



[General Reading](#) » [Hardware & System](#) » [System Project Open License \(CPOL\)](#)

License: [The Code](#)

C, Windows, Architect, Dev

Revision: **2** ([See All](#))

Posted: **4 Mar 2009**

Views: **5,144**

Bookmarked: **14 times**

A Primer of the Windows Architecture

By [logicchild](#)

An article to give some insight about the Native API.

2 votes for this article.



Popularity: 0.75 Rating: 2.50 out of 5



Introduction

This paper is meant for those who want to have a stronger understanding of the Windows system architecture. Only a few topics are covered, but a stronger understanding of some of these topics could lead to a stronger understanding of root-kits and their detection.

A Word About the Win32 Subsystem

The Win32 subsystem is the gateway to all user-interfaces in Windows. It is important to know that the components considered in the Win32 subsystem are not responsible for the entire Win32 API, only for the USER and GDI portions of it. The native API is the actual interface to the Windows NT system. In Windows NT, the Win32 API is just a layer above the native API. Because the NT kernel has nothing to do with the GUI, the Native API doesn't include any graphics-related services. In terms of functionality, the Native API is the most direct interface to the Windows kernel, providing interfaces for direct interfacing with the Windows memory manager, I/O system, object manager, processes and their contained threads that execute, and so on. The Win32 subsystem is the component responsible for every aspect of the Windows user interface. This starts with the low-level Graphical Device Interface (GDI), and ends with the USER component, which is responsible for higher-level GUI constructs, such as windows, menus, and for processing input. Application programs are never supposed to directly call into the Native API. This is why Microsoft did not deem it viable to output much documentation about the Native API: application programs are expected to use only the Win32 APIs interacting with the system. Technically, the Native API uses the Win32 API to speak for it, and is a set of functions exported from both *NTDLL.dll* (for user-mode callers) and from *NTSOKRNL.exe* (for kernel-mode callers). For instance, a thread that is executing normally has two thread stacks: one for user mode, and another for kernel mode. So, if a user mode function is called, such as `CreateFile`, it has to transition into kernel mode, wherein after receiving ring 0 kernel mode privileges, it translates to `NtCreateFile`. That is, APIs in the Native API always start out with one or two prefixes: either Nt or Zw. In their user-mode implementation in *NTDLL.dll*, the two groups of APIs are identical and point to the same code. In kernel mode, they differ: the Nt versions are the actual implementation of the APIs, while the Zw versions are stubs that go through the system-call mechanism. In older versions of NT, the Heap API was managed by *NTDLL.dll*, which took the place of *Kernel32.dll*. In some Heap API routines, *kernel32.dll* would "stub" to *NTDLL.dll*. This was called "snapping". When the export of one module snaps to the export of another module, it is called "snapping" to the module.

The System Calling Mechanism

It is almost guaranteed to run into code that invokes system calls into an Operating System API. A system call takes place when user-mode code needs to call a kernel-mode function. This often occurs when an application calls an Operating System API. The user-mode side of the API usually performs basic parameter checks, validation checks, and calls down into the

kernel to actually perform the requested operation. This could define the strength of an Operating System, the stability of which is a function of how well the systems software interacts and communicates with the application software. Note that if user-mode code could call the kernel directly, then security would go out the window. Directly calling a kernel mode function from user mode could create a serious vulnerability because applications could call into an invalid address within the kernel and crash the system, or even call into an address that would allow them to control the system. This is why Operating Systems use a mechanism for switching from user mode to kernel mode. The general idea is that user-mode code invokes a special CPU instruction that tells the processor to switch its privileged mode (for kernel-mode execution) and calls a special service dispatch routine. This dispatch routine then calls the specific system function requested from user mode.

In Windows NT Operating Systems, every system call originated in user-mode which is to be processed by the system's kernel must go through the gate of the kernel itself, where it would be dispatched and executed. This gate is called interrupt `INT 2Eh`. While `ntdll.dll` handles a system call on the user side of the library, after passing the gate, `ntoskrnl.exe` takes over invoking an internal function `KiSystemService()` and passing the original system call to it. The `KiSystemService` uses a lookup table for the information on which Native API call should be used to assign the original call to. This system service table is called SST. Consider an example of a typical Windows 2000 system call; in WinXP, `NtReadFile/ZwReadFile` in `ntdll.dll` disassembles to:

```
Ntdll!ZwReadFile:
77f8c552  mov  eax, 0x1
77f8c557  lea  edx, [esp+4]
77f8c55b  int  2e
77f8c55d  ret  0x24
```

The `EAX` register is loaded with the service number, and the `EDX` register points to the first parameter that the kernel mode function receives. This is why we add 32 bits to the stack pointer. When the `int 2e` instruction is invoked, the processor uses the Interrupt Descriptor Table (IDT) in order to determine which handler to call. The IDT (which used to be called the Interrupt Vector Table) is a processor-owned table that tells the processor which routine to invoke whenever an interrupt or exception takes place. The handler number could typically be a hexadecimal multiple of 4 times the interrupt number. The IDT entry for interrupt `2e` points to an internal `NTOSKRNL` function called `KiSystemService`, which is the kernel-service dispatcher. `KiSystemService` verifies that the service number and the stack pointer are valid, and calls into the specific kernel function requested. Now, let's consider the disassembly of `NtWriteFile/ZwWriteFile`:

```
MOV EAX, 112h
MOV EDX, 7FFE0300h
CALL DWORD PTR DS:[EDX]
RETN 24
```

Note that the value `7FFE0300h` is first moved into `EDX`. So, at `7FFE0300h` is a pointer to the following function:

```
MOV EDX, ESP
SYSENTER
```

Note that instead of invoking an interrupt in order to perform the switch to kernel mode, the system now uses the special `SYSENTER` instruction in order to perform the switch.

Exception Handling

An exception is a special condition in a program that makes it immediately jump to a special function called an exception handler. The exception handler then decides how to deal with the exception, and can either correct the problem or make the program continue from the

same code position or resume execution from another position. Interrupts and exceptions, then, are Operating System conditions that divert the processor to code outside the normal flow of control. Either hardware or software can detect them. The term trap refers to a processor's mechanism for capturing an executing thread when an exception or interrupt occurs and control is transferred to a fixed location in the Operating System. In Windows, the processor transfers control to a trap handler, a function specific to a particular interrupt or exception. There are two basic types of exceptions: hardware and software. Hardware exceptions are exceptions generated by the processor: for example, when a program accesses an invalid memory page (a page fault), or when a division by zero occurs.

The kernel distinguishes between interrupts and exceptions in the following way. An interrupt is an asynchronous event (one that can occur at any time) that is unrelated to what the processor is executing. Interrupts are primarily generated by I/O devices, processor clocks, or timers, and can be enabled or disabled by the change of a bit value. An exception is a synchronous condition that results from the execution of a particular instruction. Thus, either hardware or software can generate exceptions and interrupts. For example, a bus error exception is caused by a hardware problem. If you ever take your system to a technician, see if he can check for hardware problems related to the PCI Bus and then perform tests to check PCI Bus Compliance. With IRQ/DMA, have your technician check the software IRQs. This type of functionality can report on the interrupts currently loaded in memory from 0 – FFh. You can use this information to find out which interrupts are attached via the system BIOS and the addresses each interrupt points to. This data can be useful in resolving conflicts between the BIOS and software drivers. At the same time, try mapping a system thread to a device driver. Do a directory listing of your entire C: drive using a network path to access your C: drive. For example, if your computer's name is STATION1, type "dir \\system1\c\$ /s". Now, fire up Process Explorer (a powerful Process utility written by Mark Russinovich of Sysinternals at Technet). Ensure that you have selected CPU Context Switch Delta so it is part of your columns. Double-click threads on the System tab while the "dir" command is running. Look for *srv.sys!WorkerThread* and click for its properties. If you see a system thread running and you are not sure what the driver is, press the Module button while high-lighting the thread in *Srv.sys* shown in the display.

References

- Windows Internals, written by Mark Russinovich and David Solomon.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author

logicchild

Member


I started electronics training at age 33. I began studying microprocessor technology in an RF communications oriented program. I am 43 years old now. I have studied C code, opcode (mainly x86 and AT+T) for around 3 years in order to learn how to recognize viral code and the use of procedural languages. I am currently learning C# and the other virtual runtime system languages. I guess I started with the egg rather than the chicken. My past work would indicate that my primary strength is in applied mathematics.

Occupation: Other

Company: Pref. Trust

Location:  United States

Discussions and Feedback

 **2 messages** have been posted for this article. Visit <http://www.codeproject.com/KB/system/Win32.aspx> to post and view comments on this article, or click [here](#) to get a print view with messages.

[PermaLink](#) | [Privacy](#) | [Terms of Use](#)
Last Updated: 4 Mar 2009
Editor: [Smitha Vijayan](#)

Copyright 2009 by logicchild
Everything else Copyright © [CodeProject](#), 1999-2010
Web16 | [Advertise on the Code Project](#)