

Windows Kernel Programming

Second Edition

**Book
Authority**

One of the best
kernel books of
all time

Pavel Yosifovich

Windows Kernel Programming, Second Edition

Pavel Yosifovich

This book is for sale at

<http://leanpub.com/windowskernelprogrammingsecondedition>

This version was published on 2022-01-22



* * * * *

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

© 2020 - 2022 Pavel Yosifovich

Table of Contents

[Introduction](#)

[Who Should Read This Book](#)

[What You Should Know to Use This Book](#)

[Book Contents](#)

[Sample Code](#)

[Chapter 1: Windows Internals Overview](#)

[Processes](#)

[Virtual Memory](#)

[Page States](#)

[System Memory](#)

[Threads](#)

[Thread Stacks](#)

[System Services \(a.k.a. System Calls\)](#)

[General System Architecture](#)

[Handles and Objects](#)

[Object Names](#)

[Accessing Existing Objects](#)

[Chapter 2: Getting Started with Kernel Development](#)

[Installing the Tools](#)

[Creating a Driver Project](#)

[The DriverEntry and Unload Routines](#)

[Deploying the Driver](#)

[Simple Tracing](#)

[Summary](#)

[Chapter 3: Kernel Programming Basics](#)

[General Kernel Programming Guidelines](#)

[Unhandled Exceptions](#)

[Termination](#)

[Function Return Values](#)

[IRQL](#)

[C++ Usage](#)

[Testing and Debugging](#)

[Debug vs. Release Builds](#)

[The Kernel API](#)
[Functions and Error Codes](#)
[Strings](#)
[Dynamic Memory Allocation](#)
[Linked Lists](#)
[The Driver Object](#)
[Object Attributes](#)
[Device Objects](#)
[Opening Devices Directly](#)
[Summary](#)

[Chapter 4: Driver from Start to Finish](#)

[Introduction](#)
[Driver Initialization](#)
[Passing Information to the Driver](#)
[Client / Driver Communication Protocol](#)
[Creating the Device Object](#)
[Client Code](#)
[The Create and Close Dispatch Routines](#)
[The Write Dispatch Routine](#)
[Installing and Testing](#)
[Summary](#)

[Chapter 5: Debugging and Tracing](#)

[Debugging Tools for Windows](#)
[Introduction to WinDbg](#)
[Tutorial: User mode debugging basics](#)
[Kernel Debugging](#)
[Local Kernel Debugging](#)
[Local kernel Debugging Tutorial](#)
[Full Kernel Debugging](#)
[Using a Virtual Serial Port](#)
[Using the Network](#)
[Kernel Driver Debugging Tutorial](#)
[Asserts and Tracing](#)
[Asserts](#)
[Extended DbgPrint](#)
[Other Debugging Functions](#)
[Trace Logging](#)
[Viewing ETW Traces](#)

[Summary](#)

[Chapter 6: Kernel Mechanisms](#)

[Interrupt Request Level \(IRQL\)](#)

[Raising and Lowering IRQL](#)

[Thread Priorities vs. IRQLs](#)

[Deferred Procedure Calls](#)

[Using DPC with a Timer](#)

[Asynchronous Procedure Calls](#)

[Critical Regions and Guarded Regions](#)

[Structured Exception Handling](#)

[Using `\_\_try/ \_\_except`](#)

[Using `\_\_try/ \_\_finally`](#)

[Using C++ RAII Instead of `\_\_try/ \_\_finally`](#)

[System Crash](#)

[Crash Dump Information](#)

[Analyzing a Dump File](#)

[System Hang](#)

[Thread Synchronization](#)

[Interlocked Operations](#)

[Dispatcher Objects](#)

[Mutex](#)

[Fast Mutex](#)

[Semaphore](#)

[Event](#)

[Named Events](#)

[Executive Resource](#)

[High IRQL Synchronization](#)

[The Spin Lock](#)

[Queued Spin Locks](#)

[Work Items](#)

[Summary](#)

[Chapter 7: The I/O Request Packet](#)

[Introduction to IRPs](#)

[Device Nodes](#)

[IRP Flow](#)

[IRP and I/O Stack Location](#)

[Viewing IRP Information](#)

[Dispatch Routines](#)

[Completing a Request](#)

[Accessing User Buffers](#)

[Buffered I/O](#)

[Direct I/O](#)

[User Buffers for IRP_MJ_DEVICE_CONTROL](#)

[Putting it All Together: The Zero Driver](#)

[Using a Precompiled Header](#)

[The DriverEntry Routine](#)

[The Create and Close Dispatch Routines](#)

[The Read Dispatch Routine](#)

[The Write Dispatch Routine](#)

[Test Application](#)

[Read/Write Statistics](#)

[Summary](#)

[Chapter 8: Advanced Programming Techniques \(Part 1\)](#)

[Driver Created Threads](#)

[Memory Management](#)

[Pool Allocations](#)

[Secure Pools](#)

[Overloading the new and delete Operators](#)

[Lookaside Lists](#)

[The “Classic” Lookaside API](#)

[The Newer Lookaside API](#)

[Calling Other Drivers](#)

[Putting it All Together: The Melody Driver](#)

[Client Code](#)

[Invoking System Services](#)

[Example: Enumerating Processes](#)

[Summary](#)

[Chapter 9: Process and Thread Notifications](#)

[Process Notifications](#)

[Implementing Process Notifications](#)

[The DriverEntry Routine](#)

[Handling Process Exit Notifications](#)

[Handling Process Create Notifications](#)

[Providing Data to User Mode](#)

[The User Mode Client](#)

[Thread Notifications](#)

[Image Load Notifications](#)

[Final Client Code](#)

[Remote Thread Detection](#)

[The Detector Client](#)

Summary.

Introduction

Windows kernel programming is considered by many a dark art, available to select few that manage to somehow unlock the mysteries of the Windows kernel. Kernel development, however, is no different than user-mode development, at least in general terms. In both cases, a good understanding of the platform is essential for producing high quality code.

The book is a guide to programming within the Windows kernel, using the well-known Visual Studio integrated development environment (IDE). This environment is familiar to many developers in the Microsoft space, so that the learning curve is restricted to kernel understanding, coding and debugging, with less friction from the development tools.

The book targets software device drivers, a term I use to refer to drivers that do not deal with hardware. Software kernel drivers have full access to the kernel, allowing these to perform any operation allowed by the kernel. Some software drivers are more specific, such as file system mini filters, also described in the book.

Who Should Read This Book

The book is intended for software developers that target the Windows kernel, and need to write kernel drivers to achieve their goals. Common scenarios where kernel drivers are employed are in the Cyber Security space, where kernel drivers are the chief mechanism to get notified of important events, with the power to intercept certain operations. The book uses C and C++ for code examples, as the kernel API is all C. C++ is used where it makes sense, where its advantages are obvious in terms of maintenance, clarity, resource management, or any combination of these. The book does not use complex C++ constructs, such as template metaprogramming. The book is not about C++, it's about Windows kernel drivers.

What You Should Know to Use This Book

Readers should be very comfortable with the C programming language, especially with pointers, structures, and its standard library, as these occur very frequently when working with kernel APIs. Basic C++ knowledge is highly recommended, although it is possible to traverse the book with C proficiency only.

Book Contents

Here is a quick rundown of the chapters in the book:

- Chapter 1 (“Windows Internals Overview”) provides the fundamentals of the internal workings of the Windows OS at a high level, enough to get the fundamentals without being bogged down by too many details.
- Chapter 2 (“Getting Started with Kernel Development”) describes the tools and procedures needed to set up a development environment for developing kernel drivers. A very simple driver is created to make sure all the tools and procedures are working correctly.
- Chapter 3 (“Kernel Programming Basics”) looks at the fundamentals of writing drivers, including basic kernel APIs, handling of common programming tasks involving strings, linked lists, dynamic memory allocations, and more.
- Chapter 4 (“Driver from Start to Finish”) shows how to build a complete driver that performs some useful functionality, along with a client application to drive it.

If you are new to Windows kernel development, you should read chapters 1 to 7 in order. Chapter 8 contains some advanced material you may want to go back to after you have built a few simple drivers. Chapters 9 onward describe specialized techniques, and in theory at least, can be read in any order.

Sample Code

All the sample code from the book is freely available on the book’s Github repository at

<https://github.com/zodiacon/windowskernelprogrammingbook2e>. Updates to the code samples will be pushed to this repository. It’s recommended the reader clone the repository to the local machine, so it’s easy to experiment with the code directly.

All code samples have been compiled with Visual Studio 2019. It's possible to compile most code samples with earlier versions of Visual Studio if desired. There might be few features of the latest C++ standards that may not be supported in earlier versions, but these should be easy to fix.

Happy reading!

Pavel Yosifovich

June 2022

Chapter 1: Windows Internals Overview

This chapter describes the most important concepts in the internal workings of Windows. Some of the topics will be described in greater detail later in the book, where it's closely related to the topic at hand. Make sure you understand the concepts in this chapter, as these make the foundations upon any driver and even user mode low-level code, is built.

In this chapter:

- **Processes**
 - **Virtual Memory**
 - **Threads**
 - **System Services**
 - **System Architecture**
 - **Handles and Objects**
-

Processes

A process is a containment and management object that represents a running instance of a program. The term “process runs” which is used fairly often, is inaccurate. Processes don't run – processes manage. Threads are the ones that execute code and technically run. From a high-level perspective, a process owns the following:

- An executable program, which contains the initial code and data used to execute code within the process. This is true for most processes, but some special ones don't have an executable image (created directly by the kernel).
- A private virtual address space, used for allocating memory for whatever purposes the code within the process needs it.

- An access token (called *primary token*), which is an object that stores the security context of the process, used by threads executing in the process (unless a thread assumes a different token by using *impersonation*).
- A private handle table to executive objects, such as events, semaphores, and files.
- One or more threads of execution. A normal user-mode process is created with one thread (executing the classic `main/winMain` function). A user mode process without threads is mostly useless, and under normal circumstances will be destroyed by the kernel.

These elements of a process are depicted in figure 1-1.

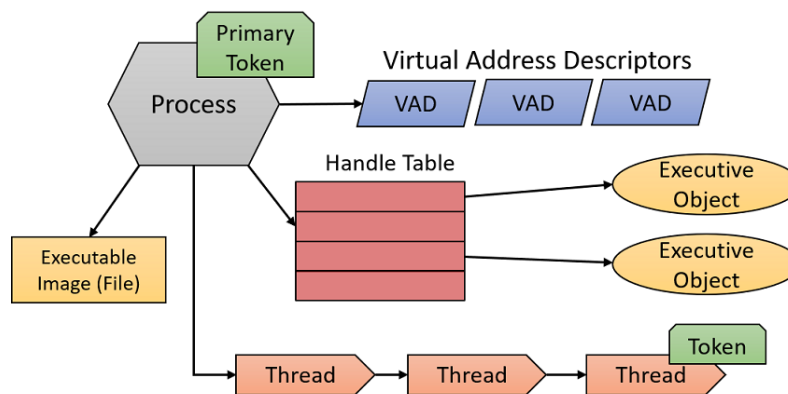


Figure 1-1: Important ingredients of a process

A process is uniquely identified by its Process ID, which remains unique as long as the kernel process object exists. Once it's destroyed, the same ID may be reused for new processes. It's important to realize that the executable file itself is not a unique identifier of a process. For example, there may be five instances of *notepad.exe* running at the same time. Each of these *Notepad* instances has its own address space, threads, handle table, process ID, etc. All those five processes are using the same image file (*notepad.exe*) as their initial code and data. Figure 1-2 shows a screenshot of *Task Manager*'s Details tab showing five instances of *Notepad.exe*, each with its own attributes.

The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. It displays a list of running processes. Five instances of 'notepad.exe' are visible, each with a unique PID and running under the 'Pavel' user. Other processes include 'MsSense.exe', 'NisSrv.exe', 'nvapiw.exe', 'nvcontainer.exe', 'NVDIContainer.exe', and 'NVIDIA Share.exe'.

Name	PID	Status	User name	Sessi...	CPU	Memory (active...	Commit size	Base priority	Handles	Threads
MsSense.exe	7120	Running	SYSTEM	0	00	258,648 K	864,744 K	Normal	1,170	39
NisSrv.exe	1616	Running	LOCAL SE...	0	00	43,428 K	59,920 K	Normal	359	13
notepad.exe	29468	Running	Pavel	1	00	1,736 K	3,076 K	Normal	244	7
notepad.exe	14504	Running	Pavel	1	00	14,416 K	33,324 K	Normal	765	25
notepad.exe	30076	Running	Pavel	1	00	1,744 K	3,072 K	Normal	245	7
notepad.exe	11396	Running	Pavel	1	00	1,732 K	3,080 K	Normal	243	7
notepad.exe	38700	Running	Pavel	1	00	1,732 K	3,076 K	Normal	243	7
nvapiw.exe	22532	Running	Pavel	1	00	1,136 K	4,124 K	Normal	136	2
nvcontainer.exe	6832	Running	SYSTEM	0	00	7,144 K	13,772 K	Normal	701	35
nvcontainer.exe	1592	Running	Pavel	1	00	67,440 K	83,188 K	Normal	523	24
nvcontainer.exe	12632	Running	Pavel	1	00	3,420 K	8,816 K	Normal	368	16
NVDIContainer.exe	5072	Running	SYSTEM	1	00	26,140 K	46,136 K	Normal	840	39
NVDIContainer.exe	4204	Running	SYSTEM	0	00	3,732 K	6,156 K	Normal	351	8
NVIDIA Share.exe	29152	Running	Pavel	1	00	18,376 K	31,820 K	Normal	788	31

Figure 1-2: Five instances of notepad

Virtual Memory

Every process has its own virtual, private, linear address space. This address space starts out empty (or close to empty, since the executable image and `Ntdll.dll` are the first to be mapped, followed by more subsystem DLLs). Once execution of the main (first) thread begins, memory is likely to be allocated, more DLLs loaded, etc. This address space is private, which means other processes cannot access it directly. The address space range starts at zero (technically the first and last 64KB of the address space cannot be committed), and goes all the way to a maximum which depends on the process “bitness” (32 or 64 bit) and the operating system “bitness” as follows:

- For 32-bit processes on 32-bit Windows systems, the process address space size is 2 GB by default.
- For 32-bit processes on 32-bit Windows systems that use the *increase user virtual address space* setting, can be configured to have up to 3GB of address space per process. To get the extended address space, the executable from which the process was created must have been marked with the `LARGEADDRESSAWARE` linker flag in its PE header. If it was not, it would still be limited to 2 GB.
- For 64-bit processes (on a 64-bit Windows system, naturally), the address space size is 8 TB (Windows 8 and earlier) or 128 TB (Windows 8.1 and later).
- For 32-bit processes on a 64-bit Windows system, the address space size is 4 GB if the executable image has the `LARGEADDRESSAWARE` flag in its PE header. Otherwise, the size remains at 2 GB.



The requirement of the `LARGEADDRESSAWARE` flag stems from the fact that a 2 GB address range requires 31 bits only, leaving the most significant bit (MSB) free for application use. Specifying this flag indicates that the program is not using bit 31 for anything and so having that bit set (which would happen for addresses larger than 2 GB) is not an issue.

Each process has its own address space, which makes any process address relative, rather than absolute. For example, when trying to determine what lies in address `0x20000`, the address itself is not enough; the process to which this address relates to must be specified.

The memory itself is called *virtual*, which means there is an indirect relationship between an address and the exact location where it's found in physical memory (RAM). A buffer within a process may be mapped to physical memory, or it may temporarily reside in a file (such as a page file). The term *virtual* refers to the fact that from an execution perspective, there is no need to know if the memory about to be accessed is in RAM or not; if the memory is indeed mapped to RAM, the CPU will perform the virtual-to-physical translation before accessing the data. If the memory is not resident (specified by a flag in the translation table entry), the CPU will raise a page fault exception that causes the memory manager's page fault handler to fetch the data from the appropriate file (if indeed it's a valid page fault), copy it to RAM, make the required changes in the page table entries that map the buffer, and instruct the CPU to try again. Figure 1-3 shows this conceptual mapping from virtual to physical memory for two processes.

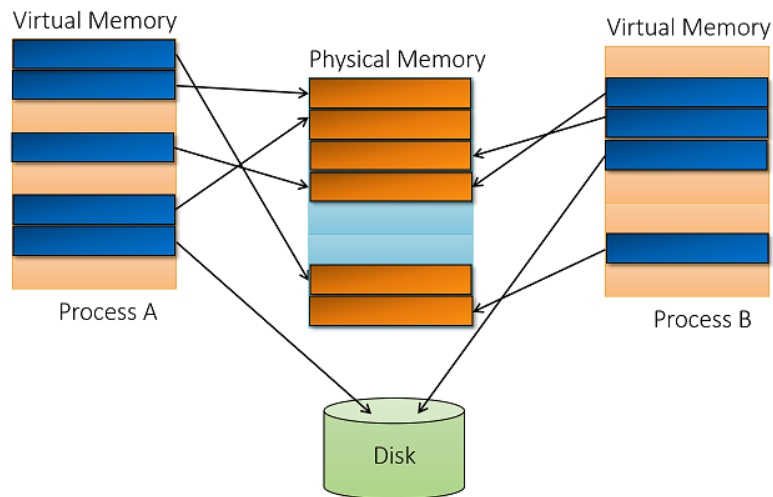


Figure 1-3: virtual memory mapping

The unit of memory management is called a *page*. Every attribute related to memory is always at a page's granularity, such as its protection or state. The size of a page is determined by CPU type (and on some processors, may be configurable), and in any case, the memory manager must follow suit. Normal (sometimes called small) page size is 4 KB on all Windows-supported architectures.

Apart from the normal (small) page size, Windows also supports large pages. The size of a large page is 2 MB (x86/x64/ARM64) or 4 MB (ARM). This is based on using the Page Directory Entry (PDE) to map the large page without using a page table. This results in quicker translation, but most importantly better use of the *Translation Lookaside Buffer* (TLB) – a cache of recently translated pages maintained by the CPU. In the case of a large page, a single TLB entry maps significantly more memory than a small page.



The downside of large pages is the need to have the memory contiguous in RAM, which can fail if memory is tight or very fragmented. Also, large pages are always non-pageable and can only use read/write protection.

Huge pages of 1 GB in size are supported on Windows 10 and Server 2016 and later. These are used automatically with large pages if an allocation is at least 1 GB in size, and that size can be located as contiguous in RAM.

Page States

Each page in virtual memory can be in one of three states:

- Free – the page is not allocated in any way; there is nothing there. Any attempt to access that page would cause an access violation exception. Most pages in a newly created process are free.
- Committed – the reverse of free; an allocated page that can be accessed successfully (assuming non-conflicting protection attributes; for example, writing to a read-only page causes an access violation). Committed pages are mapped to RAM or to a file (such as a page file).
- Reserved – the page is not committed, but the address range is reserved for possible future commitment. From the CPU's perspective, it's the same as Free – any access attempt raises an access violation exception. However, new allocation attempts using the `VirtualAlloc` function (or `NtAllocateVirtualMemory`, the related native API) that does not specify a specific address would not allocate in the reserved region. A classic example of using reserved memory to maintain contiguous virtual address space while conserving committed memory usage is described later in this chapter in the section "Thread Stacks".

System Memory

The lower part of the address space is for user-mode processes use. While a particular thread is executing, its associated process address space is visible from address zero to the upper limit as described in the previous section. The operating system, however, must also reside somewhere – and that somewhere is the upper address range that's supported on the system, as follows:

- On 32-bit systems running without the *increase user virtual address space* setting, the operating system resides in the upper 2 GB of virtual address space, from address `0x80000000` to `0xFFFFFFFF`.
- On 32-bit systems configured with the *increase user virtual address space* setting, the operating system resides in the address space left. For example, if the system is configured with 3 GB user address space per process (the maximum), the OS takes the upper 1 GB (from address `0xC0000000` to `0xFFFFFFFF`). The component that suffers mostly from this address space reduction is the file system cache.

- On 64-bit systems running Windows 8, Server 2012 and earlier, the OS takes the upper 8 TB of virtual address space.
- On 64-bit systems running Windows 8.1, Server 2012 R2 and later, the OS takes the upper 128 TB of virtual address space.

Figure 1-4 shows the virtual memory layout for the two “extreme” cases: 32-bit process on a 32-bit system (left) and a 64-bit process on a 64-bit system (right).

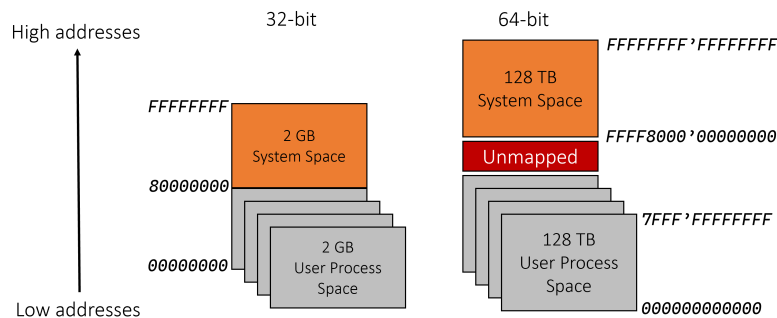


Figure 1-4: virtual memory layout

System space is not process-relative – after all, it’s the same system, the same kernel, the same drivers that service every process on the system (the exception is some system memory that is on a per-session basis but is not important for this discussion). It follows that any address in system space is absolute rather than relative, since it “looks” the same from every process context. Of course, actual access from user mode into system space results in an access violation exception.

System space is where the kernel itself, the Hardware Abstraction Layer (HAL), and kernel drivers reside once loaded. Thus, kernel drivers are automatically protected from direct user mode access. It also means they have a potentially system-wide impact. For example, if a kernel driver leaks memory, that memory will not be freed even after the driver unloads. User-mode processes, on the other hand, can never leak anything beyond their lifetime. The kernel is responsible for closing and freeing everything private to a dead process (all handles are closed and all private memory is freed).

Threads

The actual entities that execute code are threads. A Thread is contained within a process, using the resources exposed by the process to do work (such as virtual memory and handles to kernel objects). The most important information a thread owns is the following:

- Current access mode, either user or kernel.
- Execution context, including processor registers and execution state.
- One or two stacks, used for local variable allocations and call management.
- Thread Local Storage (TLS) array, which provides a way to store thread-private data with uniform access semantics.
- Base priority and a current (dynamic) priority.
- Processor affinity, indicating on which processors the thread is allowed to run on.

The most common states a thread can be in are:

- Running – currently executing code on a (logical) processor.
- Ready – waiting to be scheduled for execution because all relevant processors are busy or unavailable.
- Waiting – waiting for some event to occur before proceeding. Once the event occurs, the thread goes to the Ready state.

Figure 1-5 shows the state diagram for these states. The numbers in parenthesis indicate the state numbers, as can be viewed by tools such as *Performance Monitor*. Note that the Ready state has a sibling state called Deferred Ready, which is similar, and exists to minimize internal locking.

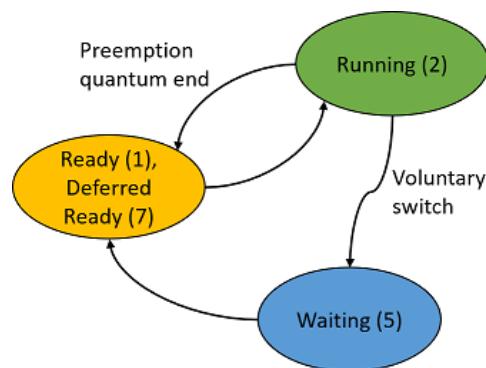


Figure 1-5: Common thread states

Thread Stacks

Each thread has a stack it uses while executing, used to store local variables, parameters passed to functions (in some cases), and where return addresses are stored prior to making function calls. A thread has at least one stack residing in system (kernel) space, and it's pretty small (default is 12 KB on 32-bit systems and 24 KB on 64-bit systems). A user-mode thread has a second stack in its process user-space address range and is considerably larger (by default can grow to 1 MB). An example with three user-mode threads and their stacks is shown in figure 1-6. In the figure, threads 1 and 2 are in process A and thread 3 is in process B.

The kernel stack always resides in RAM while the thread is in the Running or Ready states. The reason for this is subtle and will be discussed later in this chapter. The user-mode stack, on the other hand, may be paged out, just like any other user-mode memory.

The user-mode stack is handled differently than the kernel-mode stack in terms of its size. It starts out with a certain amount of committed memory (could be as small as a single page), where the next page is committed with a `PAGE_GUARD` attribute. The rest of the stack address space memory is reserved, thus not wasting memory. The idea is to grow the stack in case the thread's code needs to use more stack space. If the thread needs more stack space it would access the guard page which would throw a page-guard exception. The memory manager then removes the guard protection, and commits an additional page, marking it with a `PAGE_GUARD` attribute. This way, the stack grows as needed, avoiding the entire stack memory being committed upfront. Figure 1-7 shows this layout.

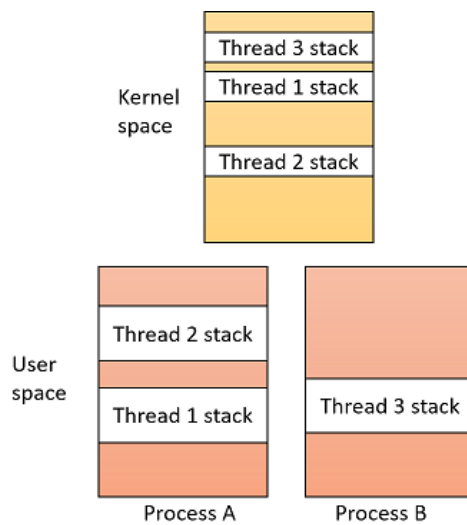


Figure 1-6: User mode threads and their stacks



Technically, Windows uses 3 guard pages rather than one in most cases.

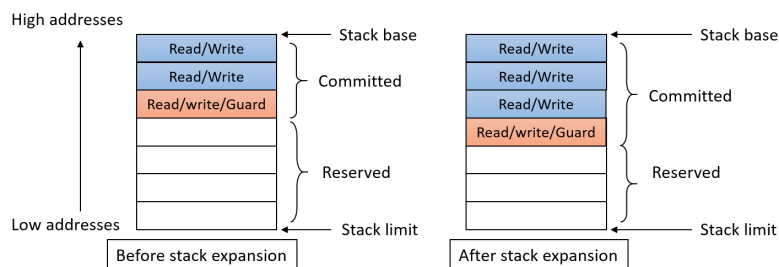


Figure 1-7: Thread's stack in user space

The sizes of a thread's user-mode stack are determined as follows:

- The executable image has a stack commit and reserved values in its Portable Executable (PE) header. These are taken as defaults if a thread does not specify alternative values. These are always used for the first thread in the process.
- When a thread is created with `CreateThread` (or similar functions), the caller can specify its required stack size, either the upfront committed size or the reserved size (but not both), depending on a flag provided to the function; specifying zero uses the defaults set in the PE header.



Curiously enough, the functions `CreateThread` and `CreateRemoteThread(Ex)` only allow specifying a single value for the stack size and can be the committed or the reserved size, but not both. The native (undocumented) function, `NtCreateThreadEx` allows specifying both values.

System Services (a.k.a. System Calls)

Applications need to perform various operations that are not purely computational, such as allocating memory, opening files, creating threads, etc. These operations can only be ultimately performed by code running in kernel mode. So how would user-mode code be able to perform such operations?

Let's take a common (simple) example: a user running a *Notepad* process uses the *File / Open* menu to request opening a file. *Notepad*'s code responds by calling the `CreateFile` documented Windows API function. `CreateFile` is documented as implemented in `kernel32.dll`, one of the Windows subsystem DLLs. This function still runs in user mode, so there is no way it can directly open a file. After some error checking, it calls `NtCreateFile`, a function implemented in `NTDLL.dll`, a foundational DLL that implements the API known as the *Native API*, and is the lowest layer of code which is still in user mode. This function (documented in the *Windows Driver Kit* for device driver developers) is the one that makes the transition to kernel mode. Before the actual transition, it puts a number, called system service number, into a CPU register (EAX on Intel/AMD architectures). Then it issues a special CPU instruction (`syscall` on x64 or `sysenter` on x86) that makes the actual transition to kernel mode while jumping to a predefined routine called the *system service dispatcher*.

The system service dispatcher, in turn, uses the value in that EAX register as an index into a *System Service Dispatch Table* (SSDT). Using this table, the code jumps to the system service (system call) itself. For our *Notepad* example, the SSDT entry would point to the `NtCreateFile` function, implemented by the kernel's I/O manager. Notice the function has the same name as the one in `NTDLL.dll`, and has the same parameters as well. On the kernel side is the real implementation. Once the system service is complete, the thread returns to user mode to execute the instruction following `sysenter/syscall`. This sequence of calls is depicted in figure 1-8.

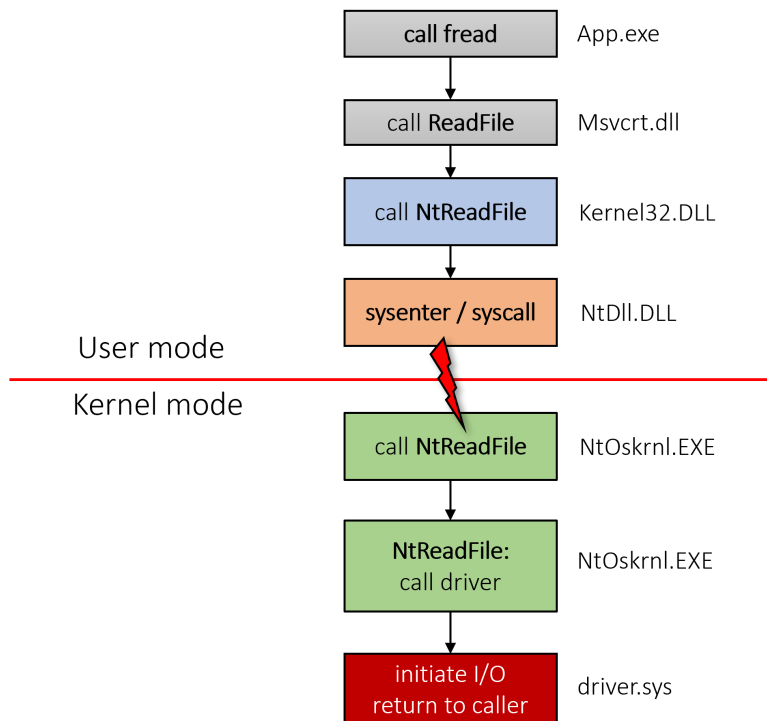


Figure 1-8: System service function call flow

General System Architecture

Figure 1-9 shows the general architecture of Windows, comprising of user-mode and kernel-mode components.

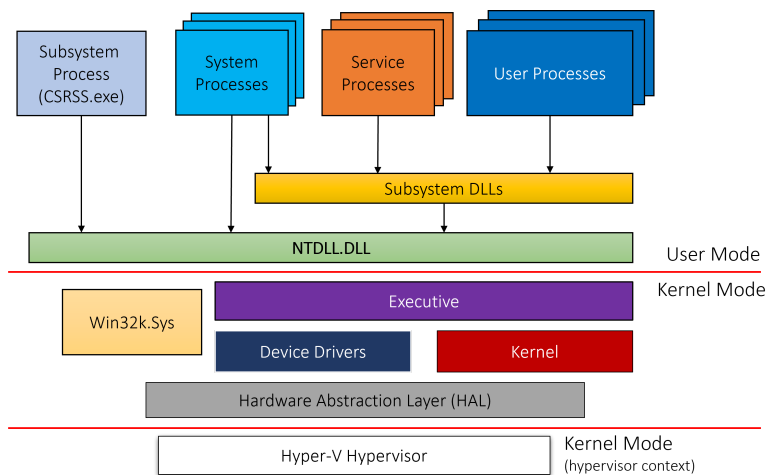


Figure 1-9: Windows system architecture

Here's a quick rundown of the named boxes appearing in figure 1-9:

- **User processes**

These are normal processes based on image files, executing on the system, such as instances of *Notepad.exe*, *cmd.exe*, *explorer.exe*, and so on.

- **Subsystem DLLs**

Subsystem DLLs are dynamic link libraries (DLLs) that implement the API of a subsystem. A subsystem is a particular view of the capabilities exposed by the kernel. Technically, starting from Windows 8.1, there is only a single subsystem – the Windows Subsystem. The subsystem DLLs include well-known files, such as *kernel32.dll*, *user32.dll*, *gdi32.dll*, *advapi32.dll*, *combase.dll*, and many others. These include mostly the officially documented API of Windows.

- **NTDLL.DLL**

A system-wide DLL, implementing the Windows native API. This is the lowest layer of code which is still in user mode. Its most important role is to make the transition to kernel mode for system call invocation. NTDLL also implements the Heap Manager, the Image Loader and some part of the user mode thread pool.

- **Service Processes**

Service processes are normal Windows processes that communicate with the Service Control Manager (SCM, implemented in *services.exe*) and allow some control over their lifetime. The SCM can start, stop, pause, resume and send other messages to services. Services typically execute under one of the special Windows accounts – local system, network service or local service.

- **Executive**

The Executive is the upper layer of *NtOskrnl.exe* (the “kernel”). It hosts most of the code that is in kernel mode. It includes mostly the various “managers”: Object Manager, Memory Manager, I/O Manager, Plug & Play Manager, Power Manager, Configuration Manager, etc. It's by far larger than the lower Kernel layer.

- **Kernel**

The Kernel layer implements the most fundamental and time-sensitive parts of kernel-mode OS code. This includes thread scheduling, interrupt and exception dispatching, and implementation of various kernel primitives such as mutexes and semaphores. Some of the kernel code is written in CPU-specific machine language for efficiency and for getting direct access to CPU-specific details.

- **Device Drivers**

Device drivers are loadable kernel modules. Their code executes in kernel mode and so has the full power of the kernel. This book is dedicated to writing certain types of kernel drivers.

- **Win32k.sys**

This is the kernel-mode component of the Windows subsystem. Essentially, it's a kernel module (driver) that handles the user interface part of Windows and the classic *Graphics Device Interface* (GDI) APIs. This means that all windowing operations (CreateWindowEx, GetMessage, PostMessage, etc.) are handled by this component. The rest of the system has little-to-none knowledge of UI.

- **Hardware Abstraction Layer (HAL)**

The HAL is a software abstraction layer over the hardware closest to the CPU. It allows device drivers to use APIs that do not require detailed and specific knowledge of things like Interrupt Controllers or DMA controller. Naturally, this layer is mostly useful for device drivers written to handle hardware devices.

- **System Processes**

System processes is an umbrella term used to describe processes that are typically “just there”, doing their thing where normally these processes are not communicated with directly. They are important nonetheless, and some in fact, critical to the system's well-being. Terminating some of them is fatal and causes a system crash. Some of the system processes are native processes, meaning they use the native API only (the API implemented by NTDLL). Example system processes include *Smss.exe*, *Lsass.exe*, *Winlogon.exe*, and *Services.exe*.

- **Subsystem Process**

The Windows subsystem process, running the image *Csrss.exe*, can be viewed as a helper to the kernel for managing processes running under the Windows subsystem. It is a critical process, meaning if killed, the system would crash. There is one *Csrss.exe* instance per session, so on a standard system two instances would exist – one for session 0 and one for the logged-on user session (typically 1). Although *Csrss.exe* is the “manager” of the Windows subsystem (the only one left these days), its importance goes beyond just this role.

- **Hyper-V Hypervisor**

The Hyper-V hypervisor exists on Windows 10 and server 2016 (and later) systems if they support *Virtualization Based Security* (VBS). VBS provides an extra layer of security, where the normal OS is a virtual machine controlled by Hyper-V. Two distinct *Virtual Trust Levels* (VTLs) are defined, where VTL 0 consists of the normal user-mode/kernel-mode we know of, and VTL 1 contains the secure kernel and *Isolated User Mode* (IUM). VBS is beyond the scope of this book. For more information, check out the *Windows Internals* book and/or the Microsoft documentation.



Windows 10 version 1607 introduced the *Windows Subsystem for Linux* (WSL). Although this may look like yet another subsystem, like the old POSIX and OS/2 subsystems supported by Windows, it is not like that at all. The old subsystems were able to execute POSIX and OS/2 apps if these were compiled using a Windows compiler to use the PE format and Windows system calls. WSL, on the other hand, has no such requirement. Existing executables from Linux (stored in ELF format) can be run as-is on Windows, without any recompilation.

To make something like this work, a new process type was created – the Pico process together with a Pico provider. Briefly, a Pico process is an empty address space (minimal process) that is used for WSL processes, where every system call (Linux system call) must be intercepted and translated to the Windows system call(s) equivalent using that Pico provider (a device driver). There is a true Linux (the user-mode part) installed on the Windows machine.

The above description is for WSL version 1. Starting with Windows 10 version 2004, Windows supports a new version of WSL known as WSL 2. WSL 2 is not based on pico processes anymore. Instead, it's based on a hybrid virtual machine technology that allows installing a full Linux system (including the Linux kernel), but still see and share the Windows machine's resources, such as the file system. WSL 2 is faster than WSL 1 and solves some edge cases that didn't work well in WSL 1, thanks to the real Linux kernel handling Linux system calls.

Handles and Objects

The Windows kernel exposes various types of objects for use by user-mode processes, the kernel itself and kernel-mode drivers. Instances of these types are data structures in system space, created by the Object Manager (part of the executive) when requested to do so by user-mode or kernel-mode code. Objects are reference counted – only when the last reference to the object is released will the object be destroyed and freed from memory.

Since these object instances reside in system space, they cannot be accessed directly by user mode. User mode must use an indirect access mechanism, known as handles. A handle is an index to an entry in a table maintained on a process by process basis, stored in kernel space, that points to a kernel object residing in system space. There are various `Create*` and `Open*` functions to create/open objects and retrieve back handles to these objects. For example, the `CreateMutex` user-mode function allows creating or opening a mutex (depending on whether the object is named and exists). If successful, the function returns a handle to the object. A return value of zero means an invalid handle (and a function call failure). The `OpenMutex` function, on the other hand, tries to open a handle to a named mutex. If the mutex with that name does not exist, the function fails and returns null (0).

Kernel (and driver) code can use either a handle or a direct pointer to an object. The choice is usually based on the API the code wants to call. In some cases, a handle given by user mode to the driver must be turned into a pointer with the `ObReferenceObjectByHandle` function. We'll discuss these details in a later chapter.



Most functions return null (zero) on failure, but some do not. Most notably, the `CreateFile` function returns `INVALID_HANDLE_VALUE` (-1) if it fails.

Handle values are multiples of 4, where the first valid handle is 4; Zero is never a valid handle value.

Kernel-mode code can use handles when creating/opening objects, but they can also use direct pointers to kernel objects. This is typically done when a certain API demands it. Kernel code can get a pointer to an object given a valid handle using the `ObReferenceObjectByHandle` function. If successful, the reference count on the object is incremented, so there is no danger that if the user-mode client holding the handle decided to close it while kernel code holds a pointer to the object would now hold a dangling pointer. The object is safe to access regardless of the handle-holder until the kernel code calls `ObDereferenceObject`, which decrements the reference count; if the kernel code missed this call, that's a resource leak which will only be resolved in the next system boot.

All objects are reference counted. The object manager maintains a handle count and total reference count for objects. Once an object is no longer needed, its client should close the handle (if a handle was used to access the object) or dereference the object (if kernel client using a pointer). From that point on, the code should consider its handle/pointer to be invalid. The Object Manager will destroy the object if its reference count reaches zero.

Each object points to an object type, which holds information on the type itself, meaning there is a single type object for each type of object. These are also exposed as exported global kernel variables, some of which are defined in the kernel headers and are needed in certain cases, as we'll see in later chapters.

Object Names

Some types of objects can have names. These names can be used to open objects by name with a suitable *Open* function. Note that not all objects have names; for example, processes and threads don't have names – they have IDs. That's why the `OpenProcess` and `OpenThread` functions require a process/thread identifier (a number) rather than a string-base name. Another somewhat weird case of an object that does not have a name is a file. The file name is not the object's name – these are different concepts.



Threads appear to have a name (starting from Windows 10), that can be set with the user-mode API `SetThreadDescription`. This is not, however, a true name, but rather a friendly name/description most useful in debugging, as Visual Studio shows a thread's description, if there is any.

From user-mode code, calling a *Create* function with a name creates the object with that name if an object with that name does not exist, but if it exists it just opens the existing object. In the latter case, calling `GetLastError` returns `ERROR_ALREADY_EXISTS`, indicating this is not a new object, and the returned handle is yet another handle to an existing object.

The name provided to a *Create* function is not actually the final name of the object. It's prepended with `\Sessions\x\BaseNamedObjects\` where *x* is the session ID of the caller. If the session is zero, the name is prepended with `\BaseNamedObjects\`. If the caller happens to be running in an AppContainer (typically a Universal Windows Platform process), then the prepended string is more complex and consists of the unique AppContainer SID: `\Sessions\x\AppDataContainerNamedObjects\{AppContainerSID}`.

All the above means is that object names are session-relative (and in the case of AppContainer – package relative). If an object must be shared across sessions it can be created in session 0 by prepending the object name with *Global*; for example, creating a mutex with the `CreateMutex` function named *Global\MyMutex* will create it under `\BaseNamedObjects`. Note that AppContainers do not have the power to use session 0 object namespace. This hierarchy can be viewed with the Sysinternals *WinObj* tool (run elevated) as shown in figure 1-10.

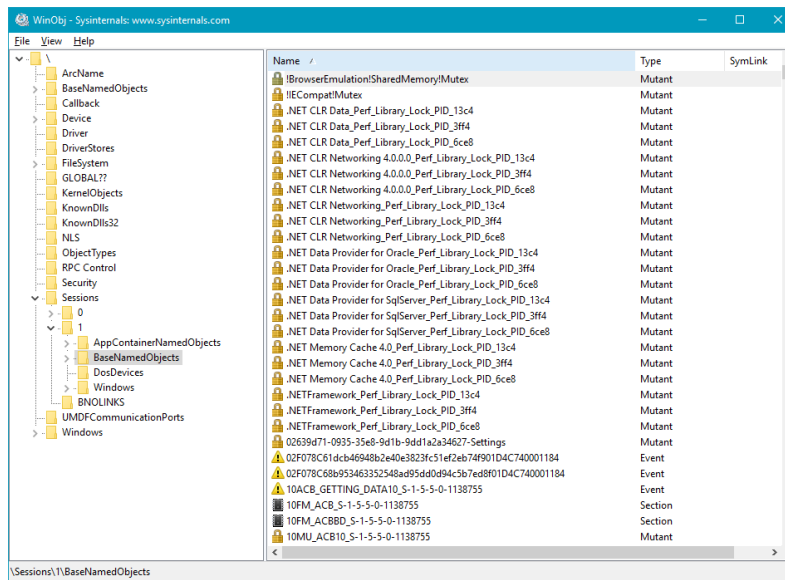


Figure 1-10: Sysinternals WinObj tool

The view shown in figure 1-9 is the object manager namespace, comprising of a hierarchy of named objects. This entire structure is held in memory and manipulated by the Object Manager (part of the Executive) as required. Note that unnamed objects are not part of this structure, meaning the objects seen in *WinObj* do not comprise all the existing objects, but rather all the objects that were created with a name.

Every process has a private handle table to kernel objects (whether named or not), which can be viewed with the *Process Explorer* and/or *Handles* Sysinternals tools. A screenshot of Process Explorer showing handles in some process is shown in figure 1-11. The default columns shown in the handles view are the object type and name only. However, there are other columns available, as shown in figure 1-11.

The *Access* column in *Process Explorer*'s handles view shows the access mask which was used to open or create the handle. This access mask is key to what operations are allowed to be performed with a specific handle. For example, if client code wants to terminate a process, it must call the `OpenProcess` function first, to obtain a handle to the required process with an access mask of (at least) `PROCESS_TERMINATE`, otherwise there is no way to terminate the process with that handle. If the call succeeds, then the call to `TerminateProcess` is bound to succeed.

Here's a user-mode example for terminating a process given a process ID:

```
bool KillProcess(DWORD pid) {
    //
    // open a powerful-enough handle to the process
    //
    HANDLE hProcess = OpenProcess(PROCESS_TERMINATE, FALSE, pid);
    if (!hProcess)
        return false;

    //
    // now kill it with some arbitrary exit code
    //
    BOOL success = TerminateProcess(hProcess, 1);

    //
    // close the handle
    //
    CloseHandle(hProcess);

    return success != FALSE;
}
```

The *Decoded Access* column provides a textual description of the access mask (for some object types), making it easier to identify the exact access allowed for a particular handle.

Double-clicking a handle entry (or right-clicking and selecting *Properties*) shows some of the object's properties. Figure 1-12 shows a screenshot of an example event object properties.

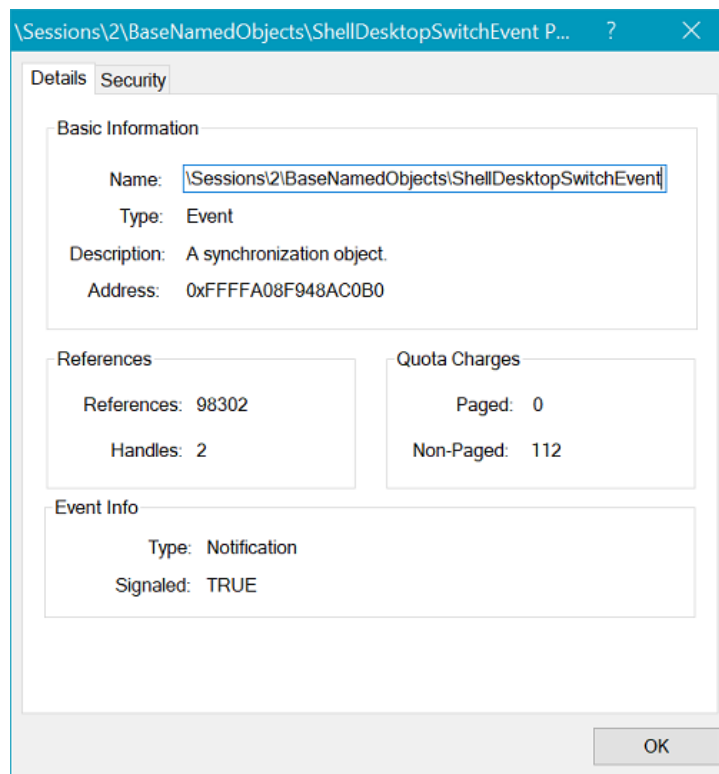


Figure 1-12: Object properties in Process Explorer

Notice that the dialog shown in figure 1-12 is for the object's properties, rather than the handle's. In other words, looking at an object's properties from any handle that points to the same object shows the same information.

The properties in figure 1-12 include the object's name (if any), its type, a short description, its address in kernel memory, the number of open handles, and some specific object information, such as the state and type of the event object shown. Note that the *References* shown do not indicate the actual number of outstanding references to the object (it does prior to Windows 8.1). A proper way to see the actual reference count for the object is to use the kernel debugger's `!truref` command, as shown here:

```
lkd> !object 0xFFFFA08F948AC0B0
Object: fffffa08f948ac0b0 Type: (ffffa08f684df140) Event
ObjectHeader: fffffa08f948ac080 (new version)
HandleCount: 2 PointerCount: 65535
Directory Object: fffff90839b63a700 Name: ShellDesktopSwitchEvent
lkd> !truref fffffa08f948ac0b0
fffffa08f948ac0b0: HandleCount: 2 PointerCount: 65535 RealPointerCount: 3
```

We'll take a closer look at the attributes of objects and the kernel debugger in later chapters.

In the next chapter, we'll start writing a very simple driver to show and use many of the tools we'll need later in this book.

Chapter 2: Getting Started with Kernel Development

This chapter deals with the fundamentals needed to get up and running with kernel driver development. During the course of this chapter, you'll install the necessary tools and write a very basic driver that can be loaded and unloaded.

In this chapter:

- **Installing the Tools**
 - **Creating a Driver Project**
 - **The `DriverEntry` and `Unload` routines**
 - **Deploying the Driver**
 - **Simple Tracing**
-

Installing the Tools

In the old days (pre-2012), the process of developing and building drivers included using a dedicated build tool from the Device Driver Kit (DDK), without having an integrated development experience developers were used to when developing user-mode applications. There were some workarounds, but none of them was perfect nor officially supported by Microsoft.

Fortunately, starting with Visual Studio 2012 and Windows Driver Kit 8, Microsoft officially supports building drivers with Visual Studio (with *msbuild*), without the need to use a separate compiler and build tools.

To get started with driver development, the following tools must be installed (in this order) on your development machine:

- Visual Studio 2019 with the latest updates. Make sure the C++ workload is selected during installation. Note that any SKU will do, including the free Community edition.

- Windows 11 SDK (generally, the latest is recommended). Make sure at least the *Debugging Tools for Windows* item is selected during installation.
- Windows 11 Driver Kit (WDK) - it supports building drivers for Windows 7 and later versions of Windows. Make sure the wizard installs the project templates for Visual Studio at the end of the installation.
- The *Sysinternals* tools, which are invaluable in any “internals” work, can be downloaded for free from <http://www.sysinternals.com>. Click on *Sysinternals Suite* on the left of that web page and download the Sysinternals Suite zip file. Unzip to any folder, and the tools are ready to go.



The SDK and WDK versions must match. Follow the guidelines in the WDK download page to load the corresponding SDK with the WDK.

A quick way to make sure the WDK templates are installed correctly is to open Visual Studio and select New Project and look for driver projects, such as “Empty WDM Driver”.

Creating a Driver Project

With the above installations in place, a new driver project can be created. The template you’ll use in this section is “WDM Empty Driver”. Figure 2-1 shows what the New Project dialog looks like for this type of driver in Visual Studio 2019. Figure 2-2 shows the same initial wizard with Visual Studio 2019 if the *Classic Project Dialog* extension is installed and enabled. The project in both figures is named “Sample”.

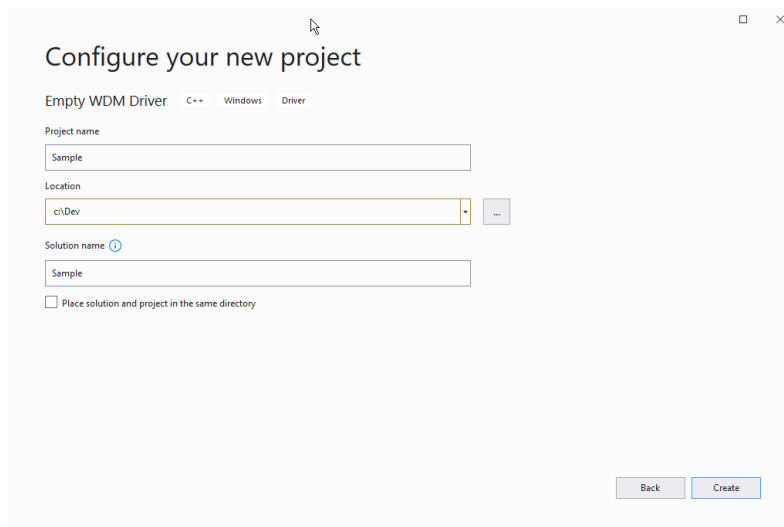


Figure 2-1: New WDM Driver Project in Visual Studio 2019

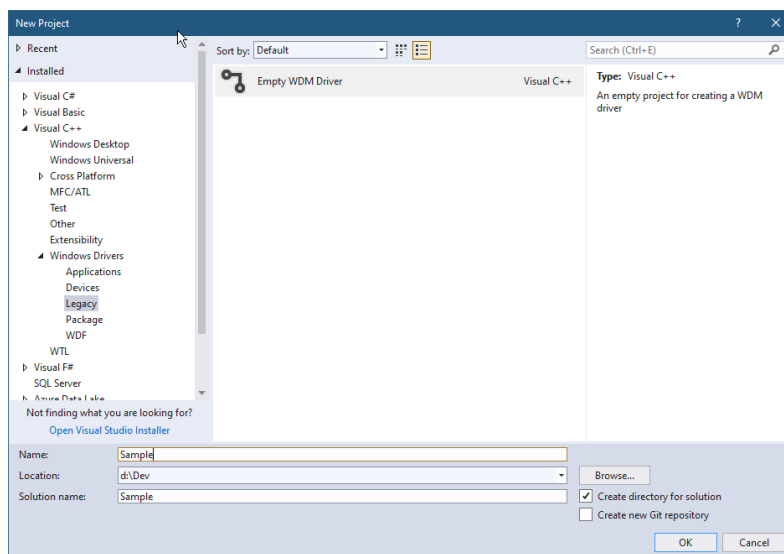


**Figure 2-2: New WDM Driver Project in Visual Studio 2019
with the *Classic Project Dialog* extension**

Once the project is created, the *Solution Explorer* shows a single file within the *Driver Files* filter - *Sample.inf*. You won't need this file in this example, so simply delete it (right-click and select *Remove* or press the *Del* key).

Now it's time to add a source file. Right-click the *Source Files* node in *Solution Explorer* and select *Add / New Item...* from the File menu. Select a C++ source file and name it *Sample.cpp*. Click *OK* to create it.

The DriverEntry and Unload Routines

Every driver has an entry point called `DriverEntry` by default. This can be considered the “main” function of the driver, comparable to the classic `main` of a user-mode application. This function is called by a system thread at IRQL `PASSIVE_LEVEL` (0). (IRQLs are discussed in detail in chapter 8.)

`DriverEntry` has a predefined prototype, shown here:

```
NTSTATUS
DriverEntry(_In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath\
);
```

The `_In_` annotations are part of the *Source (Code) Annotation Language* (SAL). These annotations are transparent to the compiler, but provide metadata useful for human readers and static analysis tools. I may remove these annotations in code samples to make it easier to read, but you should use SAL annotations whenever possible.

A minimal `DriverEntry` routine could just return a successful status, like so:

```
NTSTATUS
DriverEntry(
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath) {
    return STATUS_SUCCESS;
}
```

This code would not yet compile. First, you’ll need to include a header that has the required definitions for the types present in `DriverEntry`. Here’s one possibility:

```
#include <ntddk.h>
```

Now the code has a better chance of compiling, but would still fail. One reason is that by default, the compiler is set to treat warnings as errors, and the function does not make use of its given arguments. Removing *treat warnings as errors* from the compiler’s options is not recommended, as some warnings may be errors in disguise. These warnings can be resolved by removing the argument names entirely (or commenting them out), which is fine for C++ files.

There is another, more “classic” way to solve this, which is to use the `UNREFERENCED_PARAMETER` macro:

```
NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(DriverObject);
    UNREFERENCED_PARAMETER(RegistryPath);

    return STATUS_SUCCESS;
}
```

As it turns out, this macro actually references the argument given just by writing its value as is, and this shuts the compiler up, making the argument technically “referenced”.

Building the project now compiles fine, but causes a linker error. The `DriverEntry` function must have C-linkage, which is not the default in C++ compilation. Here’s the final version of a successful build of the driver consisting of a `DriverEntry` function only:

```
extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(DriverObject);
    UNREFERENCED_PARAMETER(RegistryPath);

    return STATUS_SUCCESS;
}
```

At some point, the driver may be unloaded. At that time, anything done in the `DriverEntry` function must be undone. Failure to do so creates a leak, which the kernel will not clean up until the next reboot. Drivers can have an `Unload` routine that is automatically called before the driver is unloaded from memory. Its pointer must be set using the `DriverUnload` member of the driver object:

```
DriverObject->DriverUnload = SampleUnload;
```

The unload routine accepts the driver object (the same one passed to `DriverEntry`) and returns `void`. As our sample driver has done nothing in terms of resource allocation in `DriverEntry`, there is nothing to do in the `Unload` routine, so we can leave it empty for now:

```
void SampleUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);
}
```

Here is the complete driver source at this point:

```
#include <ntddk.h>

void SampleUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);
}

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = SampleUnload;

    return STATUS_SUCCESS;
}
```

Deploying the Driver

Now that we have a successfully compiled *Sample.sys* driver file, let's install it on a system and then load it. Normally, you would install and load a driver on a virtual machine, to remove the risk of crashing your primary machine. Feel free to do so, or take the slight risk with this minimalist driver.

Installing a software driver, just like installing a user-mode service, requires calling the `CreateService` API with proper arguments, or using a comparable tool. One of the well-known tools for this purpose is *Sc.exe* (short for *Service Control*), a built-in Windows tool for managing services. We'll use this tool to install and then load the driver. Note that installation and loading of drivers is a privileged operation, normally available for administrators.

Open an elevated command window and type the following (the last part should be the path on your system where the SYS file resides):

```
sc create sample type= kernel binPath= c:\dev\sample\x64\debug\sample.sys
```

Note there is no space between *type* and the equal sign, and there is a space between the equal sign and *kernel*; same goes for the second part.

If all goes well, the output should indicate success. To test the installation, you can open the registry editor (*regedit.exe*) and look for the driver details at

HKLM\System\CurrentControlSet\Services\Sample. Figure 2-3 shows a screenshot of the registry editor after the previous command.

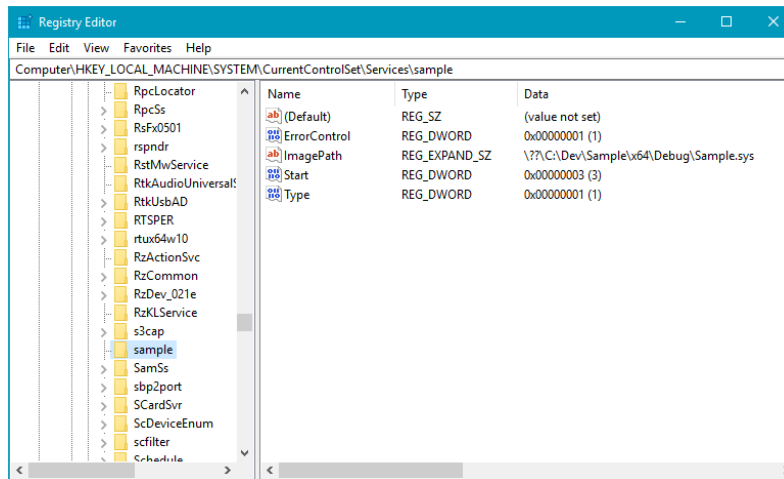


Figure 2-3: Registry for an installed driver

To load the driver, we can use the *Sc.exe* tool again, this time with the *start* option, which uses the *StartService* API to load the driver (the same API used to load services). However, on 64 bit systems drivers must be signed, and so normally the following command would fail:

```
sc start sample
```

Since it's inconvenient to sign a driver during development (maybe even not possible if you don't have a proper certificate), a better option is to put the system into test signing mode. In this mode, unsigned drivers can be loaded without a hitch.

With an elevated command window, test signing can be turned on like so:

```
bcdedit /set testsigning on
```

Unfortunately, this command requires a reboot to take effect. Once rebooted, the previous start command should succeed.



If you are testing on a Windows 10 (or later) system with *Secure Boot* enabled, changing the test signing mode will fail. This is one of the settings protected by Secure Boot (local kernel debugging is also protected by Secure Boot). If you can't disable Secure Boot through BIOS setting, because of IT policy or some other reason, your best option is to test on a virtual machine.

There is yet another setting that you may need to specify if you intend to test the driver on pre-Windows 10 machine. In this case, you have to set the target OS version in the project properties dialog, as shown in figure 2-4. Notice that I have selected all configurations and all platforms, so that when switching configurations (Debug/Release) or platforms (x86/x64/ARM/ARM64), the setting is maintained.

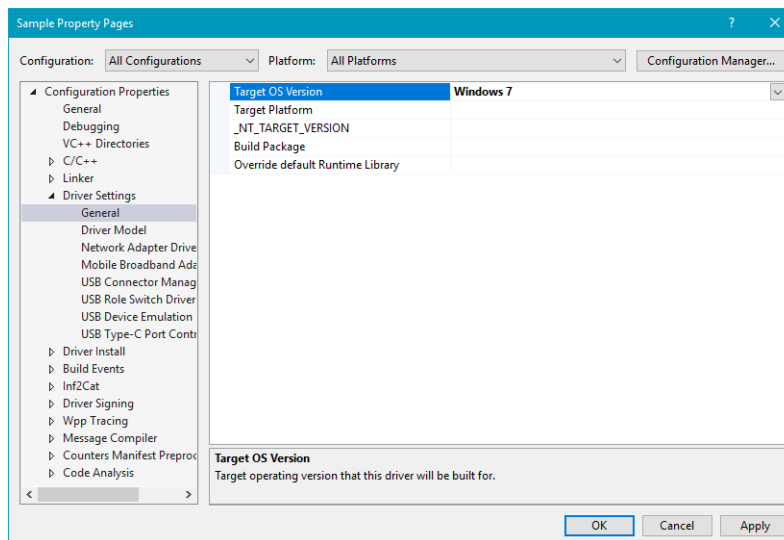


Figure 2-4: Setting Target OS Platform in the project properties

Once test signing mode is on, and the driver is loaded, this is the output you should see:

```
c:/>sc start sample
SERVICE_NAME: sample
        TYPE               : 1  KERNEL_DRIVER
        STATE                : 4  RUNNING
                                (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE       : 0  (0x0)
        SERVICE_EXIT_CODE    : 0  (0x0)
        CHECKPOINT           : 0x0
        WAIT_HINT            : 0x0
```

PID : 0
FLAGS :

This means everything is well, and the driver is loaded. To confirm, we can open *Process Explorer* and find the *Sample.Sys* driver image file. Figure 2-5 shows the details of the sample driver image loaded into system space.

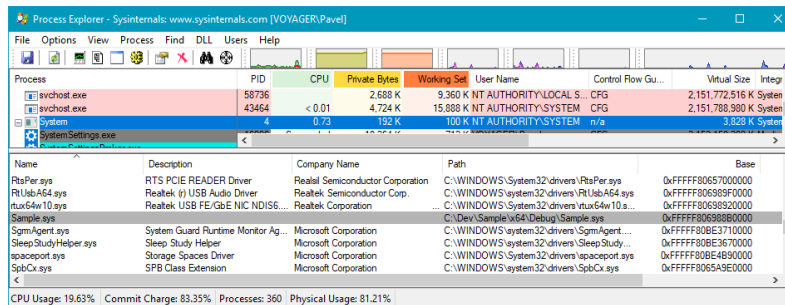


Figure 2-5: sample driver image loaded into system space

At this point, we can unload the driver using the following command:

```
sc stop sample
```

Behind the scenes, *sc.exe* calls the *ControlService* API with the *SERVICE_CONTROL_STOP* value. Unloading the driver causes the *Unload* routine to be called, which at this time does nothing. You can verify the driver is indeed unloaded by looking at *Process Explorer* again; the driver image entry should not be there anymore.

Simple Tracing

How can we know for sure that the *DriverEntry* and *Unload* routines actually executed? Let's add basic tracing to these functions. Drivers can use the *DbgPrint* function to output *printf*-style text that can be viewed using the kernel debugger, or some other tool.

Here is updated versions for *DriverEntry* and the *Unload* routine that use *KdPrint* to trace the fact their code executed:

```
void SampleUnload(PDRIVER_OBJECT DriverObject) {  
    UNREFERENCED_PARAMETER(DriverObject);
```

```

        DbgPrint("Sample driver Unload called\n");
    }

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = SampleUnload;

    DbgPrint("Sample driver initialized successfully\n");

    return STATUS_SUCCESS;
}

```

A more typical approach is to have these outputs in Debug builds only. This is because Dbgprint has some overhead that you may want to avoid in Release builds. KdPrint is a macro that is only compiled in Debug builds and calls the underlying DbgPrint kernel API. Here is a revised version that uses KdPrint:

```

void SampleUnload(PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);

    KdPrint(("Sample driver Unload called\n"));
}

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = SampleUnload;

    KdPrint(("Sample driver initialized successfully\n"));

    return STATUS_SUCCESS;
}

```

Notice the double parenthesis when using KdPrint. This is required because KdPrint is a macro, but apparently accepts any number of arguments, a-la printf. Since macros cannot receive a variable number of parameters, a compiler trick is used to call the DbgPrint function that does accept a variable number of parameters.

With these statements in place, we would like to load the driver again and see these messages. We'll use a kernel debugger in chapter 4, but for now we'll use a useful *Sysinternals* tool named *DebugView*.

Before running *DebugView*, you'll need to make some preparations. First, starting with Windows Vista, DbgPrint output is not actually generated unless a certain value is in the registry. You'll have to add a key named *Debug Print*

Filter under *HKLM\SYSTEM\CurrentControlSet\Control\Session Manager* (the key typically does not exist). Within this new key, add a *DWORD* value named *DEFAULT* (not the default value that exists in any key) and set its value to 8 (technically, any value with bit 3 set will do). Figure 2-6 shows the setting in *RegEdit*. Unfortunately, you'll have to restart the system for this setting to take effect.

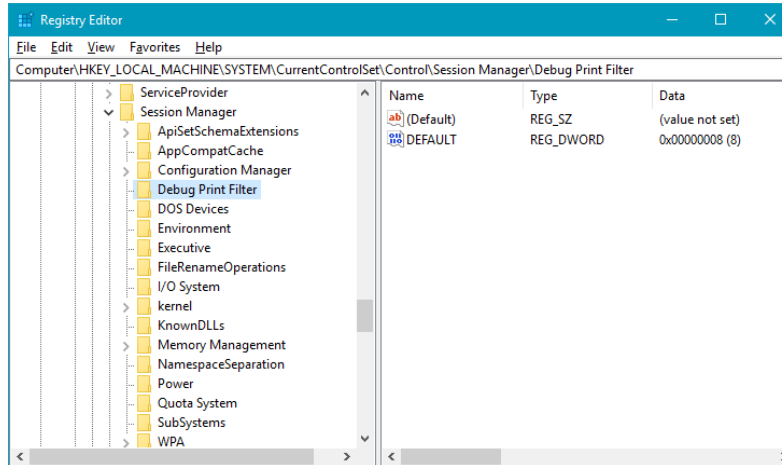


Figure 2-6: Debug Print Filter key in the registry

Once this setting has been applied, run *DebugView* (*DbgView.exe*) elevated. In the *Options* menu, make sure *Capture Kernel* is selected (or press *Ctrl+K*). You can safely deselect *Capture Win32* and *Capture Global Win32*, so that user-mode output from various processes does not clutter the display.



DebugView is able to show kernel debug output even without the Registry value shown in figure 2-6 if you select *Enable Verbose Kernel Output* from its *Capture* menu. However, it seems this option does not work on Windows 11, and the Registry setting is necessary.

Build the driver, if you haven't already. Now you can load the driver again from an elevated command window (*sc start sample*). You should see output in *DebugView* as shown in figure 2-7. If you unload the driver, you'll see another message appearing because the *Unload* routine was called. (The third output line is from another driver and has nothing to do with our sample driver)

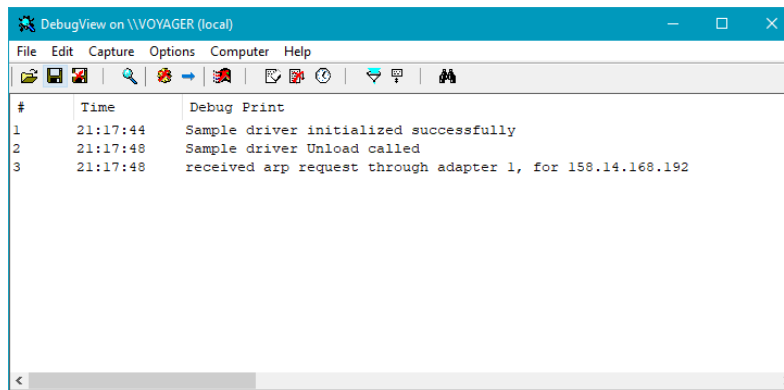


Figure 2-7: Sysinternals *DebugView* Output



Add code to the sample `DriverEntry` to output the Windows OS version: major, minor, and build number. Use the `RtlGetVersion` function to retrieve the information. Check the results with *DebugView*.

Summary

We've seen the tools you need to have for kernel development and wrote a very minimalistic driver to prove the basic tools work. In the next chapter, we'll look at the fundamental building blocks of kernel APIs, concepts, and fundamental structures.

Chapter 3: Kernel Programming Basics

In this chapter, we'll dig deeper into kernel APIs, structures, and definitions. We'll also examine some of the mechanisms that invoke code in a driver. Finally, we'll put all that knowledge together to create our first functional driver and client application.

In this chapter:

- **General Kernel Programming Guidelines**
 - **Debug vs. Release Builds**
 - **The Kernel API**
 - **Functions and Error Codes**
 - **Strings**
 - **Dynamic Memory Allocation**
 - **Linked Lists**
 - **Object Attributes**
 - **The Driver Object**
 - **Device Objects**
-

General Kernel Programming Guidelines

Developing kernel drivers requires the Windows Driver Kit (WDK), where the appropriate headers and libraries needed are located. The kernel APIs consist of C functions, very similar in essence to user-mode APIs. There are several differences, however. Table 3-1 summarizes the important differences between user-mode programming and kernel-mode programming.

Table 3-1: Differences between user mode and kernel mode development

	User Mode	Kernel Mode
Unhandled Exceptions	Unhandled exceptions crash the process	Unhandled exceptions crashes the system
Termination	When a process terminates, all private memory and resources are freed automatically	If a driver unloads without freeing everything it was using, there is a leak, only resolved in the next boot
Return values	API errors are sometimes ignored	Should (almost) never ignore errors
IRQL	Always PASSIVE_LEVEL (0)	May be DISPATCH_LEVEL (2) or higher
Bad coding	Typically localized to the process	Can have system-wide effects
Testing and Debugging	Typical testing and debugging done on the developer's machine	Debugging must be done with another machine
Libraries	Can use almost and C/C++ library (e.g. STL, boost)	Most standard libraries cannot be used
Exception Handling	Can use C++ exceptions or Structured Exception Handling (SEH)	Only SEH can be used
C++ Usage	Full C++ runtime available	No C++ runtime

Unhandled Exceptions

Exceptions occurring in user-mode that are not caught by the program cause the process to terminate prematurely. Kernel-mode code, on the other hand, being implicitly trusted, cannot recover from an unhandled exception. Such an exception causes the system to crash with the infamous *Blue screen of death* (BSOD) (newer versions of Windows have more diverse colors for the crash screen). The BSOD may first appear to be a form of punishment, but it's essentially a protection mechanism. The rationale being it, is that allowing the code to continue execution could cause irreversible damage to Windows (such as deleting important files or corrupting the registry) that may cause the system

to fail boot. It's better, then, to stop everything immediately to prevent potential damage. We'll discuss the BSOD in more detail in chapter 6.

All this leads to at least one conclusion: kernel code must be carefully programmed, meticulous, and not skip any details, or error checks.

Termination

When a process terminates, for whatever reason - either normally, because of an unhandled exception, or terminated by external code - it never leaks anything: all private memory is freed, and all handles are closed. Of course, premature handle closing may cause some loss of data, such as a file handle being closed before flushing some data to disk - but there are no resource leaks beyond the lifetime of the process; this is guaranteed by the kernel.

Kernel drivers, on the other hand, don't provide such a guarantee. If a driver unloads while still holding onto allocated memory or open kernel handles - these resources will not be freed automatically, only released at the next system boot.

Why is that? Can't the kernel keep track of a driver's allocations and resource usage so these can be freed automatically when the driver unloads? Theoretically, this would have been possible to achieve (although currently the kernel does not track such resource usage). The real issue is that it would be too dangerous for the kernel to attempt such cleanup. The kernel has no way of knowing whether the driver leaked those resources for a reason; for example, the driver could allocate some buffer and then pass it to another driver, with which it cooperates. That second driver may use the memory buffer and free it eventually. If the kernel attempted to free the buffer when the first driver unloads, the second driver would cause an access violation when accessing that now-freed buffer, causing a system crash.

This emphasizes the responsibility of a kernel driver to properly clean up after itself; no one else will do it.

Function Return Values

In typical user-mode code, return values from API functions are sometimes ignored, the developer being somewhat optimistic that the called function is

unlikely to fail. This may or may not be appropriate for one function or another, but in the worst case, an unhandled exception would later crash the process; the system, however, remains intact.

Ignoring return values from kernel APIs is much more dangerous (see the previous Termination section), and generally should be avoided. Even seemingly “innocent” looking functions can fail for unexpected reasons, so the golden rule here is - always check return status values from kernel APIs.

IRQL

Interrupt Request Level (IRQL) is an important kernel concept that will be further discussed in chapter 6. Suffice it to say at this point that normally a processor’s IRQL is zero, and in particular it’s always zero when user-mode code is executing. In kernel mode, it’s still zero most of the time - but not all the time. Some restrictions on code execution exist at IRQL 2 and higher, which means the driver writer must be careful to use only allowed APIs at that high IRQL. The effects of higher than zero IRQLs are discussed in chapter 6.

C++ Usage

In user mode programming, C++ has been used for many years, and it works well when combined with user-mode Windows APIs. With kernel code, Microsoft started officially supporting C++ with Visual Studio 2012 and WDK 8. C++ is not mandatory, of course, but it has some important benefits related to resource cleanup, with a C++ idiom called *Resource Acquisition Is Initialization* (RAII). We’ll use this RAII idiom quite a bit to make sure we don’t leak resources.

C++ as a language is almost fully supported for kernel code. But there is no C++ runtime in the kernel, and so some C++ features just cannot be used:

- The `new` and `delete` operators are not supported and will fail to compile. This is because their normal operation is to allocate from a user-mode heap, which is irrelevant within the kernel. The kernel API has “replacement” functions that are more closely modeled after the C functions `malloc` and `free`. We’ll discuss these functions later in this chapter. It is possible, however, to overload the `new` and `delete` operators similarly as is sometimes done in user-mode, and invoke the kernel

allocation and free functions in the implementation. We'll see how to do that later in this chapter as well.

- Global variables that have non-default constructors will not be called - there is no C/C++ runtime to call these constructors. These situations must be avoided, but there are some workarounds:
 - Avoid any code in the constructor and instead create some `Init` function to be called explicitly from driver code (e.g. from `DriverEntry`).
 - Allocate a pointer only as a global (or static) variable, and create the actual instance dynamically. The compiler will generate the correct code to invoke the constructor. This works assuming the `new` and `delete` operators have been overloaded, as described later in this chapter.
- The C++ exception handling keywords (`try`, `catch`, `throw`) do not compile. This is because the C++ exception handling mechanism requires its own runtime, which is not present in the kernel. Exception handling can only be done using *Structured Exception Handling* (SEH) - a kernel mechanism to handle exceptions. We'll take a detailed look at SEH in chapter 6.
- The standard C++ libraries are not available in the kernel. Although most are template-based, these do not compile, because they may depend on user-mode libraries and semantics. That said, C++ templates as a language feature work just fine. One good usage of templates is to create alternatives for a kernel-mode library types, based on similar types from the user-mode standard C++ library, such as `std::vector<>`, `std::wstring`, etc.

The code examples in this book make some use of C++. The features mostly used in the code examples are:

- The `nullptr` keyword, representing a true `NULL` pointer.
- The `auto` keyword that allows type inference when declaring and initializing variables. This is useful to reduce clutter, save some typing, and focus on the important pieces.
- Templates will be used where they make sense.
- Overloading of the `new` and `delete` operators.
- Constructors and destructors, especially for building RAII types.

Any C++ standard can be used for kernel development. The Visual Studio setting for new projects is to use C++ 14. However, you can change the C++ compiler standard to any other setting, including C++ 20 (the latest standard as of this writing). Some features we'll use later will depend on C++ 17 at least.

Strictly speaking, kernel drivers can be written in pure C without any issues. If you prefer to go that route, use files with a C extension rather than CPP. This will automatically invoke the C compiler for these files.

Testing and Debugging

With user-mode code, testing is generally done on the developer's machine (if all required dependencies can be satisfied). Debugging is typically done by attaching the debugger (Visual Studio in most cases) to the running process or launching an executable and attaching to the process.

With kernel code, testing is typically done on another machine, usually a virtual machine hosted on the developer's machine. This ensures that if a BSOD occurs, the developer's machine is unaffected. Debugging kernel code must be done with another machine, where the actual driver is executing. This is because hitting a breakpoint in kernel-mode freezes the entire machine, not just a particular process. The developer's machine hosts the debugger itself, while the second machine (again, usually a virtual machine) executes the driver code. These two machines must be connected through some mechanism so data can flow between the host (where the debugger is running) and the target. We'll look at kernel debugging in more detail in chapter 5.

Debug vs. Release Builds

Just like with user-mode projects, building kernel drivers can be done in Debug or Release mode. The differences are similar to their user-mode counterparts - Debug builds use no compiler optimizations by default, but are easier to debug. Release builds utilize full compiler optimizations by default to produce the fastest and smallest code possible. There are a few differences, however.

The terms in kernel terminology are *Checked* (Debug) and *Free* (Release). Although Visual Studio kernel projects continue to use the Debug/Release terms, older documentation uses the Checked/Free terms. From a compilation perspective, kernel Debug builds define the symbol `DBG` and set its value to 1

(compared to the `_DEBUG` symbol defined in user mode). This means you can use the `DBG` symbol to distinguish between Debug and Release builds with conditional compilation. This is, for example, what the `KdPrint` macro does: in Debug builds, it compiles to calling `DbgPrint`, while in Release builds it compiles to nothing, resulting in `KdPrint` calls having no effect in Release builds. This is usually what you want because these calls are relatively expensive. We'll discuss other ways of logging information in chapter 5.

The Kernel API

Kernel drivers use exported functions from kernel components. These functions will be referred to as the *Kernel API*. Most functions are implemented within the kernel module itself (*NtOskrnl.exe*), but some may be implemented by other kernel modules, such as the HAL (*hal.dll*).

The Kernel API is a large set of C functions. Most of these start with a prefix suggesting the component implementing that function. Table 3-2 shows some of the common prefixes and their meaning:

Table 3-2: Common kernel API prefixes

Prefix	Meaning	Example
Ex	General executive functions	ExAllocatePoolWithTag
Ke	General kernel functions	KeAcquireSpinLock
Mm	Memory manager	MmProbeAndLockPages
Rtl	General runtime library	RtlInitUnicodeString
FsRtl	file system runtime library	FsRtlGetFileSize
Flt	file system mini-filter library	FltCreateFile
Ob	Object manager	ObReferenceObject
Io	I/O manager	IoCompleteRequest
Se	Security	SeAccessCheck
Ps	Process manager	PsLookupProcessByProcessId
Po	Power manager	PoSetSystemState
Wmi	Windows management instrumentation	WmiTraceMessage
Zw	Native API wrappers	ZwCreateFile
Hal	Hardware abstraction layer	HalExamineMBR
Cm	Configuration manager (registry)	CmRegisterCallbackEx

If you take a look at the exported functions list from *NtOsKrnI.exe*, you'll find many functions that are not documented in the Windows Driver Kit; this is just a fact of a kernel developer's life - not everything is documented.

One set of functions bears discussion at this point - the *Zw* prefixed functions. These functions mirror native APIs available as gateways from *NtDll.Dll* with the actual implementation provided by the Executive. When an *Nt* function is called from user mode, such as *NtCreateFile*, it reaches the Executive at the actual *NtCreateFile* implementation. At this point, *NtCreateFile* might do various checks based on the fact that the original caller is from user mode. This caller information is stored on a thread-by-thread basis, in the undocumented *PreviousMode* member in the *KTHREAD* structure for each thread.

You can query the previous processor mode by calling the documented *ExGetPreviousMode* API.

On the other hand, if a kernel driver needs to call a system service, it should not be subjected to the same checks and constraints imposed on user-mode callers. This is where the Zw functions come into play. Calling a Zw function sets the previous caller mode to `KernelMode` (0) and then invokes the native function. For example, calling `ZwCreateFile` sets the previous caller to `KernelMode` and then calls `NtCreateFile`, causing `NtCreateFile` to bypass some security and buffer checks that would otherwise be performed. The bottom line is that kernel drivers should call the Zw functions unless there is a compelling reason to do otherwise.

Functions and Error Codes

Most kernel API functions return a status indicating success or failure of an operation. This is typed as `NTSTATUS`, a signed 32-bit integer. The value `STATUS_SUCCESS` (0) indicates success. A negative value indicates some kind of error. You can find all the defined `NTSTATUS` values in the file `<ntstatus.h>`.

Most code paths don't care about the exact nature of the error, and so testing the most significant bit is enough to find out whether an error occurred. This can be done with the `NT_SUCCESS` macro. Here is an example that tests for failure and logs an error if that is the case:

```
NTSTATUS DoWork() {
    NTSTATUS status = CallSomeKernelFunction();
    if(!NT_SUCCESS(status)) {
        KdPrint(L"Error occurred: 0x%08X\n", status);
        return status;
    }

    // continue with more operations

    return STATUS_SUCCESS;
}
```

In some cases, `NTSTATUS` values are returned from functions that eventually bubble up to user mode. In these cases, the `STATUS_xxx` value is translated to some `ERROR_yyy` value that is available to user-mode through the `GetLastError` function. Note that these are not the same numbers; for one, error codes in user-mode have positive values (zero is still success). Second, the mapping is not one-to-one. In any case, this is not generally a concern for a kernel driver.

Internal kernel driver functions also typically return `NTSTATUS` to indicate their success/failure status. This is usually convenient, as these functions make calls to kernel APIs and so can propagate any error by simply returning the same status they got back from the particular API. This also implies that the “real” return values from driver functions is typically returned through pointers or references provided as arguments to the function.



Return `NTSTATUS` from your own functions. It will make it easier and consistent to report errors.

Strings

The kernel API uses strings in many scenarios as needed. In some cases, these strings are simple Unicode pointers (`wchar_t*` or one of their typedefs such as `WCHAR*`), but most functions dealing with strings expect a structure of type `UNICODE_STRING`.

The term *Unicode* as used in this book is roughly equivalent to UTF-16, which means 2 bytes per character. This is how strings are stored internally within kernel components. *Unicode* in general is a set of standards related to character encoding. You can find more information at <https://unicode.org>.

The `UNICODE_STRING` structure represents a string with its length and maximum length known. Here is a simplified definition of the structure:

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWCH Buffer;
} UNICODE_STRING;
typedef UNICODE_STRING *PUNICODE_STRING;
typedef const UNICODE_STRING *PCUNICODE_STRING;
```

The `Length` member is in bytes (not characters) and does not include a Unicode-NULL terminator, if one exists (a NULL terminator is not mandatory). The `MaximumLength` member is the number of bytes the string can grow to without requiring a memory reallocation.

Manipulating UNICODE_STRING structures is typically done with a set of *Rtl* functions that deal specifically with strings. Table 3-3 lists some of the common functions for string manipulation provided by the *Rtl* functions.

Table 3-3: Common UNICODE_STRING functions

Function	Description
<code>RtlInitUnicodeString</code>	Initializes a UNICODE_STRING based on an existing C-string pointer. It sets <code>Buffer</code> , then calculates the <code>Length</code> and sets <code>MaximumLength</code> to the same value. Note that this function does not allocate any memory - it just initializes the internal members.
<code>RtlCopyUnicodeString</code>	Copies one UNICODE_STRING to another. The destination string pointer (<code>Buffer</code>) must be allocated before the copy and <code>MaximumLength</code> set appropriately.
<code>RtlCompareUnicodeString</code>	Compares two UNICODE_STRINGS (equal, less, greater), specifying whether to do a case sensitive or insensitive comparison.
<code>RtlEqualUnicodeString</code>	Compares two UNICODE_STRINGS for equality, with case sensitivity specification.
<code>RtlAppendUnicodeStringToString</code>	Appends one UNICODE_STRING to another.
<code>RtlAppendUnicodeToString</code>	Appends UNICODE_STRING to a C-style string.

In addition to the above functions, there are functions that work on C-string pointers. Moreover, some of the well-known string functions from the C Runtime Library are implemented within the kernel as well for convenience: `wscpy_s`, `wscat_s`, `wcslen`, `wscpy_s`, `wcschr`, `strcpy`, `strcpy_s` and others.



The *wcs* prefix works with C Unicode strings, while the *str* prefix works with C Ansi strings. The suffix *_s* in some functions indicates a *safe* function, where an additional argument indicating the maximum length of the string must be provided so the function would not transfer more data than that size.



Never use the non-safe functions. You can include `<dontuse.h>` to get errors for deprecated functions if you do use these in code.

Dynamic Memory Allocation

Drivers often need to allocate memory dynamically. As discussed in chapter 1, kernel thread stack size is rather small, so any large chunk of memory should be allocated dynamically.

The kernel provides two general memory pools for drivers to use (the kernel itself uses them as well).

- Paged pool - memory pool that can be paged out if required.
- Non-Paged Pool - memory pool that is never paged out and is guaranteed to remain in RAM.

Clearly, the non-paged pool is a “better” memory pool as it can never incur a page fault. We’ll see later in this book that some cases require allocating from non-paged pool. Drivers should use this pool sparingly, only when necessary. In all other cases, drivers should use the paged pool. The `POOL_TYPE` enumeration represents the pool types. This enumeration includes many “types” of pools, but only three should be used by drivers: `PagedPool`, `NonPagedPool`, `NonPagedPoolNx` (non-page pool without execute permissions).

Table 3-4 summarizes the most common functions used for working with the kernel memory pools.

Table 3-4: Functions for kernel memory pool allocation

Function	Description
<code>ExAllocatePool</code>	Allocate memory from one of the pools with a default tag. This function is considered obsolete. The next function in this table should be used instead
<code>ExAllocatePoolWithTag</code>	Allocate memory from one of the pools with the specified tag
<code>ExAllocatePoolZero</code>	Same as <code>ExAllocatePoolWithTag</code> , but zeroes out the memory block
<code>ExAllocatePoolWithQuotaTag</code>	Allocate memory from one of the pools with the specified tag and charge the current process quota for the allocation
<code>ExFreePool</code>	Free an allocation. The function knows from which pool the allocation was made



`ExAllocatePool` calls `ExAllocatePoolWithTag` using the tag `enon` (the word “none” in reverse). Older Windows versions used ``mdW` (`WDM` in reverse). You should avoid this function and use `ExAllocatePoolWithTag`` instead.

`ExAllocatePoolZero` is implemented inline in `wdm.h` by calling `ExAllocatePoolWithTag` and adding the `POOL_ZERO_ALLOCATION` (=1024) flag to the pool type.

Other memory management functions are covered in chapter 8, “Advanced Programming Techniques”.

The tag argument allows “tagging” an allocation with a 4-byte value. Typically this value is comprised of up to 4 ASCII characters logically identifying the driver, or some part of the driver. These tags can be used to help identify memory leaks - if any allocations tagged with the driver’s tag remain after the driver is unloaded. These pool allocations (with their tags) can be viewed with the *Poolmon* WDK tool, or my own *PoolMonXv2* tool (downloadable from

<http://www.github.com/zodiacon/AllTools>). Figure 3-1 shows a screenshot of *PoolMonXv2*.

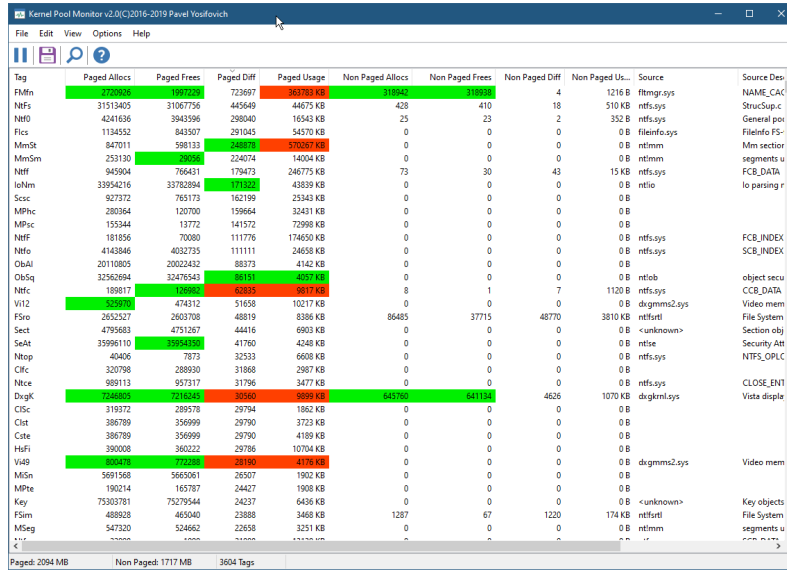


Figure 3-1: *PoolMonXv2*



You must use tags comprised of printable ASCII characters. Otherwise, running the driver under the control of the *Driver Verifier* (described in chapter 11) would lead to *Driver Verifier* complaining.

The following code example shows memory allocation and string copying to save the registry path passed to `DriverEntry`, and freeing that string in the `Unload` routine:

```
// define a tag (because of little endianness, viewed as 'abcd')
#define DRIVER_TAG 'dcba'

UNICODE_STRING g_RegistryPath;

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(DriverObject);
    DriverObject->DriverUnload = SampleUnload;

    g_RegistryPath.Buffer = (WCHAR*)ExAllocatePoolWithTag(PagedPool,
        RegistryPath->Length, DRIVER_TAG);
    if (g_RegistryPath.Buffer == nullptr) {
        KdPrint(("Failed to allocate memory\n"));
    }
}
```

```

    return STATUS_INSUFFICIENT_RESOURCES;
}

g_RegistryPath.MaximumLength = RegistryPath->Length;
RtlCopyUnicodeString(&g_RegistryPath,
    (PCUNICODE_STRING)RegistryPath);

// %wZ is for UNICODE_STRING objects
KdPrint(("Original registry path: %wZ\n", RegistryPath));
KdPrint(("Copied registry path: %wZ\n", &g_RegistryPath));
//...
return STATUS_SUCCESS;
}

void SampleUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);

    ExFreePool(g_RegistryPath.Buffer);
    KdPrint(("Sample driver Unload called\n"));
}

```

Linked Lists

The kernel uses circular doubly linked lists in many of its internal data structures. For example, all processes on the system are managed by EPROCESS structures, connected in a circular doubly linked list, where its head is stored the kernel variable PsActiveProcessHead.

All these lists are built in the same way, centered around the LIST_ENTRY structure defined like so:

```

typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY;

```

Figure 3-2 depicts an example of such a list containing a head and three instances.

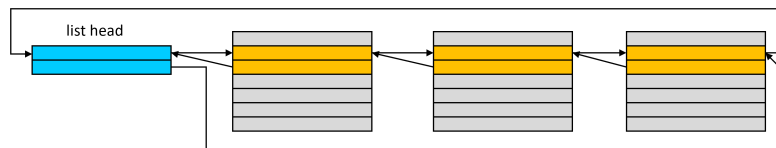


Figure 3-2: Circular linked list

One such structure is embedded inside the real structure of interest. For example, in the EPROCESS structure, the member ActiveProcessLinks is of type LIST_ENTRY, pointing to the next and previous LIST_ENTRY objects of other EPROCESS structures. The head of a list is stored separately; in the case of the process, that's PsActiveProcessHead.

To get the pointer to the actual structure of interest given the address of a LIST_ENTRY can be obtained with the CONTAINING_RECORD macro.

For example, suppose you want to manage a list of structures of type *MyDataItem* defined like so:

```
struct MyDataItem {  
    // some data members  
    LIST_ENTRY Link;  
    // more data members  
};
```

When working with these linked lists, we have a head for the list, stored in a variable. This means that natural traversal is done by using the FLink member of the list to point to the next LIST_ENTRY in the list. Given a pointer to the LIST_ENTRY, what we're really after is the MyDataItem that contains this list entry member. This is where the CONTAINING_RECORD comes in:

```
MyDataItem* GetItem(LIST_ENTRY* pEntry) {  
    return CONTAINING_RECORD(pEntry, MyDataItem, Link);  
}
```

The macro does the proper offset calculation and does the casting to the actual data type (*MyDataItem* in the example).

Table 3-5 shows the common functions for working with these linked lists. All operations use constant time.

Table 3-5: Functions for working with circular linked lists

Function	Description
<code>InitializeListHead</code>	Initializes a list head to make an empty list. The forward and back pointers point to the forward pointer.
<code>InsertHeadList</code>	Insert an item to the head of the list.
<code>InsertTailList</code>	Insert an item to the tail of the list.
<code>IsListEmpty</code>	Check if the list is empty.
<code>RemoveHeadList</code>	Remove the item at the head of the list.
<code>RemoveTailList</code>	Remove the item at the tail of the list.
<code>RemoveEntryList</code>	Remove a specific item from the list.
<code>ExInterlockedInsertHeadList</code>	Insert an item at the head of the list atomically by using the specified spinlock.
<code>ExInterlockedInsertTailList</code>	Insert an item at the tail of the list atomically by using the specified spinlock.
<code>ExInterlockedRemoveHeadList</code>	Remove an item from the head of the list atomically by using the specified spinlock.

The last three functions in table 3-4 perform the operation atomically using a synchronization primitive called a *spin lock*. Spin locks are discussed in chapter 6.

The Driver Object

We've already seen that the `DriverEntry` function accepts two arguments, the first is a driver object of some kind. This is a semi-documented structure called `DRIVER_OBJECT` defined in the WDK headers. "Semi-documented" means that some of its members are documented for driver's use and some are not. This structure is allocated by the kernel and partially initialized. Then it's provided to `DriverEntry` (and before the driver unloads to the `Unload` routine as well). The role of the driver at this point is to further initialize the structure to indicate what operations are supported by the driver.

We've seen one such "operation" in chapter 2 - the Unload routine. The other important set of operations to initialize are called *Dispatch Routines*. This is an array of function pointers, stored in the `MajorFunction` member of `DRIVER_OBJECT`. This set specifies which operations the driver supports, such as Create, Read, Write, and so on. These indices are defined with the *IRP_MJ_* prefix. Table 3-6 shows some common major function codes and their meaning.

Table 3-6: Common major function codes

Major function	Description
IRP_MJ_CREATE (0)	Create operation. Typically invoked for <code>CreateFile</code> or <code>ZwCreateFile</code> calls.
IRP_MJ_CLOSE (2)	Close operation. Normally invoked for <code>CloseHandle</code> or <code>ZwClose</code> .
IRP_MJ_READ (3)	Read operation. Typically invoked for <code>ReadFile</code> , <code>ZwReadFile</code> and similar read APIs.
IRP_MJ_WRITE (4)	Write operation. Typically invoked for <code>WriteFile</code> , <code>ZwWriteFile</code> , and similar write APIs.
IRP_MJ_DEVICE_CONTROL (14)	Generic call to a driver, invoked because of <code>DeviceIoControl</code> or <code>ZwDeviceIoControlFile</code> calls.
IRP_MJ_INTERNAL_DEVICE_CONTROL (15)	Similar to the previous one, but only available for kernel-mode callers.
IRP_MJ_SHUTDOWN (16)	Called when the system shuts down if the driver has registered for shutdown notification with <code>IoRegisterShutdownNotification</code> .
IRP_MJ_CLEANUP (18)	Invoked when the last handle to a file object is closed, but the file object's reference count is not zero.
IRP_MJ_PNP (31)	Plug and play callback invoked by the Plug and Play Manager.
IRP_MJ_POWER (22)	Generally interesting for hardware-based drivers or filters to such drivers. Power callback invoked by the Power Manager. Generally interesting for hardware-based drivers or filters to such drivers.

Initially, the `MajorFunction` array is initialized by the kernel to point to a kernel internal routine, `IopInvalidDeviceRequest`, which returns a failure status to the caller, indicating the operation is not supported. This means the driver, in its

DriverEntry routine only needs to initialize the actual operations it supports, leaving all the other entries in their default values.

For example, our Sample driver at this point does not support any dispatch routines, which means there is no way to communicate with the driver. A driver must at least support the IRP_MJ_CREATE and IRP_MJ_CLOSE operations, to allow opening a handle to one of the device objects for the driver. We'll put these ideas into practice in the next chapter.

Object Attributes

One of the common structures that shows up in many kernel APIs is OBJECT_ATTRIBUTES, defined like so:

```
typedef struct _OBJECT_ATTRIBUTES {
    ULONG Length;
    HANDLE RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG Attributes;
    PVOID SecurityDescriptor;           // SECURITY_DESCRIPTOR
    PVOID SecurityQualityOfService;     // SECURITY_QUALITY_OF_SERVICE
} OBJECT_ATTRIBUTES;
typedef OBJECT_ATTRIBUTES *POBJECT_ATTRIBUTES;
typedef CONST OBJECT_ATTRIBUTES *PCOBJECT_ATTRIBUTES;
```

The structure is typically initialized with the InitializeObjectAttributes macro, that allows specifying all the structure members except Length (set automatically by the macro), and SecurityQualityOfService, which is not normally needed. Here is the description of the members:

- ObjectName is the name of the object to be created/located, provided as a pointer to a UNICODE_STRING. In some cases it may be ok to set it to NULL. For example, the ZwOpenProcess allows opening a handle to a process given its PID. Since processes don't have names, the ObjectName in this case should be initialized to NULL.
- RootDirectory is an optional directory in the object manager namespace if the name of the object is relative one. If ObjectName specifies a fully-qualified name, RootDirectory should be set to NULL.
- Attributes allows specifying a set of flags that has effect on the operation in question. Table 3-7 shows the defined flags and their meaning.
- SecurityDescriptor is an optional security descriptor (SECURITY_DESCRIPTOR) to set on the newly created object. NULL indicates

the new object gets a default security descriptor, based on the caller's token.

- `SecurityQualityOfService` is an optional set of attributes related to the new object's impersonation level and context tracking mode. It has no meaning for most object types. Consult the documentation for more information.

Table 3-7: Object attributes flags

Flag (OBJ_)	Description
INHERIT (2)	The returned handle should be marked as inheritable
PERMANENT (0x10)	The object created should be marked as permanent. Permanent objects have an additional reference count that prevents them from dying even if all handles to them are closed
EXCLUSIVE (0x20)	If creating an object, the object is created with exclusive access. No other handles can be opened to the object. If opening an object, exclusive access is requested, which is granted only if the object was originally created with this flag
CASE_INSENSITIVE (0x40)	When opening an object, perform a case insensitive search for its name. Without this flag, the name must match exactly
OPENIF (0x80)	Open the object if it exists. Otherwise, fail the operation (don't create a new object)
OPENLINK (0x100)	If the object to open is a symbolic link object, open the symbolic link object itself, rather than following the symbolic link to its target
KERNEL_HANDLE (0x200)	The returned handle should be a kernel handle. Kernel handles are valid in any process context, and cannot be used by user mode code
FORCE_ACCESS_CHECK (0x400)	Access checks should be performed even if the object is opened in KernelMode access mode
IGNORE_IMPERSONATED_DEVICEMAP (0x800)	Use the process device map instead of the user's if it's impersonating (consult the documentation for more information on device maps)

Flag (OBJ_)	Description
DONT_REPARSE (0x1000)	Don't follow a reparse point, if encountered. Instead an error is returned (STATUS_REPARSE_POINT_ENCOUNTED). <i>Reparse points</i> are briefly discussed in chapter 11

A second way to initialize an OBJECT_ATTRIBUTES structure is available with the RTL_CONSTANT_OBJECT_ATTRIBUTES macro, that uses the most common members to set - the object's name and the attributes.

Let's look at a couple of examples that use OBJECT_ATTRIBUTES. The first one is a function that opens a handle to a process given its process ID. For this purpose, we'll use the ZwOpenProcess API, defined like so:

```
NTSTATUS ZwOpenProcess (
    _Out_ PHANDLE ProcessHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_opt_ PCLIENT_ID ClientId);
```

It uses yet another common structure, CLIENT_ID that holds a process and/or a thread ID:

```
typedef struct _CLIENT_ID {
    HANDLE UniqueProcess; // PID, not handle
    HANDLE UniqueThread;  // TID, not handle
} CLIENT_ID;
typedef CLIENT_ID *PCLIENT_ID;
```

To open a process, we need to specify the process ID in the UniqueProcess member. Note that although the type of UniqueProcess is HANDLE, it is the unique ID of the process. The reason for the HANDLE type is that process and thread IDs are generated from a private handle table. This also explains why process and thread IDs are always multiple of four (just like normal handles), and why they don't overlap.

With these details at hand, here is a process opening function:

```
NTSTATUS
OpenProcess(ACCESS_MASK accessMask, ULONG pid, PHANDLE phProcess) {
    CLIENT_ID cid;
    cid.UniqueProcess = ULongToHandle(pid);
```

```

cid.UniqueThread = nullptr;

OBJECT_ATTRIBUTES procAttributes =
    RTL_CONSTANT_OBJECT_ATTRIBUTES(nullptr, OBJ_KERNEL_HANDLE);
return ZwOpenProcess(phProcess, accessMask, &procAttributes, &cid);
}

```

The `ULONGToHandle` function performs the required casts so that the compiler is happy (`HANDLE` is 64-bit on a 64-bit system, but `ULONG` is always 32-bit). The only member used in the above code from `OBJECT_ATTRIBUTES` is the `Attributes` flags.

The second example is a function that opens a handle to a file for read access, by using the `ZwOpenFile` API, defined like so:

```

NTSTATUS ZwOpenFile(
    _Out_ PHANDLE FileHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_ ULONG ShareAccess,
    _In_ ULONG OpenOptions);

```

A full discussion of the parameters to `ZwOpenFile` is reserved for chapter 11, but one thing is obvious: the file name itself is specified using the `OBJECT_ATTRIBUTES` structure - there is no separate parameter for that. Here is the full function opening a handle to a file for read access:

```

NTSTATUS OpenFileForRead(PCWSTR path, PHANDLE phFile) {
    UNICODE_STRING name;
    RtlInitUnicodeString(&name, path);

    OBJECT_ATTRIBUTES fileAttributes;
    InitializeObjectAttributes(&fileAttributes, &name,
        OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, nullptr, nullptr);
    IO_STATUS_BLOCK ioStatus;
    return ZwOpenFile(phFile, FILE_GENERIC_READ,
        &fileAttributes, &ioStatus, FILE_SHARE_READ, 0);
}

```

`InitializeObjectAttributes` is used to initialize the `OBJECT_ATTRIBUTES` structure, although the `RTL_CONSTANT_OBJECT_ATTRIBUTES` could have been used just as well, since we're only specifying the name and attributes. Notice the need to turn the passed-in NULL-terminated C-string pointer into a `UNICODE_STRING` with `RtlInitUnicodeString`.

Device Objects

Although a driver object may look like a good candidate for clients to talk to, this is not the case. The actual communication endpoints for clients are device objects. Device objects are instances of the semi-documented `DEVICE_OBJECT` structure. Without device objects, there is no one to talk to. This means that at least one device object should be created by the driver and given a name, so that it may be contacted by clients.

The `CreateFile` function (and its variants) accepts a first argument which is called “file name” in the documentation, but really this should point to a device object’s name, where an actual file system file is just one particular case. The name `CreateFile` is somewhat misleading - the word “file” here means “file object”. Opening a handle to a file or device creates an instance of the kernel structure `FILE_OBJECT`, another semi-documented structure.

More precisely, `CreateFile` accepts a *symbolic link*, a kernel object that knows how to point to another kernel object. (You can think of a symbolic link as similar in principle to a file system shortcut.) All the symbolic links that can be used from the user mode `CreateFile` or `CreateFile2` calls are located in the *Object Manager* directory named `??`. You can see the contents of this directory with the *Sysinternals WinObj* tool. Figure 3-3 shows this directory (named *Global??* in *WinObj*).

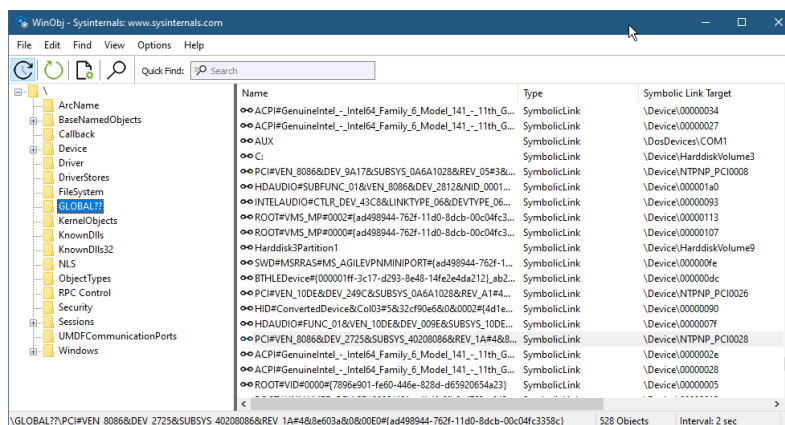


Figure 3-3: Symbolic links directory in *WinObj*

Some of the names seem familiar, such as *C:*, *Aux*, *Con*, and others. Indeed, these are valid “file names” for `CreateFile` calls. Other entries look like long

cryptic strings, and these in fact are generated by the I/O system for hardware-based drivers that call the `IoRegisterDeviceInterface` API. These types of symbolic links are not useful for the purpose of this book.

Most of the symbolic links in the `\??` directory point to an internal device name under the `\Device` directory. The names in this directory are not directly accessible by user-mode callers. But they can be accessed by kernel callers using the `IoGetDeviceObjectPointer` API.

A canonical example is the driver for *Process Explorer*. When *Process Explorer* is launched with administrator rights, it installs a driver. This driver gives *Process Explorer* powers beyond those that can be obtained by user-mode callers, even if running elevated. For example, *Process Explorer* in its *Threads* dialog for a process can show the complete call stack of a thread, including functions in kernel mode. This type of information is not possible to obtain from user mode; its driver provides the missing information.

The driver installed by *Process Explorer* creates a single device object so that *Process Explorer* is able to open a handle to that device and make requests. This means that the device object must be named, and must have a symbolic link in the `??` directory; and it's there, called `PROCEXP152`, probably indicating driver version 15.2 (at the time of writing). Figure 3-4 shows this symbolic link in *WinObj*.

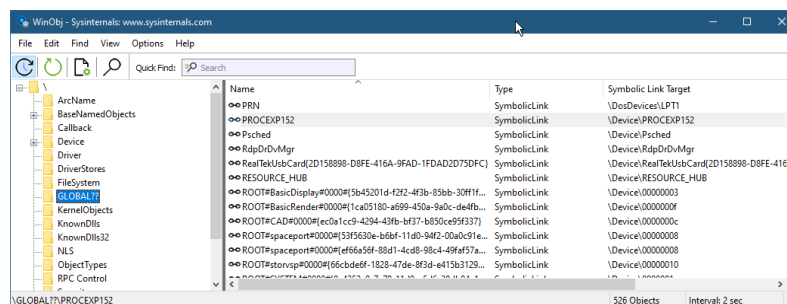


Figure 3-4: *Process Explorer*'s symbolic link in *WinObj*

Notice the symbolic link for *Process Explorer*'s device points to `\Device\PROCEXP152`, which is the internal name only accessible to kernel callers (and the native APIs `NtOpenFile` and `NtCreateFile`, as shown in the next section). The actual `CreateFile` call made by *Process Explorer* (or any other client) based on the symbolic link must be prepended with `\\.\.`. This is necessary so that the I/O manager's parser will not assume the string

“PROCEXP152” refers to a file with no extension in the current directory. Here is how *Process Explorer* would open a handle to its device object (note the double backslashes because of the backslash being an escape character in C/C++):

```
HANDLE hDevice = CreateFile(L"\\\\.\\PROCEXP152",  
    GENERIC_WRITE | GENERIC_READ, 0, nullptr, OPEN_EXISTING,  
    0, nullptr);
```



With C++ 11 and later, you can write strings without escaping the backslash character. The device path in the above code can be written like so: `LR"(\\.\PROCEXP152)"`. `L` indicates Unicode (as always), while anything between `R"(` and `)"` is not escaped.

You can try the above code yourself. If *Process Explorer* has run elevated at least once on the system since boot, its driver should be running (you can verify with the tool itself), and the call to `CreateFile` will succeed if the client is running elevated.

A driver creates a device object using the `IoCreateDevice` function. This function allocates and initializes a device object structure and returns its pointer to the caller. The device object instance is stored in the `DeviceObject` member of the `DRIVER_OBJECT` structure. If more than one device object is created, they form a singly linked list, where the member `NextDevice` of the `DEVICE_OBJECT` points to the next device object. Note that the device objects are inserted at the head of the list, so the first device object created is stored last; its `NextDevice` points to `NULL`. These relationships are depicted in figure 3-5.

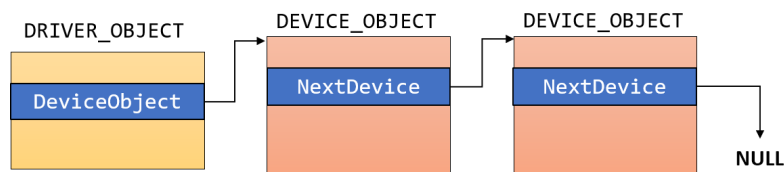


Figure 3-5: Driver and Device objects

Opening Devices Directly

The existence of a symbolic link makes it easy to open a handle to a device with the documented `CreateFile` user-mode API (or from the `ZwOpenFile` API in the kernel). It is sometimes useful, however, to be able to open device objects without going through a symbolic link. For example, a device object might not have a symbolic link, because its driver decided (for whatever reason) not to provide one.

The native `NtOpenFile` (and `NtCreateFile`) function can be used to open a device object directly. Microsoft never recommends using native APIs, but this function is somewhat documented for user-mode use. Its definition is available in the `<Winternl.h>` header file:

```
NTAPI NtOpenFile (  
    OUT PHANDLE FileHandle,  
    IN  ACCESS_MASK DesiredAccess,  
    IN  POBJECT_ATTRIBUTES ObjectAttributes,  
    OUT PIO_STATUS_BLOCK IoStatusBlock,  
    IN  ULONG ShareAccess,  
    IN  ULONG OpenOptions);
```

Notice the similarity to the `ZwOpenFile` we used in an earlier section - this is the same function prototype, just invoked here from user mode, eventually to land at `NtOpenFile` within the I/O manager. The function requires usage of an `OBJECT_ATTRIBUTES` structure, described earlier in this chapter.

The above prototype uses old macros such as `IN`, `OUT` and others. These have been replaced by SAL annotations. Unfortunately, some header files were not yet converted to SAL.

To demonstrate using `NtOpenFile` from user mode, we'll create an application to play a single sound. Normally, the `Beep` Windows user-mode API provides such a service:

```
BOOL Beep(  
    _In_ DWORD dwFreq,  
    _In_ DWORD dwDuration);
```

The function accepts the frequency to play (in Hertz), and the duration to play, in milliseconds. The function is synchronous, meaning it does not return until the duration has elapsed.

The Beep API works by calling a device named `\Device\Beep` (you can find it in `WinObj`), but the beep device driver does not create a symbolic link for it. However, we can open a handle to the beep device using `NtOpenFile`. Then, to play a sound, we can use the `DeviceIoControl` function with the correct parameters. Although it's not too difficult to reverse engineer the beep driver workings, fortunately we don't have to. The SDK provides the `<ntddbeep.h>` file with the required definitions, including the device name itself.

We'll start by creating a C++ Console application in Visual Studio. Before we get to the main function, we need some `#includes`:

```
#include <windows.h>
#include <winternl.h>
#include <stdio.h>
#include <ntddbeep.h>
```

`<winternl.h>` provides the definition for `NtOpenFile` (and related data structures), while `<ntddbeep.h>` provides the beep-specific definitions.

Since we will be using `NtOpenFile`, we must also link against `Ntdll.Dll`, which we can do by adding a `#pragma` to the source code, or add the library to the linker settings in the project's properties. Let's go with the former, as it's easier, and is not tied to the project's properties:

```
#pragma comment(lib, "ntdll")
```



Without the above linkage, the linker would issue an “unresolved external” error.

Now we can start writing `main`, where we accept optional command line arguments indicating the frequency and duration to play:

```
int main(int argc, const char* argv[]) {
    printf("beep [<frequency> <duration_in_msec>]\n");
    int freq = 800, duration = 1000;
    if (argc > 2) {
        freq = atoi(argv[1]);
        duration = atoi(argv[2]);
    }
}
```

The next step is to open the device handle using `NtOpenFile`:

```
HANDLE hFile;
OBJECT_ATTRIBUTES attr;
UNICODE_STRING name;
RtlInitUnicodeString(&name, L"\\Device\\Beep");
InitializeObjectAttributes(&attr, &name, OBJ_CASE_INSENSITIVE,
    nullptr, nullptr);
IO_STATUS_BLOCK ioStatus;
NTSTATUS status = ::NtOpenFile(&hFile, GENERIC_WRITE, &attr, &ioStatus, 0, 0);
```

The line to initialize the device name can be replaced with:

```
RtlInitUnicodeString(&name, DD_BEEP_DEVICE_NAME_U);
```

The `DD_BEEP_DEVICE_NAME_U` macro is conveniently supplied as part of `<ntddbeep.h>`.

If the call succeeds, we can play the sound. To do that, we call `DeviceIoControl` with a control code defined in `<ntddbeep.h>` and use a structure defined there as well to fill in the frequency and duration:

```
if (NT_SUCCESS(status)) {
    BEEP_SET_PARAMETERS params;
    params.Frequency = freq;
    params.Duration = duration;
    DWORD bytes;
    //
    // play the sound
    //
    printf("Playing freq: %u, duration: %u\n", freq, duration);
    ::DeviceIoControl(hFile, IOCTL_BEEP_SET, &params, sizeof(params),
        nullptr, 0, &bytes, nullptr);

    //
    // the sound starts playing and the call returns immediately
    // wait so that the app doesn't close
    //
    ::Sleep(duration);
    ::CloseHandle(hFile);
}
```

The input buffer passed to `DeviceIoControl` should be a `BEEP_SET_PARAMETERS` structure, which we pass in along with its size. The last piece of the puzzle is to use the `Sleep` API to wait based on the duration, otherwise the handle to the device would be closed and the sound cut off.



Write an application that plays an array of sounds by leveraging the above code.

Summary

In this chapter, we looked at some of the fundamental kernel data structures, concepts, and APIs. In the next chapter, we'll build a complete driver, and a client application, expanding on the information presented thus far.

Chapter 4: Driver from Start to Finish

In this chapter, we'll use many of the concepts we learned in previous chapters and build a simple, yet complete, driver, and an associated client application, while filling in some of the missing details from previous chapters. We'll deploy the driver and use its capabilities - perform some operation in kernel mode that is difficult, or impossible to do, in user mode.

In this chapter:

- **Introduction**
 - **Driver Initialization**
 - **Client Code**
 - **The Create and Close Dispatch Routines**
 - **The Write Dispatch Routine**
 - **Installing and Testing**
-

Introduction

The problem we'll solve with a simple kernel driver is the inflexibility of setting thread priorities using the Windows API. In user mode, a thread's priority is determined by a combination of its process *Priority Class* with an offset on a per thread basis, that has a limited number of levels.

Changing a process *priority class* (shown as *Base priority* column in *Task Manager*) can be achieved with the `SetPriorityClass` function that accepts a process handle and one of the six supported priority classes. Each priority class corresponds to a priority level, which is the default priority for threads created in that process. A particular thread's priority can be changed with the `SetThreadPriority` function, accepting a thread handle and one of several constants corresponding to offsets around the base priority class. Table 4-1

shows the available thread priorities based on the process priority class and the thread's priority offset.

Table 4-1: Legal values for thread priorities with the Windows APIs

Priority Class	- Sat	-2	-1	0 (default)	+1	+2	+ Sat	Comments
Idle	1	2	3	4	5	6	15	<i>Task Manager</i> refers to <i>Idle</i> as “ <i>Low</i> ”
Below Normal	1	4	5	6	7	8	15	
Normal	1	6	7	8	9	10	15	
Above Normal	1	8	9	10	11	12	15	
High	1	11	12	13	14	15	15	Only six levels are available (not seven).
Real-time	16	22	23	24	25	26	31	All levels between 16 to 31 can be selected.

The values acceptable to `SetThreadPriority` specify the offset. Five levels correspond to the offsets -2 to +2: `THREAD_PRIORITY_LOWEST` (-2), `THREAD_PRIORITY_BELOW_NORMAL` (-1), `THREAD_PRIORITY_NORMAL` (0), `THREAD_PRIORITY_ABOVE_NORMAL` (+1), `THREAD_PRIORITY_HIGHEST` (+2). The remaining two levels, called *Saturation* levels, set the priority to the two extremes supported by that priority class: `THREAD_PRIORITY_IDLE` (-Sat) and `THREAD_PRIORITY_TIME_CRITICAL` (+Sat).

The following code example changes the current thread's priority to 11:

```

SetPriorityClass(GetCurrentProcess(),
    ABOVE_NORMAL_PRIORITY_CLASS);    // process base=10
SetThreadPriority(GetCurrentThread(),
    THREAD_PRIORITY_ABOVE_NORMAL);    // +1 offset for thread

```



The Real-time priority class does not imply Windows is a real-time OS; Windows does not provide some of the timing guarantees normally provided by true real-time operating systems. Also, since Real-time priorities are very high and compete with many kernel threads doing important work, such a process must be running with administrator privileges; otherwise, attempting to set the priority class to Real-time causes the value to be set to High.

There are other differences between the real-time priorities and the lower priority classes. Consult the *Windows Internals* book for more information.

Table 4-1 shows the problem we will address quite clearly. Only a small set of priorities are available to set directly. We would like to create a driver that would circumvent these limitations and allow setting a thread's priority to any number, regardless of its process priority class.

Driver Initialization

We'll start building the driver in the same way we did in chapter 2. Create a new "WDM Empty Project" named *Booster* (or another name of your choosing) and delete the INF file created by the wizard. Next, add a new source file to the project, called *Booster.cpp* (or any other name you prefer). Add the basic `#include` for the main WDK header and an almost empty `DriverEntry`:

```
#include <ntddk.h>

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    return STATUS_SUCCESS;
}
```

Most software drivers need to do the following in `DriverEntry`:

- Set an *Unload* routine.
- Set dispatch routines the driver supports.
- Create a device object.
- Create a symbolic link to the device object.

Once all these operations are performed, the driver is ready to take requests.

The first step is to add an Unload routine and point to it from the driver object. Here is the new DriverEntry with the Unload routine:

```
// prototypes

void BoosterUnload(PDRIVER_OBJECT DriverObject);

// DriverEntry

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    DriverObject->DriverUnload = BoosterUnload;

    return STATUS_SUCCESS;
}

void BoosterUnload(PDRIVER_OBJECT DriverObject) {
    // empty for now
}
```

We'll add code to the Unload routine as needed when we do actual work in DriverEntry that needs to be undone.

Next, we need to set up the dispatch routines that we want to support. Practically all drivers must support IRP_MJ_CREATE and IRP_MJ_CLOSE, otherwise there would be no way to open a handle to any device for this driver. So we add the following to DriverEntry:

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = BoosterCreateClose;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = BoosterCreateClose;
```

We're pointing the Create and Close major functions to the same routine. This is because, as we'll see shortly, they will do the same thing: simply approve the request. In more complex cases, these could be separate functions, where in the Create case the driver can (for instance) check to see who the caller is and only let approved callers succeed with opening a handle.

All major functions have the same prototype (they are part of an array of function pointers), so we have to add a prototype for BoosterCreateClose. The prototype for these functions is as follows:

```
NTSTATUS BoosterCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp);
```

The function must return `NTSTATUS`, and accepts a pointer to a device object and a pointer to an *I/O Request Packet* (IRP). An IRP is the primary object where the request information is stored, for all types of requests. We'll dig deeper into an IRP in chapter 7, but we'll look at the basics later in this chapter, since we require it to complete our driver.

Passing Information to the Driver

The Create and Close operations we set up are required, but certainly not enough. We need a way to tell the driver which thread and to what value to set its priority. From a user mode client's perspective, there are three basic functions it can use: `WriteFile`, `ReadFile` and `DeviceIoControl`.

For our driver's purposes, we can use either `WriteFile` or `DeviceIoControl`. Read doesn't make sense, because we're passing information *to* the driver, rather than from the driver. So which is better, `WriteFile` or `DeviceIoControl`? This is mostly a matter of taste, but the general wisdom here is to use `Write` if it's really a write operation (logically); for anything else - `DeviceIoControl` is preferred, as it's a generic mechanism for passing data to and from the driver.

Since changing a thread's priority is not a purely `Write` operation, `DeviceIoControl` makes more sense, but we'll use `WriteFile`, as it's a bit easier to handle. We'll look at all the details in chapter 7. `WriteFile` has the following prototype:

```
BOOL WriteFile(  
    _In_      HANDLE hFile,  
    _In_reads_bytes_opt_(nNumberOfBytesToWrite) LPCVOID lpBuffer,  
    _In_      DWORD nNumberOfBytesToWrite,  
    _Out_opt_ LPDWORD lpNumberOfBytesWritten,  
    _Inout_opt_ LPOVERLAPPED lpOverlapped);
```

Our driver has to export its handling of a write operation capability by assigning a function pointer to the `IRP_MJ_WRITE` index of the `MajorFunction` array in the driver object:

```
DriverObject->MajorFunction[IRP_MJ_WRITE] = BoosterWrite;
```

`BoosterWrite` must have the same prototype as all major function code handlers:

```
NTSTATUS BoosterWrite(PDEVICE_OBJECT DeviceObject, PIRP Irp);
```

Client / Driver Communication Protocol

Given that we use `writeFile` for client/driver communication, we now must define the actual semantics. `writeFile` allows passing in a buffer, for which we need to define proper semantics. This buffer should contain the two pieces of information required so the driver can do its thing: the thread id and the priority to set for it.

These pieces of information must be usable both by the driver and the client. The client would supply the data, and the driver would act on it. This means these definitions must be in a separate file that must be included by both the driver and client code.

For this purpose, we'll add a header file named *BoosterCommon.h* to the driver project. This file will also be used later by the user-mode client.

Within this file, we need to define the data structure to pass to the driver in the `writeFile` buffer, containing the thread ID and the priority to set:

```
struct ThreadData {  
    ULONG ThreadId;  
    int Priority;  
};
```

We need the thread's unique ID and the target priority. Thread IDs are 32-bit unsigned integers, so we select `ULONG` as the type. The priority should be a number between 1 and 31, so a simple 32-bit integer will do.

We cannot normally use `DWORD` - a common type defined in user mode headers - because it's not defined in kernel mode headers. `ULONG`, on the other hand, is defined in both. It would be easy enough to define it ourselves, but `ULONG` is the same anyway.

Creating the Device Object

We have more initializations to do in `DriverEntry`. Currently, we don't have any device object and so there is no way to open a handle and reach the driver. A typical software driver needs just one device object, with a symbolic link

pointing to it, so that user-mode clients can obtain handles easily with `CreateFile`.

Creating the device object requires calling the `IoCreateDevice` API, declared as follows (some SAL annotations omitted/simplified for clarity):

```
NTSTATUS IoCreateDevice(  
    _In_      PDRIVER_OBJECT DriverObject,  
    _In_      ULONG DeviceExtensionSize,  
    _In_opt_  PUNICODE_STRING DeviceName,  
    _In_      DEVICE_TYPE DeviceType,  
    _In_      ULONG DeviceCharacteristics,  
    _In_      BOOLEAN Exclusive,  
    _Outptr_  PDEVICE_OBJECT *DeviceObject);
```

The parameters to `IoCreateDevice` are described below:

- *DriverObject* - the driver object to which this device object belongs to. This should be simply the driver object passed to the `DriverEntry` function.
- *DeviceExtensionSize* - extra bytes that would be allocated in addition to `sizeof(DEVICE_OBJECT)`. Useful for associating some data structure with a device. It's less useful for software drivers creating just a single device object, since the state needed for the device can simply be managed by global variables.
- *DeviceName* - the internal device name, typically created under the `\Device Object Manager` directory.
- *DeviceType* - relevant to some type of hardware-based drivers. For software drivers, the value `FILE_DEVICE_UNKNOWN` should be used.
- *DeviceCharacteristics* - a set of flags, relevant for some specific drivers. Software drivers specify zero or `FILE_DEVICE_SECURE_OPEN` if they support a true namespace (rarely needed by software drivers). More information on device security is presented in chapter 8.
- *Exclusive* - should more than one file object be allowed to open the same device? Most drivers should specify `FALSE`, but in some cases `TRUE` is more appropriate; it forces a single client at a time for the device.
- *DeviceObject* - the returned pointer, passed as an address of a pointer. If successful, `IoCreateDevice` allocates the structure from non-paged pool and stores the resulting pointer inside the dereferenced argument.

Before calling `IoCreateDevice` we must create a `UNICODE_STRING` to hold the internal device name:

```
UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\Booster");  
// alternatively,  
// RtlInitUnicodeString(&devName, L"\\Device\\Booster");
```

The device name could be anything but should be in the `\\Device` object manager directory. There are two ways to initialize a `UNICODE_STRING` with a constant string. The first is using `RtlInitUnicodeString`, which works just fine. But `RtlInitUnicodeString` must count the number of characters in the string to initialize the `Length` and `MaximumLength` appropriately. Not a big deal in this case, but there is a quicker way - using the `RTL_CONSTANT_STRING` macro, which calculates the length of the string statically (at compile time), meaning it can only work correctly with literal strings.

Now we are ready to call the `IoCreateDevice` function:

```
PDEVICE_OBJECT DeviceObject;  
NTSTATUS status = IoCreateDevice(  
    DriverObject,           // our driver object  
    0,                     // no need for extra bytes  
    &devName,               // the device name  
    FILE_DEVICE_UNKNOWN,  // device type  
    0,                     // characteristics flags  
    FALSE,                 // not exclusive  
    &DeviceObject);        // the resulting pointer  
if (!NT_SUCCESS(status)) {  
    KdPrint(("Failed to create device object (0x%08X)\n", status));  
    return status;  
}
```

If all goes well, we now have a pointer to our device object. The next step is to make this device object accessible to user-mode callers by providing a symbolic link. Creating a symbolic link involves calling `IoCreateSymbolicLink`:

```
NTSTATUS IoCreateSymbolicLink(  
    _In_ PUNICODE_STRING SymbolicLinkName,  
    _In_ PUNICODE_STRING DeviceName);
```

The following lines create a symbolic link and connect it to our device object:

```
UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\Booster");  
status = IoCreateSymbolicLink(&symLink, &devName);  
if (!NT_SUCCESS(status)) {  
    KdPrint(("Failed to create symbolic link (0x%08X)\n", status));  
    IoDeleteDevice(DeviceObject); // important!  
    return status;  
}
```

The `IoCreateSymbolicLink` does the work by accepting the symbolic link and the target of the link. Note that if the creation fails, we must undo everything done so far - in this case just the fact the device object was created - by calling `IoDeleteDevice`. More generally, if `DriverEntry` returns any failure status, the `Unload` routine is **not** called. If we had more initialization steps to do, we would have to remember to undo everything until that point in case of failure. We'll see a more elegant way of handling this in chapter 6.

Once we have the symbolic link and the device object set up, `DriverEntry` can return success, indicating the driver is now ready to accept requests.

Before we move on, we must not forget the `Unload` routine. Assuming `DriverEntry` completed successfully, the `Unload` routine must undo whatever was done in `DriverEntry`. In our case, there are two things to undo: device object creation and symbolic link creation. We'll undo them in reverse order:

```
void BoosterUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\Booster");
    // delete symbolic link
    IoDeleteSymbolicLink(&symLink);

    // delete device object
    IoDeleteDevice(DriverObject->DeviceObject);
}
```

Notice the device object pointer is extracted from the driver object, as it's the only argument we get in the `Unload` routine. It's certainly possible to store the device object pointer in a global variable and access it here directly, but there is no need. Global variables usage should be kept to a minimum.

Client Code

At this point, it's worth writing the user-mode client code. Everything we need for the client has already been defined.

Add a new C++ Console Application project to the solution named *Boost* (or some other name of your choosing). The Visual Studio wizard should create a single source file with some "hello world" type of code. You can safely delete all the contents of the file.

First, we add the required `#includes` to the *Boost.cpp* file:

```
#include <windows.h>
#include <stdio.h>
#include "..\Booster\BoosterCommon.h"
```

Note that we include the common header file created by the driver to be shared with the client.

Change the main function to accept command line arguments. We'll accept a thread ID and a priority using command line arguments and request the driver to change the priority of the thread to the given value.

```
int main(int argc, const char* argv[]) {
    if (argc < 3) {
        printf("Usage: Boost <threadid> <priority>\n");
        return 0;
    }

    //
    // extract from command line
    //
    int tid = atoi(argv[1]);
    int priority = atoi(argv[2]);
```

Next, we need to open a handle to our device. The “file name” to CreateFile should be the symbolic link prepended with “\\.”. The entire call should look like this:

```
HANDLE hDevice = CreateFile(L"\\\\.\\Booster", GENERIC_WRITE,
    0, nullptr, OPEN_EXISTING, 0, nullptr);
if (hDevice == INVALID_HANDLE_VALUE)
    return Error("Failed to open device");
```

The *Error* function simply prints some text with the last Windows API error:

```
int Error(const char* message) {
    printf("%s (error=%u)\n", message, GetLastError());
    return 1;
}
```

The CreateFile call should reach the driver in its IRP_MJ_CREATE dispatch routine. If the driver is not loaded at this time - meaning there is no device object and no symbolic link - we'll get error number 2 (file not found). Now that we have a valid handle to our device, it's time to set up the call to write. First, we need to create a ThreadData structure and fill in the details:

```
ThreadData data;
data.ThreadId = tid;
data.Priority = priority;
```

Now we're ready to call `WriteFile` and close the device handle afterwards:

```
DWORD returned;
BOOL success = WriteFile(hDevice,
    &data, sizeof(data),          // buffer and length
    &returned, nullptr);
if (!success)
    return Error("Priority change failed!");

printf("Priority change succeeded!\n");

CloseHandle(hDevice);
```

The call to `WriteFile` reaches the driver by invoking the `IRP_MJ_WRITE` major function routine.

At this point, the client code is complete. All that remains is to implement the dispatch routines we declared on the driver side.

The Create and Close Dispatch Routines

Now we're ready to implement the three dispatch routines defined by the driver. The simplest by far are the Create and Close routines. All that's needed is completing the request with a successful status. Here is the complete Create/Close dispatch routine implementation:

```
NTSTATUS BoosterCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    UNREFERENCED_PARAMETER(DeviceObject);

    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

Every dispatch routine accepts the target device object and an *I/O Request Packet* (IRP). We don't care much about the device object, since we only have one, so it must be the one we created in `DriverEntry`. The IRP on the other hand, is extremely important. We'll dig deeper into IRPs in chapter 6, but we need to take a quick look at IRPs now.

An IRP is a semi-documented structure that represents a request, typically coming from one of the managers in the Executive: the I/O Manager, the Plug & Play Manager, or the Power Manager. With a simple software driver, that would most likely be the I/O Manager. Regardless of the creator of the IRP, the driver's purpose is to handle the IRP, which means looking at the details of the request and doing what needs to be done to complete it.

Every request to the driver always arrives wrapped in an IRP, whether that's a Create, Close, Read, Write, or any other IRP. By looking at the IRP's members, we can figure out the type and details of the request (technically, the dispatch routine itself was pointed to based on the request type, so in most cases you already know the request type). It's worth mentioning that an IRP never arrives alone; it's accompanied by one or more structures of type `IO_STACK_LOCATION`. In simple cases like our driver, there is a single `IO_STACK_LOCATION`. In more complex cases where there are filter drivers above or below us, multiple `IO_STACK_LOCATION` instances exist, one for each layer in the device stack. (We'll discuss this more thoroughly in chapter 7). Simply put, some of the information we need is in the base IRP structure, and some is in the `IO_STACK_LOCATION` for our "layer" in the device stack.

In the case of Create and Close, we don't need to look into any members. We just need to set the completion status of the IRP in its `IoStatus` member (of type `IO_STATUS_BLOCK`), which has two members:

- *Status* (`NTSTATUS`) - indicating the status this request should complete with.
- *Information* (`ULONG_PTR`) - a polymorphic member, meaning different things in different request types. In the case of Create and Close, a zero value is just fine.

To complete the IRP, we call `IoCompleteRequest`. This function has a lot to do, but basically it propagates the IRP back to its creator (typically the I/O Manager), and that manager notifies the client that the operation has completed and frees the IRP. The second argument is a temporary priority boost value that a driver can provide to its client. In most cases for a software driver, a value of zero is fine (`IO_NO_INCREMENT` is defined as zero). This is especially true since the request completed synchronously, so no reason the caller should get a priority boost. More information on this function is provided in chapter 7.

The last thing to do is return the same status as the one put into the IRP. This may seem like a useless duplication, but it is necessary (the reason will be

clearer in a later chapter).



You may be tempted to write the last line of `BoosterCreateClose` like so:

`return Irp->IoStatus.Status;` So that the returned value is always the same as the one stored in the IRP. This code is buggy, however, and will cause a BSOD in most cases. The reason is that after `IoCompleteRequest` is invoked, the IRP pointer should be considered “poison”, as it’s more likely than not that it has already been deallocated by the I/O manager.

The Write Dispatch Routine

This is the crux of the matter. All the driver code so far has led to this dispatch routine. This is the one doing the actual work of setting a given thread to a requested priority.

The first thing we need to do is check for errors in the supplied data. In our case, we expect a structure of type `ThreadData`. The first thing is to do is retrieve the current IRP stack location, because the size of the buffer happens to be stored there:

```
NTSTATUS BoosterDeviceControl(PDEVICE_OBJECT, PIRP Irp) {  
    auto status = STATUS_SUCCESS;  
    ULONG_PTR information = 0; // track used bytes  
  
    // irpSp is of type PIO_STACK_LOCATION  
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);  
}
```

The key to getting the information for any IRP is to look inside the `IO_STACK_LOCATION` associated with the current device layer. Calling `IoGetCurrentIrpStackLocation` returns a pointer to the correct `IO_STACK_LOCATION`. In our case, there is just one `IO_STACK_LOCATION`, but in the general case there could be more (in fact, a filter may be above our device), so calling `IoGetCurrentIrpStackLocation` is the right thing to do.

The main ingredient in an `IO_STACK_LOCATION` is a monstrous union identified with the member named `Parameters`, which holds a set of structures, one for each type of IRP. In the case of `IRP_MJ_WRITE`, the structure to look at is `Parameters.Write`.

Now we can check the buffer size to make sure it's at least the size we expect:

```
do {  
    if (irpSp->Parameters.Write.Length < sizeof(ThreadData)) {  
        status = STATUS_BUFFER_TOO_SMALL;  
        break;  
    }  
}
```

The `do` keyword opens a simple `do/while(false)` block that allows using the `break` keyword to bail out early in case of an error. We'll discuss this technique in greater detail in chapter 7.

Next, we need to grab the user buffer's pointer, and check if the priority value is in the legal range (0 to 31). We also check if the pointer itself is `NULL`, as it's possible for the client to pass a `NULL` pointer for the buffer, but the length may be greater than zero. The buffer's address is provided in the `UserBuffer` member of the `IRP`:

```
auto data = static_cast<ThreadData*>(Irp->UserBuffer);  
if (data == nullptr || data->Priority < 1 || data->Priority > 31) {  
    status = STATUS_INVALID_PARAMETER;  
    break;  
}
```

`UserBuffer` is typed as a `void` pointer, so we need to cast it to the expected type. Then we check the priority value, and if not in range change the status to `STATUS_INVALID_PARAMETER` and break out of the "loop".



Notice the order of checks: the pointer is compared to `NULL` first, and only if non-`NULL`, the next check takes place. If `data` is `NULL`, however, no further checks are made. This behavior is guaranteed by the C/C++ standard, known as *short circuit evaluation*.



The use of `static_cast` asks the compiler to do check if the cast makes sense. Technically, the C++ compiler allows casting a `void` pointer to any other pointer, so it doesn't look that useful in this case, and perhaps a C-style cast would be simpler to write. Still, it's a good habit to have, as it can catch some errors at compile time.

We're getting closer to our goal. The API we would like to use is `KeSetPriorityThread`, prototyped as follows:

```
KPRIORITY KeSetPriorityThread(  
    _Inout_ PKTHREAD Thread,  
    _In_    KPRIORITY Priority);
```

The `KPRIORITY` type is just an 8-bit integer. The thread itself is identified by a pointer to a `KTHREAD` object. `KTHREAD` is one part of the way the kernel manages threads. It's completely undocumented, but we need the pointer value anyway. We have the thread ID from the client, and need to somehow get a hold of a pointer to the real thread object in kernel space. The function that can look up a thread by its ID is aptly named `PsLookupThreadByThreadId`. To get its definition, we need to add another `#include`:

```
#include <ntifs.h>
```



You must add this `#include` before `<ntddk.h>`, otherwise you'll get compilation errors. In fact, you can remove `<ntddk.h>` entirely, as it's included by `<ntifs.h>`.

Here is the definition for `PsLookupThreadByThreadId`:

```
NTSTATUS PsLookupThreadByThreadId(  
    _In_      HANDLE ThreadId,  
    _Outptr_  PETHREAD *Thread);
```

Again, we see that a thread ID is required, but its type is `HANDLE` - but it is the ID that we need nonetheless. The resulting pointer is typed as `PETHREAD` or pointer to `ETHREAD`. `ETHREAD` is completely opaque. Regardless, we seem to have a problem since `KeSetPriorityThread` accepts a `PKTHREAD` rather than `PETHREAD`. It turns out these are the same, because the first member of an `ETHREAD` is a `KTHREAD` (the member is named `Tcb`). We'll prove all this in the next chapter when we use the kernel debugger. Here is the beginning of the definition of `ETHREAD`:

```
typedef struct _ETHREAD {  
    KTHREAD Tcb;
```

```
    // more members  
} ETHREAD;
```

The bottom line is we can safely switch PKTHREAD for PETHREAD or vice versa when needed without a hitch.

Now we can turn our thread ID into a pointer:

```
PETHREAD thread;  
status = PsLookupThreadByThreadId(ULongToHandle(data->ThreadId),  
    &thread);  
if (!NT_SUCCESS(status))  
    break;
```

The call to PsLookupThreadByThreadId can fail, the main reason being that the thread ID does not reference any thread in the system. If the call fails, we simply break and let the resulting NTSTATUS propagate out of the “loop”.

We are finally ready to change the thread’s priority. But wait - what if after the last call succeeds, the thread is terminated, just before we set its new priority? Rest assured, this cannot happen. Technically, the thread can terminate (from an execution perspective) at that point, but that will not make our pointer a dangling one. This is because the lookup function, if successful, increments the reference count on the kernel thread object, so it cannot die until we explicitly decrement the reference count. Here is the call to make the priority change:

```
auto oldPriority = KeSetPriorityThread(thread, data->Priority);  
KdPrint(("Priority change for thread %u from %d to %d succeeded!\n",  
    data->ThreadId, oldPriority, data->Priority));
```

We get back the old priority, which we output with KdPrint for debugging purposes. All that’s left to do now is decrement the thread object’s reference; otherwise, we have a leak on our hands (the thread object will never die), which will only be resolved in the next system boot. The function that accomplishes this feat is ObDereferenceObject:

```
ObDereferenceObject(thread);
```

We should also report to the client that we used the buffer provided. This is where the information variable is used:

```
information = sizeof(data);
```

We'll write that value to the IRP before completing it. This is the value returned as the second to last argument from the client's `writeFile` call. All that's left to do is to close the while "loop" and complete the IRP with whatever status we happen to have at this time.

```
// end the while "loop"
} while (false);

//
// complete the IRP with the status we got at this point
//
Irp->IoStatus.Status = status;
Irp->IoStatus.Information = information;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
return status;
}
```

And we're done! For reference, here is the complete `IRP_MJ_WRITE` handler:

```
NTSTATUS BoosterWrite(PDEVICE_OBJECT, PIRP Irp) {
    auto status = STATUS_SUCCESS;
    ULONG_PTR information = 0;

    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    do {
        if (irpSp->Parameters.Write.Length < sizeof(ThreadData)) {
            status = STATUS_BUFFER_TOO_SMALL;
            break;
        }
        auto data = static_cast<ThreadData*>(Irp->UserBuffer);
        if (data == nullptr
            || data->Priority < 1 || data->Priority > 31) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }
        PETHREAD thread;
        status = PsLookupThreadByThreadId(
            ULongToHandle(data->ThreadId), &thread);
        if (!NT_SUCCESS(status)) {
            break;
        }
        auto oldPriority = KeSetPriorityThread(thread, data->Priority);
        KdPrint(("Priority change for thread %u from %d to %d succeeded!\n",
            data->ThreadId, oldPriority, data->Priority));

        ObDereferenceObject(thread);
        information = sizeof(data);
    } while (false);

    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = information;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}
```

Installing and Testing

At this point, we can build the driver and client successfully. Our next step is to install the driver and test its functionality. You can try the following on a virtual machine, or if you're feeling brave enough - on your development machine.

First, let's install the driver. Copy the resulting *booster.sys* file to the target machine (if it's not your development machine). On the target machine, open an elevated command window and install the driver using the *sc.exe* tool as we did back in chapter 2:

```
c:\> sc create booster type= kernel binPath= c:\Test\Booster.sys
```

Make sure *binPath* includes the full path of the resulting SYS file. The name of the driver (*booster*) in the example is the name of the created Registry key, and so must be unique. It doesn't have to be related to the SYS file name.

Now we can load the driver:

```
c:\> sc start booster
```

If all is well, the driver would have started successfully. To make sure, we can open *WinObj* and look for our device name and symbolic link. Figure 4-1 shows the symbolic link in *WinObj*.

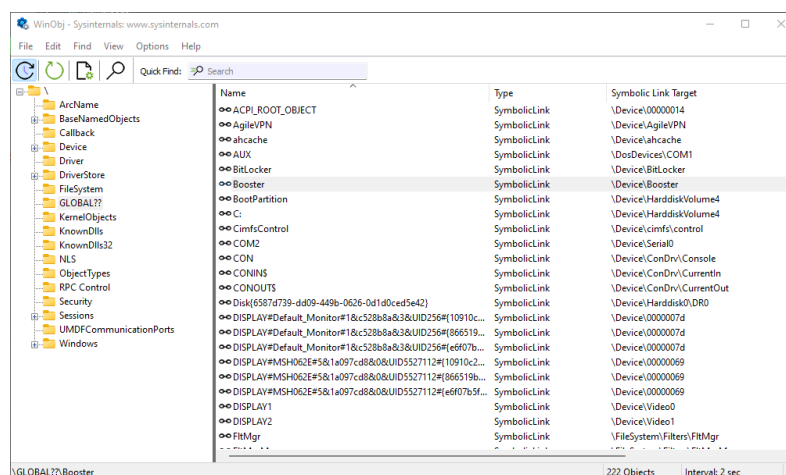


Figure 4-1: Symbolic Link in WinObj

Now we can finally run the client executable. Figure 4-2 shows a thread in *Process Explorer* of a *cmd.exe* process selected as an example for which we want set priority to a new value.

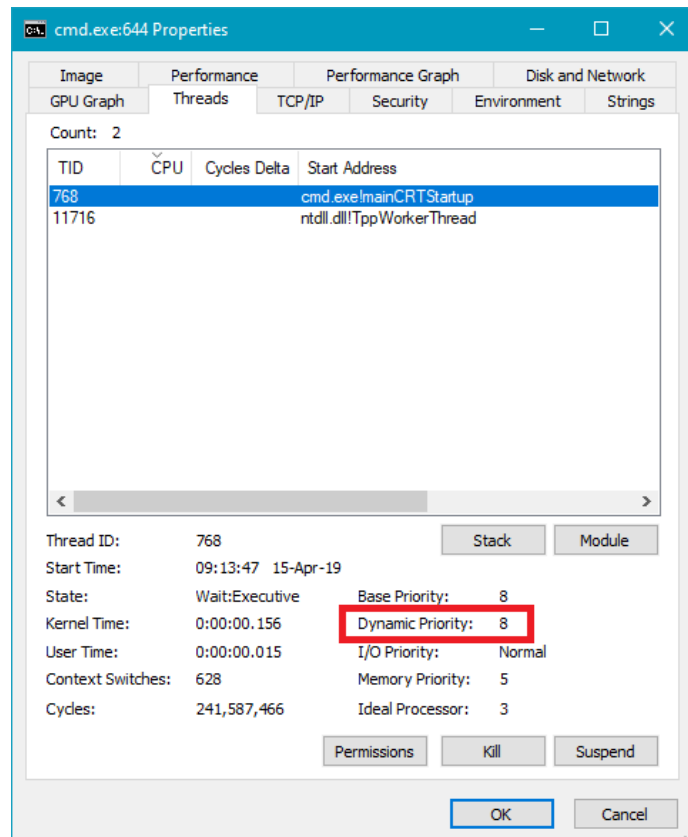


Figure 4-2: Original thread priority

Run the client with the thread ID and the desired priority (replace the thread ID as needed):

```
c:\Test> boost 768 25
```



If you get an error trying to run the executable (usually it's a Debug build), you may need to set the runtime library to a static one instead of a DLL. Go to *Project properties* in Visual Studio for the client application, C++ node, *Code Generation*, *Runtime Library*, and select **Multithreaded Debug**. Alternatively, you can compile the client in *Release* build, and that should run without any changes.

And voila! See figure 4-3.

You should also run *DbgView* and see the output when a successful priority change occurs.

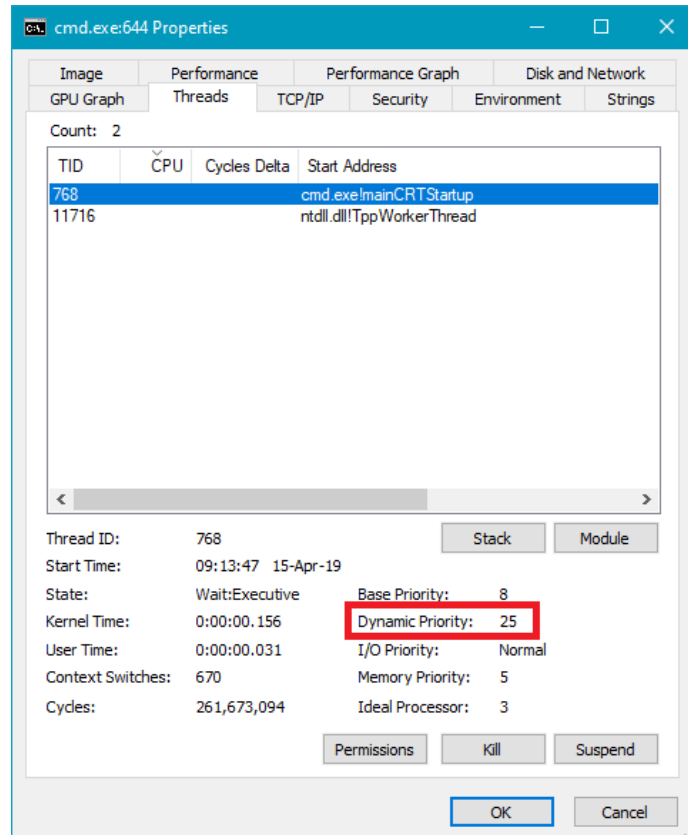


Figure 4-3: Modified thread priority

Summary

We've seen how to build a simple, yet complete, driver, from start to finish. We created a user-mode client to communicate with the driver. In the next chapter, we'll tackle debugging, which is something we're bound to do when writing drivers that may not behave as we expect.

Chapter 5: Debugging and Tracing

Just like with any software, kernel drivers tend to have bugs. Debugging drivers, as opposed to user-mode debugging, is more challenging. Driver debugging is essentially debugging an entire machine, not just a specific process. This requires a somewhat different mindset. This chapter discussed user-mode and kernel-mode debugging using the *WinDbg* debugger.

In this chapter:

- **Debugging Tools for Windows**
 - **Introduction to *WinDbg***
 - **Kernel Debugging**
 - **Full Kernel Debugging**
 - **Kernel Driver Debugging Tutorial**
 - **Asserts and Tracing**
-

Debugging Tools for Windows

The *Debugging Tools for Windows* package contains a set of debuggers, tools, and documentation focusing on the debuggers within the package. This package can be installed as part of the Windows SDK or the WDK, but there is no real “installation” done. The installation just copies files but does not touch the Registry, meaning the package depends only on its own modules and the Windows built-in DLLs. This makes it easy to copy the entire directory to any other directory including removable media.

The package contains four debuggers: *Cdb.exe*, *Ntsd.Exe*, *Kd.exe*, and *WinDbg.exe*. Here is a rundown of the basic functionality of each debugger:

- *Cdb* and *Ntsd* are user-mode, console-based debuggers. This means they can be attached to processes, just like any other user-mode debugger. Both

have console UI - type in a command, get a response, and repeat. The only difference between the two is that if launched from a console window, *Cdb* uses the same console, whereas *Ntsd* always opens a new console window. They are otherwise identical.

- *Kd* is a kernel debugger with a console user interface. It can attach to the local kernel (*Local Kernel Debugging*, described in the next section), or to another machine for a full kernel debugging experience.
- *WinDbg* is the only debugger with a graphical user interface. It can be used for user-mode debugging or kernel debugging, depending on the selection performed with its menus or the command line arguments passed to it when launched.

A relatively recent alternative to the classic *WinDbg* is *Windbg Preview*, available through the Microsoft store. This is a remake of the classic debugger with a much better user interface. It can be installed on Windows 10 version 1607 or later. From a functionality standpoint, it's similar to the classic *WinDbg*. But it is somewhat easier to use because of the modern, convenient UI, and in fact has also solved some bugs that still plague the classic debugger. All the commands we'll see in this chapter work equally well with either debugger.

Although these debuggers may seem different from one another, the user-mode debuggers are essentially the same, as are the kernel debuggers. They are all based around a single debugger engine implemented as a DLL (*DbgEng.Dll*). The various debuggers are able to use *extension DLLs*, that provide most of the power of the debuggers by loading new commands.

The Debugger Engine is documented to a large extent in the *Debugging tools for Windows* documentation, which makes it possible to write new debuggers (or other tools) that utilize the debugger engine.

Other tools that are part of the package include the following (partial list):

- *Gflags.exe* - the Global Flags tool that allows setting some kernel flags and image flags.
- *ADPlus.exe* - generate a dump file for a process crash or hang.
- *Kill.exe* - a simple tool to terminate process(es) based on process ID, name, or pattern.
- *Dumpchk.exe* - tool to do some general checking of dump files.
- *TList.exe* - lists running processes on the system with various options.

- *Umdh.exe* - analyzes heap allocations in user-mode processes.
- *UsbView.exe* - displays a hierarchical view of USB devices and hubs.

Introduction to *WinDbg*

This section describes the fundamentals of *WinDbg*, but bear in mind everything is essentially the same for the console debuggers, with the exception of the GUI windows.

WinDbg is built around commands. The user enters a command, and the debugger responds with text describing the results of the command. With the GUI, some of these results are depicted in dedicated windows, such as locals, stack, threads, etc.

WinDbg supports three types of commands:

- Intrinsic commands - these commands are built-in into the debugger (part of the debugger engine), and they operate on the target being debugged.
- Meta commands - these commands start with a period (.) and they operate on the debugging environment, rather than directly on the target being debugged.
- Extension commands (sometimes called *bang commands*) - these commands start with an exclamation point (!), providing much of the power of the debugger. All extension commands are implemented in external DLLs. By default, the debugger loads a set of predefined extension DLLs, but more can be loaded from the debugger directory or another directory with the `.load` meta command.

Writing extension DLLs is possible and is fully documented in the debugger docs. In fact, many such DLLs have been created and can be loaded from their respective source. These DLLs provide new commands that enhance the debugging experience, often targeting specific scenarios.

Tutorial: User mode debugging basics

If you have experience with *WinDbg* usage in user-mode, you can safely skip this section.

There are generally two ways to initiate user-mode debugging - either launch an executable and attach to it, or attach to an already existing process. We'll use the latter approach in this tutorial, but except for this first step, all other operations are identical.

- [illegible]

Figure 5-1: Attaching to a process with WinDbg

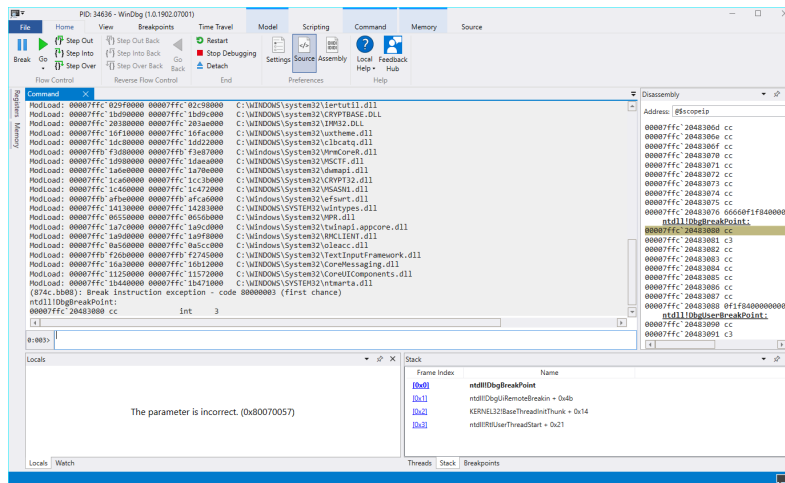


Figure 5-2: First view after process attach

The Command window is the main window of interest - it should always be open. This is the one showing the various responses of commands. Typically, most of the time in a debugging session is spent interacting with this window.

The process is suspended - we are in a breakpoint induced by the debugger.

- The first command we'll use is ~, which shows information about all threads in the debugged process:

```
0:003> ~
0 Id: 874c.18068 Suspend: 1 Teb: 00000001`2229d000 Unfrozen
1 Id: 874c.46ac Suspend: 1 Teb: 00000001`222a5000 Unfrozen
2 Id: 874c.152cc Suspend: 1 Teb: 00000001`222a7000 Unfrozen
. 3 Id: 874c.bb08 Suspend: 1 Teb: 00000001`222ab000 Unfrozen
```

The exact number of threads you'll see may be different than shown here.

One thing that is very important is the existence of proper symbols. Microsoft provides a public symbol server, which allows locating symbols for most modules by produced by Microsoft. This is essential in any low-level debugging.

- To set symbols quickly, enter the **.symfix** command.
- A better approach is to set up symbols once and have them available for all future debugging sessions. To do that, add a system environment variable named **_NT_SYMBOL_PATH** and set it to a string like the following:

SRV*c:\Symbols*http://msdl.microsoft.com/download/symbols

The middle part (between asterisks) is a local path for caching symbols on your local machine; you can select any path you like (including a network share, if sharing with a team is desired). Once this environment variable is set, next invocations of the debugger will find symbols automatically and load them from the Microsoft symbol server as needed.



The debuggers in the *Debugging Tools for Windows* are not the only tools that look for this environment variables. *Sysinternals* tools (e.g. *Process Explorer*, *Process Monitor*), Visual Studio, and others look for the same variable as well. You set it once, and get its benefit using multiple tools.

- To make sure you have proper symbols, enter the `lm` (loaded modules) command:

```
0:003> lm
start                end                module name
00007ff7`53820000 00007ff7`53863000 notepad    (deferred)
00007ffb`afbe0000 00007ffb`afca6000 efs wrt    (deferred)
...
00007ffc`1db00000 00007ffc`1dba8000 shcore     (deferred)
00007ffc`1dbb0000 00007ffc`1dc74000 OLEAUT32   (deferred)
00007ffc`1dc80000 00007ffc`1dd22000 clbcatq    (deferred)
00007ffc`1dd30000 00007ffc`1de57000 COMDLG32   (deferred)
00007ffc`1de60000 00007ffc`1f350000 SHELL32    (deferred)
00007ffc`1f500000 00007ffc`1f622000 RPCRT4     (deferred)
00007ffc`1f630000 00007ffc`1f6e3000 KERNEL32   (pdb symbols)  c:\symbols\ker\
nel32.pdb\3B92DED9912D874A2BD08735BC0199A31\kernel32.pdb
00007ffc`1f700000 00007ffc`1f729000 GDI32      (deferred)
00007ffc`1f790000 00007ffc`1f7e2000 SHLWAPI    (deferred)
00007ffc`1f8d0000 00007ffc`1f96e000 sechost    (deferred)
00007ffc`1f970000 00007ffc`1fc9c000 combase    (deferred)
00007ffc`1fca0000 00007ffc`1fd3e000 msvcrt     (deferred)
00007ffc`1fe50000 00007ffc`1fef3000 ADVAPI32   (deferred)
00007ffc`20380000 00007ffc`203ae000 IMM32      (deferred)
00007ffc`203e0000 00007ffc`205cd000 ntdll      (pdb symbols)  c:\symbols\ntd\
ll.pdb\E7EEB80BFAA91532B88FF026DC6B9F341\ntdll.pdb
```

The list of modules shows all modules (DLLs and the EXE) loaded into the debugged process at this time. You can see the start and end virtual addresses into which each module is loaded. Following the module name you can see the symbol status of this module (in parenthesis). Possible values include:

- *deferred* - the symbols for this module were not needed in this debugging session so far, and so are not loaded at this time. The symbols will be loaded when needed (for example, if a call stack contains a function from that module). This is the default value.
- *pdb symbols* - proper public symbols have been loaded. The local path of the PDB file is displayed.
- *private pdb symbols* - private symbols are available. This would be the case for your own modules, compiled with Visual Studio. For Microsoft modules, this is very rare (at the time of writing, *combase.dll* is provided with private symbols). With private symbols, you have information about local variables and private types.
- *export symbols* - only exported symbols are available for this DLL. This typically means there are no symbols for this module, but the debugger is able to use the exported symbols. It's better than no symbols at all, but could be confusing, as the debugger will use the closest export it can find, but the real function is most likely different.
- *no symbols* - this module's symbols were attempted to be located, but nothing was found, not even exported symbols (such modules don't have exported symbols, as is the case of an executable or driver files).

You can force loading of a module's symbols using the following command:

```
.reload /f modulename.dll
```

This will provide definitive evidence to the availability of symbols for this module.

Symbol paths can also be configured in the debugger's settings dialog.

Open the *File / Settings* menu and locate *Debugging Settings*. You can then add more paths for symbol searching. This is useful if debugging your own code, so you would like the debugger to search your directories where relevant PDB files may be found (see figure 5-3).

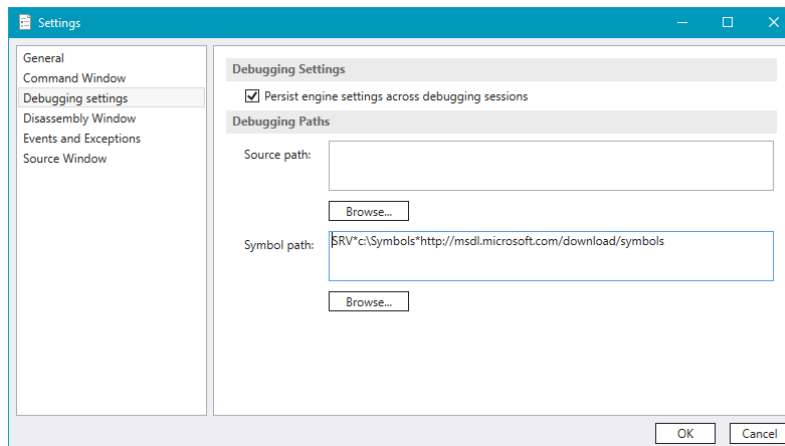


Figure 5-3: Symbols and source paths configuration

Make sure you have symbols configured correctly before you proceed. To diagnose any issues, you can enter the `!sym noisy` command that logs detailed information for symbol load attempts.

Back to the thread list - notice that one of the threads has a dot in front of its data. This is the current thread as far as the debugger is concerned. This means that any command issued that involves a thread, where the thread is not explicitly specified, will work on that thread. This “current thread” is also shown in the prompt - the number to the right of the colon is the current thread index (3 in this example).

Enter the `k` command, that shows the stack trace for the current thread:

```
0:003> k
# Child-SP          RetAddr           Call Site
00 00000001`224ffbd8 00007ffc`204aef5b ntdll!DbgBreakPoint
01 00000001`224ffbe0 00007ffc`1f647974 ntdll!DbgUiRemoteBreakin+0x4b
02 00000001`224ffc10 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
03 00000001`224ffc40 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```



How can you tell that you don't have proper symbols except using the `!m` command? If you see very large offsets from the beginning of a function, this is probably not the real function name - it's just the closest one the debugger knows about. “Large offsets” is obviously a relative term, but a good rule of thumb is that a 4-hex digit offset is almost always wrong.

You can see the list of calls made on this thread (user-mode only, of course). The top of the call stack in the above output is the function `DbgBreakPoint` located in the module `ntdll.dll`. The general format of addresses with symbols is `modulename!functionname+offset`. The offset is optional and could be zero if it's exactly the start of this function. Also notice the module name is without an extension.

In the output above, `DbgBreakpoint` was called by `DbgUiRemoteBreakIn`, which was called by `BaseThreadInitThunk`, and so on.

This thread, by the way, was injected by the debugger in order to break into the target forcefully.

To switch to a different thread, use the following command: `~ns` where *n* is the thread index. Let's switch to thread 0 and then display its call stack:

```
0:003> ~0s
win32u!NtUserGetMessage+0x14:
00007ffc`1c4b1164 c3          ret
0:000> k
# Child-SP          RetAddr          Call Site
00 00000001`2247f998 00007ffc`1d802fbd win32u!NtUserGetMessage+0x14
01 00000001`2247f9a0 00007ff7`5382449f USER32!GetMessageW+0x2d
02 00000001`2247fa00 00007ff7`5383ae07 notepad!WinMain+0x267
03 00000001`2247fb00 00007ffc`1f647974 notepad!__mainCRTStartup+0x19f
04 00000001`2247fbc0 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
05 00000001`2247fbf0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

This is *Notepad*'s main (first) thread. The top of the stack shows the thread waiting for UI messages (`win32u!NtUserGetMessage`). The thread is actually waiting in kernel mode, but this is invisible from a user-mode debugger's view.

An alternative way to show the call stack of another thread without switching to it, is to use the tilde and thread number before the actual command. The following output is for thread 1's stack:

```
0:000> ~1k
# Child-SP          RetAddr          Call Site
00 00000001`2267f4c8 00007ffc`204301f4 ntdll!NtWaitForWorkViaWorkerFactory+0x14
01 00000001`2267f4d0 00007ffc`1f647974 ntdll!TppWorkerThread+0x274
02 00000001`2267f7c0 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
03 00000001`2267f7f0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```



The above call stack is very common, and indicates a thread that is part of the thread pool. `TppWorkerThread` is the thread entry point for thread pool threads (Tpp is short for “Thread Pool Private”).

Let’s go back to the list of threads:

```
. 0 Id: 874c.18068 Suspend: 1 Teb: 00000001`2229d000 Unfrozen
  1 Id: 874c.46ac Suspend: 1 Teb: 00000001`222a5000 Unfrozen
  2 Id: 874c.152cc Suspend: 1 Teb: 00000001`222a7000 Unfrozen
# 3 Id: 874c.bb08 Suspend: 1 Teb: 00000001`222ab000 Unfrozen
```

Notice the dot has moved to thread 0 (current thread), revealing a hash sign (#) on thread 3. The thread marked with a hash (#) is the one that caused the last breakpoint (which in this case was our initial debugger attach).

The basic information for a thread provided by the `~` command is shown in figure 5-4.

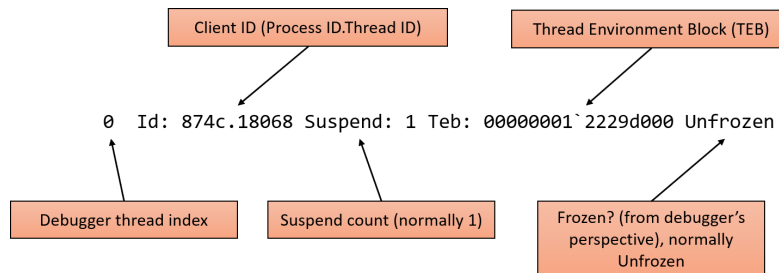


Figure 5-4: Thread information for the `~` command

Most numbers reported by *WinDbg* are hexadecimal by default. To convert a value to decimal, you can use the `?` (evaluate expression) command.

Type the following to get the decimal process ID (you can then compare to the reported PID in *Task Manager*):

```
0:000> ? 874c
Evaluate expression: 34636 = 00000000`0000874c
```

You can express decimal numbers with the `0n` prefix, so you can get the inverse result as well:

```
0:000> ? 0n34636
Evaluate expression: 34636 = 00000000`0000874c
```



The `0y` prefix can be used in *WinDbg* to specify binary values. For example, using `0y1100` is the same as `0n12` as is `0xc`. You can use the `?` command to see the converted values.

You can examine the TEB of a thread by using the `!teb` command. Using `!teb` without an address shows the TEB of the current thread:

```
0:000> !teb
TEB at 0000000012229d000
  ExceptionList: 0000000000000000
  StackBase: 00000000122480000
  StackLimit: 0000000012246f000
  SubSystemTib: 0000000000000000
  FiberData: 00000000000001e00
  ArbitraryUserPointer: 0000000000000000
  Self: 0000000012229d000
  EnvironmentPointer: 0000000000000000
  ClientId: 000000000000874c . 00000000000018068
  RpcHandle: 0000000000000000
  Tls Storage: 000001c93676c940
  PEB Address: 0000000012229c000
  LastErrorValue: 0
  LastStatusValue: 8000001a
  Count Owned Locks: 0
  HardErrorMode: 0
0:000> !teb 00000001`222a5000
TEB at 000000001222a5000
  ExceptionList: 0000000000000000
  StackBase: 00000000122680000
  StackLimit: 0000000012266f000
  SubSystemTib: 0000000000000000
  FiberData: 00000000000001e00
  ArbitraryUserPointer: 0000000000000000
  Self: 000000001222a5000
  EnvironmentPointer: 0000000000000000
  ClientId: 000000000000874c . 00000000000046ac
  RpcHandle: 0000000000000000
  Tls Storage: 000001c936764260
  PEB Address: 0000000012229c000
  LastErrorValue: 0
  LastStatusValue: c0000034
  Count Owned Locks: 0
  HardErrorMode: 0
```

Some data shown by the `!teb` command is relatively known or easy to guess:

- *StackBase* and *StackLimit* - user-mode current stack base and stack limit for the thread.
- *ClientId* - process and thread IDs.
- *LastErrorValue* - last Win32 error code (GetLastError).
- *TlsStorage* - *Thread Local Storage* (TLS) array for this thread (full explanation of TLS is beyond the scope of this book).
- *PEB Address* - address of the Process Environment Block (PEB), viewable with the `!peb` command.
- *LastStatusValue* - last NTSTATUS value returned from a system call.
- The `!teb` command (and similar commands) shows parts of the real data structure behind the scenes, in this case `_TEB`. You can always look at the real structure using the `dt` (display type) command:

```
0:000> dt ntdll!_teb
+0x000 NtTib           : _NT_TIB
+0x038 EnvironmentPointer : Ptr64 Void
+0x040 ClientId       : _CLIENT_ID
+0x050 ActiveRpcHandle : Ptr64 Void
+0x058 ThreadLocalStoragePointer : Ptr64 Void
+0x060 ProcessEnvironmentBlock : Ptr64 _PEB
...
+0x1808 LockCount      : Uint4B
+0x180c WowTebOffset   : Int4B
+0x1810 ResourceRetValue : Ptr64 Void
+0x1818 ReservedForWdf : Ptr64 Void
+0x1820 ReservedForCrt  : Uint8B
+0x1828 EffectiveContainerId : _GUID
```

Notice that *WinDbg* is **not** case sensitive when it comes to symbols. Also, notice the structure name starting with an underscore; this the way most structures are defined in Windows (user-mode and kernel-mode). Using the typedef name (without the underscore) may or may not work, so always using the underscore is recommended.



How do you know which module defines a structure you wish to view? If the structure is documented, the module would be listed in the docs for the structure. You can also try specifying the structure without the module name, forcing the debugger to search for it. Generally, you “know” where the structure is defined with experience and sometimes context.

If you attach an address to the previous command, you can get the actual values of data members:

```
0:000> dt ntdll!_teb 00000001`2229d000
+0x000 NtTib           : \_NT\_TIB
+0x038 EnvironmentPointer : (null)
+0x040 ClientId        : \_CLIENT\_ID
+0x050 ActiveRpcHandle  : (null)
+0x058 ThreadLocalStoragePointer : 0x000001c9`3676c940 Void
+0x060 ProcessEnvironmentBlock : 0x00000001`2229c000 \_PEB
+0x068 LastErrorValue   : 0
...
+0x1808 LockCount       : 0
+0x180c WowTebOffset    : 0n0
+0x1810 ResourceRetValue : 0x000001c9`3677fd00 Void
+0x1818 ReservedForWdf  : (null)
+0x1820 ReservedForCrt  : 0
+0x1828 EffectiveContainerId : \_GUID {00000000-0000-0000-0000-000000000000}
```

Each member is shown with its offset from the beginning of the structure, its name, and its value. Simple values are shown directly, while structure values (such as `NtTib` above) are shown with a hyperlink. Clicking this hyperlink provides the details of the structure.

Click on the `NtTib` member above to show the details of this data member:

```
0:000> dx -r1 (*((ntdll!_NT_TIB *)0x12229d000))
(*((ntdll!_NT_TIB *)0x12229d000)) [Type: \_NT\_TIB]
[+0x000] ExceptionList : 0x0 [Type: \_EXCEPTION\_REGISTRATION\_RECORD *]
[+0x008] StackBase      : 0x122480000 [Type: void *]
[+0x010] StackLimit     : 0x12246f000 [Type: void *]
[+0x018] SubSystemTib   : 0x0 [Type: void *]
[+0x020] FiberData      : 0x1e00 [Type: void *]
[+0x020] Version        : 0x1e00 [Type: unsigned long]
[+0x028] ArbitraryUserPointer : 0x0 [Type: void *]
[+0x030] Self           : 0x12229d000 [Type: \_NT\_TIB *]
```

The debugger uses the newer **dx** command to view data. See the section “Advanced Debugging with WinDbg” later in this chapter for more on the **dx** command.

If you don't see hyperlinks, you may be using a very old *WinDbg*, where Debugger Markup Language (DML) is not on by default. You can turn it on with the `.prefer_dml 1` command.

Now let's turn our attention to breakpoints. Let's set a breakpoint when a file is opened by *notepad*.

- Type the following command to set a breakpoint in the `CreateFile` API function:

```
0:000> bp kernel32!createfilew
```

Notice the function name is in fact `CreateFilew`, as there is no function called `CreateFile`. In code, this is a macro that expands to `CreateFilew` (wide, Unicode version) or `CreateFileA` (ASCII or Ansi version) based on a compilation constant named `UNICODE`. *WinDbg* responds with nothing. This is a good thing.



The reason there are two sets of functions for most APIs where strings are involved is a historical one. In any case, Visual Studio projects define the `UNICODE` constant by default, so Unicode is the norm. This is a good thing - most of the A functions convert their input to Unicode and call the W function.

You can list the existing breakpoints with the `bl` command:

```
0:000> bl
0 e Disable Clear 00007ffc`1f652300 0001 (0001) 0:**** KERNEL32!CreateFilew
```

You can see the breakpoint index (0), whether it's enabled or disabled (e=enabled, d=disabled), and you get DML hyperlinks to disable (`bd` command) and delete (`bc` command) the breakpoint.

Now let notepad continue execution, until the breakpoint hits:

Type the `g` command or press the *Go* button on the toolbar or hit *F5*:

You'll see the debugger showing **Busy** in the prompt and the command area shows **Debuggee is running**, meaning you cannot enter commands until the next break.

Notepad should now be alive. Go to its *File* menu and select *Open...* The debugger should spew details of module loads and then break:

Breakpoint 0 hit

KERNEL32!CreateFileW:

```
00007ffc`1f652300 ff25aa670500 jmp qword ptr [KERNEL32!_imp_CreateFileW \
(00007ffc`1f6a8ab0)] ds:00007ffc`1f6a8ab0={KERNELBASE!CreateFileW (00007ffc`1c7\
5e260)}
```

- We have hit the breakpoint! Notice the thread in which it occurred. Let's see what the call stack looks like (it may take a while to show if the debugger needs to download symbols from Microsoft's symbol server):

0:002> k

#	Child-SP	RetAddr	Call Site
00	00000001`226fab08	00007ffc`061c8368	KERNEL32!CreateFileW
01	00000001`226fab10	00007ffc`061c5d4d	mscorlib!RuntimeDesc::VerifyMainRuntimeM\ndule+0x2c
02	00000001`226fab60	00007ffc`061c6068	mscorlib!FindRuntimesInInstallRoot+0x2fb
03	00000001`226fb3e0	00007ffc`061cb748	mscorlib!GetOrCreateSxSProcessInfo+0x94
04	00000001`226fb460	00007ffc`061cb62b	mscorlib!CLRMetaHostPolicyImpl::GetReque\stedRuntimeHelper+0xfc
05	00000001`226fb740	00007ffc`061ed4e6	mscorlib!CLRMetaHostPolicyImpl::GetReque\stedRuntime+0x120
...			
21	00000001`226fede0	00007ffc`1df025b2	SHELL32!CFSIconOverlayManager::LoadNonlo\adedOverlayIdentifiers+0xaa
22	00000001`226ff320	00007ffc`1df022af	SHELL32!EnableExternalOverlayIdentifiers\+0x46
23	00000001`226ff350	00007ffc`1def434e	SHELL32!CFSIconOverlayManager::RefreshOv\erlayImages+0xff
24	00000001`226ff390	00007ffc`1cf250a3	SHELL32!SHELL32_GetIconOverlayManager+0x\6e
25	00000001`226ff3c0	00007ffc`1ceb2726	windows_storage!CFSFolder::_GetOverlayIn\fo+0x12b
26	00000001`226ff470	00007ffc`1cf3108b	windows_storage!CAutoDestItemsFolder::Ge\tOverlayIndex+0xb6
27	00000001`226ff4f0	00007ffc`1cf30f87	windows_storage!CRegFolder::_GetOverlayI\nfo+0xbf
28	00000001`226ff5c0	00007ffb`df8fc4d1	windows_storage!CRegFolder::GetOverlayIn\dex+0x47
29	00000001`226ff5f0	00007ffb`df91f095	explorerframe!CNscOverlayTask::_Extract+\0x51
2a	00000001`226ff640	00007ffb`df8f70c2	explorerframe!CNscOverlayTask::InternalR\esumeRT+0x45
2b	00000001`226ff670	00007ffc`1cf7b58c	explorerframe!CRunnableTask::Run+0xb2
2c	00000001`226ff6b0	00007ffc`1cf7b245	windows_storage!CShellTask::TT_Run+0x3c
2d	00000001`226ff6e0	00007ffc`1cf7b125	windows_storage!CShellTaskThread::Thread\Proc+0xdd
2e	00000001`226ff790	00007ffc`1db32ac6	windows_storage!CShellTaskThread::s_Thre\adProc+0x35
2f	00000001`226ff7c0	00007ffc`204521c5	shcore!ExecuteWorkItemThreadProc+0x16
30	00000001`226ff7f0	00007ffc`204305c4	ntdll!RtlpTpWorkCallback+0x165
31	00000001`226ff8d0	00007ffc`1f647974	ntdll!TppWorkerThread+0x644
32	00000001`226ffb00	00007ffc`2044a271	KERNEL32!BaseThreadInitThunk+0x14
33	00000001`226ffb00	00000000`00000000	ntdll!RtlUserThreadStart+0x21

Your call stack may be different, as it depends on the Windows version, and any extensions that may be loaded and used by the open file dialog box.

What can we do at this point? You may wonder what file is being opened. We can get that information based on the calling convention of the `CreateFileW` function. Since this is a 64-bit process (and the processor is Intel/AMD), the calling convention states that the first integer/pointer arguments are passed in the *RCX*, *RDX*, *R8*, and *R9* registers (in this order). Since the file name in `CreateFileW` is the first argument, the relevant register is *RCX*.

You can get more information on calling conventions in the Debugger documentation (or in several web resources).

Display the value of the *RCX* register with the `r` command (you'll get a different value):

```
0:002> r rcx
rcx=00000001226fabf8
```

We can view the memory pointed by *RCX* with various `d` (display) family of commands. Here is the `db` command, interpreting the data as bytes.

```
0:002> db 00000001226fabf8
00000001`226fabf8  43 00 3a 00 5c 00 57 00-69 00 6e 00 64 00 6f 00  C:.\.W.i.n\
.d.o.
00000001`226fac08  77 00 73 00 5c 00 4d 00-69 00 63 00 72 00 6f 00  w.s.\.M.i.c\
.r.o.
00000001`226fac18  73 00 6f 00 66 00 74 00-2e 00 4e 00 45 00 54 00  s.o.f.t...N\
.E.T.
00000001`226fac28  5c 00 46 00 72 00 61 00-6d 00 65 00 77 00 6f 00  \.F.r.a.m.e\
.W.O.
00000001`226fac38  72 00 6b 00 36 00 34 00-5c 00 5c 00 76 00 32 00  r.k.6.4.\.\
.v.2.
00000001`226fac48  2e 00 30 00 2e 00 35 00-30 00 37 00 32 00 37 00  ..0...5.0.7\
.2.7.
00000001`226fac58  5c 00 63 00 6c 00 72 00-2e 00 64 00 6c 00 6c 00  \.c.l.r...d\
.l.l.
00000001`226fac68  00 00 76 1c fc 7f 00 00-00 00 00 00 00 00 00 00  ..v.....\
.....
```

The `db` command shows the memory in bytes, and ASCII characters on the right. It's pretty clear what the file name is, but because the string is Unicode,

it's not very convenient to see.

Use the du command to view Unicode string more conveniently:

```
0:002> du 00000001226fabf8
00000001`226fabf8  "C:\Windows\Microsoft.NET\Frameworko"
00000001`226fac38  "rk64\\v2.0.50727\clr.dll"
```

You can use a register value directly by prefixing its name with @:

```
0:002> du @rcx
00000001`226fabf8  "C:\Windows\Microsoft.NET\Frameworko"
00000001`226fac38  "rk64\\v2.0.50727\clr.dll"
```

Similarly, you can view the value of the second argument by looking at the rdx register.

Now let's set another breakpoint in the native API that is called by CreateFileW - NtCreateFile:

```
0:002> bp ntdll!ntcreatefile
0:002> bl
  0 e Disable Clear  00007ffc`1f652300  0001 (0001)  0:****  KERNEL32!CreateFil\
ew
  1 e Disable Clear  00007ffc`20480120  0001 (0001)  0:****  ntdll!NtCreateFile
```

Notice the native API never uses W or A - it always works with Unicode strings (in fact it expects UNICODE_STRING structures, as we've seen already).

Continue execution with the g command. The debugger should break:

```
Breakpoint 1 hit
ntdll!NtCreateFile:
00007ffc`20480120  4c8bd1          mov     r10,rcx
```

Check the call stack again:

```
0:002> k
# Child-SP          RetAddr          Call Site
00 00000001`226fa938 00007ffc`1c75e5d6 ntdll!NtCreateFile
01 00000001`226fa940 00007ffc`1c75e2c6 KERNELBASE!CreateFileInternal+0x2f6
02 00000001`226faab0 00007ffc`061c8368 KERNELBASE!CreateFileW+0x66
03 00000001`226fab10 00007ffc`061c5d4d mscoree!RuntimeDesc::VerifyMainRuntimeM\
odule+0x2c
04 00000001`226fab60 00007ffc`061c6068 mscoree!FindRuntimesInInstallRoot+0x2fb
```

```
05 00000001`226fb3e0 00007ffc`061cb748 mscoreei!GetOrCreateSxSProcessInfo+0x94
...
```

List the next 8 instructions that are about to be executed with the **u** (unassemble or disassemble) command:

```
0:002> u
ntdll!NtCreateFile:
00007ffc`20480120 4c8bd1      mov     r10,rcx
00007ffc`20480123 b855000000      mov     eax,55h
00007ffc`20480128 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (0000\
0000`7ffe0308)],1
00007ffc`20480130 7503           jne     ntdll!NtCreateFile+0x15 (00007ffc`204\
80135)
00007ffc`20480132 0f05           syscall
00007ffc`20480134 c3             ret
00007ffc`20480135 cd2e           int     2Eh
00007ffc`20480137 c3             ret
```

Notice the value 0x55 is copied to the *EAX* register. This is the system service number for *NtCreateFile*, as described in chapter 1. The *syscall* instruction shown is the one causing the transition to kernel-mode, and then executing the *NtCreateFile* system service itself.

You can step over the next instruction with the **p** command (step - hit *F10* as an alternative). You can step into a function (in case of assembly, this is the *call* instruction) with the **t** command (trace - hit *F11* as an alternative):

```
0:002> p
Breakpoint 1 hit
ntdll!NtCreateFile:
00007ffc`20480120 4c8bd1      mov     r10,rcx
0:002> p
ntdll!NtCreateFile+0x3:
00007ffc`20480123 b855000000      mov     eax,55h
0:002> p
ntdll!NtCreateFile+0x8:
00007ffc`20480128 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (0000\
0000`7ffe0308)],1 ds:00000000`7ffe0308=00
0:002> p
ntdll!NtCreateFile+0x10:
00007ffc`20480130 7503           jne     ntdll!NtCreateFile+0x15 (00007ffc`204\
80135) [br=0]
0:002> p
ntdll!NtCreateFile+0x12:
00007ffc`20480132 0f05           syscall
```

Stepping inside a *syscall* is not possible, as we're in user-mode. When we step over/into it, all is done and we get back a result.

```
0:002> p
ntdll!NtCreateFile+0x14:
00007ffc`20480134 c3          ret
```

The return value of functions in x64 calling convention is stored in *EAX* or *RAX*. For system calls, it's an NTSTATUS, so *EAX* contains the returned status:

```
0:002> r eax
eax=c0000034
```

Zero means success, and a negative value (in two's complement, most significant bit is set) means an error. We can get a textual description of the error with the `!error` command:

```
0:002> !error @eax
Error code: (NTSTATUS) 0xc0000034 (3221225524) - Object Name not found.
```

This means the file wasn't found on the system.

Disable all breakpoints and let *Notepad* continue execution normally:

```
0:002> bd *
0:002> g
```

Since we have no breakpoints at this time, we can force a break by clicking the *Break* button on the toolbar, or hitting *Ctrl+Break* on the keyboard:

```
874c.16a54): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ffc`20483080 cc          int     3
```

Notice the thread number in the prompt. Show all current threads:

```
0:022> ~
 0 Id: 874c.18068 Suspend: 1 Teb: 00000001`2229d000 Unfrozen
 1 Id: 874c.46ac Suspend: 1 Teb: 00000001`222a5000 Unfrozen
 2 Id: 874c.152cc Suspend: 1 Teb: 00000001`222a7000 Unfrozen
 3 Id: 874c.f7ec Suspend: 1 Teb: 00000001`222ad000 Unfrozen
 4 Id: 874c.145b4 Suspend: 1 Teb: 00000001`222af000 Unfrozen
...
18 Id: 874c.f0c4 Suspend: 1 Teb: 00000001`222d1000 Unfrozen
19 Id: 874c.17414 Suspend: 1 Teb: 00000001`222d3000 Unfrozen
20 Id: 874c.c878 Suspend: 1 Teb: 00000001`222d5000 Unfrozen
21 Id: 874c.d8c0 Suspend: 1 Teb: 00000001`222d7000 Unfrozen
. 22 Id: 874c.16a54 Suspend: 1 Teb: 00000001`222e1000 Unfrozen
```

```
23 Id: 874c.10838 Suspend: 1 Teb: 00000001`222db000 Unfrozen
24 Id: 874c.10cf0 Suspend: 1 Teb: 00000001`222dd000 Unfrozen
```

Lots of threads, right? These were created by the common open dialog, so not the direct fault of *Notepad*.

Continue exploring the debugger in any way you want!



Find out the system service numbers for `NtWriteFile` and `NtReadFile`.

If you close *Notepad*, you'll hit a breakpoint at process termination:

```
ntdll!NtTerminateProcess+0x14:
00007ffc`2047fc14 c3          ret
0:000> k
# Child-SP          RetAddr           Call Site
00 00000001`2247f6a8 00007ffc`20446dd8 ntdll!NtTerminateProcess+0x14
01 00000001`2247f6b0 00007ffc`1f64d62a ntdll!RtlExitUserProcess+0xb8
02 00000001`2247f6e0 00007ffc`061cee58 KERNEL32!ExitProcessImplementation+0xa
03 00000001`2247f710 00007ffc`0644719e mscoree!RuntimeDesc::ShutdownAllActiveR\
untimes+0x287
04 00000001`2247fa00 00007ffc`1fcda291 mscoree!ShellShim_CorExitProcess+0x11e
05 00000001`2247fa30 00007ffc`1fcda2ad msvcrt!_crtCorExitProcess+0x4d
06 00000001`2247fa60 00007ffc`1fcda925 msvcrt!_crtExitProcess+0xd
07 00000001`2247fa90 00007ff7`5383ae1e msvcrt!doexit+0x171
08 00000001`2247fb00 00007ffc`1f647974 notepad!__mainCRTStartup+0x1b6
09 00000001`2247fbc0 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
0a 00000001`2247fbf0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

You can use the `q` command to quit the debugger. If the process is still alive, it will be terminated. An alternative is to use the `.detach` command to disconnect from the target without killing it.

Kernel Debugging

User-mode debugging involves the debugger attaching to a process, setting breakpoints that cause the process' threads to become suspended, and so on. Kernel-mode debugging, on the other hand, involves controlling the entire machine with the debugger. This means that if a breakpoint is set and then hit, the entire machine is frozen. Clearly, this cannot be achieved with a single machine. In full kernel debugging, two machines are involved: a host (where

the debugger runs) and a target (being debugged). The target can, however, be a virtual machine hosted on the same machine (host) where the debugger executes. Figure 5-5 shows a host and target connected via some connection medium.

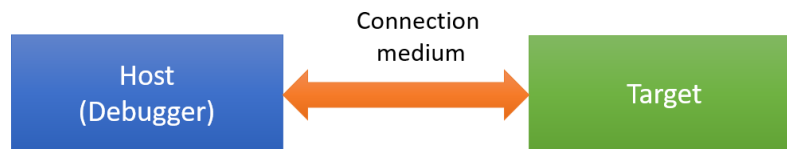


Figure 5-5: Host-target connection

Before we get into full kernel debugging, we'll take a look at its simpler cousin - local kernel debugging.

Local Kernel Debugging

Local kernel debugging (LKD) allows viewing system memory and other system information on the local machine. The primary difference between local and full kernel debugging, is that with LKD there is no way to set up breakpoints, which means you're always looking at the current state of the system. It also means that things change, even while commands are being executed, so some information may be stale or unreliable. With full kernel debugging, commands can only be entered while the target system is in a breakpoint, so system state is unchanged.

To configure LKD, enter the following in an elevated command prompt and then restart the system:

```
bcdedit /debug on
```



Local Kernel Debugging is protected by *Secure Boot* on Windows 10, Server 2016, and later. To activate LKD you'll have to disable Secure Boot in the machine's BIOS settings. If, for whatever reason, this is not possible, there is an alternative using the Sysinternals *LiveKd* tool. Copy *LiveKd.exe* to the *Debugging Tools for Windows* main directory. Then launch *WinDbg* using LiveKd with the following command: `livekd -w`. The experience is not the same, as data may become stale because of the way *Livekd* works, and you may need to exit the debugger and relaunch from time to time.

After the system is restarted, launch *WinDbg* elevated (the 64-bit one, if you are on a 64-bit system). Select the menu *File / Attach To Kernel* (WinDbg preview) or *File / Kernel Debug...* (classic WinDbg). Select the *Local* tab and click *OK*. You should see output similar to the following:

```
Microsoft (R) Windows Debugger Version 10.0.22415.1003 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Connected to Windows 10 22000 x64 target at (Wed Sep 29 10:57:30.682 2021 (UTC \
+ 3:00)), ptr64 TRUE

***** Path validation summary *****
Response                               Time (ms)      Location
Deferred                               SRV*c:\symbols*https://msdl.micr\
osoft.com/download/symbols
Symbol search path is: SRV*c:\symbols*https://msdl.microsoft.com/download/symbo\
ls
Executable search path is:
Windows 10 Kernel Version 22000 MP (6 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Edition build lab: 22000.1.amd64fre.co_release.210604-1628
Machine Name:
Kernel base = 0xffffffff802`07a00000 PsLoadedModuleList = 0xffffffff802`08629710
Debug session time: Wed Sep 29 10:57:30.867 2021 (UTC + 3:00)
System Uptime: 0 days 16:44:39.106
```

Note the prompt displays *lkd*. This indicates Local Kernel Debugging is active.

Local kernel Debugging Tutorial

If you're familiar with kernel debugging commands, you can safely skip this section.

You can display basic information for all processes running on the system with the process `! 0 0` command:

```

lkd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS fffffd104936c8040
  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
  DirBase: 006d5000  ObjectTable: fffffa58d3cc44d00  HandleCount: 3909.
  Image: System

PROCESS fffffd104936e2080
  SessionId: none  Cid: 0058    Peb: 00000000  ParentCid: 0004
  DirBase: 0182c000  ObjectTable: fffffa58d3cc4ea40  HandleCount: 0.
  Image: Secure System

PROCESS fffffd1049370a080
  SessionId: none  Cid: 0090    Peb: 00000000  ParentCid: 0004
  DirBase: 011b6000  ObjectTable: fffffa58d3cc65a80  HandleCount: 0.
  Image: Registry

PROCESS fffffd10497dd0080
  SessionId: none  Cid: 024c    Peb: bc6c2ba000  ParentCid: 0004
  DirBase: 10be4b000  ObjectTable: fffffa58d3d49ddc0  HandleCount: 60.
  Image: smss.exe
...

```

For each process, the following information is displayed:

- The address attached to the *PROCESS* text is the *EPROCESS* address of the process (in kernel space, of course).
- *SessionId* - the session the process is running under.
- *Cid* - (client ID) the unique process ID.
- *Peb* - the address of the *Process Environment Block* (PEB). This address is in user space, naturally.
- *ParentCid* - (parent client ID) the process ID of the parent process. Note that it's possible the parent process no longer exists, so this ID may belong to some process created after the parent process terminated.
- *DirBase* - physical address of the Master Page Directory for this process, used as the basis for virtual to physical address translation. On x64, this is known as *Page Map Level 4*, and on x86 it's *Page Directory Pointer Table* (PDPT).
- *ObjectTable* - pointer to the private handle table for the process.
- *HandleCount* - number of handles in the handle table for this process.
- *Image* - executable name, or special process name for those not associated with an executable (such as *Secure System*, *System*, *Mem Compression*).

The `!process` command accepts at least two arguments. The first indicates the process of interest using its *EPROCESS* address or the unique Process ID, where zero means “all or any process”. The second argument is the level of detail to

display (a bit mask), where zero means the least amount of detail. A third argument can be added to search for a particular executable. Here are a few examples:

List all processes running *explorer.exe*:

```
lkd> !process 0 0 explorer.exe
PROCESS fffffd1049e118080
  SessionId: 1 Cid: 1780 Peb: 0076b000 ParentCid: 16d0
  DirBase: 362ea5000 ObjectTable: fffffa58d45891680 HandleCount: 3208.
  Image: explorer.exe

PROCESS fffffd104a14e2080
  SessionId: 1 Cid: 2548 Peb: 005c1000 ParentCid: 0314
  DirBase: 140fe9000 ObjectTable: fffffa58d46a99500 HandleCount: 2613.
  Image: explorer.exe
```

List more information for a specific process by specifying its address and a higher level of detail:

```
lkd> !process fffffd1049e7a60c0 1
PROCESS fffffd1049e7a60c0
  SessionId: 1 Cid: 1374 Peb: d3e343000 ParentCid: 0314
  DirBase: 37eb97000 ObjectTable: fffffa58d58a9de00 HandleCount: 224.
  Image: dllhost.exe
  VadRoot fffffd104b81c7db0 Vads 94 Clone 0 Private 455. Modified 2. Locked 0.
  DeviceMap fffffa58d41354230
  Token fffffa58d466e0060
  ElapsedTime 01:04:36.652
  UserTime 00:00:00.015
  KernelTime 00:00:00.015
  QuotaPoolUsage[PagedPool] 201696
  QuotaPoolUsage[NonPagedPool] 13048
  Working Set Sizes (now,min,max) (4330, 50, 345) (17320KB, 200KB, 1380KB)
  PeakWorkingSetSize 4581
  VirtualSize 2101383 Mb
  PeakVirtualSize 2101392 Mb
  PageFaultCount 5427
  MemoryPriority BACKGROUND
  BasePriority 8
  CommitCharge 678
  Job fffffd104a05ed380
```

As can be seen from the above output, more information on the process is displayed. Some of this information is hyperlinked, allowing easy further examination. For example, the job this process is part of (if any) is a hyperlink, executing the `!job` command if clicked.

Click on the Job address hyperlink:

```

lkd> !job fffffd104a05ed380
Job at fffffd104a05ed380
  Basic Accounting Information
    TotalUserTime:           0x0
    TotalKernelTime:         0x0
    TotalCycleTime:          0x0
    ThisPeriodTotalUserTime: 0x0
    ThisPeriodTotalKernelTime: 0x0
    TotalPageFaultCount:     0x0
    TotalProcesses:          0x1
    ActiveProcesses:         0x1
    FreezeCount:             0
    BackgroundCount:         0
    TotalTerminatedProcesses: 0x0
    PeakJobMemoryUsed:       0x2f5
    PeakProcessMemoryUsed:   0x2f5
  Job Flags
    [wake notification allocated]
    [wake notification enabled]
    [timers virtualized]
  Limit Information (LimitFlags: 0x800)
  Limit Information (EffectiveLimitFlags: 0x403800)
    JOB_OBJECT_LIMIT_BREAKAWAY_OK

```



A Job is a kernel object that manages one or more processes, for which it can apply various limits and get accounting information. A discussion of jobs is beyond the scope of this book. More information can be found in the *Windows Internals 7th edition part 1* and *Windows 10 System Programming, Part 1* books.

As usual, a command such as `!job` hides some information available in the real data structure. In this case, the type is `EJOB`. Use the command `dt nt!_ejob` with the job address to see all the details.

The PEB of a process can be viewed as well by clicking its hyperlink. This is similar to the `!peb` command used in user mode, but the twist here is that the correct process context must be set first, as the address is in user space. Click the *Peb* hyperlink. You should see something like this:

```

lkd> .process /p fffffd1049e7a60c0; !peb d3e343000
Implicit process is now fffffd104`9e7a60c0
PEB at 00000000d3e343000
  InheritedAddressSpace:    No
  ReadImageFileExecOptions: No
  BeingDebugged:           No
  ImageBaseAddress:         00007ffb661180000
  NtGlobalFlag:             0
  NtGlobalFlag2:            0
  Ldr                      00007ffb37ef9120

```

```

Ldr.Initialized:          Yes
Ldr.InInitializationOrderModuleList: 000001d950004560 . 000001d95005a960
Ldr.InLoadOrderModuleList: 000001d9500046f0 . 000001d95005a940
Ldr.InMemoryOrderModuleList: 000001d950004700 . 000001d95005a950
      Base TimeStamp          Module
ost.exe      7ff661180000 93f44fbf Aug 29 00:12:31 2048 C:\WINDOWS\system32\DLLH\
l.dll        7ffb37d80000 50702a8c Oct 06 15:56:44 2012 C:\WINDOWS\SYSTEM32\ntdl\
EL32.DLL     7ffb36790000 ae0b35b0 Jul 13 01:50:24 2062 C:\WINDOWS\System32\KERN\
...

```

The correct process context is set with the .process meta command, and then the PEB is displayed. This is a general technique you need to use to show memory that is in user space - always make sure the debugger is set to the correct process context.

Execute the !process command again, but with the second bit set for the details:

```

lkd> !process fffffd1049e7a60c0 2
PROCESS fffffd1049e7a60c0
  SessionId: 1  Cid: 1374  Peb: d3e343000  ParentCid: 0314
  DirBase: 37eb97000  ObjectTable: fffffa58d58a9de00  HandleCount: 221.
  Image: dllhost.exe

  THREAD fffffd104a02de080  Cid 1374.022c  Teb: 00000000d3e344000 Win32Thread: \
ffffd104b82ccbb0 WAIT: (UserRequest) UserMode Non-Alertable
      fffffd104b71d2860  SynchronizationEvent

  THREAD fffffd104a45e8080  Cid 1374.0f04  Teb: 00000000d3e352000 Win32Thread: \
ffffd104b82ccd90 WAIT: (WrUserRequest) UserMode Non-Alertable
      fffffd104adc5e0c0  QueueObject

  THREAD fffffd104a229a080  Cid 1374.1ed8  Teb: 00000000d3e358000 Win32Thread: \
ffffd104b82cf900 WAIT: (UserRequest) UserMode Non-Alertable
      fffffd104b71dfb60  NotificationEvent
      fffffd104ad02a740  QueueObject

  THREAD fffffd104b78ee040  Cid 1374.0330  Teb: 00000000d3e37a000 Win32Thread: \
0000000000000000 WAIT: (WrQueue) UserMode Alertable
      fffffd104adc4f640  QueueObject

```

Detail level 2 shows a summary of the threads in the process along with the object(s) they are waiting on (if any).

You can use other detail values (4, 8), or combine them, such as 3 (1 or 2).

Repeat the !process command again, but this time with no detail level. More information is shown for the process (the default in this case is full details):

```
lkd> !process fffffd1049e7a60c0
PROCESS fffffd1049e7a60c0
  SessionId: 1 Cid: 1374 Peb: d3e343000 ParentCid: 0314
  DirBase: 37eb97000 ObjectTable: fffffa58d58a9de00 HandleCount: 223.
  Image: dllhost.exe
  VadRoot fffffd104b81c7db0 Vads 94 Clone 0 Private 452. Modified 2. Locked 0.
  DeviceMap fffffa58d41354230
  Token fffffa58d466e0060
  ElapsedTime 01:10:30.521
  UserTime 00:00:00.015
  KernelTime 00:00:00.015
  QuotaPoolUsage[PagedPool] 201696
  QuotaPoolUsage[NonPagedPool] 13048
  Working Set Sizes (now,min,max) (4329, 50, 345) (17316KB, 200KB, 1380KB)
  PeakWorkingSetSize 4581
  VirtualSize 2101383 Mb
  PeakVirtualSize 2101392 Mb
  PageFaultCount 5442
  MemoryPriority BACKGROUND
  BasePriority 8
  CommitCharge 678
  Job fffffd104a05ed380

  THREAD fffffd104a02de080 Cid 1374.022c Teb: 00000000d3e344000 Win32Thread: \
fffffd104b82ccbb0 WAIT: (UserRequest) UserMode Non-Alertable
  fffffd104b71d2860 SynchronizationEvent
  Not impersonating
  DeviceMap fffffa58d41354230
  Owning Process fffffd1049e7a60c0 Image: dllhost.exe
  Attached Process N/A Image: N/A
  Wait Start TickCount 3641927 Ticks: 270880 (0:01:10:32.500)
  Context Switch Count 27 IdealProcessor: 2
  UserTime 00:00:00.000
  KernelTime 00:00:00.000
  Win32 Start Address 0x00007ff661181310
  Stack Init fffffbe88b4bdf630 Current fffffbe88b4bdf010
  Base fffffbe88b4be0000 Limit fffffbe88b4bd9000 Call 0000000000000000
  Priority 8 BasePriority 2 PriorityDecrement 0 IoPriority 2 PagePriority 5
  Kernel stack not resident.
  THREAD fffffd104a45e8080 Cid 1374.0f04 Teb: 00000000d3e352000 Win32Thread: \
fffffd104b82ccd90 WAIT: (WrUserRequest) UserMode Non-Alertable
  fffffd104adc5e0c0 QueueObject
  Not impersonating
  DeviceMap fffffa58d41354230
  Owning Process fffffd1049e7a60c0 Image: dllhost.exe
  Attached Process N/A Image: N/A
  Wait Start TickCount 3910734 Ticks: 2211 (0:00:00:34.546)
  Context Switch Count 2684 IdealProcessor: 4
  UserTime 00:00:00.046
  KernelTime 00:00:00.078
  Win32 Start Address 0x00007ffb3630f230
  Stack Init fffffbe88b4c87630 Current fffffbe88b4c86a10
  Base fffffbe88b4c88000 Limit fffffbe88b4c81000 Call 0000000000000000
  Priority 10 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
  Child-SP RetAddr Call Site
```

```

ffffbe88`b4c86a50 ffffff802`07c5dc17 nt!KiSwapContext+0x76
ffffbe88`b4c86b90 ffffff802`07c5fac9 nt!KiSwapThread+0x3a7
ffffbe88`b4c86c70 ffffff802`07c59d24 nt!KiCommitThreadWait+0x159
ffffbe88`b4c86d10 ffffff802`07c8ac70 nt!KeWaitForSingleObject+0x234
ffffbe88`b4c86e00 ffffff9da`6d577d46 nt!KeWaitForMultipleObjects+0x540
ffffbe88`b4c86f00 ffffff99c`c175d920 0xfffff9da`6d577d46
ffffbe88`b4c86f08 ffffff99c`c175d920 0xfffff99c`c175d920
ffffbe88`b4c86f10 00000000`00000001 0xfffff99c`c175d920
ffffbe88`b4c86f18 fffffd104`9a423df0 0x1
ffffbe88`b4c86f20 00000000`00000001 0xfffffd104`9a423df0
ffffbe88`b4c86f28 fffffbe88`b4c87100 0x1
ffffbe88`b4c86f30 00000000`00000000 0xffffbe88`b4c87100
...

```

The command lists all threads within the process. Each thread is represented by its ETHREAD address attached to the text “THREAD”. The call stack is listed as well - the module prefix “nt” represents the kernel - there is no need to use the real kernel module name.

One of the reasons to use “nt” instead of explicitly stating the kernel’s module name is because these are different between 64 and 32 bit systems (*ntoskrnl.exe* on 64 bit, and *ntkrnlpa.exe* on 32 bit); and it’s a lot shorter.

User-mode symbols are not loaded by default, so thread stacks that span to user mode show just numeric addresses. You can load user symbols explicitly with `.reload /user` after setting the process context to the process of interest with the `.process` command:

```

lkd> !process 0 0 explorer.exe
PROCESS fffffd1049e118080
  SessionId: 1 Cid: 1780 Peb: 0076b000 ParentCid: 16d0
  DirBase: 362ea5000 ObjectTable: fffffa58d45891680 HandleCount: 3217.
  Image: explorer.exe

PROCESS fffffd104a14e2080
  SessionId: 1 Cid: 2548 Peb: 005c1000 ParentCid: 0314
  DirBase: 140fe9000 ObjectTable: fffffa58d46a99500 HandleCount: 2633.
  Image: explorer.exe

lkd> .process /p fffffd1049e118080
Implicit process is now fffffd104`9e118080
lkd> .reload /user
Loading User Symbols
.....
lkd> !process fffffd1049e118080
PROCESS fffffd1049e118080
  SessionId: 1 Cid: 1780 Peb: 0076b000 ParentCid: 16d0
  DirBase: 362ea5000 ObjectTable: fffffa58d45891680 HandleCount: 3223.
  Image: explorer.exe
...

```

```

THREAD fffffd1049e47c400 Cid 1780.1754 Teb: 000000000078c000 Win32Thread: \
fffffd1049e5da7a0 WAIT: (WrQueue) UserMode Alerttable
fffffd1049e076480 QueueObject
IRP List:
fffffd1049f9bea9b0: (0006,0478) Flags: 00060000 Mdl: 00000000
fffffd1049efd6aa0: (0006,0478) Flags: 00060000 Mdl: 00000000
fffffd1049efee010: (0006,0478) Flags: 00060000 Mdl: 00000000
fffffd1049f3ef8a0: (0006,0478) Flags: 00060000 Mdl: 00000000
Not impersonating
DeviceMap fffffa58d41354230
Owning Process fffffd1049e118080 Image: explorer.exe
Attached Process N/A Image: N/A
Wait Start TickCount 3921033 Ticks: 7089 (0:00:01:50.765)
Context Switch Count 16410 IdealProcessor: 5
UserTime 00:00:00.265
KernelTime 00:00:00.234
Win32 Start Address ntdll!TppWorkerThread (0x00007ffb37d96830)
Stack Init fffffbe88b5fc7630 Current fffffbe88b5fc6d20
Base fffffbe88b5fc8000 Limit fffffbe88b5fc1000 Call 0000000000000000
Priority 9 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP RetAddr Call Site
ffffbe88`b5fc6d60 ffffff802`07c5dc17 nt!KiSwapContext+0x76
ffffbe88`b5fc6ea0 ffffff802`07c5fac9 nt!KiSwapThread+0x3a7
ffffbe88`b5fc6f80 ffffff802`07c62526 nt!KiCommitThreadWait+0x159
ffffbe88`b5fc7020 ffffff802`07c61f38 nt!KeRemoveQueueEx+0x2b6
ffffbe88`b5fc70d0 ffffff802`07c6479c nt!IoRemoveIoCompletion+0x98
ffffbe88`b5fc71f0 ffffff802`07e25075 nt!NtWaitForWorkViaWorkerFactory+0x\
39c fffffbe88`b5fc7430 00007ffb`37e26e84 nt!KiSystemServiceCopyEnd+0x25 (Tra\
pFrame @ fffffbe88`b5fc74a0)
00000000`03def858 00007ffb`37d96b0f ntdll!NtWaitForWorkViaWorkerFactory\
+0x14
00000000`03def860 00007ffb`367a54e0 ntdll!TppWorkerThread+0x2df
00000000`03defb50 00007ffb`37d8485b KERNEL32!BaseThreadInitThunk+0x10
00000000`03defb80 00000000`00000000 ntdll!RtlUserThreadStart+0x2b
...

```

Notice the thread above has issued several IRPs as well. We'll discuss this in greater detail in chapter 7.

A thread's information can be viewed separately with the `!thread` command and the address of the thread. Check the debugger documentation for the description of the various pieces of information displayed by this command.

Other generally useful/interesting commands in kernel-mode debugging include:

- `!pcr` - display the Process Control Region (PCR) for a processor specified as an additional index (processor 0 is displayed by default if no index is specified).
- `!vm` - display memory statistics for the system and processes.

- `!running` - displays information on threads running on all processors on the system.

We'll look at more specific commands useful for debugging drivers in subsequent chapters.

Full Kernel Debugging

Full kernel debugging requires configuration on the host and target. In this section, we'll see how to configure a virtual machine as a target for kernel debugging. This is the recommended and most convenient setup for kernel driver work (when not developing device drivers for hardware). We'll go through the steps for configuring a Hyper-V virtual machine. If you're using a different virtualization technology (e.g. VMWare or VirtualBox), please consult that product's documentation or the web for the correct procedure to get the same results.

The target and host machine^x must communicate using some communication media. There are several options available. The fastest communication option is to use the network. Unfortunately, this requires the host and target to run Windows 8 at a minimum. Since Windows 7 is still a viable target, there is another convenient option - the COM (serial) port, which can be exposed as a named pipe to the host machine. All virtualization platforms allow redirecting a virtual serial port to a named pipe on the host. We'll look at both options.



Just like Local Kernel Debugging, the target machine cannot use *Secure Boot*. With full kernel debugging, there is no workaround.

Using a Virtual Serial Port

In this section, we'll configure the target and host to use a virtual COM port exposed as a named pipe to the host. In the next section, we'll configure kernel debugging using the network.

Configuring the Target

The target VM must be configured for kernel debugging, similar to local kernel debugging, but with the added connection media set to a virtual serial port on that machine.

One way to do the configuration is using *bcdedit* in an elevated command window:

```
bcdedit /debug on  
bcdedit /dbgsettings serial debugport:1 baudrate:115200
```

Change the debug port number according to the actual virtual serial number (typically 1).

The VM must be restarted for these configurations to take effect. Before you do that, we can map the serial port to a named pipe. Here is the procedure for Hyper-V virtual machines:

If the Hyper-V VM is Generation 1 (older), there is a simple UI in the VM's settings to do the configuration. Use the *Add Hardware* option to add a serial port if there are none defined. Then configure the serial port to be mapped to a named port of your choosing. Figure 5-6 shows this dialog.

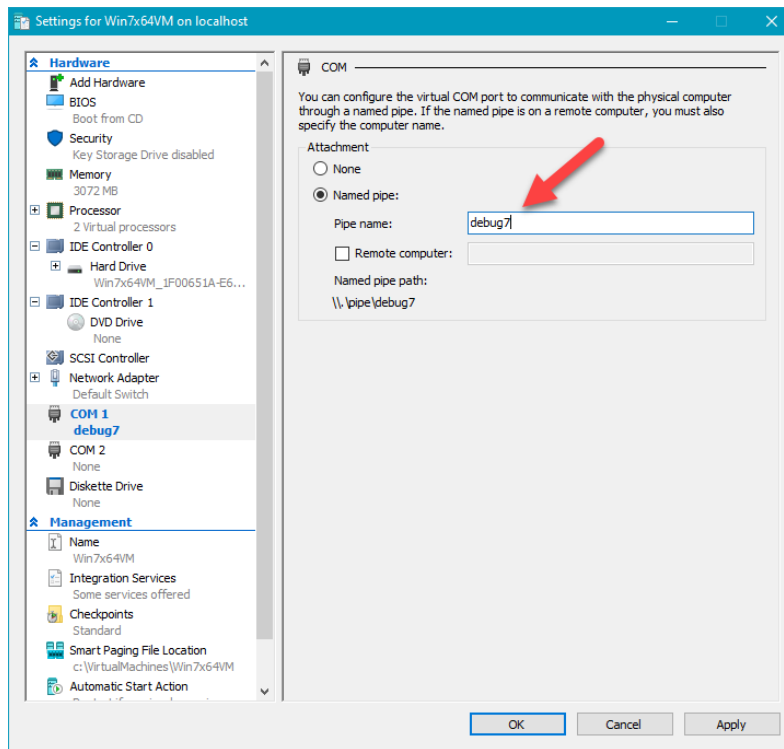


Figure 5-6: Mapping serial port to named pipe for Hyper-V Gen-1 VM

For Generation 2 VMs, no UI is currently available. To configure this, make sure the VM is shut down, and open an elevated *PowerShell* window.

Type the following to set a serial port mapped to a named pipe:

```
PS C:\>Set-VMComPort myvmname -Number 1 -Path "\\.\pipe\debug"
```

Change the VM name appropriately and the COM port number as set inside the VM earlier with *bcdedit*. Make sure the pipe path is unique.

You can verify the settings are as expected with *Get-VMComPort*:

```
PS C:\>Get-VMComPort myvmname
```

VMName	Name	Path
-----	----	----
myvmname	COM 1	\\.\pipe\debug
myvmname	COM 2	

You can boot the VM - the target is now ready.

Configuring the Host

The kernel debugger must be properly configured to connect with the VM on the same serial port mapped to the same named pipe exposed on the host.

Launch the kernel debugger elevated, and select *File / Attach To Kernel*. Navigate to the *COM* tab. Fill in the correct details as they were set on the target. Figure 5-7 shows what these settings look like.

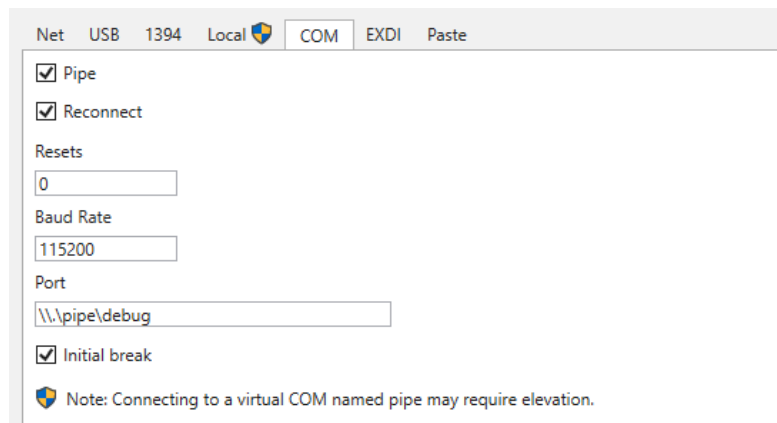


Figure 5-7: Setting host COM port configuration

Click OK. The debugger should attach to the target. If it does not, click the *Break* toolbar button. Here is some typical output:

Microsoft (R) Windows Debugger Version 10.0.18317.1001 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

```
Opened \\.\pipe\debug
Waiting to reconnect...
Connected to Windows 10 18362 x64 target at (Sun Apr 21 11:28:11.300 2019 (UTC \
+ 3:00)), ptr64 TRUE
Kernel Debugger connection established. (Initial Breakpoint requested)
```

```
***** Path validation summary *****
```

```
Response          Time (ms)      Location
Deferred
soft.com/download/symbols
Symbol search path is: SRV*c:\Symbols*http://msdl.micro\
Executable search path is:
Windows 10 Kernel Version 18362 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 18362.1.amd64fre.19h1_release.190318-1202
Machine Name:
Kernel base = 0xfffff801`36a09000 PsLoadedModuleList = 0xfffff801`36e4c2d0
Debug session time: Sun Apr 21 11:28:09.669 2019 (UTC + 3:00)
System Uptime: 1 days 0:12:28.864
```

```

Break instruction exception - code 80000003 (first chance)
*****
*
*   You are seeing this message because you pressed either
*   CTRL+C (if you run console kernel debugger) or,
*   CTRL+BREAK (if you run GUI kernel debugger),
*   on your debugger machine's keyboard.
*
*
*           THIS IS NOT A BUG OR A SYSTEM CRASH
*
* If you did not intend to break into the debugger, press the "g" key, then
* press the "Enter" key now. This message might immediately reappear. If it
* does, press "g" and "Enter" again.
*
*****
nt!DbgBreakPointWithStatus:
fffff801`36bcd580 cc          int      3

```

Note the prompt has an index and the word *kd*. The index is the current processor that induced the break. At this point, the target VM is completely frozen. You can now debug normally, bearing in mind anytime you break somewhere, the entire machine is frozen.

Using the Network

In this section, we'll configure full kernel debugging using the network, focusing on the differences compared to the virtual COM port setup.

Configuring the Target

On the target machine, running with an elevated command window, configure network debugging using the following format with *bcdedit*:

```
bcdedit /dbgsettings net hostip:<ip> port: <port> [key: <key>]
```

The *hostip* must be the IP address of the host accessible from the target. *port* can be any available port on the host, but the documentation recommends working with port 50000 and up. The key is optional. If you don't specify it, the command generates a random key. For example:

```
c:/>bcdedit /dbgsettings net hostip:10.100.102.53 port:51111
Key=1rhvit77hdpv7.rxgwjdvxj7v.312gs2roip4sf.3w25wrjeocobh
```

The alternative is provide your own key for simplicity, which must be in the format *a.b.c.d*. This is acceptable from a security standpoint when working with local virtual machines:

```
c: />bcdedit /dbgsettings net hostip:10.100.102.53 port:51111 key:1.2.3.4
Key=1.2.3.4
```

You can always display the current debug configuration with */dbgsettings* alone:

```
c:\>bcdedit /dbgsettings
key                1.2.3.4
debugtype          NET
hostip             10.100.102.53
port               51111
dhcp               Yes
The operation completed successfully.
```

Finally, restart the target.

Configuring the Host

On the host machine, launch the debugger and select the *File / Attach the Kernel* option (or *File / Kernel Debug...* in the classic *WinDbg*). Navigate to the *NET* tab, and enter the information corresponding to your settings (figure 5-7).

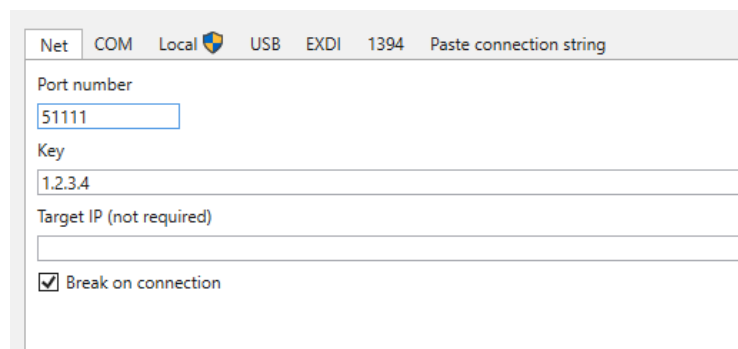


Figure 5-8: Attach to kernel dialog

You may need to click the *Break* button (possibly multiple times) to establish a connection. More information and troubleshooting tips can be found at <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-network-debugging-connection>.

Kernel Driver Debugging Tutorial

Once host and target are connected, debugging can begin. We will use the *Booster* driver we developed in chapter 4 to demonstrate full kernel debugging.

Install (but don't load) the driver on the target as was done in chapter 4. Make sure you copy the driver's PDB file alongside the driver SYS file itself. This simplifies getting correct symbols for the driver.

Let's set a breakpoint in `DriverEntry`. We cannot load the driver just yet because that would cause `DriverEntry` to execute, and we'll miss the chance to set a breakpoint there. Since the driver is not loaded yet, we can use the `bu` command (unresolved breakpoint) to set a future breakpoint. Break into the target if it's currently running, and type the following command in the debugger:

```
0: kd> bu booster!driverentry
0: kd> bl
   0 e Disable Clear u          0001 (0001) (booster!driverentry)
```

The breakpoint is unresolved at this point, since our module (driver) is not yet loaded. The debugger will re-evaluate the breakpoint any time a new module is loaded.

Issue the `g` command to let the target continue execution, and load the driver with `sc start booster` (assuming the driver's name is *booster*). If all goes well, the breakpoint should hit, and the source file should open automatically, showing the following output in the command window:

```
0: kd> g
Breakpoint 0 hit
Booster!DriverEntry:
fffff802`13da11c0 4889542410      mov     qword ptr [rsp+10h],rdx
```



The index on the left of the colon is the CPU index running the code when the breakpoint hit (CPU 0 in the above output).

Figure 5-9 shows a screenshot of *WinDbg Preview* source window automatically opening and the correct line marked. The *Locals* window is also shown as expected.

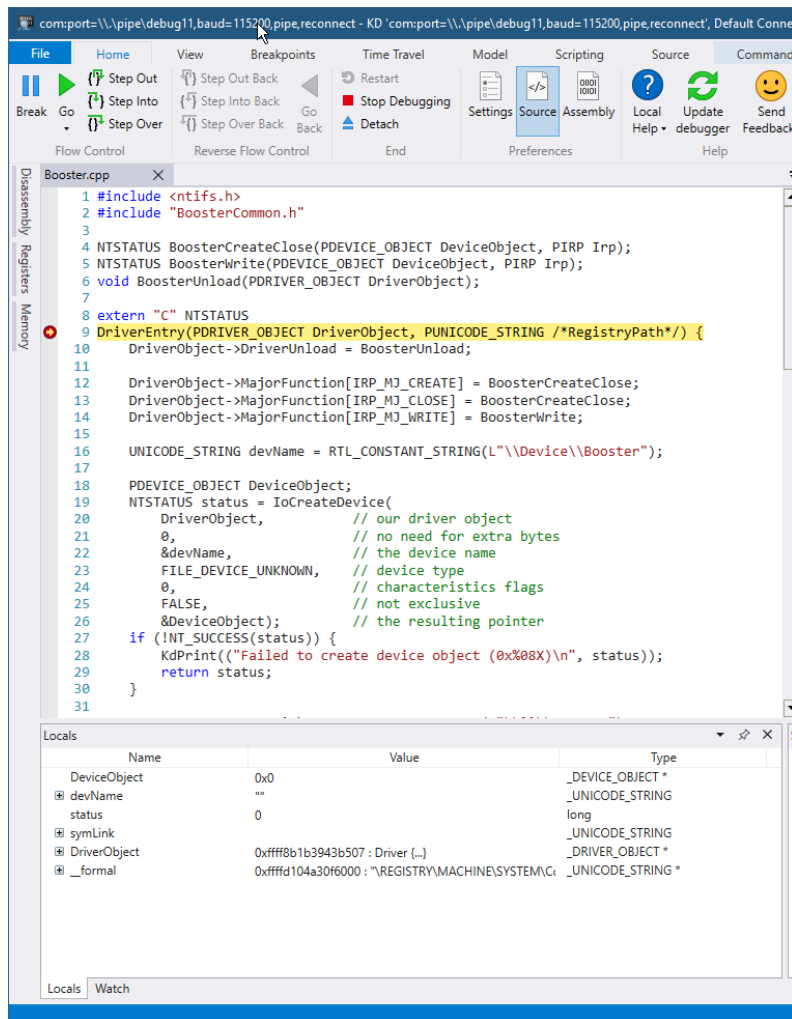


Figure 5-9: Breakpoint hit in DriverEntry

At this point, you can step over source lines, look at variables in the *Locals* window, and even add expressions to the *Watch* window. You can also change values using the *Locals* window just like you would normally do with other debuggers.

The *Command* window is still available as always, but some operations are just easier with the GUI. Setting breakpoints, for example, can be done with the normal bp command, but you can simply open a source file (if it's not already open), go to the line where you want to set a breakpoint, and hit *F9* or click the

appropriate button on the toolbar. Either way, the bp command will be executed in the *Command* window. The *Breakpoints* window can serve as a quick overview of the currently set breakpoints.

- Issue the k command to see how DriverEntry is being invoked:

```
0: kd> k
# Child-SP      RetAddr      Call Site
00 fffffbe88`b3f4f138 ffffff802`13da5020  Booster!DriverEntry [D:\Dev\windowsk\
ernelprogrammingbook2e\Chapter04\Booster\Booster.cpp @ 9]
01 fffffbe88`b3f4f140 ffffff802`081cafc0  Booster!GsDriverEntry+0x20 [minkerne\
l\tools\gs_support\kmode\gs_support.c @ 128]
02 fffffbe88`b3f4f170 ffffff802`080858e2  nt!PnpCallDriverEntry+0x4c
03 fffffbe88`b3f4f1d0 ffffff802`081aeab7  nt!IopLoadDriver+0x8ba
04 fffffbe88`b3f4f380 ffffff802`07c48aaf  nt!IopLoadUnloadDriver+0x57
05 fffffbe88`b3f4f3c0 ffffff802`07d5b615  nt!ExpWorkerThread+0x14f
06 fffffbe88`b3f4f5b0 ffffff802`07e16c24  nt!PspSystemThreadStartup+0x55
07 fffffbe88`b3f4f600 00000000`00000000  nt!KiStartSystemThread+0x34
```



If breakpoints fail to hit, it may be a symbols issue. Execute the .reload command and see if the issues are resolved. Setting breakpoints in user space is also possible, but first execute .reload /user to force the debugger to load user-mode symbols.

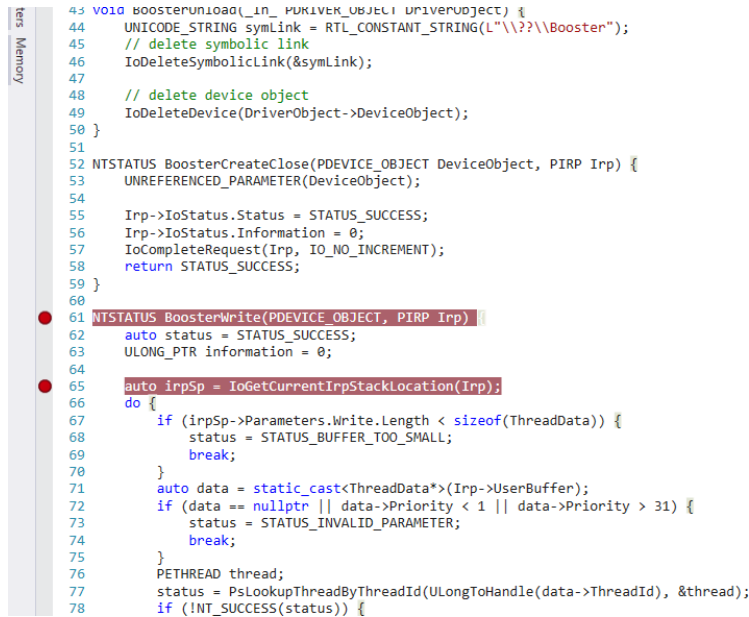
It may be the case that a breakpoint should hit only when a specific process is the one executing the code. This can be done by adding the /p switch to a breakpoint. In the following example, a breakpoint is set only if the process is a specific *explorer.exe*:

```
0: kd> !process 0 0 explorer.exe
PROCESS fffffd1049e118080
  SessionId: 1  Cid: 1780  Peb: 0076b000  ParentCid: 16d0
  DirBase: 362ea5000  ObjectTable: fffffa58d45891680  HandleCount: 3918.
  Image: explorer.exe

PROCESS fffffd104a14e2080
  SessionId: 1  Cid: 2548  Peb: 005c1000  ParentCid: 0314
  DirBase: 140fe9000  ObjectTable: fffffa58d46a99500  HandleCount: 4524.
  Image: explorer.exe

0: kd> bp /p fffffd1049e118080 booster!boosterwrite
0: kd> bl
0 e Disable Clear ffffff802`13da11c0 [D:\Dev\Chapter04\Booster\Booster.cp\
p @ 9] 0001 (0001) Booster!DriverEntry
1 e Disable Clear ffffff802`13da1090 [D:\Dev\Chapter04\Booster\Booster.cp\
p @ 61] 0001 (0001) Booster!BoosterWrite
Match process data fffffd104`9e118080
```

Let's set a normal breakpoint somewhere in the `BoosterWrite` function, by hitting *F9* on the line in source view, as shown in figure 5-10 (the earlier conditional breakpoint is shown as well).



```

43 void BoosterUnload(_IN_ PUNICODE_STRING DriverObject) {
44     UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\?\\Booster");
45     // delete symbolic link
46     IoDeleteSymbolicLink(&symLink);
47
48     // delete device object
49     IoDeleteDevice(DriverObject->DeviceObject);
50 }
51
52 NTSTATUS BoosterCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
53     UNREFERENCED_PARAMETER(DeviceObject);
54
55     Irp->IoStatus.Status = STATUS_SUCCESS;
56     Irp->IoStatus.Information = 0;
57     IoCompleteRequest(Irp, IO_NO_INCREMENT);
58     return STATUS_SUCCESS;
59 }
60
61 NTSTATUS BoosterWrite(PDEVICE_OBJECT, PIRP Irp) {
62     auto status = STATUS_SUCCESS;
63     ULONG_PTR information = 0;
64
65     auto irpSp = IoGetCurrentIrpStackLocation(Irp);
66     do {
67         if (irpSp->Parameters.Write.Length < sizeof(ThreadData)) {
68             status = STATUS_BUFFER_TOO_SMALL;
69             break;
70         }
71         auto data = static_cast<ThreadData*>(Irp->UserBuffer);
72         if (data == nullptr || data->Priority < 1 || data->Priority > 31) {
73             status = STATUS_INVALID_PARAMETER;
74             break;
75         }
76         PETHREAD thread;
77         status = PsLookupThreadByThreadId(ULONGToHandle(data->ThreadId), &thread);
78         if (!NT_SUCCESS(status)) {

```

Figure 5-10: Breakpoint hit in `DriverEntry`

Listing the breakpoints reflect the new breakpoint with the offset calculated by the debugger:

```

0: kd> bl
0 e Disable Clear fffff802`13da11c0 [D:\Dev\Chapter04\Booster\Booster.cpp @\
9] 0001 (0001) Booster!DriverEntry
1 e Disable Clear fffff802`13da1090 [D:\Dev\Chapter04\Booster\Booster.cpp @\
61] 0001 (0001) Booster!BoosterWrite
Match process data fffffd104`9e118080
2 e Disable Clear fffff802`13da10af [D:\Dev\Chapter04\Booster\Booster.cpp @\
65] 0001 (0001) Booster!BoosterWrite+0x1f

```

Enter the *g* command to release the target, and then run the *boost* application with some thread ID and priority:

```
c:\Test> boost 5964 30
```

The breakpoint within `BoosterWrite` should hit:

```

Breakpoint 2 hit
Booster!BoosterWrite+0x1f:

```

```
ffffff802`13da10af 488b4c2468      mov     rcx,qword ptr [rsp+68h]
```

You can continue debugging normally, looking at local variables, stepping over/into functions, etc.

Finally, if you would like to disconnect from the target, enter the `.detach` command. If it does not resume the target, click the *Stop Debugging* toolbar button (you may need to click it multiple times).

Asserts and Tracing

Although using a debugger is sometimes necessary, some coding can go a long way in making a debugger less needed. In this section we'll examine asserts and powerful logging that is suitable for both debug and release builds of a driver.

Asserts

Just like in user mode, asserts can be used to verify that certain assumptions are correct. An invalid assumption means something is very wrong, so it's best to stop. The WDK header provides the `NT_ASSERT` macro for this purpose.

`NT_ASSERT` accepts something that can be converted to a Boolean value. If the result is non-zero (true), execution continues. Otherwise, the assertion has failed, and the system takes one of the following actions:

- If a kernel debugger is attached, an assertion failure breakpoint is raised, allowing debugging the assertion.
- If a kernel debugger is not attached, the system bugchecks. The resulting dump file will pinpoint the exact line where the assertion has failed.

Here is a simple assert usage added to the `DriverEntry` function in the *Booster* driver from chapter 4:

```
extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    DriverObject->DriverUnload = BoosterUnload;

    DriverObject->MajorFunction[IRP_MJ_CREATE] = BoosterCreateClose;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = BoosterCreateClose;
    DriverObject->MajorFunction[IRP_MJ_WRITE] = BoosterWrite;
```

```

UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\Booster");

PDEVICE_OBJECT DeviceObject;
NTSTATUS status = IoCreateDevice(
    DriverObject,           // our driver object
    0,                     // no need for extra bytes
    &devName,               // the device name
    FILE_DEVICE_UNKNOWN,   // device type
    0,                     // characteristics flags
    FALSE,                 // not exclusive
    &DeviceObject);        // the resulting pointer
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to create device object (0x%08X)\n", status));
    return status;
}

NT_ASSERT(DeviceObject);

UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\Booster");
status = IoCreateSymbolicLink(&symLink, &devName);
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to create symbolic link (0x%08X)\n", status));
    IoDeleteDevice(DeviceObject);
    return status;
}

NT_ASSERT(NT_SUCCESS(status));
return STATUS_SUCCESS;
}

```

The first assert makes sure the device object pointer is non-NULL:

```
NT_ASSERT(DeviceObject);
```

The second makes sure the status at the end of DriverEntry is a successful one:

```
NT_ASSERT(NT_SUCCESS(status));
```

NT_ASSERT only compiles its expression in Debug builds, which makes using asserts practically free from a performance standpoint, as these will not be part of the final released driver. This also means you need to be careful that the expression inside NT_ASSERT has no side effects. For example, the following code is wrong:

```
NT_ASSERT(NT_SUCCESS(IoCreateSymbolicLink(...)));
```

This is because the call to IoCreateSymbolicLink will disappear completely in *Release* build. The correct way to assert would be something like the following:

```
status = IoCreateSymbolicLink(...);  
NT_ASSERT(NT_SUCCESS(status));
```

Asserts are useful and should be used liberally because they only have an effect in *Debug* builds.

Extended DbgPrint

We've seen usage of the DbgPrint function (and the KdPrint macro) to generate output that can be viewed with the kernel debugger or a comparable tool, such as *DebugView*. This works, and is simple to use, but has some significant downsides:

- All the output is generated - there is no easy way to filter output to show just some output (such as errors and warnings only). This is partially mitigated with the extended DbgPrintEx function described in the next paragraph.
- DbgPrint(Ex) is a relatively slow function, which is why it's mostly used with KdPrint so that the overhead is removed in *Release* builds. But output in *Release* builds could be very important. Some bugs may only happen in Release builds, where good output could be useful for diagnosing issues.
- There is no semantic meaning associated with DbgPrint - it's just text. There is no way to add values with property name or type information.
- There is no built-in way to save the output to a file rather than just see it in the debugger. if using *DebugView*, it allows saving its output to a file.



The output from DbgPrint(Ex) is limited to 512 bytes. Any remaining bytes are lost.

The DbgPrintEx function (and the associated KdPrintEx macro) were added to provide some filtering support for DbgPrint output:

```
ULONG DbgPrintEx (  
    _In_ ULONG ComponentId,  
    _In_ ULONG Level,  
    _In_z_ _Printf_format_string_ PCSTR Format,  
    ...);           // any number of args
```

A list of component Ids is present in the `<dpfilter.h>` header (common to user and kernel mode), currently containing 155 valid values (0 to 154). Most values are used by the kernel and Microsoft drivers, except for a handful that are meant to be used by third-party drivers:

- `DPFLTR_IHVVIDEO_ID` (78) - for video drivers.
- `DPFLTR_IHVAUDIO_ID` (79) - for audio drivers.
- `DPFLTR_IHVNETWORK_ID` (80) - for network drivers.
- `DPFLTR_IHVSTREAMING_ID` (81) - for streaming drivers.
- `DPFLTR_IHVBUS_ID` (82) - for bus drivers.
- `DPFLTR_IHVDRIVER_ID` (77) - for all other drivers.
- `DPFLTR_DEFAULT_ID` (101) - used with `DbgPrint` or if an illegal component number is used.

For most drivers, the `DPFLTR_IHVDRIVER_ID` component ID should be used.

The `Level` parameter indicates the severity of the message (error, warning, information, etc.), but can technically mean anything you want. The interpretation of this value depends on whether the value is between 0 and 31, or greater than 31:

- 0 to 31 - the level is a single bit formed by the expression `1 << Level`. For example, if `Level` is 5, then the value is 32.
- Anything greater than 31 - the value is used as is.

`<dpfilter.h>` defines a few constants that can be used as is for `Level`:

```
#define DPFLTR_ERROR_LEVEL    0
#define DPFLTR_WARNING_LEVEL  1
#define DPFLTR_TRACE_LEVEL    2
#define DPFLTR_INFO_LEVEL     3
```

You can define more (or different) values as needed. The final result of whether the output will make its way to its destination depends on the component ID, the bit mask formed by the `Level` argument, and on a global mask read from the *Debug Print Filter* Registry key at system startup. Since the *Debug Print Filter* key does not exist by default, there is a default value for all component IDs, which is zero. This means that actual level value is 1 (`1 << 0`). The output will go through if either of the following conditions is true (`value` is the value specified by the `Level` argument to `DbgPrintEx`):

- If value & (Debug print Filter value for that component) is non-zero, the output goes through. With the default, it's (value & 1) != 0.
- If the result of the value ANDed with the Level of the ComponentId is non-zero, the output goes through.

If neither is true, the output is dropped.

Setting the component ID level can be done in one of three ways:

- Using the *Debug Print Filter* key under *HKLM\System\CCS\Control\Session Manager*. DWORD values can be specified where their name is the macro name of a component ID without the prefix or suffix. For example, for DPFLTR_IHVVIDEO_ID, you would set the name to "IHVVIDEO".
- If a kernel debugger is connected, the level of a component can be changed during debugging. For example, the following command changes the level of DPFLTR_IHVVIDEO_ID to 0x1ff:

```
ed Kd_IHVVIDEO_Mask 0x1ff
```



The *Debug Print Filter* value can also be changed with the kernel debugger by using the global kernel variable `Kd_WIN2000_Mask`.

- The last option is to make the change through the `NtSetDebugFilterState` native API. It's undocumented, but it may be useful in practice. The *Dbgkflt* tool, available in the *Tools* folder in the book's samples repository, makes use of this API (and its query counterpart, `NtQueryDebugFilterState`), so that changes can be made even if a kernel debugger is not attached.

If `NtSetDebugFilterState` is called from user mode, the caller must have the *Debug* privilege in its token. Since administrators have this privilege by default (but not non-admin users), you must run *dbgkflt* from an elevated command window for the change to succeed.



The kernel-mode APIs provided by the `<wdm.h>` are `DbgQueryDebugFilterState` and `DbgSetDebugFilterState`. These are still undocumented, but at least their declaration is available. They use the same parameters and return type as their native invokers. This means you can call these APIs from the driver itself if desired (perhaps based on configuration read from the Registry).

Using *Dbgkflt*

Running *Dbgkflt* with no arguments shows its usage.

To query the effective level of a given component, add the component name (without the prefix or suffix). For example:

```
dbgkflt default
```

This returns the effective bits for the `DPFLTR_DEFAULT_ID` component. To change the value to something else, specify the value you want. It's always ORed with `0x80000000` so that the bits you specify are directly used, rather than interpreting numbers lower than 32 as `(1 << number)`. For example, the following sets the first 4 bits for the `DEFAULT` component:

```
dbgkflt default 0xf
```

`DbgPrint` is just a shortcut that calls `DbgPrintEx` with the `DPFLTR_DEFAULT_ID` component like so (this is conceptual and will not compile):

```
ULONG DbgPrint (PCSTR Format, arguments) {  
    return DbgPrintEx(DPFLTR_DEFAULT_ID, DPFLTR_INFO_LEVEL, Format, arguments);  
}
```

This explains why the `DWORD` named `DEFAULT` with a value of 8 (`1 << DPFLTR_INFO_LEVEL`) is the value to write in the Registry to get `DbgPrint` output to go through.

Given the above details, a driver can use `DbgPrintEx` (or the `KdPrintEx` macro) to specify different levels so that output can be filtered as needed. Each call, however, may be somewhat verbose. For example:

```
DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL,  
    "Booster: DriverEntry called. Registry Path: %wZ\n", RegistryPath);
```

Obviously, we might prefer a simpler function that always uses DPFLTR_IHVDRIVER_ID (the one that should be used for generic third-party drivers), like so:

```
Log(DPFLTR_INFO_LEVEL,  
    "Booster: DriverEntry called. Registry Path: %wZ\n", RegistryPath);
```

We can go even further by defining specific functions that use the a log level implicitly:

```
LogInfo("Booster: DriverEntry called. Registry Path: %wZ\n", RegistryPath);
```

Here is an example where we define several bits to be used by creating an enumeration (there is no necessity to used the defined ones):

```
enum class LogLevel {  
    Error = 0,  
    Warning,  
    Information,  
    Debug,  
    Verbose  
};
```

Each value is associated with a small number (below 32), so that the values are interpreted as powers of two by DbgPrintEx. Now we can define functions like the following:

```
ULONG Log(LogLevel level, PCSTR format, ...);  
ULONG LogError(PCSTR format, ...);  
ULONG LogWarning(PCSTR format, ...);  
ULONG LogInfo(PCSTR format, ...);  
ULONG LogDebug(PCSTR format, ...);
```

and so on. Log is the most generic function, while the others use a predefined log level. Here is the implementation of the first two functions:

```
#include <stdarg.h>  
  
ULONG Log(LogLevel level, PCSTR format, ...) {  
    va_list list;  
    va_start(list, format);  
    return vDbgPrintEx(DPFLTR_IHVDRIVER_ID,  
        static_cast<ULONG>(level), format, list);  
}  
  
ULONG LogError(PCSTR format, ...) {
```



```
va_list list;
va_start(list, format);
return vDbgPrintEx(DPFLTR_IHVDRIVER_ID,
    static_cast<ULONG>(LogLevel::Error), format, list);
}
```



The use of `static_cast` in the above code is required in C++, as scoped enums don't automatically convert to integers. You can use a C-style cast instead, if you prefer. If you're using pure C, change the scoped enum to a standard enum (remove the `class` keyword).



The return value from the various `DbgPrint` variants is typed as a `ULONG`, but is in fact a standard `NTSTATUS`.

The implementation uses the classic C variable arguments ellipsis (`...`) and implements these as you would in standard C. The implementation calls `vDbgPrintEx` that accepts a `va_list`, which is necessary for this to work correctly.



It's possible to create something more elaborate using the C++ variadic template feature. This is left as an exercise to the interested (and enthusiastic) reader.

The above code can be found in the *Booster2* project, part of the samples for this chapter. As part of that project, here are a few examples where these functions are used:

```
// in DriverEntry
Log(LogLevel::Information, "Booster2: DriverEntry called. Registry Path: %wZ\n" \
,
    RegistryPath);

// unload routine
LogInfo("Booster2: unload called\n");

// when an error is encountered creating a device object
LogError("Failed to create device object (0x%08X)\n", status);

// error locating thread ID
```

```
LogError("Failed to locate thread %u (0x%X)\n",
        data->ThreadId, status);

// success in changing thread priority
LogInfo("Priority for thread %u changed from %d to %d\n",
        data->ThreadId, oldPriority, data->Priority);
```

Other Debugging Functions

The previous section used `vDebugPrintEx`, defined like so:

```
ULONG vDbgPrintEx(
    _In_ ULONG ComponentId,
    _In_ ULONG Level,
    _In_z_ PCCH Format,
    _In_ va_list arglist);
```

It's identical to `DbgPrintEx`, except its last argument is an already constructed `va_list`. A wrapper macro exists as well - `vKdPrintEx` (compiled in *Debug* builds only).

Lastly, there is yet another extended function for printing - `cDbgPrintExWithPrefix`:

```
ULONG vDbgPrintExWithPrefix (
    _In_z_ PCCH Prefix,
    _In_ ULONG ComponentId,
    _In_ ULONG Level,
    _In_z_ PCCH Format,
    _In_ va_list arglist);
```

It adds a prefix (first parameter) to the output. This is useful to distinguish our driver from other drivers using the same functions. It also allows easy filtering in tools such as *DebugView*. For example, this code snippet shown earlier uses an explicit prefix:

```
LogInfo("Booster2: unload called\n");
```

We can define one as a macro, and use it as the first word in any output like so:

```
#define DRIVER_PREFIX "Booster2: "

LogInfo(DRIVER_PREFIX "unload called\n");
```

This works, but it could be nicer by adding the prefix in every call automatically, by calling `vDbgPrintExWithPrefix` instead of `vDbgPrintEx` in the `Log` implementations. For example:

```
ULONG Log(LogLevel level, PCSTR format, ...) {  
    va_list list;  
    va_start(list, format);  
    return vDbgPrintExWithPrefix("Booster2", DPFLTR_IHVDRIVER_ID,  
        static_cast<ULONG>(level), format, list);  
}
```



Complete the implementation of the `Log` functions variants.

Trace Logging

Using `DbgPrint` and its variants is convenient enough, but as discussed earlier has some drawbacks. Trace logging is a powerful alternative (or complementary) that uses *Event Tracing for Windows* (ETW) for logging purposes, that can be captured live or to a log file. ETW has the additional benefits of being performant (can be used to log thousands of events per second without any noticeable delay), and has semantic information not available with the simple strings generated by the `DbgPrint` functions.



Trace logging can be used in exactly the same way in user mode as well.



ETW is beyond the scope of this book. You can find more information in the official documentation or in my book “Windows 10 System Programming, Part 2”.

To get started with trace logging, an ETW provider has to be defined. Contrary to “classic” ETW, no provider registration is necessary, as trace logging ensures the even metadata is part of the logged information, and as such is self-contained.

A provider must have a unique GUID. You can generate one with the *Create GUID* tool available with Visual Studio (*Tools* menu). Figure 5-11 shows a screenshot of the tool with the second radio button selected, as it's the closest to the format we need. Click the *Copy* button to copy that text to the clipboard.

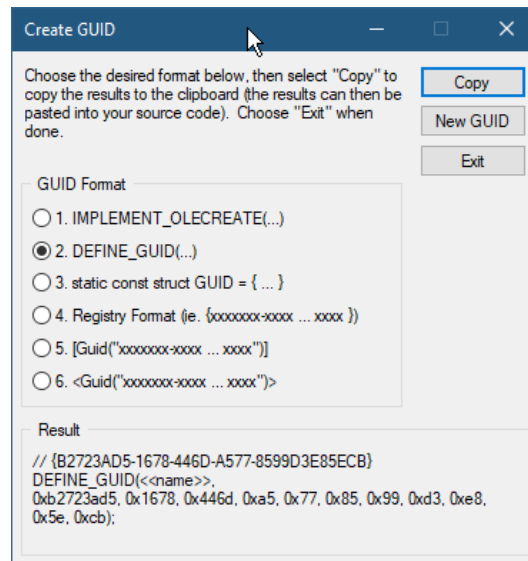


Figure 5-11: The *Create GUID* tool

Paste the text to the main source file of the driver and change the pasted macro to `TRACELOGGING_DEFINE_PROVIDER` to look like this:

```
// {B2723AD5-1678-446D-A577-8599D3E85ECB}
TRACELOGGING_DEFINE_PROVIDER(g_Provider, "Booster", \
    (0xb2723ad5, 0x1678, 0x446d, 0xa5, 0x77, 0x85, 0x99, 0xd3, 0xe8, 0x5e, 0xcb\
));
```

`g_Provider` is a global variable created to represent the ETW provider, where “Booster” is set as its friendly name.

You will need to add the following `#includes` (these are common with user-mode):

```
#include <TraceLoggingProvider.h>
#include <evntrace.h>
```

In `DriverEntry`, call `TraceLoggingRegister` to register the provider:

```
TraceLoggingRegister(g_Provider);
```

Similarly, the provider should be deregistered in the unload routine like so:

```
TraceLoggingUnregister(g_Provider);
```

The logging is done with the `TraceLoggingWrite` macro that is provided a variable number of arguments using another set of macros that provide convenient usage for typed properties. Here is an example of a logging call in `DriverEntry`:

```
TraceLoggingWrite(g_Provider, "DriverEntry started",    // provider, event name
    TraceLoggingLevel(TRACE_LEVEL_INFORMATION),        // log level
    TraceLoggingValue("Booster Driver", "DriverName"), // value, name
    TraceLoggingUnicodeString(RegistryPath, "RegistryPath")); // value, name
```

The above call means the following:

- Use the provider described by `g_Provider`.
- The event name is “DriverEntry started”.
- The logging level is *Information* (several levels are defined).
- A property named “DriverName” has the value “Boster Driver”.
- A property named “RegistryPath” has the value of the `RegistryPath` variable.

Notice the usage of the `TraceLoggingValue` macro - it’s the most generic and uses the type inferred by the first argument (the value). Many other type-safe macros exist, such as the `TraceLoggingUnicodeString` macro above that ensures its first argument is indeed a `UNICODE_STRING`.

Here is another example - if symbolic link creation fails:

```
TraceLoggingWrite(g_Provider, "Error",
    TraceLoggingLevel(TRACE_LEVEL_ERROR),
    TraceLoggingValue("Symbolic link creation failed", "Message"),
    TraceLoggingNTSTATUS(status, "Status", "Returned status"));
```

You can use any “properties” you want. Try to provide the most important details for the event.

Here are a couple of more examples, taken from the *Booster* project part of the samples for this chapter:

```
// Create/Close dispatch IRP
TraceLoggingWrite(g_Provider, "Create/Close",
    TraceLoggingLevel(TRACE_LEVEL_INFORMATION),
    TraceLoggingValue(
        IoGetCurrentIrpStackLocation(Irp)->MajorFunction == IRP_MJ_CREATE ?
        "Create" : "Close", "Operation"));

// success in changing priority
TraceLoggingWrite(g_Provider, "Boosting",
    TraceLoggingLevel(TRACE_LEVEL_INFORMATION),
    TraceLoggingUInt32(data->ThreadId, "ThreadId"),
    TraceLoggingInt32(oldPriority, "OldPriority"),
    TraceLoggingInt32(data->Priority, "NewPriority"));
```

Viewing ETW Traces

Where do all the above traces go to? Normally, they are just dropped. Someone has to configure listening to the provider and log the events to a real-time session or a file. The WDK provides a tool called *TraceView* that can be used for just that purpose.

You can open a Developer's Command window and run *TraceView.exe* directly. If you can't locate it, it's installed by default in a directory such as *C:\Program Files (x86)\Windows Kits\10\bin\10.0.22000.0\x64*.

You can copy the executable to the target machine where the driver is supposed to run. When you run *TraceView.exe*, an empty window is shown (figure 5-12).

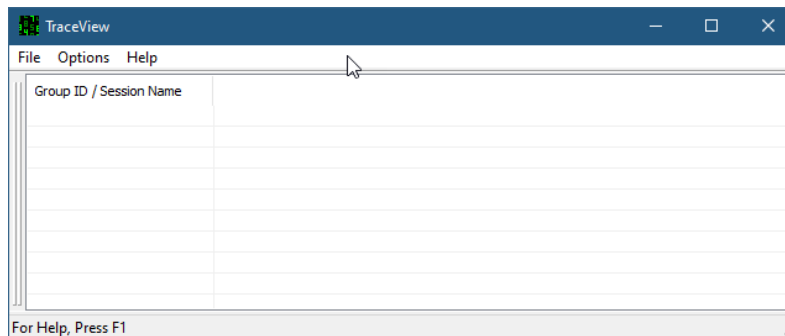


Figure 5-12: The *TraceView.exe* tool

Select the *File / Create New log Session* menu to create a new session. This opens up the dialog shown in figure 5-13.

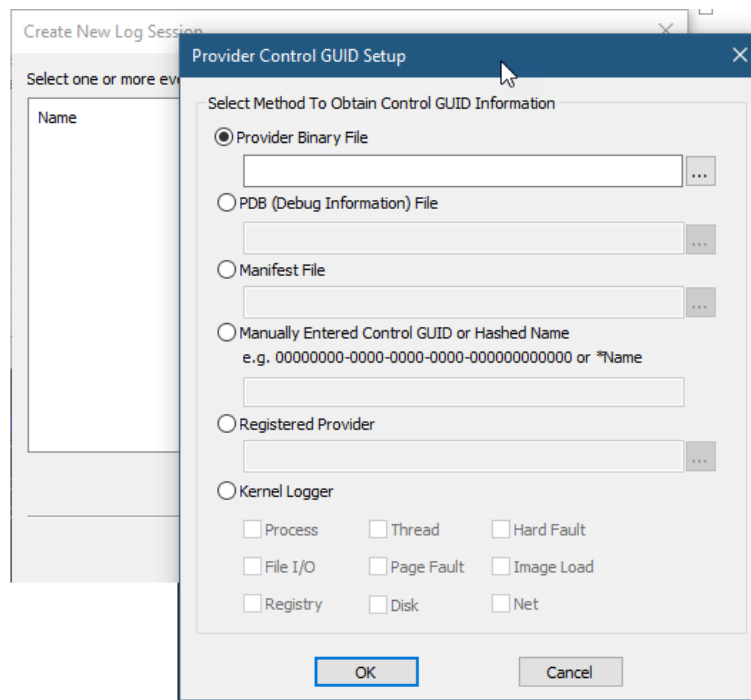


Figure 5-13: New session dialog with a new provider

TraceView provides several methods of locating providers. We can add multiple providers to the same session to get information from other components in the system. For now, we'll add our provider by using the *Manually Entered Control GUID* option, and type in our GUID (figure 5-14):

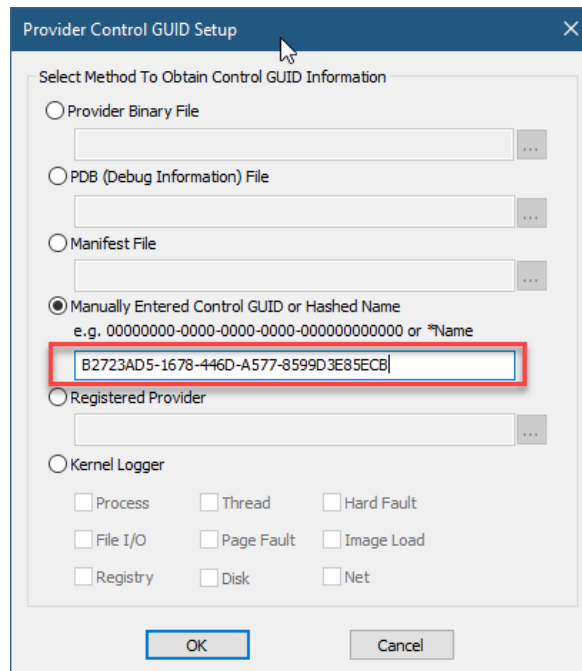


Figure 5-14: Adding a provider GUID manually

Click *OK*. A dialog will pop up asking the source for decoding information. Use the default *Auto* option, as trace logging does not require any outside source. You'll see the single provider in the *Create New Log Session* dialog. Click the *Next* button. The last step of the wizard allows you to select where the output should go to: a real-time session (shown with *TraceView*), a file, or both (figure 5-15).

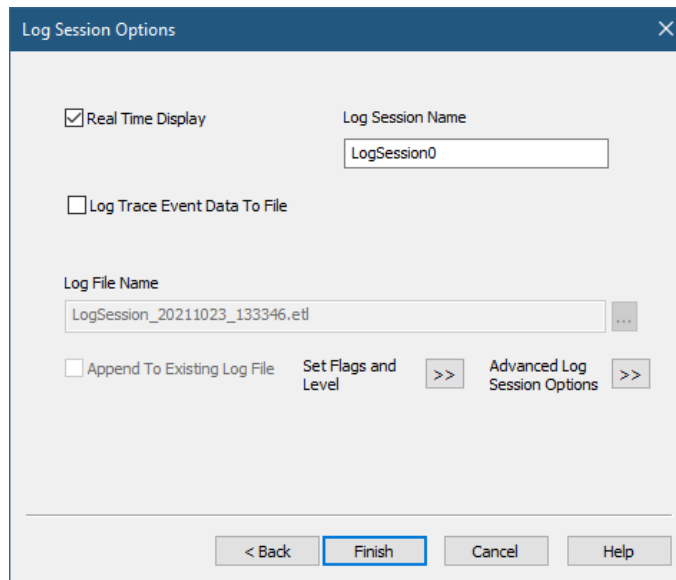


Figure 5-15: Output selection for a session

Click *Finish*. Now you can load/use the driver normally. You should see the output generated in the main *TraceView* window (figure 5-16).

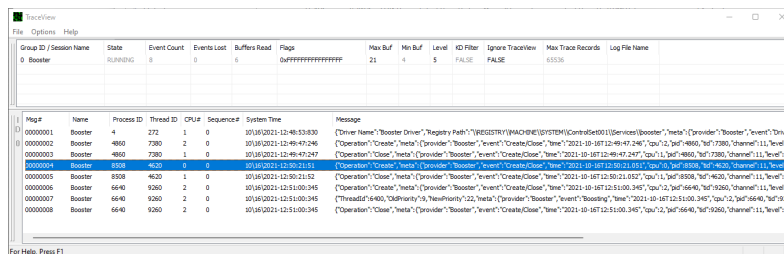


Figure 5-16: ETW real-time session in action

You can see the various properties shown in the *Message* column. When logging to a file, you can open the file later with *TraceView* and see what was logged.

There are other ways to use *TraceView*, and other tools to record and view ETW information. You could also write your own tools to parse the ETW log, as the events have semantic information and so can easily be analyzed.

Summary

In this chapter, we looked at the basics of debugging with *WinDbg*, as well as tracing activities within the driver. Debugging is an essential skill to develop, as software of all kinds, including kernel drivers, may have bugs.

In the next chapter, we'll delve into some kernel mechanisms we need to get acquainted with, as these come up frequently while developing and debugging drivers.

Chapter 6: Kernel Mechanisms

This chapter discussed various mechanisms the Windows kernel provides. Some of these are directly useful for driver writers. Others are mechanisms that a driver developer needs to understand as it helps with debugging and general understanding of activities in the system.

In this chapter:

- **Interrupt Request Level**
 - **Deferred Procedure Calls**
 - **Asynchronous Procedure Calls**
 - **Structured Exception Handling**
 - **System Crash**
 - **Thread Synchronization**
 - **High IRQL Synchronization**
 - **Work Items**
-

Interrupt Request Level (IRQL)

In chapter 1, we discussed threads and thread priorities. These priorities are taken into consideration when more threads want to execute than there are available processors. At the same time, hardware devices need to notify the system that something requires attention. A simple example is an I/O operation that is carried out by a disk drive. Once the operation completes, the disk drive notifies completion by requesting an interrupt. This interrupt is connected to an Interrupt Controller hardware that then sends the request to a processor for handling. The next question is, which thread should execute the associated Interrupt Service Routine (ISR)?

Every hardware interrupt is associated with a priority, called *Interrupt Request Level* (IRQL) (not to be confused with an interrupt physical line known as IRQ), determined by the HAL. Each processor's context has its own IRQL, just like any register. IRQLs may or may not be implemented by the CPU hardware, but this is essentially unimportant. IRQL should be treated just like any other CPU register.

The basic rule is that a processor executes the code with the highest IRQL. For example, if a CPU's IRQL is zero at some point, and an interrupt with an associated IRQL of 5 comes in, it will save its state (context) in the current thread's kernel stack, raise its IRQL to 5 and then execute the ISR associated with the interrupt. Once the ISR completes, the IRQL will drop to its previous level, resuming the previously executed code as though the interrupt never happened. While the ISR is executing, other interrupts coming in with an IRQL of 5 or less cannot interrupt this processor. If, on the other hand, the IRQL of the new interrupt is above 5, the CPU will save its state again, raise IRQL to the new level, execute the second ISR associated with the second interrupt and when completed, will drop back to IRQL 5, restore its state and continue executing the original ISR. Essentially, raising IRQL blocks code with equal or lower IRQL temporarily.

The basic sequence of events when an interrupt occurs is depicted in figure 6-1. Figure 6-2 shows what interrupt nesting looks like.

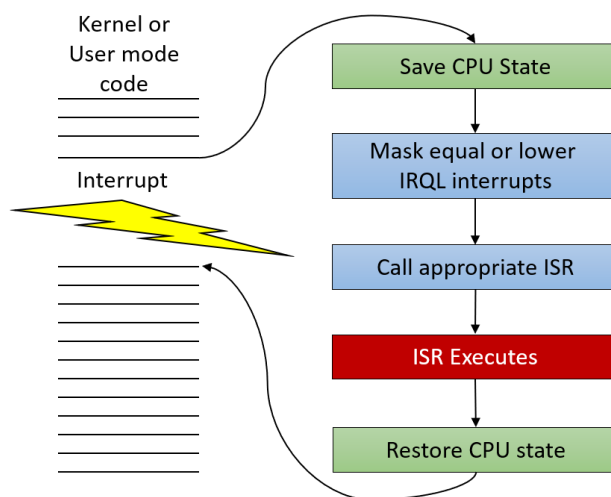


Figure 6-1: Basic interrupt dispatching

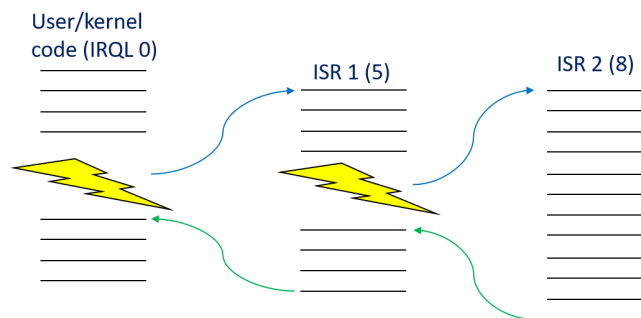


Figure 6-2: Nested interrupts

An important fact for the depicted scenarios in figures 6-1 and 6-2 is that execution of all ISRs is done by the same thread - which got interrupted in the first place. Windows does not have a special thread to handle interrupts; they are handled by whatever thread was running at that time on the interrupted processor. As we'll soon discover, context switching is not possible when the IRQL of the processor is 2 or higher, so there is no way another thread can sneak in while these ISRs execute.

The interrupted thread does not get its quantum reduced because of these “interruptions”. It’s not its fault, so to speak.

When user-mode code is executing, the IRQL is always zero. This is one reason why the term IRQL is not mentioned in any user-mode documentation - it’s always zero and cannot be changed. Most kernel-mode code runs with IRQL zero as well. It’s possible, however, in kernel mode, to raise the IRQL on the current processor.

The important IRQLs are described below:

- `PASSIVE_LEVEL` in WDK (0) - this is the “normal” IRQL for a CPU. User-mode code always runs at this level. Thread scheduling working normally, as described in chapter 1.
- `APC_LEVEL` (1) - used for special kernel APCs (*Asynchronous Procedure Calls* will be discussed later in this chapter). Thread scheduling works normally.
- `DISPATCH_LEVEL` (2) - this is where things change radically. The scheduler cannot wake up on this CPU. Paged memory access is not allowed - such access causes a system crash. Since the scheduler cannot interfere, waiting on kernel objects is not allowed (causes a system crash if used).
- Device IRQL - a range of levels used for hardware interrupts (3 to 11 on x64/ARM/ARM64, 3 to 26 on x86). All rules from IRQL 2 apply here as well.
- Highest level (`HIGH_LEVEL`) - this is the highest IRQL, masking all interrupts. Used by some APIs dealing with linked list manipulation. The actual values are 15 (x64/ARM/ARM64) and 31 (x86).

When a processor’s IRQL is raised to 2 or higher (for whatever reason), certain restrictions apply on the executing code:

- Accessing memory not in physical memory is fatal and causes a system crash. This means accessing data from non-paged pool is always safe, whereas accessing data from paged pool or from user-supplied buffers is not safe and should be avoided.
- Waiting on any kernel object (e.g. mutex or event) causes a system crash, unless the wait timeout is zero, which is still allowed. (we’ll discuss dispatcher object and waiting later in this chapter in the *Thread Synchronization** section.)

These restrictions are due to the fact that the scheduler “runs” at IRQL 2; so if a processor’s IRQL is already 2 or higher, the scheduler cannot wake up on that processor, so context switches (replacing the running thread with another on this CPU) cannot occur. Only higher level interrupts can temporarily divert code into an associated ISR, but it’s still the same thread - no context switch can occur; the thread’s context is saved, the ISR executes and the thread’s state resumes.



The current IRQL of a processor can be viewed while debugging with the `!irql` command. An optional CPU number can be specified, which shows the IRQL of that CPU.

You can view the registered interrupts on a system using the `!idt` debugger command.

Raising and Lowering IRQL

As previously discussed, in user mode the concept of IRQL is not mentioned and there is no way to change it. In kernel mode, the IRQL can be raised with the `KeRaiseIrql` function and lowered back with `KeLowerIrql`. Here is a code snippet that raises the IRQL to `DISPATCH_LEVEL` (2), and then lowers it back after executing some instructions at this IRQL.

```
// assuming current IRQL <= DISPATCH_LEVEL
KIRQL oldIrql;      // typedefed as UCHAR
KeRaiseIrql(DISPATCH_LEVEL, &oldIrql);
```

```
NT_ASSERT(KeGetCurrentIrql() == DISPATCH_LEVEL);

// do work at IRQL DISPATCH_LEVEL

KeLowerIrql(oldIrql);
```

i If you raise IRQL, make sure you lower it in the same function. It's too dangerous to return from a function with a higher IRQL than it was entered. Also, make sure KeRaiseIrql actually raises the IRQL and KeLowerIrql actually lowers it; otherwise, a system crash will follow.

Thread Priorities vs. IRQLs

IRQL is an attribute of a processor. Priority is an attribute of a thread. Thread priorities only have meaning at IRQL < 2. Once an executing thread raised IRQL to 2 or higher, its priority does not mean anything anymore - it has theoretically an infinite quantum - it will continue execution until it lowers the IRQL to below 2.

Naturally, spending a lot of time at IRQL >= 2 is not a good thing; user mode code is not running for sure. This is just one reason there are severe restrictions on what executing code can do at these levels.

Task Manager shows the amount of CPU time spent in IRQL 2 or higher using a pseudo-process called *System Interrupts*; *Process Explorer* calls it *Interrupts*. Figure 6-3 shows a screenshot from *Task Manager* and figure 6-4 shows the same information in *Process Explorer*.

Processes Performance App history Startup Users Details Services											
Name	PID	Status	User name	Sec...	CPU	Memory (a...	Commit size	Base priority	H...	Th...	Description
System Interrupts	-	Running	SYSTEM	0	01	0 K	0 K	N/A	-	-	Deferred procedure calls and interrupt service routines
System Idle Process	0	Running	SYSTEM	0	86	8 K	60 K	N/A	-	12	Percentage of time the processor is idle
System	4	Running	SYSTEM	0	00	20 K	204 K	N/A	8...	382	NT Kernel & System
smss.exe	80	Running	SYSTEM	0	00	80,372 K	164 K	N/A	-	-	NT Kernel & System

Figure 6-3: IRQL 2+ CPU time in Task Manager

Process	PID	CPU	Private Bytes	Working Set	Description	User Name
System	4	0.06	60 K	6 K	Hardware Interrupts and DPCs	NT AUTHORITY\SYSTEM
System Idle Process	0	82.38	60 K	8 K		NT AUTHORITY\SYSTEM
System	4	0.88	204 K	3,932 K		NT AUTHORITY\SYSTEM
Secure System	80	Suspended	164 K	80,372 K		NT AUTHORITY\SYSTEM

Figure 6-4: IRQL 2+ CPU time in Process Explorer

Deferred Procedure Calls

Figure 6-5 shows a typical sequence of events when a client invokes some I/O operation. In this figure, a user mode thread opens a handle to a file, and issues a read operation using the ReadFile function. Since the thread can make an asynchronous call, it regains control almost immediately and can do other work. The driver receiving this request, calls the file system driver (e.g. NTFS), which may call other drivers below it, until the request reaches the disk driver, which initiates the operation on the actual disk hardware. At that point, no code needs to execute, since the hardware “does its thing”.

When the hardware is done with the read operation, it issues an interrupt. This causes the Interrupt Service Routine associated with the interrupt to execute at Device IRQL (note that the thread handling the request is arbitrary, since the interrupt arrives asynchronously). A typical ISR accesses the device's hardware to get the result of the operation. Its final act should be to complete the initial request.

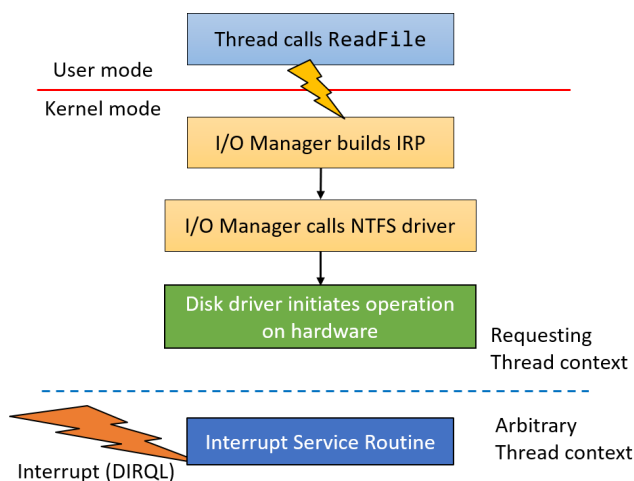


Figure 6-5: Typical I/O request processing (part 1)

As we've seen in chapter 4, completing a request is done by calling `IoCompleteRequest`. The problem is that the documentation states this function can only be called at `IRQL <= DISPATCH_LEVEL (2)`. This means the ISR cannot call `IoCompleteRequest` or it will crash the system. So what is the ISR to do?



You may wonder why there is such a restriction. One of the reasons has to do with the work done by `IoCompleteRequest`. We'll discuss this in more detail in the next chapter, but the bottom line is that this function is relatively expensive. If the call would have been allowed, that would mean the ISR will take substantially longer to execute, and since it executes in a high IRQL, it will mask off other interrupts for a longer period of time.

The mechanism that allows the ISR to call `IoCompleteRequest` (and other functions with similar limitations) as soon as possible is using a *Deferred Procedure Call (DPC)*. A DPC is an object encapsulating a function that is to be called at `IRQL DISPATCH_LEVEL`. At this IRQL, calling `IoCompleteRequest` is permitted.



You may wonder why does the ISR not simply lower the current IRQL to `DISPATCH_LEVEL`, call `IoCompleteRequest`, and then raise the IRQL back to its original value. This can cause a deadlock. We'll discuss the reason for that later in this chapter in the section *Spin Locks*.

The driver which registered the ISR prepares a DPC in advance, by allocating a `KDPC` structure from non-paged pool and initializing it with a callback function using `KeInitializeDpc`. Then,

when the ISR is called, just before exiting the function, the ISR requests the DPC to execute as soon as possible by queuing it using `KeInsertQueueDpc`. When the DPC function executes, it calls `IoCompleteRequest`. So the DPC serves as a compromise - it's running at `IRQL_DISPATCH_LEVEL`, meaning no scheduling can occur, no paged memory access, etc. but it's not high enough to prevent hardware interrupts from coming in and being served on the same processor.

Every processor on the system has its own queue of DPCs. By default, `KeInsertQueueDpc` queues the DPC to the current processor's DPC queue. When the ISR returns, before the `IRQL` can drop back to zero, a check is made to see whether DPCs exist in the processor's queue. If there are, the processor drops to `IRQL_DISPATCH_LEVEL (2)` and then processes the DPCs in the queue in a First In First Out (FIFO) manner, calling the respective functions, until the queue is empty. Only then can the processor's `IRQL` drop to zero, and resume executing the original code that was disturbed at the time the interrupt arrived.



DPCs can be customized in some ways. Check out the docs for the functions `KeSetImportanceDpc` and `KeSetTargetProcessorDpc`.

Figure 6-6 augments figure 6-5 with the DPC routine execution.

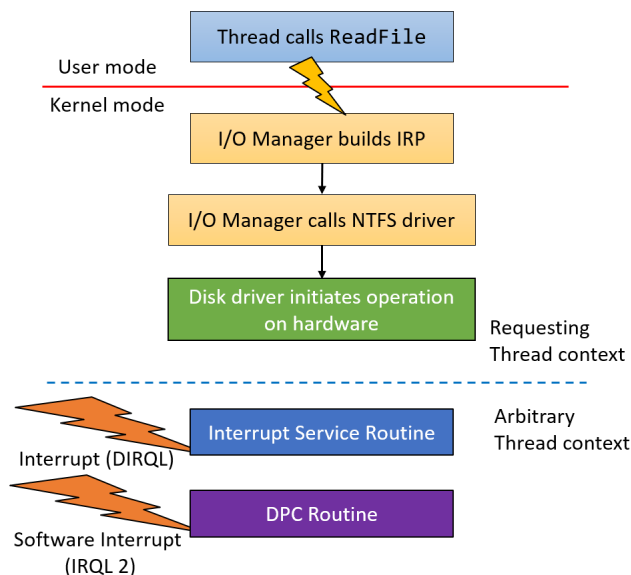


Figure 6-6: Typical I/O request processing (part 2)

Using DPC with a Timer

DPCs were originally created for use by ISRs. However, there are other mechanisms in the kernel that utilize DPCs.

One such use is with a kernel timer. A kernel timer, represented by the `KTIMER` structure allows setting up a timer to expire some time in the future, based on a relative interval or absolute time. This timer is a dispatcher object and so can be waited upon with `KeWaitForSingleObject` (discussed later in this chapter in the section “Synchronization”). Although waiting is possible, it’s inconvenient for a timer. A simpler approach is to call some callback when the timer expires. This is exactly what the kernel timer provides using a DPC as its callback.

The following code snippet shows how to configure a timer and associate it with a DPC. When the timer expires, the DPC is inserted into a CPU’s DPC queue and so executes as soon as possible. Using a DPC is more powerful than a zero IRQL based callback, since it is guaranteed to execute before any user mode code (and most kernel mode code).

```
KTIMER Timer;
KDPC TimerDpc;

void InitializeAndStartTimer(ULONG msec) {
    KeInitializeTimer(&Timer);
    KeInitializeDpc(&TimerDpc,
        OnTimerExpired,    // callback function
        nullptr);         // passed to callback as "context"

    // relative interval is in 100nsec units (and must be negative)
    // convert to msec by multiplying by 10000

    LARGE_INTEGER interval;
    interval.QuadPart = -10000LL * msec;
    KeSetTimer(&Timer, interval, &TimerDpc);
}

void OnTimerExpired(KDPC* Dpc, PVOID context, PVOID, PVOID) {
    UNREFERENCED_PARAMETER(Dpc);
    UNREFERENCED_PARAMETER(context);

    NT_ASSERT(KeGetCurrentIrql() == DISPATCH_LEVEL);

    // handle timer expiration
}
```

Asynchronous Procedure Calls

We’ve seen in the previous section that DPCs are objects encapsulating a function to be called at IRQL `DISPATCH_LEVEL`. The calling thread does not matter, as far as DPCs are concerned.

Asynchronous Procedure Calls (APCs) are also data structures that encapsulate a function to be called. But contrary to a DPC, an APC is targeted towards a particular thread, so only that thread can execute the function. This means each thread has an APC queue associated with it.

There are three types of APCs:

- User mode APCs - these execute in user mode at IRQL `PASSIVE_LEVEL` only when the thread goes into alertable state. This is typically accomplished by calling an API such as `SleepEx`, `WaitForSingleObjectEx`, `WaitForMultipleObjectsEx` and similar APIs. The last argument to these functions can be set to `TRUE` to put the thread in alertable state. In this state it looks at its APC queue, and if not empty - the APCs now execute until the queue is empty.
- Normal kernel-mode APCs - these execute in kernel mode at IRQL `PASSIVE_LEVEL` and preempt user-mode code (and user-mode APCs).

- Special kernel APCs - these execute in kernel mode at IRQL APC_LEVEL (1) and preempt user-mode code, normal kernel APCs, and user-mode APCs. These APCs are used by the I/O manager to complete I/O operations as will be discussed in the next chapter.

The APC API is undocumented in kernel mode (but has been reversed engineered enough to allow usage if desired).



User-mode can use (user mode) APCs by calling certain APIs. For example, calling `ReadFileEx` or `WriteFileEx` start an asynchronous I/O operation. When the operation completes, a user-mode APC is attached to the calling thread. This APC will execute when the thread enters an *alertable* state as described earlier. Another useful function in user mode to explicitly generate an APC is `QueueUserAPC`. Check out the Windows API documentation for more information.

Critical Regions and Guarded Regions

A *Critical Region* prevents user mode and normal kernel APCs from executing (special kernel APCs can still execute). A thread enters a critical region with `KeEnterCriticalRegion` and leaves it with `KeLeaveCriticalRegion`. Some functions in the kernel require being inside a critical region, especially when working with executive resources (see the section “Executive Resources” later in this chapter).

A *Guarded Region* prevents all APCs from executing. Call `KeEnterGuardedRegion` to enter a guarded region and `KeLeaveGuardedRegion` to leave it. Recursive calls to `KeEnterGuardedRegion` must be matched with the same number of calls to `KeLeaveGuardedRegion`.



Raising the IRQL to APC_LEVEL disables delivery of all APCs.



Write RAIL wrappers for entering/leaving critical and guarded regions.

Structured Exception Handling

An exception is an event that occurs because of a certain instruction that did something that caused the processor to raise an error. Exceptions are in some ways similar to interrupts, the main difference being that an exception is synchronous and technically reproducible under the same conditions, whereas an interrupt is asynchronous and can arrive at any time. Examples of exceptions include division by zero, breakpoint, page fault, stack overflow and invalid instruction.

If an exception occurs, the kernel catches this and allows code to handle the exception, if possible. This mechanism is called *Structured Exception Handling* (SEH) and is available for user-mode code as well as kernel-mode code.

The kernel exception handlers are called based on the *Interrupt Dispatch Table* (IDT), the same one holding mapping between interrupt vectors and ISRs. Using a kernel debugger, the `!idt` command shows all these mappings. The low numbered interrupt vectors are in fact exception handlers. Here's a sample output from this command:

```
lkd> !idt

Dumping IDT: ffffff8011d941000

00: ffffff8011dd6c100 nt!KiDivideErrorFaultShadow
01: ffffff8011dd6c180 nt!KiDebugTrap0rFaultShadow      Stack = 0xFFFFF8011D9459D0
02: ffffff8011dd6c200 nt!KiNmiInterruptShadow          Stack = 0xFFFFF8011D9457D0
03: ffffff8011dd6c280 nt!KiBreakpointTrapShadow
04: ffffff8011dd6c300 nt!KiOverflowTrapShadow
05: ffffff8011dd6c380 nt!KiBoundFaultShadow
06: ffffff8011dd6c400 nt!KiInvalidOpcodeFaultShadow
07: ffffff8011dd6c480 nt!KiNpxNotAvailableFaultShadow
08: ffffff8011dd6c500 nt!KiDoubleFaultAbortShadow      Stack = 0xFFFFF8011D9453D0
09: ffffff8011dd6c580 nt!KiNpxSegmentOverrunAbortShadow
0a: ffffff8011dd6c600 nt!KiInvalidTssFaultShadow
0b: ffffff8011dd6c680 nt!KiSegmentNotPresentFaultShadow
0c: ffffff8011dd6c700 nt!KiStackFaultShadow
0d: ffffff8011dd6c780 nt!KiGeneralProtectionFaultShadow
0e: ffffff8011dd6c800 nt!KiPageFaultShadow
10: ffffff8011dd6c880 nt!KiFloatingErrorFaultShadow
11: ffffff8011dd6c900 nt!KiAlignmentFaultShadow

(truncated)
```

Note the function names - most are very descriptive. These entries are connected to Intel/AMD (in this example) faults. Some common examples of exceptions include:

- Division by zero (0)
- Breakpoint (3) - the kernel handles this transparently, passing control to an attached debugger (if any).
- Invalid opcode (6) - this fault is raised by the CPU if it encounters an unknown instruction.
- Page fault (14) - this fault is raised by the CPU if the page table entry used for translating virtual to physical addresses has the Valid bit set to zero, indicating (as far as the CPU is concerned) that the page is not resident in physical memory.

Some other exceptions are raised by the kernel as a result of a previous CPU fault. For example, if a page fault is raised, the Memory Manager's page fault handler will try to locate the page that is not resident in RAM. If the page happens not to exist at all, the Memory Manager will raise an Access Violation exception.

Once an exception is raised, the kernel searches the function where the exception occurred for a handler (except for some exceptions which it handles transparently, such as Breakpoint (3)). If not found, it will search up the call stack, until such handler is found. If the call stack is exhausted, the system will crash.

How can a driver handle these types of exceptions? Microsoft added four keywords to the C language to allow developers to handle such exceptions, as well as have code execute no matter what. Table 6-1 shows the added keywords with a brief description.

Table 6-1: Keywords for working with SEH

Keyword	Description
<code>__try</code>	Starts a block of code where exceptions may occur.
<code>__except</code>	Indicates if an exception is handled, and provides the handling code if it is.
<code>__finally</code>	Unrelated to exceptions directly. Provides code that is guaranteed to execute no matter what - whether the <code>__try</code> block is exited normally, with a return statement, or because of an exception.
<code>__leave</code>	Provides an optimized mechanism to jump to the <code>__finally</code> block from somewhere within a <code>__try</code> block.

The valid combination of keywords is `__try/__except` and `__try/__finally`. However, these can be combined by using nesting to any level.



These same keywords work in user mode as well, in much the same way.

Using `__try/__except`

In chapter 4, we implemented a driver that accesses a user-mode buffer to get data needed for the driver's operation. We used a direct pointer to the user's buffer. However, this is not guaranteed to be safe. For example, the user-mode code (say from another thread) could free the buffer, just before the driver accesses it. In such a case, the driver would cause a system crash, essentially because of a user's error (or malicious intent). Since user data should never be trusted, such access should be wrapped in a `__try/__except` block to make sure a bad buffer does not crash the driver.

Here is the important part of a revised `IRP_MJ_WRITE` handler using an exception handler:

```
do {
    if (irpSp->Parameters.Write.Length < sizeof(ThreadData)) {
        status = STATUS_BUFFER_TOO_SMALL;
        break;
    }
    auto data = (ThreadData*)Irp->UserBuffer;
    if (data == nullptr) {
        status = STATUS_INVALID_PARAMETER;
        break;
    }
    __try {
        if (data->Priority < 1 || data->Priority > 31) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }
        PETHREAD Thread;
        status = PsLookupThreadByThreadId(
            ULongToHandle(data->ThreadId), &Thread);
        if (!NT_SUCCESS(status))
            break;
        KeSetPriorityThread((PKTHREAD)Thread, data->Priority);
        ObDereferenceObject(Thread);
        KdPrint(("Thread Priority change for %d to %d succeeded!\n",
            data->ThreadId, data->Priority));
        break;
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        // probably something wrong with the buffer
    }
}
```

```
        status = STATUS_ACCESS_VIOLATION;
    }
} while(false);
```

Placing `EXCEPTION_EXECUTE_HANDLER` in `__except` says that any exception is to be handled. We can be more selective by calling `GetExceptionCode` and looking at the actual exception. If we don't expect this, we can tell the kernel to continue looking for handlers up the call stack:

```
__except (GetExceptionCode() == STATUS_ACCESS_VIOLATION
        ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
    // handle exception
}
```

Does all this mean that the driver can catch any and all exceptions? If so, the driver will never cause a system crash. Fortunately (or unfortunately, depending on your perspective), this is not the case. Access violation, for example, is something that can only be caught if the violated address is in user space. If it's in kernel space, it cannot be caught and still cause a system crash. This makes sense, since something bad has happened and the kernel will not let the driver get away with it. User mode addresses, on the other hand, are not at the control of the driver, so such exceptions can be caught and handled.

The SEH mechanism can also be used by drivers (and user-mode code) to raise custom exceptions. The kernel provides the generic function `ExRaiseStatus` to raise any exception and some specific functions like `ExRaiseAccessViolation`:

```
void ExRaiseStatus(NTSTATUS Status);
```

A driver can also crash the system explicitly if it concludes that something really bad is going on, such as data being corrupted from underneath the driver. The kernel provides the `KeBugCheckEx` for this purpose:

```
VOID KeBugCheckEx(
    _In_ ULONG BugCheckCode,
    _In_ ULONG_PTR BugCheckParameter1,
    _In_ ULONG_PTR BugCheckParameter2,
    _In_ ULONG_PTR BugCheckParameter3,
    _In_ ULONG_PTR BugCheckParameter4);
```

`KeBugCheckEx` is the normal kernel function that generates a crash. `BugCheckCode` is the crash code to be reported, and the other 4 numbers can provide more details about the crash. If the bugcheck code is one of those documented by Microsoft, the meaning of the other 4 numbers must be provided as documented. (See the next section *System Crash* for more details).

Using `__try/__finally`

Using a block of `__try` and `__finally` is not directly related to exceptions. This is about making sure some piece of code executes no matter what - whether the code exits cleanly or mid-way because of an exception. This is similar in concept to the `finally` keyword popular in some high level languages (e.g. Java, C#). Here is a simple example to show the problem:

```
void foo() {
    void* p = ExAllocatePoolWithTag(PagedPool, 1024, DRIVER_TAG);
```

```
    if(p == nullptr)
        return;

    // do something with p

    ExFreePool(p);
}
```

The above code seems harmless enough. However, there are several issues with it:

- If an exception is thrown between the allocation and the release, a handler in the caller will be searched, but the memory will not be freed.
- If a return statement is used in some conditional between the allocation and release, the buffer will not be freed. This requires the code to be careful to make sure all exit points from the function pass through the code freeing the buffer.

The second bullet can be implemented with careful coding, but is a burden best avoided. The first bullet cannot be handled with standard coding techniques. This is where `__try/__finally` come in. Using this combination, we can make sure the buffer is freed no matter what happens in the `__try` block:

```
void foo() {
    void* p = ExAllocatePoolWithTag(PagedPool, 1024, DRIVER_TAG);
    if(p == nullptr)
        return;
    __try {
        // do something with p
    }
    __finally {
        // called no matter what
        ExFreePool(p);
    }
}
```

With the above code in place, even if return statements appear within the `__try` body, the `__finally` code will be called before actually returning from the function. If some exception occurs, the `__finally` block runs first before the kernel searches up the call stack for possible handlers.

`__try/__finally` is useful not just with memory allocations, but also with other resources, where some acquisition and release need to take place. One common example is when synchronizing threads accessing some shared data. Here is an example of acquiring and releasing a fast mutex (fast mutex and other synchronization primitives are described later in this chapter):

```
FAST_MUTEX MyMutex;

void foo() {
    ExAcquireFastMutex(&MyMutex);
    __try {
        // do work while the fast mutex is held
    }
    __finally {
        ExReleaseFastMutex(&MyMutex);
    }
}
```

Using C++ RAII Instead of `__try / __finally`

Although the preceding examples with `__try/__finally` work, they are not terribly convenient. Using C++ we can build RAII wrappers that do the right thing without the need to use `__try/__finally`. C++ does not have a `finally` keyword like C# or Java, but it doesn't need one - it has destructors.

Here is a very simple, bare minimum, example that manages a buffer allocation with a RAII class:

```
template<typename T = void>
struct kunique_ptr {
    explicit kunique_ptr(T* p = nullptr) : _p(p) {}
    ~kunique_ptr() {
        if (_p)
            ExFreePool(_p);
    }

    T* operator->() const {
        return _p;
    }

    T& operator*() const {
        return *_p;
    }

private:
    T* _p;
};
```

The class uses templates to allow working easily with any type of data. An example usage follows:

```
struct MyData {
    ULONG Data1;
    HANDLE Data2;
};

void foo() {
    // take charge of the allocation
    kunique_ptr<MyData> data((MyData*)ExAllocatePool(PagedPool, sizeof(MyData)))\
};
    // use the pointer
    data->Data1 = 10;
    // when the object goes out of scope, the destructor frees the buffer
}
```

If you don't normally use C++ as your primary programming language, you may find the above code confusing. You can continue working with `__try/__finally`, but I recommend getting acquainted with this type of code. In any case, even if you struggle with the implementation of `kunique_ptr` above, you can still use it without needing to understand every little detail.

The `kunique_ptr` type presented above is a bare minimum. You should also remove the copy constructor and copy assignment, and allow move copy and assignment (C++ 11 and later, for ownership transfer). Here is a more complete implementation:

```
template<typename T = void>
struct kunique_ptr {
    explicit kunique_ptr(T* p = nullptr) : _p(p) {}

    // remove copy ctor and copy = (single owner)
    kunique_ptr(const kunique_ptr&) = delete;
```

```

kunique_ptr& operator=(const kunique_ptr&) = delete;

// allow ownership transfer
kunique_ptr(kunique_ptr&& other) : _p(other._p) {
    other._p = nullptr;
}

kunique_ptr& operator=(kunique_ptr&& other) {
    if (&other != this) {
        Release();
        _p = other._p;
        other._p = nullptr;
    }
    return *this;
}

~kunique_ptr() {
    Release();
}

operator bool() const {
    return _p != nullptr;
}

T* operator->() const {
    return _p;
}

T& operator*() const {
    return *_p;
}

void Release() {
    if (_p)
        ExFreePool(_p);
}

private:
    T* _p;
};

```

We'll build other RAII wrappers for synchronization primitives later in this chapter.



Using C++ RAII wrappers has one missing piece - if an exception occurs, the destructor will **not** be called, so a leak of some sort occurs. The reason this does not work (as it does in user-mode), is the lack of a C++ runtime and the current inability of the compiler to set up elaborate code with `__try/__finally` to mimic this effect. Even so, it's still very useful, as in many cases exceptions are not expected, and even if they are, no handler exists in the driver for that and the system should probably crash anyway.

System Crash

As we already know, if an unhandled exception occurs in kernel mode, the system crashes, typically with the “Blue Screen of Death” (BSOD) showing its face (on Windows 8+, that's literally a face - *saddy* or *frowny* - the inverse of *smiley*). In this section, we'll discuss what happens when the system crashes and how to deal with it.

The system crash has many names, all meaning the same thing - “Blue screen of Death”, “System failure”, “Bugcheck”, “Stop error”. The BSOD is not some punishment, as may seem at first, but a protection mechanism. If kernel code, which is supposed to be trusted, did something bad,

stopping everything is probably the safest approach, as perhaps letting the code continue roaming around may result in an unbootable system if some important files or Registry data is corrupted.

Recent versions of Windows 10 have some alternate colors for when the system crashes. Green is used for insider preview builds, and I actually encountered a pink as well (power-related errors).

If the crashed system is connected to a kernel debugger, the debugger will break. This allows examining the state of the system before other actions take place.

The system can be configured to perform some operations if the system crashes. This can be done with the *System Properties* UI on the *Advanced* tab. Clicking *Settings...* at the *Startup and Recovery* section brings the *Startup and Recovery* dialog where the *System Failure* section shows the available options. Figure 6-7 shows these two dialogs.

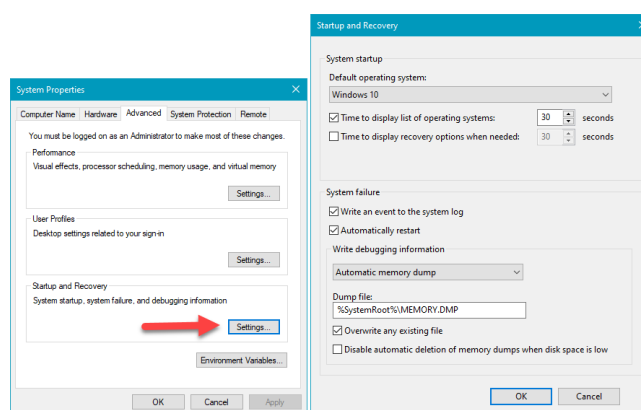


Figure 6-7: Startup and recovery settings

If the system crashes, an event entry can be written to the event log. It's checked by default, and there is no good reason to change it. The system is configured to automatically restart; this has been the default since Windows 2000.

The most important setting is the generation of a dump file. The dump file captures the system state at the time of the crash, so it can later be analyzed by loading the dump file into the debugger. The type of the dump file is important since it determines what information will be present in the dump. The dump is not written to the target file at crash time, but instead written to the first page file.

Only when the system restarts, the kernel notices there is dump information in the page file, and it copies the data to the target file. The reason has to do with the fact that at system crash time it may be too dangerous to write something to a new file (or overwrite an existing file); the I/O system may not be stable enough. The best bet is to write the data to a page file, which is already open anyway. The downside is that the page file must be large enough to contain the dump, otherwise the dump file will not be generated.



The dump file contains physical memory only.

The dump type determines what data would be written and hints at the page file size that may be required. Here are the options:

- *Small memory dump* (256 KB on Windows 8 and later, 64 KB on older systems) - a very minimal dump, containing basic system information and information on the thread that caused the crash. Usually this is too little to determine what happened in all but the most trivial cases. The upside is that the file is small, so it can be easily moved.
- *Kernel memory dump* - this is the default on Windows 7 and earlier versions. This setting captures all kernel memory but no user memory. This is usually good enough, since a system crash can only be caused by kernel code misbehaving. It's extremely unlikely that user-mode had anything to do with it.
- *Complete memory dump* - this provides a dump of all physical memory, user memory and kernel memory. This is the most complete information available. The downside is the size of the dump, which could be gigantic depending on the size of RAM (the total size of the final file). The obvious optimization is not to include unused pages, but *Complete Memory Dump* does not do that.
- *Automatic memory dump* (Windows 8+) - this is the default on Windows 8 and later. This is the same as kernel memory dump, but the kernel resizes the page file on boot to a size that guarantees with high probability that the page file size would be large enough to contain a kernel dump. This is only done if the page file size is specified as "System managed" (the default).
- *Active memory dump* (Windows 10+) - this is similar to a complete memory dump, with two exceptions. First, unused pages are not written. Second, if the crashed system is hosting guest virtual machines, the memory they were using at the time is not captured (as it's unlikely these have anything to do with the host crashing). These optimizations help in reducing the dump file size.

Crash Dump Information

Once you have a crash dump in hand, you can open it in *WinDbg* by selecting *File/Open Dump File* and navigating to the file. The debugger will spew some basic information similar to the following:

```
Microsoft (R) Windows Debugger Version 10.0.18317.1001 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Loading Dump File [C:\Windows\MEMORY.DMP]
Kernel Bitmap Dump File: Kernel address space is available, User address space \
may not be available.

***** Path validation summary *****
Response           Time (ms)          Location
Deferred
soft.com/download/symbols
Symbol search path is: SRV*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 18362 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
```

```

Built by: 18362.1.amd64fre.19h1_release.190318-1202
Machine Name:
Kernel base = 0xffffffff803`70abc000 PsLoadedModuleList = 0xffffffff803`70eff2d0
Debug session time: Wed Apr 24 15:36:55.613 2019 (UTC + 3:00)
System Uptime: 0 days 0:05:38.923
Loading Kernel Symbols
.....Page 2001b5efc too large to be in the dump \
file.
Page 20001ebfb too large to be in the dump file.
.....
Loading User Symbols
PEB is paged out (Peb.Ldr = 00000054`34256018). Type ".hh dbgerr001" for detail\
ls
Loading unloaded module list
.....
For analysis of this file, run !analyze -v
nt!KeBugCheckEx:
fffff803`70c78810 48894c2408      mov     qword ptr [rsp+8],rcx ss:fffff988`53b\
0f6b0=0000000000000000a

```

The debugger suggests running `!analyze -v` and it's the most common thing to do at the start of dump analysis. Notice the call stack is at `KeBugCheckEx`, which is the function generating the bugcheck.

The default logic behind `!analyze -v` performs basic analysis on the thread that caused the crash and shows a few pieces of information related to the crash dump code:

```

2: kd> !analyze -v
*****
*
*                               Bugcheck Analysis                               *
*
*****

DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high. This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: fffffd907b0dc7660, memory referenced
Arg2: 0000000000000002, IRQL
Arg3: 0000000000000000, value 0 = read operation, 1 = write operation
Arg4: fffff80375261530, address which referenced memory

Debugging Details:
-----

(truncated)

DUMP_TYPE: 1

BUGCHECK_P1: fffffd907b0dc7660

BUGCHECK_P2: 2

BUGCHECK_P3: 0

BUGCHECK_P4: fffff80375261530

READ_ADDRESS: Unable to get offset of nt!_MI_VISIBLE_STATE.SpecialPool
Unable to get value of nt!_MI_VISIBLE_STATE.SessionSpecialPool
fffffd907b0dc7660 Paged pool

CURRENT_IRQL: 2

FAULTING_IP:
myfault+1530
fffff803`75261530 8b03      mov     eax,dword ptr [rbx]

```

(truncated)

ANALYSIS_VERSION: 10.0.18317.1001 amd64fre

TRAP_FRAME: ffffff98853b0f7f0 -- (.trap 0xfffff98853b0f7f0)

NOTE: The trap frame does **not** contain **all** registers.

Some register **values** may be zeroed **or** incorrect.

rax=0000000000000000 rbx=0000000000000000 rcx=ffffd90797400340

rdx=00000000000000880 rsi=0000000000000000 rdi=0000000000000000

rip=fffff80375261530 rsp=fffff98853b0f980 rbp=0000000000000002

r8=ffffd9079c5cec10 r9=0000000000000000 r10=ffffd907974002c0

r11=ffffd907b0dc1650 r12=0000000000000000 r13=0000000000000000

r14=0000000000000000 r15=0000000000000000

iopl=0 nv up ei ng nz na po nc

myfault+0x1530:

fffff803`75261530 8b03 mov eax,dword ptr [rbx] ds:00000000`00000\

000=????????

Resetting default scope

LAST_CONTROL_TRANSFER: from ffffff80370c8a469 to ffffff80370c78810

STACK_TEXT:

fffff988`53b0f6a8 ffffff803`70c8a469 : 00000000`0000000a fffffd907`b0dc7660 00000\

000`00000002 00000000`00000000 : nt!KeBugCheckEx

fffff988`53b0f6b0 ffffff803`70c867a5 : ffff8788`e4604080 ffffff4c`c66c7010 00000\

000`00000003 00000000`00000880 : nt!KiBugCheckDispatch+0x69

fffff988`53b0f7f0 ffffff803`75261530 : ffffff4c`c66c7000 00000000`00000000 fffff\

988`53b0f9e0 00000000`00000000 : nt!KiPageFault+0x465

fffff988`53b0f980 ffffff803`75261e2d : ffffff988`00000000 00000000`00000000 fffff\

788`ec7cf520 00000000`00000000 : myfault+0x1530

fffff988`53b0f9b0 ffffff803`75261f88 : ffffff4c`c66c7010 00000000`000000f0 00000\

000`00000001 ffffff30`21ea80aa : myfault+0x1e2d

fffff988`53b0fb00 ffffff803`70ae3da9 : ffff8788`e6d8e400 00000000`00000001 00000\

000`83360018 00000000`00000001 : myfault+0x1f88

fffff988`53b0fb40 ffffff803`710d1dd5 : ffffff988`53b0fec0 ffff8788`e6d8e400 00000\

000`00000001 ffff8788`ecdb6690 : nt!IofCallDriver+0x59

fffff988`53b0fb80 ffffff803`710d172a : ffff8788`00000000 00000000`83360018 00000\

000`00000000 ffffff988`53b0fec0 : nt!IopSynchronousServiceTail+0x1a5

fffff988`53b0fc20 ffffff803`710d1146 : 00000054`344feb28 00000000`00000000 00000\

000`00000000 00000000`00000000 : nt!IopXxxControlFile+0x5ca

fffff988`53b0fd60 ffffff803`70c89e95 : ffff8788`e4604080 ffffff988`53b0fec0 00000\

054`344feb28 ffffff988`569fd630 : nt!NtDeviceIoControlFile+0x56

fffff988`53b0fdd0 00007ff8`ba39c147 : 00000000`00000000 00000000`00000000 00000\

000`00000000 00000000`00000000 : nt!KiSystemServiceCopyEnd+0x25

00000054`344feb48 00000000`00000000 : 00000000`00000000 00000000`00000000 00000\

000`00000000 00000000`00000000 : 0x00007ff8`ba39c147

(truncated)

FOLLOWUP_IP:

myfault+1530

fffff803`75261530 8b03 mov eax,dword ptr [rbx]

FAULT_INSTR_CODE: 8d48038b

SYMBOL_STACK_INDEX: 3

SYMBOL_NAME: myfault+1530

FOLLOWUP_NAME: MachineOwner

MODULE_NAME: myfault

IMAGE_NAME: myfault.sys

(truncated)

Every crash dump code can have up to 4 numbers that provide more information about the crash. In this case, we can see the code is DRIVER_IRQL_NOT_LESS_OR_EQUAL (0xd1) and the next four

numbers named *Arg1* through *Arg4* mean (in order): memory referenced, the IRQL at the time of the call, read vs. write operation and the accessing address.

The command clearly recognizes *myfault.sys* as the faulting module (driver). That's because this is an easy crash - the culprit is on the call stack as can be seen in the *STACK TEXT* section above (you can also simply use the *k* command to see it again).



The `!analyze -v` command is extensible and it's possible to add more analysis to that command using an extension DLL. You may be able to find such extensions on the web. Consult the debugger API documentation for more information on how to add your own analysis code to this command.

More complex crash dumps may show calls from the kernel only on the call stack of the offending thread. Before you conclude that you found a bug in the Windows kernel, consider this more likely scenario: A driver did something that was not fatal in itself, such as experience a buffer overflow - wrote data beyond its allocated buffer, but unfortunately ,the memory following that buffer was allocated by some other driver or the kernel, and so nothing bad happened at that time. Some time later, the kernel accessed that memory and got bad data and caused a system crash. But the faulting driver is nowhere to be found on any call stack; this is much harder to diagnose.



One way to help diagnose such issues is using *Driver Verifier*. We'll look at the basics of Driver Verifier in module 12.

Once you get the crash dump code, it's helpful to look in the debugger documentation at the topic "Bugcheck Code Reference", where common bugcheck codes are explained more fully with typical causes and ideas on what to investigate next.

Analyzing a Dump File

A dump file is a snapshot of a system's memory. Other than that, it's the same as any other kernel debugging session. You just can't set breakpoints, and certainly cannot use any *go* command. All other commands are available as usual. Commands such as `!process`, `!thread`, `!m`, `k` can be used normally. Here are some other commands and tips:

- The prompt indicates the current processor. Switching processors can be done with the command `~ns` where *n* is the CPU index (it looks like switching threads in user mode).
- The `!running` command can be used to list the threads that were running on all processors at the time of the crash. Adding `-t` as an option shows the call stack for each thread. Here is an example with the above crash dump:

```
2: kd> !running -t
```

```
System Processors: (000000000000000f)
Idle Processors:  (0000000000000002)
```

	Prcbs	Current	(pri) Next	(pri) Idle
0	fffff8036ef3f180	fffff8788e91cf080	(8)	fffff80371\
048400				

```

# Child-SP          RetAddr          Call Site
00 00000094`ed6ee8a0 00000000`00000000 0x00007ff8`b74c4b57

2 fffffb000c1944180 ffff8788e4604080 (12) fffffb000c1\
955140 .....

# Child-SP          RetAddr          Call Site
00 fffff988`53b0f6a8 fffff803`70c8a469 nt!KeBugCheckEx
01 fffff988`53b0f6b0 fffff803`70c867a5 nt!KiBugCheckDispatch+0x69
02 fffff988`53b0f7f0 fffff803`75261530 nt!KiPageFault+0x465
03 fffff988`53b0f980 fffff803`75261e2d myfault+0x1530
04 fffff988`53b0f9b0 fffff803`75261f88 myfault+0x1e2d
05 fffff988`53b0fb00 fffff803`70ae3da9 myfault+0x1f88
06 fffff988`53b0fb40 fffff803`710d1dd5 nt!IofCallDriver+0x59
07 fffff988`53b0fb80 fffff803`710d172a nt!IopSynchronousServiceTail+0x1a5
08 fffff988`53b0fc20 fffff803`710d1146 nt!IopXxxControlFile+0x5ca
09 fffff988`53b0fd60 fffff803`70c89e95 nt!NtDeviceIoControlFile+0x56
0a fffff988`53b0fdd0 00007ff8`ba39c147 nt!KiSystemServiceCopyEnd+0x25
0b 00000054`344feb48 00000000`00000000 0x00007ff8`ba39c147

3 fffffb000c1c80180 ffff8788e917e0c0 ( 5) fffffb000c1\
c91140 .....

# Child-SP          RetAddr          Call Site
00 fffff988`5683ec38 fffff803`70ae3da9 Ntfs!NtfsFsdClose
01 fffff988`5683ec40 fffff803`702bb5de nt!IofCallDriver+0x59
02 fffff988`5683ec80 fffff803`702b9f16 FLTMRGR!FltpLegacyProcessingAfterPreCallb\
acksCompleted+0x15e
03 fffff988`5683ed00 fffff803`70ae3da9 FLTMRGR!FltpDispatch+0xb6
04 fffff988`5683ed60 fffff803`710cfe4d nt!IofCallDriver+0x59
05 fffff988`5683eda0 fffff803`710de470 nt!IopDeleteFile+0x12d
06 fffff988`5683ee20 fffff803`70aea9d4 nt!ObpRemoveObjectRoutine+0x80
07 fffff988`5683ee80 fffff803`723391f5 nt!ObfDereferenceObject+0xa4
08 fffff988`5683eec0 fffff803`72218ca7 Ntfs!NtfsDeleteInternalAttributeStream+0\
x111
09 fffff988`5683ef00 fffff803`722ff7cf Ntfs!NtfsDecrementCleanupCounts+0x147
0a fffff988`5683ef40 fffff803`722fe87d Ntfs!NtfsCommonCleanup+0xadf
0b fffff988`5683f390 fffff803`70ae3da9 Ntfs!NtfsFsdCleanup+0x1ad
0c fffff988`5683f6e0 fffff803`702bb5de nt!IofCallDriver+0x59
0d fffff988`5683f720 fffff803`702b9f16 FLTMRGR!FltpLegacyProcessingAfterPreCallb\
acksCompleted+0x15e
0e fffff988`5683f7a0 fffff803`70ae3da9 FLTMRGR!FltpDispatch+0xb6
0f fffff988`5683f800 fffff803`710ccc38 nt!IofCallDriver+0x59
10 fffff988`5683f840 fffff803`710d4bf8 nt!IopCloseFile+0x188
11 fffff988`5683f8d0 fffff803`710d9f3e nt!ObcCloseHandleTableEntry+0x278
12 fffff988`5683fa10 fffff803`70c89e95 nt!NtClose+0xde
13 fffff988`5683fa80 00007ff8`ba39c247 nt!KiSystemServiceCopyEnd+0x25
14 000000b5`aacf9df8 00000000`00000000 0x00007ff8`ba39c247

```

The command gives a pretty good idea of what was going on at the time of the crash.

- The `!stacks` command lists all thread stacks for all threads by default. A more useful variant is a search string that lists only threads where a module or function containing this string appears. This allows locating driver's code throughout the system (because it may not have been running at the time of the crash, but it's on some thread's call stack). Here's an example for the above dump:

```

2: kd> !stacks
Proc.Thread .Thread Ticks ThreadState Blocker
[fffff803710459c0 Idle]
0.000000 fffff80371048400 00000003 RUNNING nt!KiIdleLoop+0x15e
0.000000 fffffb000c17b1140 00000ed9 RUNNING hal!HalProcessorIdle+0xf
0.000000 fffffb000c1955140 0000b6e RUNNING nt!KiIdleLoop+0x15e
0.000000 fffffb000c1c91140 000012b RUNNING nt!KiIdleLoop+0x15e
[fffff8788d6a81300 System]
4.000018 fffff8788d6b8a080 0005483 Blocked nt!PopFxEmergencyWorker+0x3e
4.00001c fffff8788d6bc5140 0000982 Blocked nt!ExpWorkQueueManagerThread+0x\

```

```

127
  4.000020  fffff8788d6bc9140 000085a Blocked    nt!KeRemovePriQueue+0x25c

(truncated)

2: kd> !stacks 0 myfault
Proc.Thread  .Thread  Ticks    ThreadState Blocker
           [fffff803710459c0 Idle]
           [fffff8788d6a81300 System]

(truncated)
           [fffff8788e99070c0 notmyfault64.exe]
af4.00160c  fffff8788e4604080 0000006 RUNNING    nt!KeBugCheckEx

(truncated)

```

The address next to each line is the thread's ETHTHREAD address that can be fed to the `!thread` command.

System Hang

A system crash is the most common type of dump that is typically investigated. However, there is yet another type of dump that you may need to work with: a hung system. A hung system is a non-responsive or near non-responsive system. Things seem to be halted or deadlocked in some way - the system does not crash, so the first issue to deal with is how to get a dump of the system.

A dump file contains some system state, it does not have to be related to a crash or any other bad state. There are tools (including the kernel debugger) that can generate a dump file at any time.

If the system is still responsive to some extent, the Sysinternals *NotMyFault* tool can force a system crash and so force a dump file to be generated (this is in fact the way the dump in the previous section was generated). Figure 6-8 shows a screenshot of *NotMyFault*. Selecting the first (default) option and clicking *Crash* immediately crashes the system and will generate a dump file (if configured to do so).

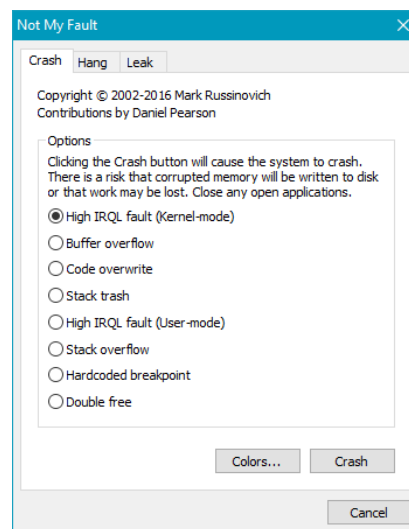


Figure 6-8: NotMyFault

NotMyFault uses a driver, *myfault.sys* that is actually responsible for the crash.



NotMyFault has 32 and 64 bit versions (the later file name ends with “64”). Remember to use the correct one for the system at hand, otherwise its driver will fail to load.

If the system is completely unresponsive, and you can attach a kernel debugger (the target was configured for debugging), then debug normally or generate a dump file using the `.dump` command.

If the system is unresponsive and a kernel debugger cannot be attached, it's possible to generate a crash manually if configured in the Registry beforehand (this assumes the hang was somehow expected). When a certain key combination is detected, the keyboard driver will generate a crash. Consult [this link](#) to get the full details. The crash code in this case is `0xe2` (`MANUALLY_INITIATED_CRASH`).

Thread Synchronization

Threads sometimes need to coordinate work. A canonical example is a driver using a linked list to gather data items. The driver can be invoked by multiple clients, coming from many threads in one or more processes. This means manipulating the linked list must be done atomically, so it's not corrupted. If multiple threads access the same memory where at least one is a writer (making changes), this is referred to as a *data race*. If a data race occurs, all bets are off and anything can happen. Typically, within a driver, a system crash occurs sooner or later; data corruption is practically guaranteed.

In such a scenario, it's essential that while one thread manipulates the linked list, all other threads back off the linked list, and wait in some way for the first thread to finish its work. Only then another thread (just one) can manipulate the list. This is an example of thread synchronization.

The kernel provides several primitives that help in accomplishing proper synchronization to protect data from concurrent access. The following discussed various primitives and techniques for thread synchronization.

Interlocked Operations

The Interlocked set of functions provide convenient operations that are performed atomically by utilizing the hardware, which means no software objects are involved. If using these functions gets the job done, then they should be used, as these are as efficient as they can possibly be.

Technically, these Interlocked-family of functions are called *compiler intrinsics*, as they are instructions to the processor, disguised as functions.



The same functions (intrinsics) are available in user-mode as well.

A simple example is incrementing an integer by one. Generally, this is not an atomic operation. If two (or more) threads try to perform this at the same time on the same memory location, it's possible (and likely) some of the increments will be lost. Figure 6-9 shows a simple scenario where incrementing a value by 1 done from two threads ends up with result of 1 instead of 2.

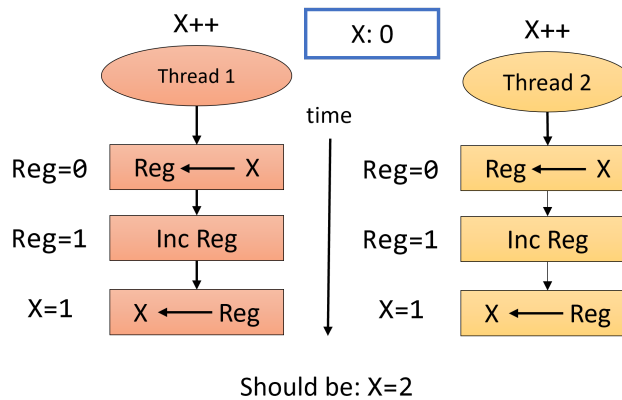


Figure 6-9: Concurrent increment



The example in figure 6-9 is extremely simplistic. With real CPUs there are other effects to consider, especially caching, which makes the shown scenario even more likely. CPU caching, store buffers, and other aspects of modern CPUs are non-trivial topics, well beyond the scope of this book.

Table 6-2 lists some of the Interlocked functions available for drivers use.

Table 6-2: Some Interlocked functions

Function	Description
InterlockedIncrement / InterlockedIncrement16 / InterlockedIncrement64	Atomically increment a 32/16/64 bit integer by one
InterlockedDecrement / 16 / 64	Atomically decrement a 32/16/64 bit integer by one.
InterlockedAdd / InterlockedAdd64	Atomically add one 32/64 bit integer to a variable.
InterlockedExchange / 8 / 16 / 64	Atomically exchange two 32/8/16/64 bit values.
InterlockedCompareExchange / 64 / 128	Atomically compare a variable with a value. If equal exchange with the provided value and return TRUE; otherwise, place the current value in the variable and return FALSE.



The InterlockedCompareExchange family of functions are used in *lock-free programming*, a programming technique to perform complex atomic operations without using software objects. This topic is well beyond the scope of this book.



The functions in table 6-2 are also available in user mode, as these are not really functions, but rather CPU *intrinsics* - special instructions to the CPU.

Dispatcher Objects

The kernel provides a set of primitives known as *Dispatcher Objects*, also called *Waitable Objects*. These objects have a state, either **signaled** or **non-signaled**, where the meaning of signaled and non-signaled depends on the type of object. They are called “waitable” because a thread can wait on such objects until they become signaled. While waiting, the thread does not consume CPU cycles as it’s in a *Waiting* state.

The primary functions used for waiting are `KeWaitForSingleObject` and `KeWaitForMultipleObjects`. Their prototypes (with simplified SAL annotations for clarity) are shown below:

```
NTSTATUS KeWaitForSingleObject (
    _In_ PVOID Object,
    _In_ KWAIT_REASON WaitReason,
    _In_ KPROCESSOR_MODE WaitMode,
    _In_ BOOLEAN Alertable,
    _In_opt_ PLARGE_INTEGER Timeout);

NTSTATUS KeWaitForMultipleObjects (
    _In_ ULONG Count,
    _In_reads_(Count) PVOID Object[],
    _In_ WAIT_TYPE WaitType,
    _In_ KWAIT_REASON WaitReason,
    _In_ KPROCESSOR_MODE WaitMode,
    _In_ BOOLEAN Alertable,
    _In_opt_ PLARGE_INTEGER Timeout,
    _Out_opt_ PKWAIT_BLOCK WaitBlockArray);
```

Here is a rundown of the arguments to these functions:

- *Object* - specifies the object to wait for. Note these functions work with objects, not handles. If you have a handle (maybe provided by user mode), call `ObReferenceObjectByHandle` to get the pointer to the object.
- *WaitReason* - specifies the wait reason. The list of wait reasons is pretty long, but drivers should typically set it to `Executive`, unless it’s waiting because of a user request, and if so specify `UserRequest`.
- *WaitMode* - can be `UserMode` or `KernelMode`. Most drivers should specify `KernelMode`.
- *Alertable* - indicates if the thread should be in an alertable state during the wait. Alertable state allows delivering of user mode *Asynchronous Procedure Calls* (APCs). User mode APCs can be delivered if wait mode is `UserMode`. Most drivers should specify `FALSE`.

- *Timeout* - specifies the time to wait. If `NULL` is specified, the wait is indefinite - as long as it takes for the object to become signaled. The units of this argument are in 100nsec chunks, where a negative number is relative wait, while a positive number is an absolute wait measured from January 1, 1601 at midnight.
- *Count* - the number of objects to wait on.
- *Object[]* - an array of object pointers to wait on.
- *WaitType* - specifies whether to wait for all object to become signaled at once (`waitAll`) or just one object (`waitAny`).
- *WaitBlockArray* - an array of structures used internally to manage the wait operation. It's optional if the number of objects is `<= THREAD_WAIT_OBJECTS` (currently 3) - the kernel will use the built-in array present in each thread. If the number of objects is higher, the driver must allocate the correct size of structures from non-paged memory, and deallocate them after the wait is over.

The main return values from `KeWaitForSingleObject` are:

- `STATUS_SUCCESS` - the wait is satisfied because the object state has become signaled.
- `STATUS_TIMEOUT` - the wait is satisfied because the timeout has elapsed.



Note that all return values from the wait functions pass the `NT_SUCCESS` macro with `true`.

`KeWaitForMultipleObjects` return values include `STATUS_TIMEOUT` just as `KeWaitForSingleObject`. `STATUS_SUCCESS` is returned if `waitAll` wait type is specified and all objects become signaled. For `waitAny` waits, if one of the objects became signaled, the return value is `STATUS_WAIT_0` plus its index in the array of objects (Note that `STATUS_WAIT_0` is defined to be zero).



There are some fine details associated with the wait functions, especially if wait mode is `UserMode` and the wait is alertable. Check the WDK docs for the details.

Table 6-3 lists some of the common dispatcher objects and the meaning of *signaled* and *non-signaled* for these objects.

Table 6-3: Object Types and signaled meaning

Object Type	Signaled meaning	Non-Signaled meaning
Process	process has terminated (for whatever reason)	process has not terminated
Thread	thread has terminated (for whatever reason)	thread has not terminated
Mutex	mutex is free (unowned)	mutex is held
Event	event is set	event is reset
Semaphore	semaphore count is greater than zero	semaphore count is zero
Timer	timer has expired	timer has not yet expired
File	asynchronous I/O operation completed	asynchronous I/O operation is in progress



All the object types from table 6-3 are also exported to user mode. The primary waiting functions in user mode are `WaitForSingleObject` and `WaitForMultipleObjects`.

The following sections will discuss some of common object types useful for synchronization in drivers. Some other objects will be discussed as well that are not dispatcher objects, but support waiting as well.

Mutex

Mutex is the classic object for the canonical problem of one thread among many that can access a shared resource at any one time.



Mutex is sometimes referred to as *Mutant* (its original name). These are the same thing.

A mutex is signaled when it's free. Once a thread calls a wait function and the wait is satisfied, the mutex becomes non-signaled and the thread becomes the owner of the mutex. Ownership is critical for a mutex. It means the following:

- If a thread is the owner of a mutex, it's the only one that can release the mutex.
- A mutex can be acquired more than once by the same thread. The second attempt succeeds automatically since the thread is the current owner of the mutex. This also means the thread needs to release the mutex the same number of times it was acquired; only then the mutex becomes free (signaled) again.

Using a mutex requires allocating a `KMUTEX` structure from non-paged memory. The mutex API contains the following functions working on that `KMUTEX`:

- `KeInitializeMutex` or `KeInitializeMutant` must be called once to initialize the mutex.

- One of the waiting functions, passing the address of the allocated KMUTEX structure.
- KeReleaseMutex is called when a thread that is the owner of the mutex wants to release it.

Here are the definitions of the APIs that can initialize a mutex:

```
VOID KeInitializeMutex (
    _Out_ PKMUTEX Mutex,
    _In_ ULONG Level);
VOID KeInitializeMutant ( // defined in ntifs.h
    _Out_ PKMUTANT Mutant,
    _In_ BOOLEAN InitialOwner);
```

The Level parameter in KeInitializeMutex is not used, so zero is a good value as any. KeInitializeMutant allows specifying if the current thread should be the initial owner of the mutex. KeInitializeMutex initializes the mutex to be unowned.

Releasing the mutex is done with KeReleaseMutex:

```
LONG KeReleaseMutex (
    _Inout_ PKMUTEX Mutex,
    _In_ BOOLEAN Wait);
```

The returned value is the previous state of the mutex object (including recursive ownership count), and should mostly be ignored (although it may sometimes be useful for debugging purposes). The Wait parameter indicates whether the next API call is going to be one of the wait functions. This is used as a hint to the kernel that can optimize slightly if the thread is about to enter a wait state.



As part of calling KeReleaseMutex, the IRQL is raised to DISPATCH_LEVEL. If Wait is TRUE, the IRQL is not lowered, which would allow the next wait function (KeWaitForSingleObject or KeWaitForSingleObject) to execute more efficiently, as no context switch can interfere.

Given the above functions, here is an example using a mutex to access some shared data so that only a single thread does so at a time:

```
KMUTEX MyMutex;
LIST_ENTRY DataHead;

void Init() {
    KeInitializeMutex(&MyMutex, 0);
}

void DoWork() {
    // wait for the mutex to be available

    KeWaitForSingleObject(&MyMutex, Executive, KernelMode, FALSE, nullptr);

    // access DataHead freely

    // once done, release the mutex

    KeReleaseMutex(&MyMutex, FALSE);
}
```

It's important to release the mutex no matter what, so it's better to use `__try / __finally` to make sure it's executed however the `__try` block is exited:

```
void DoWork() {
    // wait for the mutex to be available

    KeWaitForSingleObject(&MyMutex, Executive, KernelMode, FALSE, nullptr);
    __try {
        // access DataHead freely
    }
    __finally {
        // once done, release the mutex

        KeReleaseMutex(&MyMutex, FALSE);
    }
}
```

Figure 6-10 shows two threads attempting to acquire the mutex at roughly the same time, as they want to access the same data. One thread succeeds in acquiring the mutex, the other has to wait until the mutex is released by the owner before it can acquire it.

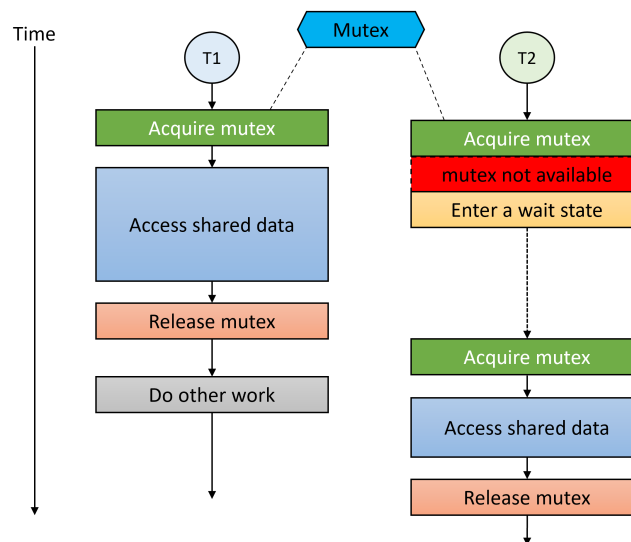


Figure 6-10: Acquiring a mutex

Since using `__try/__finally` is a bit awkward, we can use C++ to create a RAII wrapper for waits. This could also be used for other synchronization primitives.

First, we'll create a mutex wrapper that provides functions named `Lock` and `Unlock`:

```
struct Mutex {
    void Init() {
        KeInitializeMutex(&_amp;mutex, 0);
    }

    void Lock() {
        KeWaitForSingleObject(&_amp;mutex, Executive, KernelMode, FALSE, nullptr);
    }

    void Unlock() {
        KeReleaseMutex(&_amp;mutex, FALSE);
    }
}
```

```

    }

private:
    KMUTEX _mutex;
};

```

Then we can create a generic RAII wrapper for waiting for any type that has a `Lock` and `Unlock` functions:

```

template<typename TLock>
struct Locker {
    explicit Locker(TLock& lock) : _lock(lock) {
        lock.Lock();
    }

    ~Locker() {
        _lock.Unlock();
    }

private:
    TLock& _lock;
};

```

With these definitions in place, we can replace the code using the mutex with the following:

```

Mutex MyMutex;

void Init() {
    MyMutex.Init();
}

void DoWork() {
    Locker<Mutex> locker(MyMutex);

    // access DataHead freely
}

```



Since locking should be done for the shortest time possible, you can use an artificial C/C++ scope containing `Locker` and the code to execute while the mutex is owned, to acquire the mutex as late as possible and release it as soon as possible.

With C++ 17 and later, `Locker` can be used without specifying the type like so:

```

Locker locker(MyMutex);

```

Since Visual Studio currently uses C++ 14 as its default language standard, you'll have to change that in the project properties under the General node / C++ Language Standard.

We'll use the same `Locker` type with other synchronization primitives in subsequent sections.

Abandoned Mutex

A thread that acquires a mutex becomes the mutex owner. The owner thread is the only one that can release the mutex. What happens to the mutex if the owner thread dies for whatever reason? The mutex then becomes an *abandoned mutex*. The kernel explicitly releases the mutex (as no thread can do it) to prevent a deadlock, so another thread would be able to acquire that mutex

normally. However, the returned value from the next successful wait call is `STATUS_ABANDONED` rather than `STATUS_SUCCESS`. A driver should log such an occurrence, as it frequently indicates a bug.

Other Mutex Functions

Mutexes support a few miscellaneous functions that may be useful at times, mostly for debugging purposes. `KeReadStateMutex` returns the current state (recursive count) of the mutex, where 0 means “unowned”:

```
LONG KeReadStateMutex (_In_ PKMUTEX Mutex);
```

Just remember that after the call returns, the result may no longer be correct as the mutex state may have changed because some other thread has acquired or released the mutex before the code gets to examine the result. The benefit of this function is in debugging scenarios only.

You can get the current mutex owner with a call to `KeQueryOwnerMutant` (defined in `<ntifs.h>`) as a `CLIENT_ID` data structure, containing the thread and process IDs:

```
VOID KeQueryOwnerMutant (  
    _In_ PKMUTANT Mutant,  
    _Out_ PCLIENT_ID ClientId);
```

Just like with `KeReadStateMutex`, the returned information may be stale if other threads are doing work with that mutex.

Fast Mutex

A fast mutex is an alternative to the classic mutex, providing better performance. It’s not a dispatcher object, and so has its own API for acquiring and releasing it. A fast mutex has the following characteristics compared with a regular mutex:

- A fast mutex cannot be acquired recursively. Doing so causes a deadlock.
- When a fast mutex is acquired, the CPU IRQL is raised to `APC_LEVEL (1)`. This prevents any delivery of APCs to that thread.
- A fast mutex can only be waited on indefinitely - there is no way to specify a timeout.

Because of the first two bullets above, the fast mutex is slightly faster than a regular mutex. In fact, most drivers requiring a mutex use a fast mutex unless there is a compelling reason to use a regular mutex.



Don’t use I/O operations while holding on to a fast mutex. I/O completions are delivered with a special kernel APC, but those are blocked while holding a fast mutex, creating a deadlock.

A fast mutex is initialized by allocating a `FAST_MUTEX` structure from non-paged memory and calling `ExInitializeFastMutex`. Acquiring the mutex is done with `ExAcquireFastMutex` or

ExAcquireFastMutexUnsafe (if the current IRQL happens to be APC_LEVEL already). Releasing a fast mutex is accomplished with ExReleaseFastMutex or ExReleaseFastMutexUnsafe.

Semaphore

The primary goal of a semaphore is to limit something, such as the length of a queue. The semaphore is initialized with its maximum and initial count (typically set to the maximum value) by calling KeInitializeSemaphore. While its internal count is greater than zero, the semaphore is signaled. A thread that calls KeWaitForSingleObject has its wait satisfied, and the semaphore count drops by one. This continues until the count reaches zero, at which point the semaphore becomes non-signalized.

Semaphores use the KSEMAPHORE structure to hold their state, which must be allocated from non-paged memory. Here is the definition of KeInitializeSemaphore:

```
VOID KeInitializeSemaphore (
    _Out_ PRKSEMAPHORE Semaphore,
    _In_ LONG Count,           // starting count
    _In_ LONG Limit);         // maximum count
```

As an example, imagine a queue of work items managed by the driver. Some threads want to add items to the queue. Each such thread calls KeWaitForSingleObject to obtain one “count” of the semaphore. As long as the count is greater than zero, the thread continues and adds an item to the queue, increasing its length, and semaphore “loses” a count. Some other threads are tasked with processing work items from the queue. Once a thread removes an item from the queue, it calls KeReleaseSemaphore that increments the count of the semaphore, moving it to the signaled state again, allowing potentially another thread to make progress and add a new item to the queue.

KeReleaseSemaphore is defined like so:

```
LONG KeReleaseSemaphore (
    _Inout_ PRKSEMAPHORE Semaphore,
    _In_ KPRIORITY Increment,
    _In_ LONG Adjustment,
    _In_ BOOLEAN Wait);
```

The Increment parameter indicates the priority boost to apply to the thread that has a successful waiting on the semaphore. The details of how this boost works are described in the next chapter. Most drivers should provide the value 1 (that’s the default used by the kernel when a semaphore is released by the user mode ReleaseSemaphore API). Adjustment is the value to add to the semaphore’s current count. It’s typically one, but can be a higher value if that makes sense. The last parameter (wait) indicates whether a wait operation (KeWaitForSingleObject or KeWaitForMultipleObjects) immediately follows (see the information bar in the mutex discussion above). The function returns the old count of the semaphore.



Is a semaphore with a maximum count of one equivalent to a mutex? At first, it seems so, but this is not the case. A semaphore lacks ownership, meaning one thread can acquire the semaphore, while another can release it. This is a strength, not a weakness, as described in the above example. A Semaphore's purpose is very different from that of a mutex.

You can read the current count of the semaphore by calling `KeReadStateSemaphore`:

```
LONG KeReadStateSemaphore (_In_ PRKSEMAPHORE Semaphore);
```

Event

An event encapsulates a boolean flag - either true (signaled) or false (non-signaled). The primary purpose of an event is to signal something has happened, to provide *flow synchronization*. For example, if some condition becomes true, an event can be set, and a bunch of threads can be released from waiting and continue working on some data that perhaps is now ready for processing.

There are two types of events, the type being specified at event initialization time:

- Notification event (manual reset) - when this event is set, it releases any number of waiting threads, and the event state remains set (signaled) until explicitly reset.
- Synchronization event (auto reset) - when this event is set, it releases at most one thread (no matter how many are waiting for the event), and once released the event goes back to the reset (non-signaled) state automatically.

An event is created by allocating a `KEVENT` structure from non-paged memory and then calling `KeInitializeEvent` to initialize it, specifying the event type (`NotificationEvent` or `SynchronizationEvent`) and the initial event state (signaled or non-signaled):

```
VOID KeInitializeEvent (
    _Out_ PRKEVENT Event,
    _In_ EVENT_TYPE Type,      // NotificationEvent or SynchronizationEvent
    _In_ BOOLEAN State);      // initial state (signaled=TRUE)
```

Notification events are called *Manual-reset* in user-mode terminology, while Synchronization events are called *Auto-reset*. Despite the name changes, these are the same.

Waiting for an event is done normally with the `KeWaitXXX` functions. Calling `KeSetEvent` sets the event to the signaled state, while calling `KeResetEvent` or `KeClearEvent` resets it (non-signaled state) (the latter function being a bit quicker as it does not return the previous state of the event):

```
LONG KeSetEvent (
    _Inout_ PRKEVENT Event,
    _In_ KPRIORITY Increment,
    _In_ BOOLEAN Wait);
VOID KeClearEvent (_Inout_ PRKEVENT Event);
LONG KeResetEvent (_Inout_ PRKEVENT Event);
```

Just like with a semaphore, setting an event allows providing a priority boost to the next successful wait on the event.

Finally, The current state of an event (signaled or non-signaled) can be read with `KeReadStateEvent`:

```
LONG KeReadStateEvent (_In_ PRKEVENT Event);
```

Named Events

Event objects can be named (as can mutexes and semaphores). This can be used as an easy way of sharing an event object with other drivers or with user-mode clients. One way of creating or opening a named event by name is with the helper functions `IoCreateSynchronizationEvent` and `IoCreateNotificationEvent` APIs:

```
PKEVENT IoCreateSynchronizationEvent(  
    _In_ PUNICODE_STRING EventName,  
    _Out_ PHANDLE EventHandle);  
PKEVENT IoCreateNotificationEvent(  
    _In_ PUNICODE_STRING EventName,  
    _Out_ PHANDLE EventHandle);
```

These APIs create the named event object if it does not exist and set its state to signaled, or obtain another handle to the named event if it does exist. The name itself is provided as a normal `UNICODE_STRING` and must be a full path in the Object Manager's namespace, as can be observed in the Sysinternals *WinObj* tool.

These APIs return two values: the pointer to the event object (direct returned value) and an open handle in the `EventHandle` parameter. The returned handle is a kernel handle, to be used by the driver only. The functions return `NULL` on failure.

You can use the previously described events API to manipulate the returned event by address. Don't forget to close the returned handle (`ZwClose`) to prevent a leak. Alternatively, you can call `ObReferenceObject` on the returned pointer to make sure it's not prematurely destroyed and close the handle immediately. In that case, call `ObDereferenceObject` when you're done with the event.

Built-in Named Kernel Events

One use of the `IoCreateNotificationEvent` API is to gain access to a bunch of named event objects the kernel provides in the `\KernelObjects` directory. These events provide various notifications for memory related status, that may be useful for kernel drivers.

Figure 6-11 shows the named events in *WinObj*. Note that the lower symbolic links are actually events, as these are internally implemented as *Dynamic Symbolic Links* (see more details at <https://scorpiosoftware.net/2021/04/30/dynamic-symbolic-links/>).

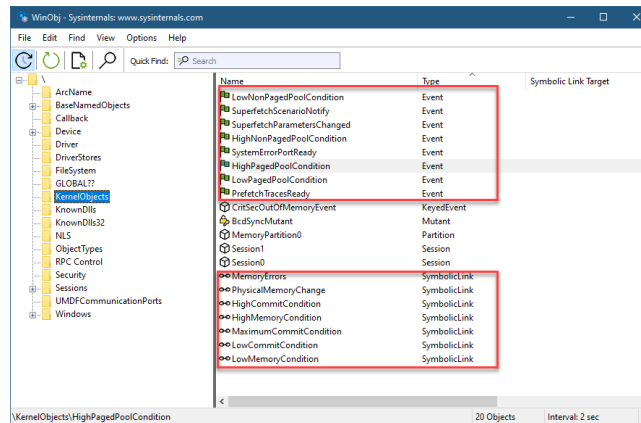


Figure 6-11: Kernel Named Events

All the events shown in figure 6-11 are Notification events. Table 6-5 lists these events with their meaning.

Name	Table 6-5: Named kernel events Description
HighMemoryCondition	The system has lots of free physical memory
LowMemoryCondition	The system is low on physical memory
HighPagedPoolCondition	The system has lots of free paged pool memory
LowPagedPoolCondition	The system is low on paged pool memory
HighNonPagedPoolCondition	The system has lots of free non-paged pool memory
LowNonPagedPoolCondition	The system is low on non-paged pool memory
HighCommitCondition	The system has lots of free memory in RAM and paging file(s)
LowCommitCondition	The system is low on RAM and paging file(s)
MaximumCommitCondition	The system is almost out of memory, and no further increase in page files size is possible

Drivers can use these events as hints to either allocate more memory or free memory as required. The following example shows how to obtain one of these events and wait for it on some thread (error handling ommitted):

```

UNICODE_STRING name;
RtlInitUnicodeString(&name, L"\\KernelObjects\\LowCommitCondition");
HANDLE hEvent;
auto event = IoCreateNotificationEvent(&name, &hEvent);

// on some driver-created thread...
KeWaitForSingleObject(event, Executive, KernelMode, FALSE, nullptr);
// free some memory if possible...
//
// close the handle
ZwClose(hEvent);

```



Write a driver that waits on all these named events and uses `DbgPrint` to indicate a signaled event with its description.

Executive Resource

The classic synchronization problem of accessing a shared resource by multiple threads was dealt with by using a mutex or fast mutex. This works, but mutexes are pessimistic, meaning they allow a single thread to access a shared resource. That may be unfortunate in cases where multiple threads access a shared resource by reading only.

In cases where it's possible to distinguish data changes (writes) vs. just looking at the data (reading) - there is a possible optimization. A thread that requires access to the shared resource can declare its intentions - read or write. If it declares read, other threads declaring read can do so concurrently, improving performance. This is especially useful if the shared data changes infrequently, i.e. there are considerably more reads than writes.

Mutexes by their very nature are pessimistic locks, since they enforce a single thread at a time execution. This makes them always work at the expense of possible performance gains with concurrency.

The kernel provides yet another synchronization primitive that is geared towards this scenario, known as *single writer, multiple readers*. This object is the *Executive Resource*, another special object which is not a dispatcher object.

Initializing an executive resource is done by allocating an `ERESOURCE` structure from non-paged pool and calling `ExInitializeResourceLite`. Once initialized, threads can acquire either the exclusive lock (for writes) using `ExAcquireResourceExclusiveLite` or the shared lock by calling `ExAcquireResourceSharedLite`. Once done the work, a thread releases the executive resource with `ExReleaseResourceLite` (no matter whether it acquired as exclusive or not).

The requirement for using the acquire and release functions is that normal kernel APCs must be disabled. This can be done with `KeEnterCriticalRegion` just before the acquire call, and then `KeLeaveCriticalRegion` just after the release call. The following code snippet demonstrates that:

```
ERESOURCE resource;

void WriteData() {
    KeEnterCriticalRegion();
    ExAcquireResourceExclusiveLite(&resource, TRUE);    // wait until acquired

    // Write to the data

    ExReleaseResourceLite(&resource);
    KeLeaveCriticalRegion();
}
```

Since these calls are so common when working with executive resources, there are functions that perform both operations with a single call:

```
void WriteData() {
    ExEnterCriticalRegionAndAcquireResourceExclusive(&resource);

    // Write to the data

    ExReleaseResourceAndLeaveCriticalRegion(&resource);
}
```

A similar function exists for shared acquisition, `ExEnterCriticalRegionAndAcquireResourceShared`. Finally, before freeing the memory the resource occupies, call `ExDeleteResourceLite` to remove the resource from the kernel's resource list:

```
NTSTATUS ExDeleteResourceLite(
    _Inout_ PERESOURCE Resource);
```

You can query the number of waiting threads for exclusive and shared access of a resource with the functions `ExGetExclusiveWaiterCount` and `ExGetSharedWaiterCount`, respectively.

There are other functions for working with executive resources for some specialized cases. Consult the WDK documentation for more information.



Create appropriate C++ RAII wrappers for executive resources.

High IRQL Synchronization

The sections on synchronization so far have dealt with threads waiting for various types of objects. However, in some scenarios, threads cannot wait - specifically, when the processor's IRQL is `DISPATCH_LEVEL` (2) or higher. This section discusses these scenarios and how to handle them.

Let's examine an example scenario: A driver has a timer, set up with `KeSetTimer` and uses a DPC to execute code when the timer expires. At the same time, other functions in the driver, such as `IRP_MJ_DEVICE_CONTROL` may execute at the same time (runs at IRQL 0). If both these functions need to access a shared resource (e.g. a linked list), they must synchronize access to prevent data corruption.

The problem is that a DPC cannot call `KeWaitForSingleObject` or any other waiting function - calling any of these is fatal. So how can these functions synchronize access?

The simple case is where the system has a single CPU. In this case, when accessing the shared resource, the low IRQL function just needs to raise IRQL to `DISPATCH_LEVEL` and then access the resource. During that time a DPC cannot interfere with this code since the CPU's IRQL is already 2. Once the code is done with the shared resource, it can lower the IRQL back to zero, allowing

the DPC to execute. This prevents execution of these routines at the same time. Figure 6-12 shows this setup.

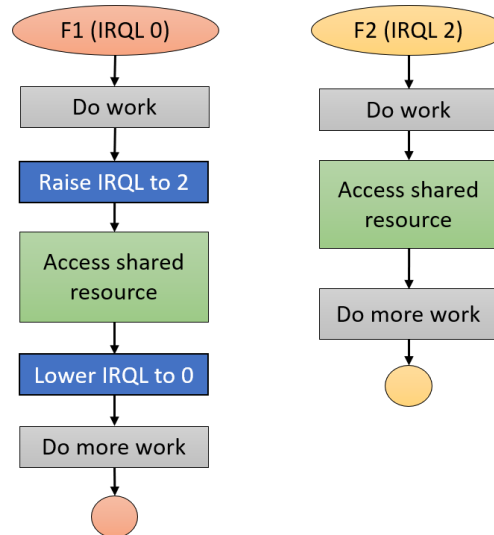


Figure 6-12: High-IRQL synchronization by manipulating IRQL

In standard systems, where there is more than one CPU, this synchronization method is not enough, because the IRQL is a CPU's property, not a system-wide property. If one CPU's IRQL is raised to 2, if a DPC needs to execute, it can execute on another CPU whose IRQL may be zero. In this case, it's possible that both functions execute at the same time, accessing the shared data, causing a data race.

How can we solve that? We need something like a mutex, but that can synchronize between processors - not threads. That's because when the CPU's IRQL is 2 or higher, the thread itself loses meaning because the scheduler cannot do work on that CPU. This kind of object exists - the *Spin Lock*.

The Spin Lock

The Spin Lock is just a bit in memory that is used with atomic test-and-set operations via an API. When a CPU tries to acquire a spin lock, and that spin lock is not currently free (the bit is set), the CPU keeps spinning on the spin lock, busy waiting for it to be released by another CPU (remember, putting the thread into a waiting state cannot be done at IRQL DISPATCH_LEVEL or higher).

In the scenario depicted in the previous section, a spin lock would need to be allocated and initialized. Each function that requires access to the shared data needs to raise IRQL to 2 (if not already there), acquire the spin lock, perform the work on the shared data, and finally release the spin lock and lower IRQL back (if applicable; not so for a DPC). This chain of events is depicted in figure 6-13.

Creating a spin lock requires allocating a KSPIN_LOCK structure from non-paged pool, and calling KeInitializeSpinLock. This puts the spin lock in the unowned state.

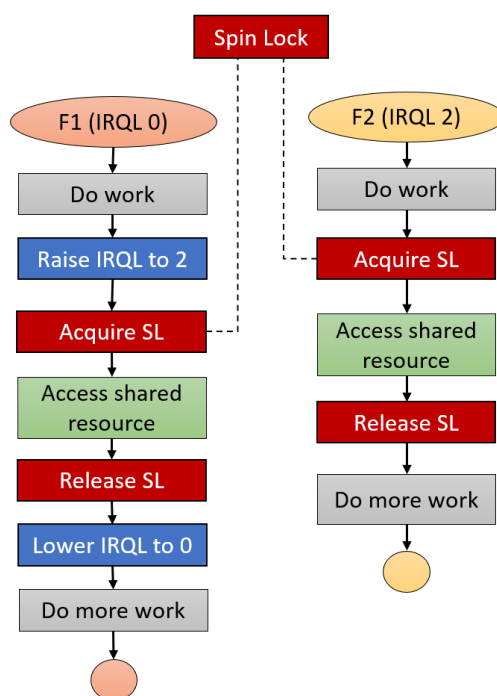


Figure 6-13: High-IRQL synchronization with a Spin Lock

Acquiring a spin lock is always a two-step process: first, raise the IRQL to the proper level, which is the highest level of any function trying to synchronize access to a shared resource. In the previous example, this *associated IRQL* is 2. Second, acquire the spin lock. These two steps are combined by using the appropriate API. This process is depicted in figure 6-14.

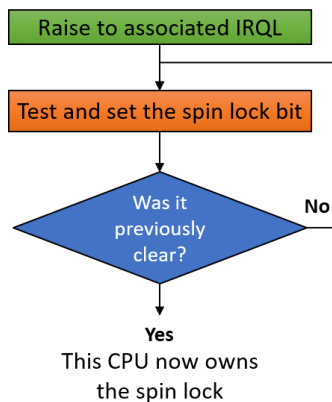


Figure 6-14: Acquiring a Spin Lock

Acquiring and releasing a spin lock is done using an API that performs the two steps outlined in figure 6-12. Table 6-4 shows the relevant APIs and the associated IRQL for the spin locks they

operate on.

Table 6-4: APIs for working with spin locks

IRQL	Acquire function	Release function	Remarks
DISPATCH_LEVEL (2)	KeAcquireSpinLock	KeReleaseSpinLock	
DISPATCH_LEVEL (2)	KeAcquireSpinLockAtDpcLevel	KeReleaseSpinLockFromDpcLevel	(a)
Device IRQL	KeAcquireInterruptSpinLock	KeReleaseInterruptSpinLock	(b)
Device IRQL	KeSynchronizeExecution	(none)	(c)
HIGH_LEVEL	ExInterlockedXxx	(none)	(d)

Remarks on table 6-4:

(a) Can be called at IRQL 2 only. Provides an optimization that just acquires the spin lock without changing IRQLs. The canonical scenario is calling these APIs within a DPC routine.

(b) Useful for synchronizing an ISR with any other function. Hardware-based drivers with an interrupt source use these routines. The argument is an interrupt object (KINTERRUPT), where the spin lock is part of it.

(c) KeSynchronizeExecution acquires the interrupt object spin lock, calls the provided callback and releases the spin lock. The net effect is the same as calling the pair KeAcquireInterruptSpinLock / KeReleaseInterruptSpinLock.

(d) A set of three functions for manipulating LIST_ENTRY-based linked lists. These functions use the provided spin lock and raise IRQL to HIGH_LEVEL. Because of the high IRQL, these routines can be used in any IRQL, since raising IRQL is always a safe operation.



If you acquire a spin lock, be sure to release it in the same function. Otherwise, you're risking a deadlock or a system crash.



Where do spin locks come from? The scenario described here requires the driver to allocate its own spin lock to protect concurrent access to its own data from high-IRQL functions. Some spin locks exist as part of other objects, such as the KINTERRUPT object used by hardware-based drivers that handle interrupts. Another example is a system-wide spin lock known as the *Cancel spin lock*, which is acquired by the kernel before calling a cancellation routine registered by a driver. This is the only case where a driver released a spin lock it has not acquired explicitly.

If several CPUs try to acquire the same spin lock at the same time, which CPU gets the spin lock first? Normally, there is no order - the CPU with fastest electrons wins :). The kernel does provide an alternative, called *Queued spin locks* that serve CPUs on a FIFO basis. These only work with IRQL DISPATCH_LEVEL. The relevant APIs are KeAcquireInStackQueuedSpinLock and KeReleaseInStackQueuedSpinLock. Check the WDK documentation for more details.



Write a C++ wrapper for a DISPATCH_LEVEL spin lock that works with the Locker RAII class defined earlier in this chapter.

Queued Spin Locks

A variant on classic spin locks are *queued spin locks*. These behave the same as normal spin locks, with the following differences:

- Queued spin locks always raise to IRQL DISPATCH_LEVEL (2). This means they cannot be used for synchronizing with an ISR, for example.
- There is a queue of CPU waiting to acquire the spin lock, on a FIFO basis. This is more efficient when high contention is expected. Normal spin locks provide no guarantee as to the order of acquisition when multiple CPUs attempt to acquire a spin lock.

A queued spin lock is initialized just like a normal spin lock (`KeInitializeSpinLock`). Acquiring and releasing a queued spin lock is achieved with different APIs:

```
void KeAcquireInStackQueuedSpinLock (
    _Inout_ PKSPIN_LOCK SpinLock,
    _Out_ PKLOCK_QUEUE_HANDLE LockHandle);
void KeReleaseInStackQueuedSpinLock (
    _In_ PKLOCK_QUEUE_HANDLE LockHandle);
```

Except for a spin lock, the caller provides an opaque `KLOCK_QUEUE_HANDLE` structure that is filled in by `KeAcquireInStackQueuedSpinLock`. The same one must be passed to `KeReleaseInStackQueuedSpinLock`.

Just like with normal dispatch-level spin locks, shortcuts exist if the caller is already at IRQL DISPATCH_LEVEL. `KeAcquireInStackQueuedSpinLockAtDpcLevel` acquires the spin lock with no IRQL changes, while `KeReleaseInStackQueuedSpinLockFromDpcLevel` releases it.



Write a C++ RAII wrapper for a queued spin lock.

Work Items

Sometimes there is a need to run a piece of code on a different thread than the executing one. One way to do that is to create a thread explicitly and task it with running the code. The kernel provides functions that allow a driver to create a separate thread of execution:

`PsCreateSystemThread` and `IoCreateSystemThread` (available in Windows 8+). These functions are appropriate if the driver needs to run code in the background for a long time. However, for time-bound operations, it's better to use a kernel-provided thread pool that will execute your code on some system worker thread.

`PsCreateSystemThread` and `IoCreateSystemThread` are discussed in chapter 8.



`IoCreateSystemThread` is preferred over `PsCreateSystemThread`, because it allows associating a device or driver object with the thread. This makes the I/O system add a reference to the object, which makes sure the driver cannot be unloaded prematurely while the thread is still executing.

A driver-created thread must terminate itself eventually by calling `PsTerminateSystemThread`. This function never returns if successful.

Work items is the term used to describe functions queued to the system thread pool. A driver can allocate and initialize a work item, pointing to the function the driver wishes to execute, and then the work item can be queued to the pool. This may seem very similar to a DPC, the primary difference being work items always execute at IRQL `PASSIVE_LEVEL` (0). Thus, work items can be used by IRQL 2 code (such as DPCs) to perform operations not normally allowed at IRQL 2 (such as I/O operations).

Creating and initializing a work item can be done in one of two ways:

- Allocate and initialize the work item with `IoAllocateWorkItem`. The function returns a pointer to the opaque `IO_WORKITEM`. When finished with the work item it must be freed with `IoFreeWorkItem`.
- Allocate an `IO_WORKITEM` structure dynamically with size provided by `IoSizeofWorkItem`. Then call `IoInitializeWorkItem`. When finished with the work item, call `IoUninitializeWorkItem`.

These functions accept a device object, so make sure the driver is not unloaded while there is a work item queued or executing.



There is another set of APIs for work items, all start with `Ex`, such as `ExQueueWorkItem`. These functions do not associate the work item with anything in the driver, so it's possible for the driver to be unloaded while a work item is still executing. These APIs are marked as *deprecated* - always prefer using the `Io` functions.

To queue the work item, call `IoQueueWorkItem`. Here is its definition:

```

viuid IoQueueWorkItem(
    _Inout_ PIO_WORKITEM IoWorkItem,           // the work item
    _In_ PIO_WORKITEM_ROUTINE WorkerRoutine,    // the function to be called
    _In_ WORK_QUEUE_TYPE QueueType,            // queue type
    _In_opt_ PVOID Context);                   // driver-defined value

```

The callback function the driver needs to provide has the following prototype:

```

IO_WORKITEM_ROUTINE WorkItem;

void WorkItem(
    _In_ PDEVICE_OBJECT DeviceObject,
    _In_opt_ PVOID Context);

```

The system thread pool has several queues (at least logically), based on the thread priorities that serve these work items. There are several levels defined:

```
typedef enum _WORK_QUEUE_TYPE {
    CriticalWorkQueue,           // priority 13
    DelayedWorkQueue,           // priority 12
    HyperCriticalWorkQueue,     // priority 15
    NormalWorkQueue,           // priority 8
    BackgroundWorkQueue,       // priority 7
    RealTimeWorkQueue,         // priority 18
    SuperCriticalWorkQueue,     // priority 14
    MaximumWorkQueue,
    CustomPriorityWorkQueue = 32
} WORK_QUEUE_TYPE;
```

The documentation indicates `DelayedWorkQueue` must be used, but in reality, any other supported level can be used.



There is another function that can be used to queue a work item: `IoQueueWorkItemEx`. This function uses a different callback that has an added parameter which is the work item itself. This is useful if the work item function needs to free the work item before it exits.

Summary

In this chapter, we looked at various kernel mechanisms driver developers should be aware of and use. In the next chapter, we'll take a closer look at *I/O Request Packets* (IRPs).

Chapter 7: The I/O Request Packet

After a typical driver completes its initialization in `DriverEntry`, its primary job is to handle requests. These requests are packaged as the semi-documented *I/O Request Packet* (IRP) structure. In this chapter, we'll take a deeper look at IRPs and how a driver handles common IRP types.

In This chapter:

- **Introduction to IRPs**
 - **Device Nodes**
 - **IRP and I/O Stack Location**
 - **Dispatch Routines**
 - **Accessing User Buffers**
 - **Putting it All Together: The Zero Driver**
-

Introduction to IRPs

An IRP is a structure that is allocated from non-paged pool typically by one of the “managers” in the Executive (I/O Manager, Plug & Play Manager, Power Manager), but can also be allocated by the driver, perhaps for passing a request to another driver. Whichever entity allocating the IRP is also responsible for freeing it.

An IRP is never allocated alone. It's always accompanied by one or more I/O Stack Location structures (`IO_STACK_LOCATION`). In fact, when an IRP is allocated, the caller must specify how many I/O stack locations need to be allocated with the IRP. These I/O stack locations follow the IRP directly in memory. The number of I/O stack locations is the number of device objects in the device stack. We'll discuss device stacks in the next section. When a driver receives an IRP, it gets a pointer to the IRP structure itself, knowing it's followed by a set of I/O stack location, one of which is for the driver's use. To

get the correct I/O stack location, a driver calls `IoGetCurrentIrpStackLocation` (actually a macro). Figure 7-1 shows a conceptual view of an IRP and its associated I/O stack locations.

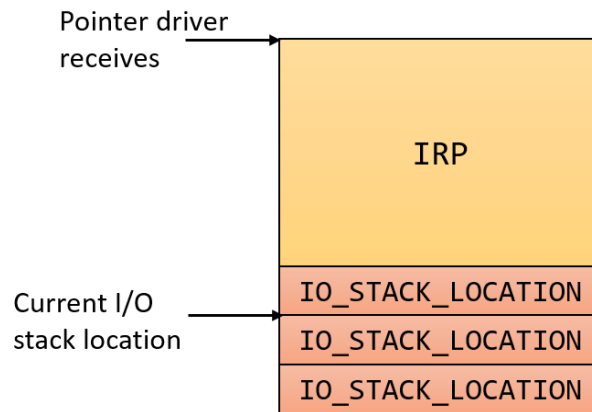


Figure 7-1: IRP and its I/O stack locations

The parameters of the request are somehow “split” between the main IRP structure and the current `IO_STACK_LOCATION`.

Device Nodes

The I/O system in Windows is device-centric, rather than driver-centric. This has several implications:

- Device objects can be named, and handles to device objects can be opened. The `CreateFile` function accepts a symbolic link that leads to a device object. `CreateFile` cannot accept a driver’s name as an argument.
- Windows supports device layering - one device can be layered on top of another. Any request destined for a lower device will reach the uppermost device first. This layering is common for hardware-based devices, but it works with any device type.

Figure 7-2 shows an example of several layers of devices, “stacked” one on top of the other. This set of devices is known as a *device stack*, sometimes referred to as *device node* (although the term *device node* is often used with hardware device stacks). Figure 7-1 shows six layers, or six devices. Each of these devices is represented by a `DEVICE_OBJECT` structure created by calling the standard `IoCreateDevice` function.

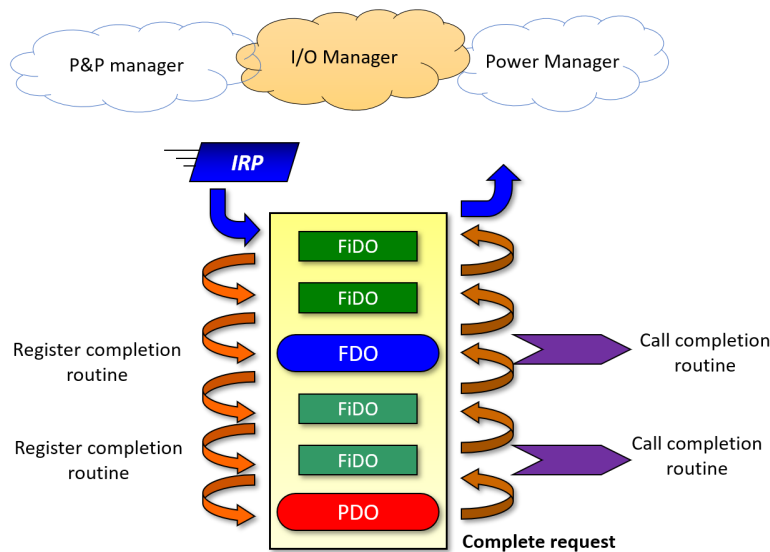


Figure 7-2: Layered devices

The different device objects that comprise the device node (devnode) layers are labeled according to their role in the devnode. These roles are relevant in a hardware-based devnode.

All the device objects in figure 7-2 are just `DEVICE_OBJECT` structures, each created by a different driver that is in charge of that layer. More generically, this kind of device node does not have to be related to hardware-based device drivers.

Here is a quick rundown of the meaning of the labels present in figure 7-2:

- *PDO (Physical Device Object)* - Despite the name, there is nothing “physical” about it. This device object is created by a bus driver - the driver that is in charge of the particular bus (e.g. PCI, USB, etc.). This device object represents the fact that there is some device in that slot on that bus.
- *FDO (Functional Device Object)* - This device object is created by the “real” driver; that is, the driver typically provided by the hardware’s vendor that understands the details of the device intimately.
- *FiDO (Filter Device Object)* - These are optional filter devices created by filter drivers.

The Plug & Play (P&P) manager, in this case, is responsible for loading the appropriate drivers, starting from the bottom. As an example, suppose the devnode in figure 7-2 represents a set of drivers that manage a PCI network card. The sequence of events leading to the creation of this devnode can be summarized as follows:

1. The PCI bus driver (*pci.sys*) recognizes the fact that there is something in that particular slot. It creates a PDO (*IoCreateDevice*) to represent this fact. The bus driver has no idea whether this is a network card, video card or something else; it only knows there is something there and can extract basic information from its controller, such as the Vendor ID and Device ID of the device.
2. The PCI bus driver notifies the P&P manager that it has changes on its bus.
3. The P&P manager requests a list of PDOs managed by the bus driver. It receives back a list of PDOs, in which this new PDO is included.
4. Now the P&P manager's job is to find and load the proper driver that new PDO. It issues a query to the bus driver to request the full hardware device ID.
5. With this hardware ID in hand, the P&P manager looks in the Registry at *HKLM\System\CurrentControlSet\Enum\PCI\{HardwareID}*. If the driver has been loaded before, it will be registered there, and the P&P manager will load it. Figure 7-3 shows an example hardware ID in the registry (NVIDIA display driver).
6. The driver loads and creates the FDO (another call to *IoCreateDevice*), but adds an additional call to *IoAttachDeviceToDeviceStack*, thus attaching itself over the previous layer (typically the PDO).

We'll see how to write filter drivers that take advantage of *IoAttachDeviceToDeviceStack* in chapter 13.

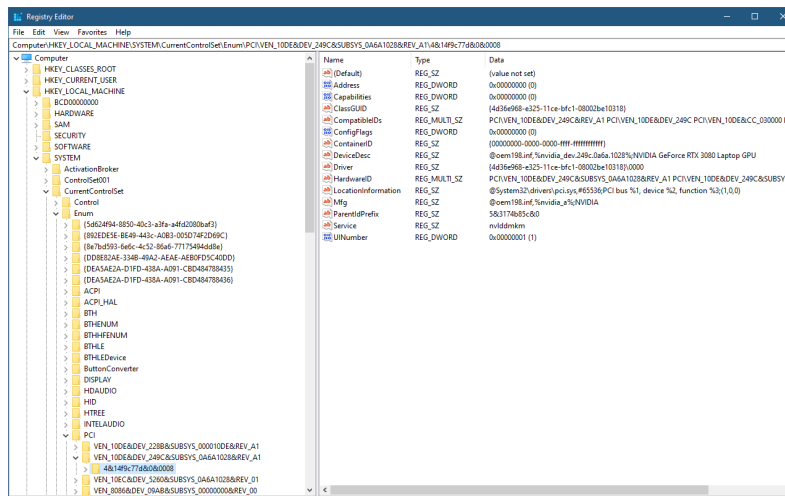


Figure 7-3: Hardware ID information



The value **Service** in figure 7-3 indirectly points to the actual driver at *HKLM\System\CurrentControlSet\Services\{ServiceName}* where all drivers must be registered.

The filter device objects are loaded as well, if they are registered correctly in the Registry. Lower filters (below the FDO) load in order, from the bottom. Each filter driver loaded creates its own device object and attached it on top of the previous layer. Upper filters work the same way but are loaded after the FDO. All this means that with operational P&P devnodes, there are at least two layers - PDO and FDO, but there could be more if filters are involved. We'll look at basic filter development for hardware-based drivers in chapter 13.

Full discussion of Plug & Play and the exact way this kind of devnode is built is beyond the scope of this book. The previous description is incomplete and glances over some details, but it should give you the basic idea. Every devnode is built from the bottom up, regardless of whether it is related to hardware or not.

Lower filters are searched in two locations: the hardware ID key shown in figure 7-3 and in the corresponding class based on the **ClassGuid** value listed under *HKLM\System\CurrentControlSet\Control\Classes*. The value name itself is **LowerFilters** and is a multiple string value holding service names, pointing to the same *Services* key. Upper filters are searched in a similar manner, but the

value name is **UpperFilters**. Figure 7-4 shows the registry settings for the *DiskDrive* class, which has a lower filter and an upper filter.

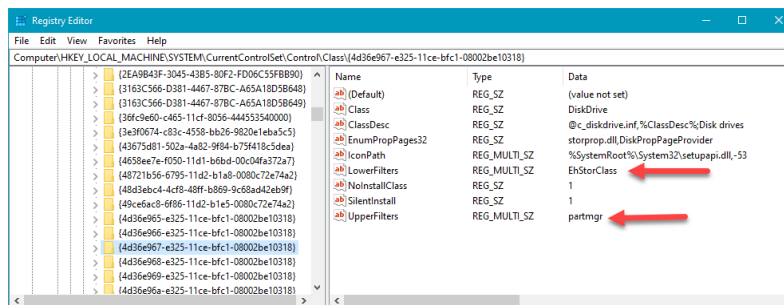


Figure 7-4: The *DiskDrive* class key

IRP Flow

Figure 7-2 shows an example devnode, whether related to hardware or not. An IRP is created by one of the managers in the Executive - for most of our drivers that is the I/O Manager.

The manager creates an IRP with its associated `IO_STACK_LOCATIONS` - six in the example in figure 7-2. The manager initializes the main IRP structure and the first I/O stack location only. Then it passes the IRP's pointer to the uppermost layer.

A driver receives the IRP in its appropriate dispatch routine. For example, if this is a Read IRP, then the driver will be called in its `IRP_MJ_READ` index of its `MajorFunction` array from its driver object. At this point, a driver has several options when dealing with IRP:

- Pass the request down - if the driver's device is not the last device in the devnode, the driver can pass the request along if it's not interesting for the driver. This is typically done by a filter driver that receives a request that it's not interested in, and in order not to hurt the functionality of the device (since the request is actually destined for a lower-layer device), the driver can pass it down. This must be done with two calls:
 - Call `IoSkipCurrentIrpStackLocation` to make sure the next device in line is going to see the same information given to this device - it should see the same I/O stack location.

- Call `IoCallDriver` passing the lower device object (which the driver received at the time it called `IoAttachDeviceToDeviceStack`) and the IRP.

Before passing the request down, the driver must prepare the next I/O stack location with proper information. Since the I/O manager only initializes the first I/O stack location, it's the responsibility of each driver to initialize the next one. One way to do that is to call `IoCopyIrpStackLocationToNext` before calling `IoCallDriver`. This works, but is a bit wasteful if the driver just wants the lower layer to see the same information. Calling `IoSkipCurrentIrpStackLocation` is an optimization which decrements the current I/O stack location pointer inside the IRP, which is later incremented by `IoCallDriver`, so the next layer sees the same `IO_STACK_LOCATION` this driver has seen. This decrement/increment dance is more efficient than making an actual copy.

- Handle the IRP fully - the driver receiving the IRP can just handle the IRP without propagating it down by eventually calling `IoCompleteRequest`. Any lower devices will never see the request.
- Do a combination of the above options - the driver can examine the IRP, do something (such as log the request), and then pass it down. Or it can make some changes to the next I/O stack location, and then pass the request down.
- Pass the request down (with or without changes) and be notified when the request completes by a lower layer device - Any layer (except the lowest one) can set up an I/O completion routine by calling `IoSetCompletionRoutine` before passing the request down. When one of the lower layers completes the request, the driver's completion routine will be called.
- Start some asynchronous IRP handling - the driver may want to handle the request, but if the request is lengthy (typical of a hardware driver, but also could be the case for a software driver), the driver may mark the IRP as pending by calling `IoMarkIrpPending` and return a `STATUS_PENDING` from its dispatch routine. Eventually, it will have to complete the IRP.

Once some layer calls `IoCompleteRequest`, the IRP turns around and starts "bubbling up" towards the originator of the IRP (typically one of the I/O System Managers). If completion routines have been registered, they will be invoked in reverse order of registration.

In most drivers in this book, layering will not be considered, since the driver is most likely the single device in its devnode. The driver will handle the request then and there or handle it asynchronously; it will not pass it down, as there is no device underneath.

We'll discuss other aspects of IRP handling in filter drivers, including completion routines, in chapter 13.

IRP and I/O Stack Location

Figure 7-5 shows some of the important fields in an IRP.

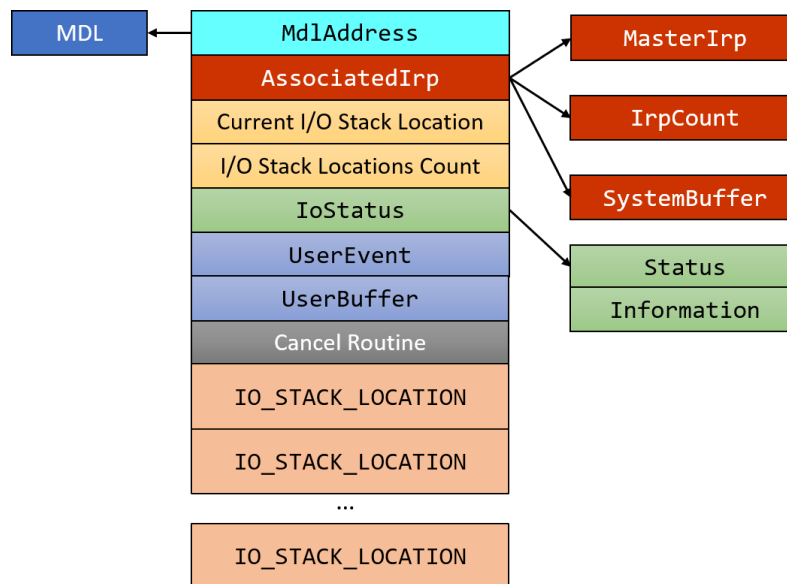


Figure 7-5: Important fields of the IRP structure

Here is a quick rundown of these fields:

- **IoStatus** - contains the Status (NT_STATUS) of the IRP and an Information field. The Information field is a polymorphic one, typed as ULONG_PTR (32 or 64-bit integer), but its meaning depends on the type of IRP. For example, for Read and Write IRPs, its meaning is the number of bytes transferred in the operation.
- **UserBuffer** - contains the raw buffer pointer to the user's buffer for relevant IRPs. Read and Write IRPs, for instance, store the user's buffer pointer in this field. In DeviceIoControl IRPs, this points to the output buffer provided in the request.

- **UserEvent** - this is a pointer to an event object (KEVENT) that was provided by a client if the call is asynchronous and such an event was supplied. From user mode, this event can be provided (with a HANDLE) in the OVERLAPPED structure that is mandatory for invoking I/O operations asynchronously.
- **AssociatedIrp** - this union holds three members, only one (at most) of which is valid:

* **SystemBuffer** - the most often used member. This points to a system-allocated non-paged pool buffer used for Buffered I/O operations. See the section “Buffered I/O” later in this chapter for the details.

* **MasterIrp** - A pointer to a “master” IRP, if this IRP is an *associated IRP*. This idea is supported by the I/O manager, where one IRP is a “master” that may have several “associated” IRPs. Once all the associated IRPs complete, the master IRP is completed automatically. MasterIrp is valid for an associated IRP - it points to the master IRP.

* **IrpCount** - for the master IRP itself, this field indicates the number of associated IRPs associated with this master IRP.

Usage of master and associated IRPs is pretty rare. We will not be using this mechanism in this book.

- **Cancel Routine** - a pointer to a cancel routine that is invoked (if not NULL) if the driver is asked to cancel the IRP, such as with the user mode functions CancelIo and CancelIoEx. Software drivers rarely need cancellation routines, so we will not be using those in most examples.
- **MdlAddress** - points to an optional *Memory Descriptor List* (MDL). An MDL is a kernel data structure that knows how to describe a buffer in RAM. MdlAddress is used primarily with Direct I/O (see the section “Direct I/O” later in this chapter).

Every IRP is accompanied by one or more IO_STACK_LOCATIONS. Figure 7-6 shows the important fields in an IO_STACK_LOCATION.

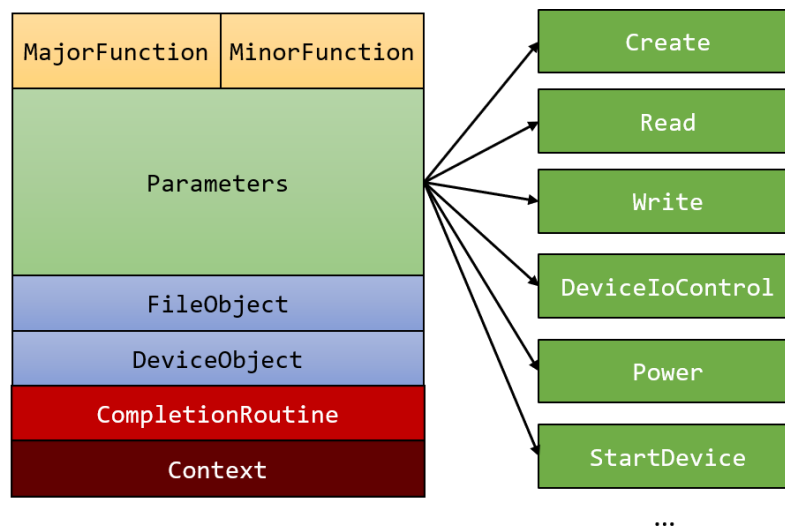


Figure 7-6: Important fields of the `IO_STACK_LOCATION` structure

Here's a rundown of the fields shown in figure 7-6:

- **MajorFunction** - this is the major function of the IRP (`IRP_MJ_CREATE`, `IRP_MJ_READ`, etc.). This field is sometimes useful if the driver points more than one major function code to the same handling routine. In that routine, the driver may want to distinguish between the major function codes using this field.
- **MinorFunction** - some IRP types have minor functions. These are `IRP_MJ_PNP`, `IRP_MJ_POWER` and `IRP_MJ_SYSTEM_CONTROL` (WMI). Typical code for these handlers has a switch statement based on the `MinorFunction`. We will not be using these types of IRPs in this book, except in the case of filter drivers for hardware-based devices, which we'll examine in some detail in chapter 13.
- **FileObject** - the `FILE_OBJECT` associated with this IRP. Not needed in most cases, but is available for dispatch routines that need it.
- **DeviceObject** - the device object associated with this IRP. Dispatch routines receive a pointer to this, so typically accessing this field is not required.
- **CompletionRoutine** - the completion routine that is set for the *previous* (upper) layer (set with `IoSetCompletionRoutine`), if any.
- **Context** - the argument to pass to the completion routine (if any).
- **Parameters** - this monster union contains multiple structures, each valid for a particular operation. For example, in a Read (`IRP_MJ_READ`)

operation, the `Parameters.Read` structure field should be used to get more information about the Read operation.

The current I/O stack location obtained with `IoGetCurrentIrpStackLocation` hosts most of the parameters of the request in the `Parameters` union. It's up to the driver to access the correct structure, as we've already seen in chapter 4 and will see again in this and subsequent chapters.

Viewing IRP Information

While debugging or analyzing kernel dumps, a couple of commands may be useful for searching or examining IRPs.

The `!irpfind` command can be used to find IRPs - either all IRPs, or IRPs that meet certain criteria. Using `!irpfind` without any arguments searches the non-paged pool(s) for all IRPs. Check out the debugger documentation on how to specify specific criteria to limit the search. Here's an example of some output when searching for all IRPs:

```
lkd> !irpfind
Unable to get offset of nt!_MI_VISIBLE_STATE.SpecialPool
Unable to get value of nt!_MI_VISIBLE_STATE.SessionSpecialPool

Scanning large pool allocation table for tag 0x3f707249 (Irp?) (ffffbf0a8761000\
0 : fffffbf0a87910000)

   Irp           [ Thread ]           irpStack: (Mj,Mn)   DevObj           [Driver\
]           MDL Process
ffffbf0aa795ca30 [ffffbf0a7fcde080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\
System\Ntfs]
ffffbf0a9a8ef010 [ffffbf0a7fcde080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\
System\Ntfs]
ffffbf0a8e68ea20 [ffffbf0a7fcde080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\
System\Ntfs]
ffffbf0a90deb710 [ffffbf0a808a1080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\
System\Ntfs]
ffffbf0a99d1da90 [0000000000000000] Irp is complete (CurrentLocation 10 > Stack\
Count 9)
ffffbf0a74cec940 [0000000000000000] Irp is complete (CurrentLocation 8 > StackC\
ount 7)
ffffbf0aa0640a20 [ffffbf0a7fcde080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\
System\Ntfs]
ffffbf0a89acf4e0 [ffffbf0a7fcde080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\
System\Ntfs]
ffffbf0a89acfa50 [ffffbf0a7fcde080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\
System\Ntfs]
(truncated)
```

Faced with a specific IRP, the command `!irp` examines the IRP, providing a nice overview of its data. As always, the `dt` command can be used with the `nt!_IRP` type to look at the entire IRP structure. Here's an example of one IRP viewed with `!irp`:

```
kd> !irp fffffbf0a8bbada20
Irp is active with 13 stacks 12 is current (= 0xffffbf0a8bbade08)
No Mdl: No System Buffer: Thread fffffbf0a7fcde080: Irp stack trace.
  cmd  flg  cl Device   File      Completion-Context
[N/A(0), N/A(0)]
  0  0 00000000 00000000 00000000-00000000

  Args: 00000000 00000000 00000000 00000000
[N/A(0), N/A(0)]
  0  0 00000000 00000000 00000000-00000000

(truncated)

  Args: 00000000 00000000 00000000 00000000
[N/A(0), N/A(0)]
  0  0 00000000 00000000 00000000-00000000

  Args: 00000000 00000000 00000000 00000000
>[IRP_MJ_DIRECTORY_CONTROL(c), N/A(2)]
  0 e1 fffffbf0a74d20050 fffffbf0a7f52f790 fffff8015c0b50a0-ffffbf0a91d99010 Su\
ccess Error Cancel pending
  \FileSystem\Ntfs
  Args: 00004000 00000051 00000000 00000000
[IRP_MJ_DIRECTORY_CONTROL(c), N/A(2)]
  0  0 fffffbf0a60e83dc0 fffffbf0a7f52f790 00000000-00000000
  \FileSystem\FltMgr
  Args: 00004000 00000051 00000000 00000000
```

The `!irp` commands lists the I/O stack locations and the information stored in them. The current I/O stack location is marked with a `>` symbol (see the `IRP_MJ_DIRECTORY_CONTROL` line above).

The details for each `IO_STACK_LOCATION` are as follows (in order):

- first line:
 - Major function code (e.g. `IRP_MJ_DEVICE_CONTROL`).
 - Minor function code.
- second line:
 - Flags (mostly unimportant)
 - Control flags
 - Device object pointer
 - File object pointer
 - Completion routine (if any)

- Completion context (for the completion routine)
- *Success, Error, Cancel* indicate the IRP completion cases where the completion routine would be invoked
- “pending” if the IRP was marked as pending (SL_PENDING_RETURNED flag is set in the Control flags)
- Driver name for that layer
- “Args” line:
 - The value of `Parameters.Others.Argument1` in the I/O stack location. Essentially the first pointer-size member in the `Parameters` union.
 - The value of `Parameters.Others.Argument2` in the I/O stack location (the second pointer-size member in the `Parameters` union)
 - Device I/O control code (if `IRP_MJ_DEVICE_CONTROL` or `IRP_MJ_INTERNAL_DEVICE_CONTROL`). It’s shown as a DML link that invokes the `!ioctldecode` command to decode the control code (more on device I/O control codes later in this chapter). For other major function codes, shows the third pointer-size member (`Parameters.Others.Argument3`)
 - The forth pointer-size member (`Parameters.Others.Argument4`)

The `!irp` command accepts an optional *details* argument. The default is zero, which provides the output described above (considered a summary). Specifying 1 provides additional information in a concrete form. Here is an example for an IRP targeted towards the console driver (you can locate those easily by looking for *cmd.exe* processes):

```
lkd> !irp fffffdb899e82a6f0 1
Irp is active with 2 stacks 1 is current (= 0xfffffdb899e82a7c0)
No Mdl: System buffer=ffffdb89c1c84ac0: Thread fffffdb89b6efa080: Irp stack tr\
ace.
Flags = 00060030
ThreadListEntry.Flink = fffffdb89b6efa530
ThreadListEntry.Blink = fffffdb89b6efa530
IoStatus.Status = 00000000
IoStatus.Information = 00000000
RequestorMode = 00000001
Cancel = 00
CancelIrql = 0
ApcEnvironment = 00
UserIosb = 73d598f420
UserEvent = 00000000
Overlay.AsynchronousParameters.UserApcRoutine = 00000000
Overlay.AsynchronousParameters.UserApcContext = 00000000
Overlay.AllocationSize = 00000000 - 00000000
CancelRoutine = fffff8026f481730
UserBuffer = 00000000
```

```

&Tail.Overlay.DeviceQueueEntry = fffffdb899e82a768
Tail.Overlay.Thread = fffffdb89b6efa080
Tail.Overlay.AuxiliaryBuffer = 00000000
Tail.Overlay.ListEntry.Flink = fffff8006d16437b8
Tail.Overlay.ListEntry.Blink = fffff8006d16437b8
Tail.Overlay.CurrentStackLocation = fffffdb899e82a7c0
Tail.Overlay.OriginalFileObject = fffffdb89c1c0a240
Tail.Apc = 8b8b7240
Tail.CompletionKey = 15f8b8b7240
cmd flg cl Device File Completion-Context
>[N/A(f), N/A(7)]
0 1 00000000 00000000 00000000-00000000 pending

Args: fffff8006d1643790 15f8d92c340 0xa0e666b0 fffffdb899e7a53c0
[IRP_MJ_DEVICE_CONTROL(e), N/A(0)]
5 0 fffffdb89846f9e10 fffffdb89c1c0a240 00000000-00000000
\Driver\condrv
Args: 00000000 00000060 0x500016 00000000

```

Additionally, specifying detail value of 4 shows *Driver Verifier* information related to the IRP (if the driver handling this IRP is under the verifier's microscope). *Driver Verifier* will be discussed in chapter 13.

Dispatch Routines

In chapter 4, we have seen an important aspect of `DriverEntry` - setting up dispatch routines. These are the functions connected with major function codes. The `MajorFunction` field in `DRIVER_OBJECT` is the array of function pointers index by the major function code.

All dispatch routines have the same prototype, repeated here for convenience using the `DRIVER_DISPATCH` typedef from the WDK (somewhat simplified for clarity):

```

typedef NTSTATUS DRIVER_DISPATCH (
    _In_      PDEVICE_OBJECT DeviceObject,
    _Inout_   PIRP Irp);

```

The relevant dispatch routine (based on the major function code) is the first routine in a driver that sees the request. Normally, it's called in the requesting thread context, i.e. the thread that called the relevant API (e.g. `ReadFile`) in IRQL `PASSIVE_LEVEL` (0). However, it's possible that a filter driver sitting on top of this device sent the request down in a different context - it may be some other thread unrelated to the original requestor and even in higher IRQL, such as `DISPATCH_LEVEL` (2). Robust drivers need to be ready to deal with this kind

of situation, even though for software drivers this “inconvenient” context is rare. We’ll discuss the way to properly deal with this situation in the section “Accessing User Buffers”, later in this chapter.

The first thing a typical dispatch routine does is check for errors. For example, read and write operations contain buffers - do these buffers have appropriate size? For `DeviceIoControl`, there is a control code in addition to potentially two buffers. The driver needs to make sure the control code is something it recognizes. If any error is identified, the IRP is typically completed immediately with an appropriate status.

If all checks turn up ok, then the driver can deal with performing the requested operation.

Here is the list of the most common dispatch routines for a software driver:

- `IRP_MJ_CREATE` - corresponds to a `CreateFile` call from user mode or `ZwCreateFile` in kernel mode. This major function is essentially mandatory, otherwise no client will be able to open a handle to a device controlled by this driver. Most drivers just complete the IRP with a success status.
- `IRP_MJ_CLOSE` - the opposite of `IRP_MJ_CREATE`. Called by `CloseHandle` from user mode or `ZwClose` from kernel mode when the last handle to the file object is about to be closed. Most drivers just complete the request successfully, but if something meaningful was done in `IRP_MJ_CREATE`, this is where it should be undone.
- `IRP_MJ_READ` - corresponds to a read operation, typically invoked from user mode by `ReadFile` or kernel mode with `ZwReadFile`.
- `IRP_MJ_WRITE` - corresponds to a write operation, typically invoked from user mode by `WriteFile` or kernel mode with `ZwWriteFile`.
- `IRP_MJ_DEVICE_CONTROL` - corresponds to the `DeviceIoControl` call from user mode or `ZwDeviceIoControlFile` from kernel mode (there are other APIs in the kernel that can generate `IRP_MJ_DEVICE_CONTROL` IRPs).
- `IRP_MJ_INTERNAL_DEVICE_CONTROL` - similar to `IRP_MJ_DEVICE_CONTROL`, but only available to kernel callers.

Completing a Request

Once a driver decides to handle an IRP (meaning it's not passing down to another driver), it must eventually complete it. Otherwise, we have a leak on our hands - the requesting thread cannot really terminate and by extension, its containing process will linger on as well, resulting in a "zombie process".

Completing a request means calling `IoCompleteRequest` after setting the request status and extra information. If the completion is done in the dispatch routine itself (a common case for software drivers), the routine must return the same status that was placed in the IRP.

The following code snippet shows how to complete a request in a dispatch routine:

```
NTSTATUS MyDispatchRoutine(PDEVICE_OBJECT, PIRP Irp) {
    //...
    Irp->IoStatus.Status = STATUS_XXX;
    Irp->IoStatus.Information = bytes;    // depends on request type
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_XXX;
}
```



Since the dispatch routine must return the same status as was placed in the IRP, it's tempting to write the last statement like so: `return Irp->IoStatus.Status;` This, however, will likely result in a system crash. Can you guess why?

After the IRP is completed, touching any of its members is a bad idea. The IRP has probably already been freed and you're touching deallocated memory. It can actually be worse, since another IRP may have been allocated in its place (this is common), and so the code may return the status of some random IRP.

The `Information` field should be zero in case of an error (a failure status). Its exact meaning for a successful operation depends on the type of IRP.

The `IoCompleteRequest` API accepts two arguments: the IRP itself and an optional value to temporarily boost the original thread's priority (the thread that initiated the request in the first place). In most cases, for software drivers, the thread in question is the executing thread, so a thread boost is inappropriate. The value `IO_NO_INCREMENT` is defined as zero, so no increment in the above code snippet.

However, the driver may choose to give the thread a boost, regardless of whether it's the calling thread or not. In this case, the thread's priority jumps with the given boost, and then it's allowed to execute one quantum with that new priority before the priority decreases by one, it can then get another quantum with the reduced priority, and so on, until its priority returns to its original level. Figure 7-7 illustrates this scenario.

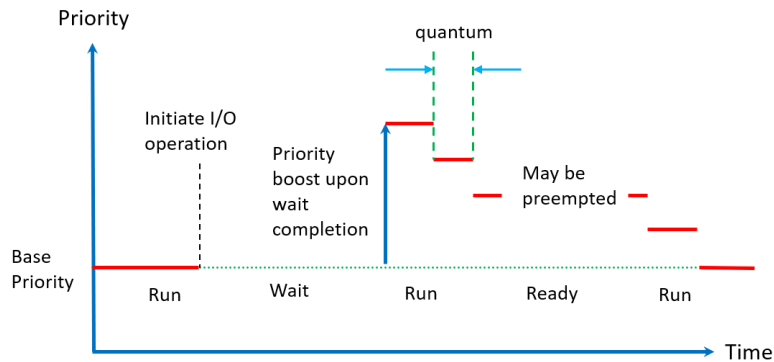


Figure 7-7: Thread priority boost and decay



The thread's priority after the boost can never go above 15. If it's supposed to, it will be 15. If the original thread's priority is above 15 already, boosting has no effect.

Accessing User Buffers

A given dispatch routine is the first to see the IRP. Some dispatch routines, mainly `IRP_MJ_READ`, `IRP_MJ_WRITE` and `IRP_MJ_DEVICE_CONTROL` accept buffers provided by a client - in most cases from user mode. Typically, a dispatch routine is called in IRQL 0 and in the requesting thread context, which means the buffers pointers provided by user mode are trivially accessible: the IRQL is 0, so page faults are handled normally, and the thread is the requestor, so the pointers are valid in this process context.

However, there could be issues. As we've seen in chapter 6, even in this convenient context (requesting thread and IRQL 0), it's possible for another thread in the client's process to free the passed-in buffer(s), before the driver gets a chance to examine them, and so cause an access violation. The solution

we've used in chapter 6 is to use a `__try / __except` block to handle any access violation by returning failure to the client.

In some cases, even that is not enough. For example, if we have some code running at IRQL 2 (such as a DPC running as a result of timer expiration), we cannot safely access the user's buffers in this context. In general, there are two potential issues here:

- IRQL of the calling CPU is 2 (or higher), meaning no page fault handling can occur.
- The thread calling the driver may be some arbitrary thread, and not the original requestor. This means that the buffer pointer(s) provided are meaningless, since the wrong process address space is accessible.

Using exception handling in such a case will not work as expected, because we'll be accessing some memory location that is essentially invalid in this random process context. Even if the access succeeds (because that memory happens to be allocated in this random process and is resident in RAM), you'll be accessing random memory, and certainly not the original buffer provided to the client.

All this means that there must be some good way to access the original user's buffer in such an inconvenient context. In fact, there are two such ways provided by the I/O manager for this purpose, called *Buffered I/O* and *Direct I/O*. In the next two sections, we'll see what each of these schemes mean and how to use them.



Some data structures are always safe to access, since they are allocated from non-paged pool (and are in system space). Common examples are device objects (created with `IoCreateDevice`) and IRPs.

Buffered I/O

Buffered I/O is the simplest of the two ways. To get support for Buffered I/O for Read and Write operations, a flag must be set on the device object like so:

```
DeviceObject->Flags |= DO_BUFFERED_IO; // DO = Device Object
```

DeviceObject is the allocated pointer from a previous call to IoCreateDevice (or IoCreateDeviceSecure).

For IRP_MJ_DEVICE_CONTROL buffers, see the section “User Buffers for IRP_MJ_DEVICE_CONTROL” later in this chapter.

Here are the steps taken by the I/O Manager and the driver when a read or write request arrives:

1. The I/O Manager allocates a buffer from non-paged pool with the same size as the user’s buffer. It stores the pointer to this new buffer in the AssociatedIrp->SystemBuffer member of the IRP. (The buffer size can be found in the current I/O stack location’s Parameters.Read.Length or Parameters.Write.Length.)
2. For a write request, the I/O Manager copies the user’s buffer to the system buffer.
3. Only now the driver’s dispatch routine is called. The driver can use the system buffer pointer directly without any checks, because the buffer is in system space (its address is absolute - the same from any process context), and in any IRQL, because the buffer is allocated from non-paged pool, so it cannot be paged out.
4. Once the driver completes the IRP (IoCompleteRequest), the I/O manager (for read requests) copies the system buffer back to the user’s buffer (the size of the copy is determined by the IoStatus.Information field in the IRP set by the driver).
5. Finally, the I/O Manager frees the system buffer.



You may be wondering how does the I/O Manager copy back the system buffer to the original user’s buffer from IoCompleteRequest. This function can be called from any thread, in IRQL ≤ 2 . The way it’s done is by queuing a *special kernel APC* to the thread that requested the operation. Once this thread is scheduled for execution, the first thing it does is run this APC which performs the actual copying. The requesting thread is obviously in the correct process context, and the IRQL is 1, so page faults can be handled normally.

Figures 7-8a to 7-8e illustrate the steps taken with Buffered I/O.

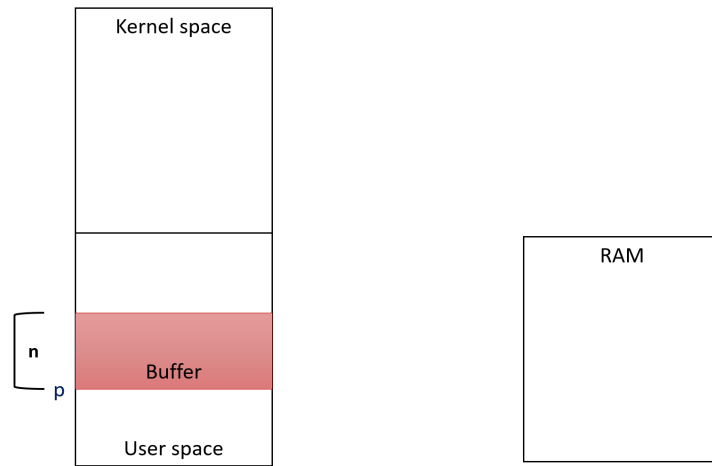


Figure 7-8a: Buffered I/O: initial state

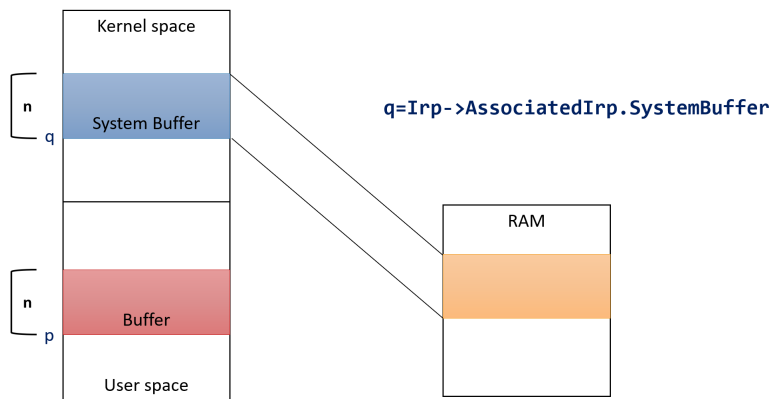


Figure 7-8b: Buffered I/O: system buffer allocated

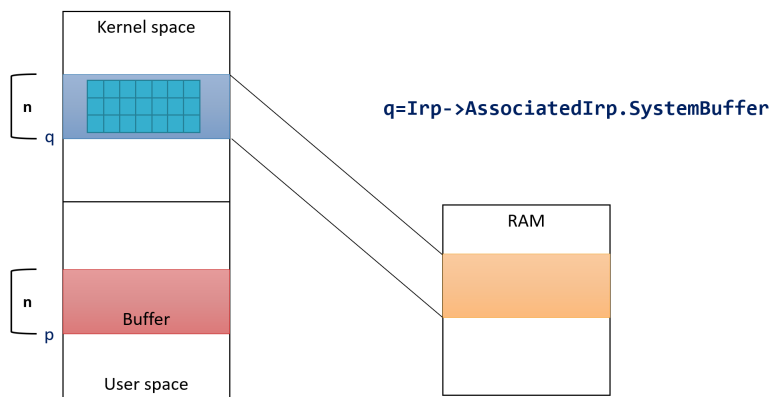


Figure 7-8c: Buffered I/O: driver accesses system buffer

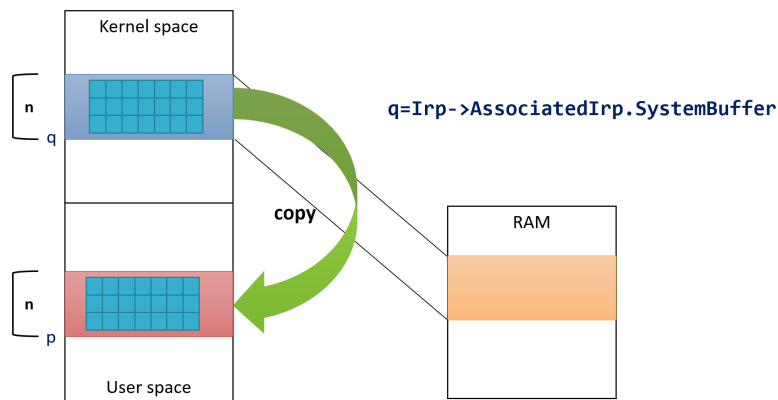


Figure 7-8d: Buffered I/O: on IRP completion, I/O manager copies buffer back (for read)

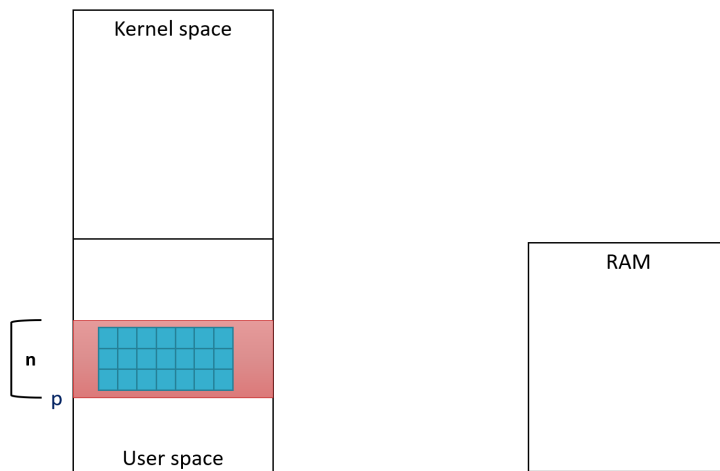


Figure 7-8e: Buffered I/O: final state - I/O manager frees system buffer

Buffered I/O has the following characteristics:

- Easy to use - just specify the flag in the device object, and everything else is taken care of by the I/O Manager.
- It always involves a copy - which means it's best used for small buffers (typically up to one page). Large buffers may be expensive to copy. In this case, the other option, Direct I/O, should be used instead.

Direct I/O

The purpose of Direct I/O is to allow access to a user's buffer in any IRQL and any thread but without any copying going around.

For read and write requests, selecting Direct I/O is done with a different flag of the device object:

```
DeviceObject->Flags |= DO_DIRECT_IO;
```

As with Buffered I/O, this selection only affects read and write requests. For `DeviceIoControl` see the next section.

Here are the steps involved in handling Direct I/O:

1. The I/O Manager first makes sure the user's buffer is valid and then pages it into physical memory (if it wasn't already there).
2. It then locks the buffer in memory, so it cannot be paged out until further notice. This solves one of the issues with buffer access - page faults cannot happen, so accessing the buffer in any IRQL is safe.
3. The I/O Manager builds a *Memory Descriptor List* (MDL), a data structure that describes a buffer in physical memory. The address of this data structure is stored in the `MdlAddress` field of the IRP.
4. At this point, the driver gets the call to its dispatch routine. The user's buffer, although locked in RAM, cannot be accessed from an arbitrary thread just yet. When the driver requires access to the buffer, it must call a function that maps the same user buffer to a system address, which by definition is valid in any process context. So essentially, we get two mappings to the same memory buffer. One is from the original address (valid only in the context of the requestor process) and the other in system space, which is always valid. The API to call is `MmGetSystemAddressForMdlSafe`, passing the MDL built by the I/O Manager. The return value is the system address.
5. Once the driver completes the request, the I/O Manager removes the second mapping (to system space), frees the MDL, and unlocks the user's buffer, so it can be paged normally just like any other user-mode memory.

The MDL is in actually a list of MDL structures, each one describing a piece of the buffer that is contiguous in physical memory. Remember, that a buffer that is contiguous in virtual memory is not necessary contiguous in physical memory (the smallest piece is a page size). In most case, we don't need to care about this detail. One case where this matters is in *Direct*

Memory Access (DMA) operations. Fortunately, this is in the realm of hardware-based drivers.

Figures 7-9a to 7-9f illustrate the steps taken with Direct I/O.

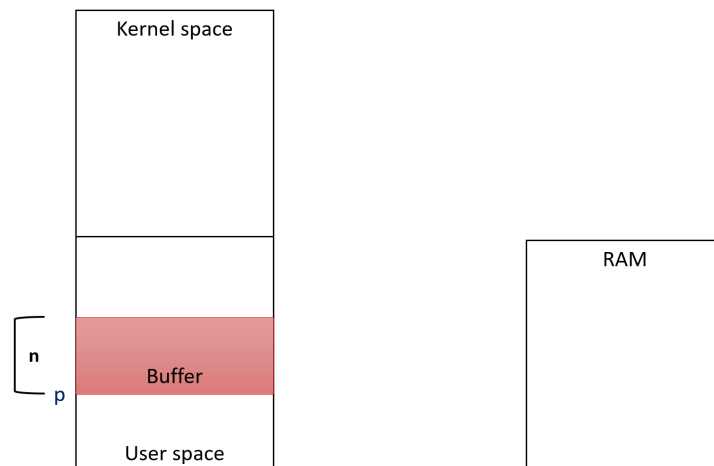


Figure 7-9a: Direct I/O: initial state

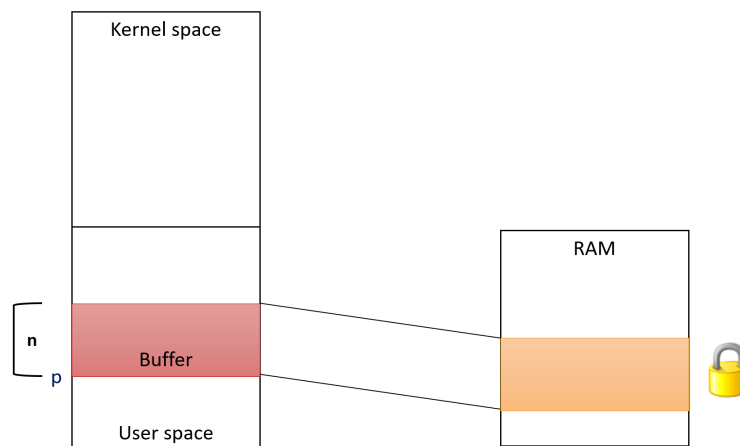


Figure 7-9b: Direct I/O: I/O manager faults buffer's pages to RAM and locks them

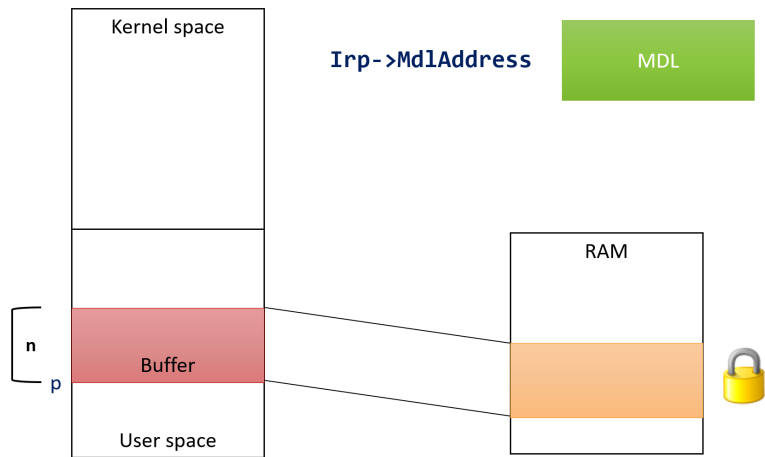


Figure 7-9c: Direct I/O: the MDL describing the buffer is stored in the IRP

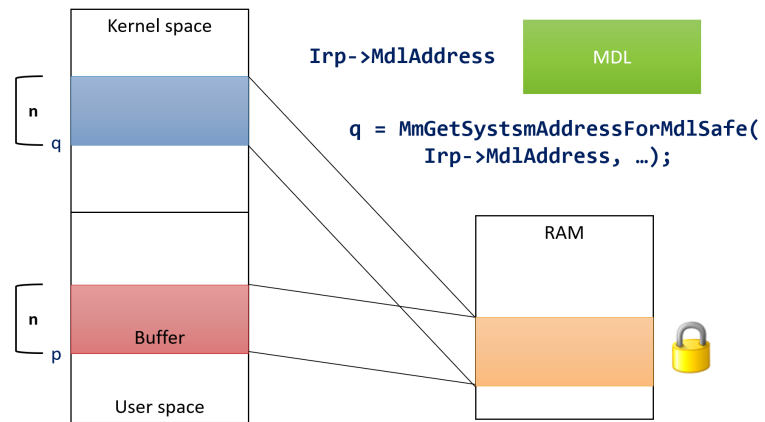


Figure 7-9d: Direct I/O: the driver double-maps the buffer to a system address

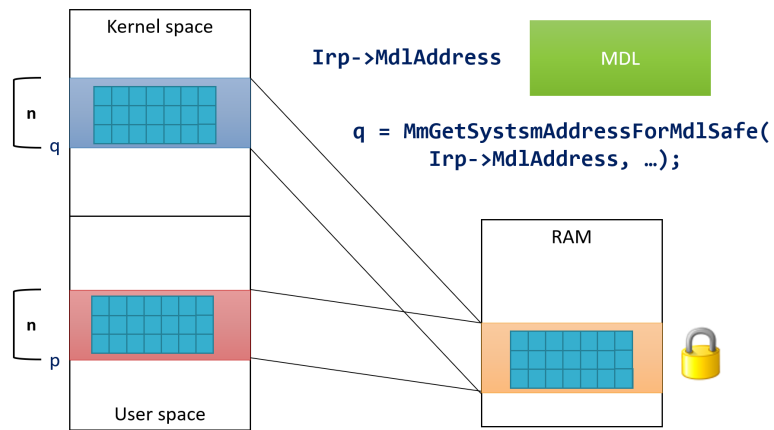


Figure 7-9e: Direct I/O: the driver accesses the buffer using the system address

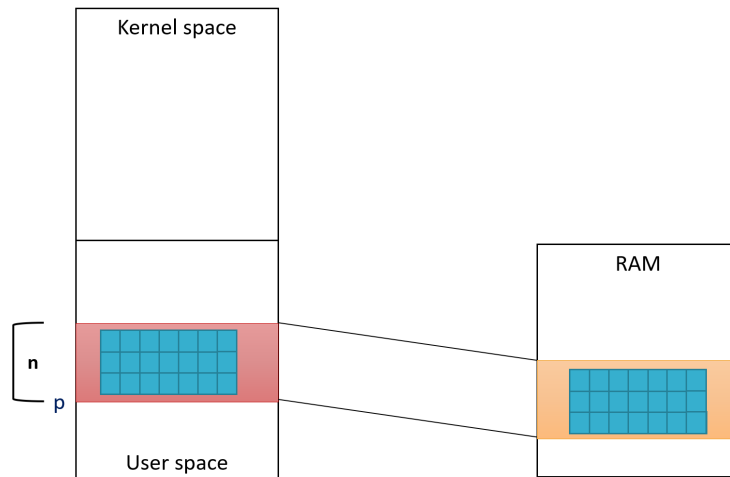


Figure 7-9f: Direct I/O: when the IRP is completed, the I/O manager frees the mapping, the MDL and unlocks the buffer

Notice there is no copying at all. The driver just reads/writes to the user's buffer directly, using the system address.



Locking the user's buffer is done with the `MmProbeAndLockPages` API, fully documented in the WDK. Unlocking is done with `MmUnlockPages`, also documented. This means a driver can use these routines outside the narrow context of Direct I/O.



Calling `MmGetSystemAddressForMdlSafe` can be done multiple times. The MDL stores a flag indicating whether the system mapping has already been done. If so, it just returns the existing pointer.

Here is the prototype of `MmGetSystemAddressForMdlSafe`:

```
PVOID MmGetSystemAddressForMdlSafe (  
    _Inout_ PMDL Mdl,  
    _In_     ULONG Priority);
```

The function is implemented inline within the *wdm.h* header by calling the more generic `MmMapLockedPagesSpecifyCache` function:

```
PVOID MmGetSystemAddressForMdlSafe(PMDL Mdl, ULONG Priority) {  
    if (Mdl->MdlFlags & (MDL_MAPPED_TO_SYSTEM_VA|MDL_SOURCE_IS_NONPAGED_POOL)) {  
        return Mdl->MappedSystemVa;  
    } else {  
        return MmMapLockedPagesSpecifyCache(Mdl, KernelMode, MmCached,  
                                             NULL, FALSE, Priority);  
    }  
}
```

`MmGetSystemAddressForMdlSafe` accepts the MDL and a page priority (`MM_PAGE_PRIORITY` enumeration). Most drivers specify `NormalPagePriority`, but there is also `LowPagePriority` and `HighPagePriority`. This priority gives a hint to the system of the importance of the mapping in low memory conditions. Check the WDK documentation for more information.

If `MmGetSystemAddressForMdlSafe` fails, it returns `NULL`. This means the system is out of system page tables or very low on system page tables (depends on the priority argument above). This should be a rare occurrence, but still can happen in low memory conditions. A driver must check for this; if `NULL` is

returned, the driver should complete the IRP with the status `STATUS_INSUFFICIENT_RESOURCES`.



There is a similar function, called `MmGetSystemAddressForMdl`, which if it fails, crashes the system. Do not use this function.

You may be wondering why doesn't the I/O manager call `MmGetSystemAddressForMdlSafe` automatically, which would be simple enough to do. This is an optimization, where the driver may not need to call this function at all if there is any error in the request, so that the mapping doesn't have to occur at all.

Drivers that don't set either of the flags `DO_BUFFERED_IO` nor `DO_DIRECT_IO` in the device object flags implicitly use *Neither I/O*, which simply means the driver doesn't get any special help from the I/O manager, and it's up to the driver to deal with the user's buffer.

User Buffers for `IRP_MJ_DEVICE_CONTROL`

The last two sections discussed Buffered I/O and Direct I/O as they pertain to read and write requests. For `IRP_MJ_DEVICE_CONTROL` (and `IRP_MJ_INTERNAL_DEVICE_CONTROL`), the buffering access method is supplied on a control code basis. Here is the prototype of the user-mode API `DeviceIoControl` (it's similar with the kernel function `ZwDeviceIoControlFile`):

```
BOOL DeviceIoControl(
    HANDLE hDevice,           // handle to device or file
    DWORD dwIoControlCode,    // IOCTL code (see <winioctl.h>)
    PVOID lpInBuffer,         // input buffer
    DWORD nInBufferSize,     // size of input buffer
    PVOID lpOutBuffer,        // output buffer
    DWORD nOutBufferSize,     // size of output buffer
    PDWORD lpdwBytesReturned, // # of bytes actually returned
    LPOVERLAPPED lpOverlapped); // for async. operation
```

There are three important parameters here: the I/O control code, and optional two buffers designated “input” and “output”. As it turns out, the way these buffers are accessed depends on the control code, which is very convenient,

because different requests may have different requirements related to accessing the user's buffer(s).

The control code defined by a driver must be built with the CTL_CODE macro, defined in the WDK and user-mode headers, defined like so:

```
#define CTL_CODE( DeviceType, Function, Method, Access ) ( \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
```

The first parameter, DeviceType can be one of a set of constants defined by Microsoft for various known device types (such as FILE_DEVICE_DISK and FILE_DEVICE_KEYBOARD). For custom devices (like the ones we are writing), it can be any value, but the documentation states that the minimum value for custom codes should be 0x8000.

The second parameter, Function is a running index, that must be different between multiple control codes defined by the same driver. If all other components of the macro are same (possible), at least the Function would be a differentiating factor. Similarly to device type, the official documentation states that custom devices should use values starting from 0x800.

The third parameter (*Method*) is the key to selecting the buffering method for accessing the input and output buffers provided with DeviceIoControl. Here are the options:

- METHOD_NEITHER - this value means no help is required of the I/O manager, so the driver is left dealing with the buffers on its own. This could be useful, for instance, if the particular code does not require any buffer - the control code itself is all the information needed - it's best to let the I/O manager know that it does not need to do any additional work.
 - In this case, the pointer to the user's input buffer is store in the current I/O stack location's Parameters.DeviceIoControl.Type3InputBuffer field, and the output buffer is stored in the IRP's UserBuffer field.
- METHOD_BUFFERED - this value indicates Buffered I/O for both the input and output buffer. When the request starts, the I/O manager allocates the system buffer from non-paged pool with the size that is the maximum of the lengths of the input and output buffers. It then copies the input buffer to the system buffer. Only now the IRP_MJ_DEVICE_CONTROL dispatch routine is invoked. When the request completes, the I/O manager copies

the number of bytes indicated with the `IoStatus.Information` field in the IRP to the user's output buffer.

- The system buffer pointer is at the usual location:
`AssociatedIrp.SystemBuffer` inside the IRP structure.
- `METHOD_IN_DIRECT` and `METHOD_OUT_DIRECT` - contrary to intuition, both of these values mean the same thing as far as buffering methods are concerned: the input buffer uses Buffered I/O and the output buffer uses Direct I/O. The only difference between these two values is whether the output buffer can be read (`METHOD_IN_DIRECT`) or written (`METHOD_OUT_DIRECT`).



The last bullet indicates that the output buffer can also be treated as input by using `METHOD_IN_DIRECT`.

Table 7-1 summarizes these buffering methods.

Table 7-1: Buffering method based on control code Method parameter		
Method	Input buffer	Output buffer
<code>METHOD_NEITHER</code>	Neither	Neither
<code>METHOD_BUFFERED</code>	Buffered	Buffered
<code>METHOD_IN_DIRECT</code>	Buffered	Direct
<code>METHOD_OUT_DIRECT</code>	Buffered	Direct

Finally, the `Access` parameter to the macro indicates the direction of data flow. `FILE_WRITE_ACCESS` means from the client to the driver, `FILE_READ_ACCESS` means the opposite, and `FILE_ANY_ACCESS` means bi-directional access (the input and output buffers are used). You should always use `FILE_ANY_ACCESS`. Besides simplifying the control code building, you guarantee that if later on, once the driver is already deployed, you may want to use the other buffer, you wouldn't need to change the `Access` parameter, and so not disturb existing clients that would not know about the control code change.



If a control code is built with `METHOD_NEITHER`, the I/O manager does nothing to help with accessing the buffer(s). The values for the input and output buffer pointers provided by the client are copied as-is to the IRP. No checking is done by the I/O manager to make sure these pointers point to valid memory. A driver should not use these pointers as memory pointers, but they can be used as two arbitrary values propagating to the driver that may mean something.

Putting it All Together: The *Zero* Driver

In this section, we'll use what we've learned in this (and earlier) chapter and build a driver and a client application. The driver is named *Zero* and has the following characteristics:

- For read requests, it zeros out the provided buffer.
- For write requests, it just consumes the provided buffer, similar to a classic *null* device.

The driver will use Direct I/O so as not to incur the overhead of copies, as the buffers provided by the client can potentially be very large.

We'll start the project by creating an “Empty WDM Project” in Visual Studio and name it *Zero*. Then we'll delete the created INF file, resulting in an empty project, just like in previous examples.

Using a Precompiled Header

One technique that we can use that is not specific to driver development, but is generally useful, is using a *precompiled header*. Precompiled headers is a Visual Studio feature that helps with faster compilation times. The precompiled header is a header file that has `#include` statements for headers that rarely change, such as `ntddk.h` for drivers. The precompiled header is compiled once, stored in an internal binary format, and used in subsequent compilations, which become considerably faster.



Many user mode projects created by Visual Studio already use precompiled headers. Kernel-mode projects provided by the WDK templates currently don't use precompiled headers. Since we're starting with an empty project, we have to set up precompiled headers manually anyway.

Follow these steps to create and use a precompiled header:

- Add a new header file to the project and call it *pch.h*. This file will serve as the precompiled header. Add all rarely-changing `#include`s here:

```
// pch.h  
  
#pragma once  
  
#include <ntddk.h>
```

- Add a source file named *pch.cpp* and put a single `#include` in it: the precompiled header itself:

```
#include "pch.h"
```

- Now comes the tricky part. Letting the compiler know that *pch.h* is the precompiled header and *pch.cpp* is the one creating it. Open project properties, select *All Configurations* and *All Platforms* so you won't need to configure every configuration/platform separately, navigate to *C/C++ / Precompiled Headers* and set *Precompiled Header* to **Use** and the file name to "pch.h" (see figure 7-10). Click OK and to close the dialog box.

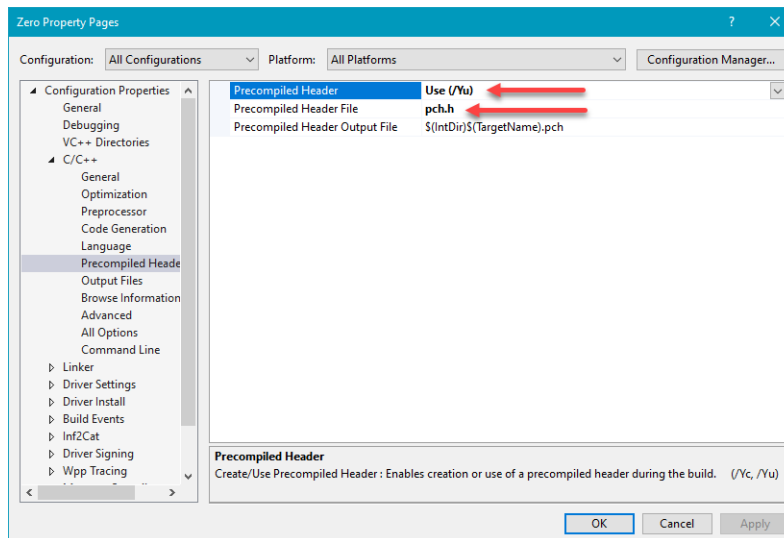


Figure 7-10: Setting precompiled header for the project

- The *pch.cpp* file should be set as the creator of the precompiled header. Right click this file in *Solution Explorer*, and select *Properties*. Navigate to *C/C++ / Precompiled Headers* and set *Precompiled Header* to **Create** (see figure 7-11). Click OK to accept the setting.

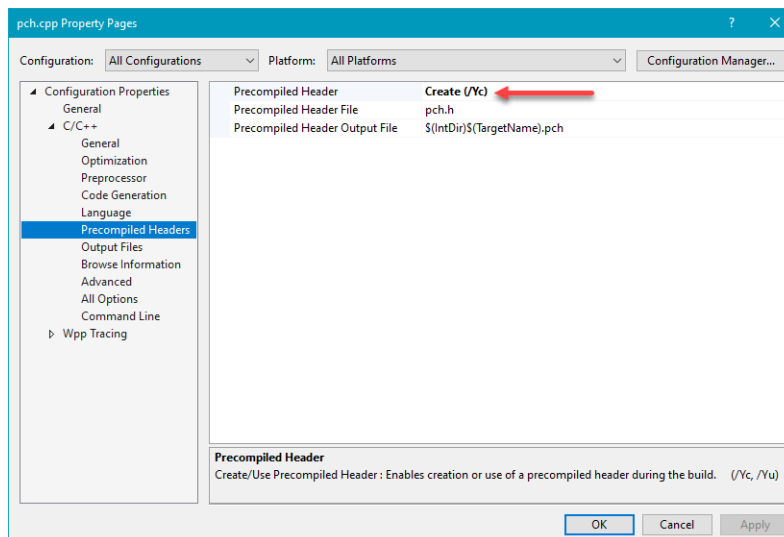


Figure 7-10: Setting precompiled header for pch.cpp

From this point on, every C/CPP file in the project must `#include "pch.h"` as the first thing in the file. Without this include, the project will not compile.



Make sure there is nothing before this `#include "pch.h"` in a source file. Anything before this line does not get compiled at all!

The DriverEntry Routine

The `DriverEntry` routine for the *Zero* driver is very similar to the one we created for the driver in chapter 4. However, in chapter 4's driver the code in *DriverEntry* had to undo any operation that was already done in case of a later error. We had just two operations that could be undone: creation of the device object and creation of the symbolic link. The *Zero* driver is similar, but we'll create a more robust and less error-prone code to handle errors during initialization. Let's start with the basics of setting up an unload routine and the dispatch routines:

```
#define DRIVER_PREFIX "Zero: "

// DriverEntry

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = ZeroUnload;
    DriverObject->MajorFunction[IRP_MJ_CREATE] =
        DriverObject->MajorFunction[IRP_MJ_CLOSE] = ZeroCreateClose;
    DriverObject->MajorFunction[IRP_MJ_READ] = ZeroRead;
    DriverObject->MajorFunction[IRP_MJ_WRITE] = ZeroWrite;
```

Now we need to create the device object and symbolic link and handle errors in a more general and robust way. The trick we'll use is a `do / while(false)` block, which is not really a loop, but it allows getting out of the block with a simple `break` statement in case something goes wrong:

```
UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\Zero");
UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\Zero");
PDEVICE_OBJECT DeviceObject = nullptr;
auto status = STATUS_SUCCESS;

do {
    status = IoCreateDevice(DriverObject, 0, &devName, FILE_DEVICE_UNKNOWN,
        0, FALSE, &DeviceObject);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "failed to create device (0x%08X)\\n", status));
```

```

        break;
    }

    // set up Direct I/O
    DeviceObject->Flags |= DO_DIRECT_IO;

    status = IoCreateSymbolicLink(&symLink, &devName);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "failed to create symbolic link (0x%08X)\n",
            status));
        break;
    }
} while (false);

if (!NT_SUCCESS(status)) {
    if (DeviceObject)
        IoDeleteDevice(DeviceObject);
}
return status;

```

The pattern is simple: if an error occurs in any call, just break out of the “loop”. Outside the loop, check the status, and if it’s a failure, undo any operations done so far. With this scheme in hand, it’s easy to add more initializations (which we’ll need in more complex drivers), while keeping the cleanup code localized and appearing just once.

It’s possible to use goto statements instead of the do / while(false) approach, but as the great Dijkstra wrote, “goto considered harmful”, so I tend to avoid it if I can.

Notice we’re also initializing the device to use Direct I/O for our read and write operations.

The Create and Close Dispatch Routines

Before we get to the actual implementation of IRP_MJ_CREATE and IRP_MJ_CLOSE (pointing to the same function), let’s create a helper function that simplifies completing an IRP with a given status and information:

```

NTSTATUS CompleteIrp(PIRP Irp,
    NTSTATUS status = STATUS_SUCCESS,
    ULONG_PTR info = 0) {
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = info;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

```

Notice the default values for the status and information. The Create/Close dispatch routine implementation becomes almost trivial:

```
NTSTATUS ZeroCreateClose(PDEVICE_OBJECT, PIRP Irp) {  
    return CompleteIrp(Irp);  
}
```

The Read Dispatch Routine

The Read routine is the most interesting. First we need to check the length of the buffer to make sure it's not zero. If it is, just complete the IRP with a failure status:

```
NTSTATUS ZeroRead(PDEVICE_OBJECT, PIRP Irp) {  
    auto stack = IoGetCurrentIrpStackLocation(Irp);  
    auto len = stack->Parameters.Read.Length;  
    if (len == 0)  
        return CompleteIrp(Irp, STATUS_INVALID_BUFFER_SIZE);  
}
```

Note that the length of the user's buffer is provided through the `Parameters.Read` member inside the current I/O stack location.

We have configured Direct I/O, so we need to map the locked buffer to system space using `MmGetSystemAddressForMdlSafe`:

```
NT_ASSERT(Irp->MdlAddress);    // make sure Direct I/O flag was set  
auto buffer = MmGetSystemAddressForMdlSafe(Irp->MdlAddress, NormalPagePriority);  
if (!buffer)  
    return CompleteIrp(Irp, STATUS_INSUFFICIENT_RESOURCES);
```

The functionality we need to implement is to zero out the given buffer. We can use a simple `memset` call to fill the buffer with zeros and then complete the request:

```
memset(buffer, 0, len);  
  
return CompleteIrp(Irp, STATUS_SUCCESS, len);  
}
```

If you prefer a more “fancy” function to zero out memory, call `RtlZeroMemory`. It's a macro, defined in terms of `memset`.

It's important to set the Information field to the length of the buffer. This indicates to the client the number of bytes transferred in the operation (returned in the second to last parameter to ReadFile). This is all we need for the read operation.

The Write Dispatch Routine

The write dispatch routine is even simpler. All it needs to do is complete the request with the buffer length provided by the client (essentially swallowing the buffer):

```
NTSTATUS ZeroWrite(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto len = stack->Parameters.Write.Length;

    return CompleteIrp(Irp, STATUS_SUCCESS, len);
}
```

Note that we don't even bother calling MmGetSystemAddressForMdlSafe, as we don't need to access the actual buffer. This is also the reason this call is not made beforehand by the I/O manager: the driver may not even need it, or perhaps need it in certain conditions only; so the I/O manager prepares everything (the MDL) and lets the driver decide when and if to map the buffer.

Test Application

We'll add a new console application project to the solution to test the read and write operations.

Here is some simple code to test these operations:

```
int Error(const char* msg) {
    printf("%s: error=%u\n", msg, ::GetLastError());
    return 1;
}

int main() {
    HANDLE hDevice = CreateFile(L"\\\\.\\Zero", GENERIC_READ | GENERIC_WRITE,
        0, nullptr, OPEN_EXISTING, 0, nullptr);
    if (hDevice == INVALID_HANDLE_VALUE) {
        return Error("Failed to open device");
    }

    // test read
    BYTE buffer[64];

    // store some non-zero data
```



```

    for (int i = 0; i < sizeof(buffer); ++i)
        buffer[i] = i + 1;

    DWORD bytes;
    BOOL ok = ReadFile(hDevice, buffer, sizeof(buffer), &bytes, nullptr);
    if (!ok)
        return Error("failed to read");
    if (bytes != sizeof(buffer))
        printf("Wrong number of bytes\n");

    // check that all bytes are zero
    for (auto n : buffer)
        if (n != 0) {
            printf("Wrong data!\n");
            break;
        }

    // test write
    BYTE buffer2[1024]; // contains junk
    ok = WriteFile(hDevice, buffer2, sizeof(buffer2), &bytes, nullptr);
    if (!ok)
        return Error("failed to write");
    if (bytes != sizeof(buffer2))
        printf("Wrong byte count\n");
    CloseHandle(hDevice);
}

```

Read/Write Statistics

Let's add some more functionality to the *Zero* driver. We may want to count the total bytes read/written throughout the lifetime of the driver. A user-mode client should be able to read these statistics, and perhaps even zero them out.

We'll start by defining two global variables to keep track of the total number of bytes read/written (in *Zero.cpp*):

```

long long g_TotalRead;
long long g_TotalWritten;

```

You could certainly put these in a structure for easier maintenance and extension. The `long long` C++ type is a signed 64-bit value. You can add unsigned if you wish, or use a typedef such as `LONG64` or `ULONG64`, which would mean the same thing. Since these are global variables, they are zeroed out by default.

We'll create a new file that contains information common to user-mode clients and the driver called *ZeroCommon.h*. here is where we define the control codes we support, as well as data structures to be shared with user-mode.

First, we'll add two control codes: one for getting the stats and another for clearing them:

```
#define DEVICE_ZERO 0x8022

#define IOCTL_ZERO_GET_STATS \
    CTL_CODE(DEVICE_ZERO, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_ZERO_CLEAR_STATS \
    CTL_CODE(DEVICE_ZERO, 0x801, METHOD_NEITHER, FILE_ANY_ACCESS)
```

The `DEVICE_ZERO` is defined as some number from `0x8000` as the documentation recommends. The *function* number starts with `0x800` and incremented with each control code. `METHOD_BUFFERED` is used for getting the stats, as the size of the returned data is small (2 x 8 bytes). Clearing the stats requires no buffers, so `METHOD_NEITHER` is selected.

Next, we'll add a structure that can be used by clients (and the driver) for storing the stats:

```
struct ZeroStats {
    long long TotalRead;
    long long TotalWritten;
};
```

In `DriverEntry`, we add a dispatch routine for `IRP_MJ_DEVICE_CONTROL` like so:

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = ZeroDeviceControl;
```

All the work is done in `ZeroDeviceControl`. First, some initialization:

```
NTSTATUS ZeroDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto& dic = irpSp->Parameters.DeviceIoControl;
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG_PTR len = 0;
```

The details for `IRP_MJ_DEVICE_CONTROL` are located in the current I/O stack location in the `Parameters.DeviceIoControl` structure. The status is initialized to an error in case the control code provided is unsupported. `len` keeps track of the number of valid bytes returned in the output buffer.

Implementing the `IOCTL_ZERO_GET_STATS` is done in the usual way. First, check for errors. If all goes well, the stats are written to the output buffer:

```

switch (dic.IoControlCode) {
    case IOCTL_ZERO_GET_STATS:
    { // artificial scope so the compiler doesn't complain
      // about defining variables skipped by a case
      if (dic.OutputBufferLength < sizeof(ZeroStats)) {
          status = STATUS_BUFFER_TOO_SMALL;
          break;
      }

      auto stats = (ZeroStats*)Irp->AssociatedIrp.SystemBuffer;
      if (stats == nullptr) {
          status = STATUS_INVALID_PARAMETER;
          break;
      }
      //
      // fill in the output buffer
      //
      stats->TotalRead = g_TotalRead;
      stats->TotalWritten = g_TotalWritten;
      len = sizeof(ZeroStats);
      break;
    }
}

```

Once out of the switch, the IRP would be completed. Here is the stats clearing Ioctl handling:

```

    case IOCTL_ZERO_CLEAR_STATS:
        g_TotalRead = g_TotalWritten = 0;
        break;
}

```

All that's left to do is complete the IRP with whatever status and length values are:

```

return CompleteIrp(Irp, status, len);

```

For easier viewing, here is the complete IRP_MJ_DEVICE_CONTROL handling:

```

NTSTATUS ZeroDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto& dic = irpSp->Parameters.DeviceIoControl;
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG_PTR len = 0;

    switch (dic.IoControlCode) {
        case IOCTL_ZERO_GET_STATS:
        {
            if (dic.OutputBufferLength < sizeof(ZeroStats)) {
                status = STATUS_BUFFER_TOO_SMALL;
                break;
            }
        }
    }
}

```

```

        auto stats = (ZeroStats*)Irp->AssociatedIrp.SystemBuffer;
        if (stats == nullptr) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }
        stats->TotalRead = g_TotalRead;
        stats->TotalWritten = g_TotalWritten;
        len = sizeof(ZeroStats);
        break;
    }

    case IOCTL_ZERO_CLEAR_STATS:
        g_TotalRead = g_TotalWritten = 0;
        break;
}

return CompleteIrp(Irp, status, len);
}

```

The stats have to be updated when data is read/written. It must be done in a thread safe way, as multiple clients may bombard the driver with read/write requests. Here is the updates ZeroWrite function:

```

NTSTATUS ZeroWrite(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto len = stack->Parameters.Write.Length;
    // update the number of bytes written
    InterlockedAdd64(&g_TotalWritten, len);
    return CompleteIrp(Irp, STATUS_SUCCESS, len);
}

```

The change to ZeroRead is very similar.

Astute readers may question the safety of the Ioctl implementations. For example, is reading the total number of bytes read/written with no multithreaded protection (while possible read/write operations are in effect) a correct operation, or is it a data race? Technically, it's a data race, as the driver might be updating to the stats globals while some client is reading the values, that could result in torn reads. One way to resolve that is by dispensing with the interlocked instructions and use a mutex or a fast mutex to protect access to these variables. Alternatively, There are functions to deal with these scenario, such as ReadAcquire64. Their implementation is CPU dependent. For x86/x64, they are actually normal reads, as the processor provides safety against such torn reads. On ARM CPUs, this requires a memory barrier to be inserted (memory barriers are beyond the scope of this book).



Save the number of bytes read/written to the Registry before the driver unloads. Read it back when the driver loads.

Replace the Interlocked instructions with a fast mutex to protect access to the stats.

Here is some client code to retrieve these stats:

```
ZeroStats stats;
if (!DeviceIoControl(hDevice, IOCTL_ZERO_GET_STATS,
    nullptr, 0, &stats, sizeof(stats), &bytes, nullptr))
    return Error("failed in DeviceIoControl");

printf("Total Read: %lld, Total Write: %lld\n",
    stats.TotalRead, stats.TotalWritten);
```

Summary

In this chapter, we learned how to handle IRPs, which drivers deal with all the time. Armed with this knowledge, we can start leveraging more kernel functionality, starting with process and thread callbacks in chapter 9. Before getting to that, however, there are more techniques and kernel APIs that may be useful for a driver developer, described in the next chapter.

Chapter 8: Advanced Programming Techniques (Part 1)

In this chapter we'll examine various techniques of various degrees of usefulness to driver developers.

In this chapter:

- **Driver Created Threads**
 - **Memory Management**
 - **Calling Other Drivers**
 - **Putting it All Together: The Melody Driver**
 - **Invoking System Services**
-

Driver Created Threads

We've seen how to create work items in chapter 6. Work items are useful when some code needs to execute on a separate thread, and that code is "bound" in time - that is, it's not too long, so that the driver doesn't "steal" a thread from the kernel worker threads. For long operations, however, it's preferable that drivers create their own separate thread(s). Two functions are available for this purpose:

```
NTSTATUS PsCreateSystemThread(  
    _Out_ PHANDLE ThreadHandle,  
    _In_  ULONG DesiredAccess,  
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,  
    _In_opt_ HANDLE ProcessHandle,  
    _Out_opt_ PCLIENT_ID ClientId,  
    _In_ PKSTART_ROUTINE StartRoutine,  
    _In_opt_ PVOID StartContext);  
  
NTSTATUS IoCreateSystemThread( // Win 8 and later  
    _Inout_ PVOID IoObject,  
    _Out_ PHANDLE ThreadHandle,  
    _In_  ULONG DesiredAccess,  
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,  
    _In_opt_ HANDLE ProcessHandle,
```

```
_Out_opt_ PCLIENT_ID ClientId,  
_In_ PKSTART_ROUTINE StartRoutine,  
_In_opt_ PVOID StartContext);
```

Both functions have the same set of parameters except the additional first parameter to `IoCreateSystemThread`. The latter function takes an additional reference on the object passed in (which must be a device object or a driver object), so the driver is not unloaded prematurely while the thread is alive. `IoCreateSystemThread` is only available for Windows 8 and later systems. Here is a description of the other parameters:

- `ThreadHandle` is the address of a handle to the created thread if successful. The driver must use `ZwClose` to close the handle at some point.
- `DesiredAccess` is the access mask requested. Drivers should simply use `THREAD_ALL_ACCESS` to get all possible access with the resulting handle.
- `ObjectAttributes` is the standard `OBJECT_ATTRIBUTES` structure. Most members have no meaning for a thread. The most common attributes to request of the returned handle is `OBJ_KERNEL_HANDLE`, but it's not needed if the thread is to be created in the System process - just pass `NULL`, which will always return a kernel handle.
- `ProcessHandle` is a handle to the process where this thread should be created. Drivers should pass `NULL` to indicate the thread should be part of the System process so it's not tied to any specific process' lifetime.
- `ClientId` is an optional output structure, providing the process and thread ID of the newly created thread. In most cases, this information is not needed, and `NULL` can be specified.
- `StartRoutine` is the function to execute in a separate thread of execution. This function must have the following prototype:

```
VOID KSTART_ROUTINE (_In_ PVOID StartContext);
```

The `StartContext` value is provided by the last parameter to `Ps/IoCreateSystemThread`. This could be anything (or `NULL`) that would give the new thread data to work with.

The function indicated by `StartRoutine` will start execution on a separate thread. It's executed with the IRQL being `PASSIVE_LEVEL` (0) in a critical region (where normal kernel APCs are disabled).

For `PsCreateSystemThread`, exiting the thread function is not enough to terminate the thread. An explicit call to `PsTerminateSystemThread` is required to properly manage the thread's lifetime:

```
NTSTATUS PsTerminateSystemThread(_In_ NTSTATUS ExitStatus);
```

The exit status is the exit code of the thread, which can be retrieved with `PsGetThreadExitStatus` if desired.

For `IoCreateSystemThread`, exiting the thread function is sufficient, as `PsTerminateSystemThread` is called on its behalf when the thread function returns. The exit code of the thread is always `STATUS_SUCCESS`.



`IoCreateSystemThread` is a wrapper around `PsCreateSystemThread` that increments the ref count of the passed in device/driver object, calls `PsCreateSystemThread` and then decrements the ref count and calls `PsTerminateSystemThread`.

Memory Management

We have looked at the most common functions for dynamic memory allocation in chapter 3. The most useful is `ExAllocatePoolWithTag`, which we have used multiple times in previous chapters. There are other functions for dynamic memory allocation you might find useful. Then, we'll examine lookaside lists, that allow more efficient memory management if fixed-size chunks are needed.

Pool Allocations

In addition to `ExAllocatePoolWithTag`, the Executive provides an extended version that indicates the importance of an allocation, taken into account in low memory conditions:

```
typedef enum _EX_POOL_PRIORITY {  
    LowPoolPriority,  
    LowPoolPrioritySpecialPoolOverrun = 8,  
    LowPoolPrioritySpecialPoolUnderrun = 9,  
    NormalPoolPriority = 16,  
    NormalPoolPrioritySpecialPoolOverrun = 24,  
    NormalPoolPrioritySpecialPoolUnderrun = 25,  
};
```

```

        HighPoolPriority = 32,
        HighPoolPrioritySpecialPoolOverrun = 40,
        HighPoolPrioritySpecialPoolUnderrun = 41
    } EX_POOL_PRIORITY;

PVOID ExAllocatePoolWithTagPriority (
    _In_ POOL_TYPE PoolType,
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG Tag,
    _In_ EX_POOL_PRIORITY Priority);

```

The priority-related values indicate the importance of succeeding an allocation if system memory is low (LowPoolPriority), very low (NormalPoolPriority), or completely out of memory (HighPoolPriority). In any case, the driver should be prepared to handle a failure.

The “special pool” values tell the Executive to make the allocation at the end of a page (“Overrun” values) or beginning of a page (“Underrun”) values, so it’s easier to catch buffer overflow or underflow. These values should only be used while tracking memory corruptions, as each allocation costs at least one page.

Starting with Windows 10 version 1909 (and Windows 11), two new pool allocation functions are supported. The first is ExAllocatePool2 declared like so:

```

PVOID ExAllocatePool2 (
    _In_ POOL_FLAGS Flags,
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG Tag);

```

Where the POOL_FLAGS enumeration consists of a combination of values shown in table 8-1:

Table 8-1: Flags for ExAllocatePool2

Flag (POOL_FLAG_)	Must recognize?	Description
USE_QUOTA	Yes	Charge allocation to calling process
UNINITIALIZED	Yes	Contents of allocated memory is not touched. Without this flag, the memory is zeroed out
CACHE_ALIGNED	Yes	Address should be CPU-cache aligned. This is “best effort”
RAISE_ON_FAILURE	Yes	Raises an exception (STATUS_INSUFFICIENT_RESOURCES) instead of returning NULL if allocation fails
NON_PAGED	Yes	Allocate from non-paged pool. The memory is executable on x86, and non-executable on all other platforms
PAGED	Yes	Allocate from paged pool. The memory is executable on x86, and non-executable on all other platforms
NON_PAGED_EXECUTABLE	Yes	Non paged pool with execute permissions
SPECIAL_POOL	No	Allocates from “special” pool (separate from the normal pool so it’s easier to find memory corruptions)

The *Must recognize?* column indicates whether failure to recognize or satisfy the flag causes the function to fail.

The second allocation function, ExAllocatePool3, is extensible, so new functions of this sort are unlikely to pop up in the future:

```
PVOID ExAllocatePool3 (
    _In_ POOL_FLAGS Flags,
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG Tag,
    _In_reads_opt_(ExtendedParametersCount)
        PCPOOL_EXTENDED_PARAMETER ExtendedParameters,
    _In_ ULONG ExtendedParametersCount);
```

This function allows customization with an array of “parameters”, where the supported parameter types may be extended in future kernel versions. The currently available parameters are defined with the `POOL_EXTENDED_PARAMETER_TYPE` enumeration:

```
typedef enum POOL_EXTENDED_PARAMETER_TYPE {
    PoolExtendedParameterInvalidType = 0,
    PoolExtendedParameterPriority,
    PoolExtendedParameterSecurePool,
    PoolExtendedParameterNumaNode,
    PoolExtendedParameterMax
} POOL_EXTENDED_PARAMETER_TYPE, *PPOOL_EXTENDED_PARAMETER_TYPE;
```

The array provided to `ExAllocatePool3` consists of structures of type `POOL_EXTENDED_PARAMETER`, each one specifying one parameter:

```
typedef struct _POOL_EXTENDED_PARAMETER {
    struct {
        ULONG64 Type : 8;
        ULONG64 Optional : 1;
        ULONG64 Reserved : 64 - 9;
    };

    union {
        ULONG64 Reserved2;
        PVOID Reserved3;
        EX_POOL_PRIORITY Priority;
        POOL_EXTENDED_PARAMS_SECURE_POOL* SecurePoolParams;
        POOL_NODE_REQUIREMENT PreferredNode; // ULONG
    };
} POOL_EXTENDED_PARAMETER, *PPOOL_EXTENDED_PARAMETER;
```

The `Type` member indicates which of the union members is valid for this parameter (`POOL_EXTENDED_PARAMETER_TYPE`). `Optional` indicates if the parameter set is optional or required. An optional parameter that fails to be satisfied does not cause the `ExAllocatePool3` to fail. Based on `Type`, the correct member in the union must be set. Currently, these parameters are available:

- Priority of the allocation (`Priority` member)
- Preferred NUMA node (`PreferredNode` member)
- Use secure pool (discussed later, `SecurePoolParams` member)

The following example shows using `ExAllocatePool3` to achieve the same effect as `ExAllocatePoolWithTagPriority` for non-paged memory:

```
PVOID AllocNonPagedPriority(ULONG size, ULONG tag, EX_POOL_PRIORITY priority) {
    POOL_EXTENDED_PARAMETER param;
    param.Optional = FALSE;
    param.Type = PoolExtendedParameterPriority;
    param.Priority = priority;

    return ExAllocatePool3(PPOOL_FLAG_NON_PAGED, size, tag, &param, 1);
}
```

Secure Pools

Secure pools introduced in Windows 10 version 1909 allow kernel callers to have a memory pool that cannot be accessed by other kernel components. This kind of protection is internally achieved by the Hyper-V hypervisor, leveraging its power to protect memory access even from the kernel, as the memory is part of *Virtual Trust Level* (VTL) 1 (the secure world). Currently, secure pools are not fully documented, but here are the basic steps to use a secure pool.



Secure pools are only available if *Virtualization Based Security* (VBS) is active (meaning Hyper-V exists and creates the two worlds - normal and secure). Discussion of VBS is beyond the scope of this book. Consult information online (or the Windows Internals books) for more on VBS.

A secure pool can be created with `ExCreatePool`, returning a handle to the pool:

```
#define POOL_CREATE_FLG_SECURE_POOL      0x1
#define POOL_CREATE_FLG_USE_GLOBAL_POOL 0x2
#define POOL_CREATE_FLG_VALID_FLAGS (POOL_CREATE_FLG_SECURE_POOL | \
                                      POOL_CREATE_FLG_USE_GLOBAL_POOL)

NTSTATUS ExCreatePool (
    _In_ ULONG Flags,
    _In_ ULONG_PTR Tag,
    _In_opt_ POOL_CREATE_EXTENDED_PARAMS* Params,
    _Out_ HANDLE* PoolHandle);
```

Currently, flags should be `POOL_CREATE_FLG_VALID_FLAGS` (both supported flags), and `Params` should be `NULL`. `PoolHandle` contains the pool handle if the call succeeds.

Allocating from a secure pool must be done with `ExAllocatePool3`, described in the previous section with a `POOL_EXTENDED_PARAMS_SECURE_POOL` structure as a parameter:

```
#define SECURE_POOL_FLAGS_NONE      0x0
#define SECURE_POOL_FLAGS_FREEABLE  0x1
#define SECURE_POOL_FLAGS_MODIFIABLE 0x2

typedef struct _POOL_EXTENDED_PARAMS_SECURE_POOL {
    HANDLE SecurePoolHandle;    // pool handle
    PVOID Buffer;               // initial data
    ULONG_PTR Cookie;           // for validation
    ULONG SecurePoolFlags;      // flags above
} POOL_EXTENDED_PARAMS_SECURE_POOL;
```

`Buffer` points to existing data to be initially stored in the new allocation. `Cookie` is used for validation, by calling `ExSecurePoolValidate`. Freeing memory from a secure pool must be done with a new function, `ExFreePool2`:

```
VOID ExFreePool2 (
    _Pre_notnull_ PVOID P,
    _In_ ULONG Tag,
    _In_reads_opt_(ExtendedParametersCount)
        PCPOOL_EXTENDED_PARAMETER ExtendedParameters,
    _In_ ULONG ExtendedParametersCount);
```

If `ExtendedParameters` is `NULL` (and `ExtendedParametersCount` is zero), the call is diverted to the normal `ExFreePool`, which will fail for a secure pool. For a secure pool, a single `POOL_EXTENDED_PARAMETER` is required that has the pool parameters with the pool handle only. `Buffer` should be `NULL`.

Finally, a secure pool must be destroyed with `ExDestroyPool`:

```
VOID ExDestroyPool (_In_ HANDLE PoolHandle);
```

Overloading the new and delete Operators

We know there is no C++ runtime in the kernel, which means some C++ features that work as expected in user mode don't work in kernel mode. One of these features are the `new` and `delete` C++ operators. Although we can use the dynamic memory allocation functions, `new` and `delete` have a couple of advantages over calling the raw functions:

- new causes a constructor to be invoked, and delete causes the destructor to be invoked.
- new accepts a type for which memory must be allocated, rather than specifying a number of bytes.

Fortunately, C++ allows overloading the new and delete operators, either globally or for specific types. new can be overloaded with extra parameters that are needed for kernel allocations - at least the pool type must be specified. The first argument to any overloaded new is the number of bytes to allocate, and any extra parameters can be added after that. These are specified with parenthesis when actually used. The compiler inserts a call to the appropriate constructor, if exists.

Here is a basic implementation of an overloaded new operator that calls ExAllocatePoolWithTag:

```
void* __cdecl operator new(size_t size, POOL_TYPE pool, ULONG tag) {  
    return ExAllocatePoolWithTag(pool, size, tag);  
}
```

The __cdecl modifier indicates this should be using the C calling convention (rather than the __stdcall convention). It only matters in x86 builds, but still should be specified as shown.

Here is an example usage, assuming an object of type MyData needs to be allocated from paged pool:

```
MyData* data = new (PagedPool, DRIVER_TAG) MyData;  
if(data == nullptr)  
    return STATUS_INSUFFICIENT_RESOURCES;  
// do work with data
```

The size parameter is never specified explicitly as the compiler inserts the correct size (which is essentially sizeof(MyData) in the above example). All other parameters must be specified. We can make the overload simpler to use if we default the tag to a macro such as DRIVER_TAG, expected to exist:

```
void* __cdecl operator new(size_t size, POOL_TYPE pool,  
    ULONG tag = DRIVER_TAG) {  
    return ExAllocatePoolWithTag(pool, size, tag);  
}
```

And the corresponding usage is simpler:

```
MyData* data = new (PagedPool) MyData;
```

In the above examples, the default constructor is invoked, but it's perfectly valid to invoke any other constructor that exists for the type. For example:

```
struct MyData {  
    MyData(ULONG someValue);  
    // details not shown  
};  
  
auto data = new (PagedPool) MyData(200);
```

We can easily extend the overloading idea to other overloads, such as one that wraps `ExAllocatePoolWithTagPriority`:

```
void* __cdecl operator new(size_t size, POOL_TYPE pool,  
    EX_POOL_PRIORITY priority, ULONG tag = DRIVER_TAG) {  
    return ExAllocatePoolWithTagPriority(pool, size, tag, priority);  
}
```

Using the above operator is just a matter of adding a priority in parenthesis:

```
auto data = new (PagedPool, LowPoolPriority) MyData(200);
```

Another common case is where you already have an allocated block of memory to store some object (perhaps allocated by a function out of your control), but you still want to initialize the object by invoking a constructor. Another new overload can be used for this purpose, known as *placement new*, since it does not allocate anything, but the compiler still adds a call to a constructor. Here is how to define a placement new operator overload:

```
void* __cdecl operator new(size_t size, void* p) {  
    return p;  
}
```

And an example usage:

```
void* SomeFunctionAllocatingObject();  
  
MyData* data = (MyData*)SomeFunctionAllocatingObject();  
new (data) MyData;
```

Finally, an overload for `delete` is required so the memory can be freed at some point, calling the destructor if it exists. Here is how to overload the `delete` operator:

```
void __cdecl operator delete(void* p, size_t) {  
    ExFreePool(p);  
}
```

The extra size parameter is not used in practice (zero is always the value provided), but the compiler requires it.



Remember that you cannot have global objects that have default constructors that do something, since there is no runtime to invoke them. The compiler will report a warning if you try. A way around it (of sorts) is to declare the global variable as a pointer, and then use an overloaded `new` to allocate and invoke a constructor in `DriverEntry`. of course, you must remember to call `delete` in the driver's unload routine.

Another variant of the `delete` operator the compiler might insist on if you set the compiler conformance to C++17 or newer is the following:

```
void __cdecl operator delete(void* p, size_t, std::align_val_t) {  
    ExFreePool(p);  
}
```

You can look up the meaning of `std::align_val_t` in a C++ reference, but it does not matter for our purposes.

Lookaside Lists

The dynamic memory allocation functions discussed so far (the `ExAllocatePool*` family of APIs) are generic in nature, and can accommodate allocations of any size. Internally, managing the pool is non-trivial: various lists are needed to manage allocations and deallocations of different sizes. This management aspect of the pools is not free.

One fairly common case that leaves room for optimizations is when fixed-sized allocations are needed. When such allocation is freed, it's possible to not really free it, but just mark it as available. The next allocation request can be satisfied

by the existing block, which is much faster to do than allocating a fresh block. This is exactly the purpose of lookaside lists.

There are two APIs to use for working with lookaside lists. The original one, available from Windows 2000, and a newer available from Vista. I'll describe both, as they are quite similar.

The “Classic” Lookaside API

The first thing to do is to initialize the data structure managing a lookaside list. Two functions are available, which are essentially the same, selecting the paged pool or non-paged pool where the allocations should be coming from. Here is the paged pool version:

```
VOID ExInitializePagedLookasideList (  
    _Out_ PPAGED_LOOKASIDE_LIST Lookaside,  
    _In_opt_ PALLOCATE_FUNCTION Allocate,  
    _In_opt_ PFREE_FUNCTION Free,  
    _In_ ULONG Flags,  
    _In_ SIZE_T Size,  
    _In_ ULONG Tag,  
    _In_ USHORT Depth);
```

The non-paged variant is practically the same, with the function name being `ExInitializeNPagedLookasideList`.

The first parameter is the resulting initialized structure. Although, the structure layout is described in *wdm.h* (with a macro named `GENERAL_LOOKASIDE_LAYOUT` to accommodate multiple uses that can't be shared in other ways using the C language), you should treat this structure as opaque.

The *Allocate* parameter is an optional allocation function that is called by the lookaside implementation when a new allocation is required. If specified, the allocation function must have the following prototype:

```
PVOID AllocationFunction (  
    _In_ POOL_TYPE PoolType,  
    _In_ SIZE_T NumberOfBytes,  
    _In_ ULONG Tag);
```

The allocation function receives the same parameters as `ExAllocatePoolWithTag`. In fact, if the allocation function is not specified, this

is the call made by the lookaside list manager. If you don't require any other code, just specify `NULL`. A custom allocation function could be useful for debugging purposes, for example. Another possibility is to call `ExAllocatePoolWithTagPriority` instead of `ExAllocatePoolWithTag`, if that makes sense for your driver.

If you do provide an allocation function, you might need to provide a de-allocation function in the *Free* parameter. If not specified, the lookaside list manager calls `ExFreePool`. Here is the expected prototype for this function:

```
VOID FreeFunction (  
    _In_ __drv_freesMem(Mem) PVOID Buffer);
```

The next parameter, *Flags* can be zero or `POOL_RAISE_IF_ALLOCATION_FAILURE` (Windows 8 and later) that indicates an exception should be raised (`STATUS_INSUFFICIENT_RESOURCE`) if an allocation fails, instead of returning `NULL` to the caller.

The *Size* parameter is the size of chunks managed by the lookaside list. Usually, you would specify it as `sizeof` some structure you want to manage. *Tag* is the tag to use for allocations. Finally, the last parameter, *Depth*, indicates the number of allocations to keep in a cache. The documentation indicates this parameter is “reserved” and should be zero, which makes the lookaside list manager to choose something appropriate. Regardless of the number, the “depth” is adjusted based on the allocation patterns used with the lookaside list.

Once a lookaside list is initialized, you can request a memory block (of the size specified in the initialization function, of course) by calling `ExAllocateFromPagedLookasideList`:

```
PVOID ExAllocateFromPagedLookasideList (  
    _Inout_ PPAGED_LOOKASIDE_LIST Lookaside)
```

Nothing could be simpler - no special parameters are required, since everything else is already known. The corresponding function for a non-paged pool lookaside list is `ExAllocateFromNPagedLookasideList`.

The opposite function used to free an allocation (or return it to the cache) is `ExFreeToPagedLookasideList`:

```
VOID ExFreeToPagedLookasideList (  
    _Inout_ PPAGED_LOOKASIDE_LIST Lookaside,  
    _In_ __drv_freesMem(Mem) PVOID Entry)  
;
```

The only value required is the pointer to free (or return to the cache). As you probably guess, the non-paged pool variant is `ExFreeToNPagedLookasideList`.

Finally, when the lookaside list is no longer needed, it must be freed by calling `ExDeletePagedLookasideList`:

```
VOID ExDeletePagedLookasideList (  
    _Inout_ PPAGED_LOOKASIDE_LIST Lookaside);  
;
```

One nice benefit of lookaside lists is that you don't have to return all allocations to the list by repeatedly calling `ExFreeToPagedLookasideList` before calling `ExDeletePagedLookasideList`; the latter is enough, and will free all allocated blocks automatically. `ExDeleteNPagedLookasideList` is the corresponding non-paged variant.



Write a C++ class wrapper for lookaside lists using the above APIs.

The Newer Lookaside API

The newer API provides two main benefits over the classic API:

- Uniform API for paged and non-paged blocks.
- The lookaside list structure itself is passed to the custom allocate and free functions (if provided), that allows accessing driver data (example shown later).

Initializing a lookaside list is accomplished with `ExInitializeLookasideListEx`:

```
NTSTATUS ExInitializeLookasideListEx (  
    _Out_ PLOOKASIDE_LIST_EX Lookaside,  
    _In_opt_ PALLOCATE_FUNCTION_EX Allocate,  
    _In_opt_ PFREE_FUNCTION_EX Free,  
    _In_ POOL_TYPE PoolType,  
    _In_opt_ PDRIVER_NAME String)  
;
```

```
_In_ ULONG Flags,  
_In_ SIZE_T Size,  
_In_ ULONG Tag,  
_In_ USHORT Depth);
```

PLOOKASIDE_LIST_EX is the opaque data structure to initialize, which must be allocated from non-paged memory, regardless of whether the lookaside list is to manage paged or non-paged memory.

The allocation and free functions are optional, just as they are with the classic API. These are their prototypes:

```
PVOID AllocationFunction (  
    _In_ POOL_TYPE PoolType,  
    _In_ SIZE_T NumberOfBytes,  
    _In_ ULONG Tag,  
    _Inout_ PLOOKASIDE_LIST_EX Lookaside);  
VOID FreeFunction (  
    _In_ __drv_freesMem(Mem) PVOID Buffer,  
    _Inout_ PLOOKASIDE_LIST_EX Lookaside);
```

Notice the lookaside list itself is a parameter. This could be used to access driver data that is part of a larger structure containing the lookaside list. For example, suppose the driver has the following structure:

```
struct MyData {  
    ULONG SomeData;  
    LIST_ENTRY SomeHead;  
    LOOKASIDE_LIST_EX Lookaside;  
};
```

The driver creates an instance of that structure (maybe globally, or on a per-client basis). Let's assume it's created dynamically for every client creating a file object to talk to a device the driver manages:

```
// if new is overridden as described earlier in this chapter  
MyData* pData = new (NonPagedPool) MyData;  
// or with a standard allocation call  
MyData* pData = (MyData*)ExAllocatePoolWithTag(NonPagedPool,  
    sizeof(MyData), DRIVER_TAG);  
  
// initialize the lookaside list  
ExInitializeLookasideListEx(&pData->Lookaside, MyAlloc, MyFree, ...);
```

In the allocation and free functions, we can get a pointer to our MyData object that contains whatever lookaside list is being used at the time:

```
PVOID MyAlloc(PPOOL_TYPE type, SIZE_T size, ULONG tag,
    PLOOKASIDE_LIST_EX lookaside) {
    MyData* data = CONTAINING_RECORD(lookaside, MyData, Lookaside);
    // access members
    //...
}
```

The usefulness of this technique is if you have multiple lookaside lists, each having their own “context” data. Obviously, if you just have one such list stored globally, you can just access whatever global variables you need.

Continuing with `ExInitializeLookasideListEx` - *PoolType* is the pool type to use; this is where the driver selects where allocations should be allocated from. *Size*, *Tag* and *Depth* have the same meaning as they do in the classic API.

The *Flags* parameter can be zero, or one of the following:

- `EX_LOOKASIDE_LIST_EX_FLAGS_RAISE_ON_FAIL` - raise an exception instead of returning NULL to the caller in case of an allocation failure.
- `EX_LOOKASIDE_LIST_EX_FLAGS_FAIL_NO_RAISE` - this flag can only be specified if a custom allocation routine is specified, which causes the pool type provided to the allocation function to be ORed with the `POOL_QUOTA_FAIL_INSTEAD_OF_RAISE` flag that causes a call to `ExAllocationPoolWithQuotaTag` to return NULL on quota limit violation instead of raising the `POOL_QUOTA_FAIL_INSTEAD_OF_RAISE` exception. See the docs for more details.



The above flags are mutually exclusive.

Once the lookaside list is initialized, allocation and deallocation are done with the following APIs:

```
PVOID ExAllocateFromLookasideListEx (_Inout_ PLOOKASIDE_LIST_EX Lookaside);
VOID ExFreeToLookasideListEx (
    _Inout_ PLOOKASIDE_LIST_EX Lookaside,
    _In_ __drv_freesMem(Entry) PVOID Entry);
```

Of course, the terms “allocation” and “deallocation” are in the context of a lookaside list, meaning allocations could be reused, and deallocations might

return the block to the cache.

Finally, a lookaside list must be deleted with `ExDeleteLookasideListEx`:

```
VOID ExDeleteLookasideListEx (_Inout_ PLOOKASIDE_LIST_EX Lookaside);
```

Calling Other Drivers

One way to talk to other drivers is to be a “proper” client by calling `ZwOpenFile` or `ZwCreateFile` in a similar manner to what a user-mode client does. Kernel callers have other options not available for user-mode callers. One of the options is creating IRPs and sending them to a device object directly for processing.

In most cases IRPs are created by one of the three managers, part of the Executive: I/O manager, Plug & Play manager, and Power manager. In the cases we’ve seen so far, the I/O manager is the one creating IRPs for create, close, read, write, and device I/O control request types. Drivers can create IRPs as well, initialize them and then send them directly to another driver for processing. This could be more efficient than opening a handle to the desired device, and then making calls using `ZwReadFile`, `ZwWriteFile` and similar APIs we’ll look at in more detail in a later chapter. In some cases, opening a handle to a device might not even be an option, but obtaining a device object pointer might still be possible.

The kernel provides a generic API for building IRPs, starting with `IoAllocateIrp`. Using this API requires the driver to register a completion routine so the IRP can be properly freed. We’ll examine these techniques in a later chapter (“Advanced Programming Techniques (Part 2)”). In this section, I’ll introduce a simpler function to build a device I/O control IRP using `IoBuildDeviceIoControlRequest`:

```
PIRP IoBuildDeviceIoControlRequest(
    _In_      ULONG IoControlCode,
    _In_      PDEVICE_OBJECT DeviceObject,
    _In_opt_  PVOID InputBuffer,
    _In_      ULONG InputBufferLength,
    _Out_opt_ PVOID OutputBuffer,
    _In_      ULONG OutputBufferLength,
    _In_      BOOLEAN InternalDeviceIoControl,
    _In_opt_  PKEVENT Event,
    _Out_     PIO_STATUS_BLOCK IoStatusBlock);
```

The API returns a proper IRP pointer on success, including filling in the first `IO_STACK_LOCATION`, or `NULL` on failure. Some of the parameters to `IoBuildDeviceIoControlRequest` are the same provided to the `DeviceIoControl` user-mode API (or to its kernel equivalent, `ZwDeviceIoControlFile`) - *IoControlCode*, *InputBuffer*, *InputBufferLength*, *OutputBuffer* and *OutputBufferLength*.

The other parameters are the following:

- *DeviceObject* is the target device of this request. It's needed so the API can allocate the correct number of `IO_STACK_LOCATION` structures that accompany any IRP.
- *InternalDeviceControl* indicates whether the IRP should set its major function to `IRP_MJ_INTERNAL_DEVICE_CONTROL` (`TRUE`) or `IRP_MJ_DEVICE_CONTROL` (`FALSE`). This obviously depends on the target device's expectations.
- *Event* is an optional pointer to an event object that gets signaled when the IRP is completed by the target device (or some other device the target may send the IRP to). An event is needed if the IRP is sent for synchronous processing, so that the caller can wait on the event if the operation has not yet completed. We'll see a complete example in the next section.
- *IoStatusBlock* returns the final status of the IRP (status and information), so the caller can examine it if it so wishes.

The call to `IoBuildDeviceIoControlRequest` just builds the IRP - it is not sent anywhere at this point. To actually send the IRP to a device, call the generic `IoCallDriver` API:

```
NTSTATUS IoCallDriver(  
    _In_ PDEVICE_OBJECT DeviceObject,  
    _Inout_ PIRP Irp);
```

`IoCallDriver` advances the current I/O stack location to the next, and then invokes the target driver's major function dispatch routine. It returns whatever is returned from that dispatch routine. Here is a *very* simplified implementation:

```
NTSTATUS IoCallDriver(PDEVICE_OBJECT DeviceObject, PIRP Irp {  
    // update the current layer index  
    DeviceObject->CurrentLocation--;  
    auto irpSp = IoGetNextIrpStackLocation(Irp);  
    // make the next stack location the current one  
    Irp->Tail.Overlay.CurrentStackLocation = irpSp;
```

```
// update device object
irpSp->DeviceObject = DeviceObject;

return (DeviceObject->DriverObject->MajorFunction[irpSp->MajorFunction])
    (DeviceObject, Irp);
}
```

The main question remaining is how to we get a pointer to a device object in the first place? One way is by calling `IoGetDeviceObjectPointer`:

```
NTSTATUS IoGetDeviceObjectPointer(
    _In_ PUNICODE_STRING ObjectName,
    _In_ ACCESS_MASK DesiredAccess,
    _Out_ PFILE_OBJECT *FileObject,
    _Out_ PDEVICE_OBJECT *DeviceObject);
```

The *ObjectName* parameter is the fully-qualified name of the device object in the Object Manager's namespace (as can be viewed with the *WinObj* tool from *Sysinternals*). Desired access is usually `FILE_READ_DATA`, `FILE_WRITE_DATA` or `FILE_ALL_ACCESS`. Two values are returned on success: the device object pointer (in *DeviceObject*) and an open file object pointing to the device object (in *FileObject*).

The file object is not usually needed, but it should be kept around as a means of keeping the device object referenced. When you're done with the device object, call `ObDereferenceObject` on the file object pointer to decrement the device object's reference count indirectly. Alternatively, you can increment the device object's reference count (`ObReferenceObject`) and then decrement the file object's reference count so you don't have to keep it around.

The next section demonstrates usage of these APIs.

Putting it All Together: The Melody Driver

The *Melody* driver we'll build in this section demonstrates many of the techniques shown in this chapter. The melody driver allows playing sounds asynchronously (contrary to the `Beep` user-mode API that plays sounds synchronously). A client application calls `DeviceIoControl` with a bunch of notes to play, and the driver will play them as requested without blocking. Another sequence of notes can then be sent to the driver, those notes queued to be played after the first sequence is finished.

It's possible to come up with a user-mode solution that would do essentially the same thing, but this can only be easily done in the context of a single process. A driver, on the other hand, can accept calls from multiple processes, having a “global” ordering of playback. In any case, the point is to demonstrate driver programming techniques, rather than managing a sound playing scenario.

We'll start by creating an empty WDM driver, as we've done in previous chapters, named *KMelody*. Then we'll add a file named *MelodyPublic.h* to serve as the common data to the driver and a user-mode client. This is where we define what a note looks like and an I/O control code for communication:

```
// MelodyPublic.h
#pragma once

#define MELODY_SYMLINK L"\\??\\KMelody"

struct Note {
    ULONG Frequency;
    ULONG Duration;
    ULONG Delay{ 0 };
    ULONG Repeat{ 1 };
};

#define MELODY_DEVICE 0x8003

#define IOCTL_MELODY_PLAY \
    CTL_CODE(MELODY_DEVICE, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

A note consists of a frequency (in Hertz) and duration to play. To make it a bit more interesting, a delay and repeat count are added. If Repeat is greater than one, the sound is played Repeat times, with a delay of Delay between repeats. Duration and Delay are provided in milliseconds.

The architecture we'll go for in the driver is to have a thread created when the first client opens a handle to our device, and that thread will perform the playback based on a queue of notes the driver manages. The thread will be shut down when the driver unloads.

It may seem asymmetric at this point - why not create the thread when the driver loads? As we shall see shortly, there is a little “snag” that we have to deal with that prevents creating the thread when the driver loads.

Let's start with DriverEntry. It needs to create a device object and a symbolic link. Here is the full function:

```
PlaybackState* g_State;

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    g_State = new (PagedPool) PlaybackState;
    if (g_State == nullptr)
        return STATUS_INSUFFICIENT_RESOURCES;

    auto status = STATUS_SUCCESS;
    PDEVICE_OBJECT DeviceObject = nullptr;
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\KMelody");

    do {
        UNICODE_STRING name = RTL_CONSTANT_STRING(L"\\Device\\KMelody");
        status = IoCreateDevice(DriverObject, 0, &name, FILE_DEVICE_UNKNOWN,
                                0, FALSE, &DeviceObject);
        if (!NT_SUCCESS(status))
            break;

        status = IoCreateSymbolicLink(&symLink, &name);
        if (!NT_SUCCESS(status))
            break;
    } while (false);

    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "Error (0x%08X)\\n", status));
        delete g_State;
        if (DeviceObject)
            IoDeleteDevice(DeviceObject);
        return status;
    }

    DriverObject->DriverUnload = MelodyUnload;
    DriverObject->MajorFunction[IRP_MJ_CREATE] =
        DriverObject->MajorFunction[IRP_MJ_CLOSE] = MelodyCreateClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = MelodyDeviceControl;

    return status;
}
```

Most of the code should be familiar by now. The only new code is the creation of an object of type PlaybackState. The new C++ operator is overloaded as described earlier in this chapter. If allocating a PlaybackState instance fails, DriverEntry returns STATUS_INSUFFICIENT_RESOURCES, reporting a failure to the kernel.

The PlaybackState class is going to manage the list of notes to play and most other functionality specific to the driver. Here is its declaration (in

PlaybackState.h):

```
struct PlaybackState {
    PlaybackState();
    ~PlaybackState();

    NTSTATUS AddNotes(const Note* notes, ULONG count);
    NTSTATUS Start(PVOID IoObject);
    void Stop();

private:
    static void PlayMelody(PVOID context);
    void PlayMelody();

    LIST_ENTRY m_head;
    FastMutex m_lock;
    PAGED_LOOKASIDE_LIST m_lookaside;
    KSEMAPHORE m_counter;
    KEVENT m_stopEvent;
    HANDLE m_hThread{ nullptr };
};
```

`m_head` is the head of the linked list holding the notes to play. Since multiple threads can access this list, it must be protected with a synchronization object. In this case, we'll go with a fast mutex. `FastMutex` is a wrapper class similar to the one we saw in chapter 6, with the added twist that it's initialized in its constructor rather than a separate `Init` method. This is convenient, and possible, because `PlaybackState` is allocated dynamically, causing its constructor to be invoked, along with constructors for data members (if any).

The note objects will be allocated from a lookaside list (`m_lookaside`), as each note has a fixed size, and there is a strong likelihood of many notes coming and going. `m_stopEvent` is an event object that will be used as a way to signal our playback thread to terminate. `m_hThread` is the playback thread handle. Finally, `m_counter` is a semaphore that is going to be used in a somewhat counter-intuitive way, its internal count indicating the number of notes in the queue.

As you can see, the event and semaphore don't have wrapper classes, so we need to initialize them in the `PlaybackState` constructor. Here is the constructor in full (in *PlaybackState.cpp*) with an addition of a type that is going to hold a single node:

```
struct FullNote : Note {
    LIST_ENTRY Link;
};

PlaybackState::PlaybackState() {
```

```
InitializeListHead(&m_head);
KeInitializeSemaphore(&m_counter, 0, 1000);
KeInitializeEvent(&m_stopEvent, SynchronizationEvent, FALSE);
ExInitializePagedLookasideList(&m_lookaside, nullptr, nullptr, 0,
    sizeof(FullNote), DRIVER_TAG, 0);
}
```

Here are the initialization steps taken by the constructor:

- Initialize the linked list to an empty list (`InitializeListHead`).
- Initialize the semaphore to a value of zero, meaning no notes are queued up at this point, with a maximum of 1000 queued notes. Of course, this number is arbitrary.
- Initialize the stop event as a `SynchronizationEvent` type in the non-signaled state (`KeInitializeEvent`). Technically, a `NotificationEvent` would have worked just as well, as just one thread will be waiting on this event as we'll see later.
- Initialize the lookaside list to managed paged pool allocations with size of `sizeof(FullNote)`. `FullNote` extends `Note` to include a `LIST_ENTRY` member, otherwise we can't store such objects in a linked list. The `FullNote` type should not be visible to user-mode, which is why it's defined privately in the driver's source files only.

`DRIVER_TAG` and `DRIVER_PREFIX` are defined in the file *KMelody.h*.

Before the driver finally unloads, the `PlaybackState` object is going to be destroyed, invoking its destructor:

```
PlaybackState::~PlaybackState() {
    Stop();
    ExDeletePagedLookasideList(&m_lookaside);
}
```

The call to `Stop` signals the playback thread to terminate as we'll see shortly. The only other thing left to do in terms of cleanup is to free the lookaside list.

The unload routine for the driver is similar to ones we've seen before with the addition of freeing the `PlaybackState` object:

```
void MelodyUnload(PDRIVER_OBJECT DriverObject) {
    delete g_State;
```

```

    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\KMelody");
    IoDeleteSymbolicLink(&symLink);
    IoDeleteDevice(DriverObject->DeviceObject);
}

```

The IRP_MJ_DEVICE_CONTROL handler is where notes provided by a client need to be added to the queue of notes to play. The implementation is pretty straightforward because the heavy lifting is performed by the PlaybackState::AddNotes method. Here is MelodyDeviceControl that validates the client's data and then invokes AddNotes:

```

NTSTATUS MelodyDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto& dic = irpSp->Parameters.DeviceIoControl;
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG info = 0;

    switch (dic.IoControlCode) {
        case IOCTL_MELODY_PLAY:
            if (dic.InputBufferLength == 0 ||
                dic.InputBufferLength % sizeof(Note) != 0) {
                status = STATUS_INVALID_BUFFER_SIZE;
                break;
            }
            auto data = (Note*)Irp->AssociatedIrp.SystemBuffer;
            if (data == nullptr) {
                status = STATUS_INVALID_PARAMETER;
                break;
            }

            status = g_State->AddNotes(data,
                dic.InputBufferLength / sizeof(Note));
            if (!NT_SUCCESS(status))
                break;
            info = dic.InputBufferLength;
            break;
    }
    return CompleteRequest(Irp, status, info);
}

```

CompleteRequest is a helper that we've seen before that completes the IRP with the given status and information:

```

NTSTATUS CompleteRequest(PIRP Irp,
    NTSTATUS status = STATUS_SUCCESS, ULONG_PTR info = 0);
//...
NTSTATUS CompleteRequest(PIRP Irp, NTSTATUS status, ULONG_PTR info) {
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = info;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

```

PlaybackState::AddNotes needs to iterate over the provided notes. Here is the beginning of the function:

```
NTSTATUS PlaybackState::AddNotes(const Note* notes, ULONG count) {
    KdPrint((DRIVER_PREFIX "State::AddNotes %u\n", count));

    for (ULONG i = 0; i < count; i++) {
```

For each note, it needs to allocate a FullNote structure from the lookaside list:

```
auto fullNote = (FullNote*)ExAllocateFromPagedLookasideList(&m_lookaside);
if (fullNote == nullptr)
    return STATUS_INSUFFICIENT_RESOURCES;
```

If succesful, the note data is copied to the FullNote and is added to the linked list under the protection of the fast mutex:

```
//
// copy the data from the Note structure
//
memcpy(fullNote, &notes[i], sizeof(Note));

//
// insert into the linked list
//
Locker locker(m_lock);
InsertTailList(&m_head, &fullNote->Link);
}
```

Locker<T> is the same type we looked at in chapter 6. The notes are inserted at the back of the list with InsertTailList. This is where we must provide a pointer to a LIST_ENTRY object, which is why FullNote objects are used instead of just Note. Finally, when the loop is completed, the semaphore must be incremented by the number of notes to indicate there are more count more notes to play:

```
//
// make the semaphore signaled (if it wasn't already) to
// indicate there are new note(s) to play
//
KeReleaseSemaphore(&m_counter, 2, count, FALSE);
KdPrint((DRIVER_PREFIX "Semaphore count: %u\n",
    KeReadStateSemaphore(&m_counter)));
```

The value 2 used in KeReleaseSemaphore is the temporary priority boost a driver can provide to a thread that is released because of the semaphore

becoming signaled (the same thing happens with the second parameter to `IoCompleteRequest`). I've chosen the value 2 arbitrarily. The value 0 (`IO_NO_INCREMENT`) is fine as well.

For debugging purposes, it may be useful to read the semaphore's count with `KeReadStateSemaphore` as was done in the above code. Here is the full function (without the comments):

```
NTSTATUS PlaybackState::AddNotes(const Note* notes, ULONG count) {
    KdPrint((DRIVER_PREFIX "State::AddNotes %u\n", count));

    for (ULONG i = 0; i < count; i++) {
        auto fullNote =
            (FullNote*)ExAllocateFromPagedLookasideList(&m_lookaside);
        if (fullNote == nullptr)
            return STATUS_INSUFFICIENT_RESOURCES;

        memcpy(fullNote, &notes[i], sizeof(Note));

        Locker locker(m_lock);
        InsertTailList(&m_head, &fullNote->Link);
    }
    KeReleaseSemaphore(&m_counter, 2, count, FALSE);
    KdPrint((DRIVER_PREFIX "Semaphore count: %u\n",
        KeReadStateSemaphore(&m_counter)));

    return STATUS_SUCCESS;
}
```

The next part to look at is handling `IRP_MJ_CREATE` and `IRP_MJ_CLOSE`. In earlier chapters, we just completed these IRPs successfully and that was it. This time, we need to create the playback thread when the first client opens a handle to our device. The initialization in `DriverEntry` points both indices to the same function, but the code is slightly different between the two. We could separate them to different functions, but if the difference is not great we might decide to handle both within the same function.

For `IRP_MJ_CLOSE`, there is nothing to do but complete the IRP successfully. For `IRP_MJ_CREATE`, we want to start the playback thread the first time the dispatch routine is invoked. Here is the code:

```
NTSTATUS MelodyCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    auto status = STATUS_SUCCESS;
    if (IoGetCurrentIrpStackLocation(Irp)->MajorFunction == IRP_MJ_CREATE) {
        //
        // create the "playback" thread (if needed)
        //
        status = g_State->Start(DeviceObject);
    }
}
```

```
    }  
    return CompleteRequest(Irp, status);  
}
```

The I/O stack location contains the IRP major function code we can use to make the distinction as required here. In the Create case, we call `PlaybackState::Start` with the device object pointer that would be used to keep the driver object alive as long as the thread is running. Let's see what that method looks like.

```
NTSTATUS PlaybackState::Start(PVOID IoObject) {  
    Locker locker(m_lock);  
    if (m_hThread)  
        return STATUS_SUCCESS;  
  
    return IoCreateSystemThread(  
        IoObject,           // Driver or device object  
        &m_hThread,         // resulting handle  
        THREAD_ALL_ACCESS, // access mask  
        nullptr,           // no object attributes required  
        NtCurrentProcess(), // create in the current process  
        nullptr,           // returned client ID  
        PlayMelody,        // thread function  
        this);             // passed to thread function  
}
```

Acquiring the fast mutex ensures that a second thread is not created (as `m_hThread` would already be non-NULL). The thread is created with `IoCreateSystemThread`, which is preferred over `PsCreateSystemThread` because it ensures that the driver is not unloaded while the thread is executing (this does require Windows 8 or later).

The passed-in I/O object is the device object provided by the `IRP_MJ_CREATE` handler. The most common way of creating a thread by a driver is to run it in the context of the *System* process, as it normally should not be tied to a user-mode process. Our case, however, is more complicated because we intend to use the *Beep* driver to play the notes. The *Beep* driver needs to be able to handle multiple users (that might be connected to the same system), each one playing their own sounds. This is why when asked to play a note, the *Beep* driver plays in the context of the caller's session. If we create the thread in the *System* process, which is always part of session zero, we will not hear any sound, because session 0 is not an interactive user session.

This means we need to create our thread in the context of some process running under the caller's session - Using the caller's process directly

(NtCurrentProcess) is the simplest way to get it working. You may frown at this, and rightly so, because the first process calling the driver to play something is going to have to host that thread for the lifetime of the driver. This has an unintended side effect: the process will not die. Even if it may seem to terminate, it will still show up in *Task Manager* with our thread being the single thread still keeping the process alive. We'll find a more elegant solution later in this chapter.

Yet another consequence of this arrangement is that we only handle one session - the first one where one of its processes happens to call the driver. We'll fix that as well later on.

The thread created starts running the `PlayMelody` function - a static function in the `PlaybackState` class. Callbacks must be global or static functions (because they are directly C function pointers), but in this case we would like to access the members of this instance of `PlaybackState`. The common trick is to pass the `this` pointer as the thread argument, and the callback simply invokes an instance method using this pointer:

```
// static function
void PlaybackState::PlayMelody(PVOID context) {
    ((PlaybackState*)context)->PlayMelody();
}
```

Now the instance method `PlaybackState::PlayMelody` has full access to the object's members.



There is another way to invoke the instance method without going through the intermediate static by using C++ *lambda functions*, as non-capturing lambdas are directly convertible to C function pointers:

```
IoCreateSystemThread(..., [](auto param) {
    ((PlaybackState*)param)->PlayMelody();
}, this);
```

The first order of business in the new thread is to obtain a pointer to the *Beep* device using `IoGetDeviceObjectPointer`:

```
#include <ntddbeep.h>

void PlaybackState::PlayMelody() {
    PDEVICE_OBJECT beepDevice;
    UNICODE_STRING beepDeviceName = RTL_CONSTANT_STRING(DD_BEEP_DEVICE_NAME_U);
    PFILE_OBJECT beepFileObject;
    auto status = IoGetDeviceObjectPointer(&beepDeviceName, GENERIC_WRITE,
        &beepFileObject, &beepDevice);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "Failed to locate beep device (0x%X)\n",
            status));
    }
    return;
}
```

The *Beep* device name is `\Device\Beep` as we've seen in chapter 2. Conveniently, the provided header *ntddbeep.h* declares everything we need in order to work with the device, such as the `DD_BEEP_DEVICE_NAME_U` macro that defines the Unicode name.

At this point, the thread should loop around while it has notes to play and has not been instructed to terminate. This is where the semaphore and the event come in. The thread must wait until one of them is signaled. If it's the event, it should break out of the loop. If it's the semaphore, it means the semaphore's count is greater than zero, which in turn means the list of notes is not empty:

```
PVOID objects[] = { &m_counter, &m_stopEvent };
IO_STATUS_BLOCK ioStatus;
BEEP_SET_PARAMETERS params;

for (;;) {
    status = KeWaitForMultipleObjects(2, objects, WaitAny, Executive,
        KernelMode, FALSE, nullptr, nullptr);
    if (status == STATUS_WAIT_1) {
        KdPrint((DRIVER_PREFIX "Stop event signaled. Exiting thread...\n"));
        break;
    }

    KdPrint((DRIVER_PREFIX "Semaphore count: %u\n",
        KeReadStateSemaphore(&m_counter)));
}
```

The required call is to `KeWaitForMultipleObjects` with the event and semaphore. They are put in an array, since this is the requirement for `KeWaitForMultipleObjects`. If the returned status is `STATUS_WAIT_1` (which is the same as `STATUS_WAIT_0 + 1`), meaning index number 1 is the signaled object, the loop is exited with a `break` instruction.

Now we need to extract the next note to play:

```

PLIST_ENTRY link;
{
    Locker locker(m_lock);
    link = RemoveHeadList(&m_head);
    NT_ASSERT(link != &m_head);
}
auto note = CONTAINING_RECORD(link, FullNote, Link);
KdPrint((DRIVER_PREFIX "Playing note Freq: %u Dur: %u Rep: %u Delay: %u\n",
    note->Frequency, note->Duration, note->Repeat, note->Delay));

```

We remove the head item from the list, and doing so under the fast mutex' protection. The assert ensures we are in a consistent state - remember that removing an item from an empty list returns the pointer to its head.

The actual FullNote pointer is retrieved with the help of the CONTAINING_RECORD macro, that moves the LIST_ENTRY pointer we received from RemoveHeadList to the containing FullNode that we are actually interested in.

The next step is to handle the note. If the note's frequency is zero, let's consider that as a "silence time" with the length provided by Delay:

```

if (note->Frequency == 0) {
    //
    // just do a delay
    //
    NT_ASSERT(note->Duration > 0);
    LARGE_INTEGER interval;
    interval.QuadPart = -10000LL * note->Duration;
    KeDelayExecutionThread(KernelMode, FALSE, &interval);
}

```

KeDelayExecutionThread is the rough equivalent of the Sleep/SleepEx APIs from user-mode. Here is its declaration:

```

NTSTATUS KeDelayExecutionThread (
    _In_ KPROCESSOR_MODE WaitMode,
    _In_ BOOLEAN Alertable,
    _In_ PLARGE_INTEGER Interval);

```

We've seen all these parameters as part of the wait functions. The most common invocation is with KernelMode and FALSE for *WaitMode* and *Alertable*, respectively. The interval is the most important parameter, where negative values mean relative wait in 100nsec units. Converting from milliseconds means multiplying by -10000, which is what you see in the above code.

If the frequency in the note is not zero, then we need to call the *Beep* driver with proper IRP. We already know that we need the IOCTL_BEEP_SET control code (defined in *ntddbeep.h*) and the BEEP_SET_PARAMETERS structure. All we need to do is build an IRP with the correct information using *IoBuildDeviceIoControlRequest*, and send it to the beep device with *IoCallDriver*:

```
else {
    params.Duration = note->Duration;
    params.Frequency = note->Frequency;
    int count = max(1, note->Repeat);

    KEVENT doneEvent;
    KeInitializeEvent(&doneEvent, NotificationEvent, FALSE);

    for (int i = 0; i < count; i++) {
        auto irp = IoBuildDeviceIoControlRequest(IOCTL_BEEP_SET, beepDevice,
            &params, sizeof(params),
            nullptr, 0, FALSE, &doneEvent, &ioStatus);
        if (!irp) {
            KdPrint((DRIVER_PREFIX "Failed to allocate IRP\n"));
            break;
        }

        status = IoCallDriver(beepDevice, irp);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "Beep device playback error (0x%X)\n",
                status));
            break;
        }
        if (status == STATUS_PENDING) {
            KeWaitForSingleObject(&doneEvent, Executive, KernelMode,
                FALSE, nullptr);
        }
    }
}
```

We loop around based on the Repeat member (which is usually 1). Then the IRP_MJ_DEVICE_CONTROL IRP is built with *IoBuildDeviceIoControlRequest*, supplying the frequency to play and the duration. Then, *IoCallDriver* is invoked with the *Beep* device pointer we obtained earlier, and the IRP. Unfortunately (or fortunately, depending on your perspective), the *Beep* driver just starts the operation, but does not wait for it to finish. It might (and in fact, always) returns STATUS_PENDING from the *IoCallDriver* call, which means the operation is not yet complete (the actual playing has not yet begun). Since we don't have anything else to do until then, the doneEvent event provided to *IoBuildDeviceIoControlRequest* is signaled automatically by the I/O manager when the operation completes - so we wait on the event.

Now that the sound is playing, we have to wait for the duration of that note with `KeDelayExecutionThread`:

```
LARGE_INTEGER delay;
delay.QuadPart = -100000LL * note->Duration;
KeDelayExecutionThread(KernelMode, FALSE, &delay);
```

Finally, if `Repeat` is greater than one, then we might need to wait between plays of the same note:

```
    // perform the delay if specified,
    // except for the last iteration
    //
    if (i < count - 1 && note->Delay != 0) {
        delay.QuadPart = -100000LL * note->Delay;
        KeDelayExecutionThread(KernelMode, FALSE, &delay);
    }
}
```

At this point, the note data can be freed (or just returned to the lookaside list) and the code loops back to wait for the availability of the next note:

```
    ExFreeToPagedLookasideList(&m_lookaside, note);
}
```

The loop continues until the thread is instructed to stop by signaling `stopEvent`, at which point it breaks from the infinite loop and cleans up by dereferencing the file object obtained from `IoGetDeviceObjectPointer`:

```
    ObDereferenceObject(beepFileObject);
}
```

Here is the entire thread function for convenience (comments and `KdPrint` removed):

```
void PlaybackState::PlayMelody() {
    PDEVICE_OBJECT beepDevice;
    UNICODE_STRING beepDeviceName = RTL_CONSTANT_STRING(DD_BEEP_DEVICE_NAME_U);
    PFILE_OBJECT beepFileObject;
    auto status = IoGetDeviceObjectPointer(&beepDeviceName, GENERIC_WRITE,
        &beepFileObject, &beepDevice);
    if (!NT_SUCCESS(status)) {
        return;
    }
}
```

```

PVOID objects[] = { &m_counter, &m_stopEvent };
IO_STATUS_BLOCK ioStatus;
BEEP_SET_PARAMETERS params;

for (;;) {
    status = KeWaitForMultipleObjects(2, objects, WaitAny, Executive,
        KernelMode, FALSE, nullptr, nullptr);
    if (status == STATUS_WAIT_1) {
        break;
    }

    PLIST_ENTRY link;
    {
        Locker locker(m_lock);
        link = RemoveHeadList(&m_head);
        NT_ASSERT(link != &m_head);
    }
    auto note = CONTAINING_RECORD(link, FullNote, Link);
    if (note->Frequency == 0) {
        NT_ASSERT(note->Duration > 0);
        LARGE_INTEGER interval;
        interval.QuadPart = -100000LL * note->Duration;
        KeDelayExecutionThread(KernelMode, FALSE, &interval);
    }
    else {
        params.Duration = note->Duration;
        params.Frequency = note->Frequency;
        int count = max(1, note->Repeat);

        KEVENT doneEvent;
        KeInitializeEvent(&doneEvent, SynchronizationEvent, FALSE);

        for (int i = 0; i < count; i++) {
            auto irp = IoBuildDeviceIoControlRequest(IOCTL_BEEP_SET,
                beepDevice, &params, sizeof(params),
                nullptr, 0, FALSE, &doneEvent, &ioStatus);
            if (!irp) {
                break;
            }
            NT_ASSERT(irp->UserEvent == &doneEvent);

            status = IoCallDriver(beepDevice, irp);
            if (!NT_SUCCESS(status)) {
                break;
            }
            if (status == STATUS_PENDING) {
                KeWaitForSingleObject(&doneEvent, Executive,
                    KernelMode, FALSE, nullptr);
            }

            LARGE_INTEGER delay;
            delay.QuadPart = -100000LL * note->Duration;
            KeDelayExecutionThread(KernelMode, FALSE, &delay);

            if (i < count - 1 && note->Delay != 0) {
                delay.QuadPart = -100000LL * note->Delay;
                KeDelayExecutionThread(KernelMode, FALSE, &delay);
            }
        }
    }
}

```

```

        ExFreeToPagedLookasideList(&m_lookaside, note);
    }
    ObDereferenceObject(beepFileObject);
}

```

The last piece of the puzzle is the `PlaybackState::Stop` method that signals the thread to exit:

```

void PlaybackState::Stop() {
    if (m_hThread) {
        //
        // signal the thread to stop
        //
        KeSetEvent(&m_stopEvent, 2, FALSE);

        //
        // wait for the thread to exit
        //
        PVOID thread;
        auto status = ObReferenceObjectByHandle(m_hThread, SYNCHRONIZE,
            *PsThreadType, KernelMode, &thread, nullptr);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "ObReferenceObjectByHandle error (0x%X)\n",
                status));
        }
        else {
            KeWaitForSingleObject(thread, Executive, KernelMode, FALSE, nullptr\
);
            ObDereferenceObject(thread);
        }
        ZwClose(m_hThread);
        m_hThread = nullptr;
    }
}

```

If the thread exists (`m_hThread` is non-NULL), then we set the event (`KeSetEvent`). Then we wait for the thread to actually terminate. This is technically unnecessary because the thread was created with `IoCreateSystemThread`, so there is no danger the driver is unloaded prematurely. Still, it's worthwhile showing how to get the pointer to the thread object given a handle (since `KeWaitForSingleObject` requires an object). It's important to remember to call `ObDereferenceObject` once we don't need the pointer anymore, or the thread object will remain alive forever (keeping its process and other resources alive as well).

Client Code

Here are some examples for invoking the driver (error handling omitted):

```

#include <Windows.h>
#include <stdio.h>
#include "..\KMelody\MelodyPublic.h"

int main() {
    HANDLE hDevice = CreateFile(MELODY_SYMLINK, GENERIC_WRITE, 0,
        nullptr, OPEN_EXISTING, 0, nullptr);

    Note notes[10];
    for (int i = 0; i < _countof(notes); i++) {
        notes[i].Frequency = 400 + i * 30;
        notes[i].Duration = 500;
    }
    DWORD bytes;
    DeviceIoControl(hDevice, IOCTL_MELODY_PLAY, notes, sizeof(notes),
        nullptr, 0, &bytes, nullptr);

    for (int i = 0; i < _countof(notes); i++) {
        notes[i].Frequency = 1200 - i * 100;
        notes[i].Duration = 300;
        notes[i].Repeat = 2;
        notes[i].Delay = 300;
    }
    DeviceIoControl(hDevice, IOCTL_MELODY_PLAY, notes, sizeof(notes),
        nullptr, 0, &bytes, nullptr);

    CloseHandle(hDevice);
    return 0;
}

```

I recommend you build the driver and the client and test them. The project names are *KMelody* and *Melody* in the solution for this chapter. Build your own music!



1. Replace the call to `IoCreateSystemThread` with `PsCreateSystemThread` and make the necessary adjustments.
2. Replace the lookaside list API with the newer API.

Invoking System Services

System Services (system calls) are normally invoked indirectly from user mode code. For example, calling the Windows `CreateFile` API in user mode invokes `NtCreateFile` from *NtDll.Dll*, which is a system call. This call traverses the

user/kernel boundary, eventually calling the “real” `NtCreateFile` implementation within the executive.

We already know that drivers can invoke system calls as well, using the `Nt` or the `Zw` variant (which sets the previous execution mode to `KernelMode` before invoking the system call). Some of these system calls are fully documented in the driver kit, such as `NtCreateFile/ZwCreateFile`. Others, however, are not documented or sometimes partially documented.

For example, enumerating processes in the system is fairly easy to do from user-mode - in fact, there are several APIs one can use for this purpose. They all invoke the `NtQuerySystemInformation` system call, which is not officially documented in the WDK. Ironically, it’s provided in the user-mode header *Winternl.h* like so:

```
NTSTATUS NtQuerySystemInformation (  
    IN SYSTEM_INFORMATION_CLASS SystemInformationClass,  
    OUT PVOID SystemInformation,  
    IN ULONG SystemInformationLength,  
    OUT PULONG ReturnLength OPTIONAL);
```

The macros `IN` and `OUT` expand to nothing. These were used in the old days before SAL was invented to provide some semantics for developers. For some reason, *Winternl.h* uses these macros rather than the modern SAL annotations.

We can copy this definition and tweak it a bit by turning it into its `Zw` variant, more suitable for kernel callers. The `SYSTEM_INFORMATION_CLASS` enumeration and associated data structures are the real data we’re after. Some values are provided in user-mode and/or kernel-mode headers. Most of the values have been “reversed engineered” and can be found in open source projects, such as [Process Hacker](#). Although these APIs might not be officially documented, they are unlikely to change as Microsoft’s own tools depend on many of them.

If the API question only exists in certain Windows versions, it’s possible to query dynamically for the existence of a kernel API with `MmGetSystemRoutineAddress`:

```
PVOID MmGetSystemRoutineAddress (_In_ PUNICODE_STRING SystemRoutineName);
```

You can think of `MmGetSystemRoutineAddress` as the kernel-mode equivalent of the user-mode `GetProcAddress` API.

Another very useful API is `NtQueryInformationProcess`, also defined in *Winternl.h*:

```
NTAPI NtQueryInformationProcess (
    IN HANDLE ProcessHandle,
    IN PROCESSINFOCLASS ProcessInformationClass,
    OUT PVOID ProcessInformation,
    IN ULONG ProcessInformationLength,
    OUT PULONG ReturnLength OPTIONAL);
```

Curiously enough, the kernel-mode headers provide many of the `PROCESSINFOCLASS` enumeration values, along with their associated data structures, but not the definition of this system call itself. Here is a partial set of values for `PROCESSINFOCLASS`:

```
typedef enum _PROCESSINFOCLASS {
    ProcessBasicInformation = 0,
    ProcessDebugPort = 7,
    ProcessWow64Information = 26,
    ProcessImageFileName = 27,
    ProcessBreakOnTermination = 29
} PROCESSINFOCLASS;
```



A more complete list is available in *ntddk.h*. A full list is available within the *Process Hacker* project.

The following example shows how to query the current process image file name. `ProcessImageFileName` seems to be the way to go, and it expects a `UNICODE_STRING` as the buffer:

```
ULONG size = 1024;
auto buffer = ExAllocatePoolWithTag(PagedPool, size, DRIVER_TAG);
auto status = ZwQueryInformationProcess(NtCurrentProcess(),
    ProcessImageFileName, buffer, size, nullptr);
if(NT_SUCCESS(status)) {
    auto name = (UNICODE_STRING*)buffer;
    // do something with name...
}
ExFreePool(buffer);
```

Example: Enumerating Processes

The *EnumProc* driver shows how to call `ZwQuerySystemInformation` to retrieve the list of running processes. `DriverEntry` calls the `EnumProcess` function that does all the work and dumps information using simple `DbgPrint` calls. Then `DriverEntry` returns an error so the driver is unloaded.

First, we need the definition of `ZwQuerySystemInformation` and the required enum value and structure which we can copy from *Winternl.h*:

```
#include <ntddk.h>

// copied from <Winternl.h>
enum SYSTEM_INFORMATION_CLASS {
    SystemProcessInformation = 5,
};

typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    UCHAR Reserved1[48];
    UNICODE_STRING ImageName;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    PVOID Reserved2;
    ULONG HandleCount;
    ULONG SessionId;
    PVOID Reserved3;
    SIZE_T PeakVirtualSize;
    SIZE_T VirtualSize;
    ULONG Reserved4;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    PVOID Reserved5;
    SIZE_T QuotaPagedPoolUsage;
    PVOID Reserved6;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
    SIZE_T PrivatePageCount;
    LARGE_INTEGER Reserved7[6];
} SYSTEM_PROCESS_INFORMATION, * PSYSTEM_PROCESS_INFORMATION;

extern "C" NTSTATUS ZwQuerySystemInformation(
    SYSTEM_INFORMATION_CLASS info,
    PVOID buffer,
    ULONG size,
    PULONG len);
```

Notice there are lots of “resevred” members in `SYSTEM_PROCESS_INFORMATION`. We’ll manage with what we get, but you can find the full data structure in the

Process Hacker project.

EnumProc starts by querying the number of bytes needed by calling ZwQuerySystemInformation with a null buffer and zero size, getting the last parameter as the required size:

```
void EnumProcesses() {
    ULONG size = 0;
    ZwQuerySystemInformation(SystemProcessInformation, nullptr, 0, &size);
    size += 1 << 12;    // 4KB, just to make sure the next call succeeds
}
```

We want to allocate some more in case new processes are created between this call and the next “real” call. We can write the code in a more robust way and have a loop that queries until the size is large enough, but the above solution is robust enough for most purposes.

Next, we allocate the required buffer and make the call again, this time with the real buffer:

```
auto buffer = ExAllocatePoolWithTag(PagedPool, size, 'cprP');
if (!buffer)
    return;

if (NT_SUCCESS(ZwQuerySystemInformation(SystemProcessInformation,
    buffer, size, nullptr))) {
```

if the call succeeds, we can start iterating. The returned pointer is to the first process, where the next process is located NextEntryOffset bytes from this offset. The enumeration ends when NextEntryOffset is zero:

```
ULONG count = 0;
for (;;) {
    DbgPrint("PID: %u Session: %u Handles: %u Threads: %u Image: %wZ\n",
        HandleToULong(info->UniqueProcessId),
        info->SessionId, info->HandleCount,
        info->NumberOfThreads, info->ImageName);
    count++;
    if (info->NextEntryOffset == 0)
        break;

    info = (SYSTEM_PROCESS_INFORMATION*)((PUCHAR)info + info->NextEntryOffset);
}
DbgPrint("Total Processes: %u\n", count);
```

We output some of the details provided in the SYSTEM_PROCESS_INFORMATION structure and count the number of processes while we’re at it. The only thing

left to do in this simple example is to clean up:

```
    }  
    ExFreePool(buffer);  
}
```

As mentioned, `DriverEntry` is simple:

```
extern "C" NTSTATUS  
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {  
    UNREFERENCED_PARAMETER(DriverObject);  
    UNREFERENCED_PARAMETER(RegistryPath);  
  
    EnumProcesses();  
    return STATUS_UNSUCCESSFUL;  
}
```

Given this knowledge, we can make the *KMelody* driver a bit better by creating our thread in a *Csrss.exe* process for the current session, instead of the first client process that comes in. This is better, since *Csrss* always exists, and is in fact a *critical process* - one that if killed for whatever reason, causes the system to crash.

Killing *Csrss* is not easy, since it's a protected process starting with Windows 8.1, but kernel code can certainly do that.



1. Modify the *KMelody* driver to create the thread in a *Csrss* process for the current session. Search for *Csrss* with `ZwQuerySystemInformation` and create the thread in that process.
2. Add support for multiple sessions, where there is one playback thread per session. Hint: call `ZwQueryInformationProcess` with `ProcessSessionId` to find out the session a process is part of. Manage a list of `PlaybackState` objects, one for each session. You can also use the undocumented (but exported) `PsGetCurrentProcessSessionId` API.

Summary

In this chapter, we were introduced to some programming techniques that are useful in many types of drivers. We're not done with these techniques - there

will be more in chapter 11. But for now, we can begin using some kernel-provided notifications, starting with Process and Thread notifications in the next chapter.

Chapter 9: Process and Thread Notifications

One of the powerful mechanisms available for kernel drivers is the ability to be notified when certain important events occur. In this chapter, we'll look into some of these events, namely process creation and destruction, thread creation and destruction, and image loads.

In this chapter:

- **Process Notifications**
 - **Implementing Process Notifications**
 - **Providing Data to User Mode**
 - **Thread Notifications**
 - **Image Load Notifications**
 - **Remote Thread Detection**
-

Process Notifications

Whenever a process is created or destroyed, interested drivers can be notified by the kernel of that fact. This allows drivers to keep track of processes, possibly associating some data with these processes. At the very minimum, these allow drivers to monitor process creation/destruction in real-time. By “real-time” I mean that the notifications are sent “in-line”, as part of process creation; the driver cannot miss any processes that may be created and destroyed quickly.

For process creation, drivers also have the power to stop the process from being fully created, returning an error to the caller that initiated process creation. This kind of power can only be directly achieved in kernel mode.

Windows provides other mechanisms for being notified when processes are created or destroyed. For example, using *Event Tracing for Windows* (ETW), such notifications can be

received by a user-mode process (running with elevated privileges). However, there is no way to prevent a process from being created. Furthermore, ETW has an inherent notification delay of about 1-3 seconds (it uses internal buffers for performance reasons), so a short-lived process may exit before the creation notification arrives. Opening a handle to the created process at that time would no longer be possible.

The main API for registering for process notifications is `PsSetCreateProcessNotifyRoutineEx`, defined like so:

```
NTSTATUS PsSetCreateProcessNotifyRoutineEx (  
    _In_ PCREATE_PROCESS_NOTIFY_ROUTINE_EX NotifyRoutine,  
    _In_ BOOLEAN Remove);
```



There is currently a system-wide limit of 64 registrations, so it's theoretically possible for the registration function to fail.

The first argument is the driver's callback routine, having the following prototype:

```
void ProcessNotifyCallback(  
    _Inout_ PEPROCESS Process,  
    _In_ HANDLE ProcessId,  
    _Inout_opt_ PPS_CREATE_NOTIFY_INFO CreateInfo);
```

The second argument to `PsSetCreateProcessNotifyRoutineEx` indicates whether the driver is registering or unregistering the callback (`FALSE` indicates the former). Typically, a driver will call this API with `FALSE` in its `DriverEntry` routine and call the same API with `TRUE` in its `Unload` routine.

The parameters to the process notification routine are as follows:

- *Process* - the process object of the newly created process, or the process being destroyed.
- *Process Id* - the unique process ID of the process. Although it's declared with type `HANDLE`, it's in fact an ID.
- *CreateInfo* - a structure that contains detailed information on the process being created. If the process is being destroyed, this argument is `NULL`.

For process creation, the driver's callback routine is executed by the creating thread (running as part of the creating process). For process exit, the callback is executed by the last thread to exit the process. In both cases, the callback is called inside a critical region (where normal kernel APCs are disabled).

Starting with Windows 10 version 1607, there is another function for process notifications: `PsSetCreateProcessNotifyRoutineEx2`. This “extended” function sets up a callback similar to the previous one, but the callback is also invoked for *Pico processes*. Pico processes are those used to host Linux processes for the *Windows Subsystem for Linux (WSL)* version 1. If a driver is interested in such processes, it must register with the extended function.

A driver using these callbacks must have the `IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY` flag in its *Portable Executable* (PE) image header. Without it, the call to the registration function returns `STATUS_ACCESS_DENIED` (unrelated to driver test signing mode). Currently, Visual Studio does not provide UI for setting this flag. It must be set in the linker command-line options with `/integritycheck`. Figure 9-1 shows the project properties where this setting is specified.

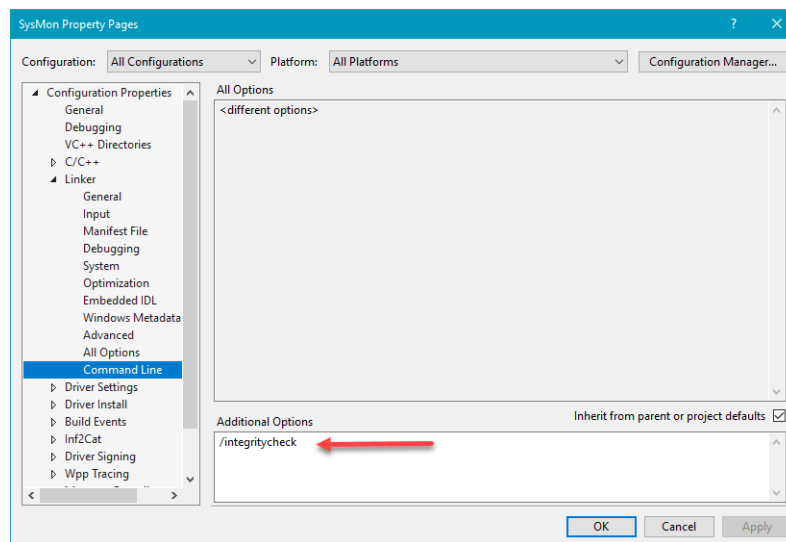


Figure 9-1: /integritycheck linker switch in Visual Studio

The data structure provided for process creation is defined like so:

```
typedef struct _PS_CREATE_NOTIFY_INFO {  
    _In_ SIZE_T Size;  
    union {
```

```

    _In_ ULONG Flags;
    struct {
        _In_ ULONG FileOpenNameAvailable : 1;
        _In_ ULONG IsSubsystemProcess : 1;
        _In_ ULONG Reserved : 30;
    };
};
_In_ HANDLE ParentProcessId;
_In_ CLIENT_ID CreatingThreadId;
_Inout_ struct _FILE_OBJECT *FileObject;
_In_ PCUNICODE_STRING ImageFileName;
_In_opt_ PCUNICODE_STRING CommandLine;
_Inout_ NTSTATUS CreationStatus;
} PS_CREATE_NOTIFY_INFO, *PPS_CREATE_NOTIFY_INFO;

```

Here is a description of the important fields in this structure:

- *CreatingThreadId* - a combination of thread and process Id of the creator of the process.
- *ParentProcessId* - the parent process ID (not a handle). This process is usually the same as provided by `CreateThreadId.UniqueProcess`, but may be different, as it's possible, as part of process creation, to pass in a different parent process to inherit certain properties from. See the user-mode documentation for `UpdateProcThreadAttribute` with the `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS` attribute.
- *ImageFileName* - the image file name of the executable, available if the flag `FileOpenNameAvailable` is set.
- *CommandLine* - the full command line used to create the process. Note that in some cases it may be `NULL`.
- *IsSubsystemProcess* - this flag is set if this process is a Pico process. This can only happen if the driver registered using `PsSetCreateProcessNotifyRoutineEx2`.
- *CreationStatus* - this is the status that would return to the caller. It's set to `STATUS_SUCCESS` when the callback is invoked. This is where the driver can stop the process from being created by placing some failure status in this member (e.g. `STATUS_ACCESS_DENIED`). if the driver fails the creation, subsequent drivers that may have set up their own callbacks will not be called.

Implementing Process Notifications

To demonstrate process notifications, we'll build a driver that gathers information on process creation and destruction and allow this information to

be consumed by a user-mode client. This is similar to tools such as *Process Monitor* and *SysMon* from *Sysinternals*, which use process and thread notifications for reporting process and thread activity. During the course of implementing this driver, we'll leverage some of the techniques we learned in previous chapters.

Our driver name is going to be *SysMon* (unrelated to the *SysMon* tool). It will store all process creation/destruction information in a linked list. Since this linked list may be accessed concurrently by multiple threads, we need to protect it with a mutex or a fast mutex; we'll go with fast mutex, as it's slightly more efficient.

The data we gather will eventually find its way to user mode, so we should declare common structures that the driver produces and a user-mode client consumes. We'll add a common header file named *SysMonPublic.h* to the driver project and define a few structures. We start with a common header for all information structures we need to collect:

```
enum class ItemType : short {
    None,
    ProcessCreate,
    ProcessExit
};

struct ItemHeader {
    ItemType Type;
    USHORT Size;
    LARGE_INTEGER Time;
};
```



The `ItemType` enum defined above uses the C++ 11 *scoped enum* feature, where enum values have a scope (`ItemType` in this case). These enums can also have a non-int size - `short` in the example. If you're using C, you can use classic enums, or even `#defines` if you prefer.

The `ItemHeader` structure holds information common to all event types: the type of the event, the time of the event (expressed as a 64-bit integer), and the size of the payload. The size is important, as each event has its own information. If we later wish to pack an array of these events and (say) provide them to a user-mode client, the client needs to know where each event ends and the next one begins.

Once we have this common header, we can derive other data structures for particular events. Let's start with the simplest - process exit:

```
struct ProcessExitInfo : ItemHeader {  
    ULONG ProcessId;  
    ULONG ExitCode;  
};
```

For process exit event, there is just one interesting piece of information (besides the header and the thread ID) - the exit status (code) of the process. This is normally the value returned from a user-mode main function.



If you're using C, then inheritance is not available to you. However, you can simulate it by having the first member be of type `ItemHeader` and then adding the specific members; The memory layout is the same.

```
struct ProcessExitInfo {  
    ItemHeader Header;  
    ULONG ProcessId;  
};
```

The type used for a process ID is `ULONG` - process IDs (and thread IDs) cannot be larger than 32-bit. `HANDLE` is not a good idea, as user mode may be confused by it. Also, `HANDLE` has a different size in a 32-bit process as opposed to a 64-bit process, so it's best to avoid "bitness"-affected members. If you're familiar with user-mode programming, `DWORD` is a common typedef for a 32-bit unsigned integer. It's not used here because `DWORD` is not defined in the WDK headers. Although it's pretty easy to define it explicitly, it's simpler just to use `ULONG`, which means the same thing and is defined in user-mode and kernel-mode headers.

Since we need to store every such structure as part of a linked list, each data structure must contain a `LIST_ENTRY` instance that points to the next and previous items. Since these `LIST_ENTRY` objects should not be exposed to user-mode, we will define extended structures containing these entries in a different file, that is not shared with user-mode.

There are several ways to define a "bigger" structure to hold the `LIST_ENTRY`. One way is to create templated type that has a `LIST_ENTRY` at the beginning (or end) like so:

```
template<typename T>
struct FullItem {
    LIST_ENTRY Entry;
    T Data;
};
```

The layout of `FullItem<T>` is shown in figure 9-2.

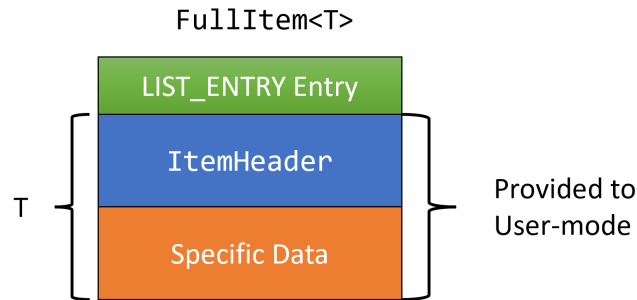


Figure 9-2: `FullItem<T>` layout

A templated class is used to avoid creating a multitude of types, one for each specific event type. For example, we could create the following structure specifically for a process exit event:

```
struct FullProcessExitInfo {
    LIST_ENTRY Entry;
    ProcessExitInfo Data;
};
```

We could even inherit from `LIST_ENTRY` and then just add the `ProcessExitInfo` structure. But this is not elegant, as our data has nothing to do with `LIST_ENTRY`, so inheriting from it is artificial and should be avoided.

The `FullItem<T>` type saves the hassle of creating these individual types.



IF you're using C, then templates are not available, and you must use the above structure approach. I'm not going to mention C again in this chapter - there is always a workaround that can be used if you have to.

Another way to accomplish something similar, without using templates is by using a union to hold on to all the possible variants. For example:

```
struct ItemData : ItemHeader {
    union {
        ProcessCreateInfo ProcessCreate;    // TBD
        ProcessExitInfo ProcessExit;
    };
};
```

Then we just extend the list of data members in the union. The full item would be just a simple extension:

```
struct FullItem {
    LIST_ENTRY Entry;
    ItemData Data;
};
```

The rest of the code uses the first option (with the template). The reader is encouraged to try the second option.

The head of our linked list must be stored somewhere. We'll create a data structure that will hold all the global state of the driver, instead of creating separate global variables. Here is the definition of our structure (in *Globals.h* in the sample code for this chapter):

```
#include "FastMutex.h"

struct Globals {
    void Init(ULONG maxItems);
    bool AddItem(LIST_ENTRY* entry);
    LIST_ENTRY* RemoveItem();

private:
    LIST_ENTRY m_ItemsHead;
    ULONG m_Count;
    ULONG m_MaxCount;
    FastMutex m_Lock;
};
```

The FastMutex type used is the same one we developed in chapter 6.

Init is used to initialize the data members of the structure. Here is its implementation (in *Globals.cpp*):

```
void Globals::Init(ULONG maxCount) {
    InitializeListHead(&m_ItemsHead);
    m_Lock.Init();
    m_Count = 0;
    m_MaxCount = maxCount;
}
```

m_MaxCount holds the maximum number of elements in the linked list. This will be used to prevent the list from growing arbitrarily large if a client does not request data for a while. m_Count holds the current number of items in the list. The list itself is initialized with the normal InitializeListHead API. Finally, the fast mutex is initialized by invoking its own Init method as implemented in chapter 6.

The DriverEntry Routine

The DriverEntry for the *SysMon* driver is similar to the one in the *Zero* driver from chapter 7. We have to add process notification registration and proper initialization of our Globals object:

```
// in SysMon.cpp
Globals g_State;

extern "C"
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    auto status = STATUS_SUCCESS;

    PDEVICE_OBJECT DeviceObject = nullptr;
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\sysmon");
    bool symLinkCreated = false;

    do {
        UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\sysmon");
        status = IoCreateDevice(DriverObject, 0, &devName,
            FILE_DEVICE_UNKNOWN, 0, TRUE, &DeviceObject);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "failed to create device (0x%08X)\\n",
                status));
            break;
        }
        DeviceObject->Flags |= DO_DIRECT_IO;

        status = IoCreateSymbolicLink(&symLink, &devName);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "failed to create sym link (0x%08X)\\n",
                status));
            break;
        }
        symLinkCreated = true;

        status = PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, FALSE);
```

```

        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX
                "failed to register process callback (0x%08X)\n",
                status));
            break;
        }
    } while (false);

    if (!NT_SUCCESS(status)) {
        if (symLinkCreated)
            IoDeleteSymbolicLink(&symLink);
        if (DeviceObject)
            IoDeleteDevice(DeviceObject);
        return status;
    }

    g_State.Init(10000);           // hard-coded limit for now

    DriverObject->DriverUnload = SysMonUnload;
    DriverObject->MajorFunction[IRP_MJ_CREATE] =
        DriverObject->MajorFunction[IRP_MJ_CLOSE] = SysMonCreateClose;
    DriverObject->MajorFunction[IRP_MJ_READ] = SysMonRead;

    return status;
}

```

The device object's flags are adjusted to use Direct I/O for read/write operations (DO_DIRECT_IO). The device is created as exclusive, so that only a single client can exist to the device. This makes sense, otherwise multiple clients might be getting data from the device, which would mean each client getting parts of the data. In this case, I decided to prevent that by creating the device as exclusive (TRUE value in the second to last argument). We'll use the read dispatch routine to return event information to a client.

The create and close dispatch routines are handled in the simplest possible way - just completing them successfully, with the help of CompleteRequest we have encountered before:

```

NTSTATUS CompleteRequest(PIRP Irp,
    NTSTATUS status = STATUS_SUCCESS, ULONG_PTR info = 0) {
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = info;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

NTSTATUS SysMonCreateClose(PDEVICE_OBJECT, PIRP Irp) {
    return CompleteRequest(Irp);
}

```

Handling Process Exit Notifications

The process notification function in the code above is `OnProcessNotify` and has the prototype outlined earlier in this chapter. This callback handles process creation and exit. Let's start with process exit, as it's much simpler than process creation (as we shall soon see). The basic outline of the callback is as follows:

```
void OnProcessNotify(PEPROCESS Process, HANDLE ProcessId,
    PPS_CREATE_NOTIFY_INFO CreateInfo) {
    if (CreateInfo) {
        // process create
    }
    else {
        // process exit
    }
}
```

For process exit we have just the process ID we need to save, along with the header data common to all events. First, we need to allocate storage for the full item representing this event:

```
auto info = (FullItem<ProcessExitInfo>*)ExAllocatePoolWithTag(PagedPool,
    sizeof(FullItem<ProcessExitInfo>), DRIVER_TAG);
if (info == nullptr) {
    KdPrint((DRIVER_PREFIX "failed allocation\n"));
    return;
}
```

If the allocation fails, there is really nothing the driver can do, so it just returns from the callback.

Now it's time to fill the generic information: time, item type and size, all of which are easy to get:

```
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Type = ItemType::ProcessExit;
item.Size = sizeof(ProcessExitInfo);
item.ProcessId = HandleToULong(ProcessId);
item.ExitCode = PsGetProcessExitStatus(Process);

PushItem(&info->Entry);
```

First, we dig into the data item itself (bypassing the `LIST_ENTRY`) with the `item` variable. Next, we fill the header information: The item type is well-known, since we are in the branch handling a process exit notification; the time can be obtained with `KeQuerySystemTimePrecise` that returns the current system time (UTC, not local time) as a 64-bit integer counting from January 1, 1601 at

midnight Universal Time. Finally, the item size is constant and is the size of the user-facing data structure (not the size of the `FullItem<ProcessExitInfo>`).



Notice the `item` variable is a reference to the data; without the reference (`&`), a copy would have been created, which is **not** what we want.



The `KeQuerySystemTimePrecise` API is available starting with Windows 8. For earlier versions, the `KeQuerySystemTime` API should be used instead.

The specific data for a process exit event consists of the process ID and the exit code. The process ID is provided directly by the callback itself. The only thing to do is call `HandleToULong` so the correct cast is used to turn a `HANDLE` value into an unsigned 32-bit integer. The exit code is not given directly, but it's easy to retrieve with `PsGetProcessExitStatus`:

```
NTSTATUS PsGetProcessExitStatus(_In_ PEPROCESS Process);
```

All that's left to do now is add the new item to the end of our linked list. For this purpose, we'll define and implement a function named `AddItem` in the `Globals` class:

```
void Globals::AddItem(LIST_ENTRY* entry) {
    Locker locker(m_Lock);
    if (m_Count == m_MaxCount) {
        auto head = RemoveHeadList(&m_ItemsHead);
        ExFreePool(CONTAINING_RECORD(head,
            FullItem<ItemHeader>, Entry));
        m_Count--;
    }

    InsertTailList(&m_ItemsHead, entry);
    m_Count++;
}
```

`AddItem` uses the `Locker<T>` we saw in earlier chapters to acquire the fast mutex (and release it when the variable goes out of scope) before manipulating the linked list. Remember to set the C++ standard to C++ 17 at least in the

project's properties so that Locker can be used without explicitly specifying the type it works on (the compiler makes the inference).

We'll add new items to the tail of the list. If the number of items in the list is at its maximum, the function removes the first item (from the head) and frees it with `ExFreePool`, decrementing the item count.

This is not the only way to handle the case where the number of items is too large. Feel free to use other ways. A more "precise" way might be tracking the number of bytes used, rather than number of items, because each item is different in size.



We don't need to use atomic increment/decrement operations in the `AddItem` function because manipulation of the item count is always done under the protection of the fast mutex.

With `AddItem` implemented, we can call it from our process notify routine:

```
g_State.AddItem(&info->Entry);
```



Implement the limit by reading from the registry in `DriverEntry`. Hint: you can use APIs such as `ZwOpenKey` or `IoOpenDeviceRegistryKey` and then `ZwQueryValueKey`. We'll look at these APIs more closely in chapter 11.

Handling Process Create Notifications

Process create notifications are more complex because the amount of information varies. The command line length is different for different processes. First we need to decide what information to store for process creation. Here is a first try:

```
struct ProcessCreateInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ParentProcessId;
    WCHAR CommandLine[1024];
};
```

We choose to store the process ID, the parent process ID and the command line. Although this structure can work and is fairly easy to deal with because its size is known in advance.



What might be an issue with the above declaration?

The potential issue here is with the command line. Declaring the command line with constant size is simple, but not ideal. If the command line is longer than allocated, the driver would have to trim it, possibly hiding important information. If the command line is shorter than the defined limit, the structure is wasting memory.



Can we use something like this?

```
struct ProcessCreateInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ParentProcessId;
    UNICODE_STRING CommandLine;    // can this work?
};
```

This cannot work. First, UNICODE_STRING is not normally defined in user mode headers. Secondly (and much worse), the internal pointer to the actual characters normally would point to system space, inaccessible to user-mode. Thirdly, how would that string be eventually freed?

Here is another option, which we'll use in our driver:

```
struct ProcessCreateInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ParentProcessId;
    ULONG CreatingThreadId;
    ULONG CreatingProcessId;
    USHORT CommandLineLength;
    WCHAR CommandLine[1];
};
```

We'll store the command line length and copy the actual characters at the end of the structure, starting from `CommandLine`. The array size is specified as 1 just to make it easier to work with in the code. The actual number of characters is provided by `CommandLineLength`.

Given this declaration, we can begin implementation for process creation (`CreateInfo` is non-NULL):

```
USHORT allocSize = sizeof(FullItem<ProcessCreateInfo>);
USHORT commandLineSize = 0;
if (CreateInfo->CommandLine) {
    commandLineSize = CreateInfo->CommandLine->Length;
    allocSize += commandLineSize;
}
auto info = (FullItem<ProcessCreateInfo>*)ExAllocatePoolWithTag(
    PagedPool, allocSize, DRIVER_TAG);
if (info == nullptr) {
    KdPrint((DRIVER_PREFIX "failed allocation\n"));
    return;
}
```

The total size for an allocation is based on the command line length (if any). Now it's time to fill in the fixed-size details:

```
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Type = ItemType::ProcessCreate;
item.Size = sizeof(ProcessCreateInfo) + commandLineSize;
item.ProcessId = HandleToULong(ProcessId);
item.ParentProcessId = HandleToULong(CreateInfo->ParentProcessId);
item.CreatingProcessId = HandleToULong(
    CreateInfo->CreatingThreadId.UniqueProcess);
item.CreatingThreadId = HandleToULong(
    CreateInfo->CreatingThreadId.UniqueThread);
```

The item size must be calculated to include the command line length.

Next, we need to copy the command line to the address where `CommandLine` begins, and set the correct command line length:

```
if (commandLineSize > 0) {
    memcpy(item.CommandLine, CreateInfo->CommandLine->Buffer, commandLineSize);
    item.CommandLineLength = commandLineSize / sizeof(WCHAR); // len in WCHARs
}
else {
    item.CommandLineLength = 0;
}
g_State.AddItem(&info->Entry);
```

The command line length is stored in characters, rather than bytes. This is not mandatory, of course, but would probably be easier to use by user mode code. Notice the command line is not NULL terminated - it's up to the client not read too many characters. As an alternative, we can make the string null terminated to simplify client code. In fact, if we do that, the command line length is not even needed.



Make the command line NULL-terminated and remove the command line length.



Astute readers may notice that the calculated data length is actually one character longer than needed, perfect for adding a NULL-terminator. Why? `sizeof(ProcessCreateInfo)` includes one character of the command line.

For easier reference, here is the complete process notify callback implementation:

```
void OnProcessNotify(PEPROCESS Process, HANDLE ProcessId,
PPS_CREATE_NOTIFY_INFO CreateInfo) {
    if (CreateInfo) {
        USHORT allocSize = sizeof(FullItem<ProcessCreateInfo>);
        USHORT commandLineSize = 0;
        if (CreateInfo->CommandLine) {
            commandLineSize = CreateInfo->CommandLine->Length;
            allocSize += commandLineSize;
        }
        auto info = (FullItem<ProcessCreateInfo>*)ExAllocatePoolWithTag(
            PagedPool, allocSize, DRIVER_TAG);
        if (info == nullptr) {
            KdPrint((DRIVER_PREFIX "failed allocation\n"));
            return;
        }

        auto& item = info->Data;
        KeQuerySystemTimePrecise(&item.Time);
        item.Type = ItemType::ProcessCreate;
        item.Size = sizeof(ProcessCreateInfo) + commandLineSize;
        item.ProcessId = HandleToULong(ProcessId);
        item.ParentProcessId = HandleToULong(CreateInfo->ParentProcessId);
        item.CreatingProcessId = HandleToULong(
            CreateInfo->CreatingThreadId.UniqueProcess);
        item.CreatingThreadId = HandleToULong(
            CreateInfo->CreatingThreadId.UniqueThread);

        if (commandLineSize > 0) {
```

```

        memcpy(item.CommandLine, CreateInfo->CommandLine->Buffer,
               commandLineSize);
        item.CommandLineLength = commandLineSize / sizeof(WCHAR);
    }
    else {
        item.CommandLineLength = 0;
    }
    g_State.AddItem(&info->Entry);
}
else {
    auto info = (FullItem<ProcessExitInfo>*)ExAllocatePoolWithTag(
        PagedPool, sizeof(FullItem<ProcessExitInfo>), DRIVER_TAG);
    if (info == nullptr) {
        KdPrint((DRIVER_PREFIX "failed allocation\n"));
        return;
    }

    auto& item = info->Data;
    KeQuerySystemTimePrecise(&item.Time);
    item.Type = ItemType::ProcessExit;
    item.ProcessId = HandleToULong(ProcessId);
    item.Size = sizeof(ProcessExitInfo);
    item.ExitCode = PsGetProcessExitStatus(Process);

    g_State.AddItem(&info->Entry);
}
}

```

Providing Data to User Mode

The next thing to consider is how to provide the gathered information to a user-mode client. There are several options that could be used, but for this driver we'll let the client poll the driver for information using a read request. The driver will fill the user-provided buffer with as many events as possible, until either the buffer is exhausted or there are no more events in the queue.

We'll start the read request by obtaining the address of the user's buffer with Direct I/O (set up in DriverEntry):

```

NTSTATUS SysMonRead(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto len = irpSp->Parameters.Read.Length;
    auto status = STATUS_SUCCESS;
    ULONG bytes = 0;
    NT_ASSERT(Irp->MdlAddress); // we're using Direct I/O

    auto buffer = (PUCHAR)MmGetSystemAddressForMdlSafe(
        Irp->MdlAddress, NormalPagePriority);
    if (!buffer) {
        status = STATUS_INSUFFICIENT_RESOURCES;
    }
}

```

Now we need to access our linked list and pull items from its head. We'll add this support to the Global class by implementing a method that removed an item from the head and returns it. If the list is empty, it returns NULL:

```
LIST_ENTRY* Globals::RemoveItem() {
    Locker locker(m_Lock);
    auto item = RemoveHeadList(&m_ItemsHead);
    if (item == &m_ItemsHead)
        return nullptr;

    m_Count--;
    return item;
}
```

If the linked list is empty, RemoveHeadList returns the head itself. It's also possible to use IsListEmpty to make that determination. Lastly, we can check if m_Count is zero - all these are equivalent. If there is an item, it's returned as a LIST_ENTRY pointer.

Back to the Read dispatch routine - we can now loop around, getting an item out, copying its data to the user-mode buffer, until the list is empty or the buffer is full:

```
else {
    while (true) {
        auto entry = g_State.RemoveItem();
        if (entry == nullptr)
            break;

        //
        // get pointer to the actual data item
        //
        auto info = CONTAINING_RECORD(entry, FullItem<ItemHeader>, Entry);
        auto size = info->Data.Size;
        if (len < size) {
            //
            // user's buffer too small, insert item back
            //
            g_State.AddHeadItem(entry);
            break;
        }
        memcpy(buffer, &info->Data, size);
        len += size;
        buffer += size;
        bytes += size;
        ExFreePool(info);
    }
}
return CompleteRequest(Irp, status, bytes);
```

`Globals::RemoveItem` is called to retrieve the head item (if any). Then we have to check if the remaining bytes in the user's buffer are enough to contain the data of this item. If not, we have to push the item back to the head of the queue, accomplished with another method in the `Globals` class:

```
void Globals::AddHeadItem(LIST_ENTRY* entry) {
    Locker locker(m_Lock);
    InsertHeadList(&m_ItemsHead, entry);
    m_Count++;
}
```

If there is enough room in the buffer, a simple `memcpy` is used to copy the actual data (everything except the `LIST_ENTRY` to the user's buffer). Finally, the variables are adjusted based on the size of this item and the loop repeats.

Once out of the loop, the only thing remaining is to complete the request with whatever status and information (bytes) have been accumulated thus far.

We need to take a look at the unload routine as well. If there are items in the linked list, they must be freed explicitly; otherwise, we have a leak on our hands:

```
void SysMonUnload(PDRIVER_OBJECT DriverObject) {
    PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, TRUE);

    LIST_ENTRY* entry;
    while ((entry = g_State.RemoveItem()) != nullptr)
        ExFreePool(CONTAINING_RECORD(entry, FullItem<ItemHeader>, Entry));

    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\sysmon");
    IoDeleteSymbolicLink(&symLink);
    IoDeleteDevice(DriverObject->DeviceObject);
}
```

The linked list items are freed by repeatedly removing items from the list and calling `ExFreePool` on each item.

The User Mode Client

Once we have all this in place, we can write a user mode client that polls data using `ReadFile` and displays the results.

The main function calls `ReadFile` in a loop, sleeping a bit so that the thread is not always consuming CPU. Once some data arrives, it's sent for display

purposes:

```
#include <Windows.h>
#include <stdio.h>
#include <memory>
#include <string>
#include "..\SysMon\SysMonPublic.h"

int main() {
    auto hFile = CreateFile(L"\\\\.\\SysMon", GENERIC_READ, 0,
        nullptr, OPEN_EXISTING, 0, nullptr);
    if (hFile == INVALID_HANDLE_VALUE)
        return Error("Failed to open file");

    int size = 1 << 16; // 64 KB
    auto buffer = std::make_unique<BYTE[]>(size);

    while (true) {
        DWORD bytes = 0;
        // error handling omitted
        ReadFile(hFile, buffer.get(), size, &bytes, nullptr);

        if (bytes)
            DisplayInfo(buffer.get(), bytes);

        // wait a bit before polling again
        Sleep(400);
    }

    // never actually reached
    CloseHandle(hFile);
    return 0;
}
```

The `DisplayInfo` function must make sense of the buffer it's given. Since all events start with a common header, the function distinguishes the various events based on the `ItemType`. After the event has been dealt with, the `Size` field in the header indicates where the next event starts:

```
void DisplayInfo(BYTE* buffer, DWORD size) {
    while (size > 0) {
        auto header = (ItemHeader*)buffer;
        switch (header->Type) {
            case ItemType::ProcessExit:
            {
                DisplayTime(header->Time);
                auto info = (ProcessExitInfo*)buffer;
                printf("Process %u Exited (Code: %u)\n",
                    info->ProcessId, info->ExitCode);
                break;
            }

            case ItemType::ProcessCreate:
            {
                DisplayTime(header->Time);
```

```

        auto info = (ProcessCreateInfo*)buffer;
        std::wstring commandline(info->CommandLine,
            info->CommandLineLength);
        printf("Process %u Created. Command line: %ws\n",
            info->ProcessId, commandline.c_str());
        break;
    }

    }
    buffer += header->Size;
    size -= header->Size;
}
}

```

To extract the command line properly, the code uses the C++ `wstring` class constructor that can build a string based on a pointer and the string length. The `DisplayTime` helper function formats the time in a human-readable way:

```

void DisplayTime(const LARGE_INTEGER& time) {
    //
    // LARGE_INTEGER and FILETIME have the same size
    // representing the same format in our case
    //
    FILETIME local;

    //
    // convert to local time first (KeQuerySystemTime(Procise) returns UTC)
    //
    FileTimeToLocalFileTime((FILETIME*)&time, &local);
    SYSTEMTIME st;
    FileTimeToSystemTime(&local, &st);
    printf("%02d:%02d:%02d.%03d: ",
        st.wHour, st.wMinute, st.wSecond, st.wMilliseconds);
}

```

`SYSTEMTIME` is a convenient structure to work with, as it contains all ingredients of a date and time. In the above code, only the time is displayed, but the date components are present as well.

That's all we need to begin testing the driver and the client. The driver can be installed and started as done in earlier chapters, similar to the following:

```

sc create sysmon type= kernel binPath= C:\Test\SysMon.sys

sc start sysmon

```

Here is some sample output when running *SysMonClient.exe*:

```
16:18:51.961: Process 13124 Created. Command line: "C:\Program Files (x86)\Micro\
osoft\Edge\Application\97.0.1072.62\identity_helper.exe" --type=utility --utili\
ty-sub-type=winrt_app_id.mojom.WinrtAppIdService --field-trial-handle=2060,1091\
8786588500781911,4196358801973005731,131072 --lang=en-US --service-sandbox-type\
=none --mojo-platform-channel-handle=5404 /prefetch:8
16:18:51.967: Process 13124 Exited (Code: 3221226029)
16:18:51.969: Process 6216 Created. Command line: "C:\Program Files (x86)\Micro\
soft\Edge\Application\97.0.1072.62\identity_helper.exe" --type=utility --utilit\
y-sub-type=winrt_app_id.mojom.WinrtAppIdService --field-trial-handle=2060,10918\
786588500781911,4196358801973005731,131072 --lang=en-US --service-sandbox-type=\
none --mojo-platform-channel-handle=5404 /prefetch:8
16:18:53.836: Thread 12456 Created in process 10720
16:18:58.159: Process 10404 Exited (Code: 1)
16:19:02.033: Process 6216 Exited (Code: 0)
16:19:28.163: Process 9360 Exited (Code: 0)
```

Thread Notifications

The kernel provides thread creation and destruction callbacks, similarly to process callbacks. The API to use for registration is `PsSetCreateThreadNotifyRoutine` and for unregistering there is another API, `PsRemoveCreateThreadNotifyRoutine`:

```
NTSTATUS PsSetCreateThreadNotifyRoutine(
    _In_ PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine);
NTSTATUS PsRemoveCreateThreadNotifyRoutine (
    _In_ PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine);
```

The arguments provided to the callback routine are the process ID, thread ID and whether the thread is being created or destroyed:

```
typedef void (*PCREATE_THREAD_NOTIFY_ROUTINE)(
    _In_ HANDLE ProcessId,
    _In_ HANDLE ThreadId,
    _In_ BOOLEAN Create);
```

If a thread is created, the callback is executed by the creator thread; if the thread exits, the callback executes on that thread.

We'll extend the existing *SysMon* driver to receive thread notifications as well as process notifications. First, we'll add enum values for thread events and a structure representing the information, all in the *SysMonCommon.h* header file:

```
enum class ItemType : short {
    None,
    ProcessCreate,
```

```

        ProcessExit,
        ThreadCreate,
        ThreadExit
    };

    struct ThreadCreateInfo : ItemHeader {
        ULONG ThreadId;
        ULONG ProcessId;
    };

    struct ThreadExitInfo : ThreadCreateInfo {
        ULONG ExitCode;
    };

```

It's convenient to have ThreadExitInfo inherit from ThreadCreateInfo, as they share the thread and process IDs. It's certainly not mandatory, but it makes the thread notification callback a bit simpler to write.

Now we can add the proper registration to DriverEntry, right after registering for process notifications:

```

status = PsSetCreateThreadNotifyRoutine(OnThreadNotify);
if (!NT_SUCCESS(status)) {
    KdPrint((DRIVER_PREFIX "failed to set thread callbacks (0x%08X)\n",
        status));
    break;
}

```

Conversley, a call to PsRemoveCreateThreadNotifyRoutine is needed in the Unload routine:

```

// in SysMonUnload
PsRemoveCreateThreadNotifyRoutine(OnThreadNotify);

```

The callback routine itself is simpler than the process notification callback, since the event structures have fixed sizes. Here is the thread callback routine in its entirety:

```

void OnThreadNotify(HANDLE ProcessId, HANDLE ThreadId, BOOLEAN Create) {
    //
    // handle create and exit with the same code block, tweaking as needed
    //
    auto size = Create ? sizeof(FullItem<ThreadCreateInfo>)
        : sizeof(FullItem<ThreadExitInfo>);
    auto info = (FullItem<ThreadExitInfo>*)ExAllocatePoolWithTag(
        PagedPool, size, DRIVER_TAG);
    if (info == nullptr) {
        KdPrint((DRIVER_PREFIX "Failed to allocate memory\n"));
        return;
    }
}

```

```

    }
    auto& item = info->Data;
    KeQuerySystemTimePrecise(&item.Time);
    item.Size = Create ? sizeof(ThreadCreateInfo) : sizeof(ThreadExitInfo);
    item.Type = Create ? ItemType::ThreadCreate : ItemType::ThreadExit;
    item.ProcessId = HandleToULong(ProcessId);
    item.ThreadId = HandleToULong(ThreadId);
    if (!Create) {
        PETHREAD thread;
        if (NT_SUCCESS(PsLookupThreadByThreadId(ThreadId, &thread))) {
            item.ExitCode = PsGetThreadExitStatus(thread);
            ObDereferenceObject(thread);
        }
    }
    g_State.AddItem(&info->Entry);
}

```

Most of this code should look pretty familiar. The slightly complex part is retrieving the thread exit code.

PsGetThreadExitStatus can be used for that, but that API requires a thread object pointer rather than an ID. PsLookupThreadByThreadId is used to obtain the thread object that is passed to PsGetThreadExitStatus. It's important to remember to call ObDereferenceObject on the thread object or else it will linger in memory until the next system restart.

To complete the implementation, we'll add code to the client that knows how to display thread creation and destruction (in the switch block inside DisplayInfo):

```

case ItemType::ThreadCreate:
{
    DisplayTime(header->Time);
    auto info = (ThreadCreateInfo*)buffer;
    printf("Thread %u Created in process %u\n",
        info->ThreadId, info->ProcessId);
    break;
}

case ItemType::ThreadExit:
{
    DisplayTime(header->Time);
    auto info = (ThreadExitInfo*)buffer;
    printf("Thread %u Exited from process %u (Code: %u)\n",
        info->ThreadId, info->ProcessId, info->ExitCode);
    break;
}

```

Here is some sample output given the updated driver and client:

```
16:19:41.500: Thread 10512 Created in process 9304
16:19:41.500: Thread 10512 Exited from process 9304 (Code: 0)
16:19:41.500: Thread 4424 Exited from process 9304 (Code: 0)
16:19:41.501: Thread 10180 Exited from process 9304 (Code: 0)
16:19:41.777: Process 14324 Created. Command line: "C:\WINDOWS\system32\defrag.\
exe" -p bf8 -s 000000000000003BC -b -OnlyPreferred C:
16:19:41.777: Thread 8120 Created in process 14324
16:19:41.780: Process 11572 Created. Command line: \??\C:\WINDOWS\system32\conh\
ost.exe 0xffffffff -ForceV1
16:19:41.780: Thread 7952 Created in process 11572
16:19:41.784: Thread 8748 Created in process 11572
16:19:41.784: Thread 6408 Created in process 11572
```



Add client code that displays the process image name for thread create and exit.

Windows 10 adds another registration function that provides additional flexibility.

```
typedef enum _PSCREATETHREADNOTIFYTYPE {
    PsCreateThreadNotifyNonSystem = 0,
    PsCreateThreadNotifySubsystems = 1
} PSCREATETHREADNOTIFYTYPE;

NTSTATUS PsSetCreateThreadNotifyRoutineEx(
    _In_ PSCREATETHREADNOTIFYTYPE NotifyType,
    _In_ PVOID NotifyInformation);    // PCREATE_THREAD_NOTIFY_ROUTINE
```

Using PsCreateThreadNotifyNonSystem indicates the callback for new threads should execute on the newly created thread, rather than the creator.

Image Load Notifications

The last callback mechanism we'll look at in this chapter is image load notifications. Whenever a PE image (EXE, DLL, driver) file loads, the driver can receive a notification.

The PsSetLoadImageNotifyRoutine API registers for these notifications, and PsRemoveImageNotifyRoutine is used for unregistering:

```
NTSTATUS PsSetLoadImageNotifyRoutine(
    _In_ PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine);
```

```
NTSTATUS PsRemoveLoadImageNotifyRoutine(  
    _In_ PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine);
```

The callback function has the following prototype:

```
typedef void (*PLOAD_IMAGE_NOTIFY_ROUTINE)(  
    _In_opt_ PUNICODE_STRING FullImageName,  
    _In_ HANDLE ProcessId,    // pid into which image is being mapped  
    _In_ PIMAGE_INFO ImageInfo);
```

Curiously enough, there is no callback mechanism for image unloads.

The *FullImageName* argument is somewhat tricky. As indicated by the SAL annotation, it's optional and can be NULL. Even if it's not NULL, it doesn't always produce the correct image file name before Windows 10. The reasons for that are rooted deep in the kernel, it's I/O system and the file system cache. In most cases, this works fine, and the format of the path is the internal NT format, starting with something like “\Device\HadrdiskVolumex\...” rather than “c:\...”. Translation can be done in a few ways, we'll see one way when we look at the client code.

The *ProcessId* argument is the process ID into which the image is loaded. For drivers (kernel modules), this value is zero.

The *ImageInfo* argument contains additional information on the image, declared as follows:

```
#define IMAGE_ADDRESSING_MODE_32BIT    3  
  
typedef struct _IMAGE_INFO {  
    union {  
        ULONG Properties;  
        struct {  
            ULONG ImageAddressingMode : 8; // Code addressing mode  
            ULONG SystemModeImage : 1; // System mode image  
            ULONG ImageMappedToAllPids : 1; // Image mapped into all processes  
            ULONG ExtendedInfoPresent : 1; // IMAGE_INFO_EX available  
            ULONG MachineTypeMismatch : 1; // Architecture type mismatch  
            ULONG resourcesignatureLevel : 4; // Signature level  
            ULONG resourcesignatureType : 3; // Signature type  
            ULONG ImagePartialMap : 1; // Nonzero if entire image is not \ mapped  
            ULONG Reserved : 12;  
        };  
    };  
};
```



```
PVOID      ImageBase;
ULONG      resourceselector;
SIZE_T     resourceSize;
ULONG      resourceSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

Here is quick rundown of the important fields in this structure:

- *SystemModeImage* - this flag is set for a kernel image, and unset for a user mode image.
- *resourcesignatureLevel* - signing level for *Protected Processes Light* (PPL) (Windows 8.1 and later). See SE_SIGNING_LEVEL_ constants in the WDK.
- *resourcesignatureType* - signature type for PPL (Windows 8.1 and later). See the SE_IMAGE_SIGNATURE_TYPE enumeration in the WDK.
- *ImageBase* - the virtual address into which the image is loaded.
- *ImageSize* - the size of the image.
- *ExtendedInfoPresent* - if this flag is set, then IMAGE_INFO is part of a larger structure, IMAGE_INFO_EX, shown here:

```
typedef struct _IMAGE_INFO_EX {
    SIZE_T      Size;
    IMAGE_INFO   ImageInfo;
    struct _FILE_OBJECT *FileObject;
} IMAGE_INFO_EX, *PIMAGE_INFO_EX;
```

To access this larger structure, a driver can use the CONTAINING_RECORD macro like so:

```
if (ImageInfo->ExtendedInfoPresent) {
    auto exinfo = CONTAINING_RECORD(ImageInfo, IMAGE_INFO_EX, ImageInfo);
    // access FileObject
}
```

The extended structure adds just one meaningful member - the file object used to open the image. This may be useful for retrieving the file name in pre-Windows 10 machines, as we'll soon see.

As with the process and thread notifications, we'll add the needed code to register in DriverEntry and the code to unregister in the Unload routine. Here is the full DriverEntry function (with KdPrint calls removed for brevity):

```
extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    auto status = STATUS_SUCCESS;
```

```

PDEVICE_OBJECT DeviceObject = nullptr;
UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\sysmon");
bool symLinkCreated = false;
bool processCallbacks = false, threadCallbacks = false;

do {
    UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\\Device\\sysmon");
    status = IoCreateDevice(DriverObject, 0, &devName,
        FILE_DEVICE_UNKNOWN, 0, TRUE, &DeviceObject);
    if (!NT_SUCCESS(status)) {
        break;
    }
    DeviceObject->Flags |= DO_DIRECT_IO;

    status = IoCreateSymbolicLink(&symLink, &devName);
    if (!NT_SUCCESS(status)) {
        break;
    }
    symLinkCreated = true;

    status = PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, FALSE);
    if (!NT_SUCCESS(status)) {
        break;
    }
    processCallbacks = true;

    status = PsSetCreateThreadNotifyRoutine(OnThreadNotify);
    if (!NT_SUCCESS(status)) {
        break;
    }
    threadCallbacks = true;

    status = PsSetLoadImageNotifyRoutine(OnImageLoadNotify);
    if (!NT_SUCCESS(status)) {
        break;
    }
} while (false);

if (!NT_SUCCESS(status)) {
    if (threadCallbacks)
        PsRemoveCreateThreadNotifyRoutine(OnThreadNotify);
    if (processCallbacks)
        PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, TRUE);
    if (symLinkCreated)
        IoDeleteSymbolicLink(&symLink);
    if (DeviceObject)
        IoDeleteDevice(DeviceObject);
    return status;
}

g_State.Init(10000);

DriverObject->DriverUnload = SysMonUnload;
DriverObject->MajorFunction[IRP_MJ_CREATE] =
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = SysMonCreateClose;
DriverObject->MajorFunction[IRP_MJ_READ] = SysMonRead;

return status;
}

```

We'll add an event type to the `ItemType` enum:

```
enum class ItemType : short {
    None,
    ProcessCreate,
    ProcessExit,
    ThreadCreate,
    ThreadExit,
    ImageLoad
};
```

As before, we need a structure to contain the information we can get from image load:

```
const int MaxImageFileSize = 300;

struct ImageLoadInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ImageSize;
    ULONG64 LoadAddress;
    WCHAR ImageFileName[MaxImageFileSize + 1];
};
```

For variety, `ImageLoadInfo` uses a fixed size array to store the path to the image file. The interested reader should change that to use a scheme similar to process create notifications.

The image load notification starts by not storing information on kernel images:

```
void OnImageLoadNotify(PUNICODE_STRING FullImageName,
    HANDLE ProcessId, PIMAGE_INFO ImageInfo) {
    if (ProcessId == nullptr) {
        // system image, ignore
        return;
    }
}
```

This is not necessary, of course. You can remove the above check so that kernel images are reported as well. Next, we allocate the data structure and fill in the usual information:

```
auto size = sizeof(FullItem<ImageLoadInfo>);
auto info = (FullItem<ImageLoadInfo>*)ExAllocatePoolWithTag(PagedPool, size, DRIVER_TAG);
if (info == nullptr) {
    KdPrint((DRIVER_PREFIX "Failed to allocate memory\n"));
    return;
}
```

```
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Size = sizeof(item);
item.Type = ItemType::ImageLoad;
item.ProcessId = HandleToULong(ProcessId);
item.ImageSize = (ULONG)ImageInfo->ImageSize;
item.LoadAddress = (ULONG64)ImageInfo->ImageBase;
```

The interesting part is the image path. The simplest option is to examine `FullImageName`, and if non-NULL, just grab its contents. But since this information might be missing or not 100% reliable, we can try something else first, and fall back on `FullImageName` if all else fails.

The secret is to use `FltGetFileNameInformationUnsafe` - a variant on `FltGetFileNameInformation` that is used with File System Mini-filters, as we'll see in chapter 12. The "Unsafe" version can be called in non-file-system contexts as is our case. A full discussion on `FltGetFileNameInformation` is saved for chapter 12. For now, let's just use it if the file object is available:

```
item.ImageFileName[0] = 0; // assume no file information
if (ImageInfo->ExtendedInfoPresent) {
    auto exinfo = CONTAINING_RECORD(ImageInfo, IMAGE_INFO_EX, ImageInfo);
    PFLT_FILE_NAME_INFORMATION nameInfo;
    if (NT_SUCCESS(FltGetFileNameInformationUnsafe(exinfo->FileObject,
        nullptr, FLT_FILE_NAME_NORMALIZED | FLT_FILE_NAME_QUERY_DEFAULT,
        &nameInfo))) {
        // copy the file path
        wcsncpy_s(item.ImageFileName, nameInfo->Name.Buffer);
        FltReleaseFileNameInformation(nameInfo);
    }
}
```

`FltGetFileNameInformationUnsafe` requires the file object that can be obtained from the extended `IMAGE_INFO_EX` structure. `wcsncpy_s` ensures we don't copy more characters than are available in the buffer. `FltReleaseFileNameInformation` must be called to free the `PFLT_FILE_NAME_INFORMATION` object allocated by `FltGetFileNameInformationUnsafe`.

To gain access to these functions, add `#include` for `<FltKernel.h>` and add `FlgMgr.lib` into the Linker Input / Additional Dependencies line.

Finally, if this method does not produce a result, we fall back to using the provided image path:

```
if (item.ImageFileName[0] == 0 && FullImageName) {
    wcsncpy_s(item.ImageFileName, FullImageName->Buffer);
}

g_State.AddItem(&info->Entry);
```

Here is the full image load notification code for easier reference (KdPrint removed):

```
void OnImageLoadNotify(PUNICODE_STRING FullImageName, HANDLE ProcessId, PIMAGE_\\
INFO ImageInfo) {
    if (ProcessId == nullptr) {
        // system image, ignore
        return;
    }

    auto size = sizeof(FullItem<ImageLoadInfo>);
    auto info = (FullItem<ImageLoadInfo>*)ExAllocatePoolWithTag(
        PagedPool, size, DRIVER_TAG);
    if (info == nullptr)
        return;

    auto& item = info->Data;
    KeQuerySystemTimePrecise(&item.Time);
    item.Size = sizeof(item);
    item.Type = ItemType::ImageLoad;
    item.ProcessId = HandleToULong(ProcessId);
    item.ImageSize = (ULONG)ImageInfo->ImageSize;
    item.LoadAddress = (ULONG64)ImageInfo->ImageBase;
    item.ImageFileName[0] = 0;

    if (ImageInfo->ExtendedInfoPresent) {
        auto exinfo = CONTAINING_RECORD(ImageInfo, IMAGE_INFO_EX, ImageInfo);
        PFLT_FILE_NAME_INFORMATION nameInfo;
        if (NT_SUCCESS(FltGetFileNameInformationUnsafe(
            exinfo->FileObject, nullptr,
            FLT_FILE_NAME_NORMALIZED | FLT_FILE_NAME_QUERY_DEFAULT,
            &nameInfo))) {
            wcsncpy_s(item.ImageFileName, nameInfo->Name.Buffer);
            FltReleaseFileNameInformation(nameInfo);
        }
    }
    if (item.ImageFileName[0] == 0 && FullImageName) {
        wcsncpy_s(item.ImageFileName, FullImageName->Buffer);
    }

    g_State.AddItem(&info->Entry);
}
```

Final Client Code

The client code must be extended for image loads. It seems easy enough except for one snag: the resulting image path retrieved in the image load notification is in NT Device form, instead of the more common, “DOS based” form with drive letters, which in fact are symbolic links. We can see these mappings in tools such as *WinObj* from *Sysinternals* (figure 9-3).

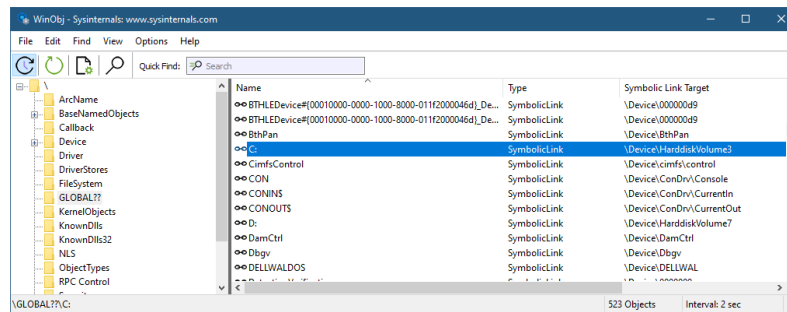


Figure 9-3: Symbolic links in WinObj

Notice the device name targets for C: and D: in figure 9-3. A file like *c:\\temp\\mydll.dll* will be reported as *\\Device\\DeviceHarddiskVolume3\\temp\\mydll.dll*. It would be nice if the display would show the common mappings instead of the NT device name.

One way of getting these mappings is by calling *QueryDosDevice*, which retrieves the target of a symbolic link stored in the “??” Object Manager directory. We are already familiar with these symbolic links, as they are valid strings to the *CreateFile* API.

Based on *QueryDosDevice*, we can loop over all existing drive letters and store the targets. Then, we can lookup every device name and find its drive letter (symbolic link). Here is a function to do that. If we can’t find a match, we’ll just return the original string:

```
#include <unordered_map>

std::wstring GetDosNameFromNTName(PCWSTR path) {
    if (path[0] != L'\\')
        return path;

    static std::unordered_map<std::wstring, std::wstring> map;
    if (map.empty()) {
        auto drives = GetLogicalDrives();
        int c = 0;
        WCHAR root[] = L"X:";
        WCHAR target[128];
        while (drives) {
```

```

        if (drives & 1) {
            root[0] = 'A' + c;
            if (QueryDosDevice(root, target, _countof(target))) {
                map.insert({ target, root });
            }
        }
        drives >>= 1;
        c++;
    }
}

auto pos = wcschr(path + 1, L'\\');
if (pos == nullptr)
    return path;

pos = wcschr(pos + 1, L'\\');
if (pos == nullptr)
    return path;

std::wstring nname(path, pos - path);
if (auto it = map.find(nname); it != map.end())
    return it->second + std::wstring(pos);

return path;
}

```

I will let the interested reader figure out how this code works. In any case, since user-mode is not the focus of this book, you can just use the function as is, as we'll do in our client.

Here is the part in DisplayInfo that handles image load notifications (within the switch):

```

case ItemType::ImageLoad:
{
    DisplayTime(header->Time);
    auto info = (ImageLoadInfo*)buffer;
    printf("Image loaded into process %u at address 0x%llx (%ws)\n",
        info->ProcessId, info->LoadAddress,
        GetDosNameFromNTName(info->ImageFileName).c_str());
    break;
}

```

Here is some example output when running the full driver and client:

```

18:59:37.660: Image loaded into process 12672 at address 0x7FFD531C0000 (C:\Win\
dows\System32\msvcp110_win.dll)
18:59:37.661: Image loaded into process 12672 at address 0x7FFD5BF30000 (C:\Win\
dows\System32\advapi32.dll)
18:59:37.676: Thread 11416 Created in process 5820
18:59:37.676: Thread 12496 Created in process 4824
18:59:37.731: Thread 6636 Created in process 3852
18:59:37.731: Image loaded into process 12672 at address 0x7FFD59F70000 (C:\Win\
dows\System32\ntmarta.dll)

```

```
18:59:37.735: Image loaded into process 12672 at address 0x7FFD51340000 (C:\Win\
dows\System32\policymanager.dll)
18:59:37.735: Image loaded into process 12672 at address 0x7FFD531C0000 (C:\Win\
dows\System32\msvcp110_win.dll)
18:59:37.737: Image loaded into process 12672 at address 0x7FFD51340000 (C:\Win\
dows\System32\policymanager.dll)
18:59:37.737: Image loaded into process 12672 at address 0x7FFD531C0000 (C:\Win\
dows\System32\msvcp110_win.dll)
18:59:37.756: Thread 6344 Created in process 704
```



Add the process name in image load notifications.

Create a driver that monitors process creation and allows a client application to configure executable paths that should not be allowed to execute.

Remote Thread Detection

One interesting example of using process and thread notifications is to detect *remote threads*. A remote thread is one that is created (injected) to a process different than its creator. This fairly well-known technique can be used (for example) to force the new thread to load a DLL, essentially injecting that DLL into another process.

This scenario is not necessarily malicious, but it could be. The most common example where this happens is when a debugger attaches to a target and wants to break into the target. This is done by creating a thread in the target process (by the debugger process) and pointing the thread function to an API such as `DebugBreak` that forces a breakpoint, allowing the debugger to gain control.

Anti-malware systems know how to detect these scenarios, as these may be malicious. Let's build a driver that can make that kind of detection. At first, it seems to be very simple: when a thread is created, compare its creator's process ID with the target process where the thread is created, and if they are different - you have a remote thread on your hands.

There is a small dent in the above description. The first thread in any process is "remote" by definition, because it's created by some other process (typically the one calling `CreateProcess`), so this "natural" occurrence should not be considered a remote thread creation.



If you feel up to it, code this driver on your own!

The core of the driver are process and thread notification callbacks. The most important is the thread creation callback, where the driver's job is to determine whether a created thread is a remote one or not. We must keep an eye for new processes as well, because the first thread in a new process is technically remote, but we need to ignore it.

The data maintained by the driver and later provided to the client contains the following (*DetectorPublic.h*):

```
struct RemoteThread {
    LARGE_INTEGER Time;
    ULONG CreatorProcessId;
    ULONG CreatorThreadId;
    ULONG ProcessId;
    ULONG ThreadId;
};
```

Here is the data we'll store as part of the driver (in *KDetector.h*):

```
struct RemoteThreadItem {
    LIST_ENTRY Link;
    RemoteThread Remote;
};

const ULONG MaxProcesses = 32;

ULONG NewProcesses[MaxProcesses];
ULONG NewProcessesCount;
ExecutiveResource ProcessesLock;
LIST_ENTRY RemoteThreadsHead;
FastMutex RemoteThreadsLock;
LookasideList<RemoteThreadItem> Lookaside;
```

There are a few class wrappers for kernel APIs we haven't seen yet. FastMutex is the same we used in the *SysMon* driver. ExecutiveResource is a wrapper for an ERESOURCE structure and APIs we looked at in chapter 6. Here is its declaration and definition:

```
// ExecutiveResource.h

struct ExecutiveResource {
```

```

    void Init();
    void Delete();

    void Lock();
    void Unlock();

    void LockShared();
    void UnlockShared();

private:
    ERESOURCE m_res;
    bool m_CritRegion;
};

// ExecutiveResource.cpp

void ExecutiveResource::Init() {
    ExInitializeResourceLite(&m_res);
}

void ExecutiveResource::Delete() {
    ExDeleteResourceLite(&m_res);
}

void ExecutiveResource::Lock() {
    m_CritRegion = KeAreApcsDisabled();
    if(m_CritRegion)
        ExAcquireResourceExclusiveLite(&m_res, TRUE);
    else
        ExEnterCriticalRegionAndAcquireResourceExclusive(&m_res);
}

void ExecutiveResource::Unlock() {
    if (m_CritRegion)
        ExReleaseResourceLite(&m_res);
    else
        ExReleaseResourceAndLeaveCriticalRegion(&m_res);
}

void ExecutiveResource::LockShared() {
    m_CritRegion = KeAreApcsDisabled();
    if (m_CritRegion)
        ExAcquireResourceSharedLite(&m_res, TRUE);
    else
        ExEnterCriticalRegionAndAcquireResourceShared(&m_res);
}

void ExecutiveResource::UnlockShared() {
    Unlock();
}

```

A few things are worth noting:

- Acquiring an Executive Resource must be done in a critical region (when normal kernel APCs are disabled). The call to `KeAreApcsDisabled` returns true if normal kernel APCs are disabled. In that case a simple acquisition

will do; otherwise, a critical region must be entered first, so the “shortcuts” to enter a critical region and acquire the Executive Resource are used.



A similar API, `KeAreAllApcsDisabled` returns true if all APCs are disabled (essentially whether the thread is in a guarded region).

- An Executive Resource is used to protect the `NewProcesses` array from concurrent write access. The idea is that more reads than writes are expected for this data. In any case, I wanted to show a possible wrapper for an Executive Resource.
- The class presents an interface that can work with the `Locker<TLock>` type we have been using for exclusive access. For shared access, the `LockShared` and `UnlockShared` methods are provided. To use them conveniently, a companion class to `Locker<>` can be written to acquire the lock in a shared manner. Here is its definition (in *Locker.h* as well):

```
template<typename TLock>
struct SharedLocker {
    SharedLocker(TLock& lock) : m_lock(lock) {
        lock.LockShared();
    }
    ~SharedLocker() {
        m_lock.UnlockShared();
    }

private:
    TLock& m_lock;
};
```

`LookasideList<T>` is a wrapper for lookaside lists we met in chapter 8. It’s using the new API, as it’s easier for selecting the pool type required. Here is its definition (in *LookasideList.h*):

```
template<typename T>
struct LookasideList {
    NTSTATUS Init(Pool_Type pool, ULONG tag) {
        return ExInitializeLookasideListEx(&m_lookaside, nullptr, nullptr,
            pool, 0, sizeof(T), tag, 0);
    }

    void Delete() {
        ExDeleteLookasideListEx(&m_lookaside);
    }
};
```

```

T* Alloc() {
    return (T*)ExAllocateFromLookasideListEx(&m_lookaside);
}

void Free(T* p) {
    ExFreeToLookasideListEx(&m_lookaside, p);
}

private:
    LOOKASIDE_LIST_EX m_lookaside;
};

```

Going back to the data members for this driver. The purpose of the `NewProcesses` array is to keep track of new processes before their first thread is created. Once the first thread is created, and identified as such, the array will drop the process in question, because from that point on, any new thread created in that process from another process is a remote thread for sure. We'll see all that in the callbacks implementations.

The driver uses a simple array rather than a linked list, because I don't expect a lot of processes with no threads to exist for more than a tiny fraction, so a fixed sized array should be good enough. However, you can change that to a linked list to make this bulletproof.

When a new process is created, it should be added to the `NewProcesses` array since the process has zero threads at that moment:

```

void OnProcessNotify(PEPROCESS Process, HANDLE ProcessId,
    PPS_CREATE_NOTIFY_INFO CreateInfo) {
    UNREFERENCED_PARAMETER(Process);

    if (CreateInfo) {
        if (!AddNewProcess(ProcessId)) {
            KdPrint((DRIVER_PREFIX "New process created, no room to store\n"));
        }
        else {
            KdPrint((DRIVER_PREFIX "New process added: %u\n", HandleToULong(ProcessId)));
        }
    }
}

```

`AddProcess` locates an empty "slot" in the array and puts the process ID in it:

```

bool AddNewProcess(HANDLE pid) {
    Locker locker(ProcessesLock);
    if (NewProcessesCount == MaxProcesses)
        return false;
}

```

```

    for(int i = 0; i < MaxProcesses; i++)
        if (NewProcesses[i] == 0) {
            NewProcesses[i] = HandleToUlong(pid);
            break;
        }
    NewProcessesCount++;
    return true;
}

```

Now comes the interesting part: the thread create/exit callback.



1. Add process names to the data maintained by the driver for each remote thread. A remote thread is when the creator (the caller) is different than the process in which the new thread is created. We also have to remove some false positives:

```

void OnThreadNotify(HANDLE ProcessId, HANDLE ThreadId, BOOLEAN Create) {
    if (Create) {
        bool remote = PsGetCurrentProcessId() != ProcessId
            && PsInitialSystemProcess != PsGetCurrentProcess()
            && PsGetProcessId(PsInitialSystemProcess) != ProcessId;
    }
}

```

The second and third checks make sure the source process or target process is not the *System* process. The reasons for the *System* process to exist in these cases are interesting to investigate, but are out of scope for this book - we'll just remove these false positives. The question is how to identify the *System* process. All versions of Windows from XP have the same PID for the *System* process: 4. We could use that number because it's unlikely to change in the future, but there is another way, which is foolproof and also allows me to introduce something new.

The kernel exports a global variable, `PsInitialSystemProcess`, which always points to the *System* process' `EPROCESS` structure. This pointer can be used just like any other opaque process pointer.

If the thread is indeed remote, we must check if it's the first thread in the process, and if so, discard this as a remote thread:

```

if (remote) {
    //
    // really remote if it's not a new process
}

```

```
//  
bool found = FindProcess(ProcessId);
```

FindProcess searches for a process ID in the NewProcesses array:

```
bool FindProcess(HANDLE pid) {  
    auto id = HandleToUlong(pid);  
    SharedLocker locker(ProcessesLock);  
    for (int i = 0; i < MaxProcesses; i++)  
        if (NewProcesses[i] == id)  
            return true;  
    return false;  
}
```

If the process is found, then it's the first thread in the process and we should remove the process from the new processes array so that subsequent remote threads (if any) can be identified as such:

```
if (found) {  
    //  
    // first thread in process, remove process from new processes array  
    //  
    RemoveProcess(ProcessId);  
}
```

RemoveProcess searches for the PID and removes it from the array by zeroing it out:

```
bool RemoveProcess(HANDLE pid) {  
    auto id = HandleToUlong(pid);  
    Locker locker(ProcessesLock);  
    for (int i = 0; i < MaxProcesses; i++)  
        if (NewProcesses[i] == id) {  
            NewProcesses[i] = 0;  
            NewProcessesCount--;  
            return true;  
        }  
    return false;  
}
```

If the process isn't found, then it's not new and we have a real remote thread on our hands:

```
else {  
    //  
    // really a remote thread  
    //  
    auto item = Lookaside.Alloc();  
    auto& data = item->Remote;
```

```

KeQuerySystemTimePrecise(&data.Time);
data.CreatorProcessId = HandleToULong(PsGetCurrentProcessId());
data.CreatorThreadId = HandleToULong(PsGetCurrentThreadId());
data.ProcessId = HandleToULong(ProcessId);
data.ThreadId = HandleToULong(ThreadId);

KdPrint((DRIVER_PREFIX
        "Remote thread detected. (PID: %u, TID: %u) -> (PID: %u, TID: %u)\n",
        data.CreatorProcessId, data.CreatorThreadId,
        data.ProcessId, data.ThreadId));

Locker locker(RemoteThreadsLock);
// TODO: check the list is not too big
InsertTailList(&RemoteThreadsHead, &item->Link);
}

```

Getting the data to a user mode client can be done in the same way as we did for the *SysMon* driver:

```

NTSTATUS DetectorRead(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto len = irpSp->Parameters.Read.Length;
    auto status = STATUS_SUCCESS;
    ULONG bytes = 0;
    NT_ASSERT(Irp->MdlAddress);

    auto buffer = (PUCHAR)MmGetSystemAddressForMdlSafe(
        Irp->MdlAddress, NormalPagePriority);
    if (!buffer) {
        status = STATUS_INSUFFICIENT_RESOURCES;
    }
    else {
        Locker locker(RemoteThreadsLock);
        while (true) {
            //
            // if the list is empty, there is nothing else to give
            //
            if (IsListEmpty(&RemoteThreadsHead))
                break;

            //
            // if remaining buffer size is too small, break
            //
            if (len < sizeof(RemoteThread))
                break;

            auto entry = RemoveHeadList(&RemoteThreadsHead);
            auto info = CONTAINING_RECORD(entry, RemoteThreadItem, Link);
            ULONG size = sizeof(RemoteThread);
            memcpy(buffer, &info->Remote, size);
            len -= size;
            buffer += size;
            bytes += size;
            //
            // return data item to the lookaside list
            //
            Lookaside.Free(info);
        }
    }
}

```

```

    }
}
return CompleteRequest(Irp, status, bytes);
}

```

Because there is just one type of “event” and it has a fixed size, the code is simpler than in the *SysMon* case.

The full driver code is in the *KDetector* project in the solution for this chapter.

The Detector Client

The client code is very similar to the *SysMon* client, but simpler, because all “events” have the same structure and are even fixed-sized. Here are the main and DisplayData functions:

```

void DisplayData(const RemoteThread* data, int count) {
    for (int i = 0; i < count; i++) {
        auto& rt = data[i];
        DisplayTime(rt.Time);
        printf("Remote Thread from PID: %u TID: %u -> PID: %u TID: %u\n",
            rt.CreatorProcessId, rt.CreatorThreadId, rt.ProcessId, rt.ThreadId);
    }
}

int main() {
    HANDLE hDevice = CreateFile(L"\\\\.\\kdetector", GENERIC_READ, 0,
        nullptr, OPEN_EXISTING, 0, nullptr);
    if (hDevice == INVALID_HANDLE_VALUE)
        return Error("Error opening device");

    RemoteThread rt[20];    // fixed array is good enough

    for (;;) {
        DWORD bytes;
        if (!ReadFile(hDevice, rt, sizeof(rt), &bytes, nullptr))
            return Error("Failed to read data");

        DisplayData(rt, bytes / sizeof(RemoteThread));
        Sleep(1000);
    }

    CloseHandle(hDevice);
    return 0;
}

```

The DisplayTime is the same one from the *SysMonClient* project.

We can test the driver by installing it and starting it normally, and launching our client (or we can use *DbgView* to see the remote thread outputs). The classic

example of a remote thread (as mentioned earlier) is when a debugger wishes to forcefully break into a target process. Here is one way to do that:

1. Run some executable, say Notepad.exe.
2. Launch WinDbg.
3. Use WinDbg to attach to the Notepad process. A remote thread notification should appear.

Here are some examples of output when the detector client is running:

```
13:08:15.280: Remote Thread from PID: 7392 TID: 4788 -> PID: 8336 TID: 9384
13:08:58.660: Remote Thread from PID: 7392 TID: 13092 -> PID: 8336 TID: 13288
13:10:52.313: Remote Thread from PID: 7392 TID: 13092 -> PID: 8336 TID: 12676
13:11:25.207: Remote Thread from PID: 15268 TID: 7564 -> PID: 1844 TID: 6688
13:11:25.209: Remote Thread from PID: 15268 TID: 15152 -> PID: 1844 TID: 7928
```

You might find some remote thread entries surprising (run *Process Explorer* for a while, for example)

The full code of the client is in the *Detector* project.



Display process names in the client.

Summary

In this chapter we looked at some of the callback mechanisms provided by the kernel: process, thread and image loads. In the next chapter, we'll continue with more callback mechanisms - opening handles to certain object types, and Registry notifications.