The Basics

These tuts explain how to write kernel-mode device drivers for Windows NT based operating system line including currently NT4.0, 2000, XP and 2003 in assembly language (using masm compiler). Writing VxD drivers for Windows 95/98/ME is beyond the scope of these tuts. Also, no doubt these tuts are far from the perfection. Probably they contain some inaccuracies. If you found anything incorrect, please feel free to contact me. I'm not native English person. So, translating all that stuff into English was a really hard work for me. I thank masquer and Volodya for proofreading. The English is probably still not perfect, sorry.

1.1 Architecture Overview

1.1.1 Main system components

Windows NT internals are divided by two distinct part concerning both address space and code permissions and responsibilities.

Address space sharing is amazingly simple. Whole four gigabytes of memory available in 32-bit architecture divided by two equal parts (4GT RAM Tuning and Physical Address Extension omitted as an exotic case). The address space for a user-mode processes is mapped into the lower 2GB of linear memory at addresses 00000000 - 7FFFFFFFh. The upper 2GB of linear memory address range 80000000h - 0FFFFFFFFh maps system components such as device drivers, system memory pools, system data structures etc. Sharing permissions and responsibilities is slightly complicated.

Basic types of user-mode processes are described as follows:

- *System Support Processes* - for example, the Logon process (contained in \%SystemRoot%\System32\Winlogon.exe);

- *Service Processes* - for example, the Spooler service (contained in \%SystemRoot%\System32\spoolsv.exe);

- *User Applications* - can be one of five types: Win32, Windows 3.1, MS-DOS, POSIX and OS/2;

- *Environment Subsystems* - Windows ships with three environment subsystems: Win32 (contained in \%SystemRoot%\System32\Csrss.exe); POSIX (contained in \%SystemRoot%\System32\Psxss.exe); OS/2 (contained in \%SystemRoot%\System32\Os2ss.exe).

> In Windows XP and later POSIX and OS/2 subsystems are removed.

The kernel-mode components include the following:

- *Executive* - memory management, process and thread management, security, etc.;
- *Kernel* - thread scheduling, interrupt and exception dispatching, etc. (Executive and Kernel contained in \%SystemRoot%\System32\Ntoskrnl.exe);
- *Device Drivers* - hardware device drivers, file system and network drivers;
- *Hardware Abstraction Layer*, HAL - isolates the kernel, device drivers, and executive from platform-specific hardware differences (contained in \%SystemRoot%\System32\Hal.dll);
- *Windowing And Graphics System* - implements the graphical user interface (GUI) functions such as dealing with windows, user interface controls, and drawing (contained in \%SystemRoot%\System32\Win32k.sys).

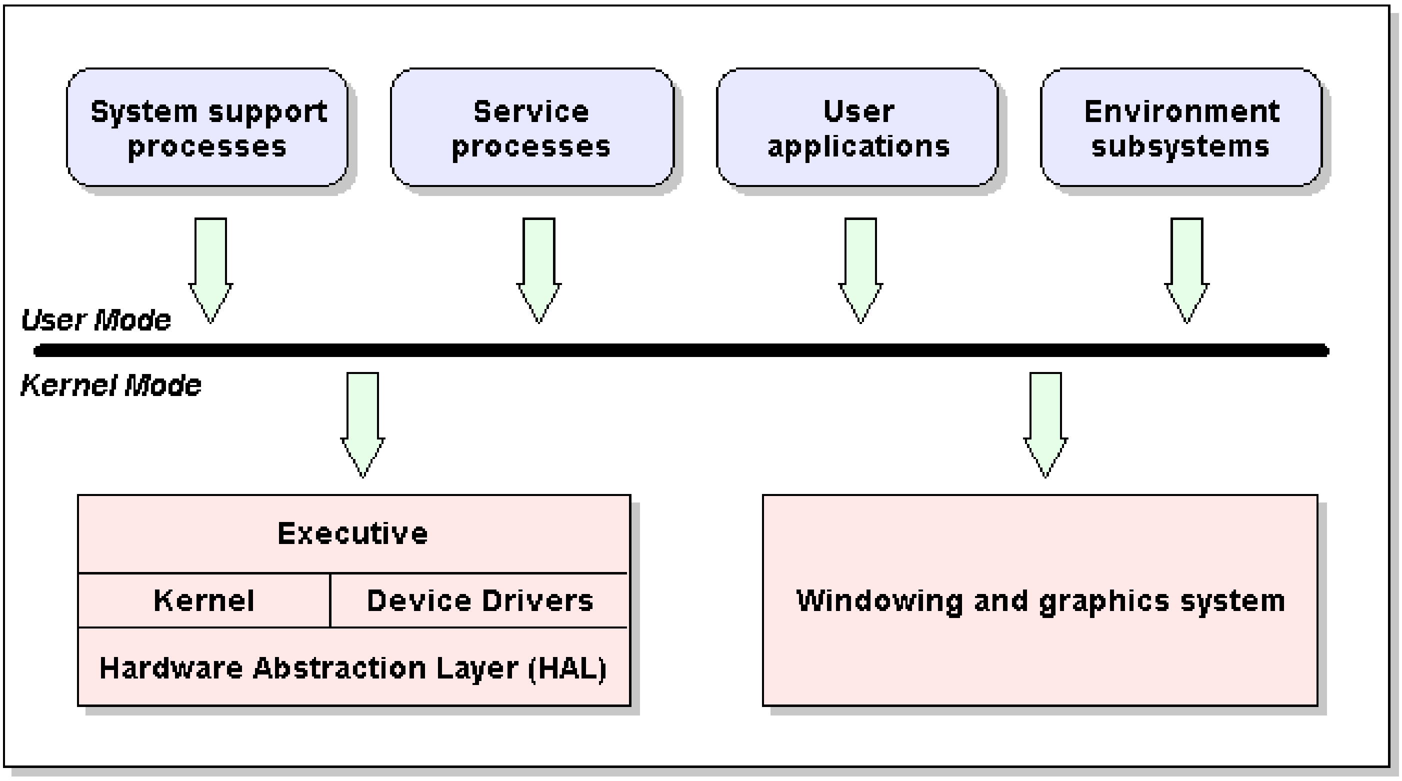

Fig. 1-1 Simplified Windows NT architecture

1.1.2 Kernel Mode vs. User Mode

> The architecture of the Intel x86 processor defines four privilege levels (known as rings). Windows uses privilege level 0 (or ring 0) for kernel-mode and privilege level 3 (or ring 3) for user-mode. The reason Windows uses only two levels is that some of the hardware architectures that were supported in the past (such as Compaq Alpha and Silicon Graphics MIPS) implemented only two privilege levels.

Each user-mode process has its own private memory space. These processes threads run at the least privileged level (called *user-mode* or ring 3) and can not execute privileged CPU instructions, have limited and indirect access to the system data and system address space and have no direct access to the hardware equipment. For example, if a user-mode process touches any address in the upper half of the 4GB address space, the system will terminate it immediately. Note that thread can switch to the kernel-mode calling system services, but in this case it completely lose its execution control until gets back to the user-mode.

The user-mode processes are considered as potentially dangerous in terms of system stability. Their rights are strictly limited. All attempts to violate these restrictions are terminated.

The kernel-mode components sharing common protected kernel-mode memory space, run at a privileged or supervisor level (called *kernel-mode* or ring 0), able to execute any CPU instructions, including privileged ones, have unlimited and unrestricted access to the system data, code and hardware resources.

The code running in the system address space is considered to be completely trusted. Being loaded, the driver becomes a part of the system itself. Therefore the driver is running as a trusted component of the kernel itself, with seemingly unrestricted power to do everything it wants.

Properly speaking the user-mode applications are completely separated from the operating system. It's good for the integrity of the operating system but it could be a headache for some kind of utility application such as debugging tools. Fortunately, unrestricted access provided by the kernel-mode drivers could be used to perform practically impossible tasks on behalf of user-mode applications. So, if you plan to access internal operating system functions or data structures that are not accessible in user-mode, the only way is to load a kernel-mode driver into the system address space. It's rather simple, yet reliable and completely supported by the operating system itself.

1.2 Windows NT Device Drivers

1.2.1 Types of Device Drivers

Windows NT supports a wide range of different device driver types, which can be divided into the following broad categories:

- *User-Mode Drivers*:
    - *Virtual Device Drivers* (VDD) - a user-mode component that are used to emulate 16-bit MS-DOS applications. Although this kind of driver shares a name of VxD used in Windows 95/98, they are completely different.
    - *Printer Drivers* - translate device-independent graphics requests to printer-specific commands.
- *Kernel-Mode Drivers*:
    - *File System Drivers* - implement the standard file system model;
    - *Legacy Drivers* - control a hardware devices without help from other drivers and are written for earlier versions of Windows NT but that run unchanged on Windows 2000/XP/2003;
    - *Video Drivers* - visual data rendering;
    - *Streaming Drivers* - support multimedia devices such as sound cards;
    - *WDM-Drivers* - adhere to the Windows Driver Model (WDM). WDM includes support for Windows NT power management and Plug and Play. WDM is implemented on Windows 2000, Windows 98, and Windows Millennium Edition, so WDM drivers are source-compatible between these operating systems and in many cases are also binary compatible.

In different papers you can meet a slightly different classification, but it really doesn't matter.

It is clear from the name, the device driver is a piece of code intended to control a device. It's not necessary for device to be the physical one. It could be a virtual one as well.

Structurally the device driver is nothing more than the file of the PE-format (Portable Executable, PE). Exactly the same as any other EXE or DLL. Device drivers are loadable kernel-mode modules (generally with "sys" extension). The only difference is that the device drivers are loaded and managed completely other way. Actually, we can consider them as kernel-mode DLL intended to perform the tasks insoluble from the user-mode. The fundamental difference here (not considering the privilege level) is that we can't directly access neither the device driver nor its code, nor its data. The only possible way to do it is through the *I/O Manager*. It provides a primitive driver management environment.

Proceeding to the studying of the kernel-mode device driver's development, you will feel yourself like a newbie, since all your previous experience of programming with Win API will not help you. Even if you have written a considerable amount of the user-mode application development - the kernel offers you a completely different set of functions and structures. You will also have to use poorly documented (defined only in the header files) or totally undocumented functions and data structures.

1.2.2 Layered and Monolithic (single-layered) Device Drivers

The most drivers that control physical devices are *layered drivers*. Layered driver is the one that handles user-mode requests, processing and passing each request from the highest-level driver to the lowest-level driver. An I/O requests handling is distributed between several drivers. For example, if an application issues a read request for a file stored on hard disk, I/O request is passed to the file system driver that would simply rout it to the disk driver, asking it to read data from certain location on the hard disk. It's possible to add any amount of filter-drivers in between (e.g. for the encryption/decryption purposes).

The *monolithic driver* is a simplest type of a device driver. This device driver type generally has no dependencies on other loaded device drivers. They just provide an interface to the user-mode applications without any support from anyone else. Developing and debugging this type of drivers is a quite simple task. This is the type of the device drivers we are about to discuss. The other device driver categories are beyond the scope of these articles.

1.3 Thread context

Since, in most cases, we have only the one CPU, the system creates an illusion of simultaneous applications execution. For all those running applications, the operating system schedules some CPU time for each process's thread. This creates the illusion of all the threads run concurrently by offering time slices to the threads in a round-robin fashion. If the machine has multiple CPUs, the operating system's algorithm is much more complex to keep the balance of the threads over the CPUs. When Windows determine it should select a new thread to run, it performs a context switch. A context switch is the procedure of saving the machine state associated with a running thread, loading another thread's state, and starting its execution. If the selected thread belongs to other process, it is necessary to load the process page directory pointer into CR3 register.

Each user process has private address space, so both different directories of pages and sets of the page tables (CPU uses them to translate the virtual addresses into the physical ones). All that has nothing to do with the driver programming directly. But since context switches waste CPU time, the drivers basically don't create their own threads. Therefore, for saving CPU time on context switching, drivers run in kernel-mode in one of three contexts:

- In the context of the user thread that initiated an I/O request;
- In the context of a kernel-mode system thread;
- As a result of an interrupt (and therefore not in the context of any particular process or thread - whichever process or thread was current when the interrupt occurred).

By processing I/O request packets (IRPs) we are always running in the same process context as the user-mode caller and thus we can directly address its memory space. When the driver is loaded/unloaded we are in the system process context and can only deal with the system memory space.

1.4 Interrupt request levels

The interrupts are integral part of the any operating system. Every interrupt requires processing therefore it diverts the processor to execute outside the normal control flow. There are both software and hardware interrupts available. Interrupts are served in the priority order, and a higher-priority interrupt preempts the serving of a lower-priority interrupt.

Windows imposes interrupt priority scheme known as *interrupt request levels* (IRQLs). The kernel represents IRQLs internally as a number from 0 (passive) through 31 (high), with higher numbers representing higher-priority interrupts. IRQL priority levels have a completely different meaning than thread-scheduling priorities, by the way.

Strictly speaking the interruption with IRQL=0 is not an interrupt, since it can't interrupt flow of any code (this requires lower IRQL, but there is no such level). Every user-mode thread runs at the passive level. The code of our drivers will flow at this IRQL too. It doesn't mean that the code of any driver should always run at the passive level.

Hence two important conclusions follow:

First: The driver's code is always preemptable by any activity that executes at an elevated IRQL, as along with any user-mode thread's code. There are functions allowing to find out the current IRQL, and also to raise or to lower it.

Second: At the passive IRQL it's possible to call any kernel function. (DDK specifies required IRQL for each function to call). Also we can address both nonpaged and paged memory. An attempt to reference paged memory at an elevated IRQL (equal or more than DISPATCH_LEVEL) results to the system crash, since Memory Manager becomes incapable of serving page faults.

1.5 System Crashes

I guess everyone saw at least once the exciting picture known by the name "Blue Screen Of Death", (BSOD). Probably, there is no need to explain, what it is and why it appears sometimes. Prepare yourself to meet BSOD often while developing kernel-mode drivers.

Windows doesn't provide any protection to the private system memory being used by drivers running in the kernel-mode. Once in the kernel-mode the driver has complete access to the system memory space and to all operating system data. So, design and test your drivers carefully to ensure that they don't violate system stability.

You can consider all above-mentioned as the very basic essential principles to understand. It is impossible to start developing the kernel-mode drivers without concepts of the thread context, interrupt request levels, kernel-mode and user-mode are, etc.

1.6 Driver Development Kit

The Windows DDK is a part of the MSDN Professional (and Universal) subscription, but it is also available at http://www.microsoft.com/ddk/). The DDK is an abundant source of Windows NT internals information including the internal system routines and data structures used by device drivers. Unfortunately, Microsoft has stopped free distribution of the DDK. Now you have to order a CD.

Besides the documentation itself, DDK includes a set of Library Files (*.lib), necessary at the linking stage. There are two packages of such files: for free version of Windows (called the free build); and for special debug version (called the checked build). These files are in the directories %ddk%\libfre\i386 and %ddk%\libchk\i386 accordingly. The checked build is a recompilation of the Windows source code with the compile-time flag DEBUG set to TRUE. It performs more strict error checking on kernel-mode functions called by device drivers or other system code. It's necessary to use files according to your Windows version.

1.7 Kernel-Mode Driver Kit for MASM programmers

KmdKit contains all you need to start programming kernel-mode drivers using asm: include files, libraries, macros, examples, tools, these articles, etc. Please, explore the package for details. In the next articles we'll analyze some examples from this package.

1.8 Driver Debugging

As we should debug a kernel-mode code an appropriate debugger is required. SoftICE by Compuware (http://www.compuware.com/products/numega/index.htm) is the best choice. Also you can use Microsoft Kernel Debugger. It requires two computers - a target and a host. The target is the system being debugged, and the host is the system running the debugger. Mark Russinovich (http://www.sysinternals.com/) has written the utility LiveKd that allows the use of the standard Microsoft Kernel Debugger on a live system, without needing two computers.

1.9 Read also

- David Solomon, Mark Russinovich, "Inside Microsoft Windows 2000. Third Edition", Microsoft Press, 2000

  Though there is no source code in this book at all, it's the number one book for the device driver programmers.

- Sven B. Schreiber, "Undocumented Windows 2000 Secrets. A Programming Cookbook", Addison-Wesley

  The especially practical book, it has many Windows 2000 secrets revealed.

- Walter Oney, "Programming the Microsoft Driver Model", Microsoft Press, 1999

- Walter Oney, "Programming the Microsoft Windows Driver Model. 2nd edition", Microsoft Press, 2003

  Excellent book. Its emphasis is Plug'n'Play drivers, but it does not minimize its importance, since the base principles of driver development are universal.

- Art Baker • Jerry Lozano, "The Windows 2000 Device Driver Book, A Guide for Programmers, Second Edition", Prentice Hall, 2000

  Also good book and conforms to our subject.

- Rajeev Nagar, "Windows NT File System Internals. A Developer's Guide", O'Reilly

  The title of this book says all.

- Prasad Dabak, Sandeep Phadke, and Milind Borate, "Undocumented Windows NT", M&T Books, 1999

  You will discover from this book a couple of undocumented things about Windows NT internals.

- Gary Nebbett, " Windows NT-2000 Native API Reference", MacMillan Technical Publishing, 2000

  If you plan to use some undocumented functions and structures in your device drivers this book is for you.

- Jeffrey Richter, "Programming Applications for Microsoft Windows. Fourth Edition", Microsoft Press, 1999

  This book has nothing to do with the device drivers programming, but is very interesting too ;-)

This list doesn't apply for entirety at all. Concerning the books, I still want to say that all of them are from bit "must have".