University of Tokyo – Windows Internals II – 26-29 July 2004

Windows Internals II - Projects

Dave Probert, PhD.
Software Architect, Advanced Operating Systems
Core Operating System Division, Microsoft Corporation
Rev 1.1

Overview

The lectures on Windows Internals will include a discussion of three projects that students will undertake as part of the course. There is also a fourth suggested investigation for later this summer.

Project I – Kernel-mode extensions

Summary

Starting with a minimal Windows kernel-mode device driver (called Trivial), consider the changes required to build a kernel extension which uses the registry callbacks.

Registry Callbacks are described in the document in the **TrivialDriver** directory. They allow a kernel extension to monitor the registry APIs (used for behavioral detection of viruses by anti-virus software).

The registry callbacks are available in the system as of XPSP1, and are supported by the Windows Server 2003 DDK ('wnet').

Compare the changes between TrivialDriver and TrivialDriver2. The second driver incorporates registry callbacks and has a applet which periodically reads a list of modified registry values from the kernel.

Read about **PsSetCreateProcessNotifyRoutine** in the DDK.

Modify Trivial2 to remove the registry callbacks and instead list the process ids of each process created.

Then further modify the applet to print out information about the processes as they are created (use the PSAPI functions).

See the details below for more step-by-step help.

Project II – Writing OS subsystems

Operating system code can execute within the application (process) as a library, or in the kernel. It can also execute within a service process, which is isolated from applications but also isolated from other services, increasing the robustness of the system.

University of Tokyo – Windows Internals II – 26-29 July 2004

As with microkernel-based systems, fairly complete operating system functionality can be implemented as a service. NT calls such services *subsystems*.

This project will demonstrate how subsystem operating system services can be built. Starting with a simple subsystem example which prints a message from an application, the project extends it to add open/read/close functionality.

Project III – NTFS and C#

Summary
Write a C# program which reads a raw NTFS volume and dumps out some of the attributes in the Master File Table.

Install the CLR and/or Visual Studio.

You will need a volume on your machine that is NTFS (say C: and administrator access).

File.Open(@"C:\$Volume")

Read bytes off the disk 1024 at a time, looking for records that begin with ANSI "FILE". These are generally entries in the MFT.

Using the information supplied in the details below, print out the file names in each record you find.

You may want to set a limit on how many entries you find and print.

To print out the filenames you have to walk through the list of attributes in each record.
Print out the typecodes and other information about each record.

The format of NTFS is not published, but you should be able to see the basic organization of the NTFS equivalent of the inode in unix.

Project IV – Monad Shell

Summary
In the lecture on Longhorn the Monad Shell was demonstrated. This command line interpreter is based on managed objects from the .NET Framework, and is expected to ultimately replace cmd.exe scripting.

Later this summer the beta of the Monad Shell will be made available on the internet.

The architect of Monad, Jeffrey Snover, is very interested in feedback from UTokyo students, and would be delighted if you download the Monad beta, try it out, and give feedback on what you like and don't like.

University of Tokyo – Windows Internals II – 26-29 July 2004

Project I – kernel-mode extensions

For this project you will need the Windows Device Driver Kit (DDK) installed. It has the tools to build drivers, as well as the documentation for the DDIs (Device Driver Interfaces).

You may also want to download the latest version of the Windows (windbg/ntsd/kd) debugger package from www.microsoft.com.

You may also want to have the SDK documentation, so that you can read about the user-mode APIs – such as those used with the Service Control Manager, which loads drivers.

Step 1 – build and install a trivial device driver.

Find the source code for the sample device driver 'TrivialDriver' under Projects\KernelExtensions.

This driver does nothing other than respond to open/close requests.

Using the DDK (after you install it):

To build TrivialDriver you must have set your environment for the DDK.
set DDK=*C:\WINDDK\37901218* - or to wherever you installed the ddk.
Then type: **%DDK%\bin\setenv.bat chk wnet**

In the TrivialDriver directory type: **build**

Type: **dir /s *.exe** and **dir /s *.sys** and you will find the user-mode application for installing the driver as well as the driver itself. Copy these to the same directory on the machine where you want to test.

Type: **trivialapp.exe**

You should see some output from the program. It will arrange for the Service Control Manager (SCM) to load the driver, and then it will open it. Then it just closes it and removes the driver service from the system. In other words, what it does is 'trivial'.

However what we have accomplished, is that we have a way of running code that we write in kernel-mode.

Of course you require the privilege to load drivers to see this work, so you must run as an administrator.

Step 2 – Add READ and IOCTL functions to TrivialDriver

TrivialDriver2 adds READ and IOCTL functions to TrivialDriver.

It also registers for registry callbacks (see the Registery Callback document), and logs the names of registry settings that have been changed.

Build and run TrivialDriver2.

Step 3 – Convert TrivialDriver2 to log process creates and exits

Read about **PsSetCreateProcessNotifyRoutine** in the DDK.

Remove the registry callback notifications from TrivialDriver2 and instead request CreateProcess notifications and log them in the buffer. Modify the applet (in the exe directory) to print out the parent and child IDs everytime a process is created or exits.

Step 4 – Final step

Modify the applet to print more information about each process created, such as the imagename.

Use the PSAPI functions described in the SDK.

Step 5 – Notes & Elaborations

The applet in this project works by polling the driver. Instead the driver could be modified to block the ioctl until data is available.

A more elaborate scheme might reduce the reliance on limited kernel memory and thus reduce the possibilities of overflow. A user thread could do a read with a large buffer, and the kernel could keep appending the log to the user's buffer instead of keeping data in the kernel and only copying it at each read.

University of Tokyo – Windows Internals II – 26-29 July 2004

Project II – Writing OS subsystems

This project uses the uniformity of objects provided by the NT Object Manager to write operating system functionality as part of a user-mode process.

A library is provided which exports simplified versions of the native NT APIs. It is used by the OS server and App client programs.

The OS server:

creates a process for the App to run in
modifies the Apps virtual memory to pass parameters
starts up a thread to begin execution of the App
listens on an LPC port for service requests
performs the service requests on behalf of the app

Step 1 – Environment

This project can be carried out with the same DDK environment from Project I.

Step 2 – Building the demo subsystem

Compile and execute the code for the OZServer and OZApp programs by typing **build** in their parent directory (Projects\subsystem).

Copy both exes to the BIN\ directory (so OZServer.exe can find OZApp.exe).
Copy the LIB* library files to the BIN\ directory.

Execute the OZServer binary, which will start OZApp, which will execute a printmsg system call.

Step 3 – Add system services for IO to OZServer

Modify OZServer to add three system calls:

```
int fd = open(char *path);  // returns -1 on failure
int read(fd, buffer, len);  // return -1 on error
void close(fd)
```

The open call should allocate a descriptor in the server, use a Win32 API (i.e. CreateFile) to open the path within the server, and return a file descriptor of some sort (e.g. a small integer associated with the allocated descriptor or the address of the descriptor or whatever you decide).

The read call should read into its own buffer (i.e. with ReadFile) and then copy to the client's buffer (using OzWriteChildMemory). Read call returns number of characters read/copied on success.

Close should close the native file and deallocate the descriptor.

Modify the app to issue these three calls on a file that exists on the system.

Step 4 – Notes & Elaborations

The OZServer program can provide OS services that differ greatly from those implemented by the kernel. It must build on the existing kernel APIs, but the native NT APIs in Windows are designed to facilitate building such OS server subsystems.

The object manager facility of the Windows kernel provides a uniform way of accessing kernel objects, including processes, threads, and sections, from any process. The native NT APIs generally take a handle to refer to a object, perform permission checks, and then allow the other process to be manipulated.

A parent/child relationship between the processes is not required.

Future versions of Windows, including Longhorn, will increasingly push system extensions that currently reside in the kernel to instead run as user-mode services. This will include many drivers, thus providing a more robust system environment where a bug in a driver will not crash the entire system.

University of Tokyo – Windows Internals II – 26-29 July 2004

Project III – NTFS and C#

This project combines two learning experiences. W

Step 1 – Get access to a C# compiler

Download the express C# installation of the Visual Studio beta currently available at www.microsoft.com. Or just make sure you have a version of the CLR installed so that you have access to the CSC compiler (but then you won't have all the on-line documentation).

The CLR is installed in %WINDIR%\Microsoft.NET\Framework

On my machine I am using the 2.0beta version:
C:\WINDOWS\Microsoft.NET\Framework\v2.0.40607

which I just put in my PATH to have access to csc.exe, the C# compiler.

Write a HelloWorld program, e.g.

HelloWorld.cs

```
using System;

namespace mine
{
     public static class App
     {
          public static void Main (String[] args)
          {
               Console.WriteLine("Hello World\n");
          }
     }
}
```

Which you can test by simply compiling with **csc HelloWorld.cs**
and then running **HelloWorld.exe**

To get better debugging and make it less noisy, you can use
csc /nologo /debug+ HelloWorld.cs

Step 2 – Write a program that reads an NTFS volume

If you are an administrator, you should be able to open your NTFS volume, say C:, by

```
FileStream vol = File.Open (@"C:\$Volume",
     FileMode.Open, FileAccess.Read, FileShare.ReadWrite);
```

Print out a block that starts with the four ANSI bytes FILE. (Expect to see these starting around offset 0xC000 in the file).

Step 3 – Parse each File Record for the file attributes

The following table should help you tear apart the records and attributes. (you can likely find additional information on the internet).

File record information	Byte offset in file record	Size in bytes
'F', 'I', 'L', 'E'	0x0, 0x1, 0x2, 0x3	4
offset of first attribute	0x14	2
length of valid file record	0x16	4

First invalid attribute begins: 0xffffffff

Attribute information	Byte offset in attribute	Size in bytes
Attribute type code	0x0	4
Form code (resident==0)	0x8	4
Name length	0x9	1
Name offset	0xA	2
Value length	0x10	4
Value offset	0x14	2

Some of the more common attribute type codes

Attribute type	code
$STANDARD_INFORMATION	0x20
$FILE_NAME	0x30
$DATA	0x80

The structure associated with $FILE_NAME 0x42 bytes of other data. The actual filename starts after that.

Step 4 – Notes & Elaborations

There is a obviously a lot more to deciphering the $Mft than encountered in this program.

Tools for deciphering the $Mft can be found on the internet, including the **Nfi** utility described in the book *Inside Microsoft Windows 2000* (page 720).

Finding non-resident attributes, decoding the encoded cluster information, dealing with security, alternate data streams, multi-record Mft entries, and EAs require further decoding. And then the $Index attributes would need to be decoded to understand the file system directory structure.