；第三个下拉列表框是选择我们已安装的 DDK 版本；最后一个是选择我们的驱动程序要运行的目标系统平台，因为驱动与系统相关性很强，我们在开发的时候需要注意这个问题。

点击【Create】按钮之后会在指定的目录下创建一个文件夹，名字就是工程名，进入该目录看看，会不会觉得很眼熟？如图 1-2 所示：

图 1-2 EasySys 生成的目录

双击如图 1-2 所示的“Test.dsw”文件便可使用 VC 6.0 打开这个工程，我们直接编辑“Test.c”和“Test.h”这两个文件，然后编译、连接就可以生成驱动目标文件。

1.3 驱动程序的运行

通常我们生成的驱动程序格式都是“.sys”，虽然它也是 PE 格式，但不能直接运行，必须被加载到系统中。

因此我们在编写好一个驱动程序后，如果想运行查看结果，则必须使用一些工具来进行加载，当然也可以自己编写，我们将会在第十章介绍如何编程加载驱动。现在我们使用一个名为“KmdManager”的小工具来加载驱动。其运行界面如图 1-3 所示：

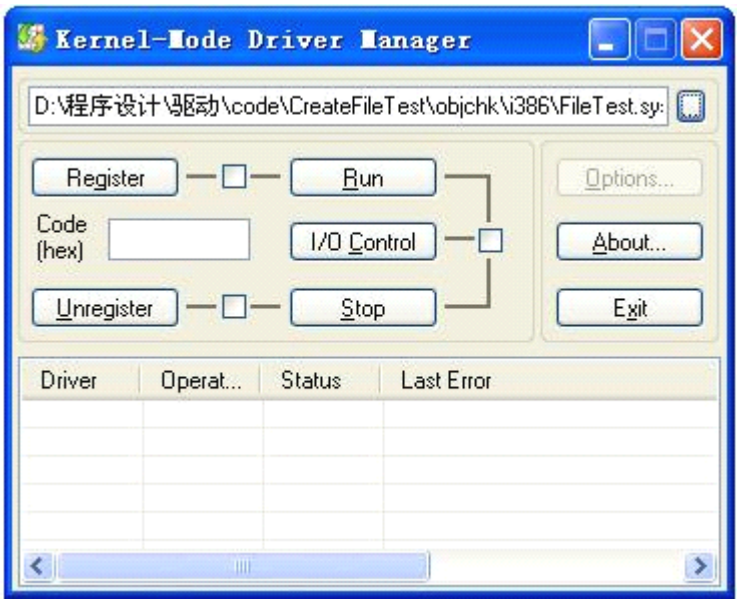

图 1-3 KmdManager 运行界面

如图 1-3，我们先浏览找到自己需要加载的驱动文件，然后依次点击【Register】和【Run】按钮即可运行驱动程序（其实是执行 DriverEntry 例程），在测试完毕后记得点击【Stop】和【Unregister】按钮将其卸载。

关于驱动程序开发环境的搭建就简单介绍到这里，如果你使用的是 VC 其他版本，请自行通过 Google 搜索相关设置指导。

【补充知识】

实际上，常见的 Windows 驱动程序是可以分成两类的：一类是不支持即插即用功能的 NT 式驱动程序，另一类是支持即插即用的 WDM 式驱动程序。

NT 式驱动的安装是基于服务的，可以通过修改注册表进行，也可以直接通过服务函数如 CreateService 进行安装；但 WDM 式驱动不同，它安装的时候需要通过编写一个 inf 文件进行控制。

除此之外，它们所使用的头文件也不大相同，例如 NT 式驱动往往需要导入一个名为“ntddk.h”的头文件，而 WDM 式驱动需要的却是“wdm.h”头文件。我们在学习的过程中所编写的大多属于 NT 式驱动，除非我们需要自己的设备支持即插即用，才需要考虑编写 WDM 式驱动程序。

如果没有特别声明的话，该电子书中所涉及到的驱动程序均为 NT 式驱动。

第二章 驱动程序的结构

驱动程序与其他的普通应用程序一样，都是有着一定的框架结构的，在《Windows 驱动开发技术详解》的第一章提供了两个最简单的完整驱动程序。由于代码量还是有点多，就不在这里贴出来了（本书不是入门教程，而是入门教程的辅助读物）。

2.1 驱动程序的头文件

在第一章我们曾经提到过，NT 式驱动需要导入的头文件是“ntddk.h”，而 WDM 式驱动需要导入的是“wdm.h”，即：#include "ntddk.h";。

在驱动中用到的变量或函数都需要指定分配在分页或非分页内存中，分页内存在物理内存不够的情况下可能会被交换出去，对于一些需要高 IRQL 的例程绝对不能被交换出页面，因此它们必须被定义为非分页内存。

关于非页内存与非分页内存的详细介绍，我看到过一篇非常不错的文章，“http://blog.chinaunix.net/u/25096/showart_463863.html”，这篇文章对于新手可能不太容易看懂，不过等学到一定程序再回来看就会很容易理解。

通常来说在驱动程序的自定义头文件中都是定义了一些宏或函数声明，没有什么特别需要注意的地方。

在配套工具目录的第二章里面有一个由 EasySys 生成的框架，有兴趣的可以看一下它的文件结构，暂时看不懂也可以先有个大概印象。

2.2 驱动程序的入口点

犹如控制台程序需要 main、Win32 程序需要 WinMain、DLL 程序需要 DllMain 一样，驱动程序也有自己的入口点，即 DriverEntry。

DriverEntry 需要被加载到 INIT 内存区域中，这样当驱动被卸载后它可以退出内存。DriverEntry 是由内核中的 I/O 管理器负责调用的，它有两个参数 DriverObject 和 RegistryPath（当然形参的名字我们可以自己改变，这个是由 EasySys 生成的）。其中 DriverObject 是由 I/O 管理器传递进来的驱动对象，RegistryPath 则指向此驱动负责的注册表。

在配套工具目录的第二章里面，我们可以看到 DriverEntry 首先是定义了一些变量，然后调用 IoCreateDevice 创建设备对象，紧接着调用 IoCreateSymbolicLink 创建符号链接，为了条理清晰，我们在 2.3 中介绍这两段内容。

现在我们来看看紧接在 IoCreateSymbolicLink 之后的内容，如下所示：

```
DriverObject->MajorFunction[IRP_MJ_CREATE]  = TestDispatchCreate;
DriverObject->MajorFunction[IRP_MJ_CLOSE]   = TestDispatchClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]= TestDispatchDeviceControl;
DriverObject->DriverUnload                  = TestUnload;
```

这是驱动程序在向 Windows 的 I/O 管理器注册一些回调函数。上面代码的含义是：当驱动程序将被卸载时自动调用 TestUnload 例程；当驱动程序接收到 IRP_MJ_CREATE 时自动调用 TestDispatchCreate；剩下两个也是一样。

关于 IRP 的详细概念我们到后面再讲，现在我们就把它理解成 ring3 的“消息”一样的东西。当我们的驱动程序接收到不同的 IRP 就表明发生了不同的事件，然后我们及时给予处理。

在驱动对象 DriverObject 中，有个函数指针数组 MajorFunction，它里面的每一个元素都记录着一个函数的地址对应着相应的 IRP，我们可以通过简单地设置这个数组将 IRP 与相应的派遣函数关联起来。诸如 IRP_MJ_CREATE 其实是使用#define 定义的一个宏，比如 IRP_MJ_CREATE 实际上就是 0x00，而 IRP_MJ_CLOSE 则是 0x02 等。

由于在进入 DriverEntry 之前，I/O 管理器会将_IopInvalidDeviceRequest 的地址填满整个 MajorFunction 数组，因此除了我们自行设置过的 IRP 之外，其他的 IRP 都与系统默认的 _IopInvalidDeviceRequest 函数关联。

2.3 创建设备例程

这里我们又遇到了一个新的概念“例程”，其实也不新，驱动中所说的例程实际上就是函数的另外一种说法，我们毋需过于关心这种细节（实际上例程与函数还是有所区别的，但我们不作关心）。

创建设备本来是在 DriverEntry 中完成的，不过这里为了讲解方便，我专门将其抽了出来，下面我们来看看相关代码：

```
// 创建设备对象
RtlInitUnicodeString(&ntDeviceName, TEST_DEVICE_NAME_W);
Status = IoCreateDevice(
            DriverObject,
            sizeof(DEVICE_EXTENSION),       // DeviceExtensionSize
            &ntDeviceName,                  // DeviceName
            FILE_DEVICE_TEST,               // DeviceType
            0,                              // DeviceCharacteristics
            TRUE,                           // Exclusive
            &deviceObject                   // [OUT]
            );
if(!NT_SUCCESS(Status))
{
    KdPrint(("[Test] IoCreateDevice Error Code = 0x%X\n", Status));

    return Status;
}

deviceExtension = (PDEVICE_EXTENSION)deviceObject->DeviceExtension;
// 创建符号链接
RtlInitUnicodeString(&dosDeviceName, TEST_DOS_DEVICE_NAME_W);
Status = IoCreateSymbolicLink(&dosDeviceName, &ntDeviceName);
if(!NT_SUCCESS(Status))
{
    KdPrint(("[Test] IoCreateSymbolicLink Error Code = 0x%X\n", Status));

    IoDeleteDevice(deviceObject);
    return Status;
}
```

在上述代码中，我们先来看创建设备对象的代码，首先我们使用 RtlInitUnicodeString 函数来初始化 UNICODE 字符串，关于字符串的用法请参考其他书籍；然后我们调用函数 IoCreateDevice 来完成创建设备对象的功能，该函数返回一个 NTSTATUS 值，有一个宏 NT_SUCCESS 可以很方便地判断这个 NTSTATUS 是否成功。

紧接着我们调用 IoCreateSymbolicLink 创建一个符号链接，符号链接有什么用？这么说吧，前面我们创建的设备对象虽然有个参数指定了设备名称，但是这个设备名称只能在内核态可见，也就说 ring3 的应用层程序是看不见它的，因此驱动程序需要向 ring3 公布一个符号链接，这个链接指向真正的设备名称，而 ring3 的应用程序可以通过该符号链接找到驱动程序进行通信。实际上我们经常所说的 C 盘、D 盘就是一个符号链接，它们在内核中的真正设备对象是“\Device\HarddiskVolume1”和“\Device\HarddiskVolume2”。

在内核模式下，符号链接是以“\??\”（或“\DosDevices\”）开头的，如C 盘就是“\??\C:”，而在用户模式下，则是以“\\.\”开头的，如 C 盘就是“\\.\C:”。

2.4 卸载驱动例程

卸载驱动例程是我们在 DriverEntry 中自己定义的，当驱动被卸载时 I/O 管理器负责调用该例程，它主要做一些扫尾处理的工作。相关代码如下所示：

```
UNICODE_STRING dosDeviceName;
// 释放其他资源

// 删除符号链接
RtlInitUnicodeString(&dosDeviceName, TEST_DOS_DEVICE_NAME_W);
IoDeleteSymbolicLink(&dosDeviceName);

// 删除设备对象
IoDeleteDevice(DriverObject->DeviceObject);
KdPrint(("[Test] Unloaded"));
```

上述代码没有难以理解的地方，我们就简单介绍一下代码最后一句的 KdPrint 吧。由于驱动程序工作于内核态，不像我们控制台的程序一样可以使用 printf 输出一些信息，也不像 Win32 程序可以通过 MessageBox 来弹出一个对话框，它要想输出一些信息，就需要调用 DbgPrint 函数，不过这个函数输出的信息我们无法直接看到，需要使用一些专门的工具，比如 DbgView 等。

有些内容我们只想在调试版输出，在发行版忽略，因此 DDK 中定义了一个宏 KdPrint，它在发行版不被编译，只在调试版才会运行。KdPrint 的用法很奇怪，由于它是这样定义的：#define KdPrint(_x_) DbgPrint _x_，这就导致了它的用法很奇怪，在使用时最外层要有两个连续的括号，真不知道 DDK 为啥这样定义，定义的时候就多加一个括号多方便啊。

2.5 派遣例程

派遣例程是处理 IRP 的，为了快速入门，现在我们暂时不处理和 IRP 有关的地方，因此这里就使用 EasySys 生成的框架，不做修改，第九章我们会详细介绍 IRP 的内容。

第三章 编写第一个驱动程序

其实第一个驱动程序挺难写的，因为整体的框架有 EasySys 帮我们生成好了，难道我们也像初学 C 语言时打印个“Hello,world!”？

这样也太低级了，就一句 KdPrint(("Hello,world!"));谁不会啊？但要写的难一点吧，又怕看不懂，很发愁啊。也罢，我们就先介绍一下字符串吧。

3.1 内核模式下的字符串操作

内核模式与用户模式一样都是有 ANSI 和 UNICODE 两种字符串，但可以这么说，Windows 内核是使用 Unicode 编码的，ANSI 只在很少的特殊场合才会使用，而这种场合往往是非常罕见的（摘自楚狂人的驱动教程），因此我们就不考虑 ANSI 字符串了，这里只介绍 Unicode 字符串的用法。

Unicode 字符串有一个结构体定义如下：

```
typedef struct _UNICODE_STRING {
    USHORT Length;                  // 字符串的长度（字节数）
    USHORT MaximumLength;           // 字符串缓冲区的长度（字节数）
    PWSTR  Buffer;                  // 字符串缓冲区
} UNICODE_STRING, *PUNICODE_STRING;
```

需要注意的是，当我们定义了一个 UNICODE_STRING 变量之后，它的 Buffer 域还没有分配空间，因此我们不能直接赋值，好的做法是使用微软提供的 Rtl 系列函数。

```
UNICODE_STRING str;
RtlInitUnicodeString(&str, L"my first string!");
```

或者如下所示：

```
#include <ntdef.h>
UNICODE_STRING str = RTL_CONSTANT_STRING(L"my first string!");
```

看了上面的代码之后我们回顾一下第二章讲解创建设备对象和符号链接的代码，是不是就用 RtlInitUnicode 函数来初始化的。

还有一个需要注意的地方是，与 ring3 不同，我们的 UNICODE 字符串并不是以“\0”来表示字符串结束的，而是依靠 UNICODE_STRING 的 Length 域来确定。

字符串的很多操作都有相应的函数，例如字符串的复制可以使用 RtlCopyUnicodeString 函数，字符串的比较可以使用 RtlCompareUnicodeString 函数，字符串转换成大写可以使用 RtlUpcaseUnicodeString 函数（没有转换成小写的），字符串与整数数字互相转换分别可以使用 RtlUnicodeStringToInteger 和 RtlIntegerToUnicodeString 函数。

下面我们来着重说明一下字符串的打印方法。

比如在输出日志记录的时候，我们往往同时涉及数字、字符等信息，在 C 语言中我们可以使用 sprintf 和 swprintf 函数来完成任务，这两个函数在驱动中仍然可以使用，但很不安全，因为有许多 C 语言的运行时函数都是基于 Win32 API 的，在驱动中绝对不能使用，如果我们不清楚哪些可以使用哪些不能使用，就都不要使用，而使用微软推荐的 Rtl 系列函数。对应 sprintf 的功能函数是 RtlStringCbPrintfW，它需要包含头文件“ntstrsafe.h”和静态连接库“ntsafestr.lib”。相关代码如下所示：

```
#include <ntstrsafe.h>
// 任何时候，假设文件路径的长度为有限的都是不对的。应该动态的分配内存。但动态分配内存的
```

```
// 方法还没有讲述，所以这里再次把内存空间定义在局部变量中，也就是所谓的“在栈中”
WCHAR buf[512] = { 0 };
UNICODE_STRING dst;
NTSTATUS status;
……
// 字符串初始化为空串。缓冲区长度为 512*sizeof(WCHAR)
RtlInitEmptyString(dst,dst_buf,512*sizeof(WCHAR));

// 调用 RtlStringCbPrintfW 来进行打印
status = RtlStringCbPrintfW(
            dst->Buffer,L”file path = %wZ file size = %d \r\n”,
            &file_path,file_size);
// 这里调用 wcslen 没问题，这是因为 RtlStringCbPrintfW 打印的字符串是以空结束的。
dst->Length = wcslen(dst->Buffer) * sizeof(WCHAR);
```

RtlStringCbPrintfW 在目标缓冲区内存不足的时候依然可以打印，但是多余的部分被截去了。返回的 status 值为 STATUS_BUFFER_OVERFLOW。调用这个函数之前很难知道究竟需要多长的缓冲区。一般都采取倍增尝试。每次都传入一个为前次尝试长度为 2 倍长度的新缓冲区，直到这个函数返回 STATUS_SUCCESS 为止。

值得注意的是 UNICODE_STRING 类型的指针，通常用%wZ 可以打印出字符串。在不能保证字符串为空结束的时候，必须避免使用%ws 或者%s。其他的打印格式字符串与传统 C 语言中的 printf 函数完全相同。可以尽情使用。

3.2 内核模式下各种开头函数的区别

在驱动开发的过程中，我们可能遇到很多不同开头的函数，如前面我们遇到过的 Rtl 和 Io 系列，此外还有比如 Ex、Ps、Nt 等等。

常见的函数开头及其含义如下表所示。

函数开头	含义
Cc	Cache manager
Cm	Configuration manager
Ex	Executive support routines
FsRtl	File system driver run-time library
Hal	Hardware abstraction layer
Io	I/O manager
Ke	Kernel
Lpc	Local Procedure Call
Lsa	Local security authentication
Mm	Memory manager
Nt	Windows 2000 system services (most of which are exported as Win32 functions) ，例如 NtCreateFile 往往导出为 CreateFile
Ob	Object manager
Po	Power manager
Pp	PnP manager

Ps	Process support
Rtl	Run-time library
Se	Security
Wmi	Windows Management Instrumentation
Zw	Mirror entry point for system services (beginning with Nt) that sets previous access mode to kernel, which eliminates parameter validation, since Nt system services validate parameters only if previous access mode is user see Inside Microsoft Windows 2000

上表中所提及到的并不完整，还有一些 Dbg、Fs、Csr、Etw 等开头的都没有提及到，这些可以等需要用到的时候再进行说明。

3.3 第一个示例程序

现在我们把第二章生成的 Test 工程复制到第三章目录下并打开，然后在“Test.c”开头的函数声明处添加以下声明：

```
BOOL
CreateFileTest(IN PUNICODE_STRING FileName);
```

然后在“Test.c”的最后添加函数定义，如下所示：

```
BOOL
CreateFileTest( IN PUNICODE_STRING FileName)
{
    BOOL bRet   = FALSE;
    HANDLE hFile = NULL;
    NTSTATUS status;
    IO_STATUS_BLOCK Io_Status_Block;

    // 初始化文件路径
    OBJECT_ATTRIBUTES    obj_attrib;
    InitializeObjectAttributes(&obj_attrib,
                               &FileName,
                               OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                               NULL,
                               NULL );
    // 创建文件
    status = ZwCreateFile(hFile,
                          GENERIC_ALL,
                          NULL,
                          Io_Status_Block,
                          NULL,
                          FILE_ATTRIBUTE_NORMAL,
                          FILE_SHARE_READ,
                          FILE_CREATE,
                          FILE_NON_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT,
```

```
                        NULL,
                        0 );

    if (Io_Status_Block.Information != FILE_CREATED)
    {
        bRet = FALSE;
    }
    else
    {
        bRet = TRUE;
    }

    if (hFile)
    {
        ZwClose(hFile);
    }

    return bRet;
}
```

上面的代码你看出问题了么，没有？好吧，我们先来编译一下：咦，出现了 6 个错误，真令人郁闷。

不要紧，我们慢慢来分析错误原因。

错误提示都指向了函数声明和定义的地方，提示“error C2061: syntax error : identifier 'CreateFileTest"，啊，忘了说这时候我们不能直接使用 BOOL 类型的，因为我们 DDK 的头文件中没有定义，它是在“Windows.h”中定义的。

现在我们有两种方法解决，一是把 BOOL 改成 BOOLEAN；二是在自己在文件开头添加一个定义“#define BOOL BOOLEAN”。我们就采用第一种办法吧，把函数声明和定义处的 BOOL 改成 BOOLEAN。

现在我们再次编译：天哪，居然提示有 46 个错误和 5 个警告。

查看错误提示，第一个提示就是“error C2065: 'BOOL' : undeclared identifier”，哦，我们忘了修改函数中第一行的 BOOL 啦，将其改成 BOOLEAN，再次编译：呵呵，变成了 4 个错误和 2 个警告。

下面这些错误比较隐蔽，而且有些错误编译器也未能发现，我就直接说出来吧。

1，我们的形参是 PUNICODE_STRING 类型，因此 InitializeObjectAttributes 函数的第二个参数应该是 FileName 而不是&FileName。

2，ZwCreateFile 函数的第四个参数是 PIO_STATUS_BLOCK 类型，因此我们在调用的时候应该写成&Io_Status_Block。

3，ZwCreateFile 函数与 ring3 的 CreateFile 函数不同，它不能直接获得需要创建或打开的文件路径信息，这个文件路径必须通过一个 OBJECT_ATTRIBUTES 变量传递，在前面我们定义了一个 obj_attrib 却在 ZwCreateFile 函数中忘了调用，因此我们需要把 ZwCreateFile 函数的第三个参数由 NULL 修改成&obj_attrib。

4，我们认真查看 DDK 的帮助文档可以发现，ZwCreateFile 的第一个参数类型应该是

PHANDLE 类型，因此我们这里应该把 hFile 改成&hFile。由于在 ring3 我们很少能够遇到返回一个句柄指针的情况，所以新手往往很容易犯这个错误。

5，最后要指出的是，我们的驱动程序是使用 C 语言编写的，因此我们在使用一个变量前必须在函数开头进行定义，而不能像 C++中那样可以“随用随定义”，否则编译器将会报错，这个务必要注意。但我们这个程序比较小，没有出现这个错误。

需要说明一点，C++同样是可以用来编写驱动程序的，不过需要稍微修改一下代码，也可以使用某些修改过的 C++版 EasySys。

根据上述说明，我们把函数定义分别修改如下所示（函数声明也要根据它修改）：

```
NTSTATUS
CreateFileTest( IN PUNICODE_STRING FileName )
{
    HANDLE hFile = NULL;
    NTSTATUS status;
    IO_STATUS_BLOCK Io_Status_Block;

    // 初始化文件路径
    OBJECT_ATTRIBUTES obj_attrib;
    InitializeObjectAttributes( &obj_attrib,
                                FileName,
                                OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                                NULL,
                                NULL );

    // 创建文件
    status = ZwCreateFile( &hFile,
                                GENERIC_ALL,
                                &obj_attrib,
                                &Io_Status_Block,
                                NULL,
                                FILE_ATTRIBUTE_NORMAL,
                                FILE_SHARE_READ,
                                FILE_CREATE,
                                FILE_NON_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT,
                                NULL,
                                0 );
    if (NT_SUCCESS(status))
    {
        status = STATUS_SUCCESS;
    }
    else
    {
        status = Io_Status_Block.Status;
    }
```

```
    KdPrint(("hFile = %08X", hFile));

    // 关闭句柄
    if (hFile)
    {
        ZwClose(hFile);
    }

    return status;
}
```

这时候再进行编译，就没有一点错误和警告了。

【补充知识】

大部分的 Win32 API 都是通过 Native API 实现的，Native API 函数一般都是 Win32 API 函数前面加上 Nt 两个字符，例如 CreateFile 函数对应着 NtCreateFile 函数，这些 Nt 函数都是在“ntdll.dll”实现的，而多数 Win32 API 都是在“kernel.dll”导出的，也有少部分 GDI 或窗口相关的函数是在“gdi32.dll”和“user32.dll”导出的。

Native API 从用户模式穿越进入到内核模式调用系统服务，这个穿越过程是通过软中断的方式进入的。这个软中断的实现方法在不同版本的 Windows 实现方式略有不同，在 Win 2K 下是通过“int 2eh”实现的，在 Win XP 是通过“sysenter”指令完成的。

软中断会将 Native API 的参数和系统服务号的参数一起传进内核模式，不同的 Native API 会对应不同的系统服务号，这个过程是由 SSDT 辅助完成的。

系统服务函数一般和 Native API 具有相同的名字，例如都是 NtCreateFile，但它们的实现不同，系统服务调用是在“ntoskrnl.exe”导出的。

第四章 在驱动中使用链表

在驱动开发的过程中，我们经常遇到一些使用数组力不从心的情况，这时候就可以考虑使用链表来完成相关功能。

4.1 内存的分配与释放

传统的 C 语言中，分配内存常常使用的函数是 malloc，但在驱动开发过程中这个函数不再有效。驱动中分配内存，最常用的是调用 ExAllocatePoolWithTag 或 ExAllocatePool。

```
// 定义一个内存分配标记
#define MEM_TAG    'MyTt'
// 目标字符串，接下来它需要分配空间。
UNICODE_STRING dst = { 0 };
// 分配空间给目标字符串。根据源字符串的长度。
dst.Buffer = (PWCHAR)ExAllocatePoolWithTag(NonpagedPool,src->Length,MEM_TAG);
if(dst.Buffer == NULL)
{
    // 错误处理
    status = STATUS_INSUFFICIENT_RESOUCRES;
    ……
}
dst.Length = dst.MaximumLength = src->Length;
```

ExAllocatePoolWithTag 的第一个参数 NonpagedPool 表明分配的内存是非分页内存，这样它们可以永远存在于物理内存，而不会被分页交换到硬盘上去；第二个参数是长度；第三个参数是一个所谓的“内存分配标记”。

内存分配标记用于检测内存泄漏。想象一下，我们根据占用越来越多的内存的分配标记，就能大概知道泄漏的来源。一般每个驱动程序定义一个自己的内存标记。也可以在每个模块中定义单独的内存标记。内存标记是随意的 32 位数字。即使冲突也不会有什么问题。

此外也可以分配可分页内存，使用 PagedPool 标识第一个参数即可。

ExAllocatePoolWithTag 分配的内存可以使用 ExFreePool 来释放，否则的话这块内存就会产生泄漏。虽然用户进程关闭后自动释放进程内分配的空间，但驱动不太一样，即使它已经被卸载，空间也不会自动释放，除非重启计算机。

ExFreePool 只需要提供需要释放的指针即可。举例如下：

```
ExFreePool(dst.Buffer);
dst.Buffer = NULL;
dst.Length = dst.MaximumLength = 0;
```

注意，ExFreePool 不能用来释放一个栈空间的指针，否则系统立刻崩溃。诸如下面的代码将会招致立刻蓝屏的灾难：

```
UNICODE_STRING src = RTL_CONST_STRING(L”My source string!”);
ExFreePool(src.Buffer);
```

请务必保持 ExAllocatePoolWithTag 或 ExAllocatePool 和 ExFreePool 的成对关系。

4.2 使用 LIST_ENTRY

Windows 内核提供了一个双向链表结构 LIST_ENTRY，此外还有一些其他的结构，比如 SINGLE_LIST_ENTRY（单向链表），我们这里不作介绍。

LIST_ENTRY 是一个双向链表结构，但直接使用它将毫无任何意义，通常的做法是我们自定义一个结构体，将 LIST_ENTRY 作为该结构体的一个子域，这样给予了我们最大限度的灵活性，因为我们的数据需求千差万别，可能是整数、字符串等等，但只要要简单修改一下便可利用 LIST_ENTRY 轻松实现一个链表。如下所示：

```
typedef struct _MYDATASTRUCT{
    ULONG number;
    LIST_ENTRY ListEntry;
} MYDATASTRUCT, *PMYDATASTRUCT;
```

实际上，如果将 LIST_ENTRY 作为结构体的第一个子域是最简单的做法，当然也可以把它放在后面，比如微软提供的很多结构就不是将其放在第一个子域。

这时候，如果我们想获取节点的地址，需要有一个计算偏移的过程，DDK 里面提供了一个宏 CONTAINING_RECORD 可以在指定结构中找到节点地址的指针。

4.3 使用自旋锁

链表之类的结构总是涉及到恼人的多线程同步问题，这时候就必须使用锁。本章只介绍最简单的自选锁。

有些读者可能疑惑锁存在的意义，其实这和多线程操作有关。在驱动开发的代码中，大多是存在于多线程执行环境的，就是说可能同时有多个线程操作一个变量，这时候就可能会引起不可预料的后果。

虽然，多线程并不是真正的并发，但操作链表的过程翻译成汇编指令后往往是由多条指令组成的，简单如 int c = a+b;之类的代码也是由多条指令组成的，这就是说这些操作不具有原子性，通过反汇编查看一下就很容易明白。

如下的代码演示了如何使用自选锁：

```
KSPIN_LOCK my_spin_lock;
KIRQL irql;
// 初始化
KeInitializeSpinLock(&my_spin_lock);

KeAcquireSpinLock(&my_spin_lock,&irql);
// do something …
KeReleaseSpinLock(&my_spin_lock,irql);
```

在使用的时候，我们把需要同步的代码加在 KeAcquireSpinLock 和 KeReleaseSpinLock 这两个函数之间，这样在一个线程操作完成调用 KeReleaseSpinLock 之前，其他线程只能在 KeAcquireSpinLock 前面等候。KIRQL 是一个中断级，KeAcquireSpinLock 函数会提高当前的中断级，目前我们忽略这个问题。

4.4 演示程序

在下面的程序中，我们演示了链表和自旋锁的操作方法。

```
VOID
LinkListTest()
{
    LIST_ENTRY linkListHead;        // 链表
    PMYDATASTRUCT pData;            // 节点数据
    ULONG i = 0;                    // 计数
    KSPIN_LOCK spin_lock;           // 自旋锁
    KIRQL   irql;                   // 中断级别

    // 初始化
    InitializeListHead(&linkListHead);
    KeInitializeSpinLock(&spin_lock);

    //向链表中插入 10 个元素
    KdPrint(("[Test] Begin insert to link list"));
    // 锁定，注意这里的 irql 是个指针
    KeAcquireSpinLock(&spin_lock, &irql);
    for (i=0 ; i<10 ; i++)
    {
        pData = (PMYDATASTRUCT)ExAllocatePool(PagedPool,sizeof(MYDATASTRUCT));
        pData->number = i;
        InsertHeadList(&linkListHead,&pData->ListEntry);
    }
    // 解锁，注意这里的 irql 不是指针
    KeReleaseSpinLock(&spin_lock, irql);

    //从链表中取出所有数据并显示
    KdPrint(("[Test] Begin remove from link list\n"));
    // 锁定
    KeAcquireSpinLock(&spin_lock, &irql);
    while(!IsListEmpty(&linkListHead))
    {
        PLIST_ENTRY pEntry = RemoveTailList(&linkListHead);
        // 获取节点地址
        pData = CONTAINING_RECORD(pEntry, MYDATASTRUCT, ListEntry);
        // 读取节点数据
        KdPrint(("[Test] %d\n",pData->number));
        ExFreePool(pData);
    }
    // 解锁
    KeReleaseSpinLock(&spin_lock, irql);
}
```

现在我们在 DriverEntry 的 IRP 分发代码后面添加一句：LinkListTest();重新编译生成驱动（注意要 Checked 版本的），然后从“.\objchk\i386”目录中复制出生成的驱动文件。

现在我们需要查看程序的运行效果，首先打开 DbgView，设置过滤条件为“*[Test]*”，然后使用第一章介绍过的 KmdManager 加载此驱动并运行（最好在虚拟机中测试），注意查看 DbgView 的输出，如图 4-1 所示：

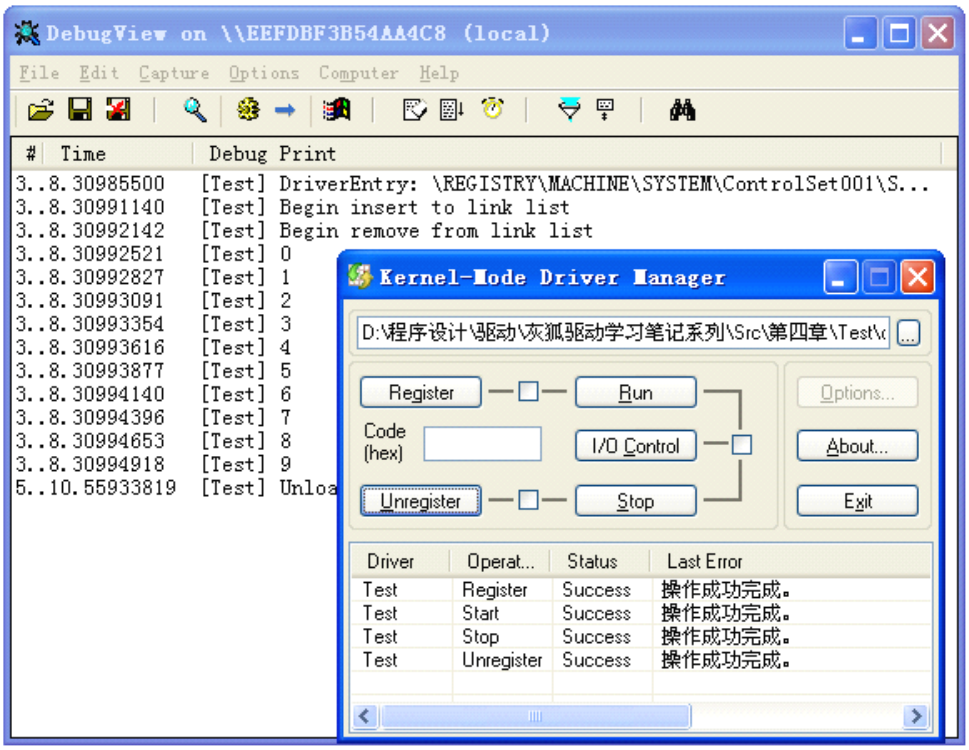

图 4-1 程序演示结果

需要注意的是，像上述代码中在函数中定义一个锁的做法是没有实际意义的，因为它是一个局部变量，被定义在栈中，每当有线程调用该函数时，都会重新初始化一个锁，因此它就失去了本来的作用。

在实际的编程中，我们应该把锁定义为一个全局变量、静态（static）变量或者将其定义在堆中。不过在驱动中应该尽量避免使用全局变量，良好的做法是将需要全局访问的变量定义为设备扩展结构体 DEVICE_EXTENSION 的一个子域。

另外，我们还可以为每个链表都定义并初始化一个锁，在需要向该链表插入或移除节点时不使用前面介绍的普通函数，而是使用如下方法：

```
ExInterlockedInsertHeadList(&linkListHead, &pData->ListEntry, &spin_lock);
pData = (PMYDATASTRUCT)ExInterlockedRemoveHeadList(&linkListHead, &spin_lock);
```

此时在向链表中插入或移除节点时会自动调用关联的锁进行加锁操作，可以有效地保证多线程安全性，详细做法请参考楚狂人的驱动教程。

第五章 在驱动中读写文件

对文件的读写操作一直是我们需要熟练掌握的内容，在 ring3 我们可以使用 CreateFile、ReadFile、WriteFile 等 API，在 ring0 同样很相似，不过函数变成了 ZwCreateFile、ZwReadFile、ZwWriteFile 等内核函数。

5.1 使用 OBJECT_ATTRIBUTES

ZwCreateFile 与 ring3 的 CreateFile 函数有所不同，它不能直接将需要打开或创建的文件路径传递过去，我们必须首先填写一个 OBJECT_ATTRIBUTES 结构，这个结构在内核中被广泛使用，例如后面我们将要介绍的操作注册表函数也会用到它。

这个结构很容易使用，初始化后就可以使用了，初始化过程如下所示：

```
UNICODE_STRING str;
OBJECT_ATTRIBUTES obj_attrib;

RtlInitUnicodeString(&str, L"\\??\\C:\\windows\\notepad.exe");
InitializeObjectAttributes(&obj_attrib,
                           &str,          // 需要操作的对象、比如文件或注册表路径等
                           OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                           NULL,
                           NULL);
```

第三个参数 OBJ_CASE_INSENSITIVE 表示不区分大小写，OBJ_KERNEL_HANDLE 表示将要打开的句柄为内核句柄。

内核句柄比起应用层句柄有很多的好处，例如它可以不受进程或线程的限制，而且在需要打开一个内核句柄时不需要考虑当前是否有权限访问该文件的问题。

5.2 创建、打开文件

创建和打开文件都可使用 ZwCreateFile 函数，它的第一个参数将返回一个文件句柄，所有后续操作都可以通过这个句柄完成，在操作结束后，需要调用 ZwClose 关闭句柄。

ZwCreateFile 函数的第三个参数就是使用我们此前填写的 OBJECT_ATTRIBUTES 结构；它返回的信息通过第四个 IO_STATUS_BLOCK 返回；第八、九个参数联合指明了如何打开或创建文件，详细用法请参考 DDK 帮助文档或 Google 查询。

其中 IO_STATUS_BLOCK 的定义如下所示：

```
typedef struct _IO_STATUS_BLOCK {
    NTSTATUS    Status;
    ULONG    Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

其中 Status 指明了函数的执行结果，如果执行成功它的值将是 STATUS_SUCCESS，否则它将会是一个形如 STATUS_XXX 的错误提示。

此外，DDK 还提供了一个函数 ZwOpenFile 用来简化打开文件的操作，它所需要的参数比 ZwCreateFile 更加简洁，使用更加简单。

5.3 读写文件操作

在内核中读写文件与用户模式下十分相似，它们分别使用 ZwReadFile 和 ZwWriteFile 函数完成。

这两个函数的参数大抵相同，因此我们这里只对 ZwReadFile 的参数作简单介绍，该函数的原型如下所示：

```
NTSTATUS
ZwReadFile(
    IN HANDLE               FileHandle,
    IN HANDLE               Event    OPTIONAL,
    IN PIO_APC_ROUTINE      ApcRoutine    OPTIONAL,
    IN PVOID                ApcContext    OPTIONAL,
    OUT PIO_STATUS_BLOCK    IoStatusBlock,
    OUT PVOID               Buffer,
    IN ULONG                Length,
    IN PLARGE_INTEGER       ByteOffset    OPTIONAL,
    IN PULONG               Key    OPTIONAL);
```

各参数的简要介绍如下所示：

FileHandle：函数 ZwCreateFile 返回的句柄。如果它是一个内核句柄，则 ZwReadFile 和 ZwCreateFile 并不需要在同一个进程中，因为内核句柄是各进程通用的。

Event ：一个事件，用于异步完成读时；我们忽略这个参数。

ApcRoutine Apc：回调例程，用于异步完成读时；我们忽略这个参数。

IoStatusBlock：返回结果状态，与 ZwCreateFile 中的同名参数相同。

Buffer：缓冲区，如果读取文件的内容成功，则内容将被读取到这里。

Length：描述缓冲区的长度，即试图读取文件的长度。

ByteOffset：要读取的文件的偏移量，也就是要读取的内容在文件中的位置。一般来说，不要将其设置为 NULL，文件句柄不一定支持直接读取当前偏移。

Key：读取文件时用的一种附加信息，一般不使用。

当函数执行成功时返回 STATUS_SUCCESS，实际上只要能够读取到任意字节的数据(不管它是否符合参数 Length 的要求)，都返回成功；但是，如果仅读取文件长度之外的部分，则返回 STATUS_END_OF_FILE。

ZwWriteFile 的参数与 ZwReadFile 基本相同，不再进行介绍。

5.4 文件相关的其他操作

除了上述介绍过的一些函数，DDK 还提供了一些函数用以文件操作。

ZwQueryInformationFile、ZwSetInformationFile 可以分别用来获取和设置文件属性，包括文件大小、文件指针位置、文件属性（如只读、隐藏）、文件创建/修改日期等。

这两个函数的参数基本完全相同，只是功能不完全相同而已，因此这里我们仅以 ZwSetInformationFile 函数为例进行介绍，这个函数的原型声明如下所示：

```
NTSTATUS
ZwSetInformationFile(
    IN HANDLE               FileHandle,
    OUT PIO_STATUS_BLOCK    IoStatusBlock,
```

```
    IN PVOID                    FileInformation,
    IN ULONG                    Length,
    IN FILE_INFORMATION_CLASS   FileInformationClass
    );
```

第一、二、四个参数就不介绍了，相信大家都应该能猜到，下面我们仅重点介绍说明 FileInformationClass 这个参数。

FileInformationClass 指定修改或查询的类别，它可能的值有很多种，详情请参考微软在线 MSDN 说明（URL: http://msdn.microsoft.com/en-us/library/ms804363.aspx）。

在内核模式下操作文件的函数不像用户模式下那样丰富，想复制文件就调用 CopyFile、想删除文件就调用 DeleteFile 等，在内核模式下除了读写文件的其他所有操作都是通过这两个 ZwQueryInformation 和 ZwSetInformationFile 函数完成的，而如何使这两个函数精确完成我们需要的功能，就需要通过 FileInformationClass 参数来指定。

5.5 演示程序

在下面的程序中，我们编写了通过文件读写来完成复制的函数，代码如下所示：

```
BOOLEAN
MyCopyFile( IN PUNICODE_STRING      ustrDestFile,
            IN PUNICODE_STRING      ustrSrcFile)
{
    HANDLE   hSrcFile, hDestFile;
    PVOID    buffer = NULL;
    ULONG    length = 0;
    LARGE_INTEGER    offset= {0};
    IO_STATUS_BLOCK  Io_Status_Block = {0};
    OBJECT_ATTRIBUTES    obj_attrib;
    NTSTATUSstatus;
    BOOLEAN      bRet = FALSE;
    do
    {   // 打开源文件
        InitializeObjectAttributes(&obj_attrib,
                                   ustrSrcFile,
                                   OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                                   NULL, NULL);
        status = ZwCreateFile(&hSrcFile,
                              GENERIC_READ,
                              &obj_attrib,
                              &Io_Status_Block,
                              NULL,
                              FILE_ATTRIBUTE_NORMAL,
                              FILE_SHARE_READ,
                              FILE_OPEN,
                              FILE_NON_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT,
                              NULL, 0);
```

```
if (!NT_SUCCESS(status))
{
    bRet = FALSE;
    goto END;
}
// 打开目标文件
InitializeObjectAttributes(&obj_attrib,
                           ustrDestFile,
                           OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                           NULL,
                           NULL);
status = ZwCreateFile(&hDestFile,
                      GENERIC_WRITE,
                      &obj_attrib,
                      &Io_Status_Block,
                      NULL,
                      FILE_ATTRIBUTE_NORMAL,
                      FILE_SHARE_READ,
                      FILE_OPEN_IF,
                      FILE_NON_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT,
                      NULL, 0);
if (!NT_SUCCESS(status))
{
    bRet = FALSE;
    goto END;
}
// 为 buffer 分配 4KB 空间
buffer = ExAllocatePool(NonPagedPool, 1024 * 4);
if (buffer == NULL)
{
    bRet = FALSE;
    goto END;
}
// 复制文件
while (1)
{
    length = 4 * 1024;
    // 读取源文件
    status = ZwReadFile(hSrcFile,
                        NULL, NULL, NULL,
                        &Io_Status_Block,
                        buffer,
                        length,
                        &offset,    NULL);
```

```
            if (!NT_SUCCESS(status))
            {
                // 如果状态为 STATUS_END_OF_FILE，说明文件已经读取到末尾
                if (status == STATUS_END_OF_FILE)
                {
                    bRet = TRUE;
                    goto END;
                }
            }
            // 获得实际读取的长度
            length = Io_Status_Block.Information;
            // 写入到目标文件
            status = ZwWriteFile(hDestFile,
                                 NULL, NULL, NULL,
                                 &Io_Status_Block,
                                 buffer,
                                 length,
                                 &offset,
                                 NULL);
            if (!NT_SUCCESS(status))
            {
                bRet = FALSE;
                goto END;
            }
            // 移动文件指针
            offset.QuadPart += length;
        }
    } while (0);

END:
    if (hSrcFile)
    {
        ZwClose(hSrcFile);
    }
    if (hDestFile)
    {
        ZwClose(hDestFile);
    }
    if (buffer = NULL)
    {
        ExFreePool(buffer);
    }
    return bRet;
}
```

在文件开头添加函数声明后，就可以进行测试了，我们在 DriverEntry 中添加如下代码：

```
RtlInitUnicodeString(&ustrSrcFile, L"\\??\\C:\\windows\\notepad.exe");
RtlInitUnicodeString(&ustrDestFile, L"\\??\\C:\\notepad.exe");

if(MyCopyFile(&ustrDestFile, &ustrSrcFile))
{
    KdPrint(("[Test] CopyFile Success!"));
}
else
{
    KdPrint(("[Test] CopyFile Error!"));
}
```

其中涉及到的两个变量 ustrSrcFile 和 ustrDestFile 需要我们自己在 DriverEntry 的开头进行声明，完整代码在“.\Src\第五章”目录中。

代码测试运行效果如图 5-1 所示：

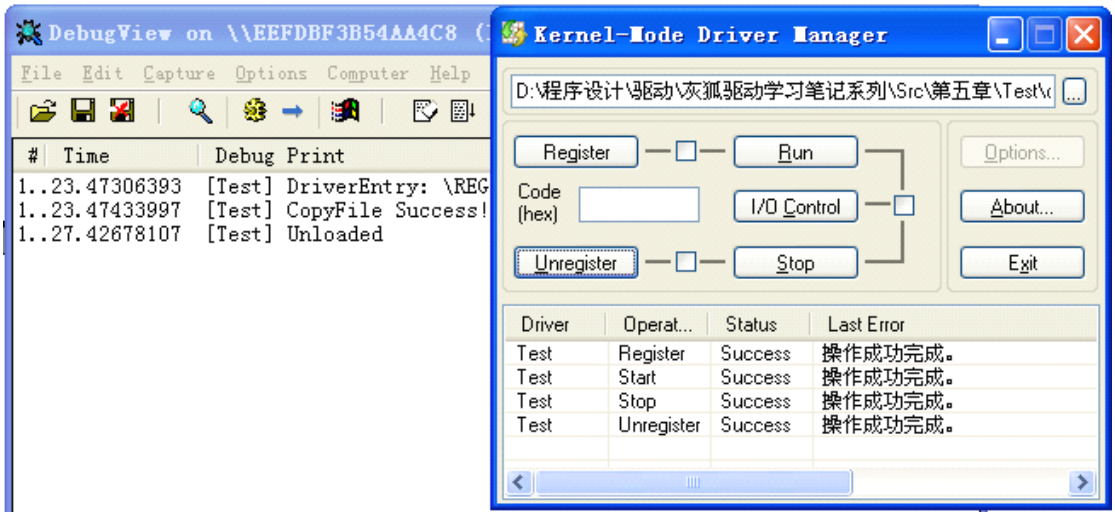

图 5-1 程序演示结果

此时查看我们的 C 盘根目录，就会发现多了一个“notepad.exe”文件，它与 Windows 目录中记事本文件完全相同，由此可见我们的复制文件函数测试成功。

第六章 在驱动中操作注册表

注册表是 Windows 的核心，日常的许多操作其实最终都是转化成了对注册表的操作，我们经常需要利用注册表达到一些特殊的效果，例如实现自启动等。

6.1 创建、打开注册表

和文件操作类似，在操作注册表之前需要首先打开注册表，获得一个句柄，这可以通过函数 ZwCreateKey 完成。

与 ZwCreateFile 函数类似，它通过一个 OBJECT_ATTRIBUTES 获得需要创建或打开的路径信息，但在内核中这个路径与用户模式下不相同，如图 6-1 所示：

应用编程中对应的子键	驱动编程中的路径写法
HKEY_LOCAL_MACHINE	\Registry\Machine
HKEY_USERS	\Registry\User
HKEY_CLASSES_ROOT	没有对应的路径
HKEY_CURRENT_USER	没有简单的对应路径，但是可以求得

图 6-1 注册表中路径的写法

实际上，因为用户模式下的应用程序总是由某个“当前用户”打开的，因此在用户模式下可以直接访问 HKEY_CLASSES_ROOT 和 HKEY_CURRENT_USER，但工作在内核模式下的驱动程序不属于任何一个用户，因此不能直接访问这两个根键。

如果 ZwCreateKey 指定的项不存在，则会直接创建该项，同时由函数的 Disposition 参数返回 REG_CREATED_NEW_KEY；如果指定项已经存在了，则 Disposition 返回值 REG_OPENED_EXISTING_KEY。

DDK 同样提供了一个 ZwOpenKey 函数用以简化打开注册表的操作。同时 DDK 还提供一系列以 Rtl 开头的运行时函数，它们可以是对 Zw 系列函数的封装，可以有效地简化对注册表的操作过程。主要的一些函数列表分别如图 6-2 和图 6-3 所示：

函数	功能
ZwCreateKey	创建或打开指定的注册表项
ZwOpenKey	打开注册表项（ZwCreateKey 的简化版）
ZwSetValueKey	添加或修改指定的键值
ZwQueryKey	查询指定的项
ZwQueryValueKey	查询指定的键值
ZwEnumerateKey	枚举子项
ZwEnumerateValueKey	枚举子键
ZwDeleteKey	删除指定的项

图 6-2 注册表相关 Zw 系列函数

函数	功能
RtlCreateRegistryKey	创建注册表项
RtlCheckRegistryKey	检查指定的注册表项是否存在
RtlWriteRegistryValue	写注册表
RtlQueryRegistryValues	读注册表
RtlDeleteRegistryValue	删除指定的键值

图 6-3 注册表相关 Rtl 系列函数

6.2 读写注册表

注册表是以二元形式存储的，即“键名”和“键值”，通过键名来设置键值，其中键值分为多种情况，如图 6-4 所示：

分类	描述
REG_BINARY	键值用二进制存储
REG_SZ	键值用宽字符串存储，字符串以\0 结尾
REG_EXPAND_SZ	同上，该字符串可扩展
REG_MULTI_SZ	键值存储多个字符串，每个字符串以\0 隔开
REG_DWORD	键值用 4 字节存储
REG_QWORD	键值用 8 字节存储

图 6-4 键值的分类

我们可以通过 ZwSetValueKey 函数添加或修改注册表键值，通过 ZwQueryValueKey 函数查询相关键值。这两个函数的使用与 ring3 没有多大差异，通过 DDK 的帮助文档可以很容易知道它的用法，因此这里不再赘述。

6.3 枚举注册表

枚举注册表是一个非常实用的功能，它通常分两种情况：枚举一个注册表项的所有子项和枚举一个注册表项的所有子键。

枚举子项使用 ZwQueryKey（注意不是 ZwQueryValueKey）和 ZwEnumerateKey 配合完成，枚举子键使用 ZwQueryKey 和 ZwEnumerateValueKey 配合完成。

我们以枚举子项来说明思路，首先利用 ZwQueryKey 获得某项究竟有多少个子项，然后利用 ZwEnumerateKey 来获取指定子项的详细信息，这个过程是通过一个子项索引（index）来完成的。

在使用 ZwQueryKey 时，可以将参数 KeyInformationClass 指定为 KeyFullInformation，它对应 KEY_FULL_INFORMATION 结构中的 SubKeys 指明了该项中有多少子项。

6.4 演示程序

下面的程序演示了如何枚举一个项的所有子项，对于注册表键的读写就不用再费笔墨了，读者可以自行写出测试程序。

```
NTSTATUS
RegEnumTest()
```

```
{
    UNICODE_STRING ustrRegString;
    UNICODE_STRING ustrKeyName;
    HANDLE hRegister;
    ULONG    ulSize, i = 0;
    OBJECT_ATTRIBUTES obj_attrib;
    NTSTATUS status;
    PKEY_FULL_INFORMATION    pfi;
    PKEY_BASIC_INFORMATION pbi;

    // 初始化
     RtlInitUnicodeString(&ustrRegString,L"\\Registry\\Machine\\SOFTWARE\\DarkShadow\\Filter");
    InitializeObjectAttributes(&obj_attrib,
                               &ustrRegString,
                               OBJ_CASE_INSENSITIVE,
                               NULL,
                               NULL);
     // 打开注册表
    status = ZwOpenKey(&hRegister, KEY_ALL_ACCESS, &obj_attrib);
    if (NT_SUCCESS(status))
    {
        KdPrint(("[Test] ZwOpenKey %wZ Success!", ustrRegString));
    }

    // 第一次调用是为了获取需要的长度
    ZwQueryKey(hRegister, KeyFullInformation, NULL, 0, &ulSize);
    pfi = (PKEY_FULL_INFORMATION)ExAllocatePool(PagedPool, ulSize);

    // 第二次调用是为了获取数据
    ZwQueryKey(hRegister, KeyFullInformation, pfi, ulSize, &ulSize);
    for (i = 0; i < pfi->SubKeys; i++)
    {
        // 获取第 i 个子项的长度
        ZwEnumerateKey(hRegister, i, KeyBasicInformation, NULL, 0, &ulSize);
        pbi = (PKEY_BASIC_INFORMATION)ExAllocatePool(PagedPool, ulSize);

        // 获取第 i 个子项的数据
        ZwEnumerateKey(hRegister, i, KeyBasicInformation, pbi, ulSize, &ulSize);
        ustrKeyName.Length = (USHORT)pbi->NameLength;
        ustrKeyName.Buffer = pbi->Name;
        KdPrint(("[Test] The %d SubItem Name : %wZ\n", i, &ustrKeyName));
        // 释放内存
      ExFreePool(pbi);
    }
```

```
    ExFreePool(pfi);
    ZwClose(hRegister);

    return STATUS_SUCCESS;
}
```

上述程序的测试效果如图 6-5 所示：

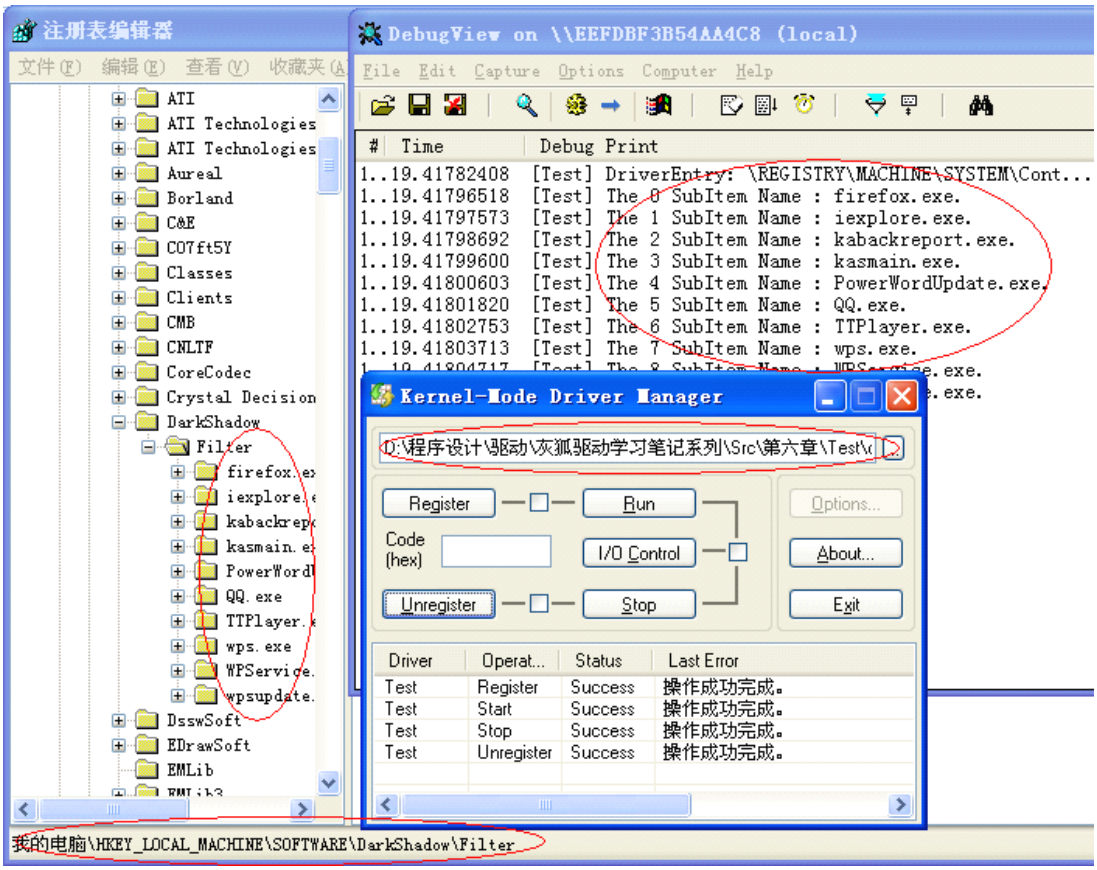

图 6-5 程序演示效果

第七章 在驱动中获取系统时间

在编程中，经常需要获得系统时间，或是需要获得一个从启动开始的毫秒数。前者往往是为了日志记录，后者很适合用来获得一个随机数种子。

7.1 获取启动毫秒数

在 ring3 我们可以通过一个 GetTickCount 函数来获得自系统启动开始的毫秒数，在ring0 也有一个与之对应的 KeQueryTickCount 函数。

不幸的是，这个函数并不能直接返回毫秒数，它返回的是“滴答”数，而一个时钟“滴答”到底是多久，这在不同的系统中可能是不同的，因此我们还需要另外一个函数的辅助，即 KeQueryTimeIncrement 函数。

KeQueryTimeIncrement 函数可以返回一个“滴答”表示多少个 100 纳秒，注意这里的单位是 100 纳秒。

7.2 获取系统时间

在 ring3 获取系统时间是非常简单的，我们直接使用 GetLocalTime 就可以通过一个系统时间结构体 SYSTEMTIME 来返回当前时间。

到了 ring0 我们可以使用 KeQuerySystemTime 来获得当前时间，但它其实是一个格林威治时间，与 ring3 得到的 LocalTime 不同，因此我们还需要使用 ExSystemTimeToLocalTime 函数将这个格林威治时间转换成当地时间。

事情到这里还没有结束，现在我们获得的当地时间不是一个容易阅读的格式，因此我们还要使用 RltTimeToTimeFieldh 函数将其转换成容易阅读的格式。

7.3 演示程序

下面两个程序分别演示了获得启动毫秒数和当前时间的方法，它们被封装在两个函数里面，我们可以很方便地拿来使用。

```
VOID
MyGetTickCount()
{
    LARGE_INTEGER    tick_count;
    ULONG                inc;

    inc = KeQueryTimeIncrement();
    KeQueryTickCount(&tick_count);
    // 因为 1 毫秒等于 1000000 纳秒，而 inc 的单位是 100 纳秒
    // 所以除以 10000 即得到当前毫秒数
    tick_count.QuadPart *= inc;
    tick_count.QuadPart /= 10000;
    KdPrint(("[Test] TickCount : %d", tick_count.QuadPart));
}
```

```
VOID
MyGetCurrentTime()
{
	LARGE_INTEGER	CurrentTime;
	LARGE_INTEGER	LocalTime;
	TIME_FIELDS		TimeFiled;
	static WCHAR		Time_String[32] = {0};

	// 这里得到的其实是格林威治时间
	KeQuerySystemTime(&CurrentTime);
	// 转换成本地时间
	ExSystemTimeToLocalTime(&CurrentTime, &LocalTime);
	// 把时间转换为容易理解的形式
	RtlTimeToTimeFields(&LocalTime, &TimeFiled);

	KdPrint(("[Test] NowTime : %4d-%2d-%2d %2d:%2d:%2d",
				TimeFiled.Year, TimeFiled.Month, TimeFiled.Day,
				TimeFiled.Hour, TimeFiled.Minute, TimeFiled.Second));
}
```

上面两段演示程序的测试效果如图 7-1 所示：

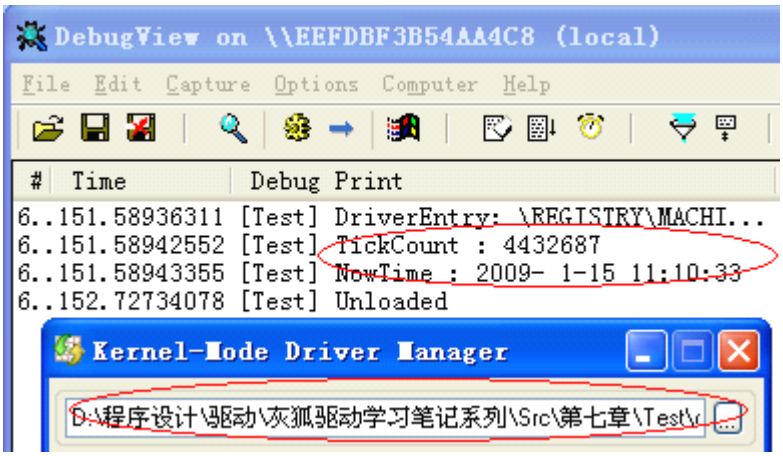

图 7-1 程序演示效果

第八章 在驱动中创建内核线程

线程是一个非常有用的东西，操作系统的最小执行单元就是线程，在内核中线程的概念尤其容易显现出来。

8.1 创建内核线程

在 ring3 我们可以使用 CreateThread 这个 Win32 API 创建线程，在 ring0 也有与之对应的内核函数 PsCreateSystemThread。

这个函数与 CreateThread 的使用很相似，它可以通过第一个参数返回线程的句柄，最后两个参数分别指定线程函数的地址和参数，在 ring3 我们就是这么做的。

我们使用 CreateThread 创建的线程只属于当前进程（不过 CreateRemoteThread 函数可以在指定进程中创建线程），而 PsCreateSystemThread 函数默认情况下创建的却是一个系统进程，它属于进程名为“system”，PID=4 的这个进程。不过 PsCreateSystemThread 也是可以创建用户线程的，这取决于它的第四个参数 ProcessHandle，如果它为空，则创建的即系统线程；如果它是一个进程句柄，则创建的就是属于该指定进程的用户线程。

线程函数是一个非常重要的部分，它决定了该线程具有什么样的功能。线程函数必须按照如下规范声明：

```
VOID ThreadProc(IN PVOID context);
```

这个 VOID 指针参数通过强制转换可以达到很多特殊效果，给予了我们很大的自由度。我们还需要注意的一点，在内核里创建的线程必须自己调用 PsTerminateSystemThread 来结束自身，它不能像 ring3 的线程那样可以在执行完毕后自动结束。

8.2 关于线程同步

提到线程就不能不提到同步的问题，虽然多线程并不是真正的并发运行，但由于 CPU 分配的时间片很短，看起来它们就像是并发运行的一样。

此前我们曾经介绍过自旋锁，它就是一种典型的同步方案，不过在线程同步的时候通常不使用它，而是使用事件通知，此外还有类似 ring3 的临界区、信号灯等方法。

下面我们介绍使用 KEVENT 事件对象进行同步的方法。

在使用 KEVENT 事件对象前，需要首先调用内核函数 KeInitializeEvent 对其初始化，这个函数的原型如下所示：

```
VOID
KeInitializeEvent(
    IN PRKEVENT         Event,
    IN EVENT_TYPE       Type,
    IN BOOLEAN          State);
```

第一个参数 Event 是初始化事件对象的指针；第二个参数 Type 表明事件的类型。事件分两种类型：一类是“通知事件”，对应参数为 NotificationEvent，另一类是“同步事件”，对应参数为 SynchronizationEvent；第三个参数 State 如果为 TRUE，则事件对象的初始化状态为激发状态，否则为未激发状态。

如果创建的事件对象是“通知事件”，当事件对象变为激发态时，需要我们手动将其改回未激发态。如果创建的事件对象是“同步事件”，当事件对象为激发态时，如果遇到相应

的 KeWaitForXXXX 等内核函数，事件对象会自动变回到未激发态。

设置事件的函数是 KeSetEvent，可通过该函数修改事件对象的状态。

8.3 演示程序

在下面的程序中，我演示了如何创建一个系统线程，并在系统线程中输出当前的进程 ID，同时使用了 KEVENT 对创建线程与启动的线程之间进行同步。

```
VOID
CreateThreadTest()
{
	HANDLE		hThread;
	NTSTATUS	status;
	UNICODE_STRING	ustrTest;

	// 初始化
	KeInitializeEvent(&kEvent, SynchronizationEvent, TRUE);
	RtlInitUnicodeString(&ustrTest, L"This is a string for test!");

	// 创建系统线程
	status = PsCreateSystemThread(&hThread, 0, NULL, NULL, NULL, MyThreadFunc,
(PVOID)(&ustrTest));
	if (!NT_SUCCESS(status))
	{
		KdPrint(("[Test] CreateThread Test Failed!"));
	}

	ZwClose(hThread);
	// 等待事件
	KeWaitForSingleObject(&kEvent, Executive, KernelMode, FALSE, 0);
}
// 线程函数
VOID
MyThreadFunc( IN PVOID context )
{
	PUNICODE_STRING str = (PUNICODE_STRING)context;
	KdPrint(("[Test] %d : %wZ", (int)PsGetCurrentProcessId(), str));

	// 设置事件对象
	KeSetEvent(&kEvent, 0, FALSE);
	// 结束线程
	PsTerminateSystemThread(STATUS_SUCCESS);
}
```

第九章 初探 IRP

对 IRP 的处理是驱动开发中很重要的一个部分，本章我们将简单介绍有关 IRP 的概念以及常规操作。

9.1 IRP 的概念

此前我们可能曾经多次听说过 IRP 这个名词，那么它究竟是什么呢？

IRP 的全名是 I/O Request Package，即输入输出请求包，它是 Windows 内核中的一种非常重要的数据结构。上层应用程序与底层驱动程序通信时，应用程序会发出 I/O 请求，操作系统将相应的 I/O 请求转换成相应的 IRP，不同的 IRP 会根据类型被分派到不同的派遣例程中进行处理。

IRP 有两个基本的属性，即 MajorFunction 和 MinorFunction，分别记录 IRP 的主类型和子类型。操作系统根据 MajorFunction 决定将 IRP 分发到哪个派遣例程，然后派遣例程根据 MinorFunction 进行细分处理。

IRP 的概念类似于 Windows 应用程序中“消息”的概念。在 Win32 编程中，程序由“消息”驱动，不同的消息被分发到不同的处理函数中，否则由系统默认处理。

文件 I/O 的相关函数例如 CreateFile、ReadFile、WriteFile、CloseHandle 等分别会引发操作系统产生 IRP_MJ_CREATE、IRP_MJ_READ、IRP_MJ_WRITE、IRP_MJ_CLOSE 等不同的 IRP，这些 IRP 会被传送到驱动程序的相应派遣例程中。

图 9-1 列出了常见的 IRP 类型并给出了简单说明：

IRP 类型	来源说明
IRP_MJ_CREATE	创建设备，CreateFile 会产生此 IRP
IRP_MJ_ CLOSE	关闭设备，CloseHandle 会产生此 IRP
IRP_MJ_CLEANUP	清理，CloseHandle 会产生此 IRP
IRP_MJ_DEVICE_CONTROL	DeviceIoControl 会产生此 IRP
IRP_MJ_PNP	即插即用消息，NT 式驱动不支持此种 IRP，只有 WDM 式驱动才支持
IRP_MJ_POWER	操作系统处理电源消息时产生此 IRP
IRP_MJ_QUERY_INFORMATION	获取文件长度，GetFileSize 会产生此 IRP
IRP_MJ_READ	读取设备内容，ReadFile 会产生此 IRP
IRP_MJ_SET_INFORMATION	设置文件长度，GetFileSize 会产生此 IRP
IRP_MJ_SHUTDOWN	关闭系统前会产生此 IRP
IRP_MJ_SYSTEM_CONTROL	系统控制信息，类似于内核调用 DeviceIoControl
IRP_MJ_WRITE	向设备写入数据，WriteFile 会产生此 IRP

图 9-1 常见 IRP 类型

9.2 IRP 的处理

在第二章我们就介绍过如何在 DriverEntry 中为不同的 IRP 设置相应的派遣例程。在派遣例程中处理 IRP 最简单做法就是将 IRP 的状态设置为成功，然后结束 IRP 请求并返回成功，同时还要记得设置这个 IRP 请求操作了多少字节。

我们在派遣函数中设置 IRP 的完成状态为 STATUS_SUCCESS，发起 I/O 请求的 Win32 API 才能返回 TRUE，否则 Win32 API 将返回 FALSE，在这个时候可以通过 GetLastError 获得错误代码，这个错误代码会和此时 IRP 被设置的状态一致。

下面的代码给出了简单的处理 IRP 例子：

```
NTSTATUS
TestDispatchRoutin(
    IN PDEVICE_OBJECT       DeviceObject,
    IN PIRP                 Irp
)
{
    NTSTATUS status = STATUS_SUCCESS;
    // 设置 IRP 完成状态
    Irp->IoStatus.Status = status;
    // 设置 IRP 操作字节
    Irp->IoStatus.Information = 0;
    // 结束 IRP
    IoCompleteRequest(Irp, IO_NO_INCREMENT);

    return status;
}
```

9.3 IRP 派遣例程示例

在这个程序中，我们首先按照 IRP_MJ_CLOSE 的格式增加一个 IRP_MJ_CLEANUP 的派遣例程，具体请参考附文代码。

然后我们再编写一个应用层的控制台程序，代码如下所示：

```
#include "windows.h"
#include "stdio.h"

int main()
{
    // 打开设备句柄，它会触发 IRP_MJ_CREATE
    HANDLE hDevice = ::CreateFile("\\\\.\\Test",                    // 符号链接
                                  GENERIC_READ | GENERIC_WRITE,
                                  0,
                                  NULL,
                                  OPEN_EXISTING,
                                  FILE_ATTRIBUTE_NORMAL,
                                  NULL);
    if (hDevice == INVALID_HANDLE_VALUE)
    {
        printf("Try to Open Device %s Error : %d!\n", "\\\\.\\Test", ::GetLastError());
        return -1;
    }
```

```
    // 关闭设备句柄，它会触发 IRP_MJ_CLEANUP 和 IRP_MJ_CLOSE
    CloseHandle(hDevice);
    return 0;
}
```

下面我们来介绍应用层程序打开的符号链接。我们查看驱动程序的 DriverEntry，可以看到它调用 IoCreateSymbolicLink 创建了一个符号链接，如下所示：

```
// Test.h
#define TEST_DOS_DEVICE_NAME_W          L"\\DosDevices\\Test"
// Test.c
RtlInitUnicodeString(&dosDeviceName, TEST_DOS_DEVICE_NAME_W);
Status = IoCreateSymbolicLink(&dosDeviceName, &ntDeviceName);
```

从上面可以看出该驱动的符号链接名为“\DosDevices\Test”，也可以写成“\??\Test”，但在编程的时候需要稍微改动一下，写成“\\.\Test”。

现在我们使用 KmdManager 加载驱动并运行，同时运行我们前面编写的应用层驱动程序，根据应用层程序的代码我们可知它应该会触发 IRP_MJ_CREATE、IRP_MJ_CLEANUP、IRP_MJ_CLOSE 这三个 IRP，而我们的验证结果如图 9-2 所示：

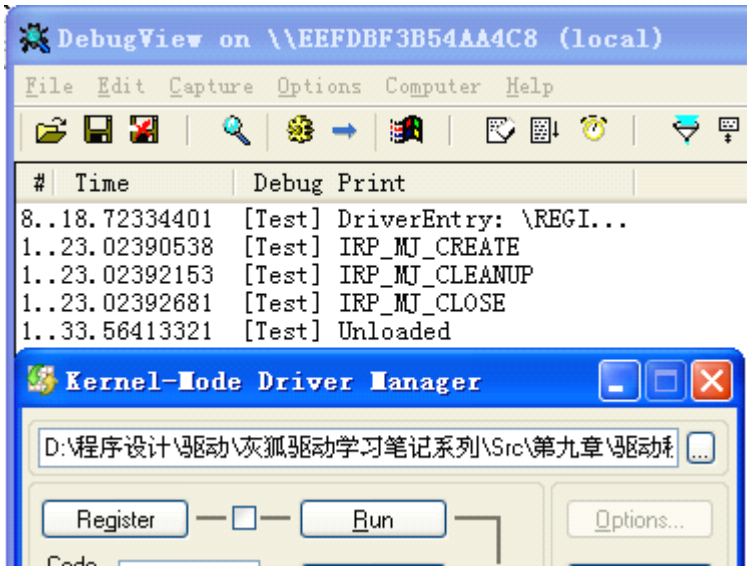

图 9-2 程序演示结果

从图 9-2 中可以看出在应用层程序中调用 CreateFile 等函数确实会产生相应的 IRP，同时我们也理解了如何在应用层打开驱动设备。

第十章 编程加载驱动

在前面几章我们都是使用一个名为 KmdManager 的小工具来加载驱动进行测试的，但在实际应用过程中，我们肯定要做到自己编程加载驱动。我们要知道 NT 式驱动和 WDM 式驱动的加载方法是不大相同的，我们只介绍 NT 式驱动的加载方法。

10.1 有关NT式驱动的加载

Windows NT 式驱动是基于服务方式加载的，可以通过修改注册表内容完成，也可以通过服务相关 API 完成。

设备驱动程序的动态加载主要由服务控制管理程序（Service Control Manager，SCM）系统组件完成，该组件可以启动、停止和控制服务等。

有关服务操作的函数就不多说了，大家可以自行参考其他资料学习。

10.2 服务编程加载驱动

下面给出了两个函数，分别完成加载和卸载服务的功能，它们均来自张帆的《Windows 驱动开发技术详解》一书。

```
BOOL LoadDriver(LPCTSTR lpszDriverName, LPCTSTR lpszDriverPath)
{
    char szDriverFilePath[MAX_PATH];
    GetFullPathName(lpszDriverPath, MAX_PATH, szDriverFilePath, NULL);

    BOOL    bRet = TRUE;
    DWORD dwReturn;
    char    szErrorMsg[MAX_PATH];
    SC_HANDLE hServiceMgr = NULL;
    SC_HANDLE hServiceSys = NULL;

    // 打开服务管理器
    hServiceMgr = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
    if (!hServiceMgr)
    {
        sprintf(szErrorMsg, "OpenSCManager() Error!");
        ::MessageBox(NULL, szErrorMsg, "Error", MB_OK);
        bRet = FALSE;
        goto END;
    }

    // 创建驱动服务
    hServiceSys = ::CreateService(hServiceMgr,
                                  lpszDriverName,           // 驱动服务名
                                  lpszDriverName,           // 驱动服务显示名
```

```
                                    SERVICE_ALL_ACCESS,
                                    SERVICE_KERNEL_DRIVER,      // 内核驱动
                                    SERVICE_DEMAND_START,       // 手动启动
                                    SERVICE_ERROR_IGNORE,       // 忽略错误
                                    szDriverFilePath,           // 服务文件路径
                                    NULL, NULL, NULL, NULL, NULL);
    if (!hServiceSys)
    {
        dwReturn = ::GetLastError();
        if ((dwReturn != ERROR_IO_PENDING) && (dwReturn != ERROR_SERVICE_EXISTS))
        {
            // 服务创建失败，未知错误
            sprintf(szErrorMsg, "CreateService() Error : %d!", dwReturn);
            ::MessageBox(NULL, szErrorMsg, "Error", MB_OK);
            bRet = FALSE;
            goto END;
        }

        // 服务已经存在，只需打开
        hServiceSys = ::OpenService(hServiceMgr, lpszDriverName, SERVICE_ALL_ACCESS);
        if (!hServiceSys)
        {
            sprintf(szErrorMsg, "OpenService() Error!");
            ::MessageBox(NULL, szErrorMsg, "Error", MB_OK);
            bRet = FALSE;
            goto END;
        }
    }

    // 启动服务
    bRet = ::StartService(hServiceSys, NULL, NULL);
    if (!bRet)
    {
        dwReturn = ::GetLastError();
        if ((dwReturn != ERROR_IO_PENDING) &&
(dwReturn != ERROR_SERVICE_ALREADY_RUNNING))
        {
            sprintf(szErrorMsg, "StartService() Error! : %d", dwReturn);
            ::MessageBox(NULL, szErrorMsg, "Error", MB_OK);
            bRet = FALSE;
            goto END;
        }
        else
        {
```

```
                if (dwReturn == ERROR_IO_PENDING)
                {
                    // 设备被挂起
                    sprintf(szErrorMsg, "StartService() Error! : ERROR_IO_PENDING");
                    ::MessageBox(NULL, szErrorMsg, "Error", MB_OK);
                    bRet = FALSE;
                    goto END;
                }
                else
                {
                    // 设备已经运行
                    bRet = TRUE;
                    goto END;
                }
            }
        }

END:
        if (hServiceSys)
        {
            ::CloseHandle(hServiceSys);
        }
        if (hServiceMgr)
        {
            ::CloseHandle(hServiceMgr);
        }
        return bRet;
}
// 卸载驱动
BOOL UnloadDriver(LPCTSTR lpszSvrName)
{
        BOOL bRet = TRUE;
        char szErrorMsg[MAX_PATH];
        SERVICE_STATUS SrvStatus;
        SC_HANDLE hServiceMgr = NULL;
        SC_HANDLE hServiceSys = NULL;

        // 打开服务管理器
        hServiceMgr = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
        if (!hServiceMgr)
        {
            sprintf(szErrorMsg, "OpenSCManager() Error!");
            ::MessageBox(NULL, szErrorMsg, "Error", MB_OK);
            bRet = FALSE;
```

```
            goto END;
        }

        // 打开驱动服务
        hServiceSys = ::OpenService(hServiceMgr, lpszSvrName, SERVICE_ALL_ACCESS);
        if (!hServiceSys)
        {
            sprintf(szErrorMsg, "OpenService() Error!");
            ::MessageBox(NULL, szErrorMsg, "Error", MB_OK);
            bRet = FALSE;
            goto END;
        }

        // 停止驱动服务
        if (!ControlService(hServiceSys, SERVICE_CONTROL_STOP, &SrvStatus))
        {
            sprintf(szErrorMsg, "ControlService() Error!");
            ::MessageBox(NULL, szErrorMsg, "Error", MB_OK);
            bRet = FALSE;
        }

        // 卸载驱动服务
        if (!DeleteService(hServiceSys))
        {
            sprintf(szErrorMsg, "DeleteService() Error!");
            ::MessageBox(NULL, szErrorMsg, "Error", MB_OK);
            bRet = FALSE;
        }

END:
        if (hServiceSys)
        {
            ::CloseHandle(hServiceSys);
        }
        if (hServiceMgr)
        {
            ::CloseHandle(hServiceMgr);
        }
        return bRet;
}
```

这两个函数的调用方法如下所示：

```
char szTitle[10];
char szFileName[MAX_PATH];
char szFilePath[MAX_PATH];
```

```
m_btnOk.GetWindowText(szTitle, sizeof(szTitle));
m_edtFilePath.GetWindowText(szFilePath, sizeof(szFilePath));
m_edtStatus.GetWindowText(szFileName, sizeof(szFileName));

if (strcmp(szTitle, "加载") == 0)
{
    if (LoadDriver(szFileName, szFilePath))
    {
        ::MessageBox(NULL, "加载驱动 SYS 成功！", "提示", MB_OK);
        m_btnOk.SetWindowText("卸载");
    }
}
else
{
    if (UnloadDriver(szFileName))
    {
        ::MessageBox(NULL, "卸载驱动服务成功！", "提示", MB_OK);
        m_btnOk.SetWindowText("加载");
    }
}
```

在这种情况下加载驱动需要调用 OpenSCManager->CreateService->StartService，此时驱动会在服务管理器中注册自己，但这往往显得麻烦，我们还有一个不需要通过服务而直接加载驱动的方法，即使用 ZwLoadDriver（这个函数通常是 ring0 中加载驱动时用，由于它被 ntdll.dll 导出，因此在 ring3 也可以用）进行直接加载。

由于篇幅限制，就不再给出完整代码了，大家可以参考下面这个 URL，有完整代码“http://hi.baidu.com/xlsdg/blog/item/93632681bf1ed4dabc3e1e97.html”。

第十一章 驱动程序与应用层的通信

此前的章节中我们都基本没有考虑过驱动程序与应用层程序之间的通信问题，但这是一个非常重要的内容，否则我们的驱动安装之后就无法被应用程序控制了。

11.1 使用 WriteFile 通信

我们可以在应用层调用 ReadFile 和 WriteFile 分别从驱动中读取和写入数据，他们通过两个不同的 IRP 来传递信息。

前面我们曾经说过，在用户模式下调用 WriteFile 函数会激发 IRP_MJ_WRITE，在这里“http://msdn.microsoft.com/zh-cn/library/ms806163(en-us).aspx”可以查看这个 IRP 的详细说明，这个在线 MSDN 真的是一个非常庞大的知识库。

下面我们就来编写一个通过 WriteFile 向驱动层写入部分数据的演示程序。

首先是我们的应用层程序代码：

```
#include "windows.h"
#include "stdio.h"
int main()
{
	char  szInBuffer[20] = {0};
	DWORD   nLen = 0;

	// 打开设备句柄
	HANDLE hDevice = ::CreateFile("\\\\.\\Test",                    // 符号链接
					GENERIC_READ | GENERIC_WRITE,
					0,
					NULL,
					OPEN_EXISTING,
					FILE_ATTRIBUTE_NORMAL,
					NULL);
	if (hDevice == INVALID_HANDLE_VALUE)
	{		return -1;
	}
	// 向驱动设备写入连续 10 个字节的 A
	memset(szInBuffer, 'A', 10);
	BOOL ret = WriteFile(hDevice,
				szInBuffer,
				10,
				&nLen,
				NULL);
	// 关闭设备句柄
	CloseHandle(hDevice);
	return 0;
}
```

下面开始写驱动层的代码，首先添加一个 IRP_MJ_WRITE 的派遣例程，如下所示：

```
NTSTATUS
TestDispatchWrite(
                    IN PDEVICE_OBJECT       DeviceObject,
                    IN PIRP                 Irp
)
{
    NTSTATUS                Status = STATUS_SUCCESS;
    PIO_STACK_LOCATION      irpStack;

    // 得到当前栈
    irpStack = IoGetCurrentIrpStackLocation(Irp);

    // 输出缓冲区字节数和内容
    DbgPrint("[Test] %d", irpStack->Parameters.Write.Length);
    DbgPrint("[Test] %s", Irp->AssociatedIrp.SystemBuffer);

    // 完成 IRP
    Irp->IoStatus.Information = irpStack->Parameters.Write.Length;
    Irp->IoStatus.Status = Status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);

    return Status;
}
```

至于函数声明，添加派遣例程等等我就不多说了，在完整源码中我有中文注释，现在我们使用 KmdManager 加载该驱动，然后运行前面编写的应用层控制台程序，发现 DbgView 输出的结果很奇怪，首先输出“[Test] 10”，说明接受到了 10 个字节的数据，看来我们的写入数据测试成功，但输出数据内容时却有误，即“[Test] null”。

这是什么原因呢？

仔细查看 MSDN 中关于 IRP_MJ_WRITE 的说明我们可以发现，在使用不同的 I/O 方式时得到的数据是在不同地方的，我加入了一行调试语句输出当前的 flag，发现它既不是缓冲区 I/O 也不是直接 I/O。

没关系，我们在 DriverEntry 中添加一行“deviceObject->Flags |= DO_BUFFERED_IO;”设置一下，现在再次运行程序，终于成功地输出结果，如图 11-1 所示：

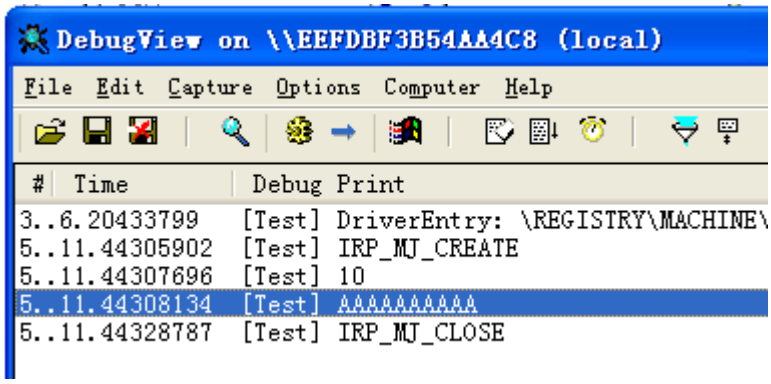

图 11-1 使用 WriteFile 程序演示结果

11.2 使用 DeviceIoControl 通信

使用前面的方法，我们就不得不分别调用 ReadFile 和 WriteFile 来读写数据，实际上我们还有更好更通用的做法，就是使用 DeviceIoControl 函数。这个函数还可以用来做一些除读写之外的操作。

DeviceIoControl 函数会使操作系统产生一个 IRP_MJ_DEVICE_CONTROL 类型的 IRP，然后这个 IRP 会被分发到相应的派遣例程中。

我们先来看一下 DeviceIoControl 函数的原型声明：

```
BOOL DeviceIoControl(
    HANDLE          hDevice,            // handle to device
    DWORD           dwIoControlCode,    // operation
    LPVOID          lpInBuffer,         // input data buffer
    DWORD           nInBufferSize,      // size of input data buffer
    LPVOID          lpOutBuffer,        // output data buffer
    DWORD           nOutBufferSize,     // size of output data buffer
    LPDWORD         lpBytesReturned,    // byte count
    LPOVERLAPPED    lpOverlapped        // overlapped information
);
```

在上面的参数中，我们需要重点掌握的是第二个参数 dwIoControlCode，它是 I/O 控制码，即 IOCTL 值，是一个 32 位的无符号整型数值。

实际上 DeviceIoControl 与 ReadFile 和 WriteFile 相差不大，不过它可以同时提供输入/输出缓冲区，而且还可以通过控制码传递一些特殊信息。

IOCTL 值的定义必须遵循 DDK 的规定，我们可以使用宏 CTL_CODE 来声明，如下：

```
#define MY_DVC_IN_CODE \
        (ULONG)CTL_CODE( FILE_DEVICE_UNKNOWN, \
        0x900, \                    // 自定义 IOCTL 码
        METHOD_BUFFERED, \  // 缓冲区 I/O
        FILE_ALL_ACCESS)
```

关于使用 DeviceIoControl 通信的例子，在楚狂人的驱动教程第 8 章有详细介绍，不幸的是我没有测试成功，因此就暂时不在这里放代码了。

第十二章 HOOK SSDT 的实现

HOOK SSDT 是早年前很常用的一种 rootkit 技术，最重要的是它的实现相对容易，运行稳定，所以被很多人青睐。

12.1 什么是 SSDT

SSDT 的全称是 System Services Descriptor Table，即系统服务描述符表。这个表的作用是把 ring3 的 Win32 API 与 ring0 的内核 API 联系起来。

当然 SSDT 并不仅仅只包含一个庞大的地址索引表，它还包含着一些其它有用的信息，诸如地址索引的基地址、服务函数个数等。

通过修改此表的函数地址可以对常用的 Windows API 进行 HOOK，从而实现对一些比较关心的系统动作进行过滤、监控的目的。一些 HIPS、防毒软件、系统监控、注册表监控软件往往会采用此接口来实现自己的监控模块。

关于 SSDT HOOK 这里有一些好文章，李马的《城里城外看 SSDT》，梧桐的《开篇：驱动扫盲，HOOK SSDT 短文一篇》，galihoo 的《更加稳定的 HOOK SSDT（通过 MDL 方式）》等，当然网络上的资料有很多，你完全可以自己去寻找更好的资料。

12.2 实现 HOOK SSDT

下面的代码基本上来自梧桐的《驱动扫盲，HOOK SSDT 短文一篇》。它是使用 MDL 方式的，同时还有一种通过内嵌汇编修改 CR0 的方式，懒得再打字了，下面代码的原理大家可以到梧桐的 BLOG 学习。

首先在你的工程文件开头添加下列代码：

```
#pragma pack(1)
typedef struct ServiceDescriptorEntry
{
	unsigned int *ServiceTableBase;
	unsigned int *ServiceCounterTableBase; //Used only in checked build
	unsigned int NumberOfServices;
	unsigned char *ParamTableBase;
} SSDTEntry;

__declspec(dllimport) SSDTEntry KeServiceDescriptorTable;
#pragma pack()


// HOOK 函数声明
NTKERNELAPI NTSTATUS ZwTerminateProcess(
	IN HANDLE ProcessHandle OPTIONAL,
	IN NTSTATUS ExitStatus
);
```

```
typedef NTSTATUS(*_ZwTerminateProcess)(
        IN HANDLE ProcessHandle OPTIONAL,
        IN NTSTATUS ExitStatus
);

_ZwTerminateProcess Old_ZwTerminateProcess;

#define GetSystemFunc(FuncName)    \
        KeServiceDescriptorTable.ServiceTableBase[*(PULONG)((PUCHAR)FuncName+1)]
PMDL MDSystemCall;
PVOID *MappedSCT;

#define GetIndex(_Function) *(PULONG)((PUCHAR)_Function+1)

#define HookOn(_Old, _New) \
(PVOID) InterlockedExchange( (PLONG) &MappedSCT[GetIndex(_Old)], (LONG) _New)

#define UnHook(_Old, _New) \
InterlockedExchange( (PLONG) &MappedSCT[GetIndex(_Old)], (LONG) _New)

// 新函数实现
NTSTATUS NewZwTerminateProcess(
        IN HANDLE ProcessHandle OPTIONAL,
        IN NTSTATUS ExitStatus
)
{
        return STATUS_UNSUCCESSFUL;
}
```

然后在 DriverEntry 和 Unload 例程中分别添加开启 HOOK 与关闭 HOOK 的代码，如下所示：

```
// DriverEntry 开启 HOOK
Old_ZwTerminateProcess =(_ZwTerminateProcess)(GetSystemFunc(ZwTerminateProcess));

MDSystemCall = MmCreateMdl(NULL, KeServiceDescriptorTable.ServiceTableBase,   \
                                KeServiceDescriptorTable.NumberOfServices*4);
if(!MDSystemCall)
        return STATUS_UNSUCCESSFUL;
MmBuildMdlForNonPagedPool(MDSystemCall);
MDSystemCall->MdlFlags = MDSystemCall->MdlFlags | MDL_MAPPED_TO_SYSTEM_VA;
MappedSCT = MmMapLockedPages(MDSystemCall, KernelMode);

HookOn( ZwTerminateProcess, NewZwTerminateProcess);
```

```
// 在卸载例程中关闭 HOOK
UnHook( ZwTerminateProcess, Old_ZwTerminateProcess);

if(MDSystemCall)
{
    MmUnmapLockedPages(MappedSCT, MDSystemCall);
    IoFreeMdl(MDSystemCall);
}
```

然后加载上述代码生成的驱动，随便在任务管理器中结束一个进程（例如记事本），测试效果如图 12-1 所示：

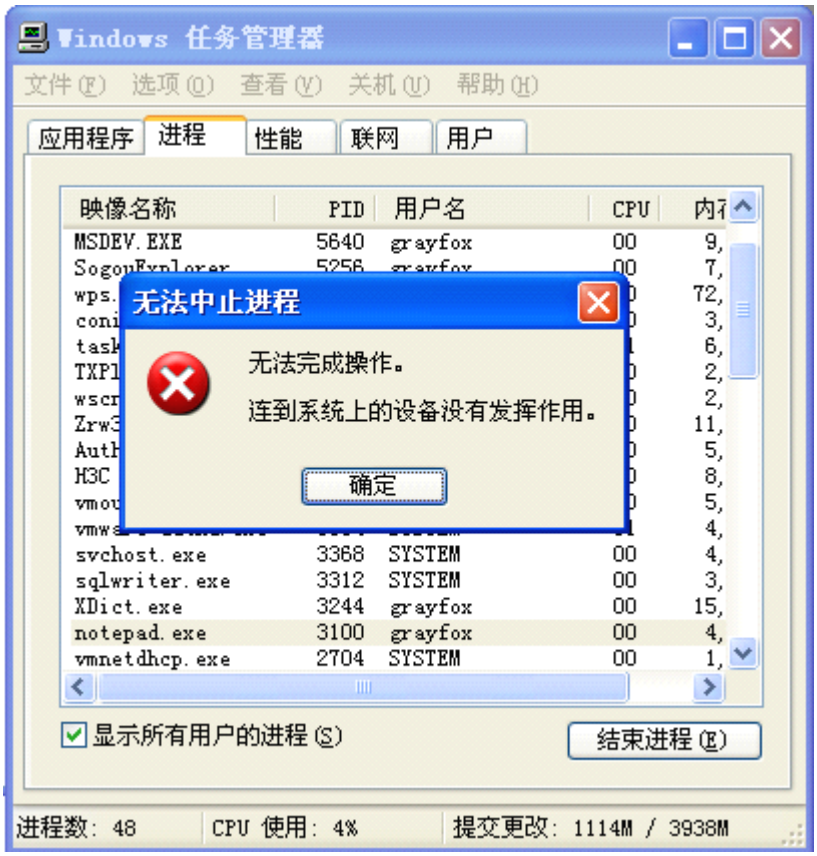

图 12-1 SSDT HOOK 演示效果

最后一篇了，写了这么多东西感到有点累，因此后面这些东西没有认真写，还望各位见谅，希望这个学习笔记能够给大家一点点启发。

实际上，驱动学习并不像我们想象的那样艰难，如果能够坚持学习，到入门之后之后，就会发现其实还是很容易的。