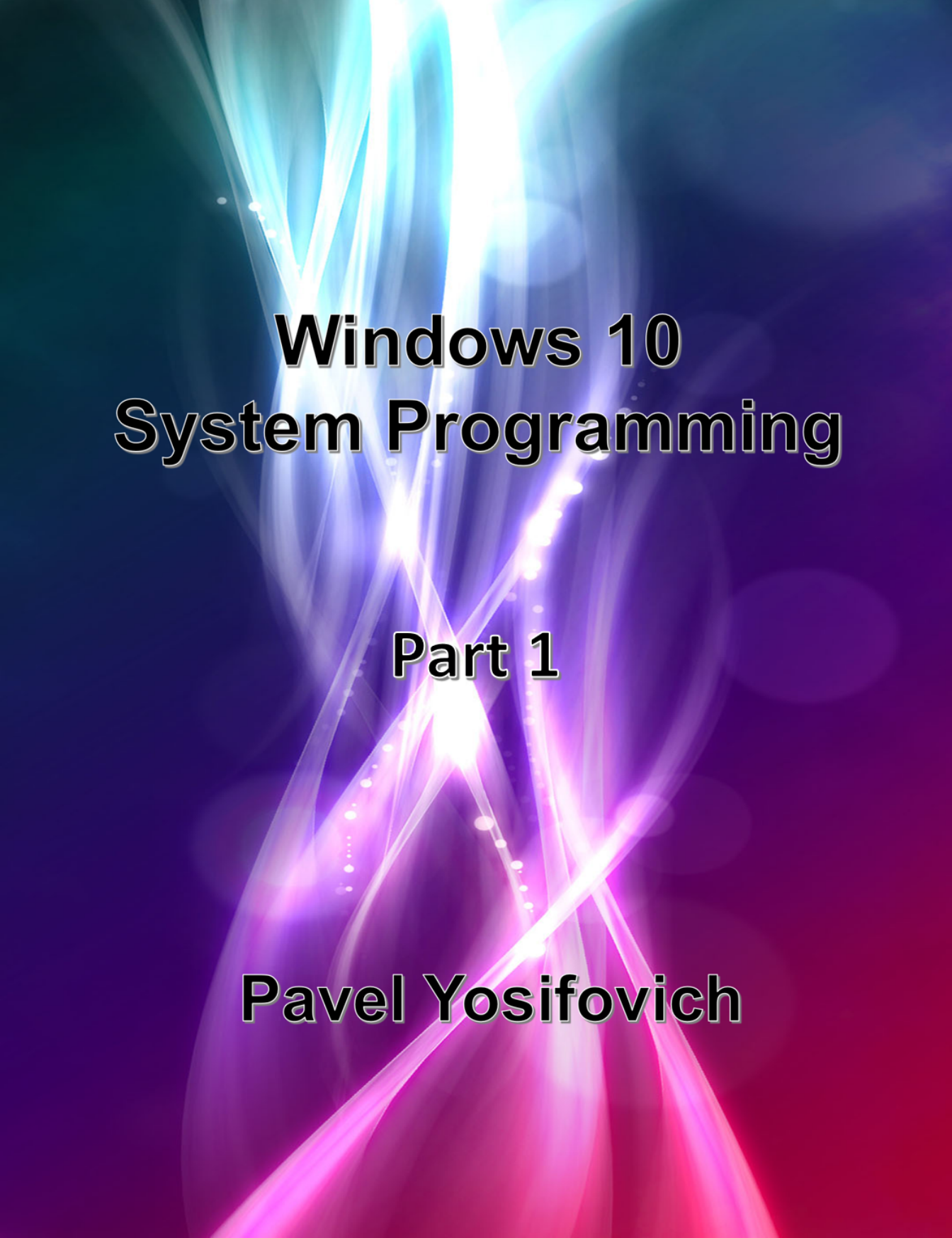

Windows 10
System Programming
Part 1
Pavel Yosifovich

Windows 10 System Programming, Part 1

Pavel Yosifovich

This book is for sale at http://leanpub.com/windows10systemprogramming

This version was published on 2023-11-09

Contents

Introduction

The term *System Programming* refers to programming close to an operating system level. *Windows 10 System Programming* provides guidance for system programmers targetting modern Windows systems, from Windows 7 up to the latest Windows 10 versions.

The book uses the documented Windows *Application Programming Interface* (API) to leverage system-level facilities, including processes, threads, synchronization primitives, virtual memory and I/O. The book is presented in two parts, due to the sheer size of the Windows API and the Windows system facilities breadth. You're holding in your hands (or your screen of choice) part 1.

Who Should Read This Book

The book is intended for software developers that target the Windows platform, and need to have a level of control not achievable by higher-level frameworks and libraries. The book uses C and C++ for code examples, as the Windows API is mostly C-based. C++ is used where it makes sense, where its advantages are obvious in terms of maintenance, clarity, resource management, and any combination of the above. The book does not use non-trivial C++ constructs, such as template metaprogramming. The book is not about C++, it's about Windows.

That said, other languages can be used to target the Windows API through their specialized interoperability mechanisms. For example, .NET languages (C#, VB, F#, etc.) can use *Platform Invoke* (P/Invoke) to make calls to the Windows API. Other languages, such as Python, Rust, Java, and many others have their own equivalent facilities.

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 in the Windows APIs. Basic C++ knowledge is highly recommended, although it is possible to traverse the book with C proficiency only.

Sample Code

All the sample code from the book is freely available in the book's Github repository at https://github.com/zodiacon/Win10SysProgBookSamples. 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

April 2020

Chapter 1: Foundations

The Windows NT line of operating systems has quite a bit of history, starting with version 3.1 launched in 1993. Today's Windows 10 is the latest successor to that initial NT 3.1. The fundamental concepts of current Windows systems is essentially the same as it was back in 1993. This shows the strength of the initial OS design. That said, Windows grew significantly since its inception, with many new features and enhancements to existing ones.

This book is about system programming, typically regarded as low-level programming of the operating system's core services, without which no significant work can be accomplished. System programming uses low level APIs to use and manipulate core objects and mechanisms in Windows, such as processes, threads and memory.

In this chapter, we'll take a look at the foundations of Windows system programming, starting from the core concepts and APIs.

In this chapter:

- **Windows Architecture Overview**
- **Windows Application Development**
- **Working with Strings**
- **32-bit vs. 64-bit Development**
- **Coding Conventions**
- **Handling API Errors**
- **The Windows Version**

Windows Architecture Overview

We'll start with a brief description of some core concepts and components in Windows. These will be elaborated on in the relevant chapters that follow.

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.
- A private virtual address space, used for allocating memory for whatever purposes the code within the process needs it.
- An access token (sometimes referred to as *primary token*), which is an object that stores the default security context of the process, used by threads executing code within the process (unless a thread assumes a different token by using impersonation).
- A private handle table to *Executive* (kernel) 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 main entry point for the process). 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.

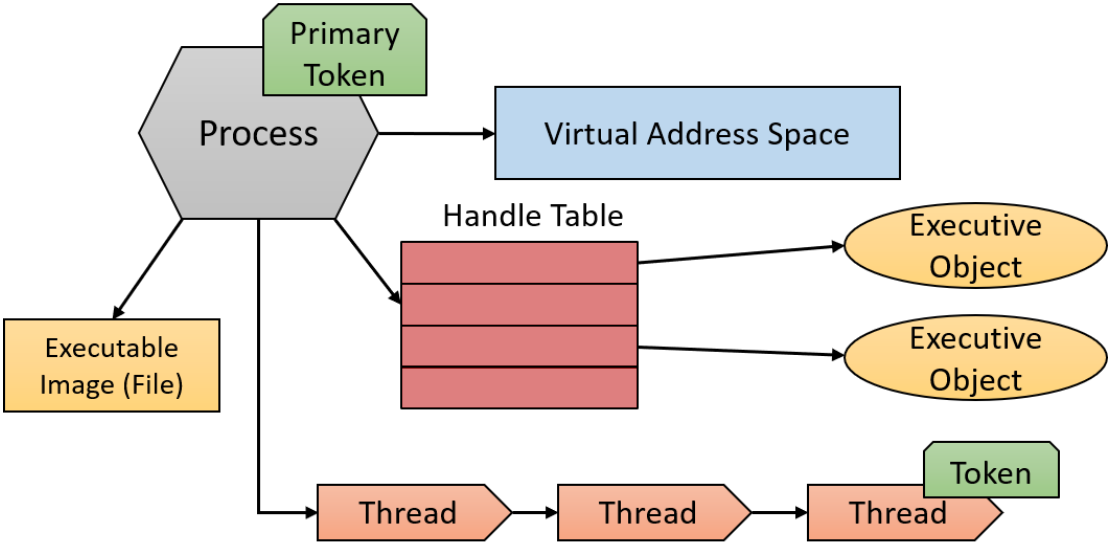


Figure 1-1: Important ingredients of a process

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 process has its own address space, its own threads, its own handle table, its own unique 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 screen shot of Task Manager's Details tab showing five instances of `Notepad.exe`, each with its own attributes.

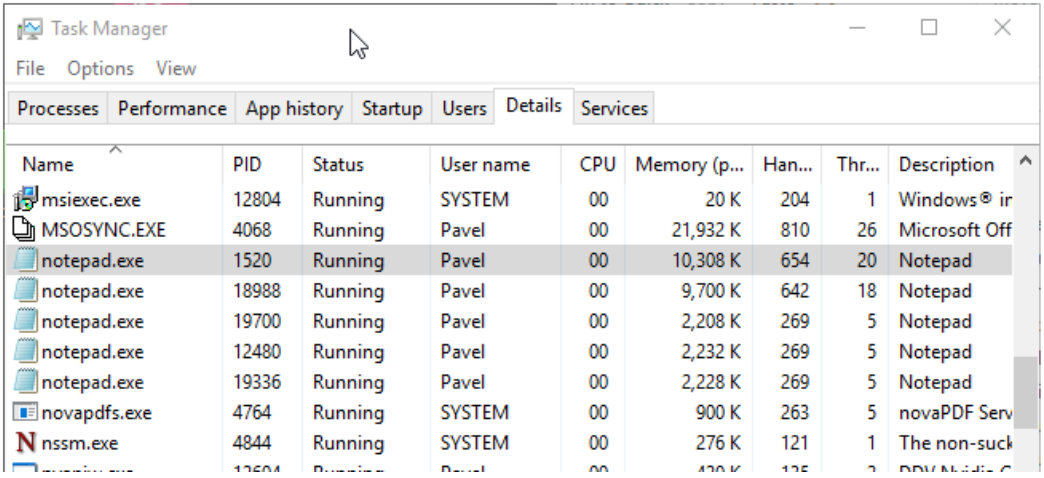


Figure 1-2: Five instances of notepad

Dynamic Link Libraries

Dynamic Link Libraries (DLLs) are executable files that can contain code, data and resources (at least one of these). DLLs are loaded dynamically into a process either at process initialization time (called *static linking*) or later when explicitly requested (*dynamic linking*). We'll look at DLLs in more detail in chapter 15. DLLs don't contain a standard *main* functions like executables, and so cannot be run directly. DLLs allow sharing their code in physical memory between multiple processes that use the same DLL, which is the case for all standard Windows DLLs stored in the *System32* directory. Some of these DLLs, known as *subsystem DLLs* implement the documented Windows API, which is the focus of this book.

Figure 1-3 shows two processes using shared DLLs mapped to the same physical (and virtual) addresses.

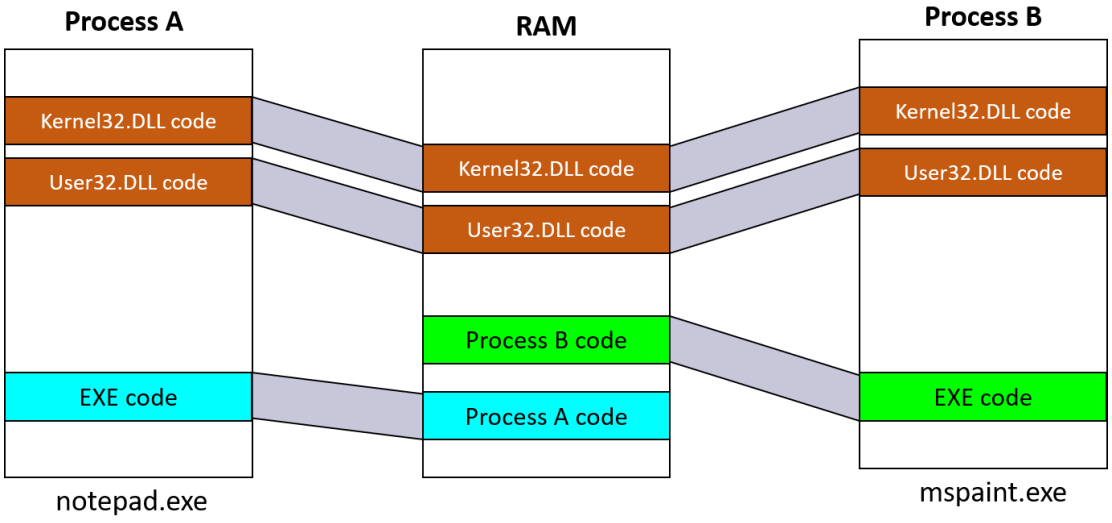


Figure 1-3: Sharing DLLs code

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 normally the first to be mapped). 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 (although technically the first 64KB of address cannot be allocated), and goes all the way to a maximum which depends on the process "bitness" (32 or 64 bit), the operating system "bitness" and a linker flag, 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 address space setting, that process address space size can be as large as 3 GB (depending on the exact setting). To get the extended address space range, the executable from which the process was created must have been marked with the `LARGEADDRESSAWARE` linker flag in its 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 is linked with the `LARGEADDRESSAWARE` flag. 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 setting that bit to 1 (which would happen for addresses larger than 2 GB) is not an issue.

The memory itself is called *virtual*, which means there is an indirect relationship between an address range 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 access the data directly. If not, the CPU will raise a page fault exception that will cause the memory manager's page fault handler to fetch the data from the appropriate file, copy it to RAM, make the required changes in the page table entries that map the buffer, and instruct the CPU to try again.

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 attributes a thread owns are the following:

- Current access mode, either user or kernel.
- Execution context, including processor registers.
- A stack, 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 moves to the *Ready* state.

General System Architecture

Figure 1-4 shows the general architecture of Windows, comprising of user-mode and kernel-mode components.

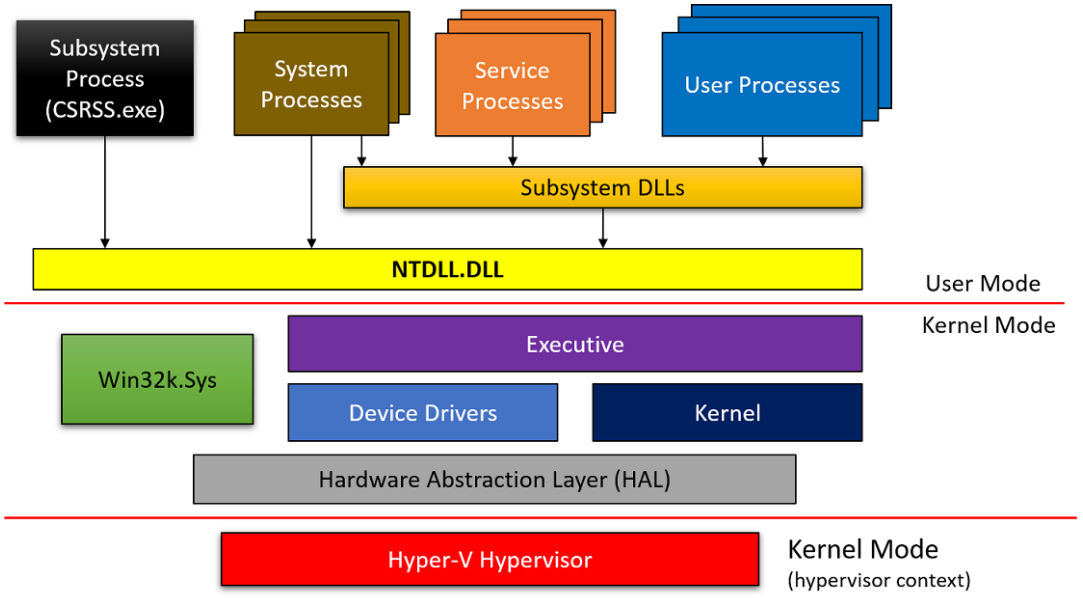


Figure 1-4: Windows system architecture

Here's a quick rundown of the named boxes appearing in figure 1-4:

- **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 certain 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. This book focuses on using the APIs exposed by these DLLs.
- **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. Although the native API is mostly undocumented, we will use some of it in this book where the standard documented Windows API is not available to achieve some goals.

- **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. Chapter 19 deals with services in more detail.

- **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 mutex and semaphore. 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. Classic device drivers provide the glue between hardware devices and the rest of the OS. Other types of drivers provide filtering capabilities. For more information on device drivers, see my book "Windows Kernel Programming".

- **Win32k.sys**

  The *kernel-mode component of the Windows subsystem*. Essentially this is 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 are handled by this component. The rest of the system has little-to-none knowledge of UI.

- **Hardware Abstraction Layer (HAL)**

  The HAL is an 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 an Interrupt Controller or a 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`, `Services.exe` and others.

- **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 system. It is a critical process, meaning if killed, the system would crash. There is normally 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 actual machine is in fact a virtual machine controlled by Hyper-V. VBS is beyond the scope of this book. For more information, check out the *Windows Internals* book.

Windows Application Development

Windows offers an *Application Programming Interface* (API) for use by developers to access a Windows' system functionality. The classic API is known as the *Windows API*, and consists mostly of a long list of C functions, providing functionality from base services dealing with processes, threads and other low-level objects, to user interface, graphics, networking and everything in between. This book focuses mostly on using this API to program Windows.

> Starting with Windows 8, Windows supports two somewhat distinct application types: the classic desktop applications that were the only application type prior to Windows 8 and Universal Windows Applications that can be uploaded to the Windows Store. From an internals perspective, these two types of applications are the same. Both types use threads, virtual memory, DLLs, handles, etc. Store applications primarily use the Windows Runtime APIs (described later in this section) but can use a subset of the classic Windows API. Conversely, desktop applications use the classic Windows API, but can also leverage a subset of the Windows Runtime APIs. This book focuses on desktop applications because the entire Windows API is available for them to use since this API contains the majority of functionality useful for system programming.

Other API styles offered by Windows, especially starting from Windows Vista are based on the *Component Object Model* (COM) technology - a component-oriented programming paradigm re-

leased in 1993 and used today by many components and services in Windows. Examples include DirectX, Windows Imaging Component (WIC), DirectShow, Media Foundation, Background Intelligent Transfer Service (BITS), Windows Management Instrumentation (WMI) and more. The most fundamentals concept in COM is the *interface* - a contract consisting of a collection of functions under a single container. We'll look at the basics of COM in chapter 18.

Naturally, over the years, various wrappers for these two fundamental API styles have been developed, some by Microsoft, some by others. Here are some of the common ones developed by Microsoft:

- *Microsoft Foundation Classes* (MFC) - C++ wrappers for (mostly) the user interface (UI) functionality exposed by Windows - working with windows, controls, menus, GDI, dialogs, etc.
- *Active Template Library* (ATL) - a C++ template-based library geared towards building COM servers and clients. We will use ATL in chapter 18 to simplify writing COM-related code.
- Windows Template Library* (WTL) - an extension to ATL, providing template-based wrappers for Windows user interface functionality. It's comparable to MFC in terms of features, but is more lightweight and does not carry a (large) DLL with it (as MFC does). We'll use WTL in this book to simplify UI-related code, as UI is not the focus of this book.
- .NET - a framework and a runtime (*Common Language Runtime* - CLR) that provide a host of services, such as Just in Time (JIT) compilation of *Intermediate Language* (IL) to native code and garbage collection. .NET can be used by leveraging new languages (C# being the most well-known) that provide lots of features, many of them abstracting Windows functionality and increasing productivity. The .NET framework uses the standard Windows APIs to accomplish its higher-level functionality. .NET is beyond the scope of this book - see the book "CLR Via C#" by Jeffrey Richter for an excellent coverage of .NET's inner workings and functionality.
- *Windows Runtime* (WinRT) - this is the newest API layer added in Windows 8 and later versions. Its primary goal is developing applications based on the *Universal Windows Platform* (UWP). These applications can be packaged and uploaded to the Windows Store and downloaded by anyone. The Windows Runtime is built around an enhanced version of COM, so it too consists of interfaces as its primary (but not only) building block. Although this platform is native (and not based on .NET), it can be used by C++, C# (and other .NET languages) and even JavaScript - Microsoft provides *language projections* to simplify accessing the Windows Runtime APIs. A subset of the Windows Runtime APIs is available for (classic) desktop applications. We'll take a look at the basics of the Windows Runtime in chapter 19.

Most of the standard Windows API function definitions are available in the `windows.h` header file. In some cases, additional headers will be needed, as well as additional import libraries. The text will point out any such headers and/or libraries.

Your First Application

This section describes the basics of using Visual Studio to write a simple application, compile and run it successfully. If you already know this, you can just skip this section.

First, you'll need to install the proper tools to develop for Windows. Here is the short list of software in order:

1. Visual Studio 2017 or 2019, any edition, including the free community edition (available at https://visualstudio.microsoft.com/downloads/). Earlier versions of Visual Studio work just fine, but it's usually best to stick with the latest versions, as these include compiler improvements as well as usability enhancements. In the installer's main window, make sure at least the *Desktop Development with C++* workload is selected, as shown in figure 1-5.

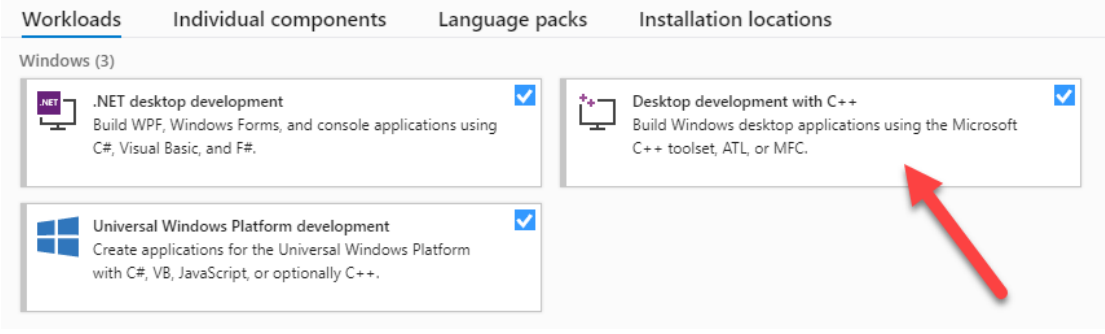


Figure 1-5: Visual Studio installer main window

2. The Windows *Software Development Kit* (SDK) is an optional installation that provides (possibly) updated headers and libraries, as well as various tools.

Once Visual Studio 2017/2019 is installed, run it and select to create a new project.

- In Visual Studio 2017, select *File / New Project...* from the menu and locate The **C++ / Desktop** node, and select the **Windows Console Application** project template, as shown in figure 1-6.
- In Visual Studio 2019, select *Create New Project* from the startup window and filter with **console** and **C++** in the project type and language, respectively, and select **Console App** (make sure the language listed is C++). This is shown in figure 1-7.

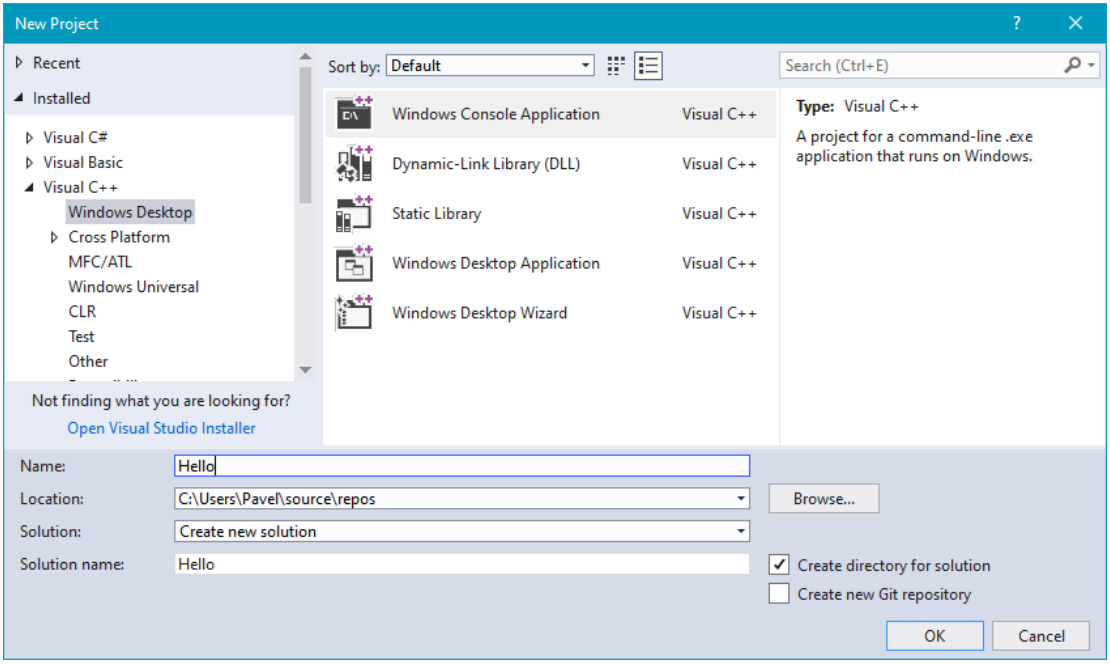


Figure 1-6: New project dialog in Visual Studio 2017

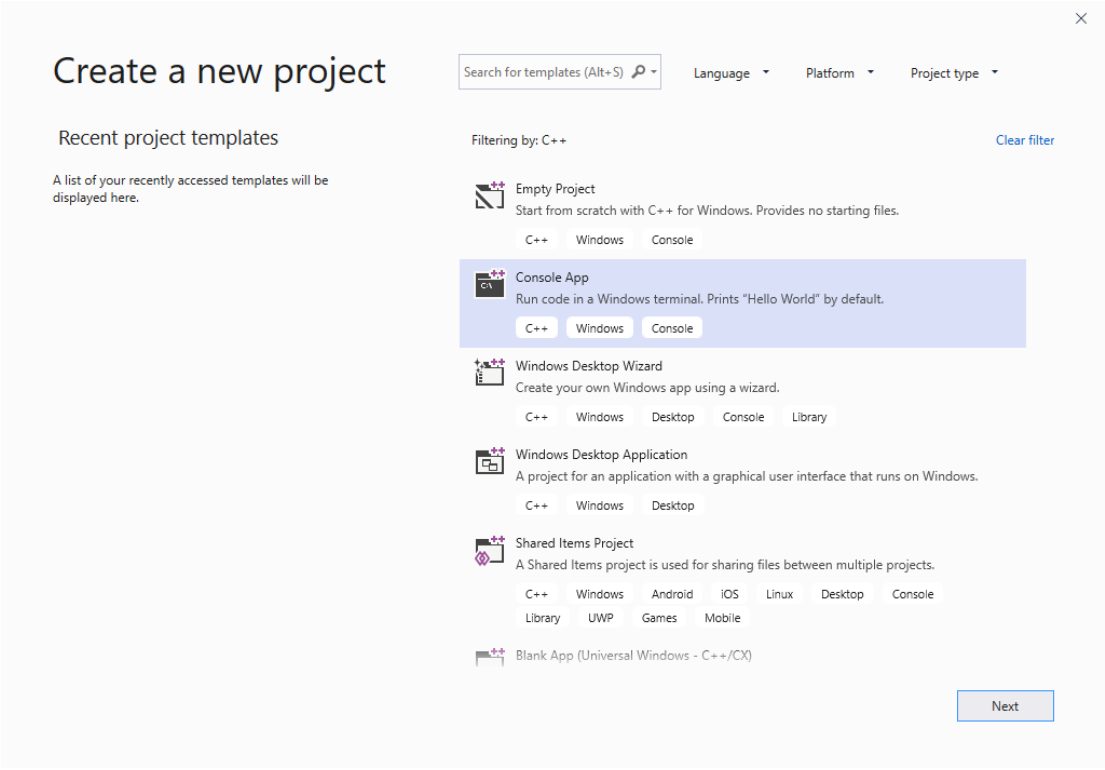


Figure 1-7: New project dialog in Visual Studio 2019

Name the project *HelloWin*, and change the destination folder if you wish, and click *OK*. The project should be created with a `HelloWin.cpp` file open in the editor with a minimal `main` function.

Add an #include at the top of the file for `windows.h`:

```
#include <windows.h>
```

> If your project has a precompiled header (an `#include "pch.h"` is at the top of every C/C++ source file,

add the `windows.h` #include to this file, so that after the first compilation, subsequent compilations will be faster.
You can also include it in a C/C++ file if you prefer, but the include must follow the `pch.h` include.

Add another #include for `stdio.h` to gain access to the `printf` function:

```
#include <stdio.h>
```

In this first application, we'll get some system information by calling the `GetNativeSystemInfo` function.
Here is the `main` function code:

```
int main() {
    SYSTEM_INFO si;
    ::GetNativeSystemInfo(&si);

    printf("Number of Logical Processors: %d\n", si.dwNumberOfProcessors);
    printf("Page size: %d Bytes\n", si.dwPageSize);
    printf("Processor Mask: 0x%p\n", (PVOID)si.dwActiveProcessorMask);
    printf("Minimum process address: 0x%p\n", si.lpMinimumApplicationAddress);
    printf("Maximum process address: 0x%p\n", si.lpMaximumApplicationAddress);

    return 0;
}
```

From the *Build* menu, select *Build Solution* to compile and link the project (technically all projects in the solution). Everything should compile and link without errors. Hit *Ctrl+F5* to launch the executable without attaching to the debugger. (Or use the *Debug* menu and select *Run Without Debugging*. A console window should open showing output similar to the following:

```
Number of Logical Processors: 12
Page size: 4096 Bytes
Processor Mask: 0x00000FFF
Minimum process address: 0x00010000
Maximum process address: 0x7FFEFFFF
```

If you run the application by pressing *F5* (*Debug* menu, *Start Debugging*), the console window will appear and very quickly disappear when the application exits. Using *Ctrl+F5* adds a convenient "Press any key to continue" prompt which lets you view the console output before dismissing the window.

Visual Studio typically creates two solution platforms (*x86* and *x64*), which can be switched easily with the Solution Platforms combobox located in the main toolbar. By default, **x86** is selected which would produce the output above. If you switch the platform to **x64** and rebuild (assuming you're running an Intel/AMD 64 bit version of Windows of course), you'll get a slightly different output:

```
Number of Logical Processors: 12
Page size: 4096 Bytes
Processor Mask: 0x0000000000000FFF
Minimum process address: 0x0000000000010000
Maximum process address: 0x00007FFFFFFEFFFF
```

The differences stem from the fact that 64-bit processes use pointers which are 8 bytes in size, while 32-bit processes use 4 bytes pointers. The address space address information from the `SYSTEM_INFO` structure is typed as pointers, so their sizes vary by process "bitness". We'll discuss 32-bit and 64-bit development in more detail in the section "32-bit vs. 64-bit Development" later in this chapter.

> Don't worry about the meaning of the information presented by this small application (although some of it is self-explanatory). We'll look at these terms in later chapters.

The use of the double colon before the function name in the above code `::GetNativeSystemInfo` is to emphasize the fact the function is part of the Windows API and not some member function of the current C++ class. In this example it's obvious as there is no C++ class around, but this convention will be used throughout the book regardless (it also slightly speeds up the compiler function lookup). More coding conventions are described later in this chapter in the section, "Coding Conventions".

Working with Strings

In classic C, strings are not real types, but are just pointers to characters that end with a zero. The Windows API uses strings this way in many cases, but not all cases. The question of encoding comes up when dealing with strings. In this section, we'll take a look at strings in general and how they are used in the Windows APIs.

In classic C, there was only a single type representing a character - `char`. The characters represented by `char` are 8 bit in size at most, where the first 7-bit values utilize the ASCII encoding. Today's systems must support multiple character sets from many languages, and these cannot fit into 8 bits. Thus, new encodings were created under the umbrella term **Unicode**, officially available online at http://www.unicode.org.

The Unicode Consortium defines several other character encodings. Here are the common ones:

- UTF-8 - the prevalent encoding used for web pages. This encoding uses one byte for Latin characters that are part of the ASCII set, and more bytes per character for other languages, such as Chinese, Hebrew, Arabic, and many others. This encoding is popular because if the text is mostly English, it's compact in size. In general, UTF-8 uses from one to four bytes for each character.
- UTF-16 - uses two bytes per character in most cases and encompasses all languages in just two bytes. Some more esoteric characters from Chinese and Japanese may require four bytes, but these are rare.
- UTF-32 - uses four bytes per character. The easiest to work with, but potentially the most wasteful.

UTF stands for *Unicode Transformation Format.*

UTF-8 may be the best when size matters, but from a programming standpoint, it's problematic because random access cannot be used. For example, to get to the 100th character in a UTF-8 string, the code needs to scan from the start of the string and work its way sequentially because there is no way to know where the 100th character may be. On the other hand, UTF-16 is much more convenient to work with programmatically (if we disregard the esoteric cases) because accessing the 100th character means adding 200 bytes to the string's start address.
UTF-32 is too wasteful and is rarely used.

Fortunately, Windows uses UTF-16 within its kernel, where each character is exactly 2 bytes. The Windows API follows suit and uses UTF-16 encoding as well, which is great as strings don't need to be converted when API calls eventually land in the kernel. However, there is a slight complication with the Windows API.

Parts of the Windows API were migrated from 16-bit Windows and Consumer Windows (Windows 95/98). These systems used ASCII as their primary way of working, which means the Windows API used ASCII strings rather than UTF-16. A further complication arises when double-byte encoding was introduced, where each character was one or two bytes in size, losing the advantage of random access.

The net result of all this is that the Windows API contains UTF-16 and ASCII functions for compatibility reasons. Since today the aforementioned systems do not exist, it's best to leave the one byte per character strings alone and use the UTF-16 functions only. Using the ASCII functions will cause the string to be converted to UTF-16 and then used with the UTF-16 function.

> UTF-16 is also beneficial when interoperating with the .NET Framework, because .NET's string type stored UTF-16 characters only. This means passing a UTF-16 string to .NET does not require any conversion or copying.

Here is an example for the function `CreateMutex`, which if searched on the web will lead to one of two functions: `CreateMutexA` and `CreateMutexW`. The offline documentation gives this prototype:

```
HANDLE CreateMutex(
  _In_opt_ LPSECURITY_ATTRIBUTES lpMutexAttributes,
  _In_     BOOL                  bInitialOwner,
  _In_opt_ LPCTSTR               lpName);
```

> The `_In_opt_` and other similar annotations are called *Syntax Annotation Language* (SAL) and are used to convey metadata information to function and structure definitions. This may be useful for humans as well as static analysis tools. The C++ compiler currently ignores these annotations, but the static analyzer available in Visual Studio Enterprise editions uses it to detect potential errors before actually running the program.

For now let's concentrate on the last parameter which is a string pointer typed as `LPCTSTR`. Let's break it down:

`L`=Long `P`=Pointer `C`=Constant `STR`=String. The only mystery is the `T` in between. `LPCTSTR` is in fact a `typedef` with one of the following definitions:

```
typedef LPCSTR  LPCTSTR;    // const char* (UNICODE not defined)
typedef LPCWSTR LPCTSTR;    // const wchar_t* (UNICODE defined)
```

> The term "long pointer" means nothing today. All pointers are the same size in a particular process (4 bytes in 32-bit processes and 8 bytes in 64-bit processes). The terms "long" and "short" (or "near") are remnants from 16-bit Windows where such terms actually had different meanings. Also, the types `LPCTSTR` and similar have another equivalent - without the `L` - `PCTSTR`, `PCWSTR`, etc. These are generally preferred in source code.

The definition of the `UNICODE` compilation constant makes `LPCTSTR` expand to a UTF-16 string, and its absence to an ASCII string. This also means `CreateMutex` cannot be a function, because the C language does not allow function overloading, where a single function name may have multiple prototypes. `CreateMutex` is a macro, expanding to `CreateMutexW` (`UNICODE` defined) or `CreateMutexA` (`UNICODE` not defined). Visual Studio defines the `UNICODE` constant by default in all new projects, which is a good thing. We always want to use the UTF-16 functions to prevent the conversion from ANSI to UTF-16 (and of course for strings that contain non-ASCII characters, such a conversion is bound to be lossy).

`W` in `CreateMutexW` stands for *Wide* and `A` in `CreateMutexA` stands for *ANSI* or *ASCII*.

If the code needs to use a constant UTF-16 string, prefix the string with `L` to instruct the compiler to convert the string to UTF-16. Here are two versions of a string, one ASCII and the other UTF-16:

```
const char    name1[] = "Hello";    //  6 bytes (including NULL terminator)
const wchar_t name2[] = L"Hello";   // 12 bytes (including UTF-16 NULL terminat\
or)
```

> From this point on, we'll use the term "Unicode" to refer to UTF-16, unless otherwise explicitly stated.

Using the macros begs the question of how can we compile code that uses constant strings without explicitly choosing ASCII vs. Unicode? The answer lies with another macro, `TEXT`. Here is an example for `CreateMutex`:

```
HANDLE hMutex = ::CreateMutex(nullptr, FALSE, TEXT("MyMutex"));
```

The `TEXT` macro expands to the constant string with or without the "L" prefix depending on whether the `UNICODE` macro is defined or not. Since the ASCII functions are more expensive since they convert their values to Unicode before calling the wide functions, we should never use the ASCII functions. This means we can simply use the "L" prefix without the `TEXT` macro. We'll adopt this convention throughout the book.

> There is a shorter version of the `TEXT` macro called `_T` defined in `<tchar.h>`. They are equivalent. Using these macros is still a fairly common practice, which is not bad in itself. However, I tend not to use it.

Similar to `LPCTSTR` there are other `typedefs` to allow using ASCII or Unicode, based on the `UNICODE` compilation constant. Table 1-1 shows some of these typedefs.

Table 1-1: Types used with strings

Common type(s)	ASCII type(s)	Unicode type(s)
`TCHAR`	`char`, `CHAR`	`wchar_t`, `WCHAR`
`LPTSTR`, `PTSTR`	`char*`, `CHAR*`, `PSTR`	`wchar_t*`, `WCHAR*`, `PWSTR`
`LPCTSTR`, `PCTSTR`	`const char*`, `PCSTR`	`const wchar_t*`, `PCWSTR`

Strings in the C/C++ Runtime

The C/C++ runtime has two sets of functions for manipulating strings. The classic (ASCII) ones begin with "str" such as `strlen`, `strcpy`, `strcat`, etc., but also Unicode versions starting with "wcs", such as `wcslen`, `wcscpy`, `wcscat` etc.
In the same vein as the Windows API, there is a set of macros that expand to either the ASCII or the Unicode version depending on another compilation constant, `_UNICODE` (notice the underscore). The prefix for these functions is "_tcs". So we have functions named `_tcslen`, `_tcscpy`, `_tcscat`, etc., all working with the `TCHAR` type.

Visual Studio defines the `_UNICODE` constant by default so we get the Unicode functions if using the "_tcs" functions. It would very weird if only one of the "UNICODE" constants would be defined, so avoid that.

String Output Parameters

Passing strings to functions as input as was done in the `CreateMutex` case is very common. Another common need is receiving results in the form of strings. The Windows API uses a few ways to pass back strings results.

The first (and the more common) case is where the client code allocates a buffer to hold the result string and provides the API with the size of the buffer (the maximum size the string can hold), and the API writes the string to the buffer up to the size specified. Some APIs also return the

actual number of characters written and/or the number of characters required if the buffer is too small.

Consider the `GetSystemDirectory` function defined like so:

```
UINT GetSystemDirectory(
  _Out_ LPTSTR lpBuffer,
  _In_  UINT   uSize);
```

The function accepts a string buffer and its size and returns the number of characters written back. Notice all sizes are in **characters**, rather than bytes, which is convenient. The function returns zero in case of failure. Here is an example usage (error handling omitted for now):

```
WCHAR path[MAX_PATH];
::GetSystemDirectory(path, MAX_PATH);
printf("System directory: %ws\n", path);
```

Don't let the pointer type confuse you - the declaration of `GetSystemDirectory` does **not** mean you provide a pointer only. Instead, you must allocate a buffer and pass a pointer to this buffer.

`MAX_PATH` is defined in the Windows headers as 260, which is the standard maximum path in Windows (this limit can be extended starting with Windows 10, as we'll see in chapter 11). Notice that `printf` uses `%ws` as the string format to indicate it's a Unicode string, since `UNICODE` is defined by default and so all strings are Unicode.

The second common case is where the client code provides a string pointer only (via its address) and the API itself allocates the memory and places the resulting pointer in the provided variable. This means the client code is tasked with freeing the memory once the resulting string is no longer needed. The trick is to use the correct function to free the memory. The API's documentation indicates which function to use. Here is an example for this usage with the `FormatMessage` function defined like so (its Unicode variant):

```
DWORD FormatMessageW(
    _In_     DWORD dwFlags,
    _In_opt_ LPCVOID lpSource,
    _In_     DWORD dwMessageId,
    _In_     DWORD dwLanguageId,
    _When_((dwFlags & FORMAT_MESSAGE_ALLOCATE_BUFFER) != 0, _At_((LPWSTR*)lpBuf\
fer, _Outptr_result_z_))
    _When_((dwFlags & FORMAT_MESSAGE_ALLOCATE_BUFFER) == 0, _Out_writes_z_(nSiz\
e))
             LPWSTR lpBuffer,
    _In_     DWORD nSize,
    _In_opt_ va_list *Arguments);
```

Looks scary, right? I purposefully included the full SAL annotations for this function as the `lpBuffer` parameter is tricky. `FormatMessage` returns a string representation of an error number (we'll discuss errors in more detail in the section "API Errors" later in this chapter). The function is flexible in the sense that it can allocate the string itself or have the client provide a buffer to hold the resulting string. The actual behavior depends on the first `dwFlags` parameter: if it includes the `FORMAT_MESSAGE_ALLOCATE_BUFFER` flag, the function will allocate the buffer of the correct size. If the flag is absent, it's up to the caller to provide storage for the returned string.

All this makes the function a bit tricky, at least because if the former option is selected, the pointer type should be `LPWSTR*` - that is, a pointer to a pointer to be filled in by the function. This requires a nasty cast to make the compiler happy.

Here is a simple `main` function that accepts an error number from the command line arguments and shows its string representation (if any). It uses the option of letting the function make the allocation. The reason is that there is no way to know what length the string should be, so it's best to let the function allocate the correct size.

```
int main(int argc, const char* argv[]) {
    if (argc < 2) {
        printf("Usage: ShowError <number>\n");
        return 0;
    }

    int message = atoi(argv[1]);

    LPWSTR text;
    DWORD chars = ::FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |    // function allocates
        FORMAT_MESSAGE_FROM_SYSTEM |
```

```
        FORMAT_MESSAGE_IGNORE_INSERTS,
        nullptr, message, 0,
        (LPWSTR)&text,      // ugly cast
        0, nullptr);

    if (chars > 0) {
        printf("Message %d: %ws\n", message, text);
        ::LocalFree(text);
    }
    else {
        printf("No such error exists\n");
    }
    return 0;
```

> The complete project is called *ShowError* in the Github repository for the book

Notice the call to the `LocalFree` function to free the string if the call is successful. The documentation for `FormatMessage` states that this is the function to call to free the buffer.

Here is an example run:

```
C:\Dev\Win10SysProg\x64\Debug>ShowError.exe 2
Message 2: The system cannot find the file specified.

C:\Dev\Win10SysProg\x64\Debug>ShowError.exe 5
Message 5: Access is denied.

C:\Dev\Win10SysProg\x64\Debug>ShowError.exe 129
Message 129: The %1 application cannot be run in Win32 mode.

C:\Dev\Win10SysProg\x64\Debug>ShowError.exe 1999
No such error exists
```

Safe String Functions

Some of the classic C/C++ runtime string functions (and some similar functions in the Windows API) are not considered "safe" from a security and reliability standpoint. For example, the `strcpy` function is problematic because it copies the source string to the target pointer until a `NULL`

terminator is reached. This could overflow the target buffer and cause a crash in the good case (the buffer could be on the stack for example, and corrupt the return address stored there), and be used as a buffer overflow attack where an alternate return address is stored on the stack, jumping to a prepared shellcode.

To mitigate these potential vulnerabilities, a set of "safe" string functions were added to the C/C++ runtime library by Microsoft, where an extra parameter is used to specify the maximum size of a target buffer, so it will never overflow. These functions have a "_s" suffix, such as `strcpy_s`, `wcscat_s`, etc.

Here are some examples using these functions:

```
void wmain(int argc, const wchar_t* argv[]) {
    // assume arc >= 2 for this demo

    WCHAR buffer[32];
    wcscpy_s(buffer, argv[1]);        // C++ version aware of static buffers

    WCHAR* buffer2 = (WCHAR*)malloc(32 * sizeof(WCHAR));
    //wcscpy_s(buffer2, argv[1]);        // does not compile
    wcscpy_s(buffer2, 32, argv[1]);        // size in characters (not bytes)

    free(buffer2);
}
```

The maximum size is always specified in characters and not bytes. Also note that these functions are able to calculate the maximum size automatically if the target buffer is statically allocated, which is convenient.

Another set of safe string functions was also added to the Windows API, at least for the purpose of reducing dependency on the C/C++ runtime. These functions are declared (and implemented) in the header `<strsafe.h>`. They are built according to the Windows API conventions, where the functions are actually macros expanding to functions with "A" or "W" suffix. Here are some simple examples of usage (using the same declarations as above):

```
StringCchCopy(buffer, _countof(buffer), argv[1]);
StringCchCat(buffer, _countof(buffer), L"cat");

StringCchCopy(buffer2, 32, argv[1]);
StringCchCat(buffer2, 32, L"cat");
```

"Cch" stands for *Count of Characters*.

Notice these functions don't have a C++ variant that knows how to handle statically allocated buffers. The solution is to use the `_countof` macro that returns the number of elements in an array. Its definition is something like `sizeof(a)/sizeof(a[0])` given an array `a`.

Which set of functions should you use? It's mostly a matter of taste. The important point is to avoid the classic, non-safe functions. If you do try to use them, you'll get a compiler error like this:

```
error C4996: 'wcscpy': This function or variable may be unsafe. Consider using \
wcscpy_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See onli\
ne help for details.
```

Clearly, this error can be disabled by defining `_CRT_SECURE_NO_WARNINGS` before including the C/C++ headers, but it would be a bad idea. This macro exists to help maintain compatibility with old source code that probably should not be touched when compiled with recent compilers.

32-bit vs. 64-bit Development

Starting with Windows Vista, Windows has official 32 bit and 64 bit versions (there was a non-commercial 64-bit version of Windows XP as well). Starting with Windows Server 2008 R2, all server versions are 64-bit only. Microsoft removed the 32-bit server versions since servers typically need lots of RAM and large process address space, making 32-bit systems too limited for server work.

The programming model of 32 bit and 64 bit is identical from an API perspective. You should be able to compile to 32 or 64 bit just by selecting the required configuration in Visual Studio and hit Build. However, if the code should build successfully for both 32-bit and 64-bit targets, coding must be done carefully to use types correctly. In 64-bit, pointers are 8 bytes in size, whereas in 32-bit they are just 4 bytes. This change can lead to errors if a pointer's size is assumed to be of a certain value. For example, consider this cast operation:

```
void* p = ...;
int value = (int)p;
// do something with value
```

This code is buggy, since in 64-bit the pointer value is truncated to 4 bytes to fit into `int` (`int` is still 4 bytes in a 64-bit compilation as well). If such a cast is truly needed, an alternate type should be used instead - `INT_PTR`:

```
void* p = ...;
INT_PTR value = (INT_PTR)p;
// do something with value
```

`INT_PTR` means: "int the size of a pointer". The Windows headers define several types like this one for this exact reason. Other types maintain their size regardless of the compilation "bitness". Table 1-2 shows some examples of common types and their sizes.

Table 1-2: Common Windows types

type name	size (32 bit)	size (64 bit)	Description
`ULONG_PTR`	4 bytes	8 bytes	unsigned integer the size of a pointer
`PVOID`, `void*`	4 bytes	8 bytes	void pointer
any pointer	4 bytes	8 bytes	
`BYTE`, `uint8_t`	1 bytes	1 bytes	unsigned 8 bit integer
`WORD`, `uint16_t`	2 bytes	2 bytes	unsigned 16 bit integer
`DWORD`, `ULONG`, `uint32_t`	4 bytes	4 bytes	unsigned 32 bit integer
`LONGLONG`, `__int64`, `int64_t`	8 bytes	8 bytes	signed 64 bit integer
`SIZE_T`, `size_t`	4 bytes	8 bytes	unsigned integer sized as native integer

The differences between 32 bit and 64 bit go beyond type sizes. The address space of a 64-bit process is 128 TB (Windows 8.1 and later) compared to a mere 2 GB for 32-bit processes. On x64 systems (Intel/AMD), 32-bit processes can execute just fine thanks to a translation layer called WOW64 (*Windows on Windows 64*). We'll take a deeper look at this layer in chapter 12. This has several implications which are discussed in that chapter as well.

All sample applications in the book should build and run successfully on x86 and x64 equally well, unless stated otherwise explicitly. It's always best to build for both x86 and x64 during development and fix any issues that may arise.

> In this book we won't cover ARM and ARM64 explicitly. All programs should build and run just fine on such systems (32 bit on ARM, 64 bit on ARM64), but I didn't have access to such systems, and so could not verify this in person.

Lastly, if code should be compiled in 64 bit only (or 32 bit only), the macro `_WIN64` is defined for 64-bit compilations. For example, we could replace the following line from *HelloWin*:

```
printf("Processor Mask: 0x%p\n", (PVOID)si.dwActiveProcessorMask);
```

with

```
#ifdef _WIN64
    printf("Processor Mask: 0x%016llX\n", si.dwActiveProcessorMask);
#else
    printf("Processor Mask: 0x%08X\n", si.dwActiveProcessorMask);
#endif
```

This would be somewhat clearer rather than using the `%p` format string which automatically expects 4 bytes in 32-bit processes and 8 bytes in 64-bit processes. This forced a cast to `PVOID` because `dwActiveProcessorMask` is of type `DWORD_PTR` and would generate a warning when used with `%p`.

A better option here is to specify `%zu` or `%zX`, which is used to format `size_t` values, equivalent to `DWORD_PTR`.

Coding Conventions

Having any coding conventions is good for consistency and clarity, but the actual conventions vary, of course. The following coding conventions are used in this book.

- Windows API functions are used with a double colon prefix. Example: `::CreateFile`.
- Type names use Pascal casing (first letter is capital, and every word starts with a capital letter as well. Examples: `Book`, `SolidBrush`. The exception is UI related classes that start with a capital 'C'; this is for consistency with WTL.
- Private member variables in C++ classes start with an underscore and use camel casing (first letter is small and subsequent words start with capital letter). Examples: `_size`, `_isRunning`. The exception is for WTL classes where private member variable names start with `m_`. This is for consistency with ATL/WTL style.
- Variable names do not use the old Hungarian notation. However, there may be some occasional exceptions, such as an `h` prefix for a handle and a `p` prefix for a pointer.
- Function names follow the Windows API convention to use Pascal casing.
- When common data types are needed such as vectors, the C++ standard library is used unless there is good reason to use something else.

- We'll be using the *Windows Implementation Library* (WIL) from Microsoft, released in a Nuget package. This library contains helpful types for easier working with the Windows API. A brief introduction to WIL is in the next chapter.
- Some samples have a user interface. The book uses the *Windows Template Library* (WTL) to simplify UI-related code. You can certainly use some other library for UI such as MFC, Qt, straight Windows API, or even .NET libraries such as WinForms or WPF (assuming you know how to call native functions from .NET). UI is not the focus of this book. If you need more details on native Windows UI development, consult the classic "Programming Windows", 6th edition, by Charles Petzold.

> *Hungarian notation* uses prefixes to make variable names hint at their type. Examples: `szName`, `dwValue`. This convention is now considered obsolete, although parameter names and structure members in the Windows API use it a lot.

There are some more coding conventions used later in this book, which will be described when these become relevant.

C++ Usage

The code samples in this book make some use of C++. We won't be using any "complex" C++ features, but mostly features that enhance productivity, help with error avoidance. Here are the main C++ features we'll use:

- 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 parts of the code.
- The `new` and `delete` operators.
- Scoped enums (`enum class`).
- Classes with member variables and functions.
- Templates, where they make sense.
- Constructors and destructors, especially for building RAII (*Resource Acquisition Is

Initialization*) types. RAII will be discussed in greater detail in the next chapter.

Handling API Errors

Windows API function may fail for a variety of reasons. Unfortunately, the way a function indicates success or failure is not consistent across all functions. That said, there are very few cases, briefly described in table 1-3.

function return type	Success is ...	Failure is ...	How to get the error number
`BOOL`	not `FALSE` (0)	`FALSE` (0)	call `GetLastError`
`HANDLE`	not `NULL` (0) and not `INVALID_HANDLE_VALUE` (-1)	0 or -1	call `GetLastError`
`void`	cannot fail (usually)	None	Not needed, but in rare cases throws an SEH exception
`LSTATUS` or `LONG`	`ERROR_SUCCESS` (0)	greater than zero	return value is the error itself
`HRESULT`	greater or equal to zero, usually `S_OK` (0)	negative number	return value is the error itself
other	depends	depends	look up function documentation

The most common case is returning a `BOOL` type. The `BOOL` type is not the same as the C++ `bool` type; `BOOL` is in fact a 32-bit signed integer. A non-zero return value indicates success, while a returned zero (`FALSE`) means the function has failed. It's important not to test against the `TRUE` (1) value explicitly, since a success can sometimes return a value different from one. If the function fails, the actual error code is available by calling `GetLastError`, responsible for storing the last error from an API function occurring on the current thread. Put another way, each thread has its own last error value, which makes sense in a multi-threaded environment like Windows - multiple threads may call API functions at the same time.

Here is an example of handling such an error:

```
BOOL success = ::CallSomeAPIThatReturnsBOOL();
if(!success) {
    // error - handle it (just print it in this example)
    printf("Error: %d\n", ::GetLastError());
}
```

The second item from table 1-3 is for functions returning `void`. There are actually very few such functions, and most cannot fail. Unfortunately, there are very few such functions that can actually fail in extreme circumstances ("extreme" usually means very low memory resources) with a *Structured Exception Handling* (SEH) exception. We'll discuss SEH in chapter 20. You probably don't need to worry too much about such functions, because if one of these does fail, it means the entire process and possibly the system is in big trouble anyway.

Next, there are functions returning `LSTATUS` or `LONG`, both of which are just signed 32-bit integers. The most common APIs using these scheme are the registry functions we'll meet in chapter 17. These functions return `ERROR_SUCCESS` (0) if successful. Otherwise, the returned value is the error itself (calling `GetLastError` is not needed).

Next on the list from table 1-3 is the `HRESULT` type, which is yet again a signed 32-bit integer. This return type is common for *Component Object Model* (COM) functions (COM is discussed in chapter 18). Zero or a positive value indicates success, while a negative value indicates an error, identified by the returned value. In most cases, checking for success or failure is done with the `SUCCEEDED` or `FAILED` macros, respectfully, returning just `true` or `false`. In rare cases the code would need to look at the actual value.

The Windows headers contain a macro to convert a Win32 error (`GetLastError`) to an appropriate `HRESULT`: `HRESULT_FROM_WIN32`, which is useful if a COM method needs to return an error based on a failed `BOOL`-returning API.
Here is an example for handling an `HRESULT` based error:

```
IGlobalInterfaceTable* pGit;
HRESULT hr = ::CoCreateInstance(CLSID_StdGlobalInterfaceTable, nullptr, CLSCTX_\
ALL,
    IID_IGlobalInterfaceTable, (void**)&pGit);
if(FAILED(hr)) {
    printf("Error: %08X\n", hr);
}
else {
    // do work
    pGit->Release();    // release interface pointer
}
```

> Don't worry about the details of the above code. Chapter 21 is dedicated to COM.

The last entry in table 1-3 is for "other" functions. For example, the `FormatMessage` function we met a few sections ago returns a `DWORD` indicating the number of characters copied to the provided buffer, or zero if the function fails. There are no hard and fast rules for these kinds of functions - the documentation is the best guide. Luckily, there are not that many of them.

Defining Custom Error Codes

The error code mechanism exposed by `GetLastError` can be used by applications as well to set error codes in a similar vein. This is accomplished by calling `SetLastError` with the error to set on the current thread. A function can use one of the many predefined error codes, or it can define its own error codes. To prevent any collision with system-defined codes, the application should set bit 29 in the defined error code.

Here is an example of a function that uses this technique:

```
#define MY_ERROR_1 ((1 << 29) | 1)
#define MY_ERROR_2 ((1 << 29) | 2)

BOOL SomeApi1(int32_t, int32_t*);
BOOL SomeApi2(int32_t, int32_t*);

bool DoWork(int32_t value, int32_t* result) {
    int32_t result1;
    BOOL ok = ::SomeApi1(value, &result1);
    if (!ok) {
        ::SetLastError(MY_ERROR_1);
        return false;
    }

    int32_t result2;
    ok = ::SomeApi2(value, &result2);
    if (!ok) {
        ::SetLastError(MY_ERROR_2);
        return false;
    }
```

```
    *result = result1 + result2;
    return true;
}
```

Note that in my functions, I'm free to use the C++ `bool` type which can be `true` or `false`, rather being a 32-bit integer (`BOOL`). The custom-defined error codes have bit 29 set, making sure they don't clash with a system defined error code.

The Windows Version

In some cases it's desirable to query the system for the Windows OS version on which the current application is executing. The official version numbers of Windows releases is shown in table 1-4.

Table 1-4: Windows version numbers

Windows release name	Official version number
Windows NT 3.1	3.1
Windows NT 3.5	3.5
Windows NT 4.0	4
Windows 2000	5.0
Windows XP	5.1
Windows Server 2003	5.2
Windows Vista / Server 2008	6.0
Windows 7 / Server 2008 R2	6.1
Windows 8 / Server 2012	6.2
Windows 8.1 / Server 2012 R2	6.3
Windows 10 / Server 2016	10.0

You may be wondering why the version numbers have these values - we'll get to that in a moment. The classic function to get this information is `GetVersionEx`, declared like so:

```
typedef struct _OSVERSIONINFO {
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR  szCSDVersion[ 128 ];     // Maintenance string for PSS usage
} OSVERSIONINFO, *POSVERSIONINFO, *LPOSVERSIONINFO;

BOOL GetVersionEx(
    _Inout_ POSVERSIONINFO pVersionInformation);
```

Using it is fairly straightforward:

```
OSVERSIONINFO vi = { sizeof(vi) };
::GetVersionEx(&vi);

printf("Version: %d.%d.%d\n",
    vi.dwMajorVersion, vi.dwMinorVersion, vi.dwBuildNumber);
```

However, compiling it with recent SDKs causes a compilation error: "error C4996: 'GetVersionExW': was declared deprecated".
The reason will become clear shortly. It's possible to remove this deprecation by adding the following definition before including *<windows.h>*:

```
#define BUILD_WINDOWS
#include <Windows.h>
```

Running the above code snippet on Windows up to 8 (inclusive) gives back the correct Windows version. However, running it on Windows 8.1 or 10 (and their server equivalents), will always display the following output:

```
Version: 6.2.9200
```

This is the Windows version for Windows 8. Why? This was a defensive mechanism devised by Microsoft after some issues applications had on Windows Vista. Since Vista came out on January 2006, almost five years after Windows XP, many applications were built in the XP days, and some went to the trouble of checking the minimum Windows version to be XP using the following code:

```
OSVERSIONINFO vi = { sizeof(vi) };
::GetVersionEx(&vi);

if(vi.dwMajorVersion >= 5 && vi.dwMinorVersion >= 1) {
    // XP or later: good to go?
}
```

This code is buggy, because it doesn't foresee the possibility of having a major number of 6 or higher with a minor version of zero. So, for Vista, the above condition failed and would notify the user "Please use XP or later". The correct check would have been this:

```
if(vi.dwMajorVersion > 5 || (vi.dwMajorVersion == 5 && vi.dwMinorVersion >= 1) {
    // XP or later: good to go!
}
```

Unfortunately, too many applications had that bug, and so Microsoft decided for Windows 7 not to increase the major version number but only increment the minor number to 1; this solved the bug. What about Windows 8? Microsoft was still afraid of the above bug and so incremented the minor number only, giving 6.2. The reasoning for Windows 8.1 was similar (6.3). But what about Windows 10? Should the version be 6.4? This seems like a total defeat - how long can Microsoft continue leaving the major version as 6? Well, Windows 10 has version number 10.0. Does that mean all is well? Not really. As we saw, the call to `GetVersionEx` returns the Windows 8 numbers even on Windows 10. What gives?

A new feature has been introduced (called *Switchback*) that returned the Windows version as no higher than 8 (6.2) to prevent compatibility issues, *unless* the application in question has declared its knowledge of a higher-version Windows in existence. This is accomplished using a manifest file - an optional XML file with configuration information - that can be used to indicate a specific Windows version awareness, from Vista to 10.

> This is not just for manipulating the returned version number, but also for some behavioral changes of some APIs for compatibility. This is accomplished using *Shims*, which changes API behavior depending on the selected OS version.

In Visual Studio, a manifest can be added by following these steps:

- Add an XML file to the project with a name like *manifest.xml*. This will hold the manifest file's contents.
- Fill in the manifest (shown after this list).

- Open *Project/Properties* and navigate to the *Manifest Tool* node, *Input and Output*. In *Additional Manifest Files*, type the name of the manifest file (figure 1-8).
- Build the project normally.

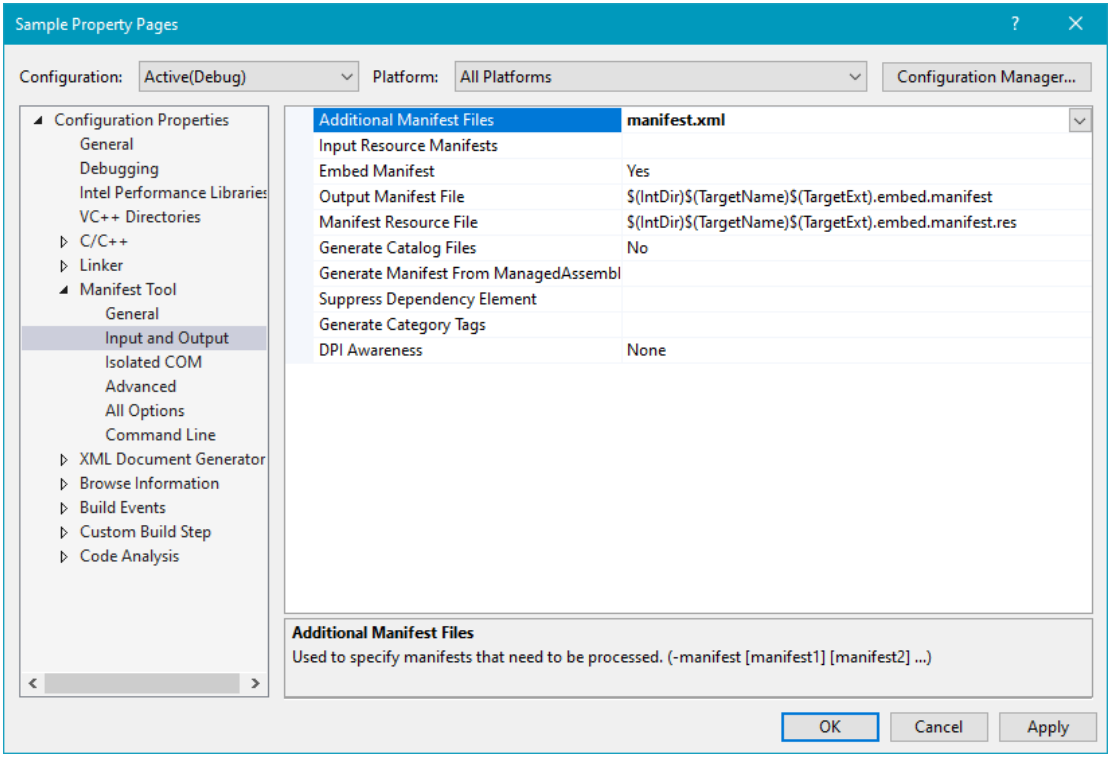


Figure 1-8: Setting manifest file

Notice the setting *Embed Manifest* = *Yes* in figure 1-8. This embeds the manifest as a resource in the executable rather than leaving it as a loose file in the same directory as the executable and always named `{exename}.exe.manifest`.

The manifest can have several elements, but we focus on just one in this chapter (we'll examine others in due course). Here it is:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
    <application>
      <!-- Windows Vista -->
      <!--<supportedOS Id="{e2011457-1546-43c5-a5fe-008deee3d3f0}" />-->

      <!-- Windows 7 -->
      <!--<supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2440225f93a}" />-->

      <!-- Windows 8 -->
      <!--<supportedOS Id="{4a2f28e3-53b9-4441-ba9c-d69d4a4a6e38}" />-->

      <!-- Windows 8.1 -->
      <!--<supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}" />-->

      <!-- Windows 10 -->
      <!--<supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}" />-->
    </application>
  </compatibility>
</assembly>
```

The easiest way to get a nice manifest file to tweak is (ironically perhaps) create a simple console C# application, then add to the project an *Application Manifest File* item, which will generate the above XML, among other elements.

The GUIDs for the various OS versions have been created when those versions were released. This means there is no way an application developed in the Windows 7 days could get a version of Windows 10, for example.
If you uncomment the Windows 8.1 version for instance and re-run the application, the output would be:

```
Version: 6.3.9600
```

If you uncomment the Windows 10 GUID (whether the Windows 8.1 GUID is commented out or not is unimportant), you'll get the real Windows 10 version (if running on a Windows 10 machine, of course):

```
Version: 10.0.18362
```

Getting the Windows Version

Given that `GetVersionEx` is deprecated (at least for the reasons discussed in the previous section), what is the proper way to get the Windows version? A new set of APIs is available that can give back the result, but not in a simple numerical sense, but by returning true/false for Windows version questions. These are available in the `<versionhelpers.h>` header file.

Here are some of the functions included: `IsWindowsXPOrGreater`, `IsWindowsXPSP3OrGreater`, `IsWindows7OrGreater`, `IsWindows8Point1OrGreater`, `IsWindows10OrGreater`, `IsWindowsServer`. Their usage is straightforward - they accept nothing and return `TRUE` or `FALSE`. Their implementation uses another version-related function, `VerifyVersionInfo`:

```
BOOL VerifyVersionInfo(
    _Inout_ POSVERSIONINFOEX pVersionInformation,
    _In_    DWORD dwTypeMask,
    _In_    DWORDLONG dwlConditionMask);
```

This function knows how to compare version numbers based on the specified criteria (`dwlConditionMask`), such as the major or minor version numbers. You can find the implementation of all the Boolean functions inside `versionhelper.h`.

> There is an undocumented (but reliable) way to get the version numbers regardless of the manifest file without calling `GetVersionEx`. It's based on a data structure called `KUSER_SHARED_DATA` that is mapped to every process to the same virtual address (`0x7FFE0000`). Its declaration is listed in this Microsoft link: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/ns-ntddk-kuser_shared_data. The Windows version numbers are part of this shared structure in the same offsets. Here is an alternative to showing the Windows version numbers:
>
> ```
> auto sharedUserData = (BYTE*)0x7FFE0000;
> printf("Version: %d.%d.%d\n",
>     *(ULONG*)(sharedUserData + 0x26c),  // major version offset
>     *(ULONG*)(sharedUserData + 0x270),  // minor version offset
>     *(ULONG*)(sharedUserData + 0x260)); // build number offset (Windows 10)
> ```
>
> Of course, it's recommended to use the official APIs rather than `KUSER_SHARED_DATA`.

Exercises

1. Write a console application that prints more information about the system than the *HelloWin* application shown earlier, by calling the following APIs: `GetNativeSystemInfo`, `GetComputerName`, `GetWindowsDirectory`, `QueryPerformanceCounter`, `GetProductInfo`, `GetComputerObjectName`. Handle errors if they occur.

Summary

In this chapter, we looked at the fundamentals of Windows, both in terms of architecture and in terms of programming. In the next chapter, we'll dig into kernel objects and handles, as these form the basis of many aspects of working with Windows.

Chapter 2: Objects and Handles

Windows is an object-based operating system, exposing various types of objects (usually referred to as *kernel Objects*), that provide the bulk of the functionality in Windows. Example object types are processes, threads and files. In this chapter we'll discuss the general theory related to kernel objects without too much details of any specific object type. The following chapters will go into details of many of these types.

In this chapter:

- **Kernel Objects**
- **Handles**
- **Creating Objects**
- **Object Names**
- **Sharing Kernel Objects**
- **Private Object Namespaces**

Kernel 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 (kernel) space, created and managed by the Object Manager (part of the Executive) when requested to do so by user or kernel code. kernel objects are reference counted, so only when the last reference to the object is released will the object be destroyed and freed from memory.

There are quite a few object types supported by the Windows kernel. To get a peek, run the *WinObj* tool from *Sysinternals* (elevated) and locate the *ObjectTypes* directory. Figure 2-1 shows what this looks like. These types can be cataloged based on their visibility and usage:

- Types that are exported to user-mode via the Windows API. Examples: mutex, semaphore, file, process, thread, timer. This book discusses many of these object types.

- Types that are not exported to user mode, but are documented in the *Windows Driver Kit* (WDK) for use by device driver writers. Examples: device, driver, callback.
- Types that are not documented even in the WDK (at least at the time of writing). These object types are for use by the kernel itself only. Examples: partition, keyed event, core messaging.

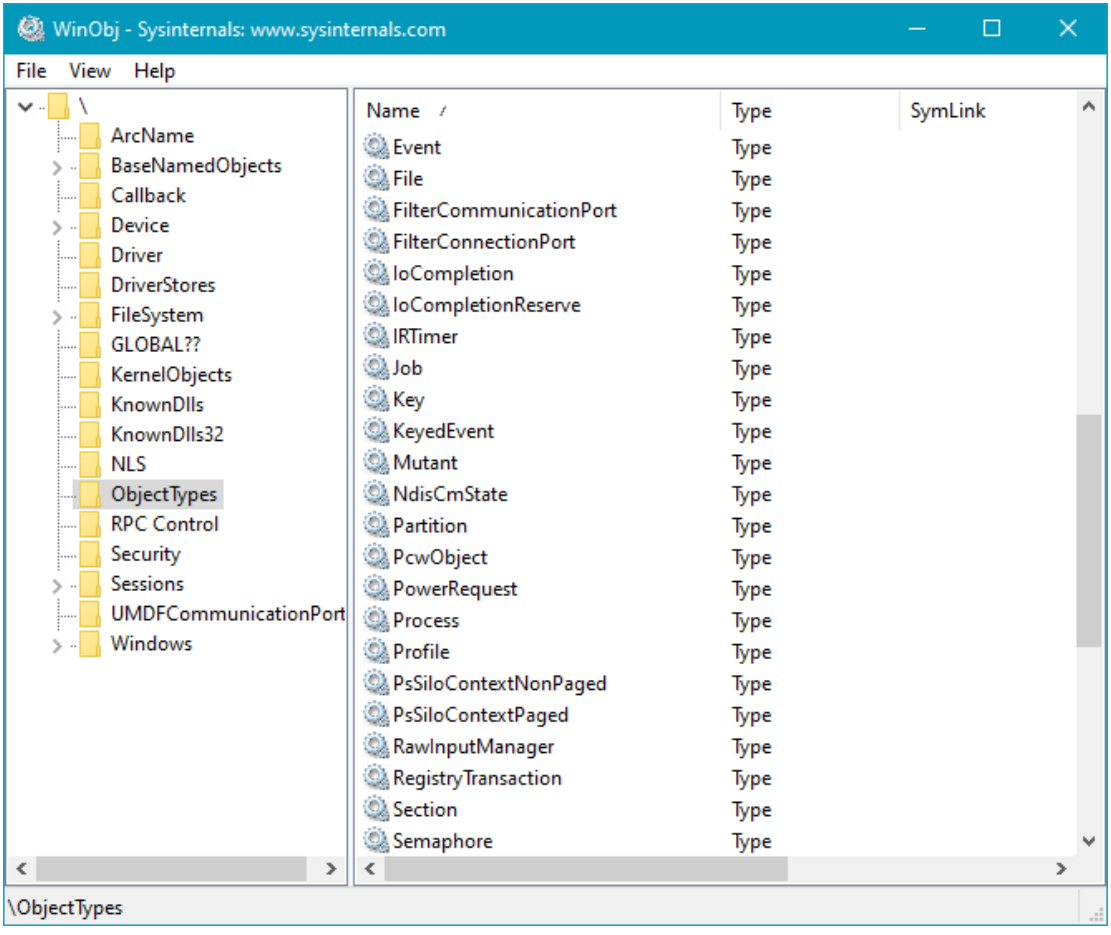


Figure 2-1: Object types

The main attributes of a kernel object are depicted in figure 2-2.

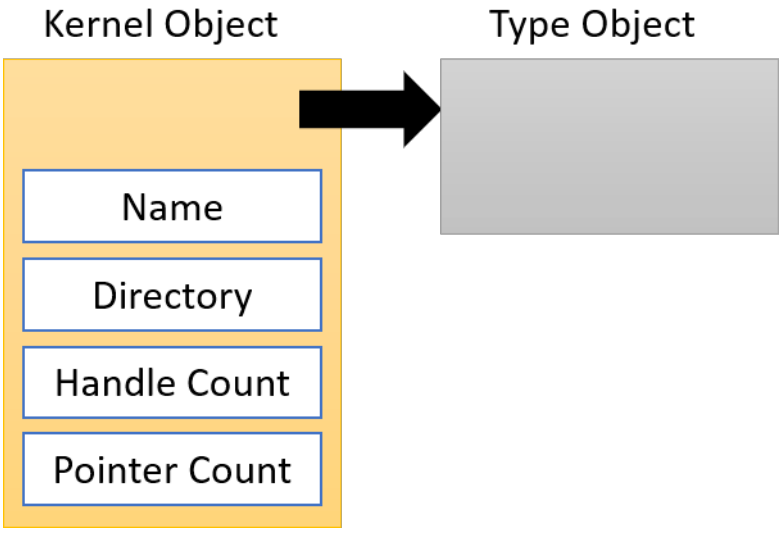


Figure 2-2: Kernel Object Attributes

Since kernel objects reside in system space, they cannot be accessed directly from user mode. Applications must use an indirect mechanism to access kernel objects, known as handles. Handles provide at least the following benefits:

- Any change in the object type's data structure in a future Windows release will not affect any client.
- Access to the object can be controlled through security access checks.
- Handles are private to a process, so having a handle to a particular object in one process means nothing in another process context.

Kernel objects are reference counted. The Object Manager maintains a handle count and a pointer count, the sum of which is the total reference count for an object (direct pointers can be obtained from kernel mode). Once an object used by a user mode client is no longer needed, the client code should close the handle used to access the object by calling `CloseHandle`. From that point on, the code should consider the handle to be invalid. Trying to access the object through the closed handle will fail, with `GetLastError` returning `ERROR_INVALID_HANDLE` (6). The client does not know, in the general case, whether the object has been destroyed or not. The Object Manager will delete the object if its reference drops to zero.

Handle values are multiples of 4, where the first valid handle is 4; Zero is never a valid handle value. This scheme does not change on 64 bit systems.

A handle is logically an index to an array of entries in a handle table maintained on a process by process basis, that points logically 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.

If the object cannot be created or opened, the returned handle is in most cases `NULL` (0). One notable exception to this rule is the `CreateFile` function that returns `INVALID_HANDLE_VALUE` (-1) if it fails.

For example, the `CreateMutex` function allows creating a new mutex or opening a mutex by name (depending whether the mutex with that name exists). If successful, the function returns a handle to the mutex. 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.

If the function succeeds and a name was provided, the returned handle can be to a new mutex or to an existing mutex with that name. The code can check this by calling `GetLastError` and comparing the result to `ERROR_ALREADY_EXISTS`. If it is, then it's not a new object, but rather another handle to an existing object. This is one of those rare cases where `GetLastError` can be called even if the API in question succeeded.

Running a Single Instance Process

One fairly well-known usage for the `ERROR_ALREADY_EXIST` case is limiting an executable to have a single process instance. Normally, if you double-click an executable in Explorer, a new process is spawned based on that executable. If you repeat this operation, another process is created based on the same executable. What if you wanted to prevent the second process from launching, or at least have it shut down if it detects another process instance with the same executable already running.

The trick is using some named kernel object (a mutex is usually employed, although any named object type can be used instead), where an object with a particular name is created. If the object already exists, there must be another instance already running, so the process can shut down (possibly notifying its sibling of that fact).

The *SingleInstance* demo application demonstrates how this can be achieved. It's a dialog-based application built with WTL. Figure 2-3 shows what this application looks like running. If you try launching more instances of this application, you'll find that the first window logs messages coming from the new process instance that then exits.

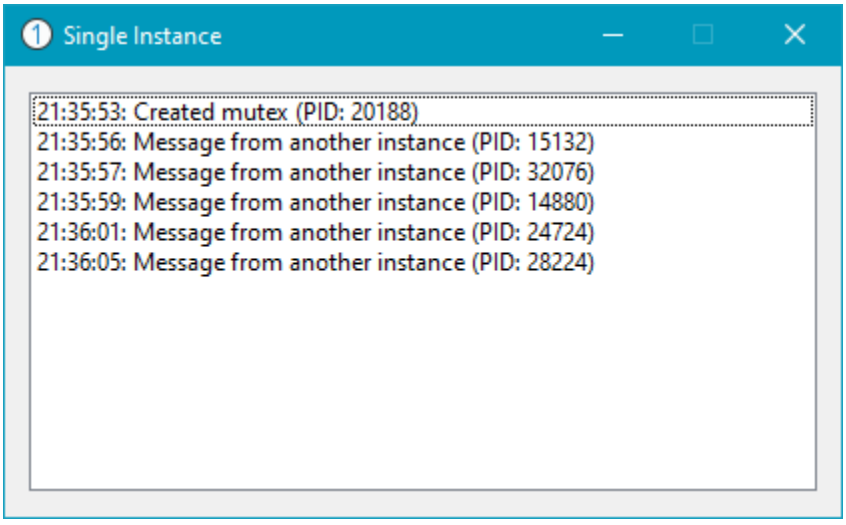


Figure 2-3: The Single Instance application

In the `WinMain` function, we create the mutex first. If this fails, then something is very wrong and we bail out.

```
HANDLE hMutex = ::CreateMutex(nullptr, FALSE, L"SingleInstanceMutex");
if (!hMutex) {
    CString text;
    text.Format(L"Failed to create mutex (Error: %d)", ::GetLastError());
    ::MessageBox(nullptr, text, L"Single Instance", MB_OK);
    return 0;
}
```

Failure to create the mutex should be extremely rare. The most likely scenario for failure is that another kernel object (which is not a mutex) with that same name already exists.

Now that we get a proper handle to the mutex, the only question is whether the mutex was actually created or we received another handle to an existing mutex (presumably created by a previous instance of this executable):

```
if (::GetLastError() == ERROR_ALREADY_EXISTS) {
    NotifyOtherInstance();
    return 0;
}
```

If the object existed prior to the `CreateMutex` call, then we call a helper function that sends some message to the existing instances and exits. Here is `NotifyOtherInstance`:

```
#define WM_NOTIFY_INSTANCE (WM_USER + 100)

void NotifyOtherInstance() {
    auto hWnd = ::FindWindow(nullptr, L"Single Instance");
    if (!hWnd) {
        ::MessageBox(nullptr, L"Failed to locate other instance window",
            L"Single Instance", MB_OK);
        return;
    }

    ::PostMessage(hWnd, WM_NOTIFY_INSTANCE, ::GetCurrentProcessId(), 0);
    ::ShowWindow(hWnd, SW_NORMAL);
    ::SetForegroundWindow(hWnd);
}
```

The function searches for the existing window with the `FindWindow` function and uses the window caption as the search criteria. This is not ideal in the general case, but it's good enough for this sample.

Once the window is located, we send a custom message to the window with the current process ID as an argument. This shows up in the dialog's list box.

The final piece of the puzzle is handling the `WM_NOTIFY_INSTANCE` message by the dialog. In WTL, window messages are mapped to functions using macros. The message map of the dialog class (`CMainDlg`) in *MainDlg.h* is repeated here:

```
BEGIN_MSG_MAP(CMainDlg)
    MESSAGE_HANDLER(WM_NOTIFY_INSTANCE, OnNotifyInstance)
    MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
    COMMAND_ID_HANDLER(IDCANCEL, OnCancel)
END_MSG_MAP()
```

The custom message is mapped to the `OnNotifyInstance` member function, implemented like so:

```
LRESULT CMainDlg::OnNotifyInstance(UINT, WPARAM wParam, LPARAM, BOOL &) {
    CString text;
    text.Format(L"Message from another instance (PID: %d)", wParam);

    AddText(text);
    return 0;
}
```

The process ID is extracted from the `wParam` parameter and some text is added to the list box with the `AddText` helper function:

```
void CMainDlg::AddText(PCWSTR text) {
    CTime dt = CTime::GetCurrentTime();
    m_List.AddString(dt.Format(L"%T") + L": " + text);
}
```

`m_List` is of type `CListBox`, a WTL wrapper for a Windows list box control.

Handles

As mentioned in the previous section, a handle points indirectly to a small data structure in kernel space that holds a few pieces of information for that handle. Figure 2-4 depicts this data structure for 32 bit and 64 bit systems.

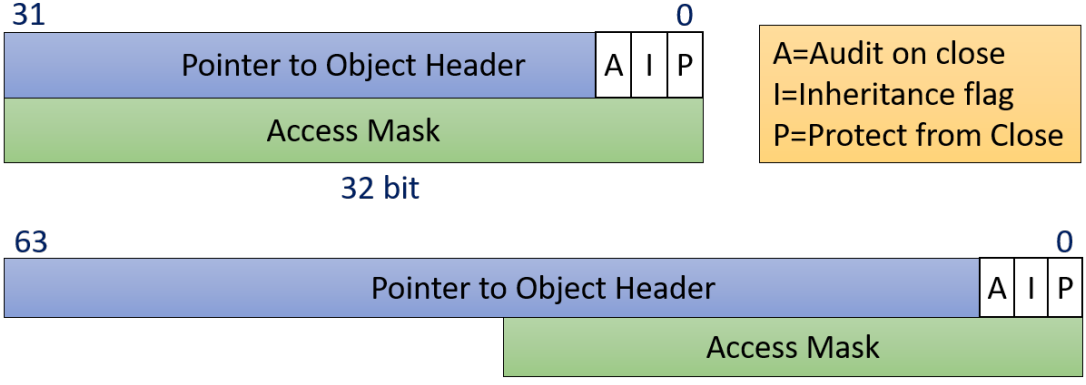


Figure 2-4: Handle Entry

On 32 bit systems, this handle entry is 8 bytes in size, and it's 16 bytes in size on 64 bit systems (technically 12 bytes are enough, but it's extended to 16 bytes for alignment purposes). Each entry has the following ingredients:

- Pointer to the actual object. Since the lower bits are used for flags and to improve CPU access times by address alignment, an object's address is multiple of 8 on 32 bit systems and multiple of 16 on 64 bit systems.
- Access mask, indicating what can be done with this handle. In other words, the access mask is the power of the handle.
- Three flags: Inheritance, Protect from close and Audit on close (discussed shortly).

The access mask is a bitmask, where each "1" bit indicating a certain operation that can be carried using that handle. The access mask is set when the handle is created by creating an object or opening an existing object. If the object is created, then the caller typically has full access to the object. But if the object is opened, the caller needs to specify the required access mask, which it may or may not get.

For example, if an application wants to terminate a certain 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 is an example of terminating a process given its 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 `OpenProcess` function has the following prototype:

```
HANDLE OpenProcess(
  _In_ DWORD dwDesiredAccess,   // the access mask
  _In_ BOOL  bInheritHandle,    // inheritance flag
  _In_ DWORD dwProcessId);      // process ID
```

Since this is an *Open* operation, the object in question already exists, the client needs to specify what access mask it requires to access the object. An access mask has two types of access bits: generic and specific. We'll discuss these details in chapter 16 ("Security"). One of the specific access bits for a process is `PROCESS_TERMINATE` used in the above example. Other bits include `PROCESS_QUERY_INFORMATION`, `PROCESS_VM_OPERATION` and more. Refer to the documentation of `OpenProcess` to locate the complete list.

What access mask should be used by client code? Generally, it should reflect the operations the client code intends to perform with the object. Asking for more than needed may fail, and asking less is obviously not good enough.

The flags associated with each handle are the following:

- Inheritance - this flag is used for handle inheritance - a mechanism that allows sharing an object between cooperating processes. We'll discuss handle inheritance in chapter 3.
- Audit on close - this flag indicates whether an audit entry in the security log should be written when that handle is closed. This flag is rarely used and is off by default.
- Protect from close - setting this flag prevents the handle from being closed. A call to `CloseHandle` will return `FALSE` and `GetLastError` returns `ERROR_INVALID_HANLDLE` (6). If the process is running under a debugger, an exception is raised with the following message: "0xC0000235: NtClose was called on a handle that was protected from close via NtSetInformationObject". This flag is rarely useful.

Changing the inheritance and protection flags can be done with the `SetHandleInformation` function defined like so:

```
#define HANDLE_FLAG_INHERIT             0x00000001
#define HANDLE_FLAG_PROTECT_FROM_CLOSE  0x00000002

BOOL SetHandleInformation(
    _In_ HANDLE hObject,
    _In_ DWORD dwMask,
    _In_ DWORD dwFlags);
```

The first parameter is the handle itself. The second parameter is a bit mask indicating which flags to operate on. The last parameter is the actual value for these flags. For example, to set the "protect from close" bit on some handle, the following code could be used:

```
::SetHandleInformation(h, HANDLE_FLAG_PROTECT_FROM_CLOSE,
    HANDLE_FLAG_PROTECT_FROM_CLOSE);
```

Conversely, the following code snippet removes this same bit:

```
::SetHandleInformation(h, HANDLE_FLAG_PROTECT_FROM_CLOSE, 0);
```

The opposite function to read back these flags exists as well:

```
BOOL GetHandleInformation(
    _In_ HANDLE hObject,
    _Out_ LPDWORD lpdwFlags);
```

The handles opened from a particular process can be viewed with the *Process Explorer* tool from *Sysinternals*. Navigate to a process you're interested in, and make sure the lower pane is visible (*View* menu, *Show Lower Pane*). The lower pane shows one of two views - switch to Handle view (*View* menu, *Lower Pane View*, *Handles*). Figure 2-5 is a screenshot of the tool showing open handles in an *Explorer* process. The columns shown by default are *Type* and *Name* only. I added the following columns by right-clicking the header area and clicking *Select Columns*: *Handle*, *Object Address*, *Access* and *Decoded Access*.

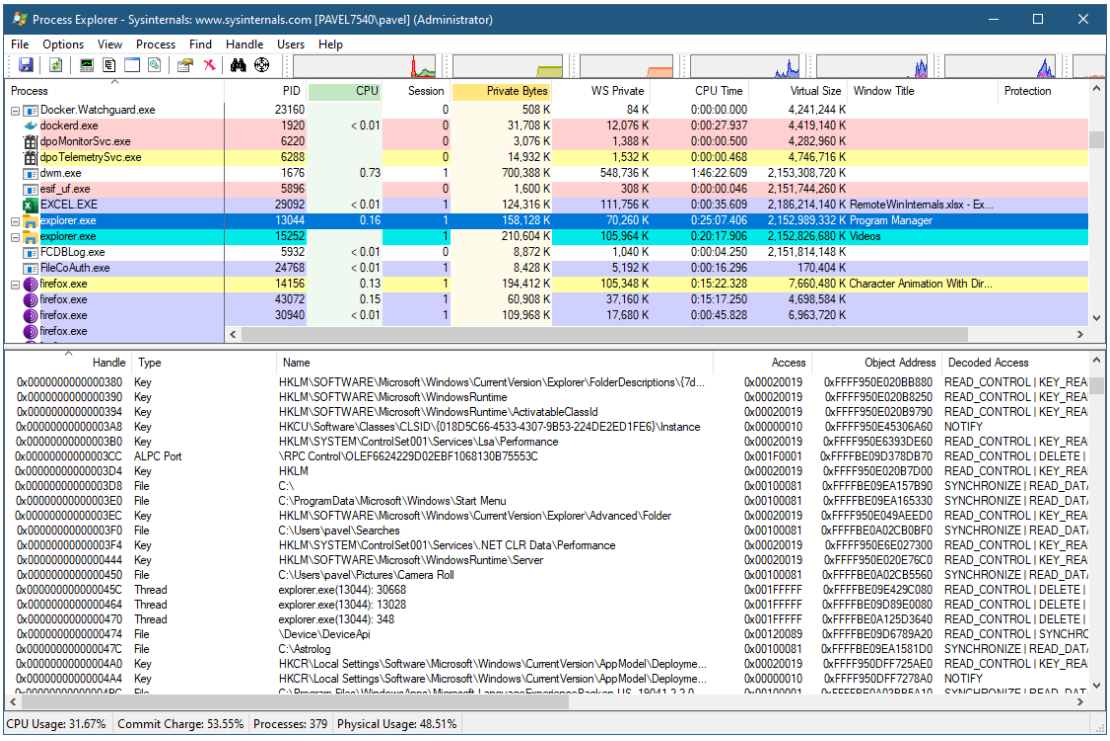


Figure 2-5: Handle View in Process Explorer

Here is a brief description of the columns:

- Handle - this is the handle value itself, relevant to this process only. The same handle value can have a different meaning, i.e. - points to a different object, or maybe even an empty index.
- Type - the object type name. This corresponds to the *Object Types* directory in *WinObj* shown in figure 2-1.
- Object Address - this is the kernel address where the real object structure resides. Notice these addresses end with a zero hex digit on 64 bit (on 32 bit systems, the addresses end with "8" or "0"). There is nothing user-mode code with this information, but it can be used for debugging purposes: if you have two handles to an object and you want to know whether they point to the same object you can compare object addresses; if they are the same, it's the same object. Otherwise, the handles point to different objects.
- Access - this is the access mask discussed above. To interpret the bits stored in this hex value, you need to locate the access mask bits in the documentation. To alleviate that, use the *Decoded Access* column.
- Decoded Access - provides a string representation of the access mask bits for common object types. This makes it easier to interpret the access mask bits without digging into the documentation.

> I personally implemented this column to *Process Explorer*.

Process Explorer's handle view shows only handles to named objects by default. To view all handles, Enable *Show unnamed handles and mappings* option from the *View* menu. Figure 2-6 shows how the view changes when this option is checked.

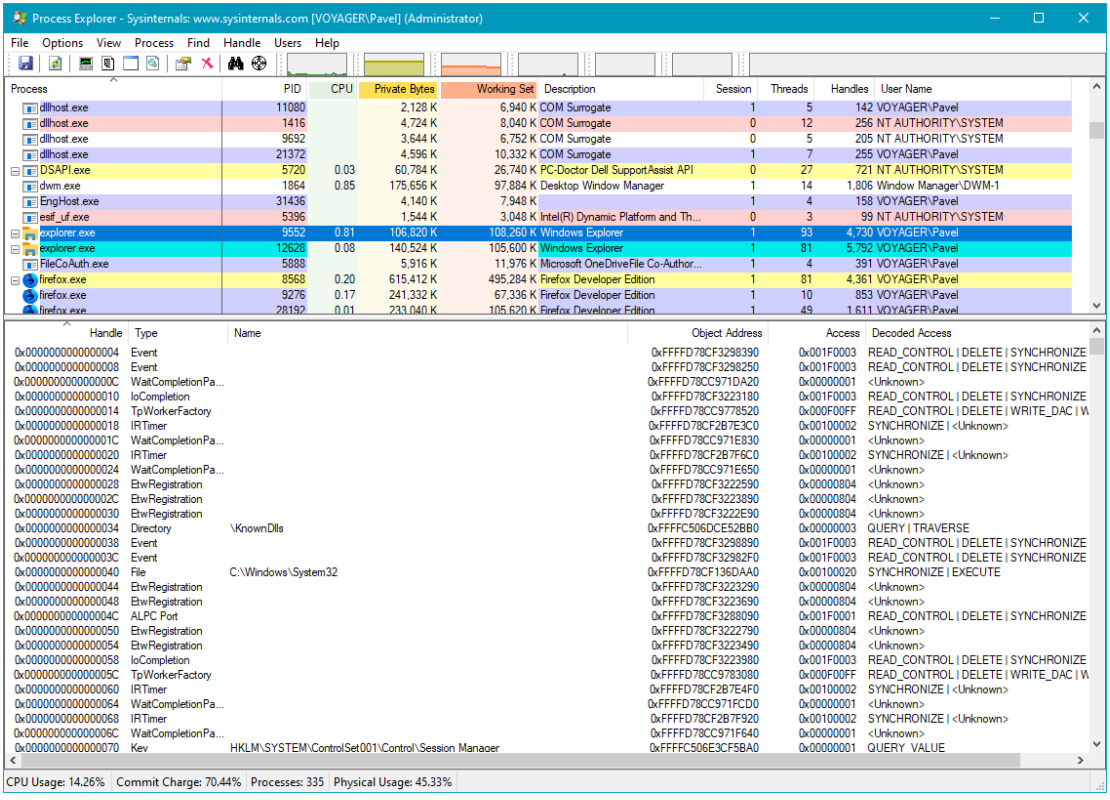


Figure 2-6: Handle View in Process Explorer (including unnamed objects)

The term "Name" is trickier than it seems. What *Process Explorer* considers named objects are not necessarily actual names, but in some cases are convenient monikers. For example, process and thread handles are shown in figure 2-5, even though processes and threads cannot have string-based names. There are other object types with a "Name" which is not their name; the most confusing are *File* and *Key*. We'll discuss this "weirdness" in the section "Object Names", later in this chapter.

The total number of handles in a process' handle table is available as a column in *Process Explorer* and *Task Manager*. Figure 2-7 shows this column added to *Task Manager*.

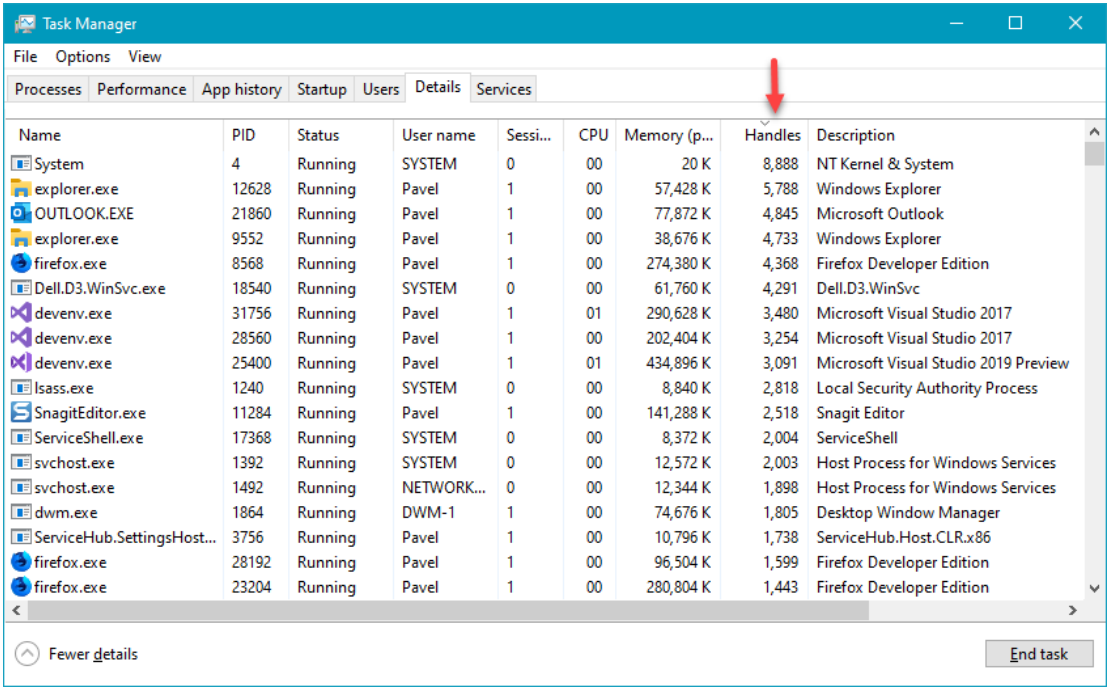


Figure 2-7: Handle count column in Task Manager

Note that the number shown is the handle count, rather than the object count. This is because more than one handle can exist that reference the same object.

Double-clicking a handle entry in *Process Explorer* opens a dialog that shows some properties of the object (not the handle). Figure 2-8 is a screenshot of such a dialog.

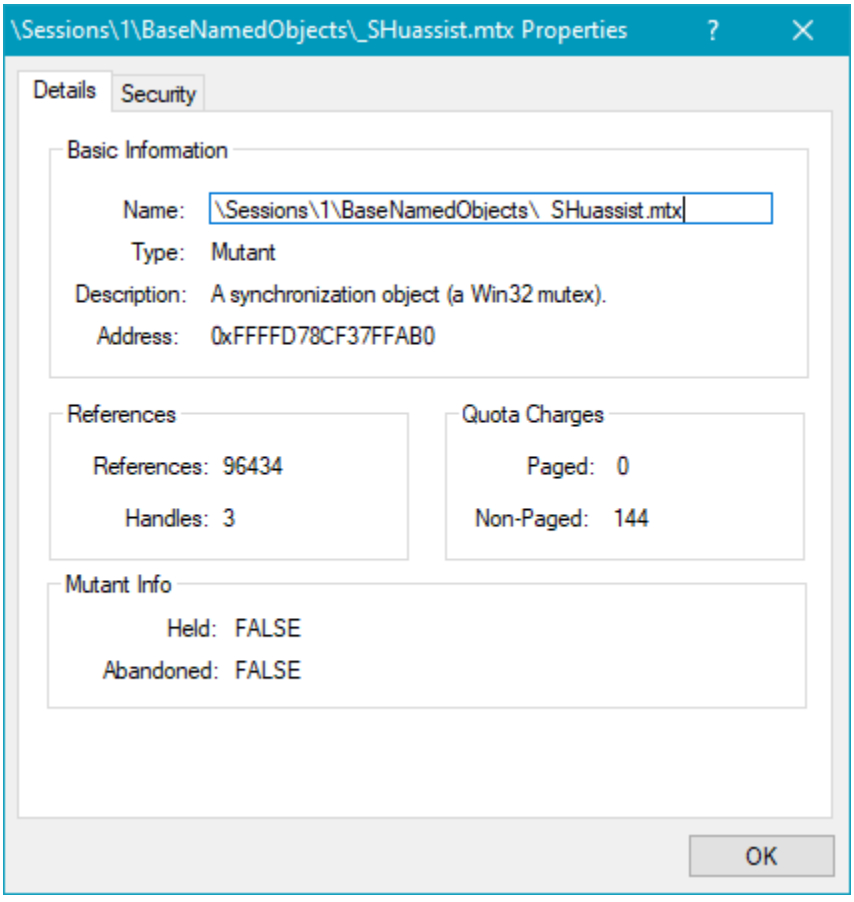
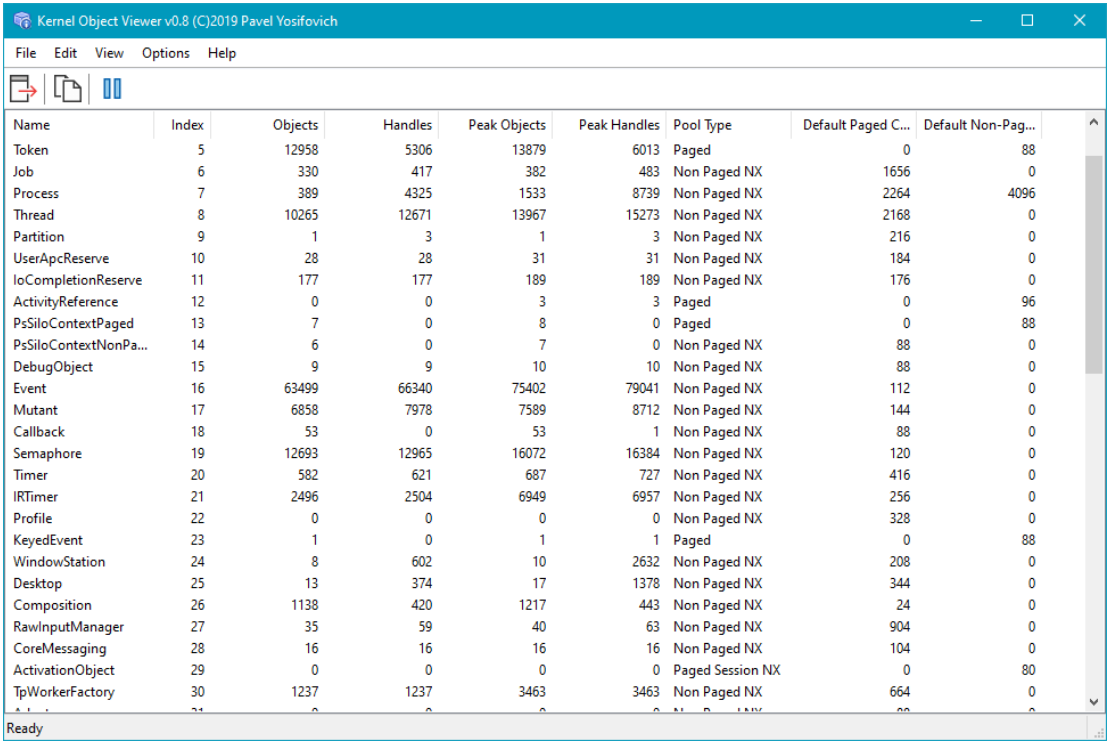


Figure 2-8: Kernel object properties in Process Explorer

The basic object information is repeated from the handle entry (name, type, and address). This particular object (a mutex) has 3 open handles. The references number is misleading and does not reflect the actual object reference count. For some types of objects (such as mutexes), extra information is shown. In this particular case, it's whether the mutex is currently held and whether it's abandoned. (we'll discuss mutexes in detail in chapter 8).

To get a sense of the number of objects and handles in the system at a given moment, you can run the *KernelObjectView* tool from my Github repository at https://github.com/zodiacon/AllTools. Figure 2-9 shows a screenshot of the tool. The total number of objects (per object type) is shown along with the total number of handles. You can sort by any column; which object types have the most objects? The most handles?



Name	Index	Objects	Handles	Peak Objects	Peak Handles	Pool Type	Default Paged C...	Default Non-Pag...
Token	5	12958	5306	13879	6013	Paged	0	88
Job	6	330	417	382	483	Non Paged NX	1656	0
Process	7	389	4325	1533	8739	Non Paged NX	2264	4096
Thread	8	10265	12671	13967	15273	Non Paged NX	2168	0
Partition	9	1	3	1	3	Non Paged NX	216	0
UserApcReserve	10	28	28	31	31	Non Paged NX	184	0
IoCompletionReserve	11	177	177	189	189	Non Paged NX	176	0
ActivityReference	12	0	0	3	3	Paged	0	96
PsSiloContextPaged	13	7	0	8	0	Paged	0	88
PsSiloContextNonPa...	14	6	0	7	0	Non Paged NX	88	0
DebugObject	15	9	9	10	10	Non Paged NX	88	0
Event	16	63499	66340	75402	79041	Non Paged NX	112	0
Mutant	17	6858	7978	7589	8712	Non Paged NX	144	0
Callback	18	53	0	53	1	Non Paged NX	88	0
Semaphore	19	12693	12965	16072	16384	Non Paged NX	120	0
Timer	20	582	621	687	727	Non Paged NX	416	0
IRTimer	21	2496	2504	6949	6957	Non Paged NX	256	0
Profile	22	0	0	0	0	Non Paged NX	328	0
KeyedEvent	23	1	0	1	1	Paged	0	88
WindowStation	24	8	602	10	2632	Non Paged NX	208	0
Desktop	25	13	374	17	1378	Non Paged NX	344	0
Composition	26	1138	420	1217	443	Non Paged NX	24	0
RawInputManager	27	35	59	40	63	Non Paged NX	904	0
CoreMessaging	28	16	16	16	16	Non Paged NX	104	0
ActivationObject	29	0	0	0	0	Paged Session NX	0	80
TpWorkerFactory	30	1237	1237	3463	3463	Non Paged NX	664	0

Figure 2-9: Kernel object Viewer

Pseudo Handles

Some handles have special values and are not closable. These are known as *pseudo-handles*, although they are used just like any other handle when needed. Calling `CloseHandle` on pseudo-handles always fails. Here are the functions returning pseudo-handles:

- `GetCurrentProcess` (-1) - return a pseudo-handle to the calling process
- `GetCurrentThread` (-2) - return a pseudo-handle to the calling thread
- `GetCurrentProcessToken` (-4) - return a pseudo-handle to the token of the calling process
- `GetCurrentThreadToken` (-5) - return a pseudo-handle to the token of the calling thread
- `GetCurrentThreadEffectiveToken` (-6) - return a pseudo-handle to the effective token of the calling thread (if the thread has its own token - it's used, otherwise - its process token is used)

The last three pseudo handles (token handles) are only supported on Windows 8 and later, and their access mask is `TOKEN_QUERY` and `TOKEN_QUERY_SOURCE` only.

> Processes, threads, and tokens are discussed later in this book.

RAII for Handles

It's important to close a handle once it's no longer needed. Applications that fail to do that properly may exhibit "handle leak", where the number of handles grows uncontrollably if the application opens handles but "forgets" to close them. Obviously, this is bad.

One way to help code manage handles without forgetting to close them is to use C++ by implementing a well-known idiom called *Resource Acquisition is Initialization* (RAII). The name is not that good, but the idiom is. The idea is to use a destructor for a handle wrapped in a type that ensures the handle is closed when that wrapper object is destroyed.

Here is a simple RAII wrapper for a handle (implemented inline for convenience):

```
struct Handle {
    explicit Handle(HANDLE h = nullptr) : _h(h) {}
    ~Handle() { Close(); }

    // delete copy-ctor and copy-assignment
    Handle(const Handle&) = delete;
    Handle& operator=(const Handle&) = delete;

    // allow move (transfer ownership)
    Handle(Handle&& other) : _h(other._h) {
        other._h = nullptr;
    }
    Handle& operator=(Handle&& other) {
        if (this != &other) {
            Close();
            _h = other._h;
            other._h = nullptr;
        }
        return *this;
    }

    operator bool() const {
        return _h != nullptr && _h != INVALID_HANDLE_VALUE;
```

```
    }

    HANDLE Get() const {
        return _h;
    }

    void Close() {
        if (_h) {
            ::CloseHandle(_h);
            _h = nullptr;
        }
    }

private:
    HANDLE _h;
};
```

The `Handle` type provides the basic operations expected from a RAII `HANDLE` wrapper. The copy constructor and copy assignment operators are removed, as copying a handle that may have multiple owners does not make sense (causing `CloseHandle` to be called twice for the same handle). It is possible to implement these copy operations by duplicating the handle (see "Sharing Kernel Objects" later in this chapter), but it's a non-trivial operation best avoided in implicit copy scenarios. A `bool` operator returns `true` if the current handle held is valid; it considers zero and `INVALID_HANDLE_VALUE` (-1) as invalid handles. The `Close` function closes the handle and is normally called from the destructor. Finally, the `Get` function returns the underlying handle.

> It's possible to add an implicit conversion operator to `HANDLE`, removing the need to call `Get`.

Here is some example code using the above wrapper:

```
Handle hMyEvent(::CreateEvent(nullptr, TRUE, FALSE, nullptr));
if (!hMyEvent) {
    // handle failure
    return;
}
::SetEvent(hMyEvent.Get());

// move ownership
Handle hOtherEvent(std::move(hMyEvent));
::ResetEvent(hOtherEvent.Get());
```

Although writing such a RAII wrapper is possible, it's usually best to use an existing library that provides this (and other similar) functionality. For example, although `CloseHandle` is the most common closing handle function, there are other types of handles that require a different closing function. One such library that is used by Microsoft in Windows code is the *Windows Implementation Library* (WIL). This library has been released on Github and is available as a *Nuget* package.

Using WIL

Adding WIL to a project is done like any other *Nuget* package. Right-click the *References* node in a Visual Studio project and select *Manage Nuget Packages...*. In the *Browse* tab's search text box, type "wil" to quickly search for WIL. The full name of the package is "Microsoft.Windows.ImplementationLibrary", shown in figure 2-10.

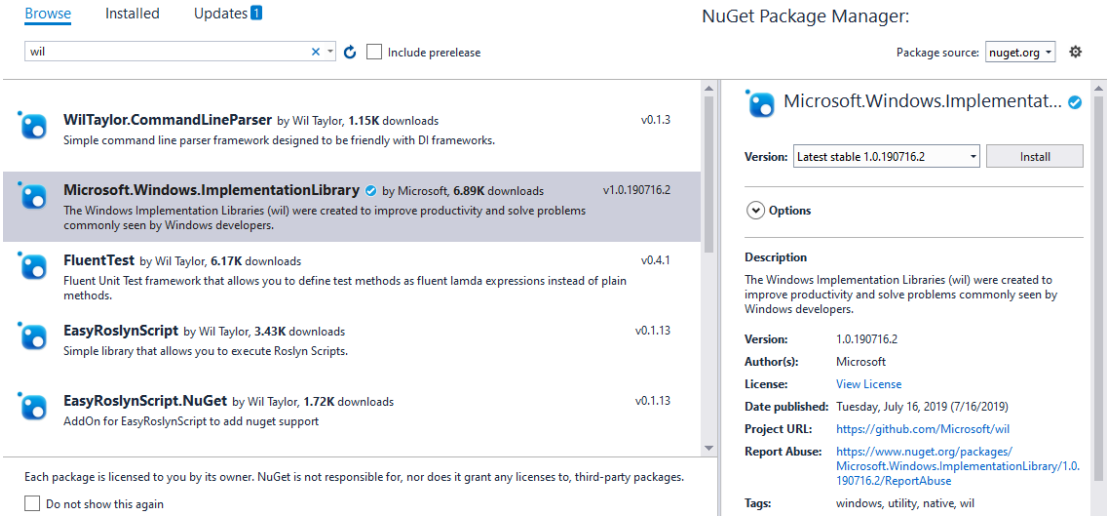


Figure 2-10: Adding WIL via Nuget

The RAII handle wrapper(s) are located in the `<wil\resource.h>` header file.

Here is the same code using WIL:

```
#include <wil\resource.h>

void DoWork() {
    wil::unique_handle hMyEvent(::CreateEvent(nullptr, TRUE, FALSE, nullptr));
    if (!hMyEvent) {
        // handle failure
        return;
    }
    ::SetEvent(hMyEvent.get());

    // move ownership
    auto hOtherEvent(std::move(hMyEvent));
    ::ResetEvent(hOtherEvent.get());
}
```

`wil::unique_handle` is a `HANDLE` wrapper that calls `CloseHandle` upon destruction. It's modeled mostly after the C++ `std::unique_ptr<>` type. Notice that getting the internal `HANDLE` is done by calling `get()`. To replace the value inside a `unique_handle` (and close the old one) use the `reset` function; calling `reset` with no arguments just closes the underlying handle, making the wrapper object an empty shell.

The code can be somewhat simplified by adding `using namespace wil;` so that `wil::` need not be prepended for every type in WIL. Also, notice `auto` can be used to simplify code in some cases.

> The code samples in this book use WIL in some cases, but not all. From a learning perspective, it's sometimes better to use the raw types to make things simpler to understand.

Creating Objects

All the functions for creating new objects have some common parameters. Here is the `CreateMutex` and `CreateEvent` functions to demonstrate:

```
HANDLE CreateMutex(
    _In_opt_ LPSECURITY_ATTRIBUTES lpMutexAttributes,
    _In_ BOOL bInitialOwner,
    _In_opt_ LPCTSTR lpName);

HANDLE CreateEvent(
    _In_opt_ LPSECURITY_ATTRIBUTES lpEventAttributes,
    _In_ BOOL bManualReset,
    _In_ BOOL bInitialState,
    _In_opt_ LPCTSTR lpName);
```

Notice both functions accept a parameter of type `SECURITY_ATTRIBUTES`. This structure is common to virtually all *Create* functions is defined like so:

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD nLength;
    LPVOID lpSecurityDescriptor;
    BOOL bInheritHandle;
} SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES;
```

The `nLength` member should be set to the size of the structure. This is a common technique used by Windows to version structures. If the structure will have new members in a future Windows release, old code would still function properly because it would have set the length to the old size, so the newer Windows API know not to look at the new members as the old code had no idea these existed. That said, the `SECURITY_ATTRIBUTES` structure has yet to change from the first Windows NT release.

As its name implies, the structure has to do with security settings on the newly created object. The main member that is really about security is `lpSecurityDescriptor`, which can point to a security descriptor object, which essentially specifies who-can-do-what with the object. We'll discuss security descriptors in chapter 16.

The last member, `bInheritHandle` has security implications, which is why it's hosted in this structure as well. It's the inheritance bit mentioned earlier. This means the inheritance bit can be set by using this structure without needing to call `SetHandleInformation` when creating a new object. Here is an example for creating an event object with the returned handle having its inheritance bit set:

```
SECURITY_ATTRIBUTES sa = { sizeof(sa) };    // set nLength and zero the rest
sa.bInheritHandle = TRUE;

HANDLE hEvent = ::CreateEvent(&sa, TRUE, FALSE, nullptr);
DWORD flags;
::GetHandleInformation(hEvent, &flags);        // sets flags=1
```

> Handle inheritance is discussed in chapter 3.

Passing `NULL` for `SECURITY_ATTRIBUTES` leaves the inheritance bit clear. In terms of security, this means "default security" which is based on a security descriptor that is stored in the process access token. We'll discuss the details in chapter 16. In any case, using `NULL` for the security descriptor (either explicitly or passing `NULL` for the `SECURITY_ATTRIBUTES` pointer) is the right thing to do in most cases.

Object Names

Some types of objects can have string-based 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. Named objects can be viewed with the *WinObj* tool from *Sysinternals*.

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. In this case the parameters that affect object creation such as the `SECURITY_ATTRIBUTES` structure are not used, as the creator already set this up.

The name provided to a *Create* function is not the final name of the object. In classic (desktop) processes, 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* only. 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\AppContainerNamedObjects*.

Figure 2-11 shows named objects in session 1 in *WinObj*.

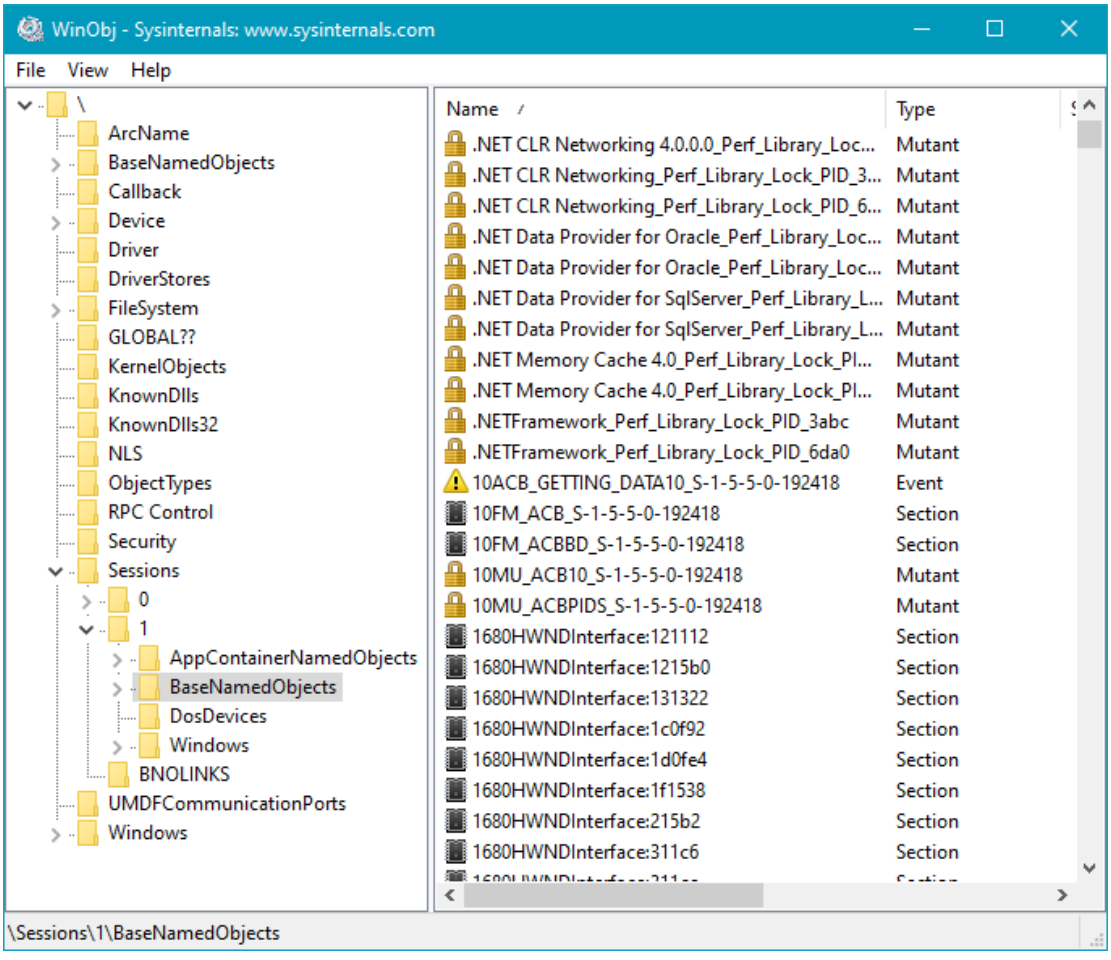


Figure 2-11: Named objects in session 1

Figure 2-12 shows named objects in session 0.

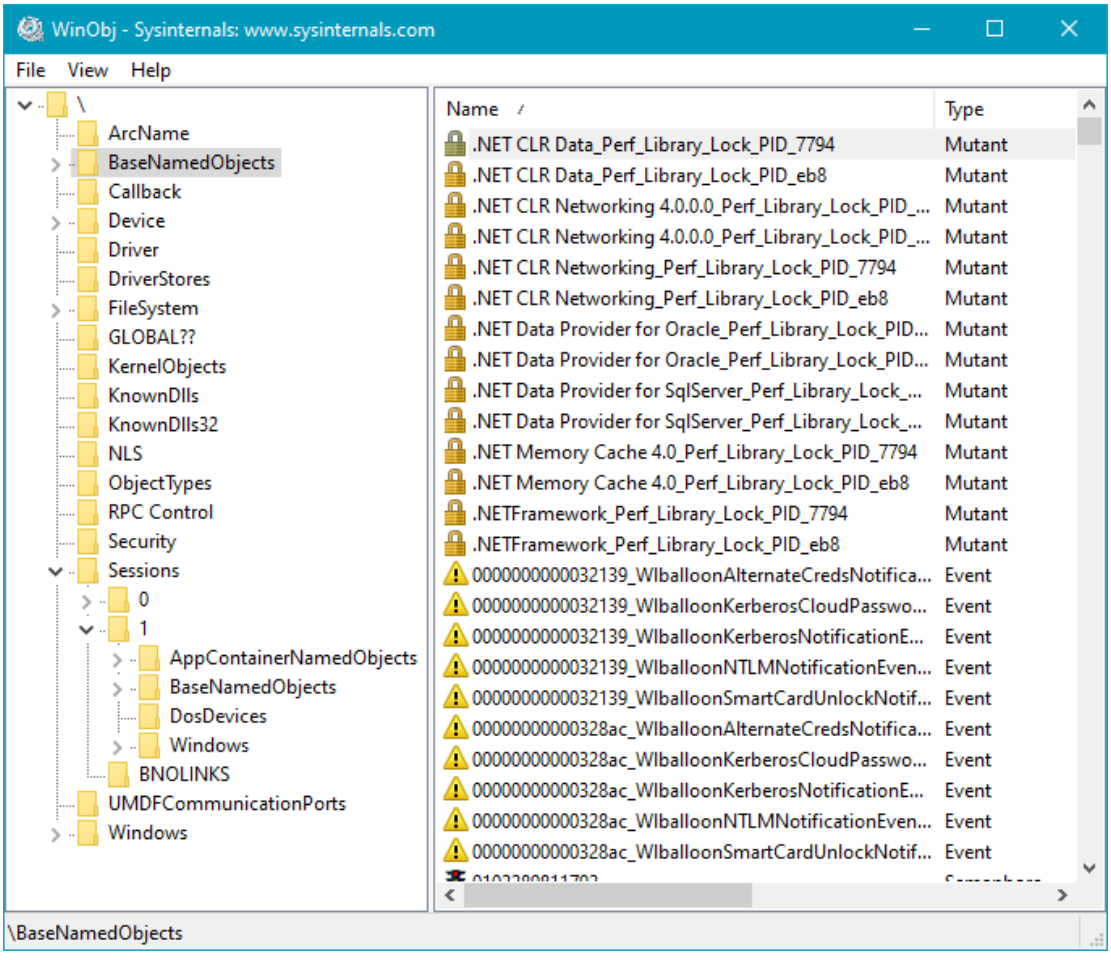


Figure 2-12: Named objects in session 0

All the above means is that object names are session-relative (and in the case of an 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.

The entire Object Manager namespace hierarchy can be viewed with *WinObj*. This entire structure is held in memory and manipulated by the Object Manager as needed. 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.

> The "directories" shown in *WinObj* actually Directory objects, which are just one kind of kernel object which acts as a logical container.

Going back to *Process Explorer*'s Handles view - it shows by default "named" objects. "Named" here means not just objects that can be named, but also other objects. Objects that can be named are Mutexes (Mutants), Semaphores, Events, Sections, ALPC Ports, Jobs, Timers, and other, less used object types. Yet others are shown with a name that has a different meaning than a true named object:

- Process and Thread objects - the name is shown as their unique ID.
- For File objects it shows the file name (or device name) pointed to by the file object. It's not the same as an object's name, as there is no way to get a handle to a file object given the file name - only a new file object may be created that accesses the same underlying file or device (assuming sharing settings for the original file object allow it).
- (Registry) Key object names are shown with the path to the registry key. This is not a name, for the same reasoning as for file objects.
- Directory objects show their logical path, rather than being a true object name. A Directory is not a file system object, but rather an object manager directory.
- Token object names are shown with the user name stored in the token.

> To verify the above statements, browse through *WinObj* and look for File or Key objects. You won't find any, which suggests these objects cannot be named.

Sharing Kernel Objects

As we've seen, handles to kernel objects are private to a process. In some cases, a process may want to share a kernel object with another process. Such a process cannot simply pass somehow the value of the handle to the other process, because in the other process' handle table that handle value may point to a different object or be empty.

Clearly, some mechanism must be in place to allow such sharing. In fact, there are three:

- Sharing by name

- Sharing by handle inheritance
- Sharing by duplicating handles

We'll look at the first and third option here, and discuss handle inheritance in the next chapter.

Sharing by Name

This is the simplest option, if available. "Available" here means that the object in question can have a name, and does have a name. The typical scenario is that the cooperating processes (2 or more) would call the appropriate *Create* function with the same object name. The first process to make the call would create the object, and subsequent calls from the other processes would open additional handles to the same object.

The sample *BasicSharing* shows an example of using sharing by name with a *Memory Mapped File* object. This object can be used to share memory between processes (normally, each process can only see its own address space). Running two instances (or more) of the application (shown in figure 2-13) allows sharing textual data between these processes.

> The full details of memory-mapped files are discussed in chapter 14.

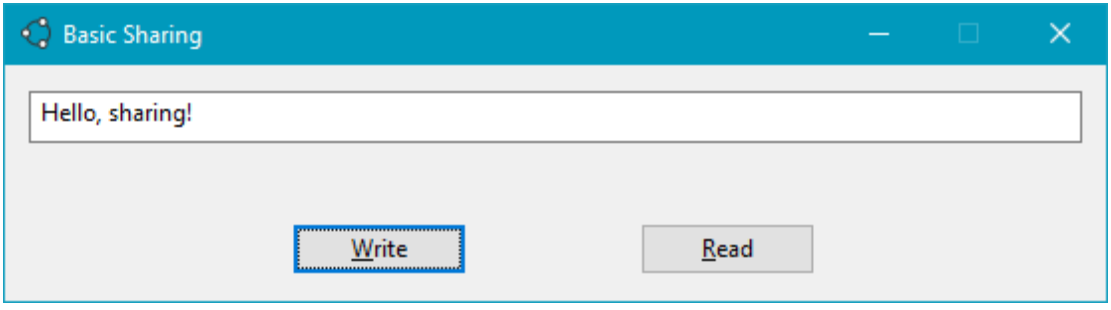


Figure 2-13: The Basic Sharing application

To test it out, type something in the edit box and click *Write*. Then switch to another instance, and just click *Read*. The text you entered should appear in the other application's edit box. Of course you can swap roles. If you launch another instance, you can click *Read* and the last text would appear as well. This is because all these processes are reading and writing to the same (shared) memory.

By the way, these don't have to be processes based on the same executable - this is just used here for convenience. The determining factor is the object's name.

Before we look at the code, let's see what this looks like in *Process Explorer*. Run two instances of the executable, open *Process Explorer* and locate the two processes. Make sure the lower pane shows Handles (and not DLLs). The object type to look for is *Section* (the kernel name of Memory Mapped File). Find a section called "MySharedMemory" (with the session-based prefix of course), as shown in figure 2-14.

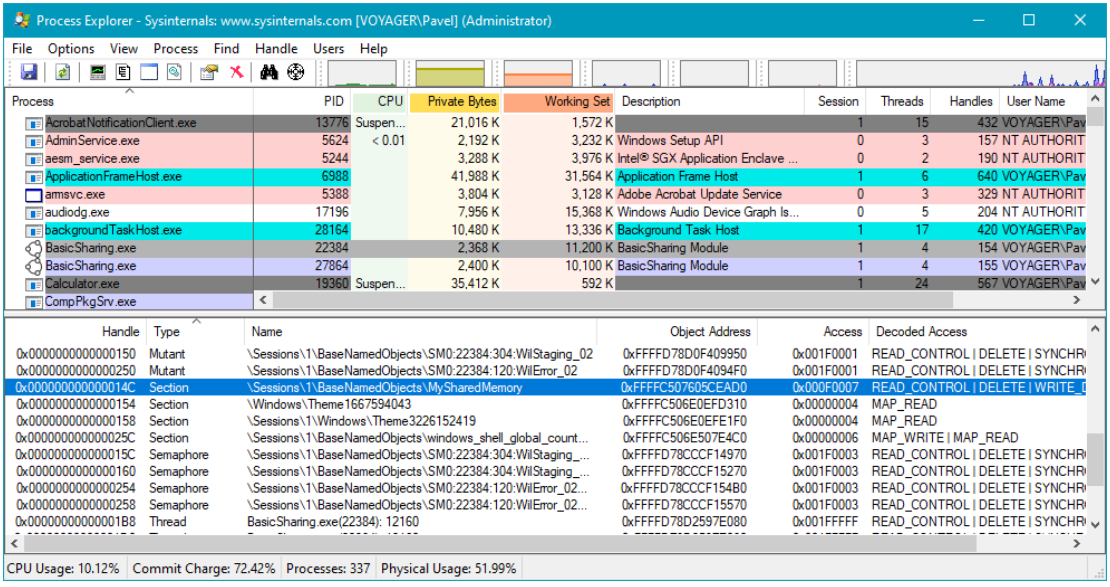


Figure 2-14: The shared section object

If you double-click the handle, you should see the properties of the section object as shown in figure 2-15.

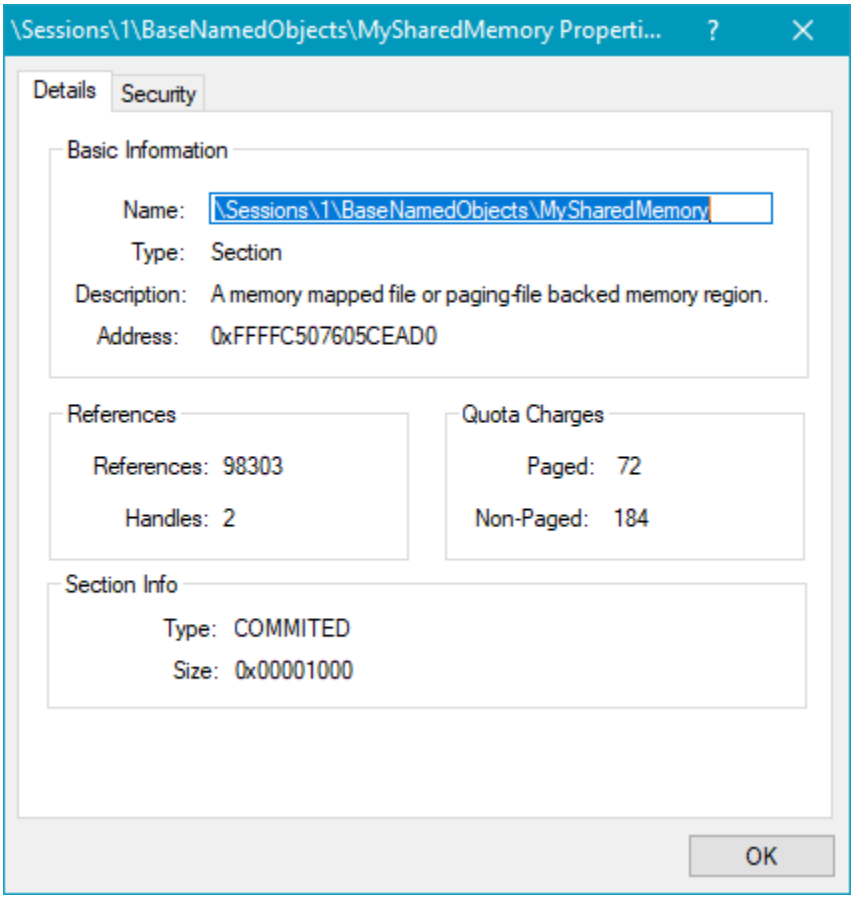


Figure 2-15: Section object properties

Notice there are two open handles to the object. Presumably, these are coming from the two processes holding handles to that object. Notice the shared memory's size: 4 KB - we'll see this reflected in the code.

If you locate the second process using this object (see figure 2-16), you should find the same information presented when double-clicking the handle. How can you be sure these are pointing to the same object? Look at the *Object Address* column. If the address is identical, this is the same object (and vice versa). Notice also that the handle values are not the same (the normal case). In figures 2-14 the handle value is 0x14c (PID 22384) and in figure 2-16 it's 0x16c (PID 27864). Still - they reference the exact same object.

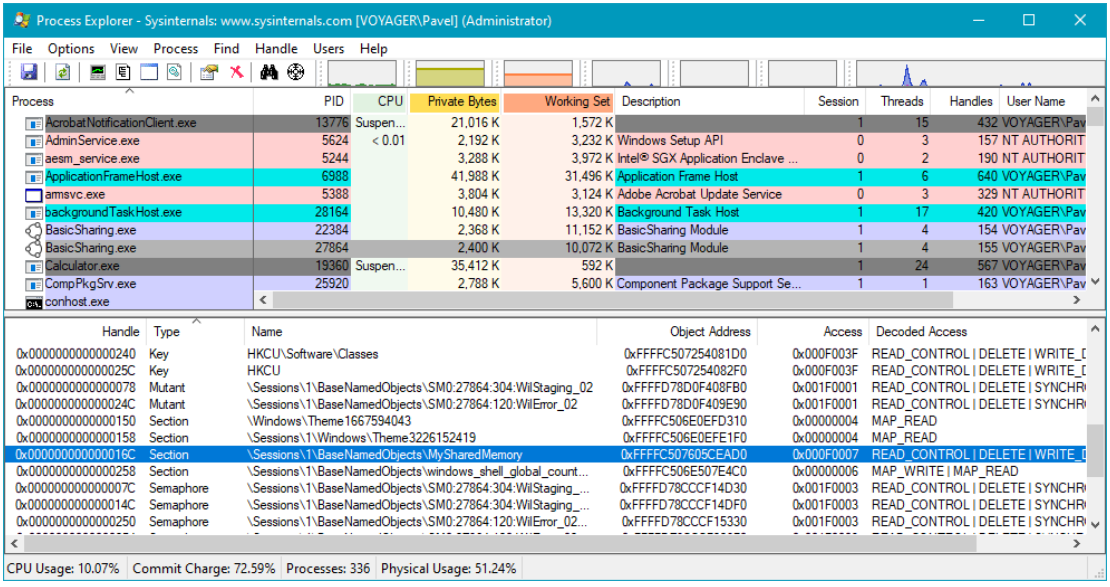


Figure 2-16: The shared section in the other process

If you were to close one of the instances, what would happen? One handle would close, but the object remains alive. This means that launching a completely new instance and clicking *Read* will show the most recent text. What would happen if we close all cooperating applications and then launch one instance again. What would we see if we click *Read*? Try and explain to yourself why this is the case.

Now let's turn our attention to the code.

BasicApplication is a WTL dialog-based project. The dialog box class (`CMainDlg`) holds a single member of interest, which is the handle to the memory-mapped file:

```
private:
    HANDLE m_hSharedMem;
```

When the dialog is created, in the `WM_INITDIALOG` message handler, we create the file mapping object and giving it a name:

```
m_hSharedMem = ::CreateFileMapping(INVALID_HANDLE_VALUE, nullptr, PAGE_READWRIT\
E,
    0, 1 << 12, L"MySharedMemory");
if (!m_hSharedMem) {
    AtlMessageBox(m_hWnd, L"Failed to create/open shared memory", IDR_MAINFRAME\
);
    EndDialog(IDCANCEL);
}
```

`CreateFileMapping` is used to create (or open) a file mapping object. The exact details of the parameters are discussed in chapter 14 (in part 2). Here we care about one parameter in particular (the last) - the object's name. This is the name we've seen in *Process Explorer* (with the standard session-related prefix). If this is the first process to attempt creating the object - it's created. Subsequent calls result in additional handles to the same object (calling `GetLastError` would return `ERROR_ALREADY_EXISTS`). In this case, we don't care whether this call is the first or not - we just want a handle to the same kernel object so that its "function" is available from multiple processes.

The second to last argument pair (`0` and `1 << 12`) determine the size of the shared memory as a 64-bit value. In this case it's set to 4 KB (`1 << 12`). If the call fails for any reason we just print a simple message and close the dialog, causing the process itself to exit.

When the dialog is closed, it's a good idea to close the handle. Strictly speaking, it's not necessary to do that in this particular case, because once the dialog is closed, the process exits, and the kernel ensures that all handles from a terminated process are properly closed. Still, it's a good habit to have (unless some RAII wrapper for the handle does that for you). For completeness, here is the call to close handle when handling the `WM_DESTROY` message for the dialog:

```
if (m_hSharedMem)
    ::CloseHandle(m_hSharedMem);
```

Now for the write and read parts. Accessing the shared memory is done by calling `MapViewOfFile`, resulting in a pointer to the shared memory (again, the exact details are in chapter 12). Then it's just a matter of copying the text to that mapped memory:

```
void* buffer = ::MapViewOfFile(m_hSharedMem, FILE_MAP_WRITE, 0, 0, 0);
if (!buffer) {
    AtlMessageBox(m_hWnd, L"Failed to map memory", IDR_MAINFRAME);
    return 0;
}

CString text;
GetDlgItemText(IDC_TEXT, text);
::wcscpy_s((PWSTR)buffer, text.GetLength() + 1, text);

::UnmapViewOfFile(buffer);
```

The copying is done with `wcscpy_s` to the mapped memory. Then the memory is unmapped with `UnmapViewOfFile`.

Reading data is very similar. The access mask is changed to `FILE_MAP_READ` rather than `FILE_MAP_WRITE`, and memory is copied in the other direction, directly to the edit box:

```
void* buffer = ::MapViewOfFile(m_hSharedMem, FILE_MAP_READ, 0, 0, 0);
if (!buffer) {
    AtlMessageBox(m_hWnd, L"Failed to map memory", IDR_MAINFRAME);
    return 0;
}

SetDlgItemText(IDC_TEXT, (PCWSTR)buffer);
::UnmapViewOfFile(buffer);
```

Sharing by Handle Duplication

Sharing kernel objects by name is certainly simple. What about objects that don't (or can't have a name)? Handle duplication may be the answer. Handle duplication has no inherent restrictions (except security) - it can work on almost any kernel object, named or unnamed and it works at any point in time (in chapter 3 we'll see that handle inheritance is only available when a process creates a child process). There is a dent, however; this is the most difficult way of sharing in practice, as we shall soon see.

A Duplicated I/O completion port handle does not work in the target process.

Duplicating a handle is as simple as calling the `DuplicateHandle` function:

```
BOOL DuplicateHandle(
    _In_ HANDLE hSourceProcessHandle,
    _In_ HANDLE hSourceHandle,
    _In_ HANDLE hTargetProcessHandle,
    _Outptr_ LPHANDLE lpTargetHandle,
    _In_ DWORD dwDesiredAccess,
    _In_ BOOL bInheritHandle,
    _In_ DWORD dwOptions);
```

Duplicating a handle requires a source process, source handle and a target process. If successful, a new handle entry is written to the target process handle table, pointing to the same object as the source handle. The "before" and "after" duplication are depicted in figures 2-17 and 2-18, respectively.

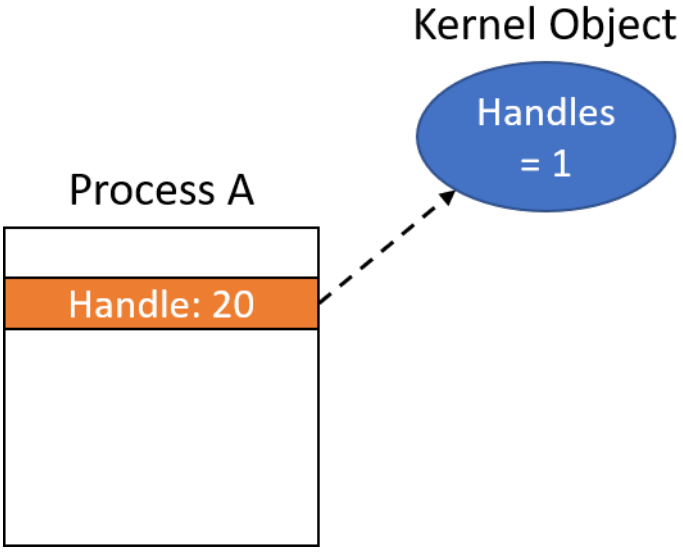


Figure 2-17: Before handle duplication

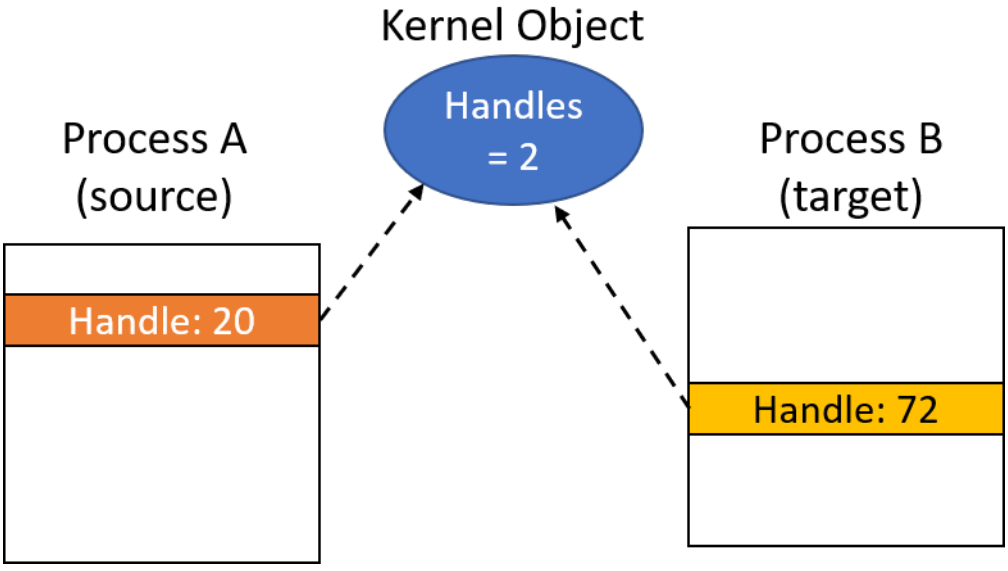


Figure 2-18: After handle duplication

Technically, `DuplicateHandle` can work on any two processes for which proper handles can be obtained, but the typical scenario is duplication one of the caller's handles into another process' handle table. Also, the source and target processes may be the same. Let's go over the parameters of `DuplicateHandle` in details:

- `hSourceProcessHandle` - this is a handle to the source process. This handle must have the `PROCESS_DUP_HANDLE` access mask. If the source is the caller's process than passing `GetCurrentProcess` will do the trick (and it always has full access).
- `hSourceHandle` - the source handle to duplicate. This handle must be valid in the context of the source process.
- `hTargetProcessHandle` - the target process handle. Typically some call to `OpenProcess` must be used to gain such a handle. As with the source process, the `PROCESS_DUP_HANDLE` access mask is required.
- `lpTargetHandle` - this is the resulting handle, **valid from the target process perspective**. In figure 2-18, the resulting handle returned to the caller was 72. This value is with respect to Process B (the caller is assumed to be process A).
- `dwDesiredAccess` - the desired access mask for the duplicated handle. If the `dwOptions` parameter has the flag `DUPLICATE_SAME_ACCESS`, then this access mask is ignored. Otherwise, this is the access mask to request for the new handle.
- `bInheritHandle` - specifies whether the new handle is inheritable or not (see chapter 3 for more on handle inheritance).

- `dwOptions` - a set of flags. One is `DUPLICATE_SAME_ACCESS` discussed above. The second supported is `DUPLICATE_CLOSE_SOURCE`; if specified, closes the source handle after a successful duplication (this means the handle count for the object is not incremented).

Here is a simple example of creating a job object and duplicating a handle to it in the same process while reducing the access mask (error handling omitted):

```
HANDLE hJob = ::CreateJobObject(nullptr, nullptr);
HANDLE hJob2;
::DuplicateHandle(::GetCurrentProcess(), hJob, ::GetCurrentProcess(), &hJob2,
    JOB_OBJECT_ASSIGN_PROCESS | JOB_OBJECT_TERMINATE, FALSE, 0);
```

The source and target process are the current process. Running this piece of code and looking at the handles in *Process Explorer* shows the differences (figure 2-19).

Handle	Type	Name	Access	Decoded Access
0x000000000000008C	IoCompletion		0x001F0003	READ_CONTROL | DELETE | SYNCHRONIZE | WRITE_DAC | WRITE_OWNER | <Unknown>
0x000000000000001C	IRTimer		0x00100002	SYNCHRONIZE | <Unknown>
0x0000000000000024	IRTimer		0x00100002	SYNCHRONIZE | <Unknown>
0x0000000000000090	IRTimer		0x00100002	SYNCHRONIZE | <Unknown>
0x000000000000009C	IRTimer		0x00100002	SYNCHRONIZE | <Unknown>
0x00000000000000AC	Job		0x001F003F	READ_CONTROL | DELETE | SYNCHRONIZE | WRITE_DAC | WRITE_OWNER | JOB_OBJECT_ALL_ACCESS
0x00000000000000B0	Job		0x00000009	ASSIGN_PROCESS | TERMINATE
0x0000000000000008	Key	HKLM\SOFTWAR...	0x00000009	QUERY_VALUE | ENUMERATE_SUB_KEYS

Figure 2-19: Simple handle duplication

One handle (0xac) has full access to the job object, while the other (duplicated) handle (0xb0) has just the specified desired access mask.

In the more common case, a handle from the current process is duplicated to a target cooperating process. The following function will duplicate a source handle from the current process to a target process:

```
HANDLE DuplicateToProcess(HANDLE hSource, DWORD pid) {
    // open a strong-enough handle to the target process
    HANDLE hProcess = ::OpenProcess(PROCESS_DUP_HANDLE, FALSE, pid);
    if (!hProcess)
        return nullptr;

    HANDLE hTarget = nullptr;

    // duplicate
    ::DuplicateHandle(::GetCurrentProcess(), hSource, hProcess,
        &hTarget, 0, FALSE, DUPLICATE_SAME_ACCESS);

    // cleanup
```

```
    ::CloseHandle(hProcess);

    return hTarget;
}
```

This is the case where handle duplication becomes non-trivial. It's not the act of duplication itself - that's rather simple - a single function call. The problem is how to convey the information to the target process. Two pieces of information must be conveyed to the target process:

- When the handle has been duplicated.
- What is the duplicated handle value?

Remember, that the caller knows the created handle value, but the target process does not. There must be some other form of inter-process communication that allows the caller process to pass the required information to the target process (since they are part of the same system and need to cooperate by sharing the kernel object in question).

> We'll look at various inter-process communication mechanisms throughout this book.

Private Object Namespaces

We've seen that some types of kernel objects can have string-based names. We've also seen that this is one (convenient) way of sharing such objects between processes. However, there are a few downsides of having named objects:

- Some other, unrelated process, may create an object with the same name, that can cause failure when creating the object later (if the object types differ), or worse, the creation "succeeds" because it's the same object type and the code gets back a handle to an existing object. The result is a mess, where processes use the same object that they don't expect.
- This is a special case of the above bullet, for emphasis. Since the name is visible (in tools, but can also be obtained programmatically), another process can "hijack" the object or otherwise interfere with object usage. From a security standpoint, the object in question is too visible. Unnamed objects are much stealthier, as there is no good way to guess what a particular object is used for.

Is there a way for processes to share named objects (since it's easy) but not be visible to other processes? Starting with Windows Vista, there is a way to create a private object namespace that only the cooperating processes know about. Using tools or APIs will not reveal its full name.

The *PrivateSharing* sample application is an enhanced version of *BasicSharing*, where the memory-mapped file object's name is now under a private object namespace and is not visible to all. Looking at this object with *Process Explorer* shows a partial name only (figure 2-20).

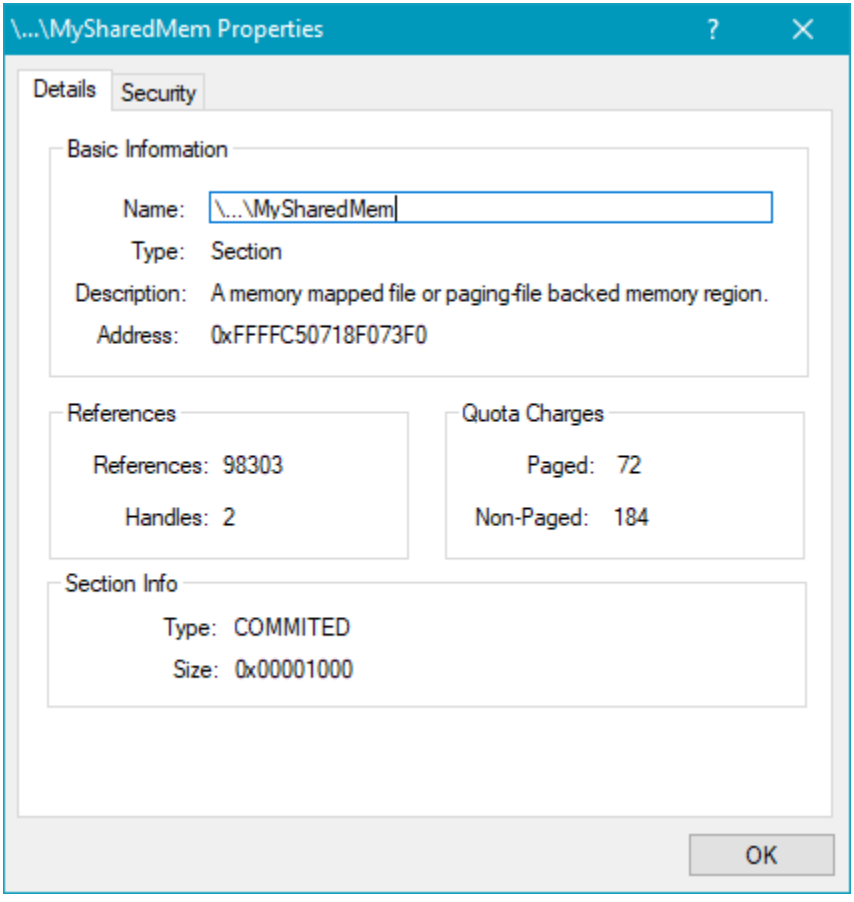


Figure 2-20: Named object with a private namespace

If some random code tries to locate an object named "MySharedMem", it would fail to do so, since this not the object's true name.

Creating a private namespace is a two-step process. First, a helper object called a *Boundary Descriptor* must be created. This descriptor allows adding certain Security IDs (SIDs) that would be able to use private namespaces created based on that boundary descriptor. This can help tighten security on the private namespace(s). To create a boundary descriptor, use `CreateBoundaryDescriptor`:

```
HANDLE CreateBoundaryDescriptor(
    _In_ LPCTSTR Name,
    _In_ ULONG Flags);  // currently unused
```

Once a boundary descriptor exists, two functions can be used to restrict access to any private namespace created through that descriptor: `AddSIDToBoundaryDescriptor` and `AddIntegrityLabelToBoundaryDescriptor` (the latter available starting from Windows 7):

```
BOOL AddSIDToBoundaryDescriptor(
    _Inout_ HANDLE* BoundaryDescriptor,
    _In_ PSID RequiredSid);

BOOL AddIntegrityLabelToBoundaryDescriptor(
    _Inout_ HANDLE * BoundaryDescriptor,
    _In_ PSID IntegrityLabel);
```

Both accept the address of the boundary descriptor's handle and a SID. With `AddSIDToBoundaryDescriptor`, the SID is typically a group's SID, allowing all users in that group access to the private namespaces. `AddIntegrityLabelToBoundaryDescriptor` allows setting a minimum integrity level for processes that wish to open objects in private namespace managed by this boundary descriptor.

> SIDs and integrity levels are discussed in chapter 16.

Once the boundary descriptor is set, the next step is creating the actual private namespace with `CreatePrivateNamespace`:

```
HANDLE CreatePrivateNamespace(
    _In_opt_ LPSECURITY_ATTRIBUTES lpPrivateNamespaceAttributes,
    _In_ LPVOID lpBoundaryDescriptor,   // the boundary descriptor
    _In_ LPCWSTR lpAliasPrefix);        // namespace name
```

Confusingly, the boundary descriptor type is `void*` rather than `HANDLE`. This is a slip in the API, but since `HANDLE` is defined as `void*`, this works fine. This mishap also hints that a boundary descriptor is not a kernel object, even though it returns a `HANDLE`; it has its own close function - `DeleteBoundaryDescriptor`.

An object namespace is also not a true kernel object. If the namespace already exists, the function fails and `OpenPrivateNamespace` must be used instead. It also has its own close function (`ClosePrivateNamespace`):

```
HANDLE OpenPrivateNamespaceW(
    _In_ LPVOID lpBoundaryDescriptor,
    _In_ LPCWSTR lpAliasPrefix);    // namespace name

BOOLEAN ClosePrivateNamespace(
    _In_ HANDLE Handle,
    _In_ ULONG Flags);      // 0 or PRIVATE_NAMESPACE_FLAG_DESTROY
```

> Another slip is the function `ClosePrivateNamespace` returning `BOOLEAN` (`typedefed` as `BYTE`) instead of the standard `BOOL`.

Once the namespace is created or opened, named objects can be created normally with the name in the form `alias\name` where "alias" is the `lpAliasPrefix` parameter from creating or opening the namespace.

Let's look at the concrete code in the *PrivateSharing* application.

The dialog class now has three members:

```
private:
    wil::unique_handle m_hSharedMem;
    HANDLE m_hBD{ nullptr }, m_hNamespace{ nullptr };
```

The code uses the WIL `unique_handle` RAII wrapper for the memory-mapped file's handle, but the boundary descriptor and the namespace are managed as raw handles.

When the dialog box is created, the same memory-mapped file is created as in *BasicSharing*, but this time under a private namespace (error handling omitted for clarity):

```
// create the boundary descriptor
m_hBD = ::CreateBoundaryDescriptor(L"MyDescriptor", 0);

BYTE sid[SECURITY_MAX_SID_SIZE];
auto psid = reinterpret_cast<PSID>(sid);
DWORD sidLen;
::CreateWellKnownSid(WinBuiltinUsersSid, nullptr, psid, &sidLen);

::AddSIDToBoundaryDescriptor(&m_hBD, psid);

// create the private namespace
```

```
m_hNamespace = ::CreatePrivateNamespace(nullptr, m_hBD, L"MyNamespace");
if (!m_hNamespace) {    // maybe created already?
    m_hNamespace = ::OpenPrivateNamespace(m_hBD, L"MyNamespace");
}

m_hSharedMem.reset(::CreateFileMapping(INVALID_HANDLE_VALUE, nullptr,
    PAGE_READWRITE, 0, 1 << 12, L"MyNamespace\\MySharedMem"));
```

In this example, a single SID was added to the boundary descriptor. This SID is for all standard users. It's possible to add something more strict, such as the Administrators group, so that processes running under standard user rights would not be able to tap into this boundary descriptor. The SID is created based on a well-known SID for the users group by calling `CreateWellKnownSid`. Then `AddSIDToBoundaryDescriptor` is called to attach the SID to the boundary descriptor.

> Don't worry about these SIDs and other security terms. They are described in detail in chapter 16.

Once the boundary descriptor is set, `CreatePrivateNamespace` or `OpenPrivateNamespace` is called with the alias "MyNamespace". This is used as the prefix for the memory-mapped file object created with `CreateFileMapping`.

Finally, the `WM_DESTROY` message handler for the dialog deletes the namespace and boundary descriptor:

```
if (m_hNamespace)
    ::ClosePrivateNamespace(m_hNamespace, 0);
if (m_hBD)
    ::DeleteBoundaryDescriptor(m_hBD);
```

Bonus: WIL Wrappers for Private Namespaces

The WIL library has many wrappers for various types of handles and pointers. Unfortunately, it doesn't have a boundary descriptor and private namespace wrappers. Fortunately, it's not too difficult to create ones. Here is one way to do it:

```
namespace wil {
    static void close_private_ns(HANDLE h) {
        ::ClosePrivateNamespace(h, 0);
    };

    using unique_private_ns = unique_any_handle_null_only<decltype(
        &close_private_ns), close_private_ns>;

    using unique_bound_desc = unique_any_handle_null_only<decltype(
        &::DeleteBoundaryDescriptor), ::DeleteBoundaryDescriptor>;
}
```

I will not go over the details of the above declarations, since they do require good acquaintance with C++ 11 `decltype`, `using` and templates.

The *PrivateSharing2* project is the same as *PrivateSharing* but uses WIL wrappers (with the above additions) to manage all handles and even the pointer returned from `MapViewOfFile`. Here is the *Read* function for example:

```
wil::unique_mapview_ptr<void> buffer(::MapViewOfFile(
    m_hSharedMem.get(), FILE_MAP_READ, 0, 0, 0));
if (!buffer) {
    AtlMessageBox(m_hWnd, L"Failed to map memory", IDR_MAINFRAME);
    return 0;
}

SetDlgItemText(IDC_TEXT, (PCWSTR)buffer.get());
```

Other Objects and Handles

Kernel objects are interesting in the context of system programming, and are the focus of this book. There are other common objects used in Windows, namely user objects and GDI objects. The following is a brief description of these objects and handles to such objects.

Task Manager can show the number of such objects for each process by adding the *User Objects* and *GDI Objects* columns, as shown in figure 2-21.

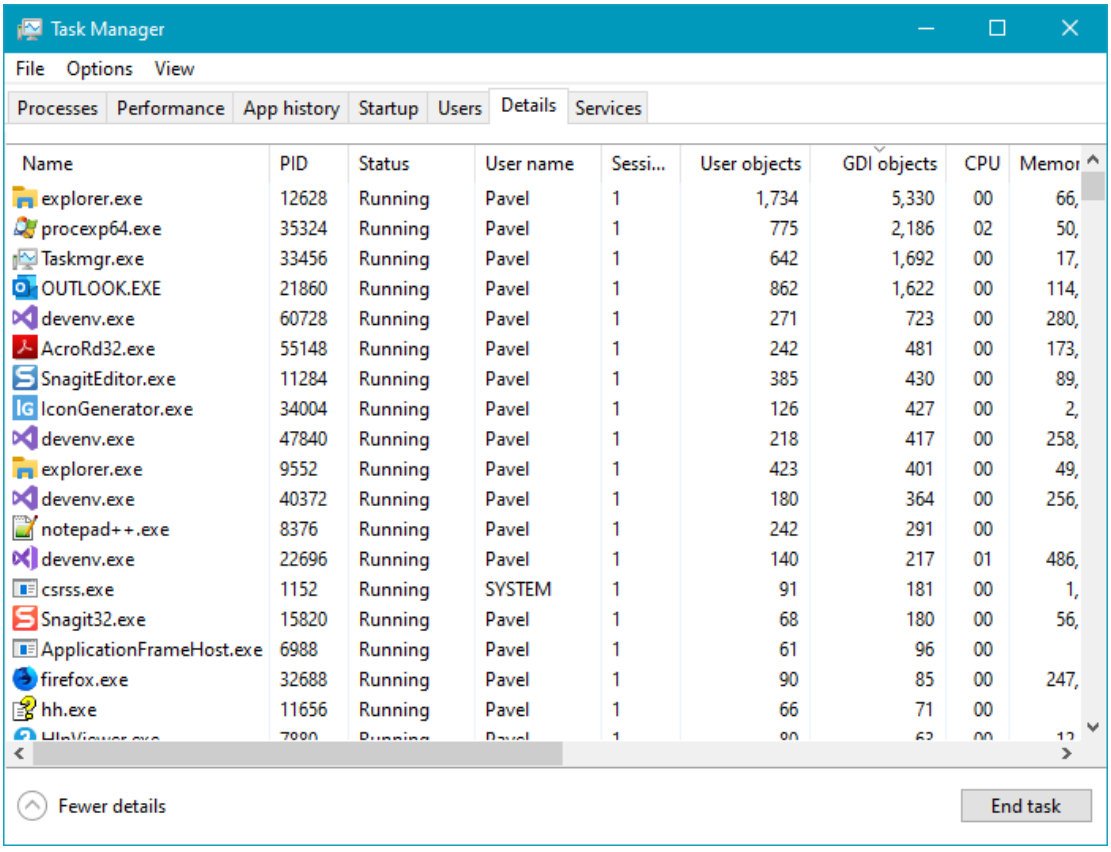


Figure 2-21: User and GDI object count

User Objects

User objects are Windows (`HWND`), Menus (`HMENU`) and hooks (`HHOOK`). Handles to these objects have the following attributes:

- No reference counting. The first caller that destroys a user object - it's gone.
- Handle values are scoped under a Window Station. A Window Station contains a clipboard, desktops and atom table. This means handles to these objects can be passed freely among all applications sharing a desktop, for instance.

> The terms Window Station and desktop will be discussed later in this book. Atom tables will

not, as these are related to the UI subsystem in Windows, which is not the focus of this book.

GDI Objects

The *Graphics Device Interface* (GDI) is the original graphics API in Windows and is still used today, even though there are richer and better APIs (Direct2D for example). Example GDI objects: device context (`HDC`), pen (`HPEN`), brush (`HBRUSH`), bitmap (`HBITMAP`) and others. Here are their attributes:

- No reference counting.
- Handles are valid only in the process in which they are created.
- Cannot be shared between processes.

Summary

In this chapter, we looked at kernel objects and the ways they can be accessed and shared by using handles. We did not look at any specific object type too closely, as these will be discussed in other chapters in more detail. In the next chapter, we'll delve into the most well-known of all kernel objects - the process.

Chapter 3: Processes

Processes are the fundamental management and containment objects in Windows. Everything that executes must be under some process context, there is no such thing as running outside of a process. This chapter examines processes from multiple points of view - from creating, to managing, to destroying and almost everything in between.

In this chapter:

- **Process Basics**
- **Process Creation**
- **Creating Processes**
- **Process Termination**
- **Enumerating Processes**

Process Basics

Although the basic structure and attributes of processes did not change since the first release of Windows NT, new process types have been introduced into the system with special behaviors or structure. The following is a quick overview of all process types currently supported, while later sections in this chapter discuss each process type in greater detail.

- **Protected Processes** - These processes were introduced in Windows Vista. They were created to support *Digital Rights Management* (DRM) protection by preventing intrusive access to processes rendering DRM-protected content. For example, no other process (even running with administrator rights) can read the memory withing a protected process address space, so the DRM-protected data cannot be directly stolen.
- **UWP Processes** - These processes, available starting with Windows 8, host the Windows Runtime, and typically are published to the Microsoft Store. A UWP process executes inside an *AppContainer* - a sandbox of sorts that limits the operations this process can carry out.

- **Protected Processes Light** (PPL) - These processes (available from Windows 8.1) extended the protection mechanism from Vista by adding several levels of protection and even allowing third-party services to run as PPL, protecting them from intrusive access, and from termination, even by admin-level processes.
- **Minimal Processes** - These processes available from Windows 10 version 1607, is a truly new form of process. The address space of a minimal process does not contain the usual images and data structures that a normal process does. For example, there is no executable file mapped into the process address space, and no DLLs. The process address space is truly empty.
- **Pico Processes** - These processes are minimal processes with one addition: a Pico provider, which is a kernel driver that intercepts Linux system calls and translates them to equivalent Windows system calls. These processes are used with the *Windows Subsystem for Linux* (WSL), available from Windows 10 version 1607.

A Process' basic information is easily visible in tools such as *Task Manager* and *Process Explorer*. Figure 3-1 shows *Task Manager Details* tab with some columns added beyond the defaults.

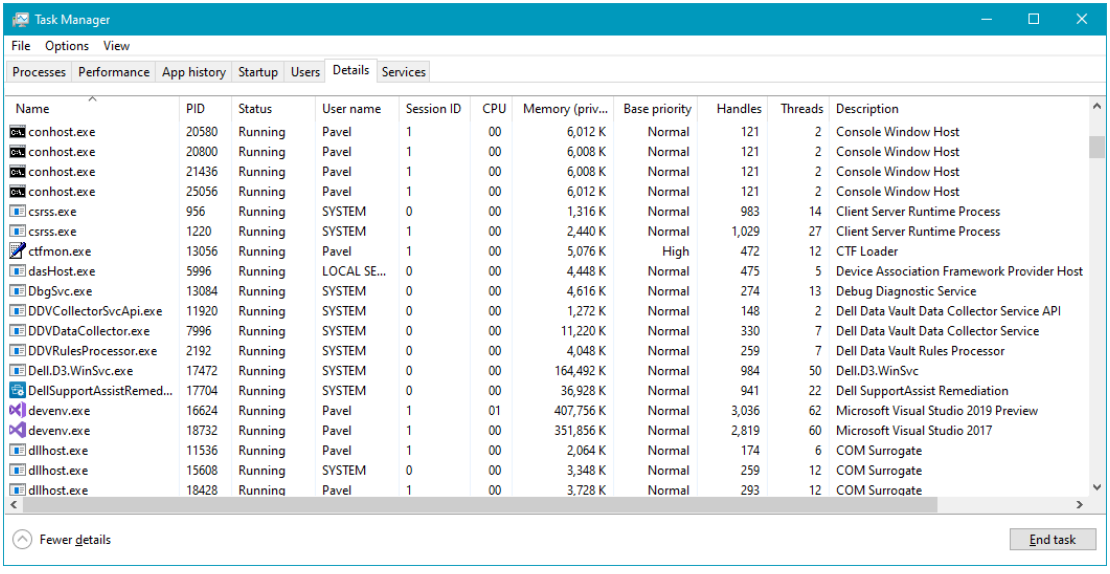


Figure 3-1: Task Manager's Details tab

Let's briefly examine the columns appearing in figure 3-1:

Name

This is normally the executable name upon which the process is based on. Remember that this name is not the unique identifier of the process. Some processes don't seem to have an executable name at all. Examples include *System*, *Secure System*, *Registry*, *Memory Compression*, *System Idle Process* and *System Interrupts*.

- *System Interrupts* is not really a process, it's just used as a way to measure the time spent in the kernel servicing hardware interrupts and *Deferred Procedure Calls*. Both are beyond the scope of this book. You can find more information in the *Windows Internals* and *Windows Kernel Programming* books.
- *System Idle Process* is also not a real process. It always has a Process ID (PID) of zero. It accounts for Windows idle time. This is where the CPUs go when there is nothing to do.
- The *System* process is a true process, technically being a minimal process as well. It always has a PID of 4. It represents everything going on in kernel space - the memory used by the kernel and kernel drivers, the open handles, threads and so on.
- The *Secure System* process is only available on Windows 10 and Server 2016 (and later) systems that boot with *Virtualization Based Security* enabled. It represents everything going on in the secure kernel. Refer to the *Windows Internals* book for more information.
- The *Registry* process is a minimal process available from Windows 10 version 1803 (RS4) that is used as "working area" for managing the registry, rather than using the Paged Pool as was done in previous versions. For the purposes of this book, it's an implementation detail that does not affect the way the registry is accessed programmatically.
- The *Memory Compression* process is a minimal process available on Windows 10 version 1607 (but not on servers) and holds compressed memory in its address space. Memory compression is a feature added in Windows 10 to conserve physical memory (RAM), especially useful for devices with limited resources such as phones and IoT (*Internet of Things*) devices. Confusingly, *Task Manager* does not show this process, but it is shown properly by *Process Explorer*.

The reason *Memory Compression* is not shown in *Task Manager* is somewhat funny. In Windows 10 before version 1607 memory compression was supported, but the compressed memory was stored in the user-mode address space of the *System* process and this made the *System* process look as though it was consuming (possibly) a lot of memory. That was *compressed* memory, so it was really *saving* memory, but appearances are sometimes more important, and so the compressed memory was moved into its own (minimal) process and the process itself was hidden purposefully from the list in *Task Manager*.

In the rest of this chapter, up to the section "Minimal and Pico Processes", deals with "normal" processes that are based on executable files. In any case, minimal and Pico processes can only be created by the kernel.

PID

The unique ID of the process. PIDs are multiple of 4, where the lowest valid PID value is 4 (belonging to the *System* processes). Process IDs are reused once a process terminates, so it's possible to see a new process with a PID that was once used for a (now gone) process. If a unique

identifier is required for a process, then a combination of the PID and the process start time is truly unique on a certain system.

> You may recall from chapter 2, that handles also start with 4 and are multiples of 4, just like PIDs. This is not a coincidence. In fact, PIDs (and thread IDs) are handle values in a special handle table used just for this purpose.

Status

The *Status* column is a curious one. It can have one of three values: *Running*, *Suspended* and *Not Responding*. Let's look at each one. Table 3-1 summarizes the meaning of these states based on process type.

Table 3-1: The Status column

Process type	Running when…	Suspended when…	Not responding when…
GUI process (not UWP)	GUI thread is responsive	All threads in the process are suspended	GUI thread does not check message queue for at least 5 seconds
CUI process (not UWP)	At least one thread is not suspended	All threads in the process are suspended	Never
UWP process	in the foreground	in the background	GUI thread does not check message queue for at least 5 seconds

A process with a GUI must have at least one thread that handles its user interface. This thread has a message queue, created for it as soon as it called any UI or GDI function. This thread must therefore pump messages - that is, listen to its message queue and process messages that arrive. The typical listening functions are `GetMessage` or `PeekMessage`. If none is called for at least 5 seconds, *Task Manager* changes the status to *Not Responding*, the windows owned by that thread become faded and "(Not Responding)" is added to the window's caption. The problematic thread did not examine its message queue for one of three possible reasons:

- It was suspended for whatever reason.
- It's waiting for some I/O operation to complete, and it takes longer than 5 seconds.
- It's doing some CPU intensive work that takes longer than 5 seconds.

We'll look at these issues in chapter 5 ("Thread Basics").

UWP processes are special in the sense that they are suspended unwillingly when they move into the background such as when the application's window is minimized. A simple experiment can verify this case: Open the modern Calculator on Windows 10, and locate it in *Task Manager*. You should see its status as *Running*, meaning it can respond to user input and generally do its thing. Now minimize Calculator, and you'll see the status changing after a few brief seconds to *Suspended*. This kind of behavior exists for UWP processes only.

Non-UWP processes that have no GUI are always shown with a *Running* status, as Windows has no idea what these processes are actually doing (or not). The only exception is if all threads in such a process are suspended, and then its status changes to *Suspended*.

> The Windows API does not have a function to suspend a process, only to suspend a thread. Technically it's possible to loop over all the threads in a certain process and suspend each one (assuming a powerful enough handle can be obtained). The native API (implemented in NtDll.Dll) does have a function for that purpose, `NtSuspendProcess`. This is the function called by *Process Explorer* if you right-click a process and choose *Suspend*. Of course the opposite function exists as well - *NtResumeProcess*.

User Name

The user name indicates under what user the process is running. A token object is attached to the process (called *Primary Token*) that holds the security context for the process based on the user. That security context contains information such as the groups the user belongs to, the privileges it has and more. We'll take a deeper look at tokens in chapter 16. A process can run under special built-in users, such as Local System (shown as *System* in Task Manager), Network Service and Local Service. These user accounts are typically used to run services, which we'll look at in chapter 16.

Session ID

The session number under which session the process executes. Session 0 is used for system process and services and session 1 and higher are used for interactive logins. We'll look at sessions in more detail in chapter 16.

CPU

This column shows the CPU percentage consumption for that process. Note it shows whole numbers only. To get better accuracy, use *Process Explorer*.

Memory

The memory-related columns are somewhat tricky. The default column shown by *Task Manager* is *Memory (active private working set)* (Windows 10 version 1903) or *Memory (private working set)* (earlier versions). The term *Working Set* means RAM (physical memory). *Private working set*

is the RAM used by the process and not shared with other processes. The most common example of shared memory is for DLL code. *Active private working Set* is the same as *Private working set*, but is set to zero for UWP processes that are currently suspended.

> Are the above two counters a good indication of the amount of memory used by a process? Unfortunately, no. These indicate the private RAM used, but what about memory that is currently paged out? There is another column for that - *Commit Size*. This is the best column to use to get a sense of the memory usage for a process. The "unfortunate" part is that *Task Manager* does not show this column by default.

Process Explorer has an equivalent column for *Commit Size*, but it's called *Private Bytes*, which is consistent with a *Performance Counter* by that name.

These memory terms are discussed further in chapter 12.

Base Priority

The *Base Priority* column, officially called *Priority Class*, shows one of six values, that provide the base scheduling priority for threads executing in that process. The possible values with the priority level associated with them are the following:

- *Idle* (called *Low* in *Task Manager*) = 4
- *Below Normal* = 6
- *Normal* = 8
- *Above Normal* = 10
- *High* = 13
- *Real-time* = 24

The most common (and default) priority class is *Normal* (8). We'll discuss priorities and scheduling in chapter 6.

Handles

The *handles* columns shows the number of handles to kernel objects that are open in a particular process. This was discussed at length in chapter 2.

Threads

The *Threads* column shows the number of threads in each process. Normally, this should be at least one, as a process with no threads is useless. However, some processes are shown with no

threads (using a dash). Specifically, *Secure System* is shown with no threads because the secure kernel actually uses the normal kernel for scheduling. The *system Interrupts* pseudo-process is not a process at all, so cannot have any threads. Lastly, the *System Idle Process* does not own threads either. The number of threads shown for this process is the number of logical processors on the system.

There are other columns of interest in *Task Manager*, which will be examined in due course.

Processes in Process Explorer

Process Explorer can be thought of as a "Task Manager on steroids". It has most of the functionality of *Task Manager* and much more. We've already seen its ability to show open handles in processes. In this section we'll examine some of its process-related capabilities.

First, *Process Explorer* can show processes with various columns, just like *Task Manager*. It has more columns than are available in *Task Manager*, however. Immediately apparent are the colors processes are shown in. Each color indicates some interesting aspect of a process. Of course a process can have more than one such "aspect", worthy of a color, and in that case one color "wins" and the "losing" color is not shown. All the available colors can be changed and enabled or disabled by selecting *Options*, *Configure Colors...* from the menu, showing the dialog in figure 3-2.

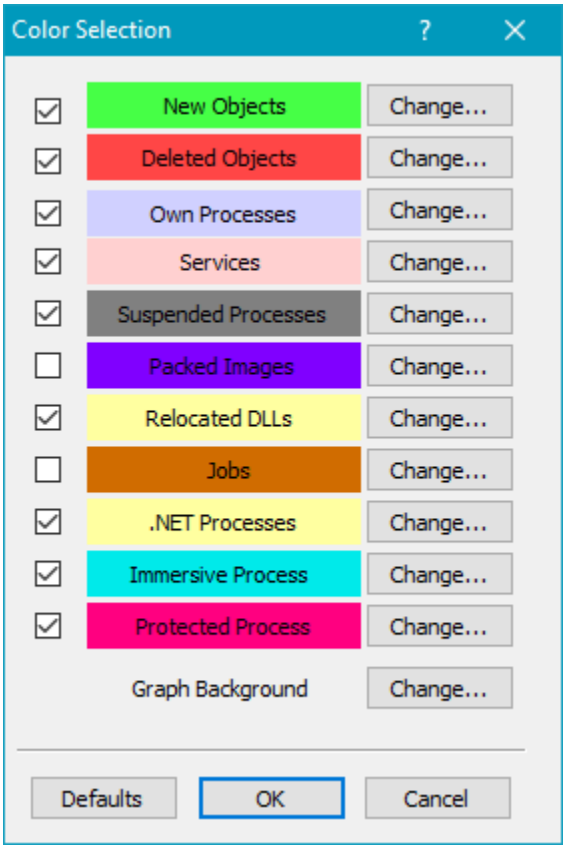


Figure 3-2: Color configurations in Process Explorer

Table 3-2 summarizes their background colors and meaning.

Table 3-2: Colors in Process Explorer

Name (default color)	Meaning
New Objects (green)	New objects created
Deleted Objects (red)	Objects that have been destroyed
Own Processes (blueish)	Processes running under the logged on user account
Services (pink)	Processes hosting Windows Services (see chapter 19)
Suspended Processes (gray)	Suspended processes
Packed Images (purple)	Executables or DLLs that use packing techniques to reduce size. In some cases, malware may be using such techniques

Table 3-2: Colors in Process Explorer

Name (default color)	Meaning
Relocated DLLs (yellowish)	Shown in the modules view (not in the main process view). Discussed in chapter 15
Jobs (brown)	Processes that are part of a job (see chapter 4)
.NET Processes (yellowish)	Processes that run some .NET code. More precisely, processes that host the .NET CLR
Immersive Processes (cyan)	Normally UWP processes (that are not suspended). More precisely, processes that host the Windows Runtime. The function used to determine this is `IsImmersiveProcess`
Protected Processes (fuchsia)	Protected processes and PPL processes (see later in this chapter)
(all other) (white)	Processes that don't have any of the enabled aspects. If all colors are enabled, what's left are mostly system processes

> I personally added the protected processes color and selected the default to be Fuchsia (unrelated to Google's new OS).

The new and destroyed objects colors are shown for a period of one second by default. You can make it longer by opening the menu *Options*, *Difference Highlight Duration...*.

Another interesting feature of *Process Explorer* is the ability to "sort" processes in a tree (more accurately, trees) of processes. If you click on the *Process* column where the image name is, you can sort normally, but a third click turns the *Process* column into trees of processes. Part of these trees is shown in figure 3-3.

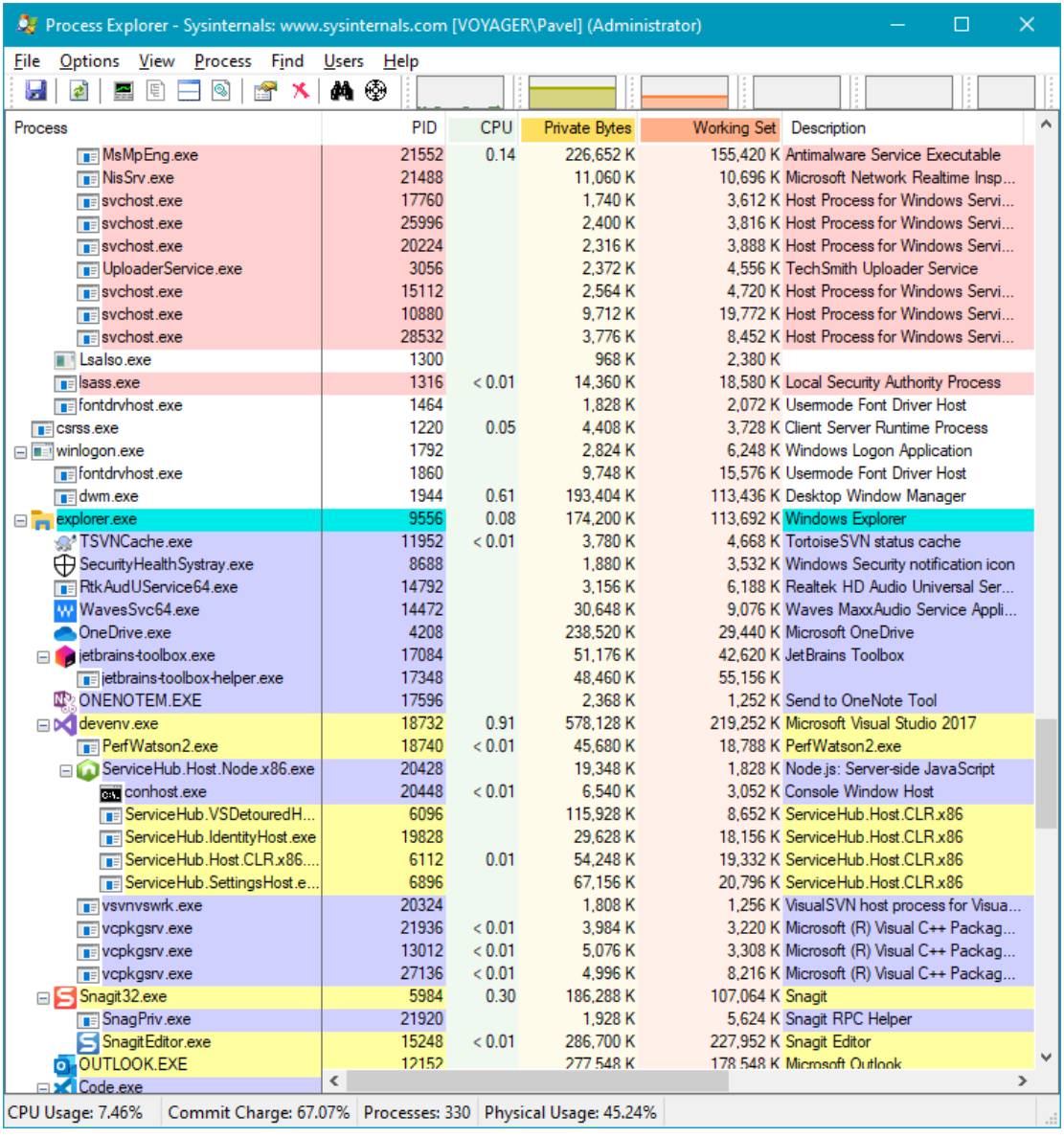


Figure 3-3: Process tree(s) in Process Explorer

Each child node in the tree is a child processes of its parent node. Some processes seem to be left-justified (see *Explorer.exe* in figure 3-3). These processes don't have a parent process, or more accurately - had a parent process that has since exited. Double-clicking such a process and switching to the *Image* tab shows basic information about the process including its parent. Figure 3-4 shows this information for that instance of *Explorer.exe*.

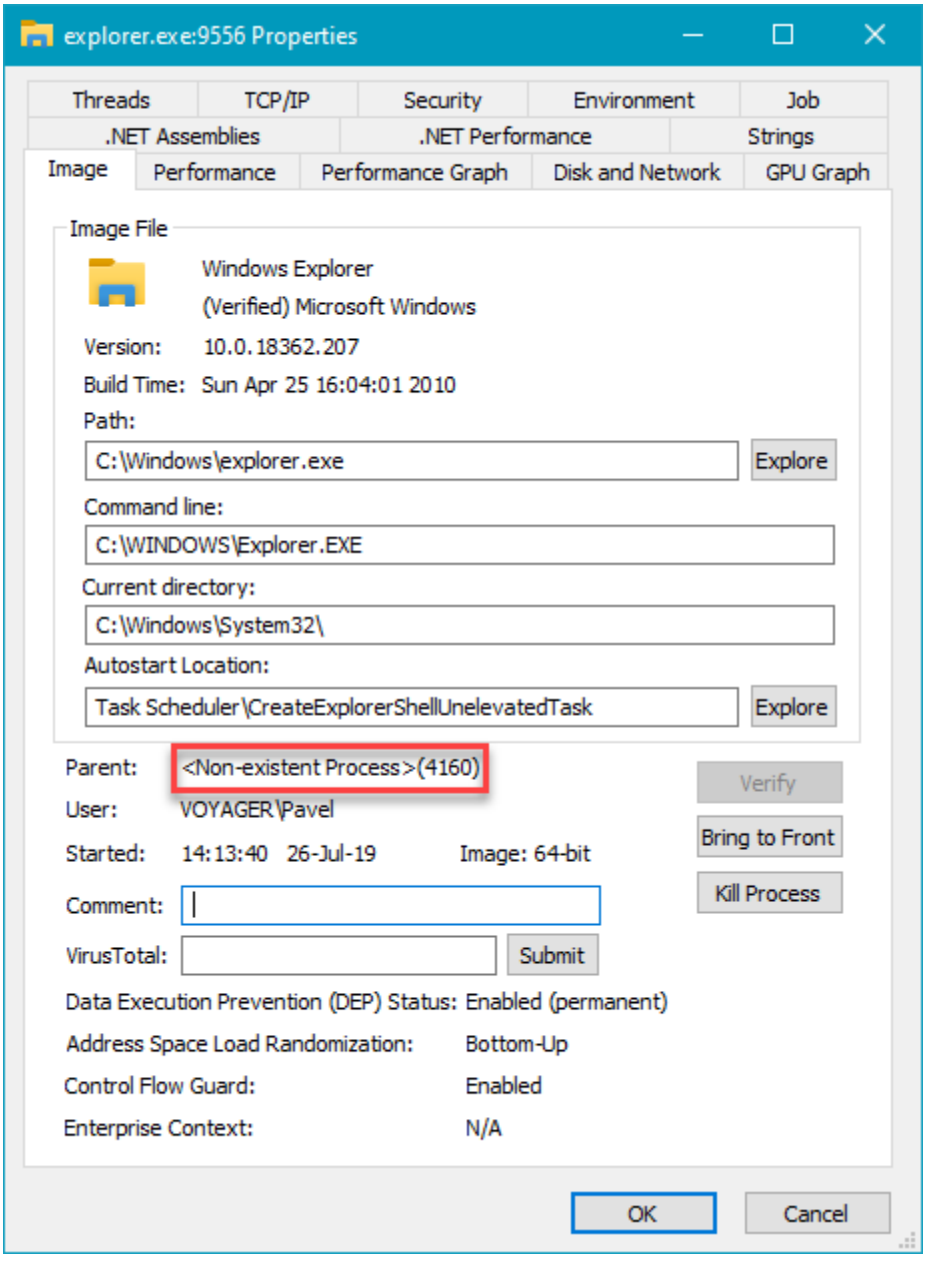


Figure 3-4: *Explorer.exe* Properties

Notice the parent process is unknown but its PID is known (4160 in figure 3-4). This means that the parent PID is stored with the child process, but if the parent no longer exists no other information about it remains.

You may be wondering what would happen if a new process is created with PID 4160 in figure 3-4, since PIDs get reused. Fortunately, *Process Explorer* is not confused, as it checks the start time of the parent process. If it's later than the child then clearly that process cannot be the parent.

Why is *Explorer.exe* parentless? This is actually the normal case, as Explorer is created by an earlier process running *UserInit.exe*, whose job (among other things) is to launch the default shell (configured in the registry by default to be *Explorer.exe*). Once its work is done, the *UserInit* process simply exits.

The important point to remember about this parent-child process relationship is this: if process A creates process B and process A dies, process B is unaffected. In other words, processes in Windows are more like siblings - they don't affect each other after creation.

Process Creation

The major parts involved in process creation is depicted in figure 3-5.

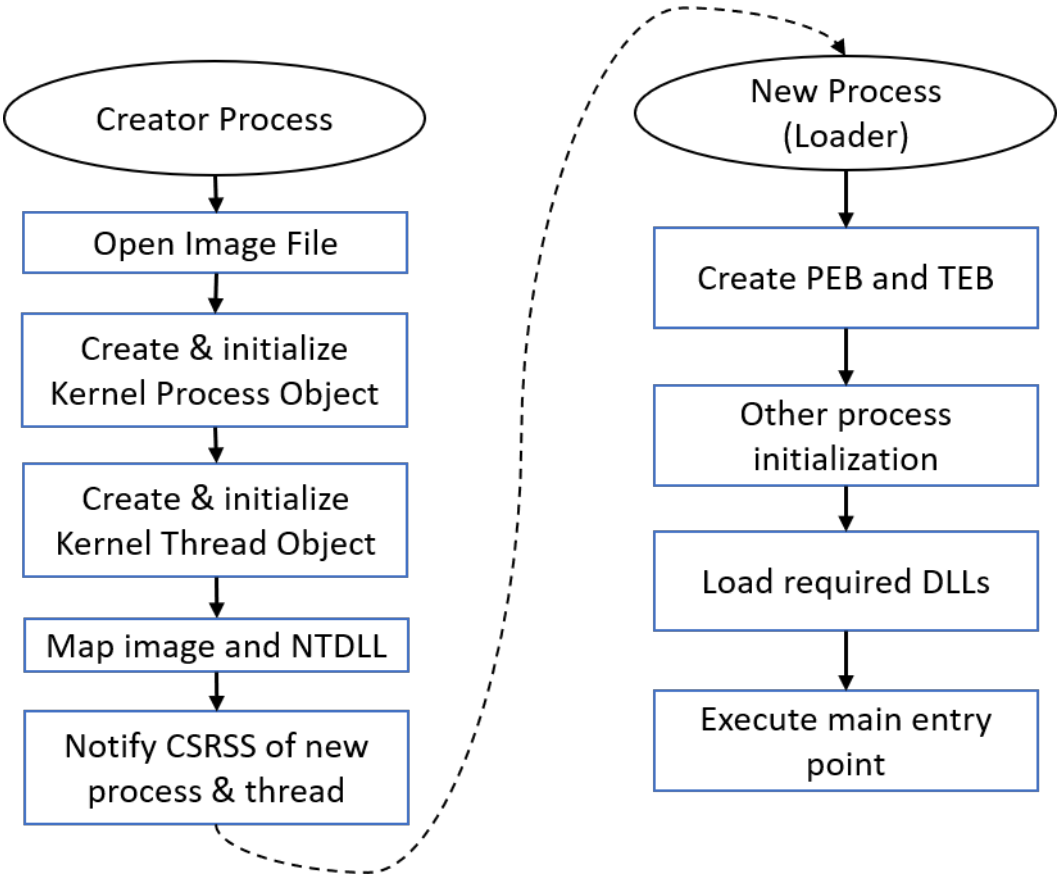


Figure 3-5: Flow of process creation

First, the kernel opens the image (executable) file and verifies that it's in the proper format known as *Portable Executable* (PE). The file extension does not matter, by the way - only the actual content does. Assuming the various headers are valid, the kernel then creates a new process kernel object and a thread kernel object, because a normal process is created with one thread that eventually should execute the main entry point.

At this point, the kernel maps the image to the address space of the new process, as well as *NtDll.Dll*. *NtDll* is mapped to every process (except Minimal and Pico processes), since it has very important duties in the final stage of process creation (outlined below) as well being the trampoline from which system calls are invoked. The final major step which is still carried out by the creator process is notifying the Windows subsystem process (*Csrss.exe*) of the fact that a new process and thread have been created. (*Csrss* can be thought of as a helper to the kernel for managing some aspects of Windows subsystem processes).

At this point, from the kernel's point of view, the process has been created successfully, so the process creation function invoked by the caller (typically `CreateProcess`, discussed in the next

section) returns success. However, the new process is not yet ready to execute its initial code. The second part of process initialization must be carried out in the context of the new process, by the newly created thread.

Some developers believe that the first thing that runs in a new process is the executable's main function. This is far from the truth, however. There is a lot going on before the actual main function starts running. The star of this part is *NtDll*, as there is no other OS level code in the process at this time. *NtDll* has several duties at this point.

First, it creates the user-mode management object for the process, known as the *Process Environment Block* (PEB) and the user mode management object for the first thread called *Thread Environment Block* (TEB). These structures are partially-documented (in `<winternl.h>`), and officially should not be used directly by developers. That said, there are cases where these structures are useful, especially when trying to achieve things that are difficult to do otherwise.

> The current thread's `TEB` is accessible via `NtCurrentTeb()`, while the `PEB` of the current process is available via `NtCurrentTeb()->ProcessEnvironmentBlock`.

Then some other initializations are carried out, including the creation of the default process heap (see chapter 13), creation and initialization of the default process thread pool (chapter 9) and more. For full details, consult the *Windows Internals* book.

The last major part before the entry point can start execution is to load required DLLs. This part of *NtDll* is often referred to as the *Loader*. The loader looks at the import section of the executable, which includes all the libraries the executable depends upon. These typically include the Windows subsystem DLLs such as *kernel32.dll*, *user32.dll*, *gdi32.dll* and *advapi32.dll*.

To get a sense for these import libraries, we can use the *DumpBin.exe* tool available as part of the Windows SDK and Visual Studio installation. Open the Developer Command Prompt to gain easy access to the various tools and type the following to look at the imports for *Notepad.exe*:

```
c:\>dumpbin /imports c:\Windows\System32\notepad.exe
```

The result is a dump of all import libraries and what symbols are imported (used) from those libraries. Here is an abbreviated output (Windows 10 version 1903):

```
Dump of file c:\Windows\System32\notepad.exe

File Type: EXECUTABLE IMAGE

  Section contains the following imports:

    GDI32.dll
             140022788 Import Address Table
             1400289E8 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference
                        35C SelectObject
                        2D0 GetTextFaceW
                        1C2 EnumFontsW
...
    USER32.dll
             140022840 Import Address Table
             140028AA0 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference
                        364 SetThreadDpiAwarenessContext
                        2AD PostMessageW
                         BA DialogBoxParamW
...
    msvcrt.dll
             140022FD8 Import Address Table
             140029238 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference
                          2F ?terminate@@YAXXZ
                        496 memset
...
    api-ms-win-core-libraryloader-l1-2-0.dll
             140022C60 Import Address Table
             140028EC0 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference
                           F GetModuleFileNameA
                          18 LoadLibraryExW
                          13 GetModuleHandleExW
...
```

```
    urlmon.dll
              00000001 Characteristics
      000000014002C0D0 Address of HMODULE
      000000014002F0E0 Import Address Table
      0000000140028368 Import Name Table
      0000000140028638 Bound Import Name Table
      0000000000000000 Unload Import Name Table
                     0 time date stamp
                                  0000000140020F31    3B FindMimeFromData
...
```

For each required DLL, `dumpbin` shows the imported functions from that DLL, i.e. the functions actually used by the executable. Some of the DLL names may look weird, and in fact you won't find them as actual files. The example in the above output is *api-ms-win-core-libraryloader-l1-2-0.dll*. This is known as an *API Set*, which is an indirect mapping from a contract (the API Set) to an actual implementation DLL (sometimes referred to as a *Host*).

API Sets exist starting from Windows 7.

Another way to view these dependencies is with a graphical. Figure 3-6 shows one such tool, *PE Explorer*, downloadable from http://github.com/zodiacon/AllTools, with *Notepad.exe*'s dependencies. For each API Set or DLL, it shows the imported functions.

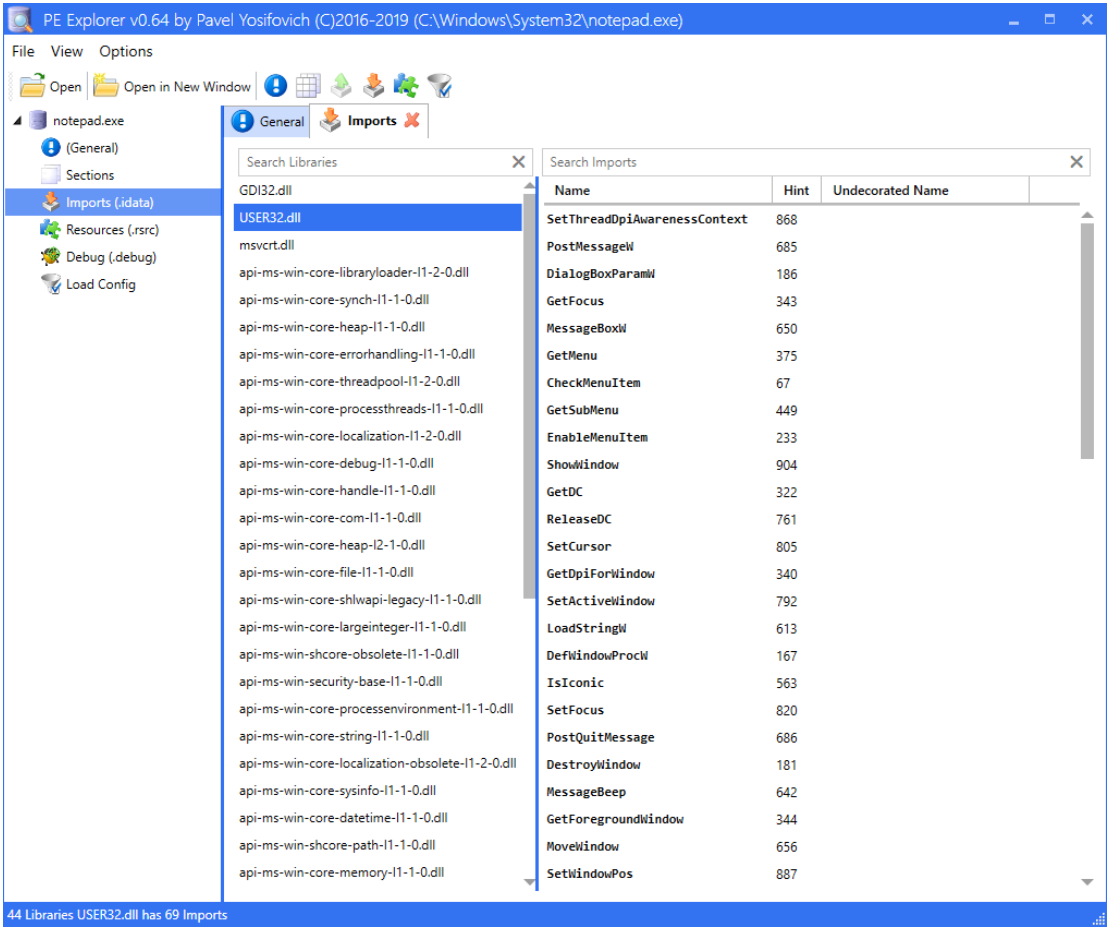


Figure 3-6: PE Explorer with *Notepad.exe*

API Sets allows Microsoft to separate function "declarations" from the actual implementation. This means the implementing DLL can change at a later Windows release, and can even be different on different form factors (IoT devices, HoloLens, Xbox, etc.). The actual mapping between API set and implementation is stored for each process in the `PEB`. You can view these mappings using the *ApiSetMap.exe* tool, downloadable from https://github.com/zodiacon/WindowsInternals/releases. Here is the first few lines of output:

```
C:\>APISetMap.exe
ApiSetMap - list API Set mappings - version 1.0
(c) Alex Ionescu, Pavel Yosifovich, and Contributors
http://www.alex-ionescu.com

      api-ms-onecoreuap-print-render-l1-1-0.dll -> s{printrenderapihost.dll}
        api-ms-win-appmodel-identity-l1-2-0.dll -> s{kernel.appcore.dll}
api-ms-win-appmodel-runtime-internal-l1-1-6.dll -> s{kernel.appcore.dll}
         api-ms-win-appmodel-runtime-l1-1-3.dll -> s{kernel.appcore.dll}
           api-ms-win-appmodel-state-l1-1-2.dll -> s{kernel.appcore.dll}
           api-ms-win-appmodel-state-l1-2-0.dll -> s{kernel.appcore.dll}
          api-ms-win-appmodel-unlock-l1-1-0.dll -> s{kernel.appcore.dll}
          api-ms-win-base-bootconfig-l1-1-0.dll -> s{advapi32.dll}
               api-ms-win-base-util-l1-1-0.dll -> s{advapi32.dll}
  api-ms-win-composition-redirection-l1-1-0.dll -> s{dwmredir.dll}
api-ms-win-composition-windowmanager-l1-1-0.dll -> s{udwm.dll}
      api-ms-win-containers-cmclient-l1-1-1.dll -> s{cmclient.dll}
           api-ms-win-core-apiquery-l1-1-1.dll -> s{ntdll.dll}
           api-ms-win-core-apiquery-l2-1-0.dll -> s{kernelbase.dll}
```

The DLLs or API Set names don't have a full path associated with them. The Loader searches in the following directories in order until the DLL is located:

1. If the DLL name is one of the *KnownDLLs* (specified in the registry), the system directory is searched first (see item 4) (Known DLLs are described in chapter 15 in part 2). This is where the Windows subsystem DLLs reside (*kernel32.dll*, *user32.dll*, *advapi32.dll*, etc.)
2. The directory of the executable
3. The current directory of the process (determined by the parent process). (This is discussed in the next section)
4. The System directory returned by `GetSystemDirectory` (e.g. *c:\windows\system32*)
5. The Windows directory returned by `GetWindowsDirectory` (e.g. *c:\Windows*)
6. The directories listed in the *PATH* environment variable

The DLLs listed in the Known DLLs registry key (*HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs*) are always loaded from the system directory, to prevent DLL hijacking, where an alternate DLL with the same name is placed in the executable's directory.

Once the DLL is found, it's loaded and its `DllMain` function (if exists) is called with the reason `DLL_PROCESS_ATTACH` indicating the DLL has now been loaded into a process. (Full discussion of DLL loading is saved for chapter 15).

This process continues recursively, because one DLL may depend on another DLL and so on. If any of the DLLs is not found, the Loader displays a message box similar to figure 3-7. Then the Loader terminates the process.

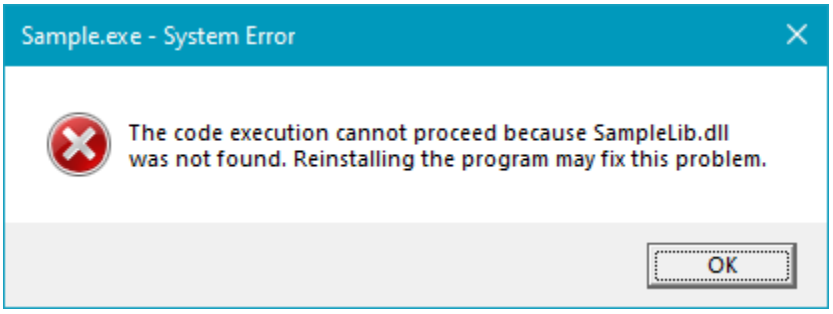


Figure 3-7: Failure to locate a required DLL

If any of the DLL's `DllMain` function returns `FALSE`, this indicates the DLL was not able to initialize successfully. Then the Loader halts further progress and shows the message box in figure 3-8, after which the process shuts down.

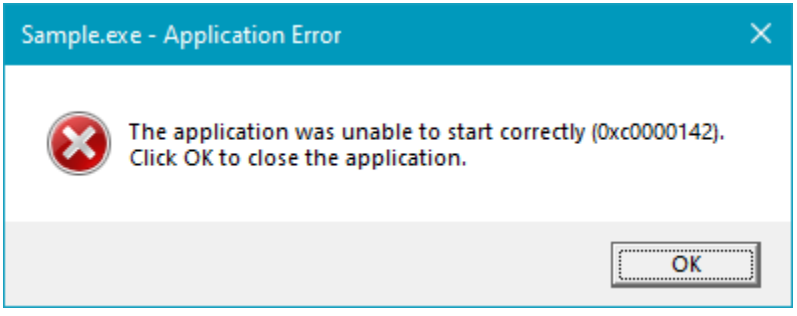


Figure 3-8: Failure to initialize a required DLL

Once all required DLLs have been loaded and initialized successfully, control transfers to the main entry point of the executable. The entry point in question is not the actual main function provided by the developer. Instead, it's a function provided by the C/C++ runtime, set appropriately by the linker. Why is that needed? Calling functions from the C/C++ runtime such as `malloc`, `operator new`, `fopen` and others require some setup. Also, global C++ objects must have their constructors called, even before your main function executes. All this is done by the C/C++ runtime startup function.

There are actually four main functions developers can write, and for each there is a corresponding C/C++ runtime function. Table 3-4 summarizes these names and when they are used.

Table 3-4: main and C/C++ startup functions

Developer's main	C/C++ runtime startup	Scenario
`main`	`mainCRTStartup`	Console application using ASCII characters
`wmain`	`wmainCRTStartup`	Console application using Unicode characters
`WinMain`	`WinMainCRTStartup`	GUI application using ASCII characters
`wWinMain`	`wWinMainCRTStartup`	GUI application using Unicode characters

The correct function is set by the linker's `/SUBSYSTEM` switch, also exposed through Visual Studio, in the Project Properties dialog shown in figure 3-9.

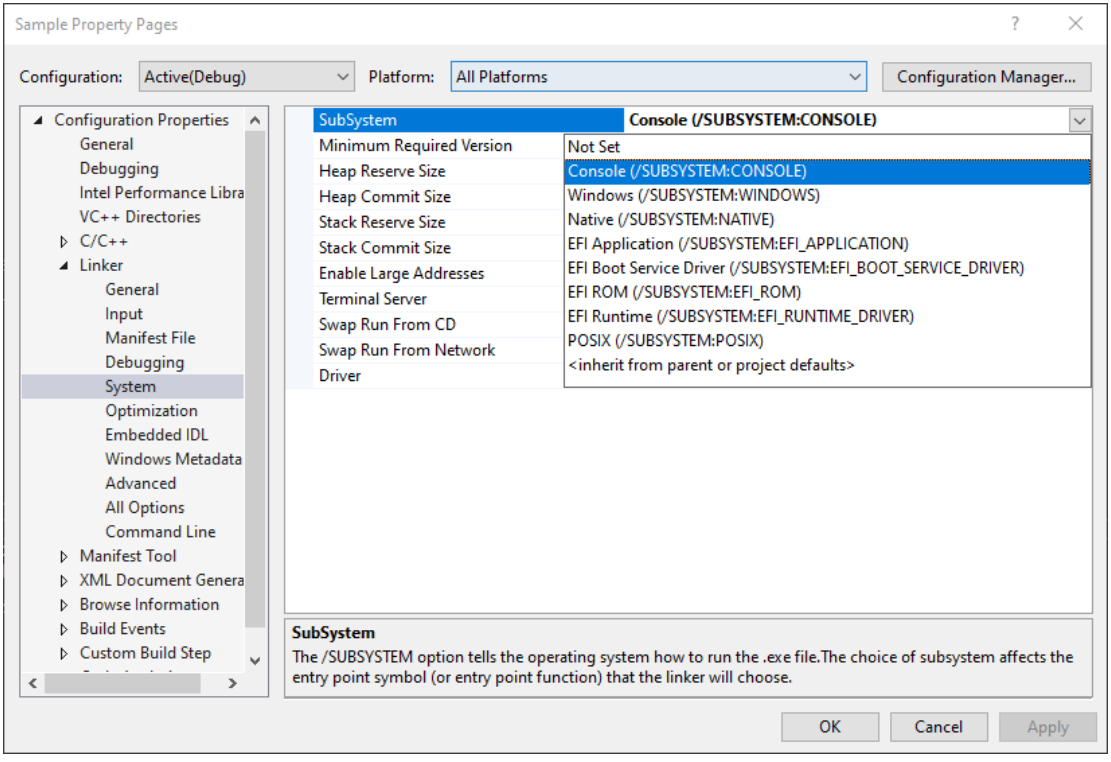


Figure 3-9: System Linker setting in Visual Studio

Is a console-based process that different from a GUI-based process? Not really. Both these types are members of the Windows subsystem. A console application can show GUI and a GUI application can use a console. The difference lies in various defaults such as the main function prototype and whether a console window should be created by default.

> A GUI application can create a console with `AllocConsole`.

The main Functions

Based on the rows in table 3-4, there are four variants of a main function written by developers:

```
int main(int argc, const char* argv[]);     // const is optional
int wmain(int argc, const wchar_t* argv[]); // const is optional
int WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR commandLine, int showCmd);
int wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPWSTR commandLine, int showCmd);
```

> Sometimes you'll see main functions written as `_tmain` or `_tWinMain`. As you probably guess, this allows compiling as Unicode or ASCII based on the compilation constants `_UNICODE` and `UNICODE`, respectively.

With the classic `main`/`wmain` functions, the command line arguments are broken down by the C/C++ runtime prior to calling (w)main. `argc` indicates the number of command-line arguments and is at least one, as the first "argument" is the full path of the executable. `argv` is an array of pointers to the parsed (split based on whitespace) arguments. This means `argv[0]` points to the full executable path.

With the w(WinMain) functions, the parameters are as follows:

- `hInstance` represents the executable module itself within the process address space. Technically, it's the address to which the executable is mapped. It's up to the linker to specify this value. By default, Visual Studio uses the linker option `/DYNAMICBASE` which generates a pseudo-random base address each time the project is built. In any case, the number itself is not important, but it is needed in various functions, such as for loading resources (`LoadIcon`, `LoadString` and others).

> The `HINSTANCE` type is just a `void` pointer. Incidentally, the `HMODULE` type which is sometimes used interchangeably with `HINSTANCE`, is indeed the same thing. The reason two types exist instead of one has to do with 16 bit Windows, where they meant different things.

- `hPrevInstance` should represent the `HINSTANCE` of a previous instance of the same executable. This value is always `NULL`, however, and is not used. In the 16-bit Windows days, it could be non-`NULL` if it was not the first instance of this executable. This means there is no direct way to know whether another process running the same executable already exists. We saw one way to deal with that if needed in the *Singleton* demo application from chapter 2. The `WinMain` signature was preserved from 16-bit Windows for easier porting to 32-bit Windows (at the time). The end result is that this parameter is often written without a variable name because it's completely useless.

> An alternative technique to silence compiler warnings (especially with pure C) is to use the `UNREFERENCED_PARAMETER` macro with the variable name like so: `UNREFERENCED_PARAMETER(hPrevInstance);`. Ironically, this macro actually *references* its argument by simply writing it with a terminating semicolon; that's enough to make the compiler happy.

- `commandLine` is the command line string excluding the executable path - it's the rest of the command line (if any). It's not "parsed" into separate tokens - it's just a single string. If parsing is beneficial, the following function can be used:

```
#include <ShellApi.h>

LPWSTR* CommandLineToArgvW(_In_ LPCWSTR lpCmdLine, _Out_ int* pNumArgs);
```

The function takes the command line and splits it into tokens, returning a pointer to an array of string pointers. The count of strings is returned via `*pNumArgs`. The function allocates a block of memory to hold the parsed command-line arguments, and it must be eventually freed by calling `LocalFree`. The following code snippet shows how to parse the command line properly in a `wWinMain` function:

```
int  wWinMain(HINSTANCE hInstance, HINSTANCE, LPWSTR lpCmdLine, int nCmdShow) {
    int count;
    PWSTR* args = CommandLineToArgvW(lpCmdLine, &count);

    WCHAR text[1024] = { 0 };
    for (int i = 0; i < count; i++) {
        ::wcscat_s(text, 1024, args[i]);
        ::wcscat_s(text, 1024, L"\n");
    }
    ::LocalFree(args);

    ::MessageBox(nullptr, text, L"Command Line Arguments", MB_OK);
```

If the string passed to `CommandLineToArgvW` is the empty string, then its return value is a single string which is the full executable path. On the other hand, if the passed-in string contains non-empty arguments, it returns an array of string pointers containing only the parsed arguments without having the full executable path as the first parsed string.

A process can get its command line at any time by calling `GetCommandLine`, and it's a proper argument to `CommandLineToArgvW`. This could be useful if parsing is required outside of `wWinMain`.

- `showCmd` is the last argument which suggests how to show the main window of the application. It's determined by the process creator, where the default is `SW_SHOWDEFAULT` (10). The application of course is free to disregard this value and show its main window in any way it pleases.

Process Environment Variables

Environment variables is a set of name/value pairs that can be set on a system or user wide basis using the dialog shown in figure 3-10 (accessible from the *System Properties* dialog or simply by searching). The names and values are stored in the Registry (like most system data in Windows).

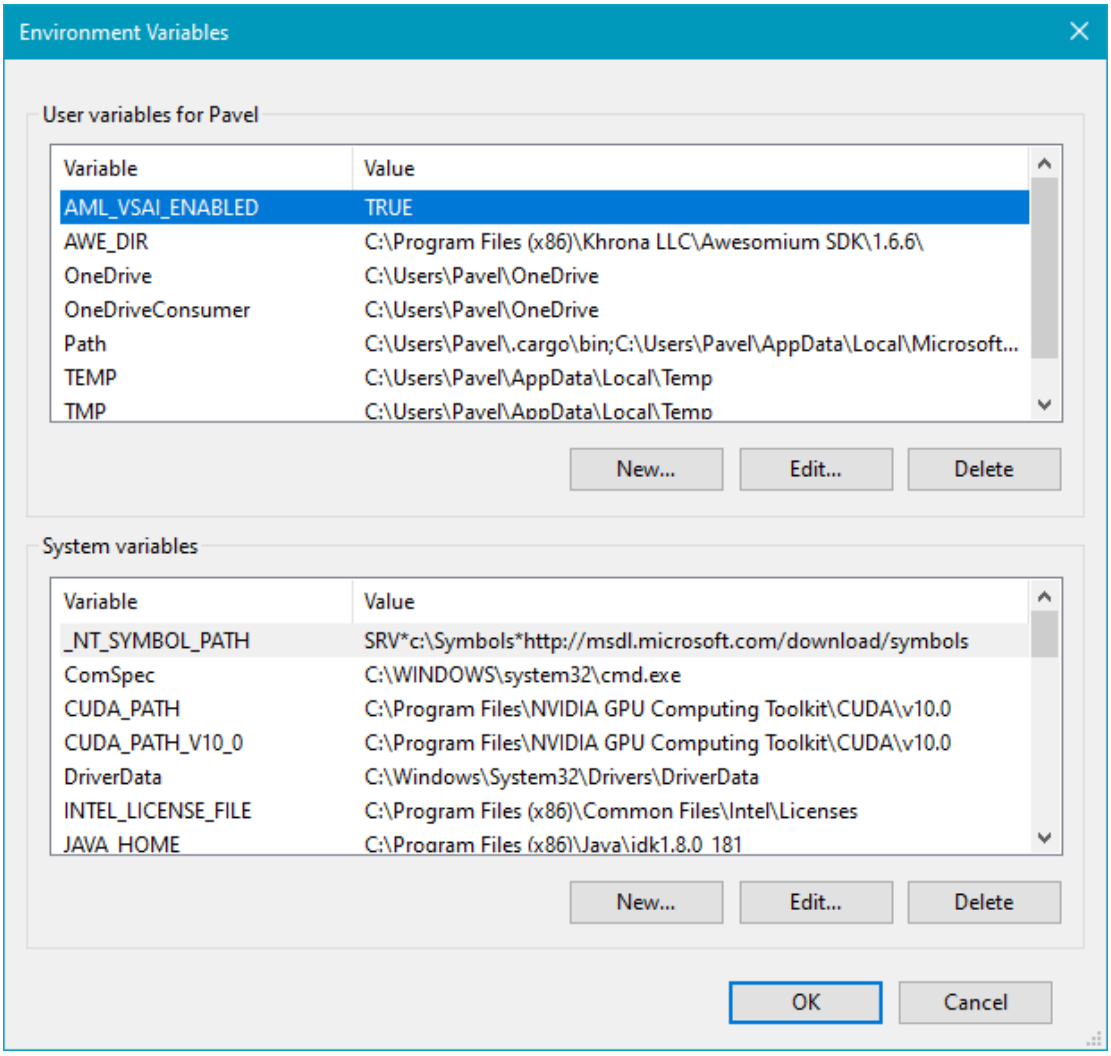


Figure 3-10: Environment variables editor

User environment variables are stored in *HKEY_CURRENT_USER\Environment*. System environment variables (applicable to all users) are stored at *HKEY_LOCAL_MACHINE\System\Current Control Set\Control\Session Manager\Environment*

A process receives environment variables from its parent process, which are a combination of system variables (applicable to all users) and user-specific variables. In most cases the environment variables a process receives are a copy of its parent's (see next section).

A console application can get the process environment variables with a third argument to `main` or `wmain`:

```
int main(int argc, char* argv[], const char* env[]);        // const is optional
int wmain(int argc, wchar_t* argv[], const wchar_t* env[]); // const is optional
```

env is an array of string pointers, where the last pointer is NULL, signaling the end of the array. Each string is built in the following format:

```
name=value
```

The equals character separates the name from the value. The following example main function prints out the names and values of each environment variable:

```
int main(int argc, const char* argv[], char* env[]) {
    for (int i = 0; ; i++) {
        if (env[i] == nullptr)
            break;

        auto equals = strchr(env[i], '=');

        // change the equals to NULL
        *equals = '\0';
        printf("%s: %s\n", env[i], equals + 1);

        // for consistency, revert the equals sign
        *equals = '=';
    }
    return 0;
}
```

GUI applications can call GetEnvironmentStrings to get a pointer to an environment variables memory block, formatted like so:

```
name1=value1\0
name2=value2\0
...
\0
```

The following code snippet uses GetEnvironmentStrings to show all environment variables in one giant message box:

```
PWSTR env = ::GetEnvironmentStrings();
WCHAR text[8192] = { 0 };

auto p = env;
while (*p) {
    auto equals = wcschr(p, L'=');
    if (equals != p) {     // eliminate empty names/values
        wcsncat_s(text, p, equals - p);
        wcscat_s(text, L": ");
        wcscat_s(text, equals + 1);
        wcscat_s(text, L"\n");
    }
    p += wcslen(p) + 1;
}

::FreeEnvironmentStrings(env);
```

> The environment block can be replaced in one swoop with `SetEnvironmentStrings` using the same format returned by `GetEnvironmentStrings`.

The environment block must be freed with `FreeEnvironmentStrings`. Normally, applications don't need to enumerate environment variables, but rather change or read a specific value. The following functions are used for this purpose:

```
BOOL SetEnvironmentVariable(
    _In_ LPCTSTR lpName,
    _In_opt_ LPCTSTR lpValue);

DWORD GetEnvironmentVariable(
    _In_opt_ LPCTSTR lpName,
    _Out_ LPTSTR lpBuffer,
    _In_ DWORD nSize);
```

`GetEnvironmentVariable` returns the number of characters copied to the buffer if it's large enough, or the length of the environment variable otherwise. It returns zero on failure (if the named variable does not exist). The usual practice is to call the function twice: first, with no buffer to get the length and then a second time after allocating a properly-sized buffer to receive the result. The following function can be used to get a variable's value by returning a C++ `std::wstring` as a result:

```
std::wstring ReadEnvironmentVariable(PCWSTR name) {
    DWORD count = ::GetEnvironmentVariable(name, nullptr, 0);
    if (count > 0) {
        std::wstring value;
        value.resize(count);
        ::GetEnvironmentVariable(name, const_cast<PWSTR>(value.data()), count);
        return value;
    }
    return L"";
}
```

The `const_cast` operator above removes the "constness" of `value.data()` as it returns `const wchar_t*`. A brutal C-style cast would work just as well: `(PWSTR)value.data()`.

Environment variables are used in many situations to specify information that is based on their current values. For example, a file path may be specified as "%windir%\explorer.exe". The name between the percent characters is an environment variable that should be expended into its real value. Normal API functions don't have any special understanding of these intentions. Instead, the application must call `ExpandEnvironmentStrings` to convert any environment variable enclosed between percent signs to its value:

```
DWORD ExpandEnvironmentStrings(
  _In_      LPCTSTR lpSrc,
  _Out_opt_ LPTSTR  lpDst,
  _In_      DWORD   nSize);
```

Just like `GetEnviromentVariable`, `ExpandEnvironmentStrings` returns the number of characters copied to the target buffer or the number of characters that is needed if the buffer is too small (plus the `NULL` terminator). Here is an example usage:

```
WCHAR path[MAX_PATH];
::ExpandEnvironmentStrings(L"%windir%\\explorer.exe", path, MAX_PATH);
printf("%ws\n", path);  // c:\windows\explorer.exe
```

Creating Processes

Processes are created under the same user account with `CreateProcess`. Extended functions exist, such as `CreateProcessAsUser`, which will be discussed in chapter 16. `CreateProcess`

requires an actual executable file. It cannot create a process based on a path to a document file. For example, passing something like *c:\MyData\data.txt*, assuming *data.txt* is some text file - will fail process creation. `CreateProcess` does not search for an associated executable to launch for *TXT* files. When a file is double-clicked in Explorer, for instance, a higher-level function from the Shell API is invoked - `ShellExecuteEx`. This function accepts any file, and if does not end with "EXE", will search the registry based on the file extension to locate the associated program to execute. Then (if located), it will eventually call `CreateProcess`.

> Where does Explorer look for these file associations? We'll look at that in chapter 17 ("Registry")

`CreateProcess` accepts 9 parameters as follows:

```
BOOL CreateProcess(
    _In_opt_ PCTSTR pApplicationName,
    _Inout_opt_ PTSTR pCommandLine,
    _In_opt_ PSECURITY_ATTRIBUTES pProcessAttributes,
    _In_opt_ PSECURITY_ATTRIBUTES pThreadAttributes,
    _In_ BOOL bInheritHandles,
    _In_ DWORD dwCreationFlags,
    _In_opt_ PVOID pEnvironment,
    _In_opt_ PCTSTR pCurrentDirectory,
    _In_ PSTARTUPINFO pStartupInfo,
    _Out_ PPROCESS_INFORMATION lpProcessInformation);
```

The function returns `TRUE` on success, which means that from the kernel's perspective the process and the initial thread have been created successfully. It's still possible for the initialization done in the context of the new process (described in the previous section) to fail.

If successful, the real returned information is available via the last argument of type `PROCESS_INFORMATION`:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION, *PPROCESS_INFORMATION;
```

There are four pieces of information provided: the unique process and thread IDs, and two open handles (with all possible permissions unless the new process is protected) to the newly created

process and thread. Using the handles, the creating (parent) process can do anything it wants with the new process and thread (again, unless the process is protected, see later in this chapter). As usual, it's a good idea to close these handles once they are no longer needed.

Now let's turn our attention to the rest of the parameters that are inputs to the function.

`pApplicationName` and **`pCommandLine`**

These parameters should provide the executable path to run as a new process and any command-line arguments that are needed. These parameters are not interchangeable, however.

In most cases, you'll use the second argument for both the executable name and any command-line arguments needed to pass to the executable and set the first argument to `NULL`. Here are some of the benefits of the second argument compared to the first:

- If the file name has no extension, an EXE extension is added implicitly before searching for a match.
- If just a file name is supplied (not a full path) as the executable, the system searches in the directories listed in the previous section where the Loader looks for required DLLs, repeated here for convenience:

1. The directory of the caller's executable
2. The current directory of the process (discussed later in this section)
3. The System directory returned by `GetSystemDirectory`
4. The Windows directory returned by `GetWindowsDirectory`
5. The directories listed in the *PATH* environment variable

If `pApplicationName` is not `NULL`, then it must be set to a full path to the executable. In that case, `pCommandLine` is still treated as command-line arguments.

One dent in the `pCommandLine` argument is that it's typed as `PTSTR`, meaning it's a non-const pointer to a string. This means `CreateProcess` actually writes (not just reads) to this buffer, which will cause an access violation if called with a constant string like so:

```
CreateProcess(nullptr, L"Notepad", ...);
```

Compile-time static buffers are placed by default in a read-only section of the executable and mapped with read-only protection, causing any writes to raise an exception. The simplest solution is to place the string in read/write memory by building it dynamically or place it on the stack (which is always read/write):

```
WCHAR name[] = L"Notepad";
CreateProcess(nullptr, name, ...);
```

> The final content of the buffer is the same as initially provided. You might be wondering why `CreateProcess` writes to the buffer. Unfortunately, there is no good reason that I know of and Microsoft should fix that. But they haven't for many years now, so I wouldn't hold my breath.

This issue does not occur with `CreateProcessA` (the ASCII version of `CreateProcess`). The reason may be obvious: `CreateProcessA` must convert its arguments to Unicode, and for that it allocates a buffer dynamically (which is read/write), converts the string and then calls `CreateProcessW` with that allocated buffer. This does *not* mean you should use `CreateProcessA`!

`pProcessAttributes` and `pThreadAttributes`

These two parameters are `SECURITY_ATTRIBUTES` pointers (for the newly created process and thread), discussed in chapter 2. In most cases, `NULL` should be passed in, unless the returned handles should be inheritable and in that case an instance with `bInheritHandle = TRUE` may be passed in.

`bInheritHandles`

This parameter is a global switch that allows or disallows handle inheritance (described in the next sub-section). If `FALSE`, no handles from the parent process are inherited by the (newly created) child process. If `TRUE`, all handles that are marked inheritable will be inherited by the child process.

`dwCreationFlags`

This parameter can be a combination of various flags, the more useful ones described in table 3-5. Zero is a reasonable default in many cases.

Table 3-5: Some flags for `CreateProcess`

Flag	Description
`CREATE_BREAKAWAY_FROM_JOB`	If the parent process is part of a job, the child process is not, unless the job does not allow breaking out of it in which case the child process is still created under the same job (see next chapter for more on jobs)
`CREATE_SUSPENDED`	The process and thread are created, but the thread is suspended. The parent process eventually should call `ResumeThread` on the returned thread handle to start execution
`DEBUG_PROCESS`	The parent process becomes a debugger, and the created process is the debuggee. The debugger will start getting debugging events related to the child process. Any processes created from the child also become debuggees under the control of the parent process
`DEBUG_ONLY_THIS_PROCESS`	Similar to `DEBUG_PROCESS`, but only the child process becomes a debuggee, rather than all child processes created by the child process
`CREATE_NEW_CONSOLE`	The new process gets its own console (if it's a CUI application) rather than inheriting its parent console
`CREATE_NO_WINDOW`	If the child is a CUI application, it's created without a console
`DETACHED_PROCESS`	Sort of the opposite of `CREATE_NEW_CONSOLE`. The child process does not get any console. If it needs one, it can call `AllocConsole` to create one
`CREATE_PROTECTED_PROCESS`	The new process must be run protected (see later in this chapter)
`CREATED_PROTECTED_PROCESS`	Create the new process as protected. This only works for executables that are signed by Microsoft specifically for this
`CREATE_UNICODE_ENVIRONEMT`	Creates the environment block for the new process as Unicode rather than the default (which is ironically ASCII)
`INHERIT_PARENT_AFFINITY`	(Windows 7+) The child process inherits its parent group affinity (see chapter 6 for more on affinity)
`EXTENDED_STARTUPINFO_PRESENT`	The process is created with an extended `STARTUPINFOEX` structure that contains process attributes (see the section "Process (and Thread) Attributes" later in this chapter
`CREATE_DEFAULT_ERROR_MODE`	Creates the process with the system default error mode rather than inheriting it from the parent. See the section on Error Mode in chapter 20

In addition to the flags listed in table 3-5, the creator can set the process priority class, based on

table 3-6.

Table 3-6: Priority class flags in `CreateProcess`

Priority class flag	Base priority value
IDLE_PRIORITY_CLASS	4
BELOW_NORMAL_PRIORITY_CLASS	6
NORMAL_PRIORITY_CLASS	8
ABOVE_NORMAL_PRIORITY_CLASS	10
HIGH_PRIORITY_CLASS	13
REALTIME_NORMAL_PRIORITY_CLASS	24

If no priority class flag is specified, the default is Normal unless the creator's priority class is Below Normal or Idle, in which case the new process inherits its parent's priority class. If the Real-time priority class is specified, the child process must execute with admin privileges; otherwise, it gets a High priority class instead.

The priority class has little meaning for the process itself. Rather, it sets the default priority for threads in the new process. We'll look at the effects of priorities in chapter 6.

`pEnvironment`

This is an optional pointer to an environment variables block to be used by the child process. Its format is the same as returned by `GetEnvironmentStrings` discussed earlier in this chapter. In most cases, `NULL` is passed in, which causes the parent's environment block to be copied to the new process' environment block.

`pCurrentDirectory`

This sets the current directory for the new process. The current directory is used as part of the search for files in case a file name only is used rather than a full path. For example, a call to the `CreateFile` function with a file named "mydata.txt", will search for the file in the process' current directory. The `pCurrentDirectory` parameter allows the parent process to set the current directory for the created process which can affect the locations in which DLL search is performed for required DLLs. In most cases, `NULL` is passed in, which sets the current directory of the new process to the current directory of the parent.

Normally, a process can change its current directory with `SetCurrentDirectory`. Note this is a process-wide rather than a thread setting:

```
BOOL SetCurrentDirectory(
  _In_ PCTSTR pPathName);
```

The current directory consists of a drive letter and path or a share name in the *Universal Naming Convention* (UNC), such as *\\MyServer\MyShare*.

Naturally, the current directory can be queried back with `GetCurrentDirectory`:

```
DWORD GetCurrentDirectory(
  _In_  DWORD  nBufferLength,
  _Out_ LPTSTR lpBuffer);
```

The return value is zero on failure, or the number of characters copied to the buffer (including the `NULL` terminator). If the buffer is too small, the returned value is the required character length (including the `NULL` terminator).

pStartupInfo

This parameter points to one of two structures, `STARTUPINFO` or `STARTUPINFOEX`, defined like so:

```
typedef struct _STARTUPINFO {
  DWORD cb;
  PTSTR lpReserved;
  PTSTR lpDesktop;
  PTSTR lpTitle;
  DWORD dwX;
  DWORD dwY;
  DWORD dwXSize;
  DWORD dwYSize;
  DWORD dwXCountChars;
  DWORD dwYCountChars;
  DWORD dwFillAttribute;
  DWORD dwFlags;
  WORD  wShowWindow;
  WORD  cbReserved2;
  PBYTE lpReserved2;
  HANDLE hStdInput;
  HANDLE hStdOutput;
  HANDLE hStdError;
} STARTUPINFO, *PSTARTUPINFO;
```

```
typedef struct _STARTUPINFOEX {
    STARTUPINFO StartupInfo;
    PPROC_THREAD_ATTRIBUTE_LIST pAttributeList;
} STARTUPINFOEXW, *LPSTARTUPINFOEXW;
```

The minimum usage for this parameter is to create a `STARTUPINFO` structure, initialize its size (`cb` member) and zero out the rest. Zeroing the structure is important, otherwise it holds junk values which will likely cause `CreateProcesss` to fail. Here is the minimal code:

```
STARTUPINFO si = { sizeof(si) };
CreateProcess(..., &si, ...);
```

`STARTUPINFOEX` is discussed in the section "Process (and Thread) Attributes" later in this chapter.

The `STARTUPINFO` and `STARTUPINFOEX` structures provide more customization options for process creation. Some of their members are only used if certain values are set in the `dwFlags` member (in addition to other flags). Table 3-7 details `dwFlags` possible values and their meaning.

Table 3-7: The `dwFlags` member of `STARTUPINFO` flags

Flag	Meaning
`STARTF_USESHOWWINDOW`	The `wShowWindow` member is valid
`STARTF_USESIZE`	The `dwXSize` and `dwYSize` members are valid
`STARTF_USEPOSITION`	The `dwX` and `dwY` members are valid
`STARTF_USECOUNTCHARS`	The `dwXCountChars` and `dwYCountChars` members are valid
`STARTF_USEFILLATTRIBUTE`	The `dwFillAttribute` member is valid
`STARTF_RUNFULLSCREEN`	For console apps, run full screen (x86 only)
`STARTF_FORCEONFEEDBACK`	Instructs Windows to show the "Working in Background" cursor, whose shape can be found in the Mouse Properties dialog shown in figure 3-11. If during the next 2 seconds, the process makes GUI calls, it gives the process an additional 5 seconds with this cursor showing. If at any point, the process calls `GetMessage`, indicating it's ready to process messages, the cursor is immediately reverted to normal.
`STARTF_FORCEOFFFEEDBACK`	Does not show the "Working in Background" cursor
`STARTF_USESTDHANDLES`	The `hStdInput`, `hStdOutput` and `hStdError` members are valid

Table 3-7: The `dwFlags` member of `STARTUPINFO` flags

Flag	Meaning
`STARTF_USEHOTKEY`	The `hStdInput` member is valid and is a value sent as `wParam` for a `WM_HOTKEY` message. Refer to the documentation for more information
`STARTF_TITLEISLINKNAME`	The `lpTitle` member is a path to a shortcut file (`.lnk`) used to start the process. The Shell (Explorer) sets this appropriately
`STARTF_TITLEISAPPID`	The `lpTitle` member is an *AppUserModelId*. See discussion after this table
`STARTF_PREVENTPINNING`	Prevents any windows created by the process from being pinned to the task bar. Only works if `STARTF_TITLEISAPPID` is also specified
`STARTF_UNTRUSTEDSOURCE`	Indicates the command line passed to the process is from an untrusted source. This is a hint to the process to check its command line carefully

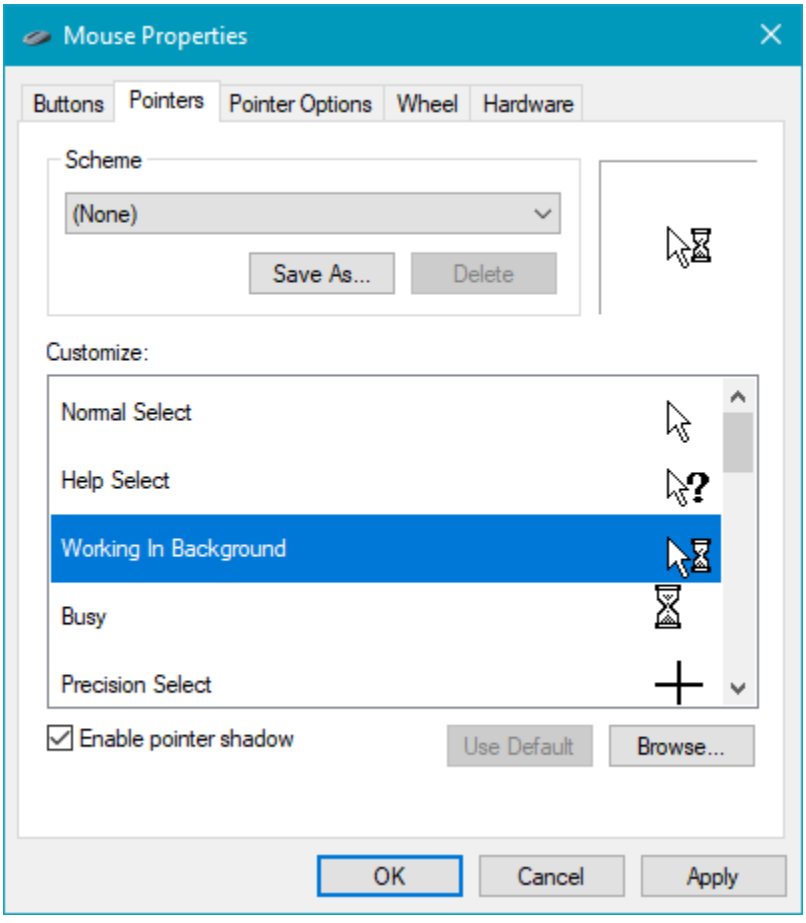


Figure 3-11: "Working in Background" mouse cursor

Let's now examine the other members of `STARTUPINFO`.

There are three reserved members, `lpRserved`, `lpReserved2` and `cbReserved2` - these should be set to `NULL`, `NULL` and zero, respectively.

The `lpDesktop` member specifies an alternate Window Station for the new process and an alternate Desktop for the new thread. If this member is `NULL` (or an empty string), the parent process Window Station and Desktop are used. Alternatively, a full desktop name can be specified in the format *windowstation\desktop*. For example "winsta0\mydesktop" could be used. See the sidebar *Window Stations and Desktops* for more information.

Window Stations and Desktops

A Window Station is a kernel object which is part of a session. It contains user related objects:

> A clipboard, an atom table and desktops. A desktop contains windows, menus and hooks. A process is associated with a single Window Station. The interactive Window Station is always named "WinSta0" and is the only one within a session that can be "interactive", meaning used with input devices.
>
> By default, a interactive logon session has a Window Station named "winsta0", in which two desktops exist: the "default" desktop, where the user normally works - where you see Explorer, the task bar and whatever else you normally run. Another desktop (created by the *Winlogon.exe* process is called "Winlogon" and is the one used when pressing the famous Ctrl+Alt+Del key combination. Windows calls the `SwitchDesktop` function to switch the input desktop to the "Winlogon" desktop. Desktops can be created or opened with `CreateDesktop` and `OpenDesktop`, respectively.
>
> You can find more details in my blog post at https://scorpiosoftware.net/2019/02/17/windows-10-desktops-vs-sysinternals-desktops/.

The `lpTitle` member can hold the title for console applications. If `NULL`, the executable name is used as the title. If `dwFlags` has the flag `STARTF_TITLEISAPPID` (Windows 7 and later), then `lpTitle` is an `AppUserModelId`, which is a string identifier that the shell uses for task bar item grouping and jump lists. Processes can set their AppUserModelId explicitly by calling `SetCurrentProcessExplicitAppUserModelID` rather than letting their parent dictate it. Working with jump lists and other task bar features are beyond the scope of this book.

`dwX` and `dwY` can be set as default location for windows created by the process. They are used only if `dwFlags` includes `STARTF_USEPOSITION`. The new process can use these values if its calls to `CreateWindow` or `CreateWindowEx` use `CW_USEDEFAULT` for the window's position. (See the `CreateWindow` function documentation for more details.) `dwXSize` and `dwYSize` are similar, specifying default width and height of new windows created by the child process if it uses `CW_USEDEFAULT` as width and height in calls to `CreateWindow`/`CreateWindowEx`. (Of course `STARTF_USESIZE` must be set in `dwFlags` for these values to propagate).

`dwXCountChars` and `dwYCountChars` set the initial width and height (in characters) of a console window created by the child process (if any). As with the previous members, `dwFlags` must have `STARTF_USECOUNTCHARS` for these values to have any effect.

`dwFillAttribute` specifies the initial text and background colors if a new console window is created with the process. As usual, this member has effect if `dwFlags` includes `STARTF_USE-FILLATTRIBUTE`. The possible color combinations have 4 bits each, leading to 16 combinations for text and background. The possible values are shown in table 3-8.

Table 3-8: Color values for consoles

Color constant	Value	Text/background
FOREGROUND_BLUE	0x01	Text
FOREGROUND_GREEN	0x02	Text
FOREGROUND_RED	0x04	Text
FOREGROUND_INTENSITY	0x08	Text
BACKGROUND_BLUE	0x10	Background
BACKGROUND_GREEN	0x20	Background
BACKGROUND_RED	0x40	Background
BACKGROUND_INTENSITY	0x80	Background

`wShowWindow` (valid if `dwFlags` includes `STARTF_USESHOWWINDOW`) indicates the way the main window should be shown by the process (assuming it has a GUI). These are values normally passed to the `ShowWindow` function, with the `SW_` prefix. `wShowWindow` is unique because it's provided directly in the `WinMain` function as the last parameter. Of course the created process can just disregard the provided value and show its windows in any way it sees fit. But it's a good practice to honor this value. If the creator does not provide this member, `SW_SHOWDEFAULT` is used, indicating the application can use any logic in displaying its main window. For example, it may have saved the last window position and state (maximized, minimized, etc.) and so it will restore the window to the saved position/state.

One scenario where this value is controllable is with shortcuts created using the shell. Figure 3-12 shows a shortcut created for running *Notepad*. In the shortcut properties, the way the initial window is shown can be selected: normal, minimized or maximized. Explorer propagates this value in the `wShowWindow` member when creating the process (`SW_SHOWNORMAL`, `SW_SHOWMINNOACTIVE`, `SW_SHOWMAXIMIZED`).

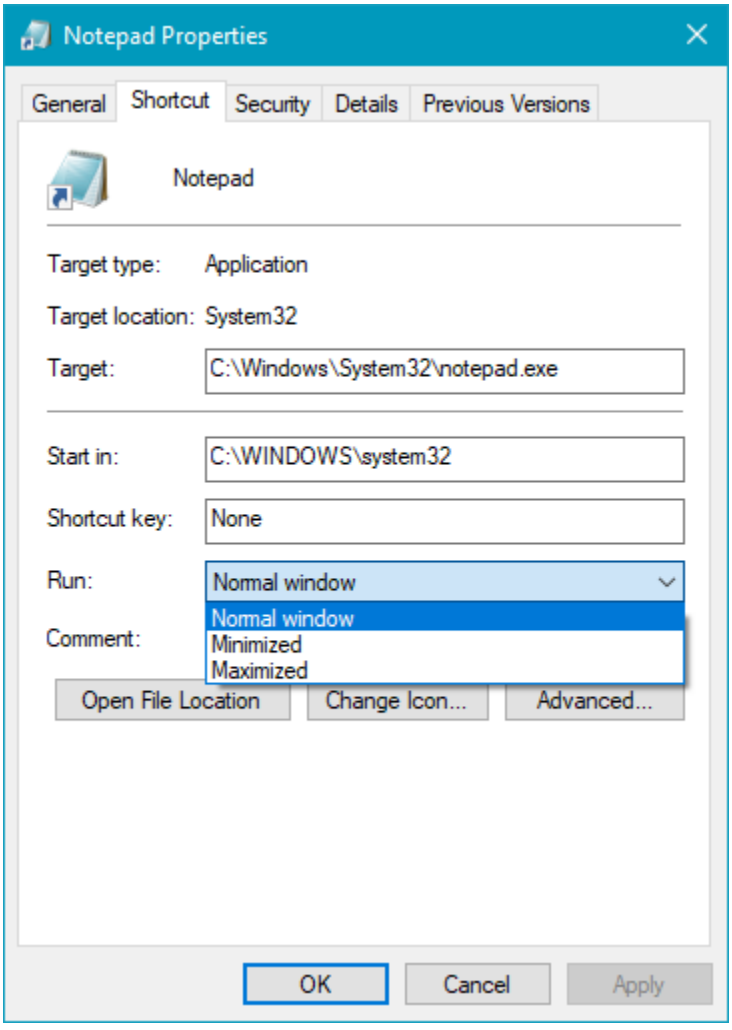


Figure 3-12: Show window in shortcuts

The last three members in `STARTUPINFO` are handles to standard input (`hStdInput`), output (`hStdOutput`) and error (`hStdError`). If `dwFlags` contains `STARTF_USEHANDLES`, these handles will be used in the new process as such. Otherwise, the new process will have the defaults: input from keyboard, output and error to the console buffer.

> If the process is launched from the taskbar or a jump list (Windows 7+), the `hStdOutput` handle is actually a handle to a monitor (`HMONITOR`) from which the process was launched.

Given all these various options for process creation (and there are more discussed in the section "Process (and Thread) Attributes" later in this chapter), it may seem daunting to create a process, but in most cases it's fairly straightforward if the defaults are acceptable. The following code snippet creates an instance of *Notepad*:

```
WCHAR name[] = L"notepad";
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;

BOOL success = ::CreateProcess(nullptr, name, nullptr, nullptr, FALSE,
    0, nullptr, nullptr, &si, &pi);
if (!success) {
    printf("Error creating process: %d\n", ::GetLastError());
}
else {
    printf("Process created. PID=%d\n", pi.dwProcessId);
    ::CloseHandle(pi.hProcess);
    ::CloseHandle(pi.hThread);
}
```

What can be done with the returned handles from `CreateProcess`? One thing is being notified when the process terminated (for whatever reason). This is done with the `WaitForSingleObject` function. This function is not specific to a process, but can wait for various kernel objects until they become **signaled**. The meaning of *signaled* depends on the object type; for a process, it means terminated. A detailed discussion of the "wait" functions is saved for chapter 8. Here, we'll look at a couple of examples. First we can wait indefinitely until the process exits:

```
// process creation succeeded
printf("Process created. PID=%d\n", pi.dwProcessId);
::WaitForSingleObject(pi.hProcess, INFINITE);
printf("Notepad terminated.\n");

::CloseHandle(pi.hProcess);
::CloseHandle(pi.hThread);
}
```

`WaitForSingleObject` puts the calling thread into a wait state until the object in question changes to the *signaled* state or the timeout expires. In case of `INFINITE` (-1), it never expires. Here is an example for a non-`INFINITE` timeout:

```
DWORD rv = ::WaitForSingleObject(pi.hProcess, 10000);   10 seconds
if (rv == WAIT_TIMEOUT)
    printf("Notepad still running...\n");
else if (rv == WAIT_OBJECT_0)
    printf("Notepad terminated.\n");
else // WAIT_ERROR (unlikely in this case)
    printf("Error! %d\n", ::GetLastError());
```

The calling thread blocks for no more then 10000 milliseconds, after which the returned value indicates the state of the process.

A process can always get the `STARTUPINFO` it was created with by calling `GetStartupInfo`.

Handle Inheritance

In chapter 2 we looked at ways to share kernel objects between processes. One is sharing by name the other by duplicating handles. The third option is to use handle inheritance. This option is only available if a process creates a child process. At the point of creation, the parent process can duplicate a selected set of handles to the target process. Once `CreateProcess` is called with the fifth argument set to `TRUE`, all handles in the parent process that have their inheritance bit set will be duplicated into the child process, where the handle values are the same as in the parent process.

The last sentence is important. The child process cooperates with the parent process (presumably they are part of the same software system), and it knows it's going to get some handles from its parent. What it does *not* know is what the values of these handles are. One simple way of providing these values is using the command line arguments sent to the process being created.

Setting a handle to be inheritable can be done in several ways:

- If the object in question is created by the parent process, then its `SECURITY_ATTRIBUTES` can be initialized with a handle inheritance flag and passed to the *Create* function like so:

```
SECURITY_ATTRIBUTES sa = { sizeof(sa) };
sa.bInheritHandles = TRUE;

HANDLE h = ::CreateEvent(&sa, FALSE, FALSE, nullptr);

// handle h will be inherited by child processes
```

- For an existing handle, call `SetHandleInformation`:

```
::SetHandleInformation(h, HANDLE_FLAG_INHERIT , HANDLE_FLAG_INHERIT);
```

- Lastly, most *Open* functions allow setting the inheritance flag on a successfully returned handle. Here is an example for a named event object:

```
HANDLE h = ::OpenEvent(EVENT_ALL_ACCESS,
    TRUE,    // inheritable
    L"MyEvent");
```

The *InheritSharing* application is yet another variation on the memory sharing applications from chapter 2. This time, the sharing is achieved by inheriting the memory mapping handle to child processes created from a first process. The dialog now has an extra *Create* button to spawn new processes with an inherited shared memory handle (figure 3-13).

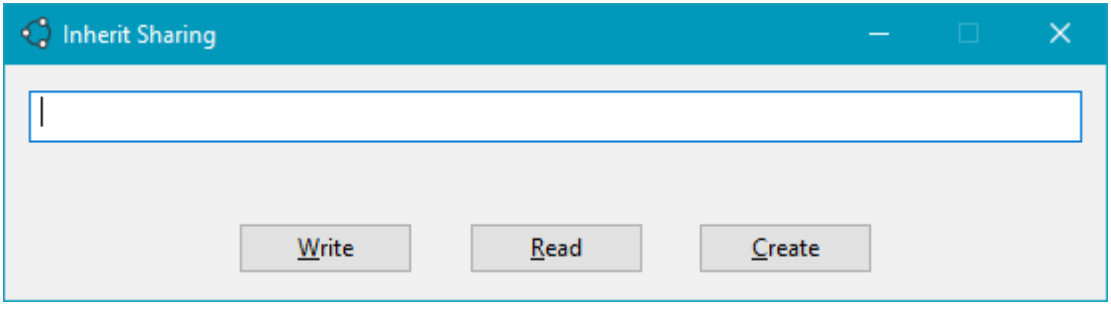


Figure 3-13: Sharing by Inheritance application

A *InheirtSharing* process creates another instance of itself when the *Create* button is clicked. The new instance must get a handle to the shared memory object, and this is done by inheritance: the existing shared memory handle (held in a `wil::unique_handle` object) needs to be made inheritable so that it can be duplicated to the new process. The *Create* button click handler starts with setting the inheritance bit:

```
::SetHandleInformation(m_hSharedMem.get(), HANDLE_FLAG_INHERIT, HANDLE_FLAG_INH\
ERIT);
```

Now the new process can be created with the fifth argument set to `TRUE`, indicating all inheritable handles are to be duplicated for the new process. In addition, the new process needs to know the value of its duplicated handle, and this is passed in the command line:

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;

// build command line
WCHAR path[MAX_PATH];
::GetModuleFileName(nullptr, path, MAX_PATH);
WCHAR handle[16];
::_itow_s((int)(ULONG_PTR)m_hSharedMem.get(), handle, 10);
::wcscat_s(path, L" ");
::wcscat_s(path, handle);

// now create the process

if (::CreateProcess(nullptr, path, nullptr, nullptr, TRUE,
    0, nullptr, nullptr, &si, &pi)) {
    // close unneeded handles
    ::CloseHandle(pi.hProcess);
    ::CloseHandle(pi.hThread);
}
else {
    MessageBox(L"Failed to create new process", L"Inherit Sharing");
}
```

The command line is built by first calling `GetModuleFileName`, which generally allows getting a full path for any DLL loaded in the process. With the first argument set to `NULL`, the executable full path is returned. This approach is robust, such that there is no dependency on the actual location of the executable in the file system.

Once this path is returned, the handle value is appended as a command-line argument. Remember that an inherited handle always has the same value as in the original process. This is possible because the new process handle table is initially empty, so the entry is definitely unused.

The last piece of the puzzle is when the process starts up. It needs to know whether it's the first instance, or an instance that gets an existing inherited handle. In the `WM_INITDIALOG` message handler, the command line needs to be examined. If there is no handle value in the command

line then the process needs to create the shared memory object. Otherwise, it needs to grab the handle and just use it.

```
int count;
PWSTR* args = ::CommandLineToArgvW(::GetCommandLine(), &count);
if (count == 1) {
    // "master" instance
    m_hSharedMem.reset(::CreateFileMapping(INVALID_HANDLE_VALUE,
        nullptr, PAGE_READWRITE, 0, 1 << 16, nullptr));
}
else {
    // first "real" argument is inherited handle value
    m_hSharedMem.reset((HANDLE)(ULONG_PTR)::_wtoi(args[1]));
}
::LocalFree(args);
```

Since this is not `WinMain`, the command line arguments are not readily available. `GetCommandLine` can always be used to get the command line at any time. Then `CommandLineToArgvW` is used to parse the arguments (discussed earlier in this chapter). If no handle value is passed in, `CreateFileMapping` is used to create the shared memory. Otherwise, the value is interpreted as a handle and attached to the `wil::unique_handle` object for safe keeping.

You can try creating a new instance from a child process - it works in exactly the same way as using the "original" handle to propagate to the child process.

Debugging Child Processes with Visual Studio

In the *InheritSharing* application, it's desirable to debug not just the main instance, but also a child process, because it's initiated with a different command line. Visual Studio by default does not debug child processes (processes created by the debugged process).

There is, however, an extension to Visual Studio that allows this. Open the extensions dialog (*Tools/Extensions and Updates* in VS 2017, *Extensions/Manage Extensions* in VS 2019), go to the *Online* node and search for *Microsoft Child Process Debugging Power Tool* and install it (figure 3-14).

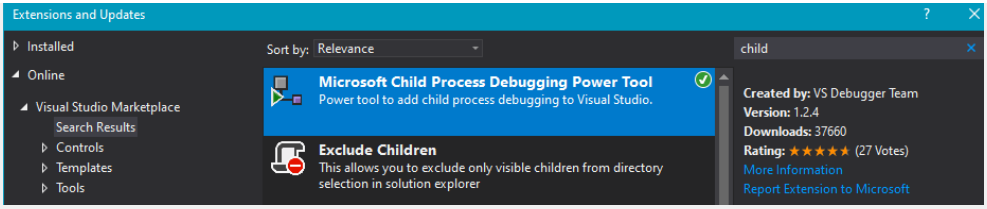


Figure 3-14: Child Process Debugging Power Tool in extensions

Once installed, go to *Debug/Other Debug Targets/Child Process Debugger Settings...*, check *Enable Child Process Debugging* and click *Save*. Now set a breakpoint at `CMainDlg::OnInitDialog` and start debugging normally (*F5*).

The first time you hit the breakpoint is when the dialog comes up for a fresh process, creating its own shared memory object. The `count` variable should be 1 (figure 3-15).

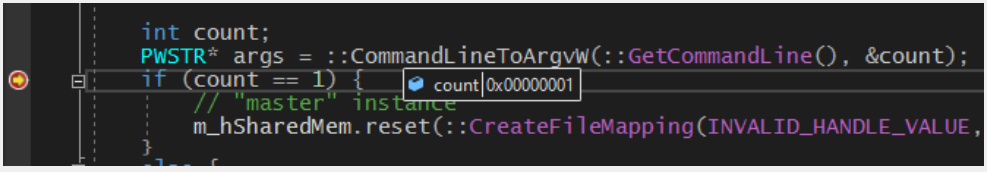


Figure 3-15: Breakpoint in the first process

Continue debugging and click *Create*. A new process should come up under the control of the debugger and the same breakpoint should hit again (figure 3-16). Notice `count` is now 2. Also notice it's a different process - the Processes toolbar combobox should show two processes (figure 3-17).

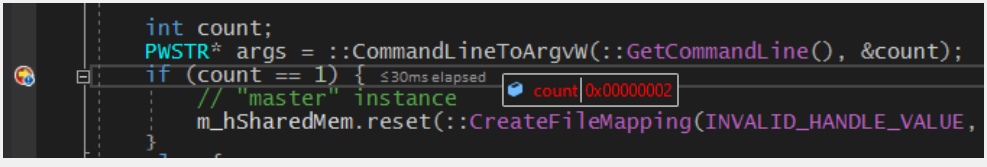


Figure 3-16: Breakpoint in the second process

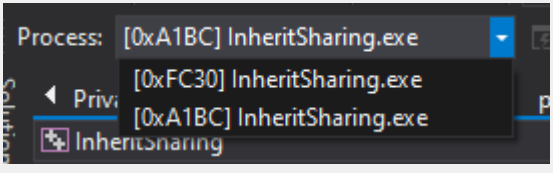


Figure 3-17: Debugging multiple processes

Process Drive Directories

Every process has its current directory, set with `SetCurrentDirectory` and retrieved with `GetCurrentDirectory`. This directory is used when accessing a file without any path prefix such as "mydata.txt". What about default directories when accessing a file with a drive prefix like "c:mydata.txt" (notice the lack of a backslash).

As it turns out, the system keeps track of the current directory for each drive using process environment variables. If you call `GetEnvironmentStrings`, you'll discover something like the following at the beginning of the block:

```
=C:=C:\Dev\Win10SysProg
=D:=D:\Temp
```

To get the current directory for a drive call `GetFullPathName`:

```
DWORD GetFullPathNameW(
    _In_ LPCWSTR lpFileName,
    _In_ DWORD nBufferLength,
    _Out_ LPWSTR lpBuffer,
    _Outptr_opt_ LPWSTR* lpFilePart);
```

Generally speaking, this function returns the full path of a given file name. Specifically, with a drive letter it returns its current directory. Here is an example:

```
WCHAR path[MAX_PATH];
::GetFullPathName(L"c:", MAX_PATH, path, nullptr);
```

Do not append a backslash to the drive letter after the colon! If you do, you'll just get the same string back.

Calling the function with a drive letter and a file name returns the resulting full path of the drive's current directory and the file name:

```
WCHAR path[MAX_PATH];
::GetFullPathName(L"c:mydata.txt", MAX_PATH, path, nullptr);    // no backslash
```

The above code might return something like "c:Win10SysProg\mydata.txt".

`GetFullPathName` does not check for the existence of the file provided.

Process (and Thread) Attributes

The STARTUPINFO structure we met in CreateProcess has quite a few fields in it. It stands to reason that future versions of Windows may require more ways to customize process creation. One possible way would be to extend the STARTUPINFO structure and add more flags to make certain members valid. Microsoft decided to extend STARTUPINFO in a different way starting with Windows Vista.

An extended structure, STARTUPINFOEX is defined, that extends STARTUPINFO, shown here again for convenience:

```
typedef struct _STARTUPINFOEX {
    STARTUPINFO StartupInfo;
    PPROC_THREAD_ATTRIBUTE_LIST pAttributeList;
} STARTUPINFOEXW, *LPSTARTUPINFOEXW;
```

The memory layout of STARRTUPINFOEX starts with a STARTUPINFO with just one added member: an opaque attribute list. This attribute list is the main extension mechanism for CreateProcess (and CreateRemoteThreadEx discussed in chapter 5). Since this attribute list can point to any number of attributes, there is no need to extend STARTUPINFOEX further.

Creating and filling an attribute list requires the following steps:

1. Allocate and initialize an attribute list with InitializeProcThreadAttributeList.
2. Add attributes as required by calling UpdateProcThreadAttribute once for each attribute.
3. Set the pAttribute member of STARTUPINFOEX to point to the attribute list.
4. Call CreateProcess with the extended structure, not forgetting to add the flag EXTENDED_STARTUPINFO_PRESENT to the creation flags (sixth parameter to CreateProcess)
5. Delete the attribute list with DeleteProcThreadAttributeList.

Let's take these steps in turn. Here is the simplified declaration of InitializeProcThreadAttributeList:

```
BOOL InitializeProcThreadAttributeList(
    _Out_ PPROC_THREAD_ATTRIBUTE_LIST pAttributeList,
    _In_ DWORD dwAttributeCount,
    _Reserved_ DWORD dwFlags,   // must be zero
    PSIZE_T pSize);
```

The first step is to allocate a buffer large enough to hold the required number of attributes. This is done by calling `InitializeProcThreadAttributeList` twice: first to get the required size, allocate a buffer, and then make a second call to initialize the buffer to hold an attribute list.

The following example performs these steps (error handling omitted):

```
SIZE_T size;
// get required size
::InitializeProcThreadAttributeList(nullptr, 1, 0, &size);

// allocate the required size
auto attlist = (PPROC_THREAD_ATTRIBUTE_LIST)malloc(size);

// initialize
::InitializeProcThreadAttributeList(attlist, 1, 0, &size);  // just one attribu\
te
```

The first call to `InitializeProcThreadAttributeList` returns `FALSE`, with `GetLastError` returning 122 ("The data area passed to a system call is too small."). This is expected, since the real return value is the required size. The attribute list itself must be allocated by the caller (`malloc` is used in the above snippet), which also means it must be freed after the call to `CreateProcess`.

Next, `UpdateProcThreadAttribute` is called a number of times according to the count of attributes. The list of possible attributes has grown with almost every Windows release, and is likely to continue growing. Table 3-9 shows the *documented* attributes for processes and threads (at the time of writing), with a brief description.

Table 3-9: Documented process and thread attributes

Attribute constant `PROC_THREAD_ATTRIBUTE_`	Applies to	Minimum version	Description
`PARENT_PROCESS`	Process	Windows Vista	Sets a different parent process from which to inherit various properties
`HANDLE_LIST`	Process	Windows Vista	Specified a list of handles to be inherited by the child process
`GROUP_AFFINITY`	Thread	Windows 7	Sets the default CPU affinity group for the new thread (see chapter 6)
`PREFERRED_NODE`	Process	Windows 7	Sets the preferred NUMA node for the new process

Table 3-9: Documented process and thread attributes

Attribute constant PROC_THREAD_ATTRIBUTE_	Applies to	Minimum version	Description
IDEAL_PROCESSOR	Thread	Windows 7	Sets the ideal CPU for the new thread (see chapter 6)
UMS_THREAD	Thread	Windows 7	Sets the User Mode Scheduling (UMS) context for the new thread (see chapter 10)
MITIGATION_POLICY	Process	Windows 7	Sets security mitigation policies for the new process (see chapter 16)
SECURITY_CAPABILITIES	Process	Windows 8	Sets the security capabilities of an AppContainer (see chapter 16)
PROTECTION_LEVEL	Process	Windows 8	Launches the new process with the same protection level as the creator
CHILD_PROCESS_POLICY	Process	Windows 10	Specifies whether the new process can create child processes
DESKTOP_APP_POLICY	Process	Windows 10 (1703)	Applies to applications converted to UWP using *Desktop Bridge*. Specifies whether the new process' child processes will be created outside the desktop app environment

> *Desktop Bridge* is discussed in chapter 18.

`UpdateProcThreadAttribute` is defined like so:

```
BOOL UpdateProcThreadAttribute(
    _Inout_ PPROC_THREAD_ATTRIBUTE_LIST pAttributeList,
    _In_ DWORD dwFlags,         // must be zero
    _In_ DWORD_PTR Attribute,
    _In_ PVOID pValue,
    _In_ SIZE_T cbSize,
    _Out_ PVOID pPreviousValue,     // must be NULL
    _In_opt_ PSIZE_T pReturnSize);  // must be NULL
```

The following example uses PROC_THREAD_ATTRIBUTE_PARENT_PROCESS attribute to set a different parent by specifying a handle to another process:

```
    HANDLE hParent = ...;
    ::UpdateProcThreadAttribute(attlist, 0, PROC_THREAD_ATTRIBUTE_PARENT_PROCES\
S,
        &hParent, sizeof(hParent), nullptr, nullptr);
```

With PROC_THREAD_ATTRIBUTE_PARENT_PROCESS, the attribute value is an open handle to the relevant process. See the documentation for details on the other attributes values.

Once the attribute list is fully updated, the call to CreateProcess can commence, being careful to use the correct structure and flags for the attributes to have any effect:

```
STARTUPINFOEX si = { sizeof(si) };
si.lpAttributeList = attlist;

PROCESS_INFORMATION pi;
WCHAR name[] = L"Notepad";

::CreateProcess(nullptr, name, nullptr, nullptr, FALSE,
    EXTENDED_STARTUPINFO_PRESENT, nullptr, nullptr, (STARTUPINFO*)&si, &pi);
```

There are two things that must be set to indicate an attribute list: the STARTUPINFOEX structure and the flag EXTENDED_STARTUPINFO_PRESENT. Without the latter, the attributes won't be applied.

The last step is cleaning up the attribute list and the allocated memory for it:

```
::DeleteProcThreadAttributeList(attList);
::free(attList);
```

Given the above steps, the following function creates a given process parented with another process based on its ID:

```
DWORD CreateProcessWithParent(PWSTR name, DWORD parentPid) {
    HANDLE hParent = ::OpenProcess(PROCESS_CREATE_PROCESS, FALSE, parentPid);
    if (!hParent)
        return 0;

    PROCESS_INFORMATION pi = { 0 };
    PPROC_THREAD_ATTRIBUTE_LIST attList = nullptr;

    do {
        SIZE_T size = 0;
        ::InitializeProcThreadAttributeList(nullptr, 1, 0, &size);
        if (size == 0)
            break;

        attList = (PPROC_THREAD_ATTRIBUTE_LIST)malloc(size);
        if (!attList)
            break;

        if (!::InitializeProcThreadAttributeList(attList, 1, 0, &size))
            break;

        if (!::UpdateProcThreadAttribute(attList, 0, PROC_THREAD_ATTRIBUTE_PARE\
NT_PROCESS,
            &hParent, sizeof(hParent), nullptr, nullptr))
            break;

        STARTUPINFOEX si = { sizeof(si) };
        si.lpAttributeList = attList;
        if (!::CreateProcess(nullptr, name, nullptr, nullptr, FALSE,
            EXTENDED_STARTUPINFO_PRESENT, nullptr, nullptr, (STARTUPINFO*)&si, \
&pi))
            break;

        ::CloseHandle(pi.hProcess);
        ::CloseHandle(pi.hThread);
    } while (false);

    ::CloseHandle(hParent);
    if (attList) {
        ::DeleteProcThreadAttributeList(attList);
        ::free(attList);
```

```
    }
    return pi.dwProcessId;
}
```

Most of the code is the same as the steps outlined for working with attributes, with error handling added and proper cleanup. Notice the process handle is opened with the `PROCESS_CREATE_PROCESS` access mask. This is required when using the `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS` attribute. This means not all processes can just arbitrarily serve as parents.

Running this function from with *Notepad* and a PID of an *Explorer* process created *Notepad* as expected. Opening its properties in *Process Explorer* shows *Explorer* as being the parent (figure 3-18).

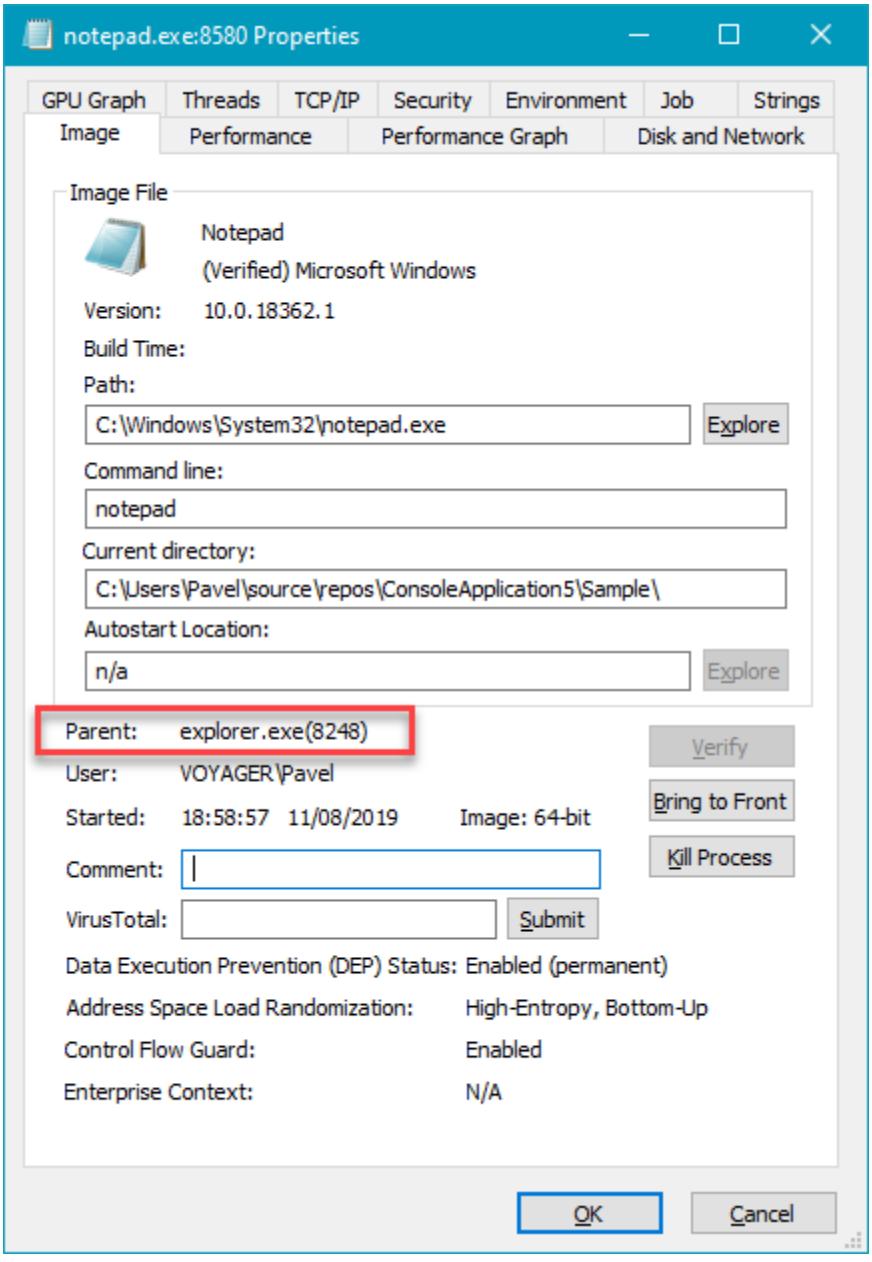


Figure 3-18: Changing parent process

As another example, consider the following function that applied a process security mitigation policy:

```
DWORD CreateProcessWithMitigations(PWSTR name, DWORD64 mitigation) {
    PROCESS_INFORMATION pi = { 0 };
    PPROC_THREAD_ATTRIBUTE_LIST attList = nullptr;

    do {
        SIZE_T size = 0;
        ::InitializeProcThreadAttributeList(nullptr, 1, 0, &size);
        if (size == 0)
            break;

        attList = (PPROC_THREAD_ATTRIBUTE_LIST)malloc(size);
        if (!attList)
            break;

        if (!::InitializeProcThreadAttributeList(attList, 1, 0, &size))
            break;

        if (!::UpdateProcThreadAttribute(attList, 0, PROC_THREAD_ATTRIBUTE_MITI\
GATION_POLICY,
            &mitigation, sizeof(mitigation), nullptr, nullptr))
            break;

        STARTUPINFOEX si = { sizeof(si) };
        si.lpAttributeList = attList;
        if (!::CreateProcess(nullptr, name, nullptr, nullptr, FALSE,
            EXTENDED_STARTUPINFO_PRESENT, nullptr, nullptr, (STARTUPINFO*)&si, \
&pi))
            break;

        ::CloseHandle(pi.hProcess);
        ::CloseHandle(pi.hThread);
    } while (false);

    if (attList) {
        ::DeleteProcThreadAttributeList(attList);
        ::free(attList);
    }
    return pi.dwProcessId;
}
```

The code is nearly identical except for the different attribute. The value associated with `PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY` is a `DWORD` or `DWORD64` indicating the mitigation(s) to

apply to the new process.

> Full discussion of process mitigations is in chapter 16.

For example, calling this function with the following arguments

```
WCHAR name[] = L"notepad";
auto pid = CreateProcessWithMitigations(name,
    PROCESS_CREATION_MITIGATION_POLICY_WIN32K_SYSTEM_CALL_DISABLE_ALWAYS_ON);
```

Causes *Notepad* to fail initialization and terminate, not before showing the message box in figure 3-19. The reason is that the specific mitigation prevents calls to *Win32k.sys* (the windowing manager), which essentially means *User32.dll* cannot properly initialize. Without this capability *Notepad* is useless and cannot properly execute. This mitigation is good for processes that have no UI and want to make sure *Win32k.sys* security vulnerabilities cannot be used in such a process.

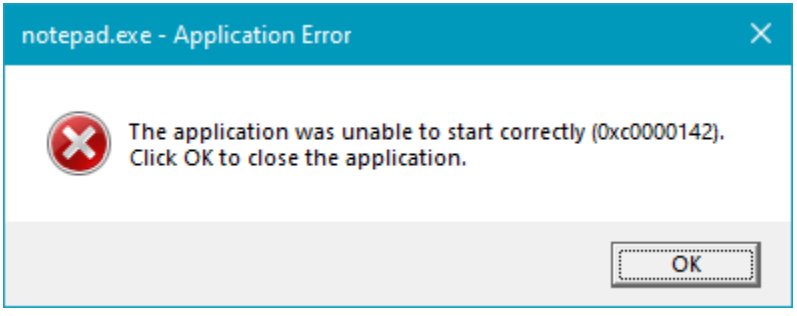


Figure 3-19: Notepad failing to initialize

Protected and PPL Processes

Protected processes were introduced in Windows Vista as a way to fight *Digital Rights Management* (DRM) infringements. These processes be granted certain access rights even by admin-level users. The only access masks allowed for protected processes are: `PROCESS_QUERY_LIMITED_INFORMATION`, `PROCESS_SET_LIMITED_INFORMATION`, `PROCESS_SUSPEND_RESUME` and `PROCESS_TERMINATE`.
Only Microsoft-signed executables with a certain *Extended Key Usage* (EKU) were allowed to execute protected.

Windows 8.1 introduced *Protected Processes Light* (PPL), which extends the protection model to include several levels of protection, where higher level protected processes have full access to

lower level ones, but not vice versa. With the extended model, it is now possible to run third party anti-malware services by negotiating with Microsoft and obtaining the proper signature. Also, some of the PPL levels (such as for anti-malware) deny `PROCESS_TERMINATE` access, so that malicious software, even with elevated permissions, cannot stop or kill these services. Table 3-10 lists the PPL signer levels with a brief description.

Table 3-10: PPL signers

PPL Signer	Level	Description
WinSystem	7	System and minimal processes
WinTcb	6	Critical Windows components. `PROCESS_TERMINATE` is denied
Windows	5	Important Windows components handling sensitive data
Lsa	4	Lsass.exe (if configured to run protected)
Antimalware	3	Anti-malware service processes, including 3rd party. `PROCESS_TERMINATE` is denied
CodeGen	2	.NET native code generation
Authenticode	1	Hosting DRM content
None	0	Not valid (no protection)

> The level (and whether the process is "normal" protected or PPL) shown in table 3-10 is stored inside the kernel process object.

The "limited" access masks allowed for protected/PPL processes cater for setting/querying superficial information about the process, such as querying its start time, its priority class or its executable path. Getting a list of loaded modules inside a protected process cannot be obtained because this requires an access mask of `PROCESS_QUERY_INFORMATION`, which is not allowed. Figure 3-20 shows *Csrss.exe* selected in *Process Explorer*. Notice that the list of modules in the bottom pane is empty. Also, the *Protection* column is shown, with the signer values from table 3-10.

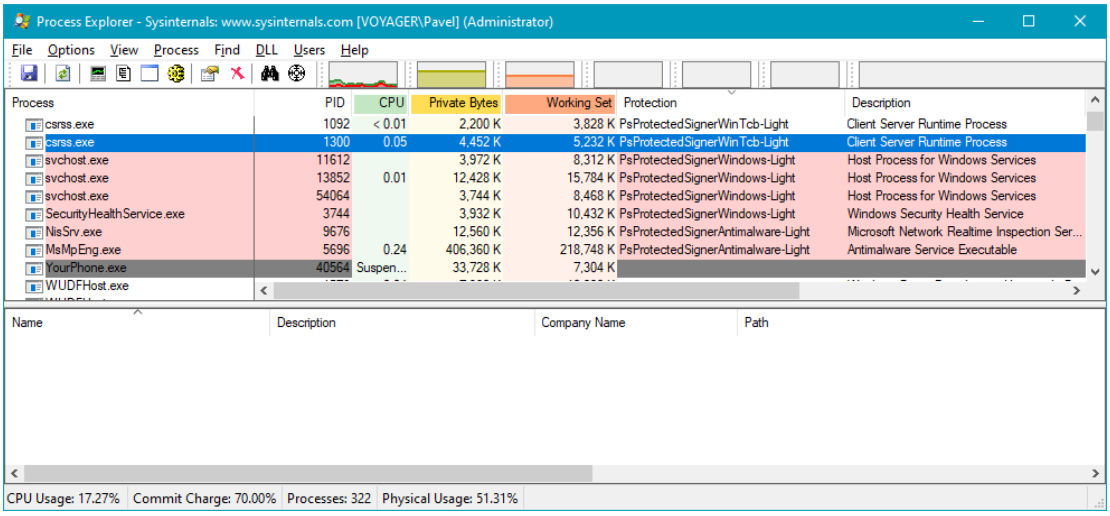


Figure 3-20: Protected Processes

Figure 3-20 also shows Microsoft's own anti-malware executables (*MsMpEng.exe* and *NisSrv.exe*, known as "Windows Defender") running as anti-malware PPL, just like other 3rd party anti-malware services.

Protected and PPL processes cannot load arbitrary DLLs, so that the protected process is not tricked into loading an untrusted DLL that would run under the protection of the process. All DLLs loaded by protected/PPL processes must be signed properly.

Creating a process as protected requires the `CREATE_PROTECTED_PROCESS` flag in `CreateProcess`. Of course, this can only work on properly signed executables. The protection mechanism is too specialized to be used by normal applications and so is not going to be discussed further in this book.

You can find more information on protected/PPL processes in the "Windows Internals 7th edition Part 1" book in chapter 3.

UWP Processes

A *Universal Windows Platform* (UWP) process is similar to any other standard process. It uses the Windows Runtime platform/API to do most of its work - be that UI, graphics, networking, background processing and so on. Some of its unique properties include:

- A UWP process always runs under an application sandbox known as *AppContainer* that limits what it can do and what it can access (discussed in more detail in chapter 16, "Security").

- A UWP process' state is managed by the *Process Lifetime Manager* (PLM), running under the *Explorer.exe* process, that can initiate process suspension, resumption, and termination based on its foreground/background activity and memory usage (discussed more in chapter 18).
- A UWP package includes a set of capabilities - declarations - of what the application wants to access (such as camera, location, Pictures folder) and those capabilities are listed in the Microsoft Store so that users can decide whether that would like to download such an application.
- UWP processes are single instance by default (multiple instances supported starting with Windows 10 version 1803).

From a process creation perspective, a standard `CreateProcess` call cannot create a UWP process. This is because a UWP application has **identity**, something that is missing from standard executables. Such an application is built into a package with the executable, libraries, resource files and anything else that is needed for the application to execute properly. This package has a universally unique name, and this name is the one required for UWP process creation.

You can view this name in *Task Manager* or *Process Explorer* by adding the *Package Name* column.

As a simple example to this requirement, run *Calculator* on Windows 10, and look at its properties with *Process Explorer* (figure 3-21). Notice the command line; disregard its length and ugliness and just copy it, and use *Start/Run* to launch another calculator with the pasted in the command-line. You'll get an error message box similar to the one in figure 3-22.

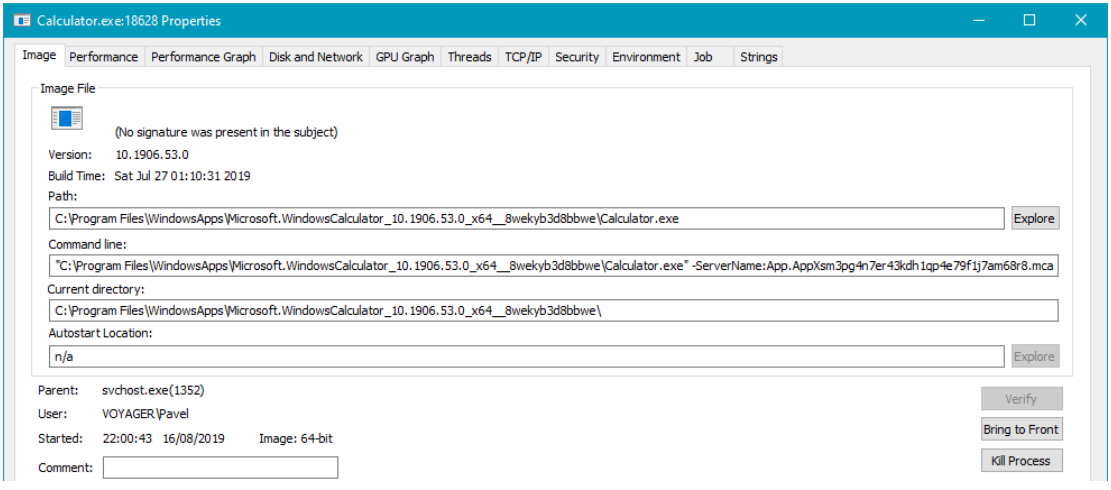


Figure 3-21: *Calculator* properties in *Process Explorer*

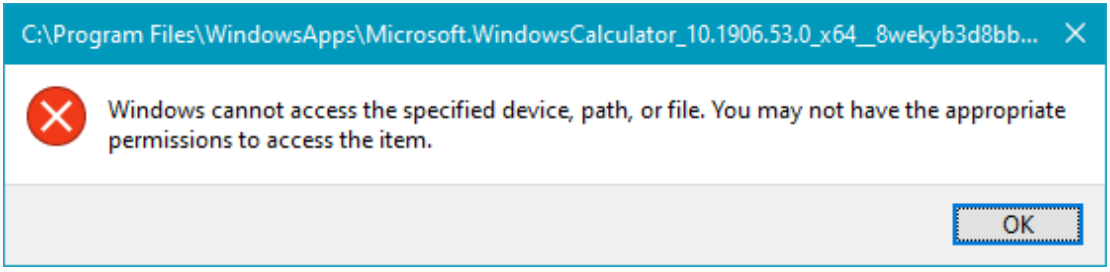


Figure 3-22: Error message box when creating Calculator manually

The error message in figure 3-22 seems unrelated to anything. The missing piece of information is the package full name, which needs to be specified as a process attribute. Unfortunately, this particular attribute is undocumented, so cannot be specified with the help of the Windows headers. There is a way to specify this parameter using another creation mechanism devised just for this purpose and exposed through a COM interface (and class).

The *MetroManager* application, shown in figure 3-23 lists the available UWP packages on the machine, and allows the user to launch any selected package. The application demonstrates a few interesting abilities:

- Consuming Windows Runtime APIs from a non-UWP application.
- Enumerating packages
- Launching UWP processes in the documented way.

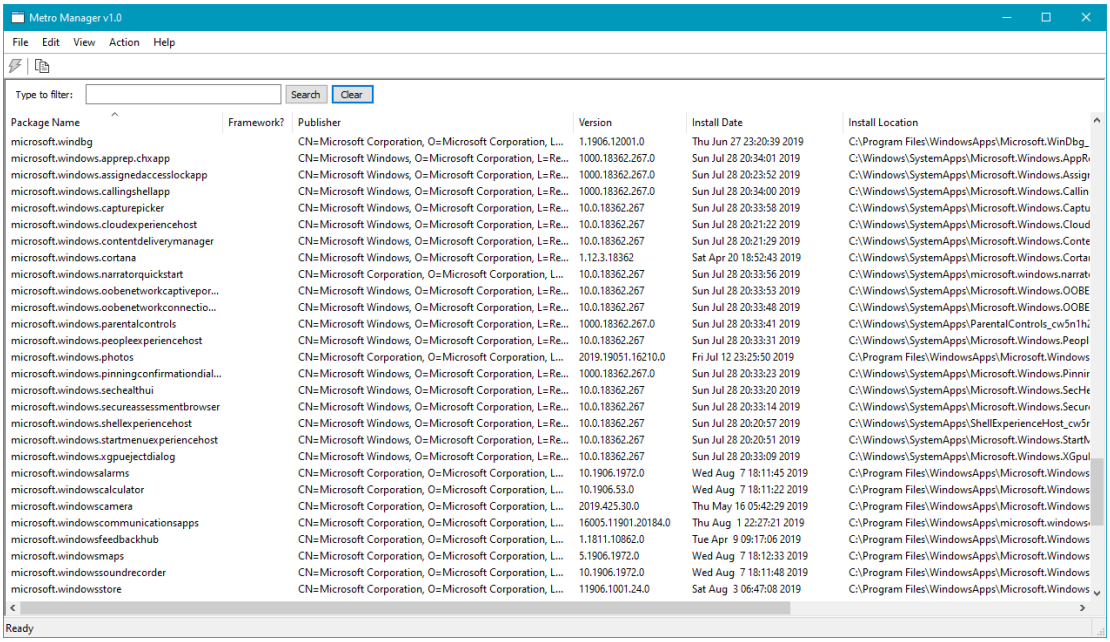


Figure 3-23: The *Metro Manager* application

The Windows Runtime (WinRT) is built on top of COM, which means it utilizes interfaces, classes, class factories, GUIDs and other concepts from the COM world (albeit giving to some new names). The Windows Runtime supports entities typically found in high level languages, including static methods and generics. We'll look at how this works in chapter 18. Here, I want to concentrate mostly on the mechanics of using the Windows Runtime.

The Windows Runtime API can be consumed by a C++ client in several ways:

1. Directly, by instantiating the proper classes and working with the low level object factories until an instance is created and then use normal COM calls.
2. Use the *Windows Runtime Library* (WRL) C++ wrappers and helpers.
3. Use the C++/CX language extensions, that provide an easy access to WinRT by extending C++ in a non-standard way.
4. Use the CppWinRT library, that provides relatively easy access to the WinRT APIs with standard C++ only.

All the above four ways are officially supported. The first option is the most tedious and is only recommended for learning purposes, as it hides very little from the developer and so is very verbose. Option 2 is easier to use, but is not favored today, not even by Microsoft. Option 3 is the easiest and was the most common in the early days of WinRT, but is frowned upon today because it forces the developer to use non-standard C++ extensions. This leaves option 4, which

is the recommended way to work with WinRT in C++, as it's easy enough to work with but still uses standard C++ constructs.

CppWinRT is beyond the scope of this book, but we'll cover the basics which can get you quite far. First, we need to add the Nuget package for the library (figure 3-24). Next, we need to add includes for the namespaces that we wish to use from WinRT. The following was added to the pre-compiled header (*pch.h* in the project source code):

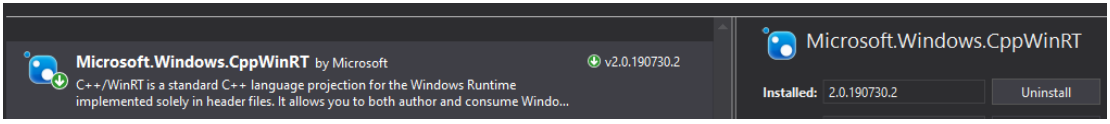


Figure 3-24: CppWinRT Nuget package

> More CppWinRT information is available in the online Microsoft documentation.

```
#include <winrt/Windows.Foundation.h>
#include <winrt/Windows.Foundation.Collections.h>
#include <winrt/Windows.ApplicationModel.h>
#include <winrt/Windows.Management.Deployment.h>
#include <winrt/Windows.Storage.h>
```

This is the general format of the CppWinRT headers: `winrt` prefix and then the namespace (found in the WinRT documentation).

> The headers without the `winrt` prefix are the "real" WinRT headers, that are included internally. You generally don't want these headers when working with CppWinRT.

Since all WinRT APIs are in namespaces and nested namespaces, the type names become long. It's much easier to add `using namespace` statements to a source file, or in some cases - a function to ease accessing the various types:

```
using namespace winrt;
using namespace winrt::Windows::Management::Deployment;
using namespace winrt::Windows::ApplicationModel;
```

The `winrt` namespace itself has some general CppWinRT helpers, so it's mandatory. The others depend on the types we use.

Enumerating the UWP packages is done with the `PackageManager` class in the `winrt::Windows::Management::Deployment` namespace:

```
auto packages = PackageManager().FindPackagesForUser(L"");
```

An empty string for the user looks at the current user (other users on the system may have installed different packages).

The `auto` keyword is real helper here as the actual returned type is `IIterable<Package>` (and I've shortened both types assuming we have `using namespace` for them). The important point is this method returns a collection (`IIterable<>`) in WinRT parlance. Because of the added conveniences in CppWinRT, any such collection can be iterated upon with the C++ enhanced range-based `for` statement:

```
for (auto package : packages) {
    auto item = std::make_shared<AppItem>();
    item->InstalledLocation = package.InstalledLocation().Path().c_str();
    item->FullName = package.Id().FullName().c_str();
    item->InstalledDate = package.InstalledDate();
    item->IsFramework = package.IsFramework();
    item->Name = package.Id().Name().c_str();
    item->Publisher = package.Id().Publisher().c_str();
    item->Version = package.Id().Version();

    m_AllPackages.push_back(item);
}
```

`AppItem` is a normal C++ class the application defines to store the information in plain C++ types rather than WinRT types where it makes sense, especially for strings:

```
struct AppItem {
    CString Name, Publisher, InstalledLocation, FullName;
    winrt::Windows::ApplicationModel::PackageVersion Version;
    winrt::Windows::Foundation::DateTime InstalledDate;
    bool IsFramework;
};
```

The standard Windows Runtime string is typed as `HSTRING`, which is an immutable array of UTF-16 characters stored with its length.

This `AppItem`-stored data is used to show the information in the application's list view.

Running a UWP package is accomplished with the following COM (not WinRT) interface, shown in C++:

```
struct IApplicationActivationManager : public IUnknown {
    virtual HRESULT __stdcall ActivateApplication(
        /* [in] */ LPCWSTR appUserModelId,
        /* [unique][in] */ LPCWSTR arguments,
        /* [in] */ ACTIVATEOPTIONS options,
        /* [out] */ DWORD *processId) = 0;

    virtual HRESULT __stdcall ActivateForFile(
        /* [in] */ LPCWSTR appUserModelId,
        /* [in] */ IShellItemArray *itemArray,
        /* [unique][in] */ LPCWSTR verb,
        /* [out] */ DWORD *processId) = 0;

    virtual HRESULT __stdcall ActivateForProtocol(
        /* [in] */ LPCWSTR appUserModelId,
        /* [in] */ IShellItemArray *itemArray,
        /* [out] */ DWORD *processId) = 0;
};
```

There are several ways to "activate" a UWP application, using something called *contracts*, where one of the contracts is called *Launch*, which naturally launches the application. `ActivateApplication` is using the Launch contract, while the others work with different contracts. We'll use the Launch contract only in this application.

The complete code for launching an app is in the `CView::RunApp` member function. First, we need to locate some information about a package using its unique package full name (stored in the `AppItem` structure shown earlier). Here is the first call:

```
bool CView::RunApp(PCWSTR fullPackageName) {
    PACKAGE_INFO_REFERENCE pir;
    int error = ::OpenPackageInfoByFullName(fullPackageName, 0, &pir);
    if (error != ERROR_SUCCESS)
        return false;
```

`OpenPackageInfoByFullName` returns an opaque pointer to an internal data structure that holds information about the requested package. Unfortunately, the full package name is not enough, because theoretically a package can contain multiple applications (which is not yet supported) and so another application ID needs to be extracted from the package:

```
UINT32 len = 0;
error = ::GetPackageApplicationIds(pir, &len, nullptr, nullptr);
if (error != ERROR_INSUFFICIENT_BUFFER)
    break;

auto buffer = std::make_unique<BYTE[]>(len);
UINT32 count;
error = ::GetPackageApplicationIds(pir, &len, buffer.get(), &count);
if (error != ERROR_SUCCESS)
    break;
```

This is accomplished in two steps: first calling `GetPackageApplicationIds` with a `NULL` pointer for the application ID and a length of zero. This causes the function to fill in the required length. Then, a buffer is constructed with `make_unique` (ensuring it's automatically destroyed when the variable goes out of scope), and a second call is made.

The application IDs returned are stored with a 4-byte length and then the data itself. Since only one application is expected, we can just skip the first 4 bytes and use the rest as the application ID. The last step is to create the instance that implements `IApplicationActivationManager` and call `ActivateApplication` with the application ID:

```
CComPtr<IApplicationActivationManager> mgr;
auto hr = mgr.CoCreateInstance(CLSID_ApplicationActivationManager);
if (FAILED(hr))
    break;

DWORD pid;
hr = mgr->ActivateApplication((PCWSTR)(buffer.get() + sizeof(ULONG_PTR)),
    nullptr, AO_NOERRORUI, &pid);
```

`ActivateApplication` is even kind enough to return the process ID of the created process. Finally, the package information data needs to be freed:

```
::ClosePackageInfo(pir);
```

If you look at the parent of any UWP process, you'll discover it's an *Svchost.exe* process, rather than the direct creator (see figure 3-22). This is because UWP processes are actually launched by the *DCOM Launch* service, hosted in a service host (figure 3-25).

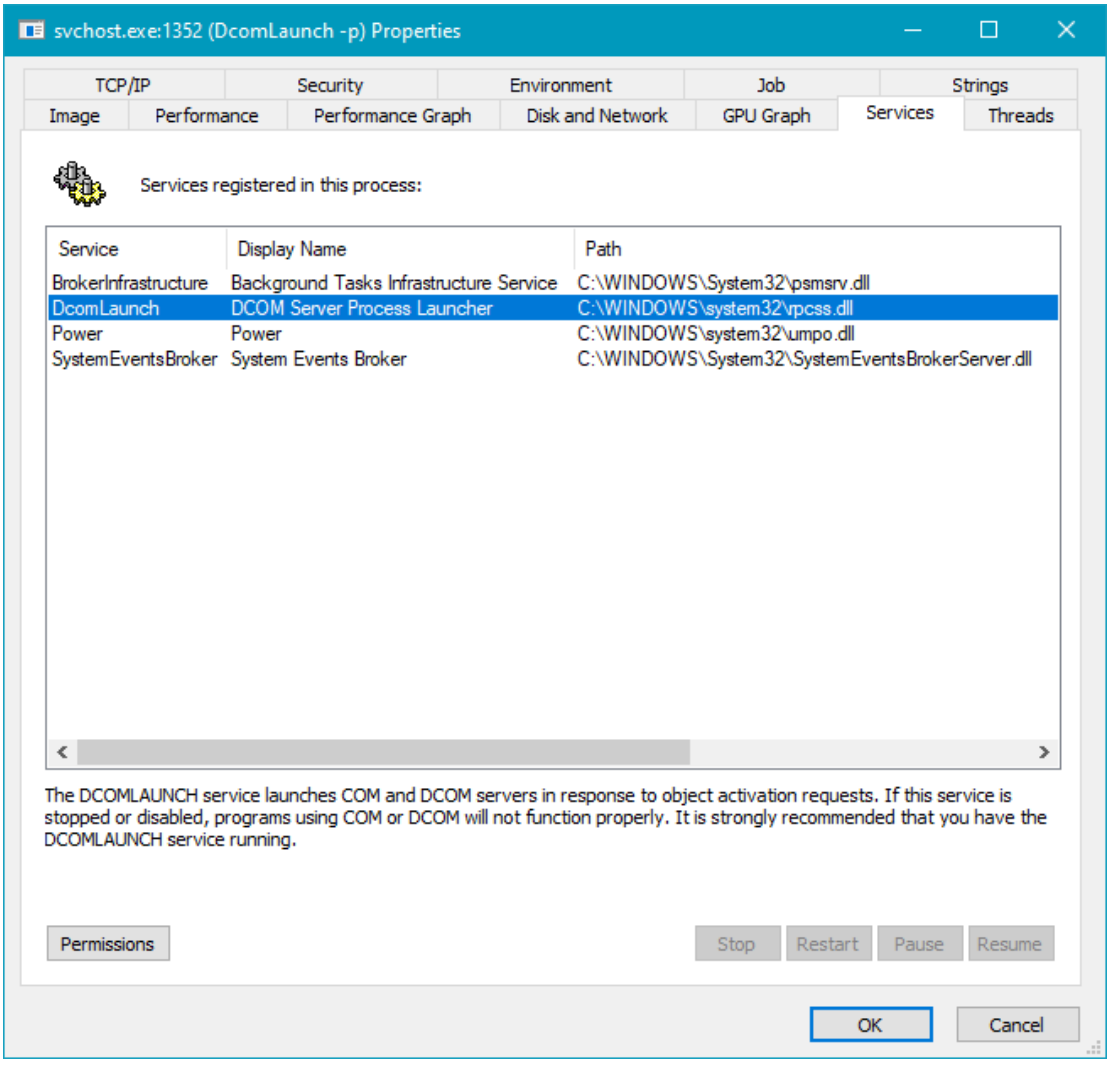


Figure 3-25: The DCOM Launch service

Minimal and Pico Processes

Minimal processes contain just a user mode address space. *Memory Compression* and *Registry* are canonical examples of minimal processes. Minimal processes can only be created by the kernel, and so will not be discussed further in this book.

Pico processes are minimal processes with an added twist: a pico provider, which is a kernel driver that is responsible for translating Linux system calls to equivalent Windows system calls. This is the basis of the *Windows Subsystem for Linux* (WSL), available in Windows 10 version

1607 (and later) and Windows 2016 (and later). Pico processes are beyond the scope of this book, although I may publish a special chapter on WSL in the future.

Process Termination

Most processes will terminate at some point before the system is shutdown. There are several ways a process might exit or terminate. One thing to keep in mind is no matter how a process terminates, the kernel ensures nothing private to the process remains: all private (non-shared) memory is freed and all handles in the process handle table are closed.

A process terminates if any one of the following conditions is met:

1. All the threads in the process exit or terminate.
2. Any thread in the process calls `ExitProcess`.
3. The process is terminated (usually externally but could be because of an unhandled exception) with `TerminateProcess`.

Anyone writing a Windows application usually finds out at some point that the thread executing the main function is "special", typically referred to as the *main thread*. It can be observed that whenever the main function returns, the process exits. This seems to be a scenario not listed in the above reasons for process exit. However, it does, and it's scenario number 2. The C/C++ runtime library invokes `main`/`WinMain` (discussed in the section "Process Creation" earlier in this chapter),
and then does required cleanup such as calling global C++ destructors, C runtime cleanup, etc., and then as its final act eventually calls `ExitProcess`, causing the process to exit.

From the kernel's perspective, all threads in a process are equal, and there is no *main* thread. The kernel destroys a process when all threads within it exit/terminate, as a process without threads is mostly useless. In practice, this scenario can only be achieved in native processes (executables that only depend on *NtDll.dll* and have no C/C++ runtime). In other words, this is unlikely to happen in normal Windows programming.

The `ExitProcess` function is defined like so:

```
void ExitProcess(_In_ UINT exitCode);
```

The calling process is the only one capable of calling `ExitProcess`, and naturally, this function never returns. External processes can attempt to terminate a process with `TerminateProcess` (discussed later). The exit code becomes the process exit code, that can be read by anyone holding a handle to the process with `GetExitCodeProcess` defined like so:

```
BOOL GetExitCodeProcess(
  _In_  HANDLE  hProcess,
  _Out_ LPDWORD lpExitCode);
```

It might seem weird that the exit code is available **after** the process exits, but since a handle to the process is still open, the kernel process structure is still alive and that is where the exit code is stored. What would happen if `GetExitCodeProcess` is called for a live process. You might expect the function to fail, but confusingly it succeeds and returns an exit code called `STILL_ACTIVE` (0x103).

In the kernel, `STILL_ACTIVE` is called `STATUS_PENDING`, indicating in this case that the process is still alive.

> The last sentence means looking at the exit code is not a 100% sure way to check if a process is still alive. The proper way to do this is call `WaitForSingleObject(hProcess, 0)` and check the return value against `WAIT_OBJECT_0`; if equal, the process is dead; Only the kernel management object still remains because there is at least one open handle to the process.

`ExitProcess` shuts down the process in an orderly fashion, performing the following important actions:

1. All other threads in the process terminate.
2. All DLLs in the process get their `DllMain` function called with a reason value of `PROCESS_DLL_DETACH`, indicating the DLL is about to be unloaded and should do its cleanup.
3. Terminates the process and the calling thread (`ExitProcess` never returns).

The third way a process might terminate is because of a call to `TerminateProcess`, defined like so:

```
BOOL TerminateProcess(
  _In_ HANDLE hProcess,
  _In_ UINT   uExitCode);
```

`TerminateProcess` can be called from outside the process, assuming a handle with the access mask `PROCESS_TERMINATE` can be obtained. The function terminates the process then and there and specifies what its exit value would be. The process in question has no say in this.

`TerminateProcess` differs from `ExitProcess` in one important aspect: `DllMain` functions for all DLLs in the target process don't get called, so cannot perform any cleanup action. This may result in loss of functionality or data. For example, if a DLL writes some information to a log file when it's unloaded, it will not get the chance to do so. Clearly, `TerminateProcess` should be used as a last resort.

Task Manager's *End Task* button in the *Details* tab calls `TerminateProcess` if it's able to open a process handle with `PROCESS_TERMINATE` access mask. The *End Task* button in the *Processes* tab is trickier, as for GUI processes it first attempts to ask the process to exit nicely by sending a close message to its main window (`SendMessage` or `PostMessage` function with `WM_CLOSE` as the message type).

Enumerating Processes

In some cases it's beneficial to enumerate existing processes. One possible reason would be to look for a particular process of interest. Another reason may be to create some sort of tool that provides information on existing processes. Well-known tools such as *Task Manager* and *Process Explorer* use process enumeration.

The Windows API provides three documented ways to enumerate processes. We'll examine all of them and then briefly discuss a fourth, semi-documented, option.

Using `EnumProcesses`

The simplest function to use is `EnumProcesses` part of the so-called *Process Status API* (PSAPI), included in `<psapi.h>`:

```
BOOL EnumProcesses(
  _Out_ DWORD *pProcessIds,
  _In_  DWORD cb,
  _Out_ DWORD *pBytesReturned);
```

This function provides the bear minimum - all process IDs. The caller must allocate a large enough buffer to store all PIDs. On return, the function indicates the number of bytes actually stored in the provided buffer. If it's lower than the buffer size, it means the buffer was large enough to contain all PIDs. If it's equal to the buffer size, then the buffer was probably too small, and the caller should make a second call with a larger buffer size until the first condition is met.

The simplest way is to call the function with a (hopefully) large enough buffer:

```
const int MaxCount = 1024;
DWORD pids[MaxCount];
DWORD actualSize;
if(::EnumProcesses(pids, sizeof(pids), &actualSize)) {
    // assume actualSize < sizeof(pids)

    int count = actualSize / sizeof(DWORD);
    for(int i = 0; i < count; i++) {
        // do something with pids[i]
    }
}
```

A more conservative approach needs to allocate the PIDs array dynamically so it can be resized when needed. A relatively simple way to do this is by leveraging the C++ `std::unique_ptr<>` template class. Here is a revised example:

```
int maxCount = 256;
std::unique_ptr<DWORD[]> pids;
int count = 0;

for (;;) {
    pids = std::make_unique<DWORD[]>(maxCount);
    DWORD actualSize;
    if (!::EnumProcesses(pids.get(), maxCount * sizeof(DWORD), &actualSize))
        break;

    count = actualSize / sizeof(DWORD);
    if (count < maxCount)
        break;

    // need to resize
    maxCount *= 2;
}

for (int i = 0; i < count; i++) {
    // do something with pids[i]
}
```

Remember to add `#include <psapi.h>` for the above to compile.

You'll need to `#include <memory>` so that `unique_ptr<>` is available.

> If you would rather use classic C++ or even C, you can of course do that by leveraging the `new` and `delete` operators or the `malloc` and `free` functions. I recommend using the modern C++ approach which is less error prone because the allocated memory is freed automatically when the relevant object is destroyed.

The downside of `EnumProcesses` is its minimal information - just the PID for each process. If anything else about the process is needed (the usual case), another call to `OpenProcess` is required to get a handle to each process of interest and then making the appropriate calls to retrieve information or perform the required operations on the process. Of course, `OpenProcess` can fail, because not every access mask is necessarily possible to obtain for a process.

The following code snippet shows how to get the process image name and its start time after a successful call to `EnumProcesses`:

```
// count is the number of processes
for (int i = 0; i < count; i++) {
    DWORD pid = pids[i];
    HANDLE hProcess = ::OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, FALSE, p\
id);
    if (!hProcess) {
        printf("Failed to open a handle to process %d (error=%d)\n",
            pid, ::GetLastError());
        continue;
    }

    FILETIME start = { 0 }, dummy;
    ::GetProcessTimes(hProcess, &start, &dummy, &dummy, &dummy);
    SYSTEMTIME st;
    ::FileTimeToLocalFileTime(&start, &start);
    ::FileTimeToSystemTime(&start, &st);

    WCHAR exeName[MAX_PATH];
    DWORD size = MAX_PATH;
    DWORD count = ::QueryFullProcessImageName(hProcess, 0, exeName, &size);
    printf("PID: %5d, Start: %d/%d/%d %02d:%02d:%02d Image: %ws\n",
        pid, st.wDay, st.wMonth, st.wYear, st.wHour, st.wMinute, st.wSecond,
        count > 0 ? exeName : L"Unknown");
}
```

`OpenProcess` is used to get a handle to the process with `PROCESS_QUERY_LIMITED_INFORMATION` access mask, which is the lowest one you can ask for. This is enough to get shallow information on a process, such as its start time (`GetProcessTimes`) or image file name (`QueryFullProcessImageName`).

Here is the prototype for `GetProcessTimes`:

```
BOOL GetProcessTimes(
    _In_ HANDLE hProcess,
    _Out_ LPFILETIME lpCreationTime,
    _Out_ LPFILETIME lpExitTime,
    _Out_ LPFILETIME lpKernelTime,
    _Out_ LPFILETIME lpUserTime);
```

`FILETIME` is a 64-bit value that is split into two 32-bit values. The creation and exit time are given in 100 nano-seconds units measured from January 1, 1601, UTC, at midnight. The creation time is converted into a more manageable form using `FileTimeToSystemTime`, which splits the 64-bit value into human-readable pieces (day, month, year, etc.).

The kernel and user times are relative, given in the same 100 nsec units. They are not used in the above code, putting their result in a dummy variable. Note that providing `NULL` for any of the parameters will cause the function to throw an access violation exception.

`QueryFullProcessImageName` allows getting the full executable path of a given process:

```
BOOL QueryFullProcessImageName(
    _In_    HANDLE hProcess,
    _In_    DWORD  dwFlags,
    _Out_   LPTSTR lpExeName,
    _Inout_ PDWORD lpdwSize
);
```

The `dwFlags` parameter is usually zero, but can be `PROCESS_NAME_NATIVE` (1), which returns the path in device form, which is the native Windows way of representing paths (something like *\Device\HarddiskVolume3\MyDir\MyApp.exe*). We'll look at this form more closely in chapter 11. The `lpExeName` parameter is the buffer allocated by the caller, and the last parameter is a pointer to the size of the buffer, in characters. This is an input/output parameter, so it must be initialized to the allocated buffer size, and the function changes it to the actual number of characters written to the buffer.

Running this code with standard user rights produces output like the following:

```
Failed to get a handle to process 0 (error=87)
Failed to get a handle to process 4 (error=5)
Failed to get a handle to process 88 (error=5)
Failed to get a handle to process 152 (error=5)
Failed to get a handle to process 900 (error=5)
Failed to get a handle to process 956 (error=5)
Failed to get a handle to process 1212 (error=5)
...
PID:  9796, Start: 26/7/2019 14:13:40 Image: C:\Windows\System32\sihost.exe
PID:  9840, Start: 26/7/2019 14:13:40 Image: C:\Windows\System32\svchost.exe
Failed to get a handle to process 9864 (error=5)
PID:  9904, Start: 26/7/2019 14:13:40 Image: C:\Windows\System32\svchost.exe
PID:  9936, Start: 26/7/2019 14:13:40 Image: C:\Windows\System32\svchost.exe
PID: 10004, Start: 26/7/2019 14:13:40 Image: C:\Windows\System32\taskhostw.exe
Failed to get a handle to process 10032 (error=5)
PID:  9556, Start: 26/7/2019 14:13:40 Image: C:\Windows\explorer.exe
...
```

Getting a handle to PID 0 (*System Idle Process* in *Task Manager*) fails, as PID 0 is not valid. This is why the error is number 87 (ERROR_INVALID_PARAMETER). Other processes in the low-PID range fail to open as well, with error 5 (ERROR_ACCESS_DENIED).

Running the same code with admin rights by opening an elevated command window, navigating to the output directory and running the executable produces the following output:

```
Failed to get a handle to process 0 (error=87)
PID:     4, Start: 26/7/2019 14:13:20 Image: Unknown
PID:    88, Start: 26/7/2019 14:13:01 Image: Unknown
PID:   152, Start: 26/7/2019 14:13:01 Image: Unknown
PID:   900, Start: 26/7/2019 14:13:20 Image: C:\Windows\System32\smss.exe
Failed to get a handle to process 956 (error=5)
PID:  1212, Start: 26/7/2019 14:13:33 Image: C:\Windows\System32\wininit.exe
Failed to get a handle to process 1220 (error=5)
PID:  1288, Start: 26/7/2019 14:13:33 Image: C:\Windows\System32\services.exe
PID:  1300, Start: 26/7/2019 14:13:33 Image: C:\Windows\System32\LsaIso.exe
PID:  1316, Start: 26/7/2019 14:13:33 Image: C:\Windows\System32\lsass.exe
...
```

There are two things to notice here. First, we do get more processes open successfully than with standard user rights. Second, with some processes we fail to retrieve the image name. This is because they don't have normal executable names. These are some of the "special" ones: 4

(*System*), 88 (*Secure System*), 152 (*Registry*) and further down (not shown) *Memory Compression*. Clearly, `QueryFullProcessImageName` cannot provide the names for these processes.

Even with administrator privileges, we can't seem to open every possible process. This situation can be improved by enabling the *Debug* privilege (which exists by default in the admin token, but is not enabled). Curiously enough, if you run the code directly from Visual Studio (running elevated), you'll get the same effect as if the *Debug* privilege was enabled, because Visual Studio already enabled its *Debug* privilege and because it's the one launching the process, its access token is duplicated for the new process, so that the *Debug* privilege is already enabled.

To make sure the *Debug* privilege is enabled regardless of where the executable is launched from, we can use the following function:

```
bool EnableDebugPrivilege() {
    wil::unique_handle hToken;
    if (!::OpenProcessToken(::GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES,
        hToken.addressof()))
        return false;

    TOKEN_PRIVILEGES tp;
    tp.PrivilegeCount = 1;
    tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
    if (!::LookupPrivilegeValue(nullptr, SE_DEBUG_NAME, &tp.Privileges[0].Luid))
        return false;

    if (!::AdjustTokenPrivileges(hToken.get(), FALSE, &tp, sizeof(tp),
        nullptr, nullptr))
        return false;

    return ::GetLastError() == ERROR_SUCCESS;
}
```

The complete project is named *ProcEnum* in the samples repository.

A thorough explanation of this function is saved for chapter 16 ("Security"). For now, we can simply use it and get better results:

```
Failed to get a handle to process 0 (error=87)
PID:     4, Start: 26/7/2019 14:13:20 Image: Unknown
PID:    88, Start: 26/7/2019 14:13:01 Image: Unknown
PID:   152, Start: 26/7/2019 14:13:01 Image: Unknown
PID:   900, Start: 26/7/2019 14:13:20 Image: C:\Windows\System32\smss.exe
PID:   956, Start: 26/7/2019 14:13:30 Image: C:\Windows\System32\csrss.exe
PID:  1212, Start: 26/7/2019 14:13:33 Image: C:\Windows\System32\wininit.exe
PID:  1220, Start: 26/7/2019 14:13:33 Image: C:\Windows\System32\csrss.exe
PID:  1288, Start: 26/7/2019 14:13:33 Image: C:\Windows\System32\services.exe
PID:  1300, Start: 26/7/2019 14:13:33 Image: C:\Windows\System32\LsaIso.exe
PID:  1316, Start: 26/7/2019 14:13:33 Image: C:\Windows\System32\lsass.exe
PID:  1436, Start: 26/7/2019 14:13:33 Image: C:\Windows\System32\svchost.exe
...
```

Now we can open a handle to each and every process (except PID 0 which of course is not a real process).

We still are unable to get the names of the special processes. The next technique for process enumeration solves this issue.

Using the Toolhelp Functions

The so-called "Toolhelp" functions provide a more convenient way to get basic information on processes, including process "names" for the special processes not based on an executable image. All this is available from a standard user rights process - no need for elevated permissions.

To gain access to these functions, include `<tlhelp32.h>`. The initial function to call is `CreateToolhelp32Snapshot`, which creates a snapshot consisting of an optional combination of processes and threads, and for a specific process - heaps and modules as well. Here is the call to create the snapshot to get information for processes only:

```
HANDLE hSnapshot = ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if (hSnapshot == INVALID_HANDLE_VALUE) {
    // handle error
}
```

The second parameter to `CreateToolhelp32Snapshot` indicates which process is the target of the snapshot in case modules or heaps are requested. For processes and threads this must be zero and all processes/threads are included in the snapshot.

Now process enumeration starts with `Process32First`, and subsequent processes are available by calling `Process32Next` until the latter returns `FALSE` meaning there are no more processes. Both functions accept a `PROCESSENTRY32` structure pointer where the information is returned for each process:

```
typedef struct tagPROCESSENTRY32 {
    DWORD   dwSize;                 // size of structure
    DWORD   cntUsage;               // unused
    DWORD   th32ProcessID;          // PID
    ULONG_PTR th32DefaultHeapID;    // unused
    DWORD   th32ModuleID;           // unused
    DWORD   cntThreads;             // # threads
    DWORD   th32ParentProcessID;    // parent PID
    LONG    pcPriClassBase;         // Base priority
    DWORD   dwFlags;                // unused
    TCHAR   szExeFile[MAX_PATH];    // Path
} PROCESSENTRY32;
```

The first member (`dwSize`) must be set to the size of the structure. As can be seen from the comments, only some of the members are actually used. The following code demonstrates getting all possible information provided by process snapshotting:

```
PROCESSENTRY32 pe;
pe.dwSize = sizeof(pe);

if (!::Process32First(hSnapshot, &pe)) {
    // unlikely - handle error
}

do {
    printf("PID:%6d (PPID:%6d): %ws (Threads=%d) (Priority=%d)\n",
        pe.th32ProcessID, pe.th32ParentProcessID, pe.szExeFile,
        pe.cntThreads, pe.pcPriClassBase);
} while (::Process32Next(hSnapshot, &pe));

::CloseHandle(hSnapshot);
```

The project *ProcList* from this chapter's samples has the complete code. Here is the first few lines of output:

```
PID:     0 (PPID:     0): [System Process] (Threads=12) (Priority=0)
PID:     4 (PPID:     0): System (Threads=359) (Priority=8)
PID:    88 (PPID:     4): Secure System (Threads=0) (Priority=8)
PID:   152 (PPID:     4): Registry (Threads=4) (Priority=8)
PID:   900 (PPID:     4): smss.exe (Threads=2) (Priority=11)
PID:   956 (PPID:   932): csrss.exe (Threads=13) (Priority=13)
PID:  1212 (PPID:   932): wininit.exe (Threads=2) (Priority=13)
PID:  1220 (PPID:  1204): csrss.exe (Threads=27) (Priority=13)
```

Using the WTS Functions

The *Windows Terminal Services* (WTS) functions are used to work in a terminal services (also called *Remote Desktop Services*) environment, where a server may host several remote (and local) sessions simultaneously. That said, the WTS API can be used on a single session machine just the same as for a multi-session machine. There are several interesting functions in this API, but for this section's purposes we'll use its process enumeration functions: `WTSEnumerateProcesses` and `WTSEnumerateProcessesEx`. The WTS API is defined in its own header (not included by default with `<windows.h>`) - `<wtsapi32.h>`. It also requires adding an import library - `wtsapi32.lib` - to link successfully.

`WTSEnumerateProcesses` is defined like so:

```
typedef struct _WTS_PROCESS_INFO {
  DWORD  SessionId;
  DWORD  ProcessId;
  LPTSTR pProcessName;
  PSID   pUserSid;
} WTS_PROCESS_INFO, *PWTS_PROCESS_INFO;

BOOL WTSEnumerateProcesses(
  _In_  HANDLE            hServer,
  _In_  DWORD             Reserved,
  _In_  DWORD             Version,
  _Out_ PWTS_PROCESS_INFO *ppProcessInfo,
  _Out_ DWORD             *pCount);
```

The function can enumerate processes on other machines with the first handle that can be obtained by calling `WTSOpenServer`. To use the local machine, the constant `WTS_CURRENT_SERVER_HANDLE` can be used instead. The `Version` parameter must be set to 1, and the real result is allocated and filled by the function itself, returning a pointer to an array of structures of type `WTS_PROCESS_INFO` for each discovered process. The last parameter returns the number of

processes in the returned array. Since the function allocates the memory, the client application must free it eventually with `WTSFreeMemory`.

The following function, part of the *ProcList2* project, uses `WTLEnumerateProcesses` to show information on all processes in the system:

```
bool EnumerateProcesses1() {
    PWTS_PROCESS_INFO info;
    DWORD count;
    if (!::WTSEnumerateProcesses(WTS_CURRENT_SERVER_HANDLE, 0, 1,
        &info, &count))
        return false;

    for (DWORD i = 0; i < count; i++) {
        auto pi = info + i;
        printf("\nPID: %5d (S: %d) (User: %ws) %ws",
            pi->ProcessId, pi->SessionId,
            (PCWSTR)GetUserNameFromSid(pi->pUserSid), pi->pProcessName);
    }
    ::WTSFreeMemory(info);

    return true;
}
```

The returned information per-process is fairly minimal - process ID, session ID, process name (executable name or a special process name such as *System*) and a *Security Identifier* (SID) of the user running the process. The above function displays all information available, and uses a helper function to turn a SID (which is a binary blob, discussed in chapter 16) into a human-readable name:

```
CString GetUserNameFromSid(PSID sid) {
    if (sid == nullptr)
        return L"";

    WCHAR name[128], domain[64];
    DWORD len = _countof(name);
    DWORD domainLen = _countof(domain);
    SID_NAME_USE use;
    if (!::LookupAccountSid(nullptr, sid, name, &len, domain, &domainLen, &use))
        return L"";
```

```
    return CString(domain) + L"\\" + name;
}
```

To link properly, `wtsapi32.lib` must be added to the linker additional dependencies in *Project / Properties*, or added in source with the appropriate `#pragma`:

```
#pragma comment(lib, "wtsapi32")
```

Running the application that calls `EnumerateProcesses1` with standard user rights shows something like the following:

```
PID:     0 (S: 0) (User: )
PID:     4 (S: 0) (User: ) System
PID:    88 (S: 0) (User: ) Secure System
PID:   152 (S: 0) (User: ) Registry
PID:   812 (S: 0) (User: ) smss.exe
PID:  1004 (S: 0) (User: ) csrss.exe
...
PID:  8904 (S: 1) (User: VOYAGER\Pavel) nvcontainer.exe
PID:  8912 (S: 1) (User: VOYAGER\Pavel) nvcontainer.exe
PID:  8992 (S: 1) (User: VOYAGER\Pavel) sihost.exe
PID:  9040 (S: 1) (User: VOYAGER\Pavel) svchost.exe
PID:  9104 (S: 0) (User: ) PresentationFontCache.exe
...
```

PID 0 (the idle process) has no name as far as the WTS APIs are concerned. As explained earlier, this is not a real process, so any name is synthetic anyway. The SID provided by the API is `NULL` in all non-running-user processes. To get the most out of this function, including the SIDs, we can run it in an elevated command window (or launch Visual Studio elevated and execute from the IDE). The results are better:

```
PID:     0 (S: 0) (User: )
PID:     4 (S: 0) (User: ) System
PID:    88 (S: 0) (User: NT AUTHORITY\SYSTEM) Secure System
PID:   152 (S: 0) (User: NT AUTHORITY\SYSTEM) Registry
PID:   812 (S: 0) (User: NT AUTHORITY\SYSTEM) smss.exe
PID:  1004 (S: 0) (User: NT AUTHORITY\SYSTEM) csrss.exe
...
PID:  1360 (S: 0) (User: Font Driver Host\UMFD-0) fontdrvhost.exe
PID:  1388 (S: 0) (User: NT AUTHORITY\LOCAL SERVICE) WUDFHost.exe
```

```
PID:  1492 (S: 0) (User: NT AUTHORITY\NETWORK SERVICE) svchost.exe
PID:  1540 (S: 0) (User: NT AUTHORITY\SYSTEM) svchost.exe
PID:  1608 (S: 0) (User: NT AUTHORITY\LOCAL SERVICE) WUDFHost.exe
PID:  1684 (S: 1) (User: NT AUTHORITY\SYSTEM) winlogon.exe
PID:  1760 (S: 1) (User: Font Driver Host\UMFD-1) fontdrvhost.exe
...
PID:  8396 (S: 0) (User: NT AUTHORITY\NETWORK SERVICE) svchost.exe
PID:  8904 (S: 1) (User: VOYAGER\Pavel) nvcontainer.exe
PID:  8912 (S: 1) (User: VOYAGER\Pavel) nvcontainer.exe
PID:  8992 (S: 1) (User: VOYAGER\Pavel) sihost.exe
PID:  9040 (S: 1) (User: VOYAGER\Pavel) svchost.exe
...
```

The `WTSEnumerateProcessesEx` API (available from Windows 7) is an extended version of `WTSEnumerateProcesses` that provides more information per process:

```
BOOL WTSEnumerateProcessesEx(
    _In_    HANDLE hServer,
    _Inout_ DWORD  *pLevel,
    _In_    DWORD  SessionID,
    _Out_   PTSTR *pProcessInfo,
    _Out_   DWORD  *pCount);
```

The `pLevel` parameter can be set to 0 or 1. With 1, an array of extended structures is returned with more information for each process:

```
typedef struct _WTS_PROCESS_INFO_EX {
    DWORD         SessionId;
    DWORD         ProcessId;
    LPTSTR        pProcessName;
    PSID          pUserSid;
    DWORD         NumberOfThreads;
    DWORD         HandleCount;
    DWORD         PagefileUsage;
    DWORD         PeakPagefileUsage;
    DWORD         WorkingSetSize;
    DWORD         PeakWorkingSetSize;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER KernelTime;
} WTS_PROCESS_INFO_EX, *PWTS_PROCESS_INFO_EX;
```

The `SessionID` parameter allows enumerating processes in a specific session of interest, but `WTS_ANY_SESSION` can be supplied to indicate all sessions are of interest.
The `ppProcessInfo` is typed as a pointer to a pointer to a string, which makes no sense. It must be a pointer to `PWTS_PROCESS_INFO` or `PWTS_PROCESS_INFO_EX` based on the level value. The following is a revised enumeration function that uses the extended structure:

```
bool EnumerateProcesses2() {
    PWTS_PROCESS_INFO_EX info;
    DWORD count;
    DWORD level = 1;    // extended info
    if (!::WTSEnumerateProcessesEx(WTS_CURRENT_SERVER_HANDLE, &level,
        WTS_ANY_SESSION, (PWSTR*)&info, &count))
        return false;

    for (DWORD i = 0; i < count; i++) {
        auto pi = info + i;
        printf("\nPID: %5d (S: %d) (T: %3d) (H: %4d) (CPU: %ws) (U: %ws) %ws",
            pi->ProcessId, pi->SessionId, pi->NumberOfThreads, pi->HandleCount,
            (PCWSTR)GetCpuTime(pi),
            (PCWSTR)GetUserNameFromSid(pi->pUserSid), pi->pProcessName);
    }
    ::WTSFreeMemoryEx(WTSTypeProcessInfoLevel1, info, count);

    return true;
}
```

The fields related to memory in `WTS_PROCESS_INFO_EX` (`PagefileUsage`, `PeakPagefileUsage`, `WorkingSetSize` and `PeakWorkingSetSize`) are actually buggy because they are 32 bit in size even in 64-bit processes (the structure should have been modified so that `DWORD` is changed to `DWORD_PTR` at the very least), so if one of these counters is above 4 GB, they will show wrong numbers.

Freeing the allocated memory block must be done with a dedicated function (`WTSFreeMemoryEx`), with the appropriate level related enumeration. The code displays some additional data compared to `EnumerateProcesses1` - number of threads in the process, number of handles and the total CPU time used. The `GetCpuTime` helper function returns a string representation of the CPU time:

```
CString GetCpuTime(PWTS_PROCESS_INFO_EX pi) {
    auto totalTime = pi->KernelTime.QuadPart + pi->UserTime.QuadPart;
    return CTimeSpan(totalTime / 10000000LL).Format(L"%D:%H:%M:%S");
}
```

The `KernelTime` and `UserTime` members of `WTS_PROCESS_INFO_EX` are the time the process' threads spent in kernel mode and user mode, respectively. This time is provided in 100-nanosecond units (100 times 10 to the -9th power), which is very common within the Windows API. This means converting to seconds requires dividing by 10 million. The above code uses the `CTimeSpan` ATL helper class that accepts a number of seconds and can provide simple string formatting for the value.

Here is some sample output from running `EnumerateProcesses2` with admin rights:

```
PID:     0 (S: 0) (T:  12) (H:    0) (CPU: 7:16:22:09) (User: )
PID:     4 (S: 0) (T: 365) (H: 27610) (CPU: 0:00:14:39) (User: ) System
PID:    88 (S: 0) (T:   0) (H:    0) (CPU: 0:00:00:00) (User: NT AUTHORITY\SYST\
EM) Secure System
PID:   152 (S: 0) (T:   4) (H:    0) (CPU: 0:00:00:01) (User: NT AUTHORITY\SYST\
EM) Registry
PID:   812 (S: 0) (T:   2) (H:   53) (CPU: 0:00:00:00) (User: NT AUTHORITY\SYST\
EM) smss.exe
PID:  1004 (S: 0) (T:  14) (H:  951) (CPU: 0:00:00:05) (User: NT AUTHORITY\SYST\
EM) csrss.exe
...
```

Using the Native API

The last option available for process (and thread) enumeration is using the native API exposed by *NtDll.dll*. This API is mostly undocumented, and in some cases partially documented. Some of the documentation is part of the *Windows Driver Kit* (WDK), as some kernel APIs are the targets of native functions and so share a prototype. In user mode, Microsoft provides very partial definitions for the native API in the file `<winternl.h>`.

One of the functions is `NtQuerySystemInformation`. This mega-function can return various pieces of information depending on a `SYSTEM_INFORMATION_CLASS` that specifies the type of information requested. Here is the definition from `<winternl.h>`:

```
typedef enum _SYSTEM_INFORMATION_CLASS {
    SystemBasicInformation = 0,
    SystemPerformanceInformation = 2,
    SystemTimeOfDayInformation = 3,
    SystemProcessInformation = 5,
    SystemProcessorPerformanceInformation = 8,
    SystemInterruptInformation = 23,
    SystemExceptionInformation = 33,
    SystemRegistryQuotaInformation = 37,
    SystemLookasideInformation = 45,
    SystemCodeIntegrityInformation = 103,
    SystemPolicyInformation = 134,
} SYSTEM_INFORMATION_CLASS;
```

As can be seen from the enumeration, there are many gaps of undocumented information classes. `SystemProcessInformation` is one value that retrieves data for all processes in the system, with more fields than are available with `WTS_PROCESS_INFO_EX`. It even includes all threads for each process. In fact, *Task Manager* and *Process Explorer* use this to get process information.

Using the above enumeration returns objects of type *SYSTEM_PROCESS_INFORMATION*, which is declared in `<winternl.h>` like so:

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    BYTE 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;
```

Although it provides quite a bit of information, there are many members named "reserved"-something, which are actually undocumented fields, rather than true reserved fields. Can we get the details for these "reserved" fields?

Although these structures and enumerations are undocumented, some have been located in some leaked Windows sources, reverse engineered or are available in the public symbols provided by Microsoft. One application that uses many of these officially undocumented functions and types is *Process Hacker*, a open-source *Process Explorer* clone available on Github. One of its sibling projects is *phnt*, which contains the largest definitions of native APIs, structures, enumerations and definitions (https://github.com/processhacker/phnt) that I know of.

As a quick example, the complete `SYSTEM_PROCESS_INFORMATION` looks something like the following:

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    LARGE_INTEGER WorkingSetPrivateSize;
    ULONG HardFaultCount;
    ULONG NumberOfThreadsHighWatermark;
    ULONGLONG CycleTime; // since WIN7
    LARGE_INTEGER CreateTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER KernelTime;
    UNICODE_STRING ImageName;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    HANDLE InheritedFromUniqueProcessId;
    ULONG HandleCount;
    ULONG SessionId;
    ULONG_PTR UniqueProcessKey;
    SIZE_T PeakVirtualSize;
    SIZE_T VirtualSize;
    ULONG PageFaultCount;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
```

```
    SIZE_T QuotaPeakPagedPoolUsage;
    SIZE_T QuotaPagedPoolUsage;
    SIZE_T QuotaPeakNonPagedPoolUsage;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
    SIZE_T PrivatePageCount;
    LARGE_INTEGER ReadOperationCount;
    LARGE_INTEGER WriteOperationCount;
    LARGE_INTEGER OtherOperationCount;
    LARGE_INTEGER ReadTransferCount;
    LARGE_INTEGER WriteTransferCount;
    LARGE_INTEGER OtherTransferCount;
    SYSTEM_THREAD_INFORMATION Threads[1];
} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;
```

Most of the fields that don't exist in the documented structures can still be obtained by other API functions. For example, process creation time can be retrieved with `GetProcessTimes`, but this does require opening a handle to the process (with access mask `PROCESS_QUERY_LIMITED_INFORMATION` in this case). Clearly, getting much of the information in one stroke is a win, which is why process information tools typically use the native API.

In this book, we won't use the native API unless there is a very good reason to do so. You can explore the code of *Process Hacker* to get a sense of how to use these APIs. In any case, if an official API exists, it's always safer to go with that, rather than to rely on a native undocumented API.

Exercises

1. Write a GUI or console application called *MiniProcExp* - a mini Process Explorer, that uses either the Toolhelp APIs, WTS APIs or the native API to show information about processes. Anything that is not readily available get by opening a proper handle to the process and using the correct function to get the information. (A basic app is provided in the *MinProcExp* project).
2. Extend the previous application by adding operations on processes, such as terminating and changing priority class.
3. Continue extending the application in any way you see fit!

Summary

Processes are the most fundamental building block in Windows. It holds a set of resources such as an address space, that allows threads to execute code using these resources. In the next chapter, we'll look at how processes can be managed as a unit using *Jobs*.

Chapter 4: Jobs

Job objects have been around since Windows 2000, being able to manage one or more processes. Most of their capability revolves around limiting the managed processes in some ways. Their usefulness has grown significantly since Windows 8. On Windows 7 and earlier, a process can be a member of a single job only, while in Windows 8 and later, a process can be associated with multiple jobs.

In this chapter:

Introduction to Jobs

Job objects are visible indirectly in *Process Explorer* if a process is under a job. In that case, a *Job* tab appears in the process' properties (this tab is absent if the process is under no job). Another way to glean at the presence of jobs is to enable the *Jobs* color (brown by default) in *Options / Configure Colors...*. Figure 4-1 shows *Process Explorer* with the Jobs color visible with all other colors removed.

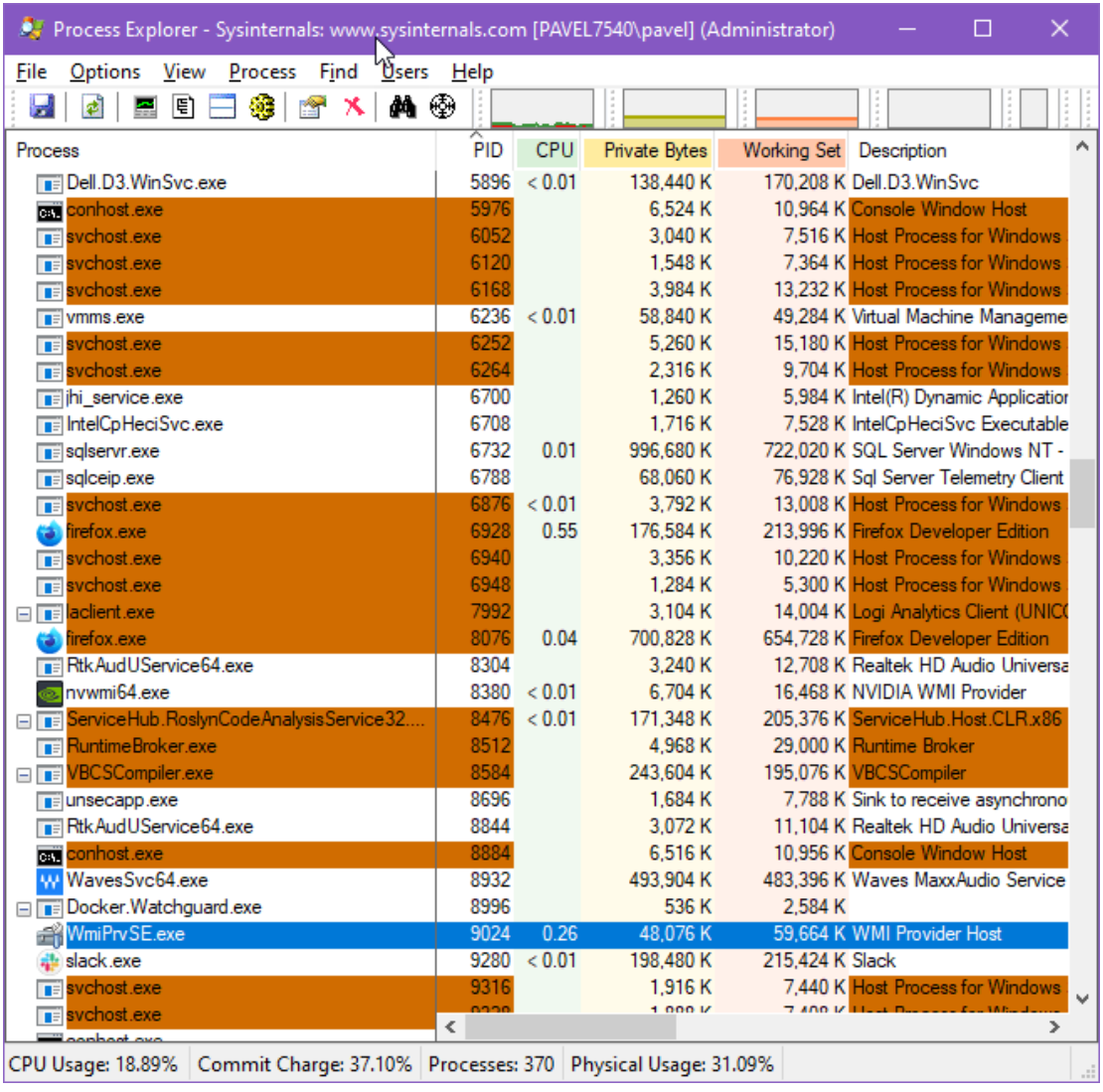


Figure 4-1: Brown Processes are in Jobs

If a process is part of a job, its properties show a *Job* tab with details listing the job's name (if any), the processes that are part of the job, and the limits imposed on the job (if any). Figure 4-2 shows a WMI Worker Process (*wmiprvse.exe*) that is part of a named job. Note the job's limits.

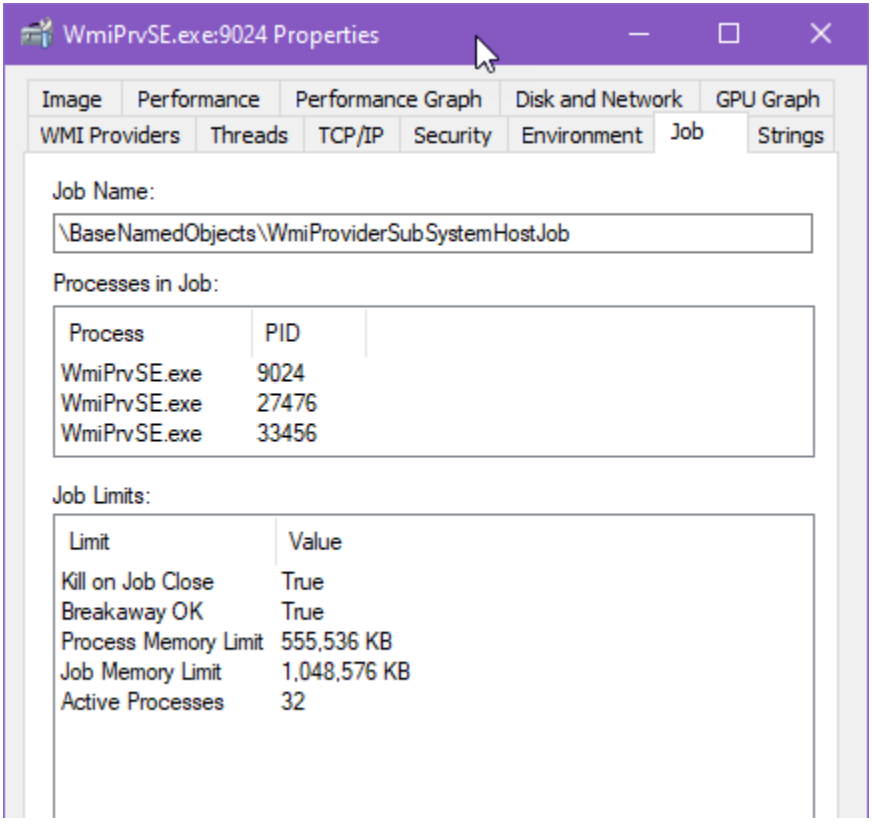


Figure 4-2: Job properties in *Process Explorer*

Once a process is associated with a job, it cannot get out. This makes sense, since if a process could be removed from a job, that would make jobs too weak to be useful in many cases.

Creating Jobs

Creating or opening a job is similar to other create/open functions of other kernel object types. Here is the `CreateJobObject` function:

```
HANDLE CreateJobObject(
  _In_opt_ LPSECURITY_ATTRIBUTES pJobAttributes,
  _In_opt_ LPCTSTR               pName);
```

The first argument is the familiar `SECURITY_ATTRIBUTES` pointer, typically set to `NULL`. The second argument is an optional name to set for the new job object. As with other *create* functions, if a name is provided, and a job with that name exists, then (barring security restrictions), another

handle to the existing job is returned. As usual, calling `GetLastError` can reveal whether the job is an existing one, by returning `ERROR_ALREADY_EXISTS`.

Opening an existing job object by name is possible with `OpenJobObject`, defined like so:

```
HANDLE OpenJobObject(
  _In_ DWORD   dwDesiredAccess,
  _In_ BOOL    bInheritHandle,
  _In_ PCTSTR pName);
```

Most of the arguments should be self-explanatory by now. The first argument specifies the access mask required for the named job object. This access mask is checked against the security descriptor of the job object, returning success only if the security descriptor includes entries that allow the requested permissions. Table 4-1 shows the valid job access masks with a brief description.

Table 4-1: Job Object access masks

Access mask	Description
`JOB_OBJECT_QUERY` (4)	Query operations against the job, such as `QueryInformationJobObject`
`JOB_OBJECT_ASSIGN_PROCESS` (1)	Allows adding processes to the job
`JOB_OBJECT_SET_ATTRIBUTES` (0x10)	Required to call `SetInformationJobObject`
`JOB_OBJECT_TERMINATE` (8)	Required to call `TerminateJobObject`
`JOB_OBJECT_ALL_ACCESS`	All possible access to the job

With a job handle in hand, processes can be associated with a job by calling `AssignProcessToJobObject`:

```
BOOL AssignProcessToJobObject(
  _In_ HANDLE hJob,
  _In_ HANDLE hProcess);
```

The job handle must have the `JOB_OBJECT_ASSIGN_PROCESS` access mask, which is always the case when a new job is created, since the caller has full control of the job. The process handle to assign to the job must have the `PROCESS_SET_QUOTA` and `PROCESS_TERMINATE` access mask bits. This means that some processes can never be part of a job, such as protected processes, since this access mask cannot be obtained for such processes.

The following example opens a process given its ID, and adds it into the provided job:

```
bool AddProcessToJob(HANDLE hJob, DWORD pid) {
    HANDLE hProcess = ::OpenProcess(PROCESS_SET_QUOTA | PROCESS_TERMINATE, FALS\
E, pid);
    if(!hProcess)
        return false;
    BOOL success = ::AssignProcessToJobObject(hJob, hProcess);
    ::CloseHandle(hProcess);
    return success ? true : false;
}
```

Once a process is associated with a job, it cannot break out. If that process creates a child process, the child process is created by default as part of the parent's process job. There are two cases where a child process may be created outside of a job:

- The `CreateProcess` call includes the `CREATE_BREAKAWAY_FROM_JOB` flag and the job allows breaking out of it (by setting the limit flag `JOB_OBJECT_LIMIT_BREAKAWAY_OK` (see the section "Setting Job Limits", later in this chapter)
- The job has the limit flag `JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK`. In this case, any child process is created outside the job without requiring any special flags.

Nested Jobs

Windows 8 introduced the ability to associate a process with more than one job. This makes jobs much more useful than they used to, since if a process you wish to control with a job was already part of a job - there was no way to associate it with another job. A process that is assigned a second job, causes a job hierarchy to be created (if possible). The second job becomes a child of the first job. The basic rules are the following:

- A limit imposed by a parent job affects the job and all child jobs (and hence all processes in those jobs).
- Any limit imposed by a parent job cannot be removed by a child job, but it can be more strict. For example, if a parent job sets a job-wide memory limit of 200 MB, a child job can set (for its processes) a limit of 150 MB, but not 250 MB.

Figure 4-3 shows a hierarchy of jobs created by invoking the following operations (in order):

1. Assign process P1 to J1.
2. Assign process P1 to J2. A hierarchy is formed.
3. Assign process P2 to J2. Process P2 is now affected by jobs J1 and J2.

4. Assign process P3 to J1.

The resulting process/job relationship is depicted in figure 4-3.

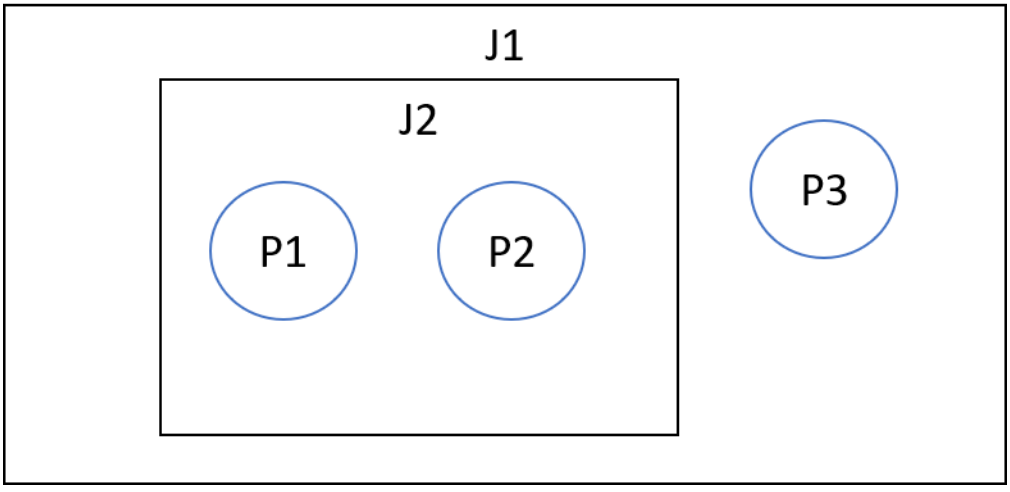


Figure 4-3: Job hierarchy

Viewing job hierarchies is not easy. *Process Explorer*, for example, showing a job's details, includes information for the shown job and all child jobs (if any). For example, viewing the information for job J1 from figure 4-3, three processes would be listed: P1, P2 and P3. Also, since job access is indirect - a *Job* tab is available if a process is under a job - the job shown is the immediate job this process is part of. Any parent jobs are not shown.

The following code creates the hierarchy depicted in figure 4-3.

```
#include <windows.h>
#include <stdio.h>
#include <assert.h>
#include <string>

HANDLE CreateSimpleProcess(PCWSTR name) {
    std::wstring sname(name);
    PROCESS_INFORMATION pi;
    STARTUPINFO si = { sizeof(si) };
    if (!::CreateProcess(nullptr, const_cast<PWSTR>(sname.data()), nullptr, nul\
lptr,
        FALSE, CREATE_BREAKAWAY_FROM_JOB | CREATE_NEW_CONSOLE,
        nullptr, nullptr, &si, &pi))
        return nullptr;
```

```
    ::CloseHandle(pi.hThread);
    return pi.hProcess;
}

HANDLE CreateJobHierarchy() {
    auto hJob1 = ::CreateJobObject(nullptr, L"Job1");
    assert(hJob1);
    auto hProcess1 = CreateSimpleProcess(L"mspaint");

    auto success = ::AssignProcessToJobObject(hJob1, hProcess1);
    assert(success);

    auto hJob2 = ::CreateJobObject(nullptr, L"Job2");
    assert(hJob2);
    success = ::AssignProcessToJobObject(hJob2, hProcess1);
    assert(success);

    auto hProcess2 = CreateSimpleProcess(L"mstsc");
    success = ::AssignProcessToJobObject(hJob2, hProcess2);
    assert(success);

    auto hProcess3 = CreateSimpleProcess(L"cmd");
    success = ::AssignProcessToJobObject(hJob1, hProcess3);
    assert(success);

    // not bothering to close process and job 2 handles

    return hJob1;
}

int main() {
    auto hJob = CreateJobHierarchy();
    printf("Press any key to terminate parent job...\n");
    ::getchar();
    ::TerminateJobObject(hJob, 0);
    return 0;
}
```

> The code is available in the *JobTree* project in this chapter's source code.

The process image names are purposefully different, so they are easier to spot (P1=mspaint, P2=mstsc, P3=cmd). The jobs are named, also for easier identification.

Each process is initially created outside of any job by specifying `CREATE_BREAKAWAY_FROM_JOB` in the `CreateProcess` call. Otherwise, running this application from a process that is already part of a job (such as Visual Studio) would complicate the job hierarchy.

Figure 4-4 shows the job under which *mspaint* is running. Notice it's *Job2*, although *mspaint* is also under *Job1*. Figure 4-5 shows the job under which *cmd* is running, showing the three processes. That's because *cmd* is part of *Job1*, and *Job1* shows all processes including those in child jobs.

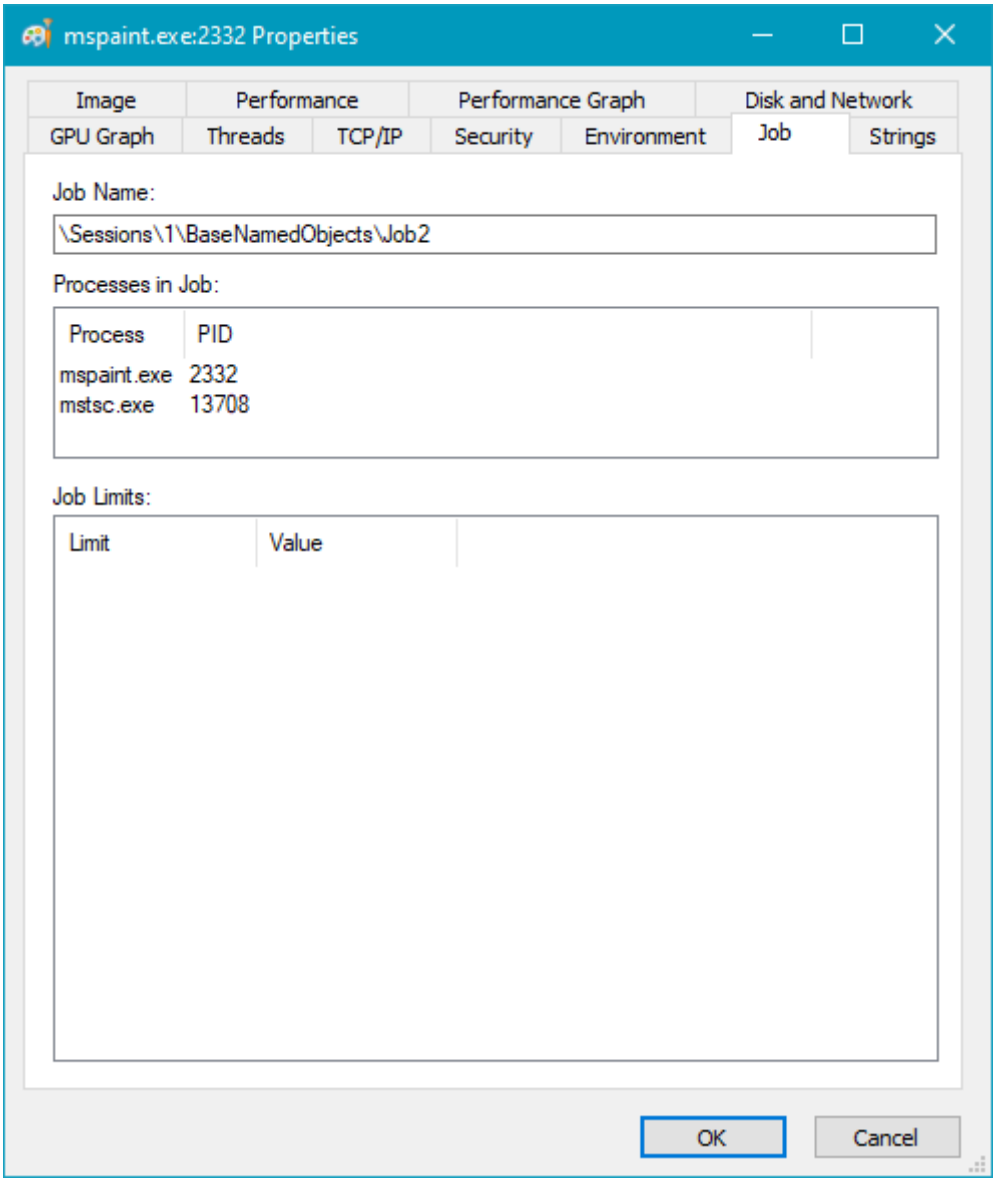


Figure 4-4: *Job2* properties in *Process Explorer*

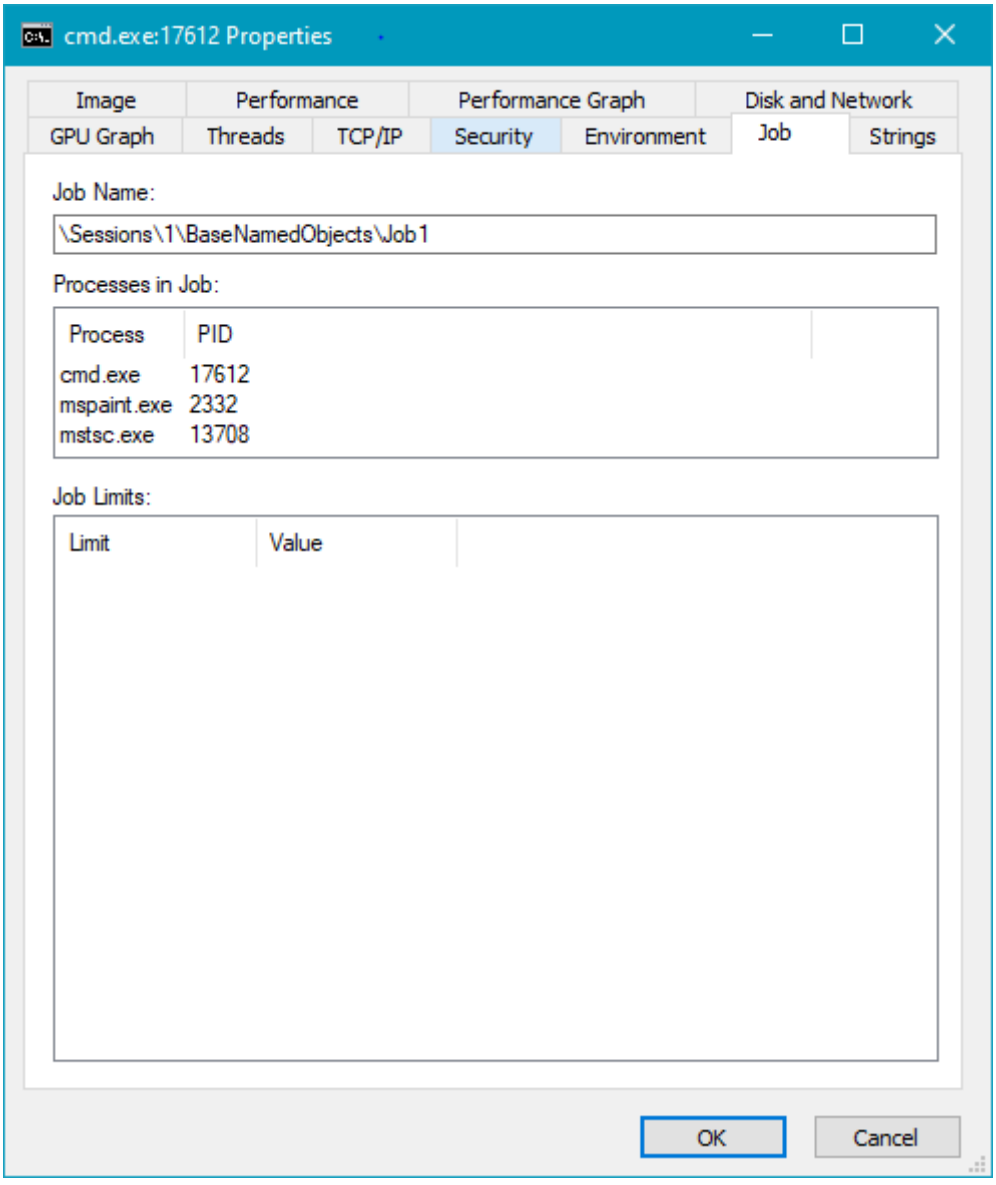


Figure 4-5: *Job1* properties in *Process Explorer*

Viewing job hierarchies is not easy, as there is no documented (or undocumented for that matter) API to enumerate jobs, let alone job hierarchies. A tool I created called *Job Explorer* tries to fill this gap. You can find it in my Github repository at https://github.com/zodiacon/jobexplorer.

Running *Job Explorer* while the *JobTree* application is waiting for a key press shows the screenshot in figure 4-6 when the "All Jobs" tree node is selected and the jobs are sorted by name.

Name	Address	Active Process...	CPU Time	Total Processes
\Sessions\1\BaseNamedObjects\WinlogonAccess	0xFFFFD10C24B51080	0	00:00:00.031	1
\Sessions\1\BaseNamedObjects\ProcessJobTracker19496	0xFFFFD10C282207D0	10	00:10:26.718	10
\Sessions\1\BaseNamedObjects\Job2	0xFFFFD10C1B3937D0	2	00:00:02.000	2
\Sessions\1\BaseNamedObjects\Job1	0xFFFFD10C2348A7D0	4	00:00:02.218	4
\BaseNamedObjects\WmiProviderSubSystemSpecialHostJob	0xFFFFD10C16CD0080	0	00:00:00.000	0
\BaseNamedObjects\WmiProviderSubSystemHostJob	0xFFFFD10C16CCF080	0	00:00:54.343	75
	0xFFFFD10C14672060	13	01:48:55.953	32
	0xFFFFD10C1C8D64...	0	00:41:09.281	98

Figure 4-6: *Job Explorer* All Jobs node

Double-clicking on *Job1* expands the job hierarchy on the left and shows the job details on the right, as shown in figure 4-7.

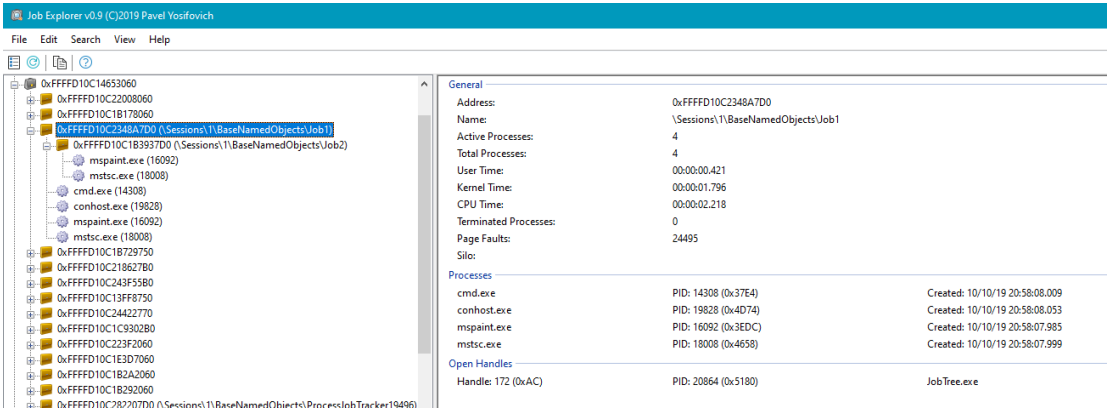


Figure 4-7: *Job Explorer* Job hierarchy view

The job hierarchy is clearly visible in the tree view. Notice that *conhost.exe* process (which is always created when *cmd.exe* is launched) is also part of the same job.

> You may be wondering how *Job Explorer* works. I plan to write a blog post about that.

Querying Job Information

A job object keeps track of some basic job statistics even without any special settings. The primary API to query information about a job object is aptly named `QueryInformationJobObject`:

```
BOOL QueryInformationJobObject(
  _In_opt_  HANDLE             hJob,
  _In_      JOBOBJECTINFOCLASS JobObjectInfoClass,
  _Out_     LPVOID             pJobObjectInfo,
  _In_      DWORD              cbJobObjectInfoLength,
  _Out_opt_ LPDWORD            pReturnLength);
```

The `hJob` parameter is the handle to a job - which must have the `JOB_QUERY` access mask; however, as is hinted by the SAL annotation, a `NULL` value is a valid value, pointing to the job the calling process is under (if any). In this way, a process can query information that may be pertinent to its execution, such as any memory limits imposed by the job. If the job is nested, then the immediate job is the one queried. `JOBOBJECTINFOCLASS` is an enumeration of the various pieces of information that can be queried. For each type of information requested, an appropriately-sized buffer must be supplied in the `pJobObjectInfo` argument to be filled by the function. The last argument is an optional value containing the returned data size in the provided buffer, which is useful for some types of queries that return variable-sized data. Finally, just as with most APIs, the function returns a non-`FALSE` value on success.

Table 4-2 summarizes the (documented) information classes available for jobs in query operations.

Table 4-2: `JOBINFOCLASS` for documented job query operations

Information class (`JobObject*`)	Information structure type	Description
`BasicAccountingInformation` (1)	`JOBOBJECT_BASIC_-ACCOUNTING_INFORMATION`	Basic accounting
`BasicLimitInformation` (2)	`JOBOBJECT_BASIC_LIMIT_-INFORMATION`	Basic limits
`BasicProcessIdList` (3)	`JOBOBJECT_BASIC_PROCESS_-ID_LIST`	List of process IDs in the job
`BasicUIRestrictions` (4)	`JOBOBJECT_BASIC_UI_-RESTRICTIONS`	User interface limits
`EndOfJobTimeInformation` (6)	`JOBOBJECT_END_OF_JOB_-TIME_INFORMATION`	What happens when the end of job time limit is reached
`BasicAndIoAccountingInformation` (8)	`JOBOBJECT_BASIC_AND_IO_-ACCOUNTING_INFORMATION`	Basic and I/O accounting
`ExtendedLimitInformation` (9)	`JOBOBJECT_EXTENDED_LIMIT_-INFORMATION`	Extended limits
`GroupInformation` (11)	`USHORT` array	(Windows 7+) Processor groups for the job (see chapter 6)

Table 4-2: `JOBINFOCLASS` for documented job query operations

Information class (`JobObject*`)	Information structure type	Description
`NotificationLimitInformation` (12)	`JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION`	(Windows 8+) Notification limits
`LimitViolationInformation` (13)	`JOBOBJECT_LIMIT_VIOLATION_INFORMATION`	(Windows 8+) Information on limits violations
`GroupInformationEx` (14)	`GROUP_AFFINITY` array	(Windows 8+) Processor group affinity
`CpuRateControlInformation` (15)	`JOBOBJECT_CPU_RATE_CONTROL_INFORMATION`	(Windows 8+) CPU rate limit information
`NetRateControlInformation` (32)	`JOBOBJECT_NET_RATE_CONTROL_INFORMATION`	(Windows 10+) Network rate limit information
`NotificationLimitInformation2` (33)	`JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION_2`	(Windows 10+) Extended limit information
`LimitViolationInformation2` (34)	`JOBOBJECT_LIMIT_VIOLATION_INFORMATION_2`	(Windows 10+) Extended limit violation information

Job Accounting Information

As mentioned earlier, a job keeps track of some pieces of information, regardless of any limits imposed on the job. The basic accounting information is available with the `JobObjectBasicAccountingInformation` enumeration and the `JOBOBJECT_BASIC_ACCOUNTING_INFORMATION` structure defined like so:

```
typedef struct _JOBOBJECT_BASIC_ACCOUNTING_INFORMATION {
  LARGE_INTEGER TotalUserTime;            // total user mode CPU time
  LARGE_INTEGER TotalKernelTime;          // total kernel mode CPU time
  LARGE_INTEGER ThisPeriodTotalUserTime;      // same counters as above
  LARGE_INTEGER ThisPeriodTotalKernelTime;    // for a "period"
  DWORD TotalPageFaultCount;            // page fault count
  DWORD TotalProcesses;                 // total processes ever existed in the \
job
  DWORD ActiveProcesses;                // live processes in the job
  DWORD TotalTerminatedProcesses;       // processes terminated because of limi\
t violation
} JOBOBJECT_BASIC_ACCOUNTING_INFORMATION, *PJOBOBJECT_BASIC_ACCOUNTING_INFORMAT\
ION;
```

The various times are provided as LARGE_INTEGER structures, each holding a 64-bit value in 100 nanosecond units. The "this period" prefix reports times since the recent per-job user/kernel time limits that have been set (if any). These values are zeroed when the job is created and when a new per-job time limit is set.

The following code snippet shows how to make a query call on a job object for basic accounting information:

```
// assume hJob is a job handle
JOBOBJECT_BASIC_ACCOUNTING_INFORMATION info;
BOOL success = QueryInformationJobObject(hJob, JobObjectBasicAccountingInformat\
ion, &info, sizeof(info), nullptr);
```

Similarly, using JobObjectBasicAndIoAccountingInformation and JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION provides extended accounting information for a job, that includes I/O operations count and sizes. This extended structure includes two structures, one of which is JOBOBJECT_BASIC_ACCOUNTING_INFORMATION:

```
typedef struct _IO_COUNTERS {
    ULONGLONG ReadOperationCount;
    ULONGLONG WriteOperationCount;
    ULONGLONG OtherOperationCount;
    ULONGLONG ReadTransferCount;
    ULONGLONG WriteTransferCount;
    ULONGLONG OtherTransferCount;
} IO_COUNTERS, *PIO_COUNTERS;

typedef struct JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION {
    JOBOBJECT_BASIC_ACCOUNTING_INFORMATION BasicInfo;
    IO_COUNTERS                            IoInfo;
} JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION, *PJOBOBJECT_BASIC_AND_IO_ACCOU\
NTING_INFORMATION;
```

Read and write operations refer to ReadFile and WriteFile (and similar) APIs, which we look at later in this book. The "other" operations refers to usage of the DeviceIoControl API, that is issued in non-read/write operations, typically targeting devices rather than file system files.

The *JobMon* project, part of this chapter's source code includes many of the features of Jobs that we discuss in this chapter. Running it shows the window in figure 4-8.

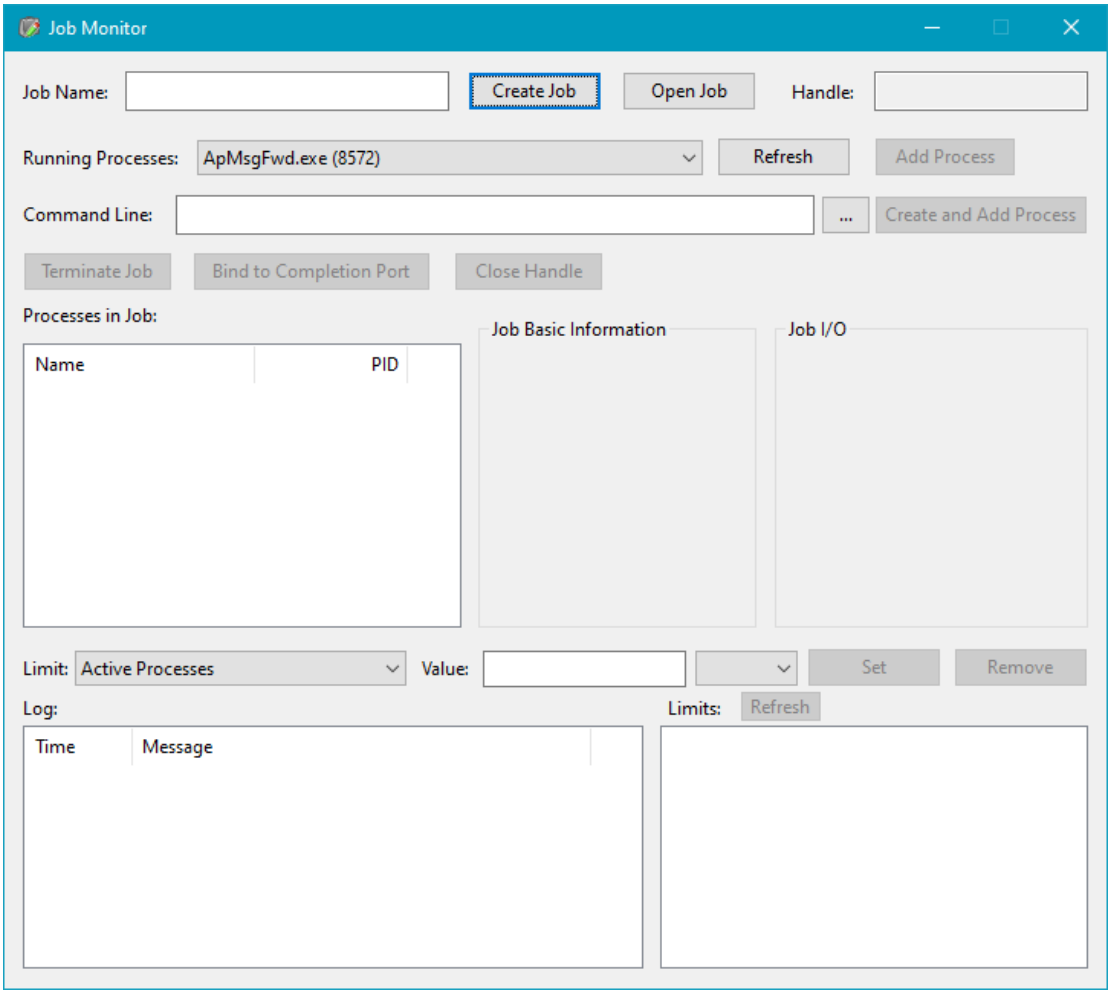


Figure 4-8: *Job Monitor* initial window

Click on the *Create Job* button to create an empty job. You can set a name for job before creating the job. The job is created with zero processes and shows the basic and I/O information (figure 4-9).

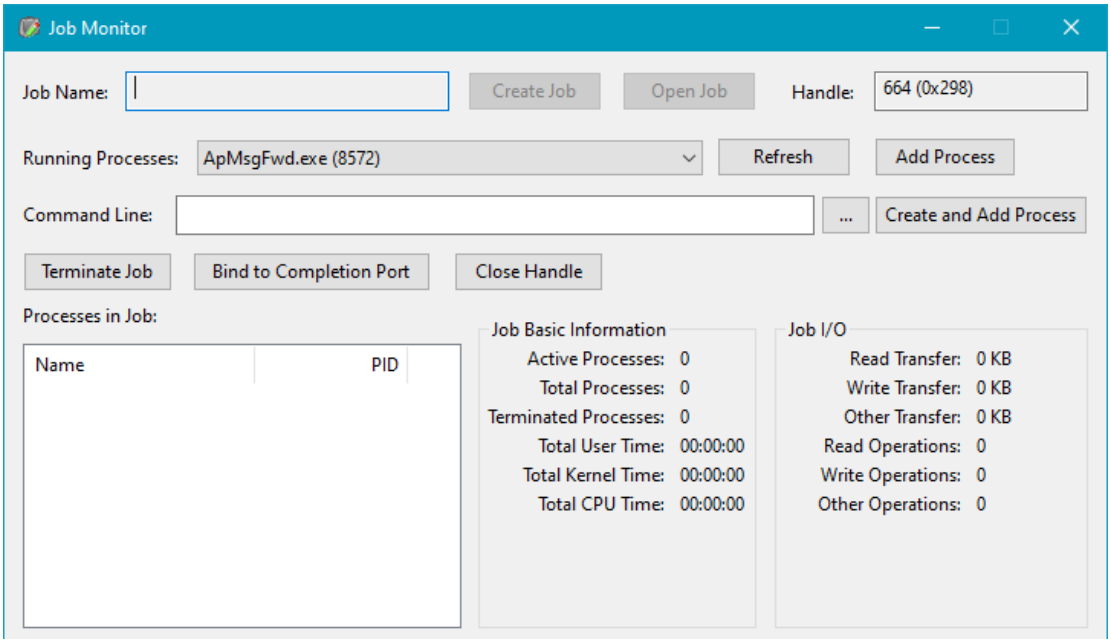


Figure 4-9: *Job Monitor* with a newly created job

To see job accounting in action, download the *CPUStres.exe* tool from https://github.com/zodiacon/AllTools. Click the three dots button to browse for *CpuStres.exe*. Then click *Create and Add Process* button several times to add instances of *CPUStres* to the job (figure 4-10). Notice the accounting information is no longer zero.

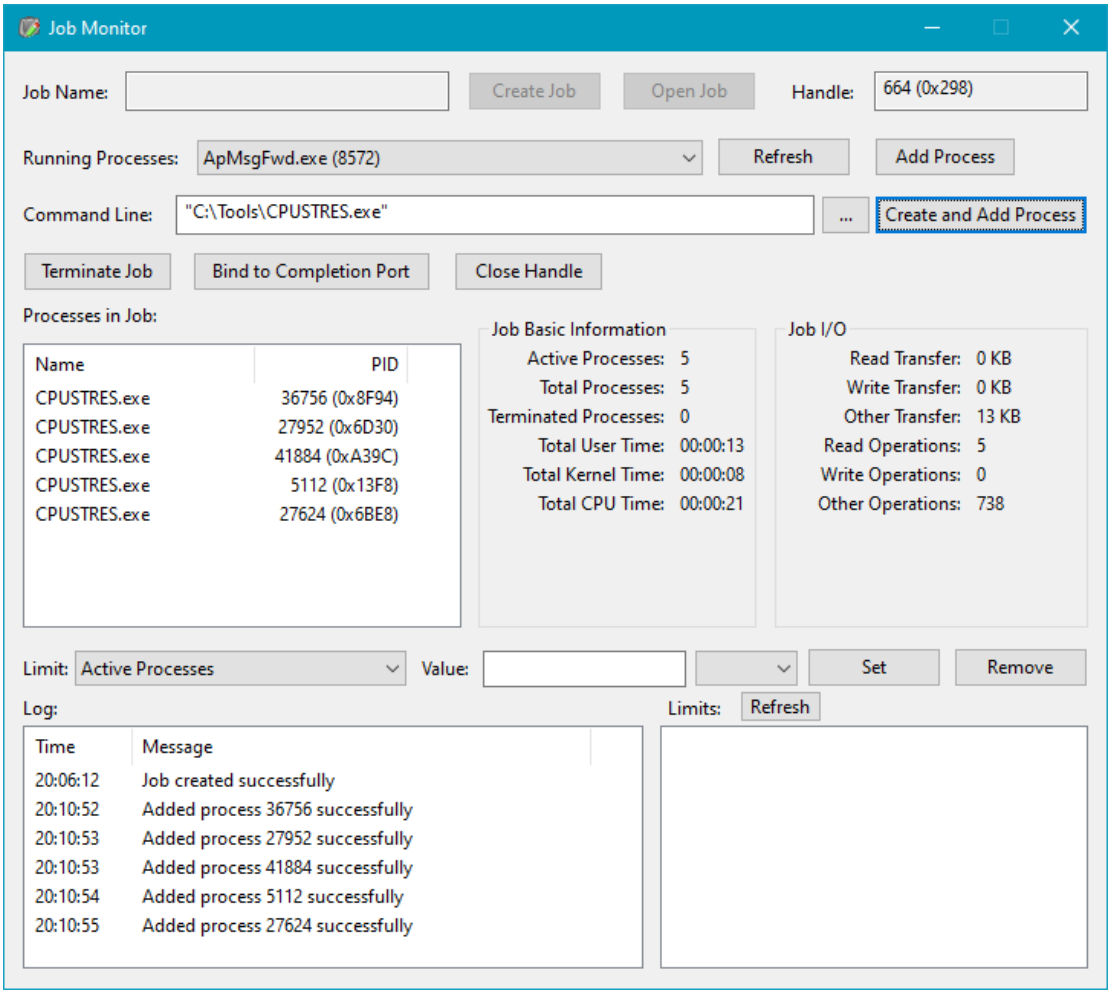


Figure 4-10: *Job Monitor* with a several *CPUStres* processes

CPUStres is a CPU-eating utility, which is used more in the next chapter. The display is updated roughly every 1.5 seconds. You can add more processes to the job (either *CPUStres* or another image), and close processes. Figure 4-11 shows *Job Monitor* after more *CPUStres* are added with some closed.

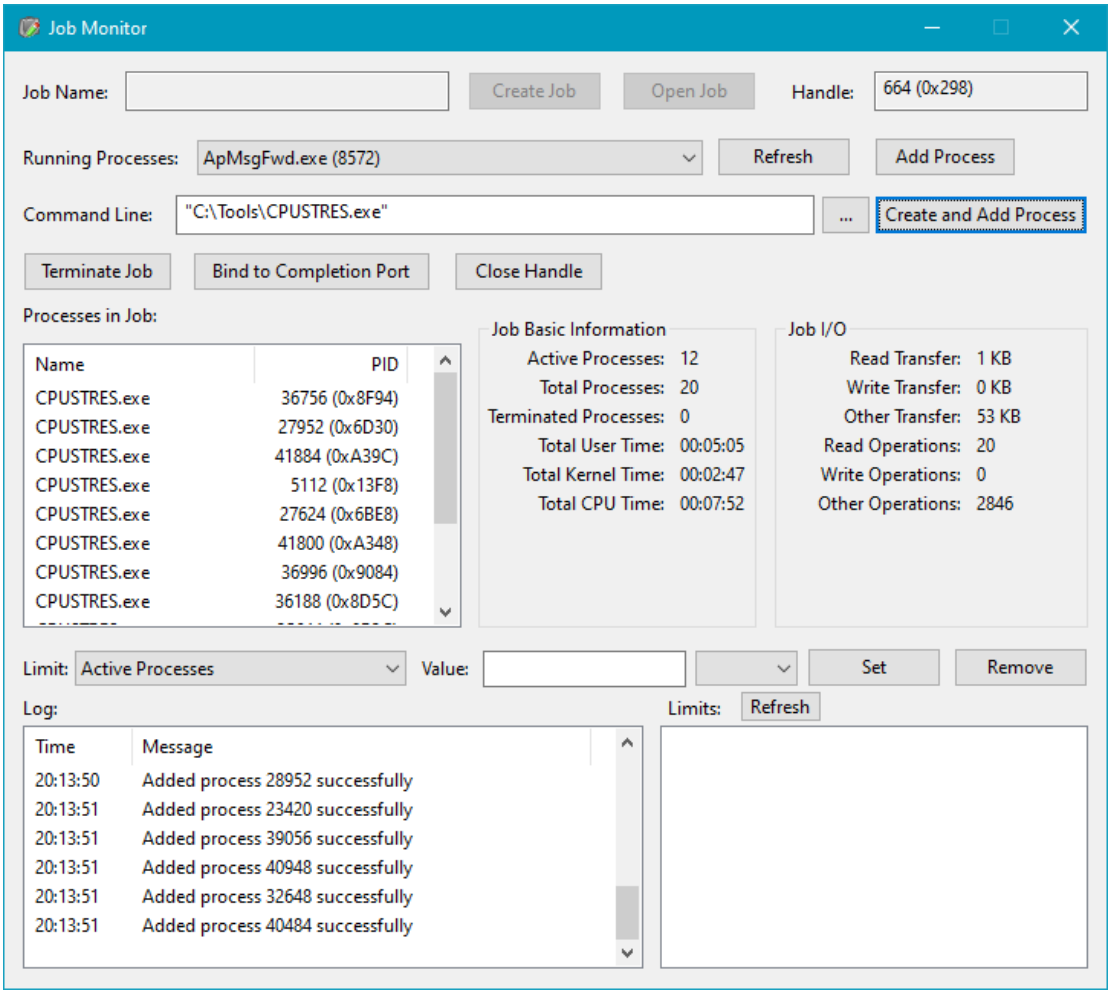


Figure 4-11: *Job Monitor* with more processes

You can click *Terminate Job* to terminate all processes in the job in one stroke. This is achieved by calling `TerminateJobObject`:

```
BOOL TerminateJobObject(
  _In_ HANDLE hJob,
  _In_ UINT   uExitCode);
```

`TerminateJobObject` behaves as if every process active in the job is terminated with `TerminateProcess` where `uExitCode` is the exit code of all processes in the job.
At this point, new processes may be added to the job, with accounting information updated normally.

Querying for Job Process List

The list of active (live) processes in a job can be retrieved by calling `QueryInformationJobObject` with the `JobObjectBasicProcessIdList` information class, returning an array of process IDs in the `JOBOBJECT_BASIC_PROCESS_ID_LIST` structure:

```
typedef struct _JOBOBJECT_BASIC_PROCESS_ID_LIST {
    DWORD NumberOfAssignedProcesses;
    DWORD NumberOfProcessIdsInList;
    ULONG_PTR ProcessIdList[1];
} JOBOBJECT_BASIC_PROCESS_ID_LIST, *PJOBOBJECT_BASIC_PROCESS_ID_LIST;
```

The structure has a variable size because of the array of process IDs. This means a fixed size array should be large enough to include all process IDs and hope for the best. Alternatively, a dynamically-allocated buffer can be used and its size adjusted if it's not large enough.
The following example shows how to retrieve the list of active processes while allocating large enough buffer as required.

```
#include <vector>
#include <memory>

std::vector<DWORD> GetJobProcessList(HANDLE hJob) {
    auto size = 256;
    std::vector<DWORD> pids;
    while (true) {
        auto buffer = std::make_unique<BYTE[]>(size);
        auto ok = ::QueryInformationJobObject(hJob, JobObjectBasicProcessIdList\
,
            buffer.get(), size, nullptr);
        if (!ok && ::GetLastError() == ERROR_MORE_DATA) {
            // buffer too small - resize and try again
            size *= 2;
            continue;
        }
        if (!ok)
            break;

        auto info = reinterpret_cast<JOBOBJECT_BASIC_PROCESS_ID_LIST*>(buffer.g\
et());
        pids.reserve(info->NumberOfAssignedProcesses);
        for (DWORD i = 0; i < info->NumberOfAssignedProcesses; i++)
```

```
            pids.push_back((DWORD)info->ProcessIdList[i]);
        break;
    }
    return pids;
}
```

The code uses some C++ constructs to simplify memory management. The function itself returns a `std::vector<DWORD>` holding the process IDs in the job. In the most general case the number of processes is not known in advance, so the function allocates a buffer with `std::make_unique<BYTE>[]`, which allocates a byte array with the given number of elements (`size`). The `unique_ptr`'s destructor frees the buffer when it goes out of scope.

Next, `QueryInformationJobObject` is called with the allocated byte buffer. If it returns `FALSE` and `GetLastError` returns `ERROR_MORE_DATA`, it means the allocated buffer is too small, so the function doubles `size` and tries again.

Once the buffer is large enough, the pointer is cast to `JOBOBJECT_BASIC_PROCESS_ID_LIST*`, and the process IDs can be retrieved and placed into the `std::vector`. Curiously enough, the process IDs returned in this structure are typed as `ULONG_PTR`, which is means each one is 64 bit in a 64 bit process. This is unusual, as process IDs are normally 32 bit values (`DWORD`). This is why we cannot simply copy the entire array in one stroke to the vector (unless the vector is changed to hold `ULONG_PTR`s).

> You may be curious why the process IDs are typed as `ULONG_PTR`. This is one of the very few cases in the Windows API where this occurs. Within the kernel, process (and thread) IDs are generated using a private handle table for just this purpose. And since handles on 64 bit systems are 64 bit values, it may be "naturally" easy to use them as is. Still, since handle tables are limited to about 16 million handles, 64 bit values are not currently needed (and not used for process IDs outside the kernel).

Setting Job Limits

The primary purpose of a job is to place limits on its processes. The function to use is the opposite of `QueryInformationJobObject` - `SetInformationJobObject` defined like so:

```
BOOL SetInformationJobObject(
  _In_ HANDLE             hJob,
  _In_ JOBOBJECTINFOCLASS JobObjectInfoClass,
  _In_ PVOID              pJobObjectInfo,
  _In_ DWORD              cbJobObjectInfoLength);
```

The arguments should be self-explanatory at this point. The job handle must have the JOB_OBJECT_SET_ATTRIBUTES access mask, and cannot be NULL. Table 4-3 summarizes the (documented) information classes that can be used with SetInformationJobObject.

Table 4-3: JOBINFOCLASS for documented job set operations

Information class (JobObject-)	Information structure	Description
BasicLimitInformation (2)	JOBOBJECT_BASIC_LIMIT_INFORMATION	Basic limits
BasicUIRestrictions (4)	JOBOBJECT_BASIC_UI_RESTRICTIONS	User interface limits
EndOfJobTimeInformation (6)	JOBOBJECT_END_OF_JOB_TIME_INFORMATION	What happens when the end of job time limit is reached
AssociateCompletionPortInformation (7)	JOBOBJECT_ASSOCIATE_COMPLETION_PORT	Associate a completion port with the job
ExtendedLimitInformation (9)	JOBOBJECT_EXTENDED_LIMIT_INFORMATION	Extended limits
GroupInformation (11)	USHORT array	(Windows 7+) Processor groups for the job (see chapter 6)
GroupInformationEx (14)	GROUP_AFFINITY array	(Windows 8+) Processor groups and affinities for the job (see chapter 6)
NotificationLimitInformation (12)	JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION	(Windows 8+) Notification limits
LimitViolationInformation (13)	JOBOBJECT_LIMIT_VIOLATION_INFORMATION	(Windows 8+) Information on limits violations
GroupInformationEx (14)	GROUP_AFFINITY array	(Windows 8+) Processor group affinity
CpuRateControlInformation (15)	JOBOBJECT_CPU_RATE_CONTROL_INFORMATION	(Windows 8+) CPU rate limit information
NetRateControlInformation (32)	JOBOBJECT_NET_RATE_CONTROL_INFORMATION	(Windows 10+) Network rate limit information

Table 4-3: JOBINFOCLASS for documented job set operations

Information class (JobObject-)	Information structure	Description
NotificationLimitInformation2 (33)	JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION_2	(Windows 10+) Extended limit information
LimitViolationInformation2 (34)	JOBOBJECT_LIMIT_VIOLATION_INFORMATION_2	(Windows 10+) Extended limit violation information

The most "fundamental" limits are specified with JobObjectBasicLimitInformation and JOBOBJECT_BASIC_LIMIT_INFORMATION, while extended limits are set with JobObjectExtendedLimitInformation and JOBOBJECT_EXTENDED_LIMIT_INFORMATION. These structures are defined like so:

```
typedef struct _JOBOBJECT_BASIC_LIMIT_INFORMATION {
    LARGE_INTEGER PerProcessUserTimeLimit;
    LARGE_INTEGER PerJobUserTimeLimit;
    DWORD LimitFlags;
    SIZE_T MinimumWorkingSetSize;
    SIZE_T MaximumWorkingSetSize;
    DWORD ActiveProcessLimit;
    ULONG_PTR Affinity;
    DWORD PriorityClass;
    DWORD SchedulingClass;
} JOBOBJECT_BASIC_LIMIT_INFORMATION, *PJOBOBJECT_BASIC_LIMIT_INFORMATION;

typedef struct _JOBOBJECT_EXTENDED_LIMIT_INFORMATION {
    JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimitInformation;
    IO_COUNTERS                       IoInfo;
    SIZE_T                            ProcessMemoryLimit;
    SIZE_T                            JobMemoryLimit;
    SIZE_T                            PeakProcessMemoryUsed;
    SIZE_T                            PeakJobMemoryUsed;
} JOBOBJECT_EXTENDED_LIMIT_INFORMATION, *PJOBOBJECT_EXTENDED_LIMIT_INFORMATION;
```

The various limits that can be set depend on the LimitFlags member in JOBOBJECT_BASIC_LIMIT_INFORMATION (whether standalone or part of JOBOBJECT_EXTENDED_LIMIT_INFORMATION). Some flags have no associated member, as these flags themselves are enough. Others make the corresponding member's value used by SetInformationJobObject. Table 4-4 summarizes the flags that have no corresponding member. Table 4-5 summarizes the flags that have associated member(s).

Table 4-4: `LimitFlags` with no associated member

Limit flag (`JOB_OBJECT_LIMIT_*`)	Description
`DIE_ON_UNHANDLED_EXCEPTION` (0x400)	Prevents processes in the job from showing a dialog box in case of an unhandled exception
`PRESERVE_JOB_TIME` (0x40)	This flag preserves any previously set job limits, so that the caller can just make further changes. This flag cannot be used with `JOB_OBJECT_LIMIT_JOB_TIME` (and vice versa)
`BREAKAWAY_OK` (0x800)	Allows processes created from a process in the job to be created outside the job if `CREATE_BREAKAWAY_FROM_JOB` flag is specified in the call to `CreateProcess`. If the process is part of a job hierarchy, the new process breaks from this job and all parent jobs that have this flag set
`SILENT_BREAKAWAY_OK` (0x1000)	Allows processes created from a process in the job to be created outside of the job without any special flag to `CreateProcess`
`KILL_ON_JOB_CLOSE` (0x2000)	Terminate all processes in the job when the last job handle is closed

Table 4-5: `LimitFlags` with associated member(s)

Limit flag (`JOB_OBJECT_LIMIT_*`)	Associated member (B/E)	Description
`WORKINGSET` (1)	`MinimumWorkingSetSize`, `MaximumWorkingSetSize` (B)	Limits the per-process working set (RAM). If the process needs to use more RAM, it will page towards itself
`PROCESS_TIME` (2)	`PerProcessUserTimeLimit` (B)	Limits the user mode execution time of each process in the job (in 100 nano-sec units)
`JOB_TIME` (4)	`PerJobUserTimeLimit` (B)	Limits the job-wide user-mode CPU time. if the process uses more CPU time, it is terminated
`ACTIVE_PROCESS` (8)	`ActiveProcessLimit` (B)	Limits the active (live) processes in the job. Newly created processes that violate this limit are automatically terminated
`AFFINITY` (0x10)	`Affinity` (B)	Sets the CPU affinity for all processes in the job (see chapter 6 for more on affinity)

Table 4-5: `LimitFlags` with associated member(s)

Limit flag (`JOB_OBJECT_LIMIT_*`)	Associated member (B/E)	Description
`PRIORITY_CLASS` (0x20)	`PriorityClass` (B)	Limits the processes in the job to use the same priority class (see chapter 6)
`SCHEDULING_CLASS` (0x80)	`SchedulingClass` (B)	Limits the scheduling class for all processes in the job. The values are 0 to 9, where 9 is the highest. The default is 5 (see chapter 6 for more information)
`PROCESS_MEMORY` (0x100)	`JobMemoryLimit` (E)	Limits the per-process committed memory
`JOB_MEMORY` (0x200)	`JobMemoryLimit` (E)	Limits the job-wide committed memory
`SUBSET_AFFINITY` (0x4000)	`Affinity` (B)	(Windows 7+) Allows processes to use a subset of the affinity specified. The `JOB_OBJECT_LIMIT_AFFINITY` flag is required as well

All the flags in table 4-4 must be used with the extended limits structure (`JOBOBJECT_EXTENDED_LIMIT_INFORMATION`), where the flags themselves are specified with the nested `JOBOBJECT_BASIC_LIMIT_INFORMATION` structures's `LimitFlags` member. In table 4-5, *B/E* indicates whether this limit is specified with the basic (B) or the extended (E) structure.

The following code sets a priority class of *Below Normal* for the given job:

```
bool SetJobPriorityClass(HANDLE hJob) {
    JOBOBJECT_BASIC_LIMIT_INFORMATION info;
    info.LimitFlags = JOB_OBJECT_LIMIT_PRIORITY_CLASS;
    info.PriorityClass = BELOW_NORMAL_PRIORITY_CLASS;
    return ::SetInformationJobObject(hJob, JobObjectBasicLimitInformation,
        &info, sizeof(info));
}
```

We can test this type of functionality with the *Job Monitor* application. Open *Job Monitor*, create a new job, and add a process to the job (*Notepad* in figure 4-12).

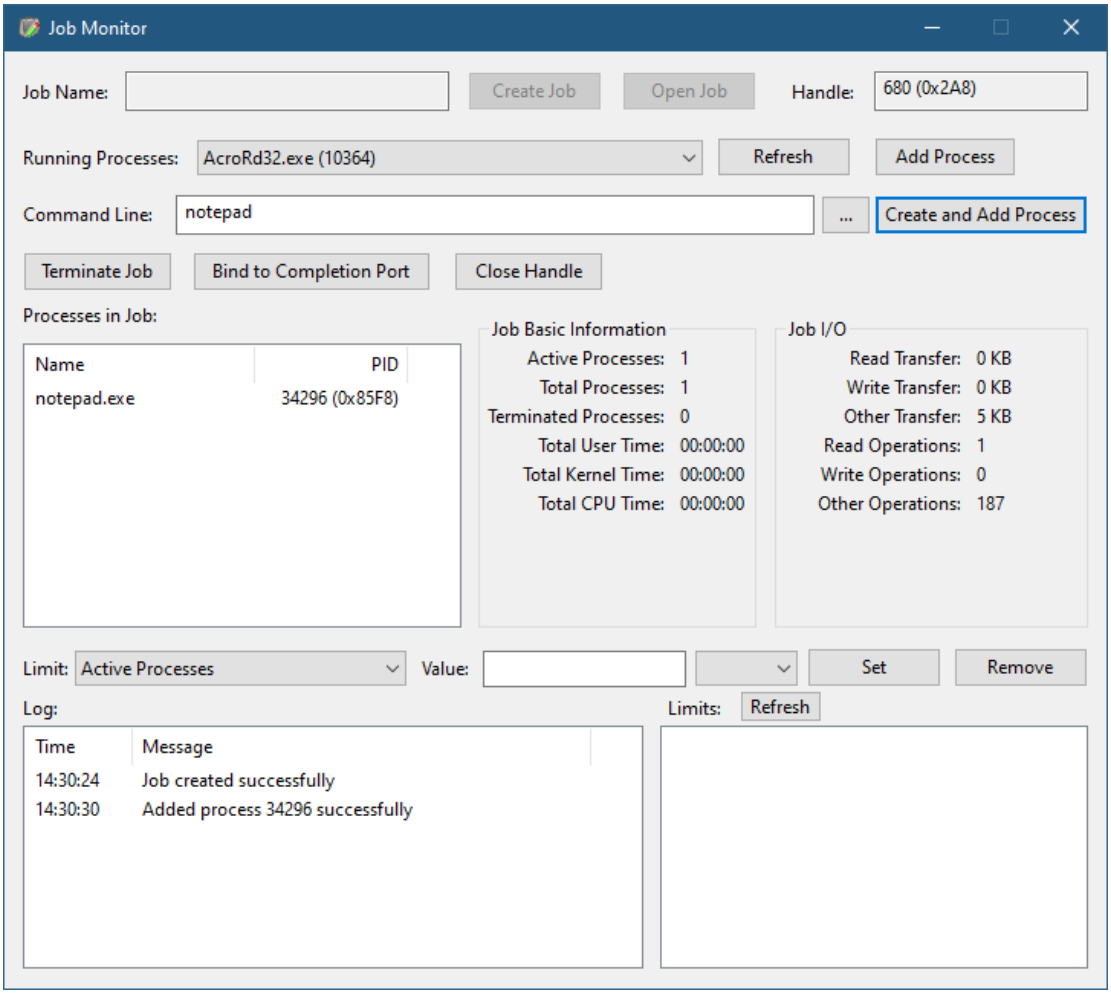


Figure 4-12: *Job Monitor* with *Notepad* added

If you examine the *Base Priority* column in *Task Manager* for this *Notepad* process, you should see the value *Normal* (figure 4-13).

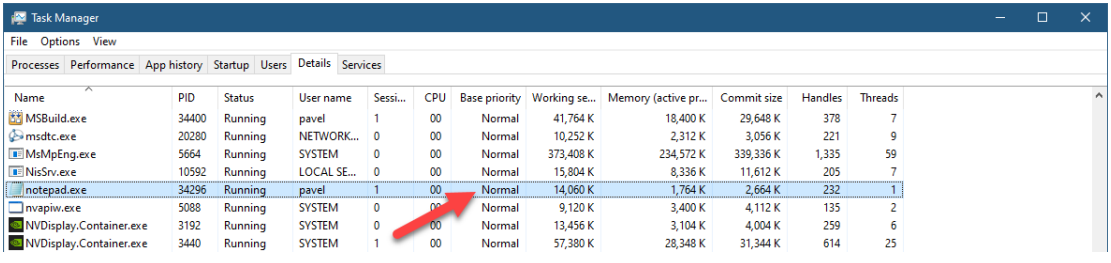


Figure 4-13: Base Priority of *Notepad* in *Task Manager*

> The exact meaning and effect of Base Priority (Priority Class) is discussed in chapter 6.

Now return to *JOb Monitor*, select the *Priority Class* limit and set its value to *Below Normal* and click *Set* (figure 4-14).

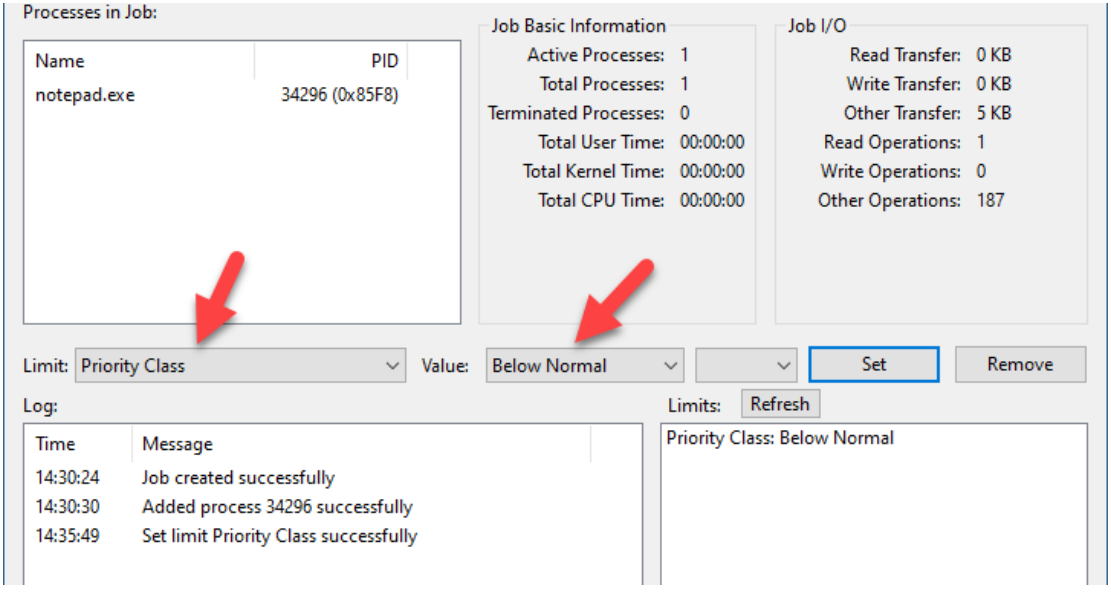


Figure 4-14: Setting Priority Class limit in *Job Monitor*

Now switch to *Task Manager*. *Notepad*'s base priority should now show *Below Normal* (figure 4-15).

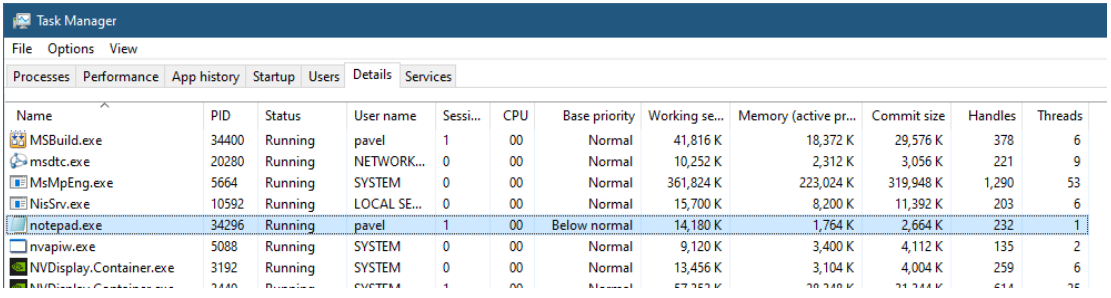


Figure 4-15: Limited base priority of *Notepad*

Trying to change the base priority with *Task Manager* by right-clicking *Notepad* and selecting *Set Priority* with any level (except *Below Normal*) will have no effect because of the job limit. Going back to *Job Monitor* and clicking *Remove* on the priority class limit will re-allow base

priority modifications.

You can find the code to set/remove most of the available job limits in the *JobMon* project in the *MainDlg.cpp* file.

CPU Rate Limit

Windows 8 added a CPU rate limit to the available job limits, which is not set by the basic or extended limits, but rather uses its own job limit enumeration: `JobObjectCpuRateControlInformation`.
The associated structure is `JOBOBJECT_CPU_RATE_CONTROL_INFORMATION` defined like so:

```
typedef struct _JOBOBJECT_CPU_RATE_CONTROL_INFORMATION {
  DWORD ControlFlags;
  union {
    DWORD  CpuRate;
    DWORD  Weight;
    struct {
      WORD MinRate;
      WORD MaxRate;
    };
  };
} JOBOBJECT_CPU_RATE_CONTROL_INFORMATION, *PJOBOBJECT_CPU_RATE_CONTROL_INFORMAT\
ION;
```

There are three different ways to set CPU rate limits, controlled by the `ControlFlags` field. Its possible values are summarized in table 4-5.

Table 4-5: Flags for CPU rate control

Flag (`JOB_OBJECT_CPU_RATE_*`)	Description
`ENABLE` (1)	Enable CPU rate control
`WEIGHT_BASED` (2)	CPU rate is based on relative weight (`Weight` member)
`HARD_CAP` (4)	Sets a hard cap for CPU consumption
`NOTIFY` (8)	Notifies the I/O completion port associated with the job (if any) for rate violations
`MIN_MAX_RATE` (0x10)	Sets the CPU rate between minimum and maximum values (`MinRate` and `MaxRate` members)

If CPU rate control is enabled, and neither JOB_OBJECT_CPU_RATE_CONTROL_WEIGHT_BASED nor JOB_OBJECT_CPU_RATE_CONTROL_MIN_MAX_RATE are specified, then the CpuRate member specifies the CPU limit percentage relative to 10000. For example, if 15 percent CPU is desired, the value should be set to 1500. This allows fractional CPU rates to be specified.
If the JOB_OBJECT_CPU_RATE_CONTROL_HARD_CAP flag is also specified, the limit is a hard one - the job won't get more CPU even if there are CPUs available. Without this flag, the job may get more CPU time if there are available processors.

> Behind the scenes, the kernel applied these restrictions by measuring the CPU consumption of the job in 300 msec time intervals, allowing/preventing it to/from executing in the next interval(s).

The *CpuLimit* project demonstrates using CPU rate control with the CpuRate member and a hard cap. The main function accepts an array of process IDs to put in a job, and a percentage to use as a hard CPU rate limit.
Here is the start of main:

```
int main(int argc, const char* argv[]) {
    if (!::IsWindows8OrGreater()) {
        printf("CPU Rate control is only available on Windows 8 and later\n");
        return 1;
    }

    if (argc < 3) {
        printf("Usage: CpuLimit <pid> [<pid> ...] <precentage>\n");
        return 0;
    }

    // create the job object

    HANDLE hJob = ::CreateJobObject(nullptr, L"CpuRateJob");
    if (!hJob)
        return Error("Failed to create object");

    for (int i = 1; i < argc - 1; i++) {
        int pid = atoi(argv[i]);
        HANDLE hProcess = ::OpenProcess(PROCESS_SET_QUOTA | PROCESS_TERMINATE,
            FALSE, pid);
        if (!hProcess) {
```

```
            printf("Failed to open handle to process %d (error=%d)\n", pid,
                ::GetLastError());
            continue;
        }
        if (!::AssignProcessToJobObject(hJob, hProcess)) {
            printf("Failed to assign process %d to job (error=%d)\n", pid,
                ::GetLastError());
        }
        else {
            printf("Added process %d to job\n", pid);
        }
        ::CloseHandle(hProcess);
    }
```

`main` starts by checking whether the application is executing on Windows 8 at least, as CPU rate control was not available in prior versions. Then, the program validates that there are at least one process ID and a CPU percentage by checking for at least 3 arguments (the program itself, PID, rate).

A job object is then created, with a name, so it's easier to identify using tools. Each process is opened and assigned to the job, if possible.
Now the program is ready to apply the CPU rate control:

```
    JOBOBJECT_CPU_RATE_CONTROL_INFORMATION info;
    info.CpuRate = atoi(argv[argc - 1]) * 100;
    info.ControlFlags = JOB_OBJECT_CPU_RATE_CONTROL_ENABLE |
        JOB_OBJECT_CPU_RATE_CONTROL_HARD_CAP;

    if (!::SetInformationJobObject(hJob, JobObjectCpuRateControlInformation,
        &info, sizeof(info)))
        return Error("Failed to set job limits");

    printf("CPU limit set successfully.\n");

    printf("Press ENTER to quit.\n");
    char dummy[10];
    gets_s(dummy);

    ::CloseHandle(hJob);
    return 0;
}
```

The CPU rate limit is calculated by taking the argument from the command line and multiplying it by 100. Finally, `SetInformationJobObject` is called to set the limit.
Before the program exits, it waits for the user to press *ENTER*. This allows the job object's handle to be open, and so easier to spot with tools. Otherwise, the handle would have been closed, marking the job for deletion. The limits, however, are still applied, as long as there are processes active in the job.

Let's test this by launching two instances of the *CpuStress* application (figure 4-16). The system used here has 16 logical processors, so we'll activate 4 threads with maximum activity in both of them. That should consume about 50% of all CPU time on the system (figure 4-17). *Task Manager* shows this is indeed the case (figure 4-18).

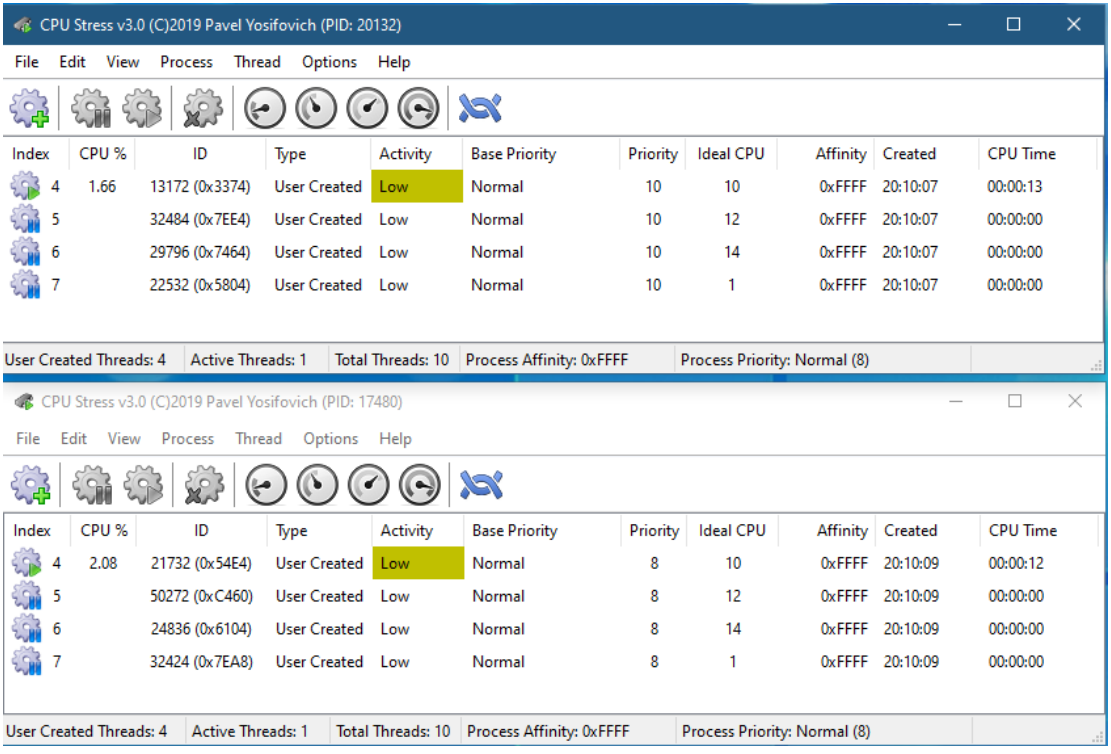


Figure 4-16: 2 instances of *CPU Stress* when launched

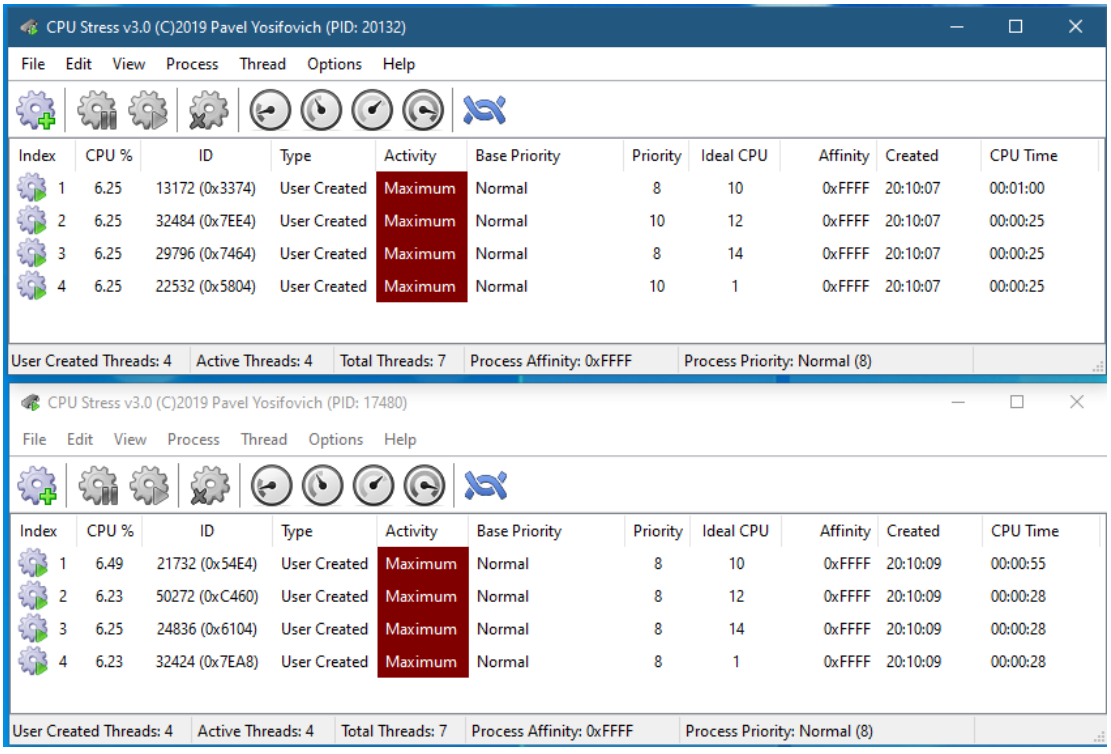


Figure 4-17: 2 *CPU Stress* with 4 maximum active threads

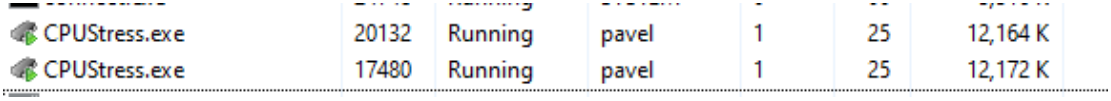


Figure 4-18: *Task Manager* showing 50% CPU utilization

Now we execute *CpuLimit* with the process IDs and the required CPU rate limit (20% in this example):

```
cpulimit 38984 28760 20
```

You should see a set of success messages like so:

```
CpuLimit.exe 20132 17480 20
Added process 20132 to job
Added process 17480 to job
CPU limit set successfully.
Press ENTER to quit.
```

At this point, you should be able to see the CPU consumption drop in both *CPUStress* instances (figure 4-19). The total CPU both consume should be around 20%, viewable in *Task Manager*.

Opening *Job Explorer* and looking at the *CpuRateJob* job (this is the name given in the code for easy identification) should show the CPU rate limit (figure 4-20).

Unfortunately, at the time of this writing, *Process Explorer* does not show CPU rate control information for jobs.

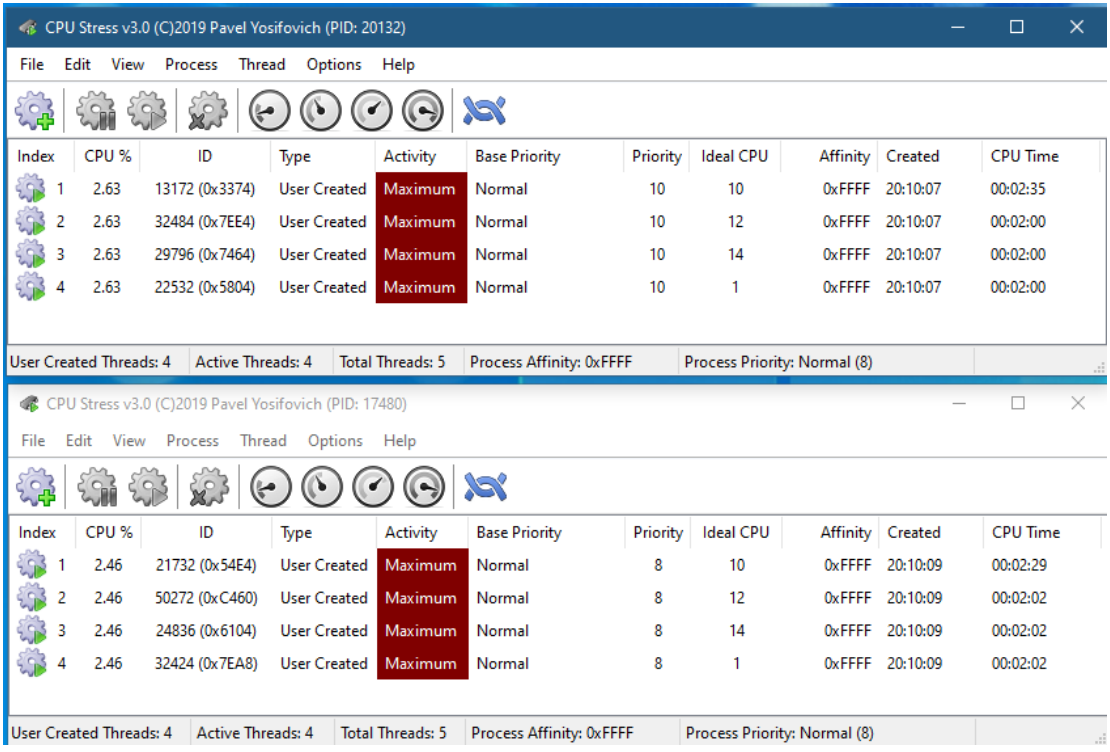


Figure 4-19: *CPU Stress* instances affected by CPU rate control

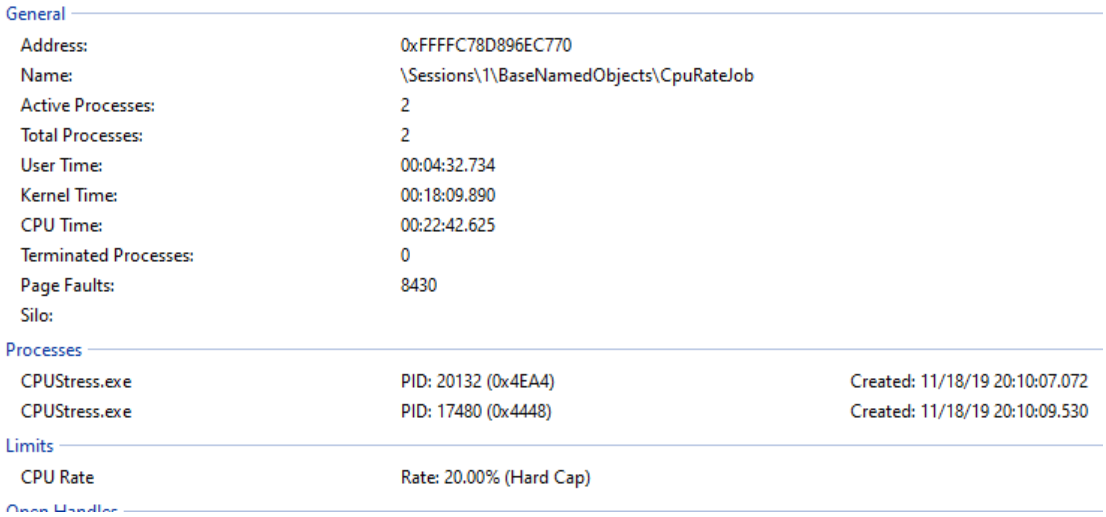


Figure 4-20: *Job Explorer* showing CPU rate control

If *CpuLimit* fails to add the processes with error 5 (access denied), run *CpuStress* from a command window rather than through explorer. If you're curious, investigate why this happens.

User Interface Limits

Another set of job limits is available through the `JobObjectBasicUIRestrictions` information class for user interface related restrictions. These are represented by a single 32-bit value stored in a simple structure:

```
typedef struct _JOBOBJECT_BASIC_UI_RESTRICTIONS {
    DWORD UIRestrictionsClass;
} JOBOBJECT_BASIC_UI_RESTRICTIONS, *PJOBOBJECT_BASIC_UI_RESTRICTIONS;
```

The available restrictions are bit flags listed in table 4-6.

A job that uses UI restrictions cannot be part of a job hierarchy.

Table 4-6: User Interface job restrictions

UI flag (`JOB_OBJECT_UILIMIT_*`)	Description
`NONE` (0)	No limits
`HANDLES` (1)	Processes in the job cannot access USER handles (e.g. windows) owned by processes not part of the job
`READCLIPBOARD` (2)	Processes in the job cannot read data from the clipboard
`WRITECLIPBOARD` (4)	Processes in the job cannot write data to the clipboard
`SYSTEMPARAMETERS` (8)	Processes in the job cannot change system parameters by calling `SystemParametersInfo`
`DISPLAYSETTINGS` (0x10)	Processes in the job cannot call `ChangeDisplaySettings`
`GLOBALATOMS` (0x20)	Processes in the job cannot access global atoms. The job has its own atom table (see the next sidebar)
`DESKTOP` (0x40)	Processes in the job cannot create or switch desktops (`CreateDesktop`, `SwitchDesktop`)
`EXITWINDOWS` (0x80)	Processes in the job cannot call `ExitWindows` or `ExitWindowsEx`

An *Atom Table* is a system-managed table that maps strings (or integers in a certain range) to integers. Each entry in the table is an *atom*. These atoms are used with user interface APIs, such as when registering a window class (`RegisterClass` / `RegisterClassEx`) or by manually manipulating an atom table (`AddAtom`, `FindAtom`, `GlobalAddAtom` and others). The *Global atom table* is available to all applications - this is the one that is not accessible in a job restricted with `JOB_OBJECT_UILIMIT_GLOBALATOMS`.

Here is a quick experiment to show one effect of UI restrictions. Open *Job Monitor*, create a new job and insert a *Notepad* instance into it. Then set a UI limit of Write Clipboard (figure 4-21).

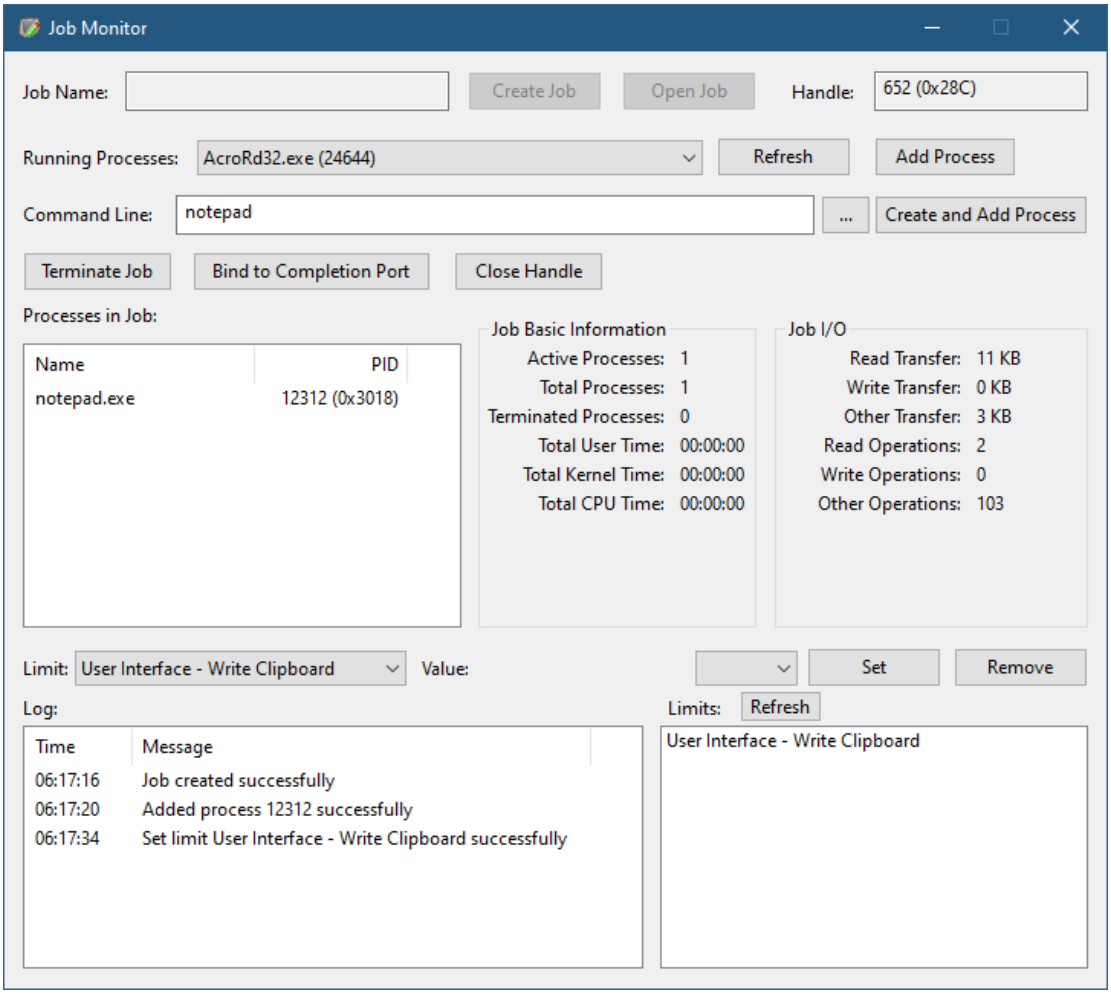


Figure 4-21: *Job Monitor* with a *Notepad* and UI restriction

Now open another *Notepad* instance outside of the job (or use any other text editing application). Copy some text from the application and try to paste it into the *Notepad* instance that is in the job. The operation should fail, even though the *Edit* menu shows the *Paste* option is enabled.

The `JOB_OBJECT_UILIMIT_HANDLES` flag prevents processes in the job from accessing other user interface objects (such as windows) outside the job. This means that calling functions such as `PostMessage` or `SendMessage` to windows outside the job fails. In some cases, there is a need to talk to a specific window outside the job from within the job. A process outside the job can grant (or remove) access to a window (or other USER object such as a menu or hook) by calling `UserHandleGrantAccess`:

```
BOOL UserHandleGrantAccess(
  _In_ HANDLE hUserHandle,      // user object handle
  _In_ HANDLE hJob,             // job handle
  _In_ BOOL   bGrant);          // TRUE to grant access, FALSE to remove it
```

The "hook" referred to in the previous paragraph is one of those that can be installed with `SetWindowsHookEx` (discussed in a later chapter). With this restriction in place, a process in the job cannot hook threads that run in processes outside the job.

Job Notifications

When job limits are violated, or when certain events occur, the job can notify an interested party via an I/O completion port, that can be associated with the job. I/O completion ports are typically used to handle completion of asynchronous I/O operations (which we'll tackle in a later chapter), but in this special case are used as the mechanism of notifying when certain job events occur.

> The job is a dispatcher (waitable) object, that becomes signaled when CPU time violation occurs. For this simple case, a thread can wait with `WaitForSingleObject` (as a common example) and then handle the CPU time violation. Setting a new CPU time limit resets the job to the non-signaled state.

The first step in getting notifications is to associate an I/O completion port with the job. Here is the relevant snippet from *JobMon* (`OnBindIoCompletion` function in *MainDlg.cpp*) (error handling omitted for clarity):

```
wil::unique_handle hCompletionPort(::CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, nullptr, 0, 0));

JOBOBJECT_ASSOCIATE_COMPLETION_PORT info;
info.CompletionKey = 0;     // application defined
info.CompletionPort = hCompletionPort.get();
::SetInformationJobObject(m_hJob.get(), JobObjectAssociateCompletionPortInforma\
tion,
    &info, sizeof(info));

// transfer ownership and store in a member
m_hCompletionPort = std::move(hCompletionPort);
```

Normally the first argument to `CreateIoCompletionPort` is a file handle, but in this case it's `INVALID_HANDLE_VALUE`, indicating no file is associated with the I/O completion port.

The next step is to wait for the completion port to fire by calling `GetQueuedCompletionStatus` defined like so:

```
BOOL GetQueuedCompletionStatus(
    _In_  HANDLE CompletionPort,
    _Out_ PDWORD pNumberOfBytesTransferred,
    _Out_ PULONG_PTR lpCompletionKey,
    _Out_ LPOVERLAPPED* pOverlapped,
    _In_  DWORD dwMilliseconds);
```

JobMon creates a thread and calls this function, waiting indefinitely until a notification arrives. A new thread is required in this case so that the UI thread of *JobMon* does not block, causing the UI to become unresponsive. Here is the relevant code from *JobMon* following the creation of the completion port:

```
// create a thread to monitor notifications
wil::unique_handle hThread(::CreateThread(nullptr, 0, [](auto p) {
    return static_cast<CMainDlg*>(p)->DoMonitorJob();
}, this, 0, nullptr));
```

> Thread creation is explained in detail in the next chapter.

The thread is passed the `this` pointer so that it can conveniently call a member function (`DoMonitorJob`). `DoMonitorJob` calls `GetQueuedCompletionStatus` and responds when the wait is over:

```
DWORD CMainDlg::DoMonitorJob() {
    for (;;) {
        DWORD message;
        ULONG_PTR key;
        LPOVERLAPPED data;
        if (::GetQueuedCompletionStatus(m_hCompletionPort.get(),
            &message, &key, &data, INFINITE)) {
            // handle notification
```

The meaning of the parameters to `GetQueuedCompletionStatus` are special when used with job notifications (as opposed to a file). `pNumberOfBytesTransferred` is the notification type, summarized in table 4-7. The `CompletionKey` parameter is the same one specified in `CreateIoCompletionPort`, and is application-defined. Finally, `pOverlapped` is extra information, whose format depends on the type of notification (table 4-7).

Table 4-7: Job notifications

Notification (`JOB_OBJECT_MSG_*`)	Associated data (`pOverlapped`)	Description
`END_OF_JOB_TIME` (1)	`NULL`	Job time limit has been exhausted. The time limit is now canceled and processes in the job continue to run
`END_OF_PROCESS_TIME` (2)	The PID of the process	A process exceeded its per-process CPU time (the process is being terminated)
`ACTIVE_PROCESS_LIMIT` (3)	`NULL`	The active processes limit has been exceeded
`ACTIVE_PROCESS_ZERO` (4)	`NULL`	The number of active processes became zero (all processes exited for whatever reason)
`NEW_PROCESS` (6)	The PID of the new process	A new process was added to the job (either directly or because another process in the job created it). When the completion port is initially associated, all active processes are reported as well
`EXIT_PROCESS` (7)	The PID of the exiting process	A process in the job has exited

Table 4-7: Job notifications

Notification (`JOB_OBJECT_MSG_*`)	Associated data (`pOverlapped`)	Description
`ABNORMAL_EXIT_PROCESS` (8)	The PID of the exiting process	A process has exited abnormally, which means it terminated because of an unhandled exception, from a given list of exceptions (check the documentation for a complete list)
`PROCESS_MEMORY_LIMIT` (9)	The PID of the process	A process in the job has exceeded its memory consumption limit
`JOB_MEMORY_LIMIT` (10)	The PID of the process	A process in the job caused the job to exceed its job-wide memory limit
`NOTIFICATION_LIMIT` (11)	The PID of the process	(Windows 8+) A process in the job that registered a to receive a notification limit has exceeded a limit.

In case of a notification limit, call `QueryInformationJobObject` with `JobObjectNotificationLimitInformation` and/or `JobObjectNotificationLimitInformation2` (Windows 10) to query for the limits being violated.

The following code snippet from *JobMon* shows how some of these notification codes are handled:

```
switch (message) {
    case JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT:
        AddLog(L"Job Notification: Active process limit exceeded");
        break;

    case JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO:
        AddLog(L"Job Notification: Active processes is zero");
        break;

    case JOB_OBJECT_MSG_NEW_PROCESS:
        AddLog(L"Job Notification: New process created (PID: "
            + std::to_wstring(PtrToUlong(data)) + L")");
        break;
```

```
    case JOB_OBJECT_MSG_EXIT_PROCESS:
        AddLog(L"Job Notification: process exited (PID: "
            + std::to_wstring(PtrToUlong(data)) + L")");
        break;

    case JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS:
        AddLog(L"Job Notification: Process " + std::to_wstring(
            PtrToUlong(data)) + L" exited abnormally");
        break;

    case JOB_OBJECT_MSG_JOB_MEMORY_LIMIT:
        AddLog(L"Job Notification: Job memory limit exceed attempt by process "
            + std::to_wstring(PtrToUlong(data)));
        break;

    case JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT:
        AddLog(L"Job Notification: Process " + std::to_wstring(
            PtrToUlong(data)) + L" exceeded its memory limit");
        break;

    case JOB_OBJECT_MSG_END_OF_JOB_TIME:
        AddLog(L"Job time limit exceeded");
        break;

    case JOB_OBJECT_MSG_END_OF_PROCESS_TIME:
        AddLog(L"Process " + std::to_wstring(PtrToUlong(data))
            + L" has exceeded its time limit");
        break;
}
```

`AddLog` is a private function that adds the corresponding message to the bottom list view.

For end-of-job time limit violation, the default action taken is to terminate all processes in the job. Each process' exit code is set to `ERROR_NOT_ENOUGH_QUOTA` (1816) and a notification is not sent. To change that, a call to `SetInformationJobObject` must be made beforehand to set a different end-of-job action with `JobObjectEndOfJobTimeInformation` information class, passing the following structure:

```
typedef struct _JOBOBJECT_END_OF_JOB_TIME_INFORMATION {
    DWORD EndOfJobTimeAction;
} JOBOBJECT_END_OF_JOB_TIME_INFORMATION, *PJOBOBJECT_END_OF_JOB_TIME_INFORMATIO\
N;

#define JOB_OBJECT_TERMINATE_AT_END_OF_JOB  0
#define JOB_OBJECT_POST_AT_END_OF_JOB       1
```

A value of `JOB_OBJECT_TERMINATE_AT_END_OF_JOB` is the default, while `JOB_OBJECT_POST_AT_END_OF_JOB` causes posting a notification message without terminating the processes. If a completion port is not associated with the job, this value has no effect and the termination protocol is used.

Silos

Windows 10 version 1607 and Windows Server 2016 introduced an enhanced version of a job known as *Silo*. A silo always starts as a job, but it can be upgraded to a silo by using `SetInformationJobObject` with an undocumented information class, `JobObjectCreateSilo` (35), that appears in the Windows SDK headers, but is not documented. Some of the Silo APIs are documented in the *Windows Driver Kit* (WDK) for use by device driver writers. Since silos are mostly controllable from kernel mode, their programmatic usage is out of scope for this book.

There are two variations to silos: Application Silos and Server Silos. Server silos are only supported on Windows server machines, starting with Server 2016. They are used today to implement *Windows Containers*, the ability to sandbox processes, creating a virtual environment that makes processes think they are on a machine of their own. This requires the redirection of file system, registry, and object namespace to be part of a particular silo, so the kernel had to go through significant changes internally to be silo-aware.

Application silos are used in applications that were converted to UWP using the *Desktop Bridge* technology. They are not nearly as powerful as server silos (nor they need to be). *Job Explorer* has a *Silo Type* column that indicates if a job is in fact a silo by listing its type. Figure 4-22 shows three application silos on the machine.

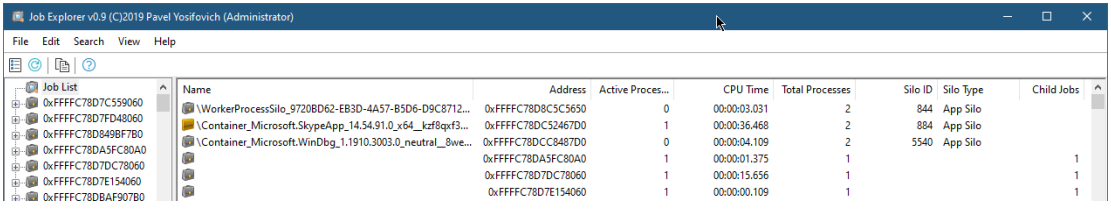


Figure 4-22: *Job Explorer* showing Silos

Notice silos have a silo ID, which is a unique job ID that is used internally to identify silos.

More detailed information on silos can be found in the "Windows Internals, 7th edition, Part 1" in chapter 3.

Exercises

1. Write a tool called *MemLimit* that accepts a process ID and a number representing the maximum committed memory for the process and set that limit using a job.
2. Extend *JobMon* to cover all remaining limits that are not currently implemented, such as I/O and network limits.

Summary

Jobs offer many opportunities to control and limit processes, all implemented by the kernel itself. The introduction of nested jobs in Windows 8 makes jobs more useful and less restrictive.

In the next chapter, we'll start looking at threads. Processes and jobs are management objects, but threads are the actual ones assigned to processors to do work, so there is no OS life without threads.

Chapter 5: Threads Basics

Processes are management objects, and do not execute code directly. To get anything done on Windows, threads must be created. As we've seen already, a user mode process is created with a single thread that eventually executes the main entry point of the executable. In many cases, this is good enough, and the application may not require any more threads.

Some applications, however, may benefit from using multiple threads executing within the process. Each thread is an independent path of execution, and so can use a different processor, resulting in true concurrency. In this chapter, we'll explore the fundamentals of creating and managing threads. In subsequent chapters, we'll delve into other aspects of threads, such as scheduling and synchronization.

In this chapter:

Introduction

The first question we should consider is why use threads in the first place? There are essentially two possible reasons:

1. Increase performance by utilizing multiple cores executing concurrently.
2. Improve application design.

Although you may come up with other reasons to use threads, these reasons somehow fall into the second category. It's always possible to design with a single thread (maybe by using timers, for example), without creating more threads. Still, the second reason is valid and is in fact the dominant one. A quick look at *Task Manager* in the *Performance/CPU* tab shows many (thousands of) threads, much more than the number of processors, and still the consumed CPU percentage at any time is low, meaning reason number 1 is not dominant (figure 5-1).

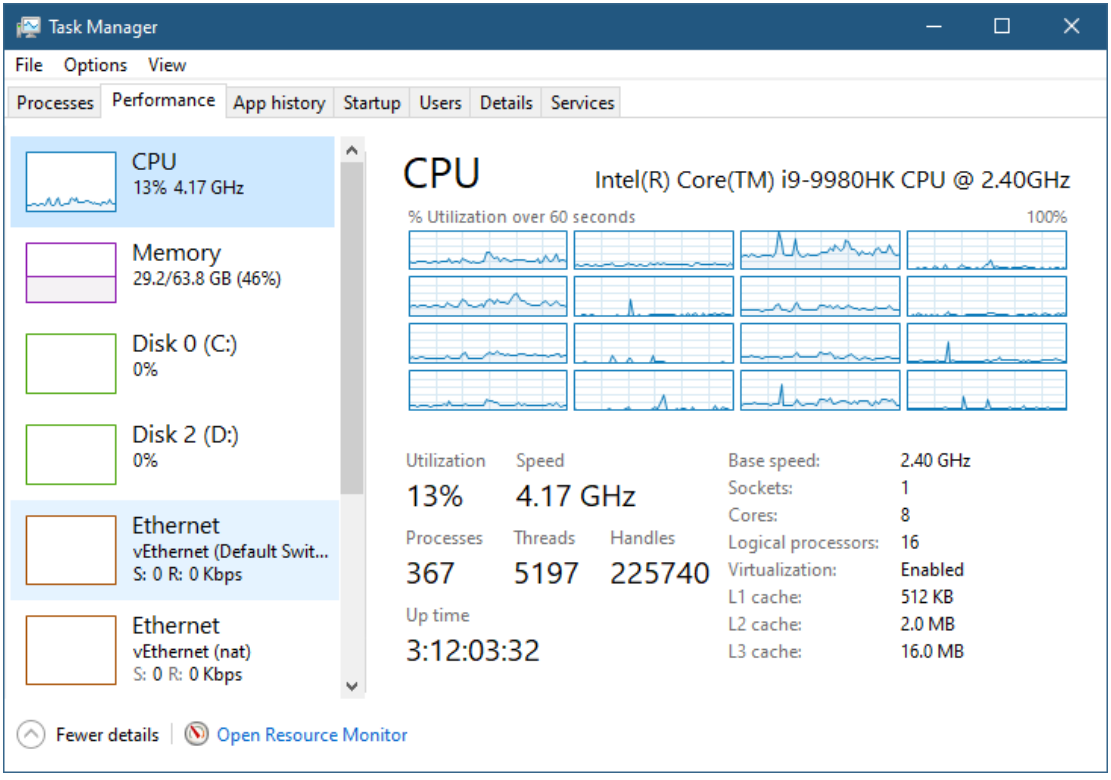


Figure 5-1: Performance/CPU in *Task Manager*

A thread abstracts an independent execution path, unrelated (from an execution perspective) to other threads that may be active at the same time. Once a thread starts execution, it may perform any of the following operations, until it exits:

- CPU-bound operations - calculations or invocation of functions that rely on CPU operations to make progress.
- I/O bound operations - operations performed against I/O devices, such as disks or the network. While waiting for an I/O operation to complete, the thread is in a wait state, and does not consume CPU cycles.
- Other operations that may result in the thread entering the wait state, such as waiting on a synchronization primitive (e.g. a mutex).

> Thread Synchronization is discussed in detail in chapter 7.

The fact that the CPU utilization in figure 5-1 is not 100% means that most threads are in a wait state (do not want to execute). In fact, if 16 threads execute code at the same time on the machine (figure 5-1), the CPU utilization would be 100%. It's only about 13%, which means there are about 2 processors active at the same time.

Sockets, Cores, and Logical Processors

Before we get any further with threads, we must realize that threads are an abstraction over processors. But what exactly is the definition of a processor? In the days where multiple cores make up a typical CPU, the terms might become confusing. Figure 5-2 shows a logical composition of a typical CPU.

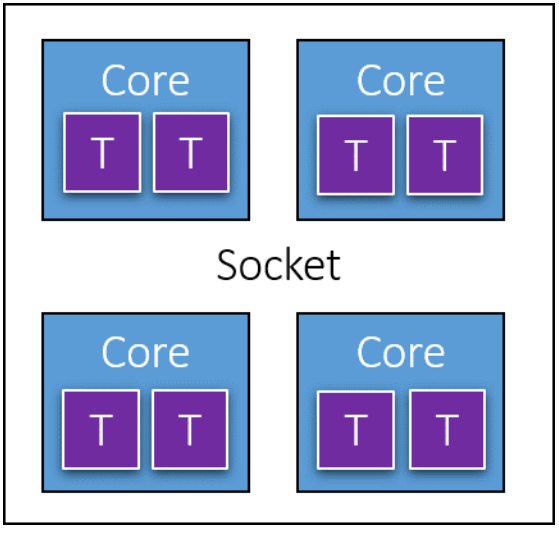


Figure 5-2: Logical composition of a CPU

In figure 5-2, there is one *socket*, which is the physical chip stuck in the computer's motherboard. Laptops and home computers typically have just one of these. Large server machines may contain more than one socket. Each socket has multiple *cores*, which are independent processors (4 in figure 5-2).

On Intel processors, each core may be split to two *logical processors*, also called *hardware threads* because of a technology called *Hyper-threading*. From Windows perspective, the number of processors is the number of logical processors (16 in figure 5-1). This means that at any given

moment, at most 16 threads may be running. The number of sockets, cores and logical processors is also shown by *Task Manager* (figure 5-1).

AMD has a similar technology called *Simultaneous Multi Threading* (SMT).

> Hyper-threading can be disabled in the BIOS settings. The potential downside of hyper-threading is that every two logical processors that share a core also share the Level 2 cache, and so might "interfere" with each other. Chapter 6 has more to say of caching.

Creating and Managing Threads

The basic function to create a thread is `CreateThread`:

```
HANDLE WINAPI CreateThread(
  _In_opt_  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  _In_      SIZE_T                 dwStackSize,
  _In_      LPTHREAD_START_ROUTINE lpStartAddress,
  _In_opt_  LPVOID                 lpParameter,
  _In_      DWORD                  dwCreationFlags,
  _Out_opt_ LPDWORD                lpThreadId);
```

The first argument to `CreateThread` should be recognizable by now, typically set to `NULL`. The `dwStackSize` parameter sets the thread's stack size, discussed in detail in the section "A Thread's Stack", later in this chapter. It's usually set to zero, which sets default sizes based on the PE header. I say "sizes" because the stack has an initial size and a maximum size (discussed later as well).

The `lpStartAddress` parameter is the most important one, specifying the user function to call from the new thread. This function can be named anything but it must adhere to the following prototype:

```
DWORD WINAPI ThreadProc(_In_ PVOID pParameter);
```

The thread function must return a 32-bit number, which is considered the thread's exit code, that can later be retrieved with `GetExitCodeThread`. The `WINAPI` macro expands to the `__stdcall`

keyword, signifying the standard calling convention which is common to most Windows APIs. Finally, the parameter passed to the function is a user-defined value that is passed as the fourth parameter to `CreateThread`, and simply passed as-is to the thread function. This value typically points to some data structure containing information that allows the thread to do its job.

Back to `CreateThread` - the `lpParameter` parameter was just discussed. In the simplest cases, `NULL` could be passed in. The `dwCreationFlags` argument can have three possible values (which can be combined). Specifying the `CREATE_SUSPENDED` flag creates the thread in a suspended state. The thread is ready to go, but a call to `ResumeThread` must be made to let it loose. Another possible value is `STACK_SIZE_PARAM_IS_A_RESERVATION`, which gives an alternate meaning to the stack size parameter (discussed in the section "A Thread's Stack" as well). Finally, specifying neither of these flags (most common), instructs the thread to start execution immediately. The last optional argument to `CreateThread` is the resulting unique thread ID of the new thread. If the caller is not interested in this information, it can simply specify `NULL` for this argument.

The return value from `CreateThread` is a handle to the newly created thread. If something goes wrong, the return value is `NULL` and `GetLastError` can be called to extract the error code. Once the handle is not needed, it should be closed with `CloseHandle` like any other kernel object handle.

The following code snippet creates a thread from the `main` function, waits for it to exit, and prints out its exit code:

```
DWORD WINAPI DoWork(PVOID) {
    printf("Thread ID running DoWork: %u\n", ::GetCurrentThreadId());
    // simulate some heavy work...
    ::Sleep(3000);
    // return a result
    return 42;
}

int main() {
    HANDLE hThread = ::CreateThread(nullptr, 0, DoWork, nullptr, 0, nullptr);
    if(!hThread) {
        printf("Failed to create thread (error=%u)\n", ::GetLastError());
        return 1;
    }

    // print ID of main thread
    printf("Main thread ID: %u\n", ::GetCurrentThreadId());

    // wait for the thread to finish
    ::WaitForSingleObject(hThread, INFINITE);
```

```
    DWORD result;
    ::GetExitCodeThread(hThread, &result);
    printf("Thread done. Result: %u\n", result);

    ::CloseHandle(hThread);
    return 0;
}
```

Here is an example output:

```
Main thread ID: 19108
Thread ID running DoWork: 23700
Thread done. Result: 42
```

The `GetExitCodeThread` allows retrieving the returned value from a thread's function:

```
BOOL GetExitCodeThread(
    _In_ HANDLE hThread,
    _Out_ LPDWORD lpExitCode);
```

You might be wondering what would happen if the function is called with a thread that has not exited yet. The function does not fail, but returns `STILL_ACTIVE` (0x103=259).

The Primes Counter Application

The following example illustrates a more complex usage of multiple threads. The *PrimesCounter* application (available in the samples for this chapter), counts the number of prime numbers in a range of numbers using a specified number of threads. The idea is to split the work between several threads, each counting prime numbers in its range of numbers. Then the main thread waits for all worker threads to exit, allowing it to simply sum the counts from all threads. This is depicted in figure 5-3.

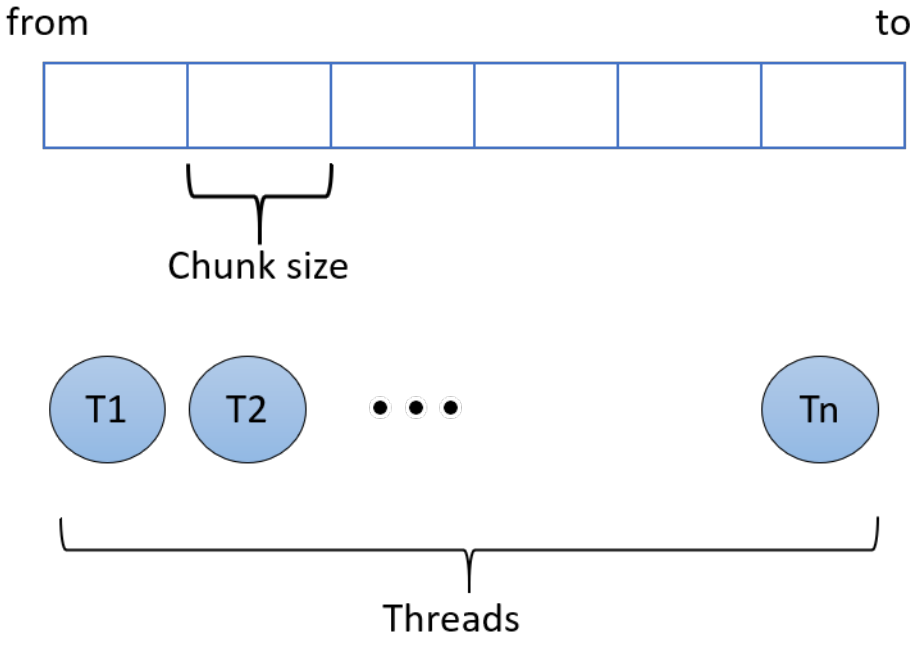


Figure 5-3: *Primes Counter* design

This idea of creating several threads that do some work, and waiting for them to exit before aggregating the results is sometimes called *Fork-Join*, because threads are "forked" from some initial thread and then "joined back" to the initial thread upon completion.

> Another name for this pattern is *Structured Parallelism*.

The number of threads used in this application is one of the parameters of the algorithm - the interesting question is what number of threads is optimal to complete the calculation the quickest? This will be discussed later; but, first - the code.

The `main` function accepts the range of numbers and the number of threads on the command line:

```
int main(int argc, const char* argv[]) {
    if (argc < 4) {
        printf("Usage: PrimesCounter <from> <to> <threads>\n");
        return 0;
    }

    int from = atoi(argv[1]);
    int to = atoi(argv[2]);
    int threads = atoi(argv[3]);

    if (from < 1 || to < 1 || threads < 1 || threads > 64) {
        printf("Invalid input.\n");
        return 1;
    }
```

The number of threads is limited to 64. Why this number? This is the maximum number of handles that can be waited upon at the same time with `WaitForMultipleObjects`, which is used later to wait until all threads exit.

The next call in `main` is to the function that initiates the work and returns the result:

```
DWORD elapsed;
int count = CalcAllPrimes(from, to, threads, elapsed);
printf("Total primes: %d. Elapsed: %d msec\n", count, elapsed);
```

`CalcPrimes` accepts the arguments extracted from the command line and returns the total primes counted, and also returns the elapsed time in milliseconds using the last `elapsed` parameter (passed by reference). Finally, the results are echoed to the console.

Each thread needs its "from" and "to" numbers and somewhere to put the result. Since a thread function can return a 32-bit unsigned integer, this can be used here. But in the general case, the return value may not be flexible enough. The typical solution is to define a structure that has all the information required by the thread, including input and output values. For our application, the following structure is defined:

```
struct PrimesData {
    int From, To;
    int Count;
};
```

`CalcAllPrimes` must allocate a `PrimesData` instance for each thread and initialize the `From` and `To` data members:

```
int CalcAllPrimes(int from, int to, int threads, DWORD& elapsed) {
    auto start = ::GetTickCount64();
    // allocate data for each thread
    auto data = std::make_unique<PrimesData[]>(threads);
    // allocate an array of handles
    auto handles = std::make_unique<HANDLE[]>(threads);
```

The current time is captured before any work is done with `GetTickCount64`. This API returns the number of milliseconds elapsed since Windows booted. Although it's not the most accurate API to use (`QueryPerformanceCounter` is more accurate), it will do for this application's purposes.

`GetTickCount64` replaces the older `GetTickCount` by returning a 64 bit number as opposed to a 32-bit number returned from `GetTickCount`. A 32-bit number of milliseconds will overflow and roll back to zero after about 49.7 days.

The code uses `std::unique_ptr<[]>` to managed an array that is cleaned up automatically when the variable goes out of scope. This is used for the `PrimesData` array as well as for the threads handles.

Next, the function calculates the chunk size for each thread and then loops to create the threads appropriately:

```
int chunk = (to - from + 1) / threads;

for (int i = 0; i < threads; i++) {
    auto& d = data[i];
    d.From = i * chunk;
    d.To = i == threads - 1 ? to : (i + 1) * chunk - 1;

    DWORD tid;
    handles[i] = ::CreateThread(nullptr, 0, CalcPrimes, &d, 0, &tid);
    assert(handles[i]);
    printf("Thread %d created. TID=%u\n", i + 1, tid);
}
```

Each thread's `PrimesData` instance is initialized with the correct `From` and `To` based on the chunk size. The only wrinkle is that the range may not be exactly divisible by the number of threads. So, the last thread is tasked with the "tail" of numbers (if any). `CreateThread` is called to create each thread, pointing each thread to the `CalcPrimes` function (discussed shortly), passing to it its personal `PrimesData` pointer. Finally, the thread index and ID are displayed.

`CalcPrimes` is the thread function tasked with counting the prime numbers in the range supplied to that thread:

```
DWORD WINAPI CalcPrimes(PVOID param) {
    auto data = static_cast<PrimesData*>(param);
    int from = data->From, to = data->To;
    int count = 0;
    for (int i = from; i <= to; i++)
        if (IsPrime(i))
            count++;

    data->Count = count;
    return count;
}
```

The parameter passed in to a thread is cast to a `PrimesData` pointer. Then a simple `for` loop checks if the number is a prime one and if so increments a counter that is eventually stored in the `PrimesData`'s `Count` member. `IsPrime` is a simple function returning `true` for a prime number and `false` otherwise:

```
bool IsPrime(int n) {
    if (n < 2)
        return false;
    if(n == 2)
        return true;

    int limit = (int)::sqrt(n);
    for (int i = 2; i <= limit; i++)
        if (n % i == 0)
            return false;

    return true;
}
```

> The algorithm used in `IsPrime` is certainly not optimal, but that's not the point.

Back in `CalcAllPrimes`, all threads are created without the `CREATE_SUSPENDED` flag, so they start immediately. All that remains is to wait until all threads exit:

```
::WaitForMultipleObjects(threads, handles.get(), TRUE, INFINITE);
```

A full discussion of the wait functions is saved for chapter 7. `WaitForMultipleObjects` above accepts the following arguments, in order:

- the number of handles in the array
- the array of handles to wait on
- a boolean flag indicating if the wait should be for all handles to become signaled (`TRUE`) or just one (`FALSE`). For threads, the meaning of "signaled" is "exited".
- a timeout in milliseconds, in this case `INFINITE` means wait as long as it takes.

Once all threads exit, the wait is over. All that's left is to gather results:

```
elapsed = static_cast<DWORD>(::GetTickCount64() - start);

FILETIME dummy, kernel, user;
int total = 0;
for (int i = 0; i < threads; i++) {
    ::GetThreadTimes(handles[i], &dummy, &dummy, &kernel, &user);
    int count = data[i].Count;
    printf("Thread %2d Count: %7d. Execution time: %4u msec\n",
        i + 1, count,
        (user.dwLowDateTime + kernel.dwLowDateTime) / 10000);
    total += count;
    ::CloseHandle(handles[i]);
}
return total;
```

The code above uses the `GetThreadTimes` API to retrieved timing information for a thread:

```
BOOL GetThreadTimes(
  _In_  HANDLE     hThread,
  _Out_ LPFILETIME lpCreationTime,
  _Out_ LPFILETIME lpExitTime,
  _Out_ LPFILETIME lpKernelTime,
  _Out_ LPFILETIME lpUserTime);
```

The function returns a thread's creation time, exit time, the time it spent executing in kernel mode and the time it spend executing in user mode. For this application, I wanted to display the

execution time, which means summing up kernel time and user time, while disregarding create and exit times.

The kernel and user times are reported in a `FILETIME` structure, which is a 64-bit value stored in two 32-bit values:

```
typedef struct _FILETIME {
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
} FILETIME, *PFILETIME, *LPFILETIME;
```

The value is in 100 nanosecond units (10 to the -7th power), which means the value in milliseconds can be obtained by dividing by 10000. The code assumes the elapsed time is not higher than a 32-bit value in 100 nanosecond units, which may not be correct in the general case.

Running Primes Counter

Here are a few runs for the same value range, starting from a baseline with one thread:

```
C:\Dev\Win10SysProg\x64\Release>PrimesCounter.exe 3 20000000 1
Thread 1 created (3 to 20000000). TID=29760
Thread  1 Count: 1270606. Execution time: 9218 msec
Total primes: 1270606. Elapsed: 9218 msec

C:\Dev\Win10SysProg\x64\Release>PrimesCounter.exe 3 20000000 2
Thread 1 created (3 to 10000001). TID=22824
Thread 2 created (10000002 to 20000000). TID=41816
Thread  1 Count:  664578. Execution time: 3625 msec
Thread  2 Count:  606028. Execution time: 5968 msec
Total primes: 1270606. Elapsed: 5984 msec

C:\Dev\Win10SysProg\x64\Release>PrimesCounter.exe 3 20000000 4
Thread 1 created (3 to 5000001). TID=52384
Thread 2 created (5000002 to 10000000). TID=47756
Thread 3 created (10000001 to 14999999). TID=42296
Thread 4 created (15000000 to 20000000). TID=34972
Thread  1 Count:  348512. Execution time: 1312 msec
Thread  2 Count:  316066. Execution time: 2218 msec
Thread  3 Count:  306125. Execution time: 2734 msec
Thread  4 Count:  299903. Execution time: 3140 msec
Total primes: 1270606. Elapsed: 3141 msec
```

```
C:\Dev\Win10SysProg\x64\Release>PrimesCounter.exe 3 20000000 8
Thread 1 created (3 to 2500001). TID=25200
Thread 2 created (2500002 to 5000000). TID=48588
Thread 3 created (5000001 to 7499999). TID=52904
Thread 4 created (7500000 to 9999998). TID=18040
Thread 5 created (9999999 to 12499997). TID=50340
Thread 6 created (12499998 to 14999996). TID=43408
Thread 7 created (14999997 to 17499995). TID=53376
Thread 8 created (17499996 to 20000000). TID=33848
Thread  1 Count:  183071. Execution time:  578 msec
Thread  2 Count:  165441. Execution time:  921 msec
Thread  3 Count:  159748. Execution time: 1171 msec
Thread  4 Count:  156318. Execution time: 1343 msec
Thread  5 Count:  154123. Execution time: 1531 msec
Thread  6 Count:  152002. Execution time: 1531 msec
Thread  7 Count:  150684. Execution time: 1718 msec
Thread  8 Count:  149219. Execution time: 1765 msec
Total primes: 1270606. Elapsed: 1766 msec

C:\Dev\Win10SysProg\x64\Release>PrimesCounter.exe 3 20000000 16
Thread 1 created (3 to 1250001). TID=50844
Thread 2 created (1250002 to 2500000). TID=9792
Thread 3 created (2500001 to 3749999). TID=12600
Thread 4 created (3750000 to 4999998). TID=52804
Thread 5 created (4999999 to 6249997). TID=5408
Thread 6 created (6249998 to 7499996). TID=42488
Thread 7 created (7499997 to 8749995). TID=49336
Thread 8 created (8749996 to 9999994). TID=13384
Thread 9 created (9999995 to 11249993). TID=41508
Thread 10 created (11249994 to 12499992). TID=12900
Thread 11 created (12499993 to 13749991). TID=39512
Thread 12 created (13749992 to 14999990). TID=3084
Thread 13 created (14999991 to 16249989). TID=52760
Thread 14 created (16249990 to 17499988). TID=17496
Thread 15 created (17499989 to 18749987). TID=39956
Thread 16 created (18749988 to 20000000). TID=31672
Thread  1 Count:   96468. Execution time:  281 msec
Thread  2 Count:   86603. Execution time:  484 msec
Thread  3 Count:   83645. Execution time:  562 msec
Thread  4 Count:   81795. Execution time:  671 msec
```

```
Thread  5 Count:   80304. Execution time:  781 msec
Thread  6 Count:   79445. Execution time:  812 msec
Thread  7 Count:   78589. Execution time:  859 msec
Thread  8 Count:   77729. Execution time:  828 msec
Thread  9 Count:   77362. Execution time:  906 msec
Thread 10 Count:   76761. Execution time: 1000 msec
Thread 11 Count:   76174. Execution time:  984 msec
Thread 12 Count:   75828. Execution time: 1046 msec
Thread 13 Count:   75448. Execution time: 1078 msec
Thread 14 Count:   75235. Execution time: 1062 msec
Thread 15 Count:   74745. Execution time: 1062 msec
Thread 16 Count:   74475. Execution time: 1109 msec
Total primes: 1270606. Elapsed: 1188 msec

C:\Dev\Win10SysProg\x64\Release>PrimesCounter.exe 3 20000000 20
Thread 1 created (3 to 1000001). TID=30496
Thread 2 created (1000002 to 2000000). TID=7300
Thread 3 created (2000001 to 2999999). TID=50580
Thread 4 created (3000000 to 3999998). TID=21536
Thread 5 created (3999999 to 4999997). TID=24664
Thread 6 created (4999998 to 5999996). TID=34464
Thread 7 created (5999997 to 6999995). TID=51124
Thread 8 created (6999996 to 7999994). TID=29972
Thread 9 created (7999995 to 8999993). TID=50092
Thread 10 created (8999994 to 9999992). TID=49396
Thread 11 created (9999993 to 10999991). TID=18264
Thread 12 created (10999992 to 11999990). TID=33496
Thread 13 created (11999991 to 12999989). TID=16924
Thread 14 created (12999990 to 13999988). TID=44692
Thread 15 created (13999989 to 14999987). TID=53132
Thread 16 created (14999988 to 15999986). TID=53692
Thread 17 created (15999987 to 16999985). TID=5848
Thread 18 created (16999986 to 17999984). TID=12760
Thread 19 created (17999985 to 18999983). TID=13180
Thread 20 created (18999984 to 20000000). TID=49980
Thread  1 Count:   78497. Execution time:  218 msec
Thread  2 Count:   70435. Execution time:  343 msec
Thread  3 Count:   67883. Execution time:  421 msec
Thread  4 Count:   66330. Execution time:  484 msec
Thread  5 Count:   65366. Execution time:  578 msec
Thread  6 Count:   64337. Execution time:  640 msec
```

```
Thread  7 Count:   63798. Execution time:  640 msec
Thread  8 Count:   63130. Execution time:  703 msec
Thread  9 Count:   62712. Execution time:  718 msec
Thread 10 Count:   62090. Execution time:  703 msec
Thread 11 Count:   61937. Execution time:  781 msec
Thread 12 Count:   61544. Execution time:  812 msec
Thread 13 Count:   61191. Execution time:  796 msec
Thread 14 Count:   60826. Execution time:  843 msec
Thread 15 Count:   60627. Execution time:  875 msec
Thread 16 Count:   60425. Execution time:  875 msec
Thread 17 Count:   60184. Execution time:  875 msec
Thread 18 Count:   60053. Execution time:  890 msec
Thread 19 Count:   59681. Execution time:  875 msec
Thread 20 Count:   59560. Execution time:  906 msec
Total primes: 1270606. Elapsed: 1109 msec
```

The system on which these runs executed has 16 logical processors. Here is a couple of interesting observations from the above output:

- Improvement of execution time with increasing number of threads is not linear (not even close).
- Using more threads than the number of logical processors reduces execution time.

Why do we get these results? What is the optimal number of threads in a fork-join style algorithm? It seems that answer should be "the number of logical processors", since and more threads will cause context switches to occur, because not all threads can execute at the same time, while using less threads would definitely leave some processors not utilized.

The reality, however, is not that simple. The single reason we get the two observations we did is this: the work is not split equally between the threads (in terms of execution time). This is simply because of the algorithm used: the larger the number, the more work has to be done because the `sqrt` function is a monotonic one and its output is in direct proportion to its input. This is often the challenge in fork-join algorithms: fair split of work. Figure 5-4 demonstrates what happens in an example case with four threads.

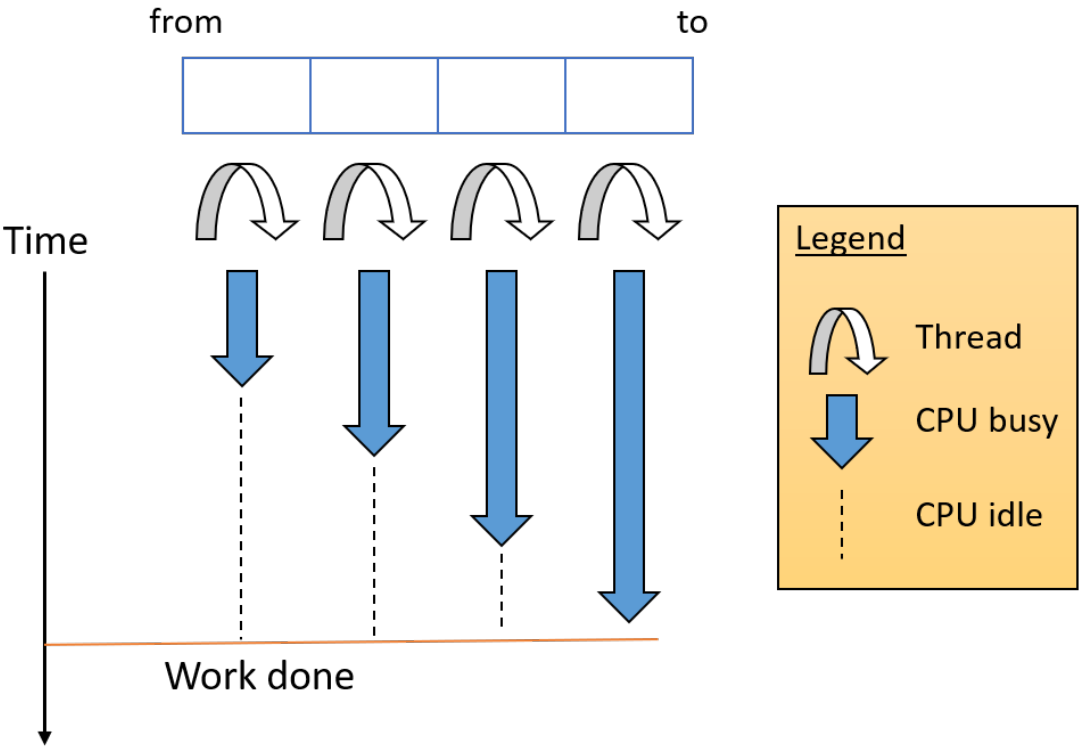


Figure 5-4: *Primes Counter* with 4 threads

Notice in the output above how later threads have longer running time, simply because they have more work to do. Now it may be obvious why we get better running time with 20 threads even if there are only 16 logical processors on the system. Early threads that finish quickly make processors free, allowing those "extra" threads (after 16) to get a processor, thus pushing work onward. Is there a limit? Of course, at some point, the context switch overhead, coupled with possible page faults because of more memory allocations for thread stacks will start making things worse. Clearly, asking what is the optimum number of processors for this program is not an easy question. And it can get even worse: this program only does CPU bound operations; no I/O. If threads need to do I/O from time to time, the question becomes even more difficult.

Terminating Threads

Every good (or bad) thread must at some point come to an end. There are three ways for a thread to terminate:

1. The thread function returns (best option)

2. The thread calls `ExitThread` (best to avoid)
3. The thread is terminated with `TerminateThread` (typically a bad idea)

The best option is to simply return from the thread function. When a thread starts execution, the thread function is not really the first or only function executed by the thread. The thread in fact starts execution inside an *NTDLL.dll* function named `RtlUserThreadStart`, which conceptually calls the thread's actual function as provided to `CreateThread`. Once the thread's function returns, `RtlUserThreadStart` does some cleanup and calls `ExitThread`. Note that `ExitThread` can only be called by a thread to terminate itself as its prototype suggests:

```
void ExitThread(_In_ DWORD exitCode);
```

> `ExitThread` from *Kernel32.dll* is actually a forwarder to `RtlExitUserThread` in *NtDll.Dll*.

The problem with calling `ExitThread` explicitly from the thread's function is at least the fact that C++ destructors won't be called, as `ExitThread` never returns. So it's always better to simply return from the thread's function to allow it to cleanup local C++ objects properly.

In any case, `ExitThread` also calls the `DllMain` function for all DLLs in the process with the `DLL_THREAD_DETACH` reason argument. This allows DLLs to perform per-thread operations. For example, DLLs can allocate some block of memory to manage something on a per-thread basis. In many cases this is combined with *Thread Local Storage* (TLS), discussed in chapter 10.

The third option of terminating a thread is with a call to `TerminateThread` that can be made from another thread (even belonging to another process). The only condition is the caller's ability to obtain a handle to the thread with a `THREAD_TERMINATE` access mask. Here is the definition of `TerminateThread`:

```
BOOL WINAPI TerminateThread(
  _Inout_ HANDLE hThread,
  _In_    DWORD  dwExitCode);
```

Terminating a thread with this call is almost always a bad idea. The problem lies in what the thread managed to do and what it has not yet done because of its termination. If the thread is terminated while doing actual work, there is no way to tell what instructions it executed and what other code it could not execute because of termination. The application could be in an inconsistent state. As an extreme (but not unlikely) example, the thread may have acquired a critical section (see chapter 7), and did not get the chance to release it, causing a deadlock, since other threads waiting for the critical section will wait forever.

Another issue with TerminateThread is that it does not call DLLs DllMain function with DLL_THREAD_DETACH. This means DLLs cannot run some code that might free memory or perform other actions to reverse what was done when the thread was created.

These problems with TerminateThread mean that calling this function safely is a rare occurrence, and there should be a better way to handle whatever scenario that seems to require it. Still, if this is desirable, the caller must obtain a powerful-enough handle having the THREAD_TERMINATE access right. Thread handles returned from CreateThread and CreateProcess always have full permissions. For other cases, obtaining a handle for an arbitrary thread can be attempted with OpenThread:

```
HANDLE OpenThread(
  _In_ DWORD dwDesiredAccess,
  _In_ BOOL  bInheritHandle,
  _In_ DWORD dwThreadId);
```

The function looks similar to OpenProcess, discussed in chapter 3. If the requested access mask can be obtained, a non-NULL handle is returned to the caller. If THREAD_TERMINATE is asked for and received, a call to TerminateThread is bound to succeed.

A Thread's Stack

Local variables and return addresses from functions reside on a thread's stack. The size of a thread's stack can be specified with the second parameter to CreateThread, but there are actually two values that affect a thread's stack: a reserved memory size that serves as the maximum size of the stack, and an initial, committed memory size, that is ready for use. The terms *reserved* and *committed* will be discussed in depth in chapter 12, but here's the gist of it: Reserved memory just marks a contiguous address space range as used for some purpose so that new allocations in the process address space won't be made from that range. For a stack, this is essential, as a stack is always contiguous. Committed memory means memory actually allocated, and so can be used.

It's possible to allocate the maximum stack size immediately, committing the entire stack upfront, but that would be wasteful, as a thread might not need the entire range for its stack related work. The memory manager has an optimization up its sleeve: commit a smaller amount of memory and if the stack grows beyond that amount, trigger an expansion of the stack, up to the reserved limit. The triggering is done by a page with a special flag, PAGE_GUARD that causes an exception if touched. This exception is caught by the memory manager, which then commits an additional page, moving the PAGE_GUARD page one page down (remember that a stack grows to lower addresses). Figure 5-5 shows this arrangement.

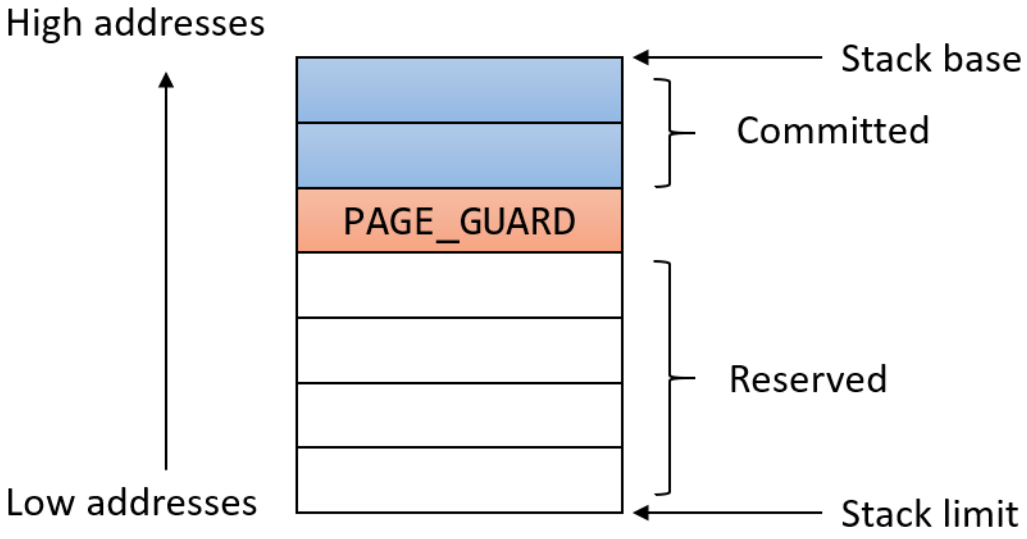


Figure 5-5: A thread's stack

> The actual minimum for a guard page is 12 KB, meaning 3 pages. This guarantees that a stack expansion will allow at least 12 KB of committed memory to be available for the stack.

Typically, zero is passed as the stack (second) argument to `CreateThread`. In that case, the defaults for the committed and reserved sizes are retrieved from values stored in the Portable Executable (PE) header. The first thread, which is created by the kernel and so is out of our control always uses these values. You can dump this values using the *dumpbin* utility that is available as part of the Windows SDK. Here is an example with *Notepad*:

```
C:\>dumpbin /headers c:\windows\system32\notepad.exe
Microsoft (R) COFF/PE Dumper Version 14.24.28314.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file c:\windows\system32\notepad.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
```

```
            8664 machine (x64)
               7 number of sections
        9E7797DD time date stamp
               0 file pointer to symbol table
               0 number of symbols
              F0 size of optional header
              22 characteristics
                   Executable
                   Application can handle large (>2GB) addresses

OPTIONAL HEADER VALUES
...
           80000 size of stack reserve
           11000 size of stack commit
          100000 size of heap reserve
            1000 size of heap commit
...
```

The default commit size for a thread's stack in *Notepad* is 0x11000 (68 KB) and the reserved size is 0x80000 (512 KB). These are the values used for *Notepad*'s first thread for sure. Other threads, explicitly created with `CreateThread` will have these values if the stack argument passed to `CreateThread` is zero.

You can also view this information in several free graphical tools, such as my own *PE Explorer v2*.

You can view this information with the *VMMap Sysinternals* tool. Run *Notepad*, and then run *VMMap*. Select the *Notepad* process from the dialog (figure 5-6). Then click *OK*.

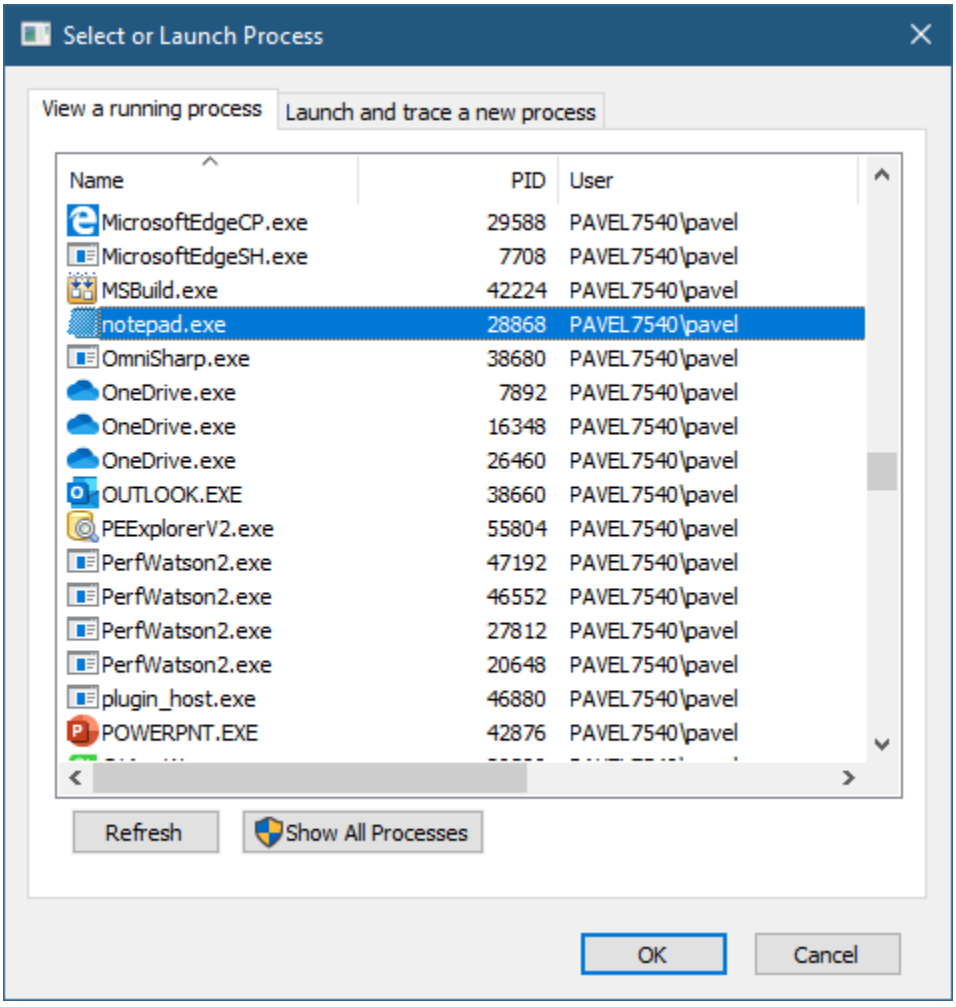


Figure 5-6: Process selector in *VMMap*

VMMap's main window opens. Select the *Stack* item in the middle list. This focuses the lower list to thread stacks only (figure 5-7).

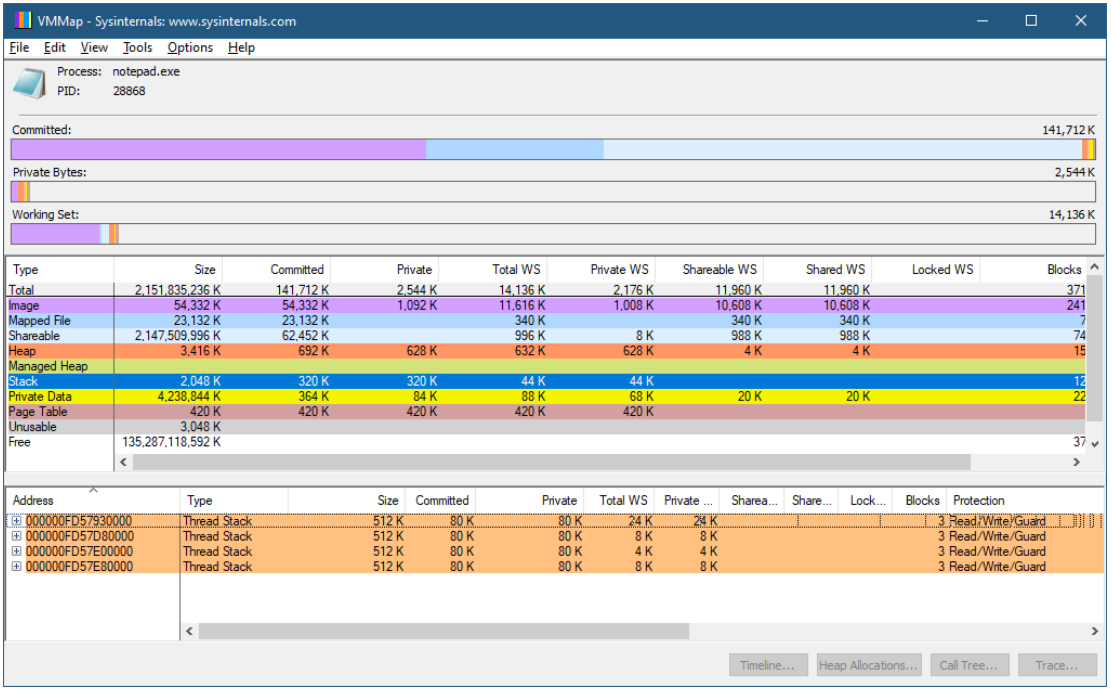


Figure 5-7: Selecting stacks in *VMMap*

Now open on of the stack items in the lower pane. You should see a committed size of 0x11000 bytes (68 KB) with a protection of Read/Write. Then a 12 KB guard page range, with the rest of the memory reserved (figure 5-8).

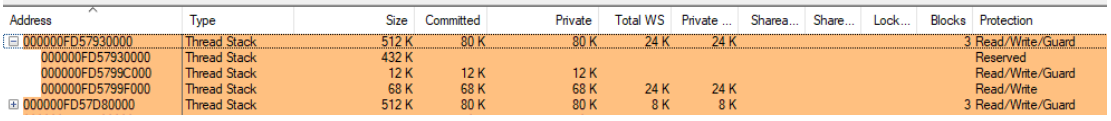


Figure 5-8: One thread's stack in *VMMap*

> *VMMap* will be discussed more thoroughly in chapter 12.

The `CreateThread` function has only one parameter for the stack size, and so it allows setting the initial committed memory or the maximum reserved memory, but not both. This is based on the flags argument. If it contains `STACK_SIZE_PARAM_IS_A_RESERVATION`, then the value is the reserved size; otherwise, it's the upfront committed size.

> The fact that `CreateThread` only allows setting one of the values seems to be an oversight. The native function (from *NtDll*) `NtCreateThreadEx` allows setting both values.

Visual Studio allows changing the default stack sizes using the project's properties, under the *Linker/System* node (figure 5-9). This simply sets the requested values in the PE header.

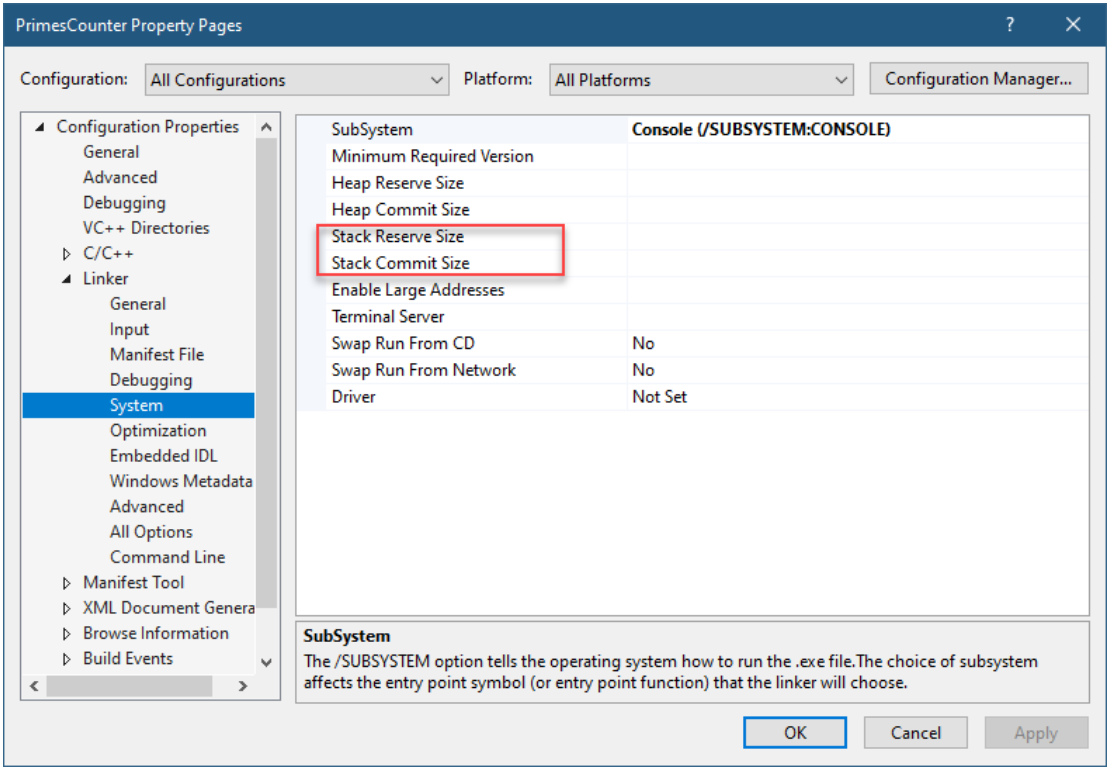


Figure 5-9: Stack sizes in Visual Studio

Finally, a thread can call `SetThreadStackGuarantee` to attempt at guaranteeing a certain stack size is available:

```
BOOL SetThreadStackGuarantee(_Inout_ PULONG StackSizeInBytes);
```

If the function succeeds, the increase in stack size is done by allocating more guard pages (which are also marked as committed), meaning they are guaranteed to be available if a stack expansion is required.

A Thread's Name

Starting with Windows 10 and Server 2016, a thread can have a string-based name or description, set with `SetThreadDescription`:

```
HRESULT SetThreadDescription(
    _In_ HANDLE hThread,
    _In_ PCWSTR lpThreadDescription);
```

The thread handle must have the `THREAD_SET_LIMITED_INFORMATION` access mask, which is easy to get for almost any thread. The name/description can be anything. Notice this function returns an `HRESULT`, where `S_OK` (0) means success. It's important to realize this is not the same as naming other kernel objects; there is no way to lookup a thread by its name/description. The name is just stored in the thread's kernel object and can be used as a debugging aid. Here is a simple example of setting the current thread's name:

```
::SetThreadDescription(::GetCurrentThread(), L"My Super Thread");
```

Visual Studio 2019 and later versions show the thread's name (if any) in the debugger's Threads window (figure 5-10).

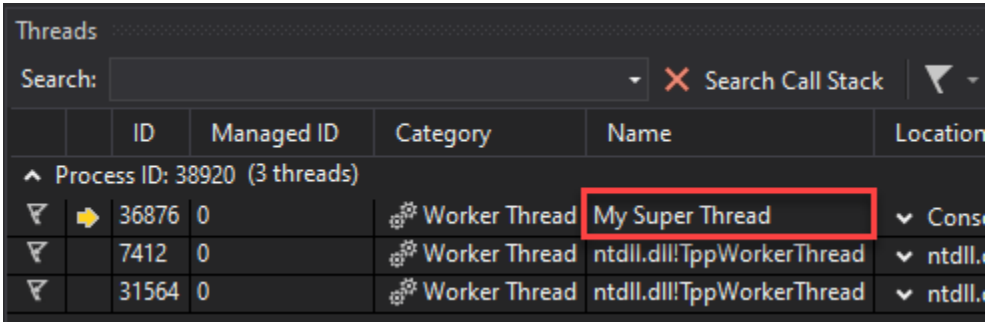


Figure 5-10: Thread's name in the Visual Studio debugger

Naturally, the opposite function exists as well:

```
HRESULT GetThreadDescription(
    _In_ HANDLE hThread,
    _Out_ PWSTR* ppszThreadDescription);
```

`GetThreadDescription` returns the result in a caller allocated pointer. The function requires calling `LocalFree` to free the memory it allocated. Here is an example:

```
PWSTR name;
if (SUCCEEDED(::GetThreadDescription(::GetCurrentThread(), &name))) {
    printf("Name: %ws\n", name);
    ::LocalFree(name);
}
```

What About the C++ Standard Library?

This book is about Windows programming, so directly discussing C++ may be out of place. Still, starting from the C++ 11 standard, the C++ standard library offers threading mechanisms (in fact, in earlier C++ standards, the word *thread* isn't even mentioned). The basic type is `std::thread` that allows creating a thread. Other classes deal with thread synchronization (see chapter 7), and there are more.

The greatest benefit of using the C++ standard library is the fact that it's, well, standard, which means it's cross-platform. If this is more important than other considerations, then by all means, go ahead and use it. The downside of the C++ standard library compared to working with the Windows API is the very few customizations available. The C++ standard library does not support thread priorities, affinities, CPU sets (all discussed in chapter 6), stack size control, and so on. This level of control can only be achieved by using Windows-specific APIs.

Exercises

1. Create a WTL dialog-based application that allows counting prime numbers (add edit boxes for number input) in a range of numbers. Perform the calculation on a separate thread so the UI thread is not blocked.
2. Add a *Cancel* button to the dialog that allows cancellation of the prime numbers counting mid-flight.
3. Create a Console application to calculate the Mandelbrot set with multiple threads concurrently, so the calculation is quicker. (You can find more on the *Mandelbrot Set* on *Wikipedia*.) The number of threads should be an input to the application, as well as the dimensions of the output bitmap. Divide the total number of lines by the number of threads, and assign each thread to the range of lines. Each pixel should be 0 (belongs to the set) or 1 (does not belong to the set). Store the results in a two-dimensional array.
4. Extend the application by writing the output to a BMP or PPM format (both relatively simple), so the results can be viewed in a paint-like application.
5. Create a WTL application and calculate the Mandelbrot set with multiple threads without freezing the UI. Add the ability to pan/zoom and recalculate as needed.

Summary

In this chapter, we looked at the basics of thread creation and management. In the next chapter, we'll discuss thread scheduling and its associated properties, such as priority and affinity.

Chapter 6: Thread Scheduling

Threads are created to execute code, or at least that should be their intention. This means at some point a logical processor needs to run the thread's function. Generally, there are many threads on a typical system, but only a subset of them actually wants to execute code at the same time. Most threads are waiting for something and so are not candidates for being scheduled on processors at that time. If the number of threads that want to run (are in the *ready* state) is less or equal to the number of logical processors on the system (and there are no affinity restrictions, discussed later in this chapter), then all the ready threads simply execute.

However, some questions may come up. How long would threads get CPU time? What happens if a new thread wakes up? What about the case where there are more threads that are ready to run than there are available processors? We'll try to answer all these (and some other) questions in this chapter.

In this chapter:

- **Priorities**
- **Scheduling Basics**
- **Multiprocessor Scheduling**
- **Background Mode**
- **Processor Groups**
- **Suspending and Resuming**

Priorities

Every thread has an associated priority, that matters in the case where there are more threads that want to execute than there are available processors. In this section we'll look at the available priorities and how these can be manipulated, and the next section we'll see how these apply in scheduling.

Thread priorities are from 0 to 31, where 31 is the highest. Technically, thread 0 is reserved for a special thread called *zero page thread* that is part of the memory manager in the kernel. It's

the only thread that is allowed to have a priority of zero. So technically, usable priorities are from 1 to 31. In user mode (where we write code in this book), the priorities cannot be set to an arbitrary value. Instead, a thread's priority is a combination of a process' *Priority Class* (called *Base Priority* in *Task Manager*) and an offset around that base priority. Figure 6-1 shows *Task Manager* with the *Base Priority* column highlighted.

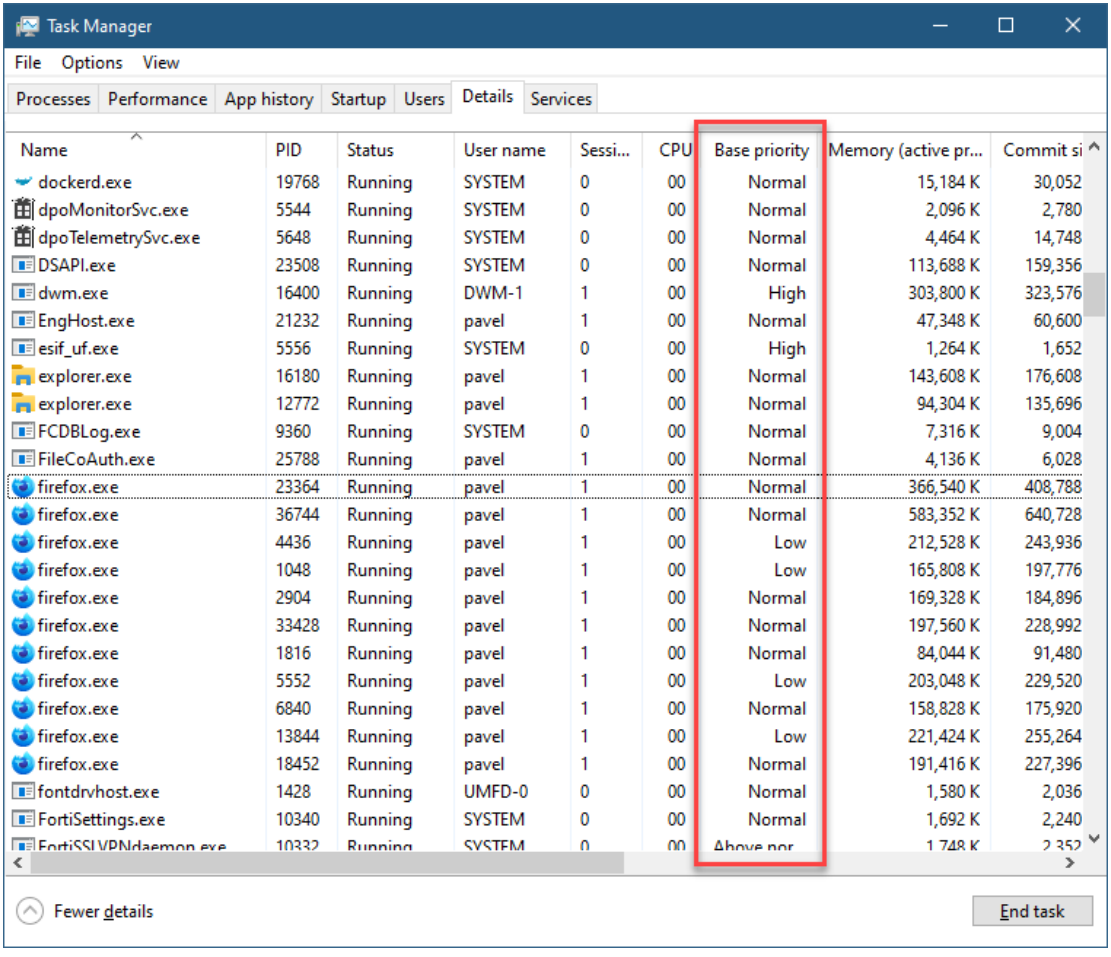


Figure 6-1: Base Priority in *Task Manager*

Each priority class is associated with a priority value, shown in table 6-1.

Table 6-1: Process priority classes

Priority Class	Priority value	API constant
Idle (Low)	4	IDLE_PRIORITY_CLASS
Below Normal	6	BELOW_NORMAL_PRIORITY_CLASS
Normal	8	NORMAL_PRIORITY_CLASS
Above Normal	10	ABOVE_NORMAL_PRIORITY_CLASS
High	13	HIGH_PRIORITY_CLASS
Real-time	24	REALTIME_PRIORITY_CLASS

> The name "real-time" in table 6-1 does not imply Windows is a real-time operating system; it's not. Windows cannot provide latency and timing guarantees that real-time operating systems do. This is because Windows works with a wide variety of hardware and so it's simply not possible to have any such guarantees when hardware range is unconstrained. "Real-time" in table 6-1 just means "higher than all the rest".

The priority class of a process can be set when that process is being created with `CreateProcess`. The sixth parameter (flags) can be combined with one of the constants in table 6-1. If no explicit priority class is specified, the priority class defaults to *Normal*, unless the creator's priority class is *Idle* or *Below Normal*, in which case the creator's priority class is used.

If a process already exists, changing the priority class is possible with `SetPriorityClass`:

```
BOOL SetPriorityClass(
  _In_ HANDLE hProcess,
  _In_ DWORD  dwPriorityClass);
```

> *Task Manager* as well as *Process Explorer* offer a context menu to change a process' priority class.

The process handle in question must have the `PROCESS_SET_INFROMATION` access mask for the call to succeed. Also, if the target priority class is Real-time, the caller must have the `SeIncreaseBasePriority` privilege. if it does not, the function does not fail, but the resulting priority class is set to *High* rather than *Real-time*.

Naturally, the opposite function exists as well to retrieve the priority class of a process:

```
DWORD GetPriorityClass(_In_ HANDLE hProcess);
```

The handle access mask needs only `PROCESS_QUERY_LIMITED_INFORMATION`, which can be obtained for almost all processes.

The process priority class has no direct effect on the process itself, since a process does not run - threads do. All threads created in a process have their default priority set to the priority class level. For example, in a *Normal* priority class process, all threads have a default priority of 8. To make changes to a thread's priority, the `SetThreadPriority` function can be used:

```
BOOL SetThreadPriority(
  _In_ HANDLE hThread,
  _In_ int    nPriority);
```

The *nPriority* parameter is not an absolute priority value. Instead, it's one of seven possible values (except for *Real-time* priority class, see next sidebar), listed in table 6-2.

Table 6-2: Thread relative priorities

Priority value	Effect
`THREAD_PRIORITY_IDLE` (-15)	priority drops to 1 except for real-time priority class, where thread priority drops to 16
`THREAD_PRIORITY_LOWSET` (-2)	priority drops by 2 relative to the priority class
`THREAD_PRIORITY_BELOW_NORMAL` (-1)	priority drops by 1 relative to the priority class
`THREAD_PRIORITY_NORMAL` (0)	priority set to the process priority class value
`THREAD_PRIORITY_ABOVE_NORMAL` (1)	priority increases by 1 relative to the priority class
`THREAD_PRIORITY_HIGHEST` (2)	priority increases by 2 relative to the priority class
`THREAD_PRIORITY_TIME_CRITICAL` (15)	priority increases to 15 except for the real-time priority class, where the thread priority increases to 31

The real-time priority class is special in relation to table 6-2. Threads in that priority class can be assigned any value from 16 to 31. `SetThreadPriority` accepts the values from -7 to -3 and from 3 to 7, corresponding to priorities 17 to 21 and 27 to 30.

The *High* priority class has only six levels. That's because threads in processes with priority class other than real-time cannot have their priorities above 15.

> The *idle* and *time critical* values are called *Saturation* values.

The net effect of table 6-1 and table 6-2 can be summarized in table 6-3 and figure 6-2. In figure 6-2, each rectangle signifies a possible thread priority value based on `SetPriorityClass/SetThreadPriority`. Table 6-3 is a summary table representation of tables 6-1 and 6-2 combined.

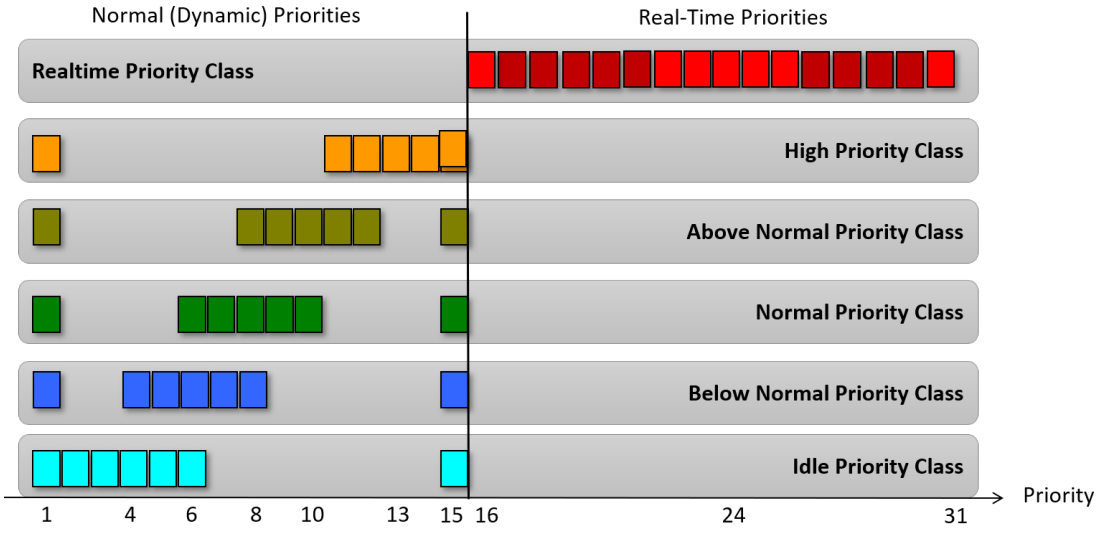


Figure 6-2: Thread Priorities

Table 6-3: Thread priorities by priority class

Priority Class (right) Relative Priority (down)	Real-Time	High	Above Normal	Normal	Below Normal	Idle
Time Critical (+15)	31	15	15	15	15	15
Highest (+2)	26	15	12	10	8	6
Above Normal (+1)	25	14	11	9	7	5
Normal (0)	24	13	10	8	6	4
Below Normal (-1)	23	12	9	7	5	3
Lowest (-2)	22	11	8	6	4	2
Idle (-15)	16	1	1	1	1	1

Table 6-3: Thread priorities by priority class

Priority Class (right) Relative Priority (down)	Real-Time	High	Above Normal	Normal	Below Normal	Idle

The resulting combination of process priority class and relative thread priority is the final thread's priority. From the kernel's scheduler perspective, only the final number is important. It doesn't care how that number came to be. For example, priority 8 can be reached in one of three ways: Normal priority class with normal (0) relative thread priority; Below normal priority class with highest (+2) relative thread priority; Above normal priority class with lowest (-2) relative thread priority. From the scheduler's perspective, they are all the same; it doesn't care about processes in general, only threads.

The real-time priority range (16-31) is used by many kernel threads that do essential work for the system as a whole, so it's important for processes whose threads run in that range not to consume too much CPU time. Of course, a process must have a very good reason to be in that range in the first place.

If we look at thread priorities in a tool such as *Process Explorer*, we find two values for the priority: *Base Priority* and *Dynamic priority* (figure 6-3).

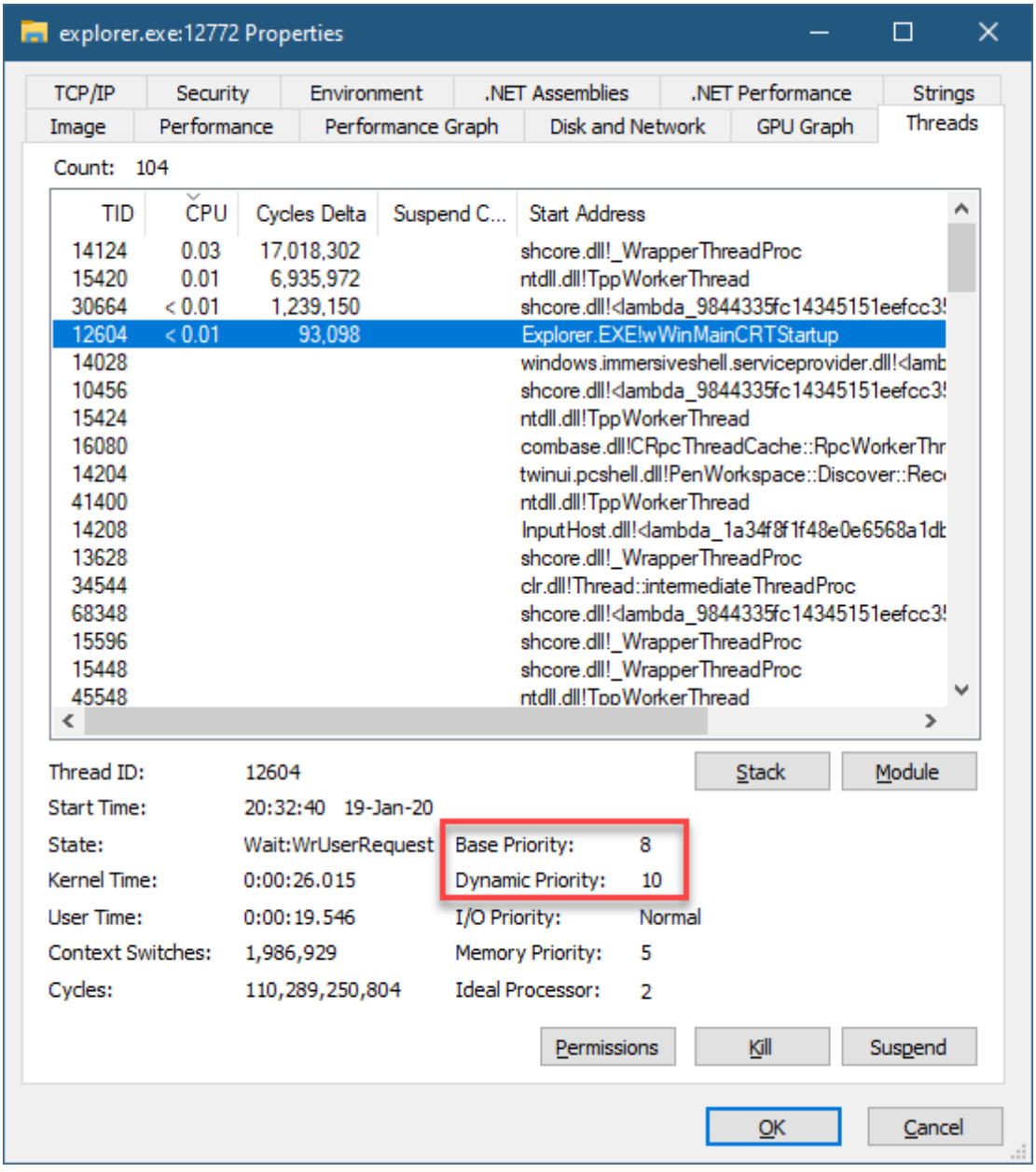


Figure 6-3: Thread properties in *Process Explorer*

The base priority is the one set by the developer (or the default value), while the dynamic priority is the actual current priority for that thread. In some cases the priority is increased (boosted) temporarily. We'll look at some of the reasons for this boost later in this chapter. From the scheduler's perspective, the dynamic priority is the determining priority value.

Threads in the real-time range never have their priority boosted.

Scheduling Basics

Scheduling in general is quite complex, taking into account several factors, some of them conflicting: multiple processors, power management (the desire to save power on the one hand and utilize all processors on the other hand), NUMA (Non-Uniform Memory Architecture), hyper-threading, caching, and more. The exact scheduling algorithms are undocumented for a reason: Microsoft can make modifications and tweaks in subsequent Windows versions and updates without developers taking any dependency on the exact algorithms. Having said that, it's possible to experience many of the scheduling algorithms by experimentation.

We'll start with the simplest scheduling possible - when there is a single processor on a system, since it's fundamental to how scheduling works. Later, we'll look at some of the ways these algorithms change on a multi-processing system.

Single CPU Scheduling

The scheduler maintains a *Ready Queue*, where threads that want to execute (are in the *Ready* state) are managed. All other threads that do not want to execute at this time (in the *Wait* state) are not being looked at since they don't want to execute. Figure 6-4 shows an example system where seven threads are in the *ready* state. They are arranged in a number of queues based on their priority in.

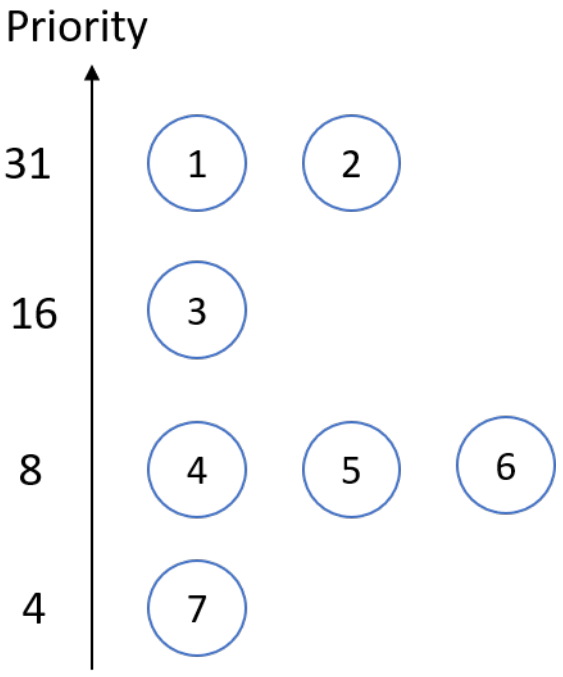


Figure 6-4: Threads in *Ready* state

> There may be thousands of threads on a system, but most are in the waiting state, so are not considered by the scheduler.

The algorithm for a single CPU goes like this:

1. The highest priority thread runs first. In figure 6-4, threads 1 and 2 have the highest (and same) priority (31), so the first thread in the queue for priority 31 runs; let's assume it's thread 1 (figure 6-5).

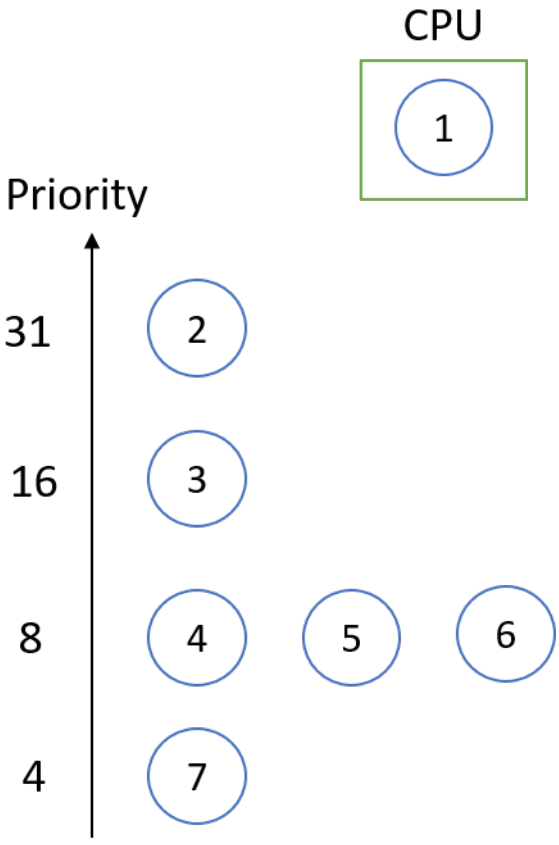


Figure 6-5: Highest priority thread runs

Thread 1 runs for a certain amount of time called a *Quantum*. The length of a quantum is discussed in the next section. Assuming thread 1 has lots to do, when its quantum expires, the scheduler *preempts* thread 1, saving its state in its kernel stack, and it goes back to the *Ready* state (since it still has things to do). Thread 2 now becomes the running thread since it has the same priority (figure 6-6).

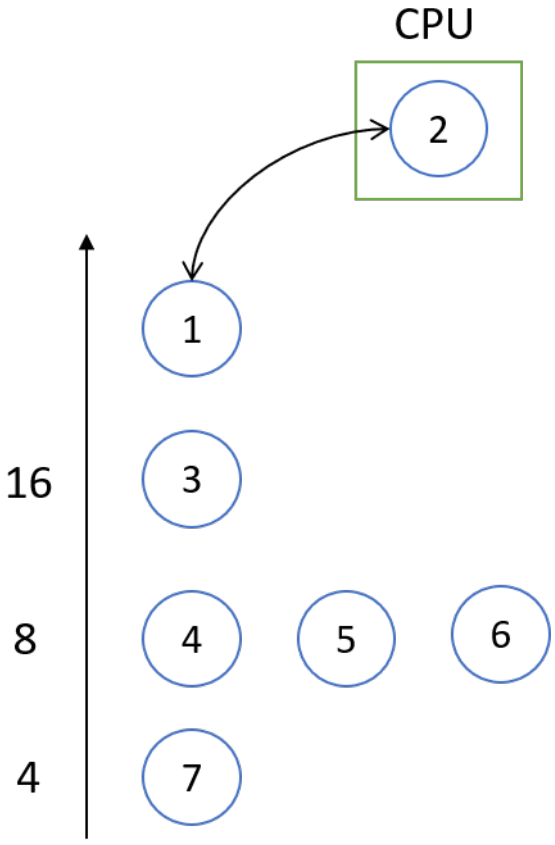


Figure 6-6: Thread 2 is running, thread 1 is back to ready

So, the priority is the determining factor. As long as threads 1 and 2 need to execute, they will round-robin on the CPU, each running for a *quantum*. Fortunately, threads typically don't run forever. Instead, they enter a wait state at some point. Here are a few examples that cause a thread to enter a wait state:

- Performing a synchronous I/O operation
- Waiting on a kernel object, that is currently not signaled
- Waiting for a UI message when there are none
- Entering a voluntary sleep

Once a thread enters *wait* state, it's removed from the scheduler's ready queue. Let's assume that threads 1 and 2 entered a wait state. Now the highest priority thread is thread 3 and it becomes the running thread (figure 6-7).

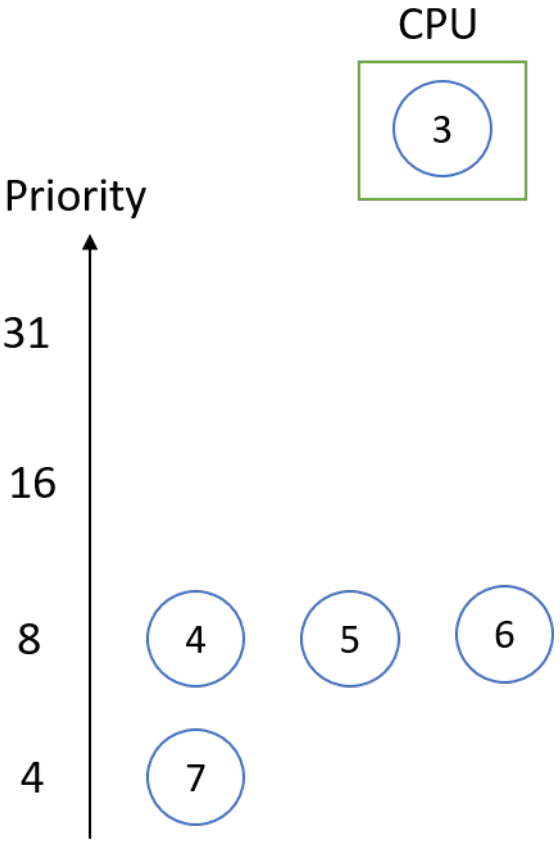


Figure 6-7: Thread 3 is running

Thread 3 runs for a quantum. If it still has work to do, it gets another quantum, since it's the only one in its priority level. If, however, thread 1 receives whatever it was waiting for, it goes to the ready state and preempts thread 3 (since thread 1 has a higher priority) and becomes the running thread. Thread 3 goes back to the ready state (figure 6-8). This switch is not at the end of thread 3's quantum, but rather at the time of the change (thread 1 finishing waiting). Thread 3's quantum is replenished if its priority is above 15 (which it is in this example).

If the preempted thread's priority is 16 or higher, it gets its quantum restored when it goes back to the ready state.

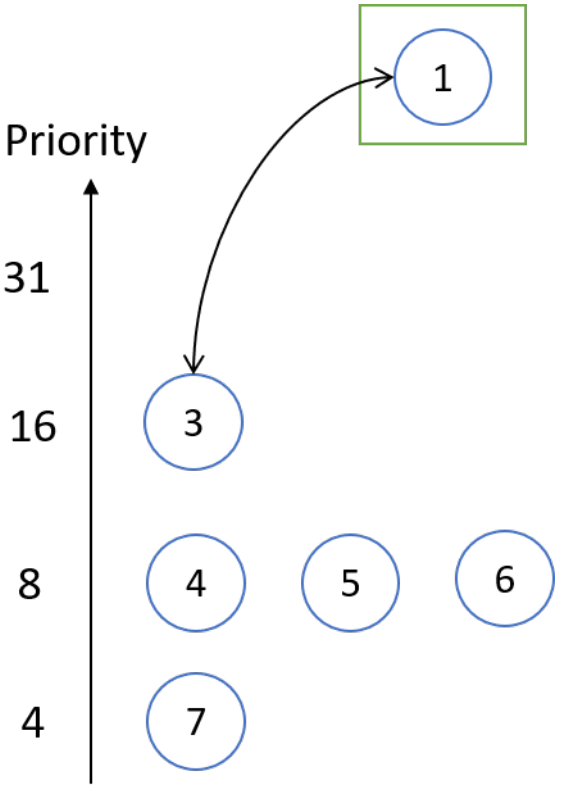


Figure 6-8: Thread 1 is running, thread 3 is back to ready

With this algorithm in mind, threads 4, 5 and 6 will run each having their own quantum if there are no higher-priority threads in the ready state.

This is the basis of scheduling. In fact, in a real single CPU scenario, this is exactly the algorithm used. However, even in this case, Windows tries to be "fair" to some extent. For example, thread 7 in figures 6-4 to 6-8 (with a priority of 4) may not run if higher-priority threads are in the ready state, and so it suffers from *CPU starvation*. Is that thread doomed in such a system? Not really; the system will boost the thread's priority to 15 every roughly 4 seconds, giving it a better chance to make forward progress. This boost lasts for one quantum of actual execution of the thread, and then the priority drops back to its initial value. This is just one example of a temporary priority boost. You'll see other examples in the section "Priority Boosts" later in this chapter.

The Quantum

The quantum was mentioned a few times in the preceding section - but how long is a quantum? The scheduler works in two orthogonal ways: the first is with a timer, that fires every 15.625 milliseconds by default, and can be obtained by calling `GetSystemTimeAdjustment` and looking at the second argument. Another way is to use the *clockres* tool from *SysInternals*:

```
C:\Users\pavel>clockres

Clockres v2.1 - Clock resolution display utility
Copyright (C) 2016 Mark Russinovich
Sysinternals

Maximum timer interval: 15.625 ms
Minimum timer interval: 0.500 ms
Current timer interval: 1.000 ms
```

The value to look at in relation to the quantum is the *Maximum time interval* value.

> The *Current time interval* from *clockres* shows the current timer firing interval. This is typically lower than the maximum interval because of multimedia timers that may have been requested. This allows getting timer notifications up to a 1 msec resolution. Regardless, the quantum itself is not affected by the Current timer interval.

The default quantum is 2 clock ticks for client machines (Home, Pro, Enterprise, XBOX, etc.) and 12 clock ticks for server machines. In other words, the quantum is 31.25 msec for client and 187.5 msec for server.

The reason server versions get a longer quantum is to increase the chances of a client request being handled completely in a single quantum. This is less of a concern on a client machine since it may have many processes doing relatively little work each, some with user interface that should be responsive, so short quantums better fit the bill. Switching from client to server (or vice versa) in terms of quantum can be achieved with the following dialog which I like to call "the most incomprehensible dialog in Windows" (figure 6-9).

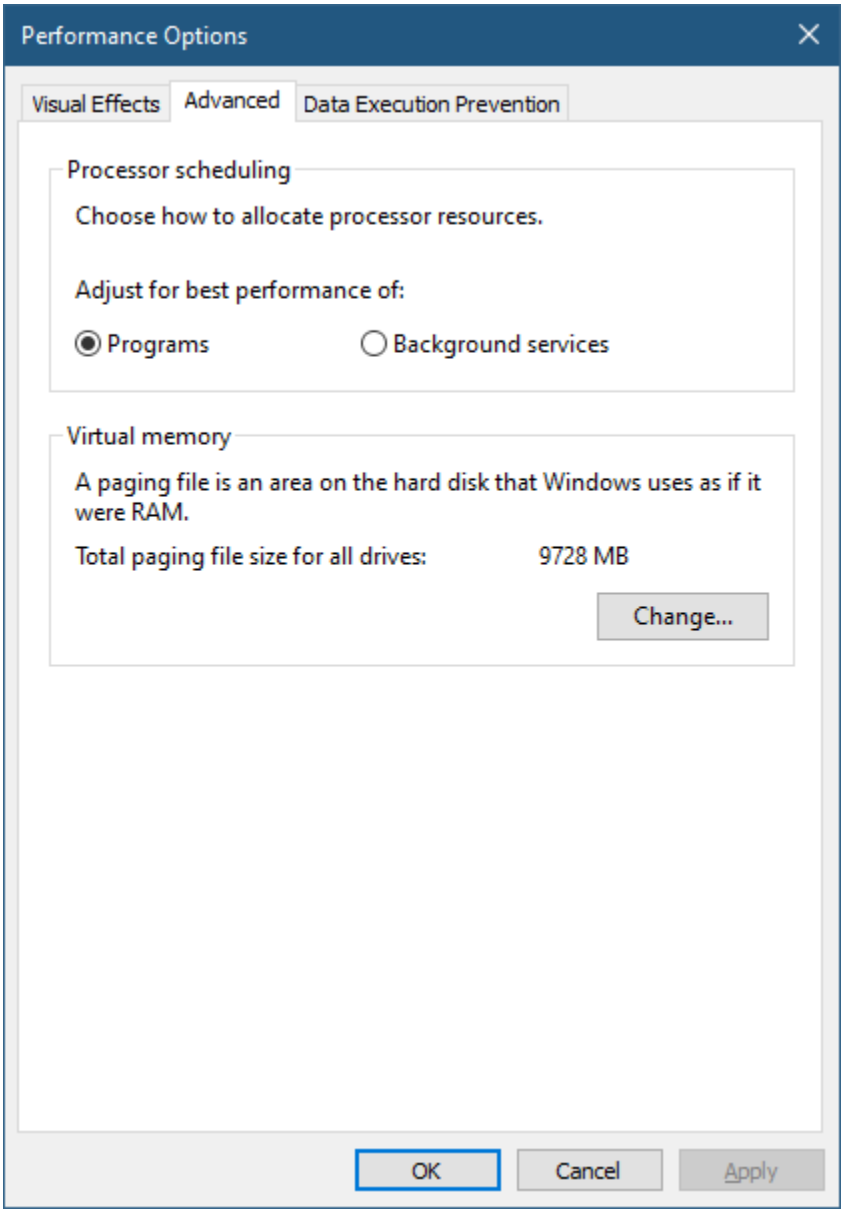


Figure 6-9: Performance Options dialog

Here is how to get to this dialog: go to System properties (from the Control Panel or right-click *This PC* in *Explorer* and select *Properties*). Then click *Advanced System Settings*. A dialog opens - click the Performance *Settings* button. The Performance Options dialog appears. Click the *Advanced* tab, and you're there.

This dialog has two parts, which are completely unrelated. The lower part controls the size of page files (if any), discussed later in this book. The upper part is the "incomprehensible" part. The option *Programs* means short quantums, while the option *Background Services* means long quantums. If you check the other radio button, the change is instantaneous.

There is yet another difference between the two options. *Programs* also means that in the foreground process (the one hosting the foreground window), all threads get triple quantum by default. This quantum stretching effect is not present for *Background Services* because a true server is unlikely to have an interactive user sitting at the console. Also, some of the priority boosts we'll look at later in this chapter do not apply when *Background Services* is selected.

There are other ways to change the quantum. For fine-grained control, the registry value *HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation* controls not just the length of a quantum, but also quantum stretching for the foreground process. See the "Windows Internals" book (chapter 4) for the details. It's best to leave this value at its default, where the rules previously described apply.

Another way to control the quantum is available by using a job object (described in detail in chapter 4) with the `SchedulingClass` field in `JOBOBJECT_BASIC_LIMIT_INFORMATION`. This works for long fixed quantum systems only (the default for server systems). The scheduling class value (between 0 and 9) sets the quantum for threads in processes that are part of the job in question in the following way:

```
Quantum = 2 * (timer interval) * (Scheduling class + 1);
```

The default scheduling class is 5, effectively giving a quantum of `12 * timer interval`, which is the default on server systems, as we saw earlier. The highest value (9) causes the threads to be non-preemptive, meaning they have an infinite quantum (can continue running theoretically indefinitely until they voluntarily enter a wait state). A value higher than 5 requires the caller to have the *SeIncreaseBasePriority* privilege, which by default is available for users in the Administrators group, but not to standard users.

The scheduling class value only applies for processes with priority classes higher than *Idle*.

Processor Groups

The original Windows NT design supported 32 processors at most, with a machine word (32 bit) used to indicate the actual processors on a system, with each bit representing a processor. When 64 bit Windows appeared, the maximum number of processors was naturally extended to 64.

Starting with Windows 7 (64 bit systems only), Microsoft wanted to support more than 64 processors, and so an additional parameter entered the scene: a *processor group*. For example, Windows 7 and Server 2008 R2 support up to 256 processors, meaning there are 4 processor groups on a system with 256 processors.

Windows 8 and Server 2012 support 640 processors (10 groups) and Windows 10 supports even more. The basic rule remains - at most 64 processors per group - the number of groups increases as needed.

A thread can be a member of one processor group, which means a thread can be scheduled on one of (at most) 64 processors that are part of its current group. When a process is created, it's assigned a processor group in a round-robin fashion, so that processes are "load balanced" across groups. Threads in a process are assigned to the process group. A parent process can affect the initial processor group of a child process in one of the following ways:

1. The parent process can use the `INHERIT_PARENT_AFFINITY` flag as one of the flags to `CreateProcess` to indicate the child process should inherit its parent processor group rather than getting it according to the round robin managed by the system. If the parent process' threads use more than one affinity group, one of the groups is selected arbitrarily as the one used for the child process group.
2. The parent process can use the `PROC_THREAD_ATTRIBUTE_GROUP_AFFINITY` process attribute to specify a desired default processor group.

The process group affinity can be retrieved with `GetProcessGroupAffinity`:

```
BOOL GetProcessGroupAffinity(
  _In_    HANDLE  hProcess,
  _Inout_ PUSHORT GroupCount,
  _Out_   PUSHORT GroupArray);
```

Controlling the processor group of a specific thread is possible with the `SetThreadGroupAffinity` function. This is discussed in the section "Hard Affinity" later in this chapter.

Multiprocessor Scheduling

Multiprocessor scheduling adds complexity to the scheduling algorithms. The only thing Windows guarantees is that the highest priority thread that wants to execute (at least one if there's more than one) is currently running. In this section, we'll take a look at some of the parameters that affect scheduling.

Affinity

Normally, a thread may be scheduled on any processor. However, the *affinity* of a thread, that is, the processors it's allowed to run on can be controlled in several ways, described in the followings sections.

Ideal Processor

The *Ideal Processor* is an attribute of a thread, also sometimes referred to as "Soft Affinity". An ideal processor serves as a hint to the scheduler as - all other things being equal - the preferred processor for executing code for that thread. The default ideal processor is selected in a round-robin fashion, starting with a random generated when the process is created. On a hyper-threaded system, the next ideal processor is selected from the next core, rather than from the next logical processor.

The ideal processor can be viewed with the *Process Explorer* tool, as one of the properties shown in the *Threads* tab (figure 6-10).

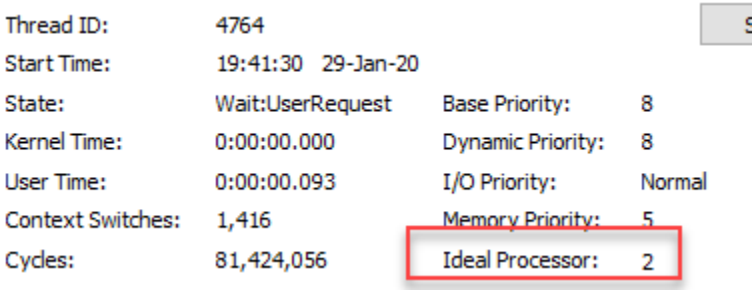


Figure 6-10: Thread's Ideal processor in *Process Explorer*

A thread's ideal processor can be changed with `SetThreadIdealProcessor`:

```
DWORD WINAPI SetThreadIdealProcessor(
  _In_ HANDLE hThread,
  _In_ DWORD  dwIdealProcessor);
```

The ideal processor changed by this function is between 0 and the maximum number of processors minus 1, where 63 is the maximum, since this is the highest processor number in any group. If more than one group is supported on the system, the current thread's group is used. The function returns the previous ideal processor number or -1 (`0xffffffff`) in case of an error. Passing the special value `MAXIMUM_PROCESSORS` (which equals 32 on 32 bit systems and 64 on 64 bit systems) for the ideal processor just returns the current ideal processor.

`SetThreadIdealProcessor` changes the ideal processor for the current processor group the thread is part of. To make the change for a different group, the extended `SetThreadIdealProcessorEx` function can be used:

```
typedef struct _PROCESSOR_NUMBER {
  WORD Group;
  BYTE Number;
  BYTE Reserved;
} PROCESSOR_NUMBER, *PPROCESSOR_NUMBER;

BOOL SetThreadIdealProcessorEx(
  _In_      HANDLE            hThread,
  _In_      PPROCESSOR_NUMBER lpIdealProcessor,
  _Out_opt_ PPROCESSOR_NUMBER lpPreviousIdealProcessor);
```

The `PROCESSOR_NUMBER`structure's `Group` member is the group to set the ideal processor in, and the `Number` member is the CPU index (0 to 63). As with the non-Ex function, the previous ideal processor can be retrieved using the last optional parameter.

Hard Affinity

While the ideal processor serves as a hint and a recommendation on which processor a thread should execute, *hard affinity* (sometimes called just *affinity*) allows specifying the allowed processors to execute on for a particular thread or process. Hard affinity works on two levels: process and thread, where the basic rule is that a thread cannot "escape" the affinity set by its process.

Generally speaking, setting hard affinity constraints is usually a bad idea. It limits the scheduler's freedom in assigning processors and can cause threads to get less CPU time than if they had no hard affinity constraints. Still, in some rare occasions this can be useful, as threads that run on the same set of processors are more likely to get better CPU cache utilization. This could be useful for systems that run specific known processes, rather than some random machine that may be running anything. Another use of hard affinity is in stress testing, such as using fewer processors for some execution to see how a system with that restricted number of processors might behave when running the same process(es).

Setting a process-wide hard affinity is achieved with `SetProcessAffinityMask`:

```
BOOL WINAPI SetProcessAffinityMask(
  _In_ HANDLE    hProcess,
  _In_ DWORD_PTR dwProcessAffinityMask);
```

The process handle must have the `PROCESS_SET_INFORMATION` access mask. The affinity mask itsef is a bit mask where a bit set to one indicates allowed processor and a zero bit indicates a forbidden processor. For example, the affinity mask `0x1a` (`11010` in binary) indicates processors 1, 3 and 4 are the only ones allowed. The function changes the affinity mask for the current process processor group.

Task Manager and *Process Explorer* allow changing a process affinity mask. For *Task Manager*, right clicking a process in the *Details* tab and selecting *Set affinity* shows a dialog with the available processors for the current process processor group (system affinity mask) (figure 6-11). Clicking *OK* calls `SetProcessAffinityMask` to set the new affinity mask. *Process Explorer* has similar functionality.

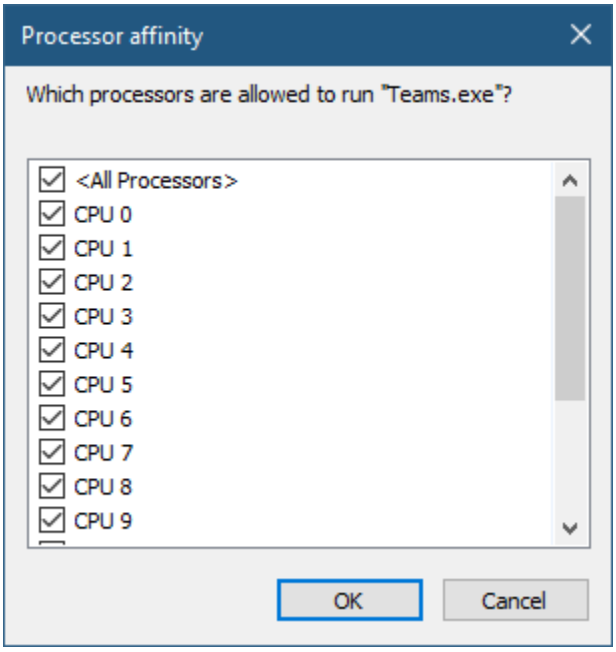


Figure 6-11: Setting hard affinity in *Task Manager*

Naturally, the inverse function is available as well, which also provides the system affinity mask:

```
BOOL WINAPI GetProcessAffinityMask(
  _In_  HANDLE     hProcess,
  _Out_ PDWORD_PTR lpProcessAffinityMask,
  _Out_ PDWORD_PTR lpSystemAffinityMask);
```

Here is an example for getting the current process affinity mask, perhaps just to retrieve the system affinity mask:

```
DWORD_PTR processAffinity, systemAffinity;
::GetProcessAffinityMask(::GetCurrentProcess(), &processAffinity,
  &systemAffinity);
```

As an example, on a system with 16 logical processors, the returned system affinity mask is `0xffff`.

Setting the process affinity mask constraints all the threads in the process to use that mask. An individual thread can further restrict its affinity mask by calling `SetThreadAffinityMask`:

```
DWORD_PTR WINAPI SetThreadAffinityMask(
  _In_ HANDLE    hThread,
  _In_ DWORD_PTR dwThreadAffinityMask);
```

The function sets the thread affinity mask, if possible. Remember the basic rule: a thread's affinity mask cannot include processors not specified in its process affinity mask. The return value is the previous affinity mask of the thread, or zero if an error occurs.

On a system with more than 64 processors, a thread may change its processor group while specifying an affinity mask with `SetThreadGroupAffinity`:

```
typedef struct _GROUP_AFFINITY {
  KAFFINITY Mask;       // affinity bit mask
  WORD      Group;      // group number
  WORD      Reserved[3];
} GROUP_AFFINITY, *PGROUP_AFFINITY;

BOOL SetThreadGroupAffinity(
  HANDLE               hThread,
  const GROUP_AFFINITY *GroupAffinity,
  PGROUP_AFFINITY      PreviousGroupAffinity);
```

The function can do two things: change the processor group for the specified thread and/or change the hard affinity mask in that group. If the group is changed, it becomes the default processor group for the thread's process. This complicates things, so it's usually better to make sure all threads in a process are part of the same group. Still, if more than 64 threads may be running concurrently in that process (and there are more than 64 processors on the system), then changing the processor group of some threads can be beneficial, as they can utilize processors from another group.

`KAFFINITY` is typedefed as `ULONG_PTR`.

As you might expect, the inverse function is available as well:

```
BOOL GetThreadGroupAffinity(
  HANDLE          hThread,
  PGROUP_AFFINITY GroupAffinity);
```

CPU Sets

As we've seen in the previous section, a thread's affinity cannot "escape" its process affinity. However, there are some scenarios where it's beneficial to have a thread (or threads) use processors that other threads in the process are forbidden to use. Windows 10 and Server 2016 added this capability, known as *CPU Sets*.

The term "CPU Set" indicates an abstract view of processors, where each CPU set is potentially mapped to one or more logical processors. Currently, however, each CPU set maps to exactly one logical processor. The system has its own CPU sets, which by default includes all processors on a system. This information is available with `GetSystemCpuSetInformation`:

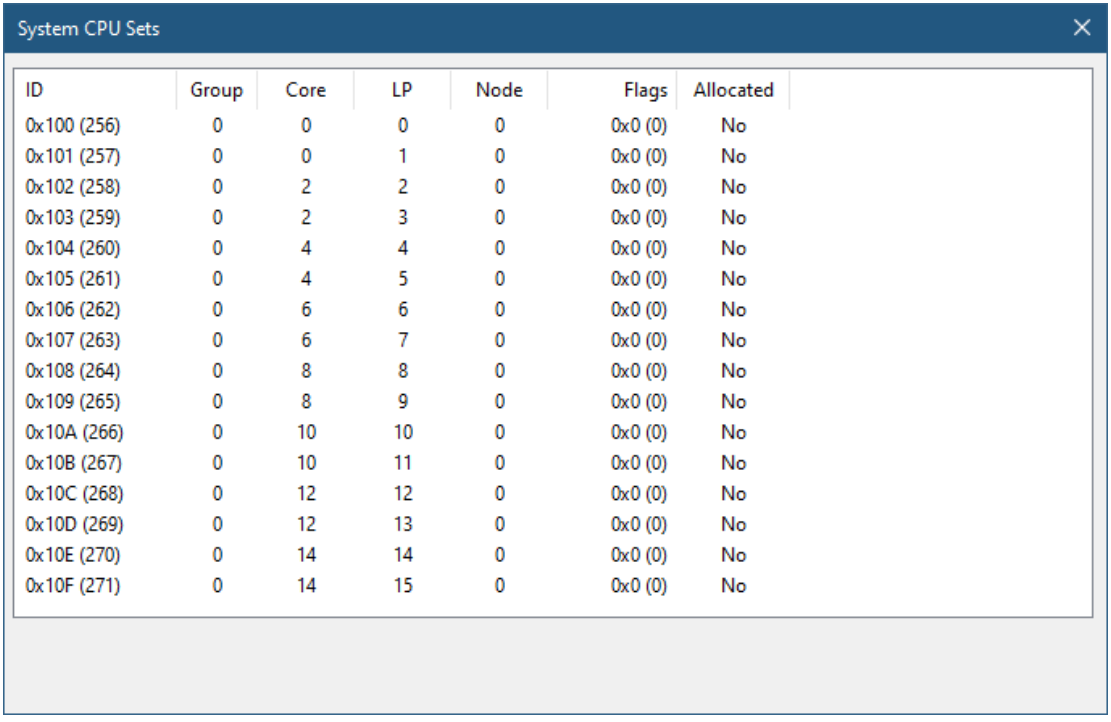
```
BOOL WINAPI GetSystemCpuSetInformation(
  _Out_opt_  PSYSTEM_CPU_SET_INFORMATION  Information,
  _In_       ULONG                        BufferLength,
  _Out_      PULONG                       ReturnedLength,
  _In_opt_   HANDLE                       Process,
  _Reserved_ ULONG                        Flags);
```

The function returns an array of structures of type `SYSTEM_CPU_SET_INFORMATION` defined like so:

```
typedef struct _SYSTEM_CPU_SET_INFORMATION {
  DWORD                    Size;
  CPU_SET_INFORMATION_TYPE Type;     // currently
  union {
    struct {
      DWORD   Id;
      WORD    Group;
      BYTE    LogicalProcessorIndex;
      BYTE    CoreIndex;
      BYTE    LastLevelCacheIndex;
      BYTE    NumaNodeIndex;
      BYTE    EfficiencyClass;
      union {
        BYTE AllFlags;
        struct {
          BYTE Parked : 1;
          BYTE Allocated : 1;
          BYTE AllocatedToTargetProcess : 1;
          BYTE RealTime : 1;
          BYTE ReservedFlags : 4;
```

```
        };
      };
      union {
        DWORD Reserved;
        BYTE  SchedulingClass;
      };
      DWORD64 AllocationTag;
    } CpuSet;
  };
} SYSTEM_CPU_SET_INFORMATION, *PSYSTEM_CPU_SET_INFORMATION;
```

To get a sense of some of these values, run *CPUStress* and select the *System / CPU Sets...* menu item. The system CPU set should be displayed. Figure 6-12 shows the output for a system with 1 socket, 8 cores and 16 logical processors.



ID	Group	Core	LP	Node	Flags	Allocated
0x100 (256)	0	0	0	0	0x0 (0)	No
0x101 (257)	0	0	1	0	0x0 (0)	No
0x102 (258)	0	2	2	0	0x0 (0)	No
0x103 (259)	0	2	3	0	0x0 (0)	No
0x104 (260)	0	4	4	0	0x0 (0)	No
0x105 (261)	0	4	5	0	0x0 (0)	No
0x106 (262)	0	6	6	0	0x0 (0)	No
0x107 (263)	0	6	7	0	0x0 (0)	No
0x108 (264)	0	8	8	0	0x0 (0)	No
0x109 (265)	0	8	9	0	0x0 (0)	No
0x10A (266)	0	10	10	0	0x0 (0)	No
0x10B (267)	0	10	11	0	0x0 (0)	No
0x10C (268)	0	12	12	0	0x0 (0)	No
0x10D (269)	0	12	13	0	0x0 (0)	No
0x10E (270)	0	14	14	0	0x0 (0)	No
0x10F (271)	0	14	15	0	0x0 (0)	No

Figure 6-12: System CPU Sets in *CPUStress*

> Currently, *Task Manager* and *Process Explorer* don't provide information on CPU sets.

The *ID* in figure 6-12 corresponds to the `CpuSet.Id` member in `SYSTEM_CPU_SET_INFORMATION`. This is an abstract value for the ID of the CPU set itself. The first CPU set currently starts at 256 (0x100) and is incremented by one for each additional CPU set. This value of 256 was arbitrarily chosen and does not mean anything by itself. These IDs are needed, however, to change CPU sets for processes and threads, described in the following paragraphs.

The *Group* column in figure 6-12 corresponds to a process group (`CpuSet.Group` in the structure above). The *Core* column indicates the core for that CPU set (`CpuSet.CoreIndex`). Normally this is the real core number (the exact definition of this member is more elaborate and theoretical - check the documentation if interested). The *LP* column shows the logical processor number for that CPU set (`CpuSet.LogicalProcessorIndex`). The *Node* column shows the NUMA node for the CPU set (`CpuSet.NumaNodeIndex`).

> For more details on other members of `SYSTEM_CPU_SET_INFORMATION`, consult the documentation.

A process can set a default CPU sets for its threads with `SetProcessDefaultCpuSets`:

```
BOOL WINAPI SetProcessDefaultCpuSets(
  _In_     HANDLE       Process,
  _In_opt_ const ULONG* CpuSetIds,
  _In_     ULONG        CpuSetIdCound);
```

The `CPUSetIds` array is expected to contain the CPU set Ids, available in the `CpuSet.Id` member of `SYSTEM_CPU_SET_INFORMATION`. A value of `NULL` removes the current CPU set assignment, meaning the CPU set constraints are removed from threads in the process that don't have specific selected CPU sets. A thread can select specific CPU sets, that may be different than its process assignment with `SetThreadSelectedCpuSets`:

```
BOOL WINAPI SetThreadSelectedCpuSets(
  _In_     HANDLE       Thread,
  _In_opt_ const ULONG* CpuSetIds,
  _In_     ULONG        CpuSetIdCount);
```

Here is an example using these functions:

```
ULONG sets[] = { 0x100, 0x101, 0x102, 0x103 };
::SetProcessDefaultCpuSets(::GetCurrentProcess(), sets, _countof(sets));
ULONG tset[] = { 0x104 };
::SetThreadSelectedCpuSets(::GetCurrentThread(), tset, _countof(tset));
```

The preceding example causes all the threads in the process to use CPU sets 0x100 to 0x103 by default, except for the current thread, which uses CPU set 0x104, essentially "escaping" its parent process CPU sets. This could be useful where a thread should have its own CPU, where other threads in the process cannot use.

CPU Sets vs. Hard Affinity

CPU sets and hard affinity may conflict with each other. In such a case, hard affinity always wins. If CPU sets contradicts hard affinity, CPU sets are ignored.

System CPU Sets

The system has its own CPU set, as can be determined with `GetSystemCpuSetInformation`, which normally returns the CPU sets available on the system. The Windows API does not provide a documented way to change it, but it is possible to do with the native `NtSetSystemInformation` call. This allows telling the "system" to avoid certain processors, so as not to interfere with user processes. This capability is used in *Game Mode*, available in Windows 10 version 1703 and later.

> A detailed discussion of Game Mode is beyond the scope of this book.

Revised Scheduling Algorithm

Multiprocessor (MP) scheduling is complex: Hard affinity, ideal processor, CPU sets, power considerations, Game Mode, and other aspects all make scheduling decisions on MP complicated, to say the least. The ready queue (actually an array of 32 queues, one per priority) described in the section "Scheduling Basics" is extended on MP systems: each processor has its own ready queue. Additionally, on Windows 8 and later, there are shared ready queues for groups of processors (currently a maximum of 4 per group). This allows the scheduler to have more options when it needs to locate a processor for a ready thread that is attached to a shared ready queue (per-CPU ready queues are still used for threads that have hard affinity constraints).

> the above details can be changed by Microsoft in future versions of Windows. The text is there to give some ideas on the complexity of scheduling.

A revised, simplified, MP scheduling algorithm is presented in figure 6-13. It assumes no affinity or CPU set constraints, no power or other special considerations.

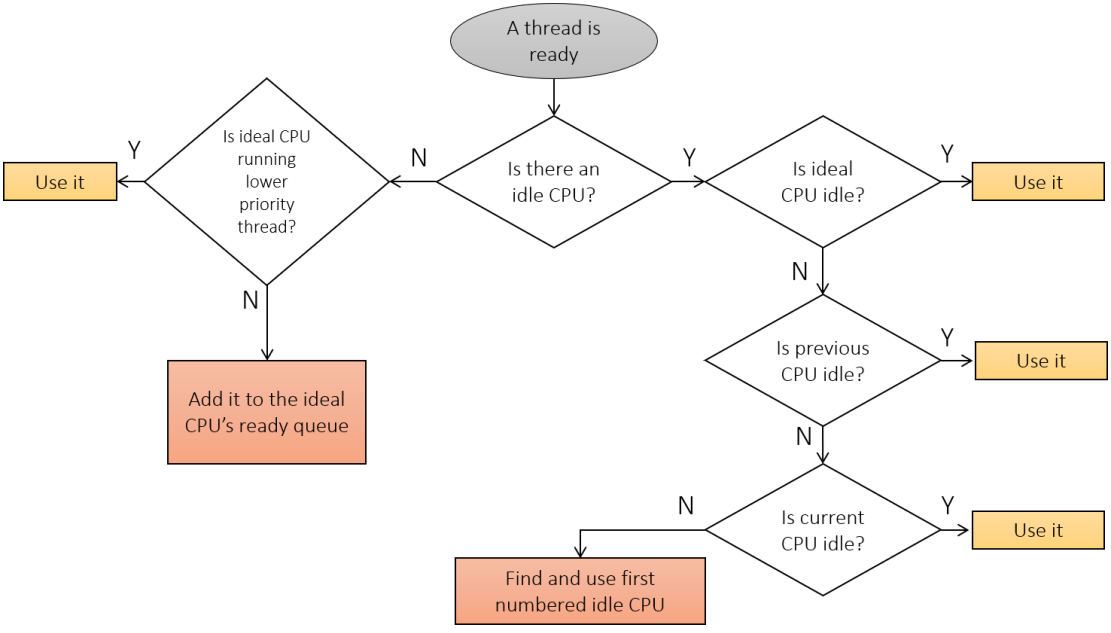


Figure 6-13: MP simplified scheduling

As can be seen in figure 6-13, the ideal processor is the preferred processor to use, followed by the last processor it ran on (processor's cache may still contain data used by that thread). If all processors are busy, the scheduler does not preempt the first processor that runs a low-priority thread; that would be inefficient, as many processors may be needed to be searched. Instead, the thread is put in the (shared) ready queue of its ideal processor.

The next Windows version, 2004 (April 2020) may change the order of check between ideal processor and previous processor. Regardless, from a developer's perspective, it matters very little.

Observing Scheduling

Scheduling changes are fairly frequent, but they can be observed by using tools. In the following sections I'll describe some experiments you can use to investigate and experience scheduling. This section is optional and can be safely skipped.

General Scheduling

To get a general sense of scheduling we, can use *Performance Monitor*. Run *CPUStress* and terminate two threads so that only two remains. Activate both threads (figure 6-14).

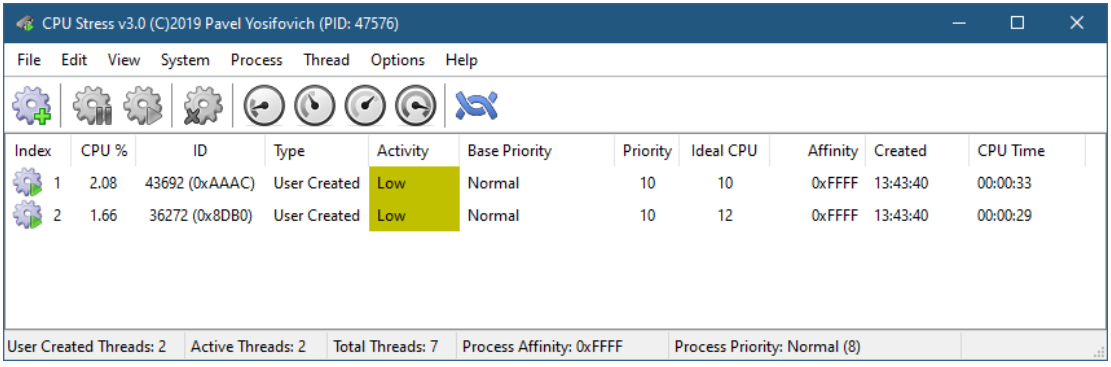


Figure 6-14: *CPUStress* with two active threads

Now open *Performance Monitor* (type *perfmon* at Run prompt or just search for it). The *Performance* console appears. Click on the *Performance Monitor* item (figure 6-15).

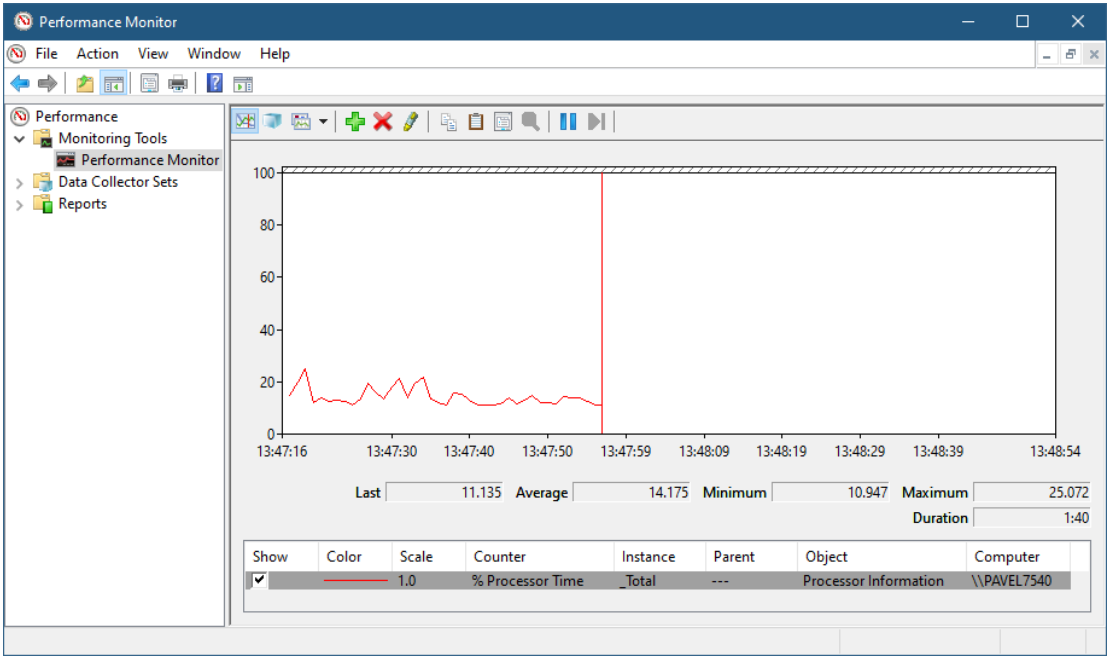


Figure 6-15: ***Performance Monitor***

> *Performance Monitor* is a built-in tool that can show *performance counters*, which are just numbers, exposed by various system components. technically, any application can register and expose performance counters. In the following examples, we use some of these built-in counters that are related to scheduling.

Delete the default counter, and Click the *Add* (green plus button) to add new counters. Search for the *Thread* category and open it (figure 6-16).

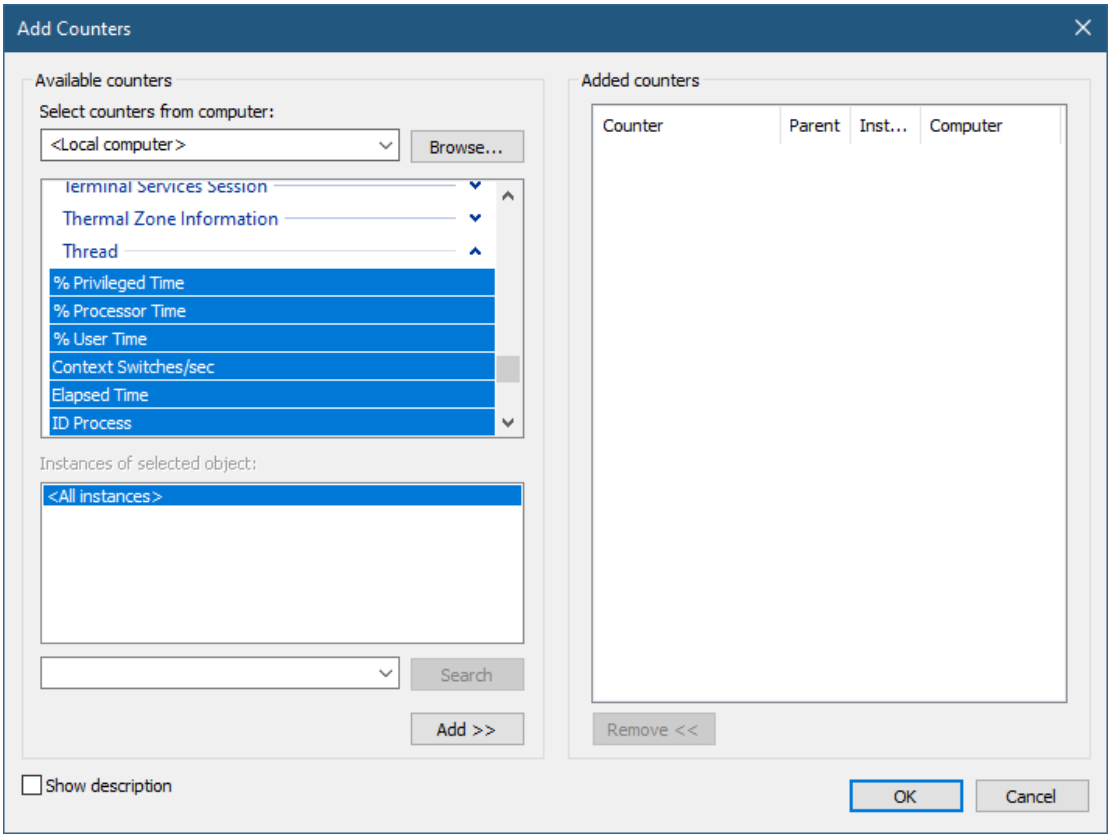


Figure 6-16: The *Thread* performance counter category

Now select the following counters: *Priority Current* and *Thread State* (use the Control key to multi-select) (figure 6-17).

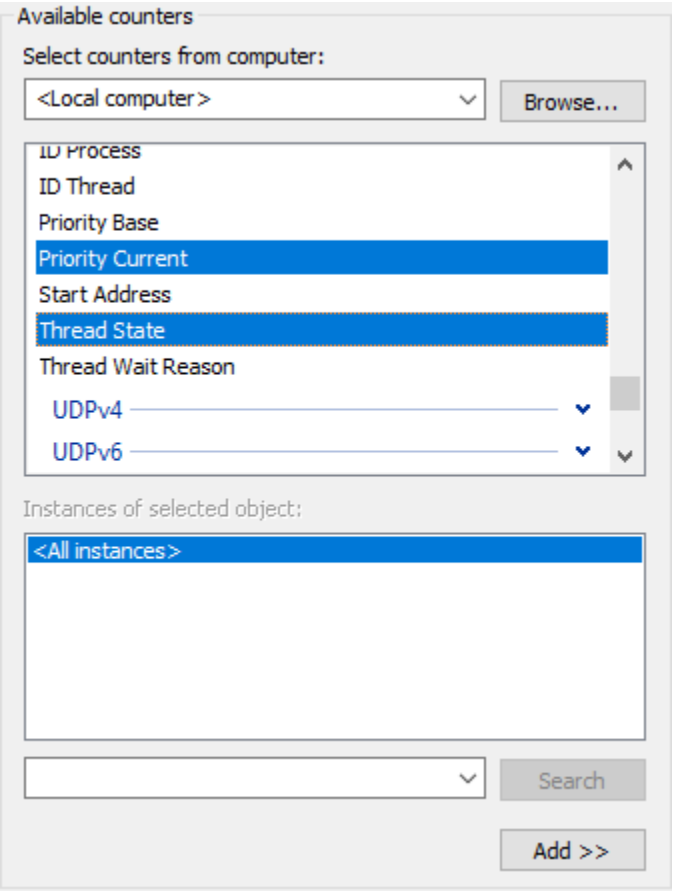


Figure 6-17: Counters selected from the *Thread* category

In the lower search box, type *CPustress* and press ENTER. A list of threads should show up. Select the thread 1 and 2 (these should be the numbers shown to left in *CPUStress*). Click *Add* (figure 6-18) and then *OK*.

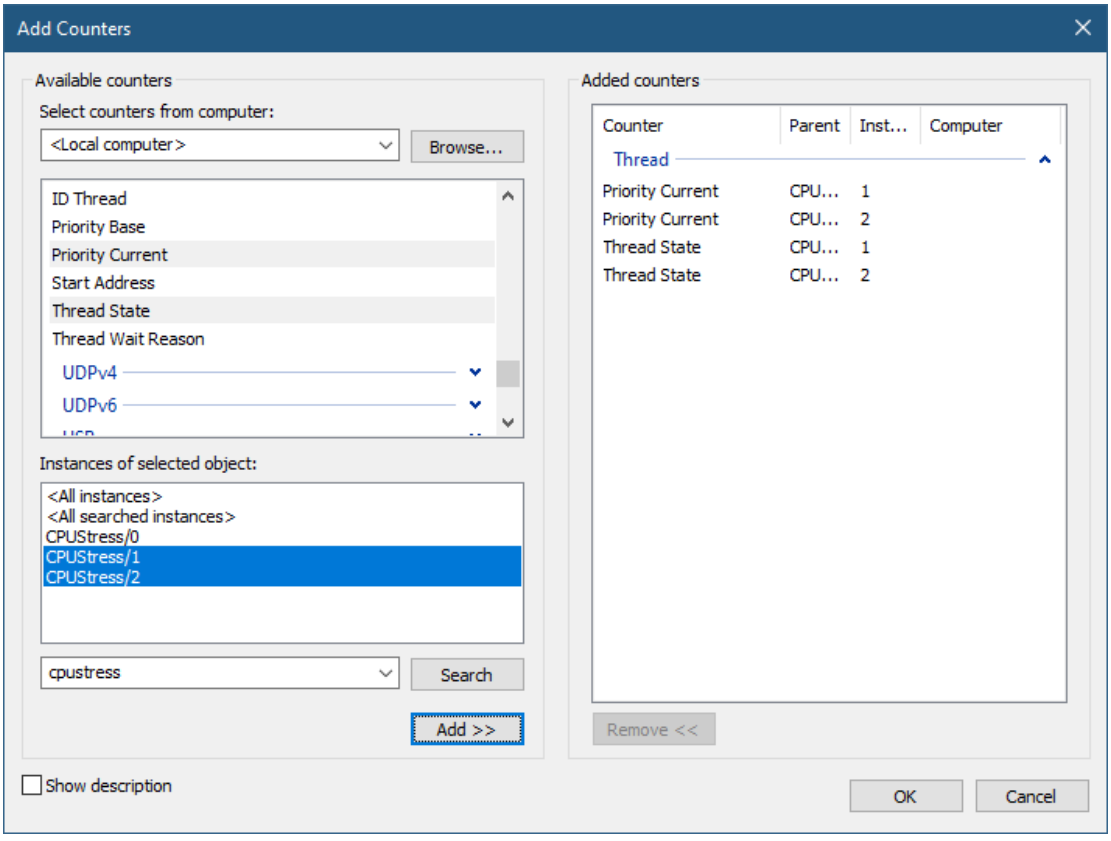


Figure 6-18: Two threads selected in *Threads* category

4 graphs are now shown, with the current priority and states of the two worker threads from *CPUStress*. Right click an empty graph region and select *Properties*. Switch to the *Graph* tab and change the vertical scale to be between 0 and 16 (figure 6-19). Click *OK*.

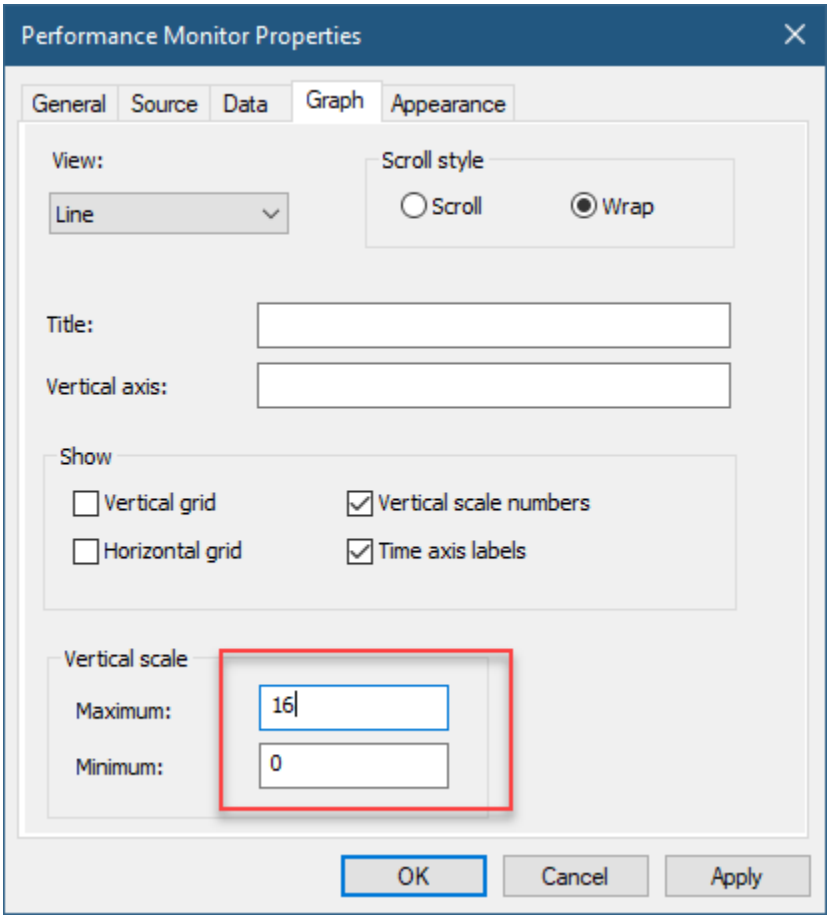


Figure 6-19: Changing graph scale

Now the priorities and states should be more easily identifiable (figure 6-20).

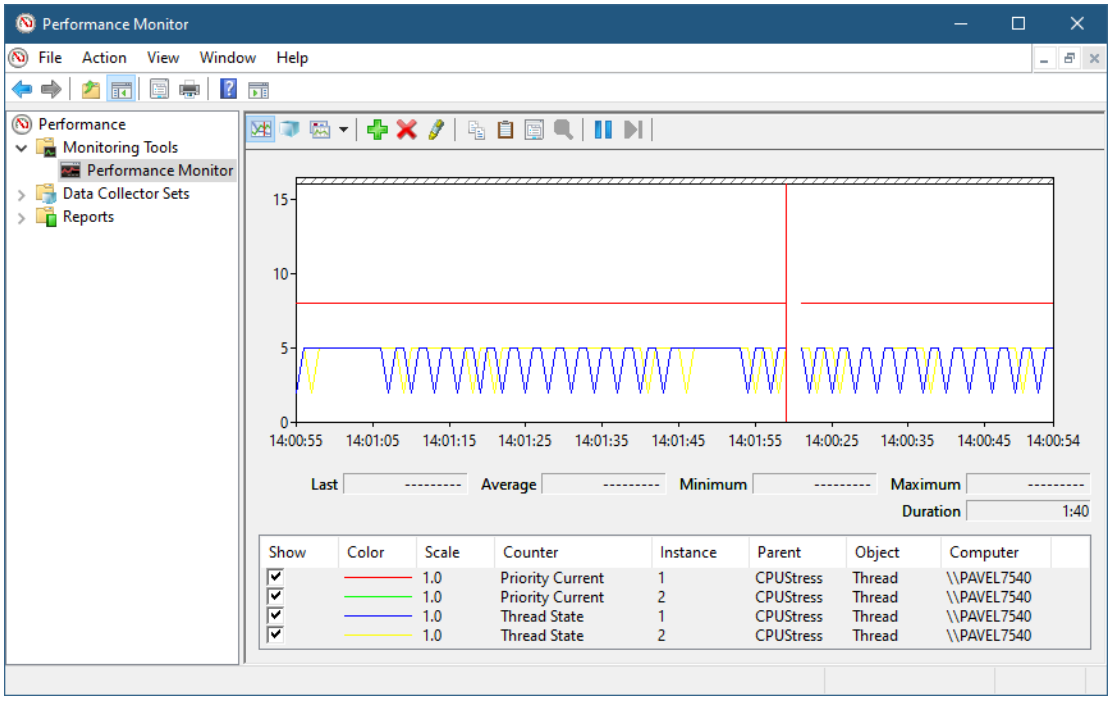


Figure 6-20: The counters at work

Notice the priority of the threads is 8 (red and green lines in figure 6-20, green is obscured by the red). The thread states alternates between 2 and 5. Here is the main state numbers: Running=2, Ready=1, Waiting=5. Now switch to the *CPUStress* application. Notice the thread priorities jump to 10. If you switch to another application, it drops back down. You can change the activity level of one thread to *high* and see the effect. (figure 6-21). You can play with priorities as well.

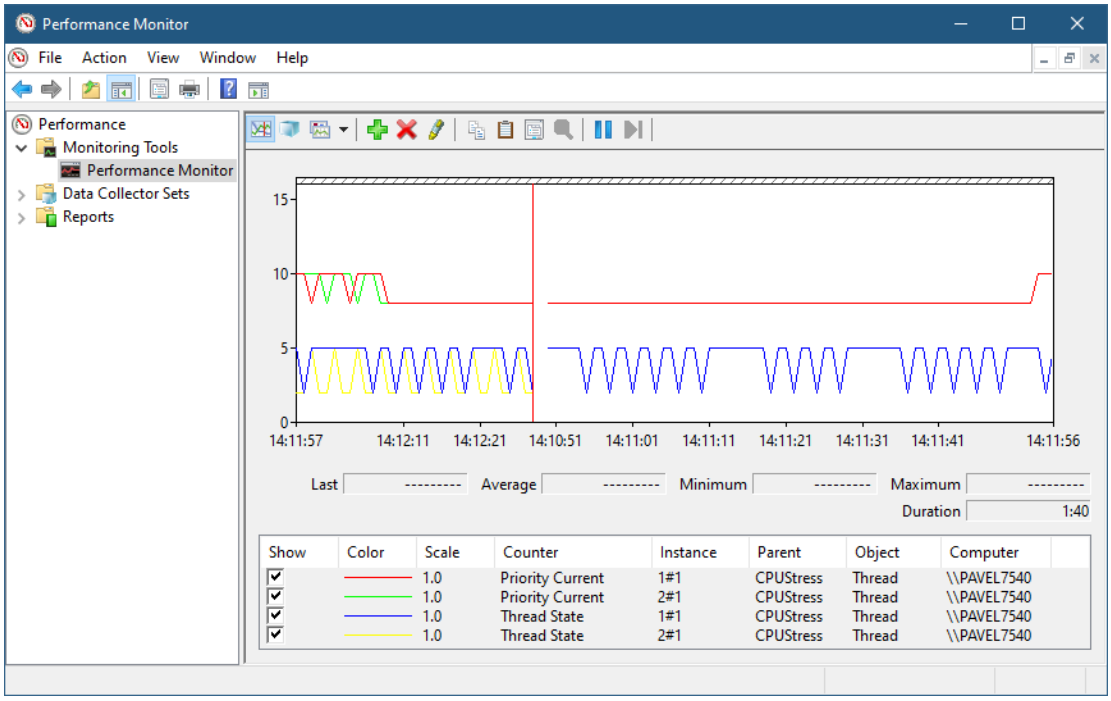


Figure 6-21: state changes in threads

Hard Affinity

You can test hard affinity by continuing from the previous experiment. *CPUStress* allows restricting affinity (you can use *Task Manager* as well). Select *Process/Affinity* from the menu and choose a single CPU as hard affinity (doesn't matter which) (figure 6-22).

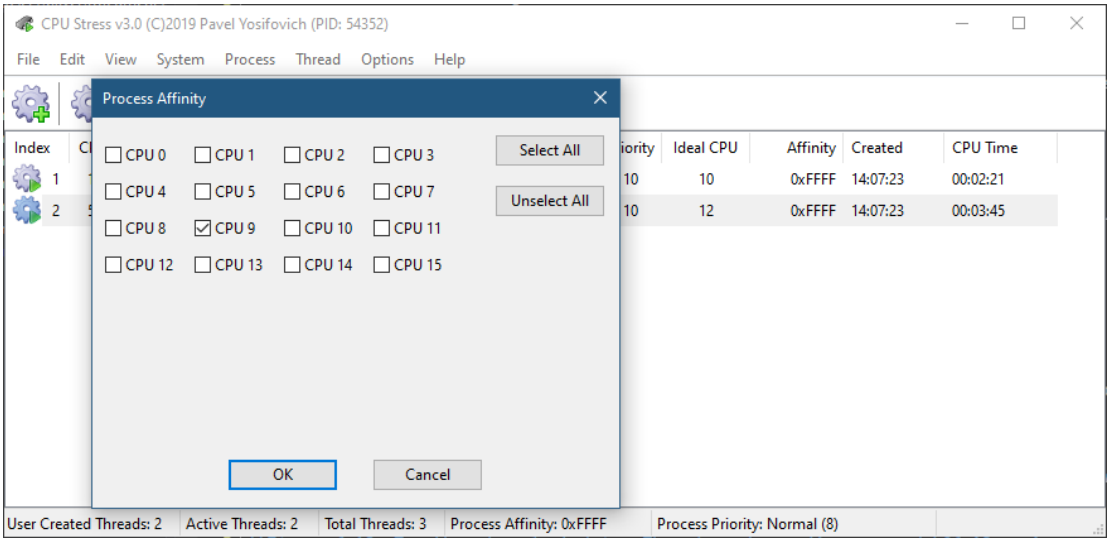


Figure 6-22: Process affinity in *CPUStress*

You should see threads going into the *ready* state from time to time, as both threads fight for the single CPU. Increase their activity to *Maximum* and observe the threads alternate between states 2 (running) and 1 (ready) (figure 6-23).

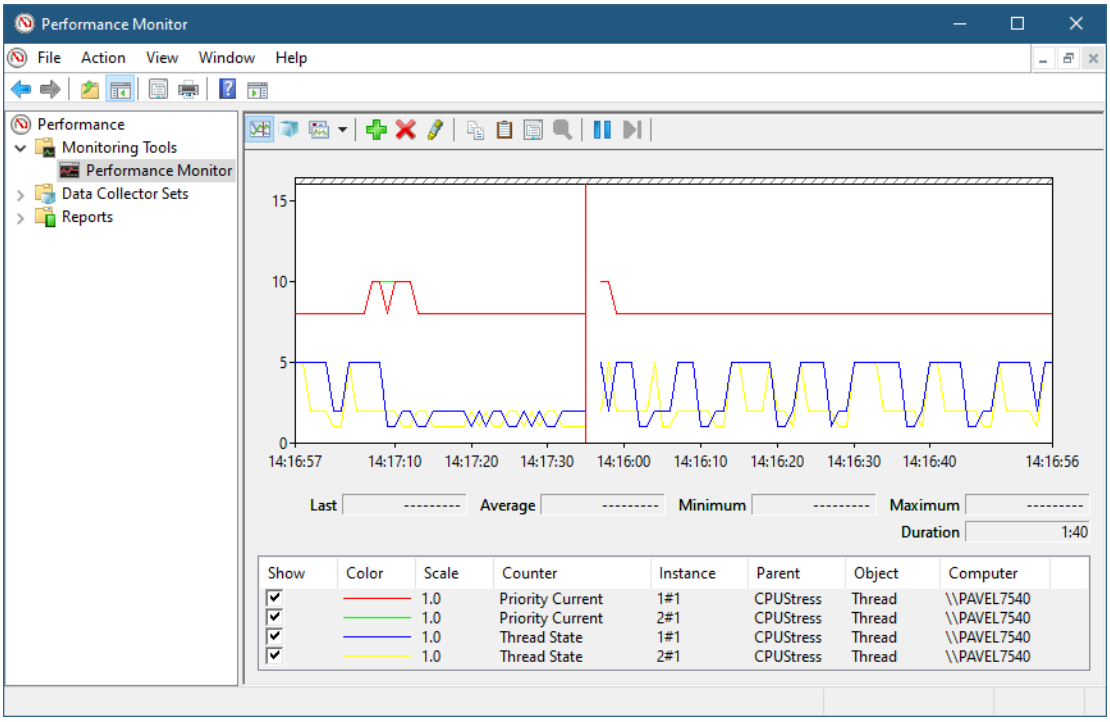


Figure 6-23: Maximum activity with a single processor affinity

Performance Monitor only updates every 1 second, which means several quantums elapse in that time. This means what you see is not completely accurate, but it gives you the general idea.

CPU Sets

Observing CPU sets requires a different tool, one that can show CPU numbers used by threads. We'll use the *Windows Performance Recorder* (WPR) from the Windows SDK's *Windows Performance Toolkit.*

Search for *Windows Performance Recorder* (*wprui.exe*). If you can't find it, most likely it isn't installed. Run the Windows 10 SDK installation again and add *Windows Performance Toolkit.*

In the main *WPR* UI, select *First Level Triage* only. This captures CPU, memory, I/O and other events, from which we'll look into CPU related events (figure 6-24).

WPR uses *Event Tracing for Windows* (ETW) to capture various events emitted by various system components including the scheduler.

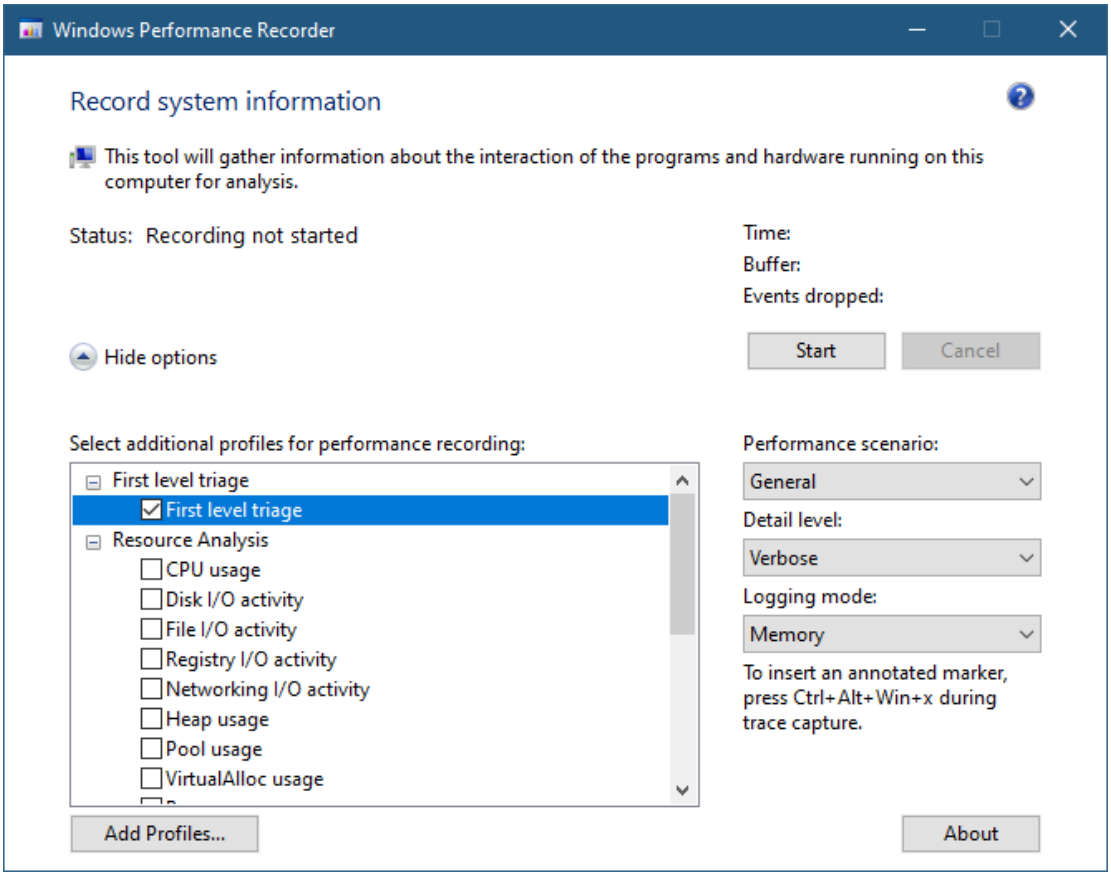


Figure 6-24: WPR main user interface

Go to *CPUStress* and reset the affinity to all processors. Also, set the process CPU sets to the first 4 processors using the *Process/CPU Sets* menu item (figure 6-25).

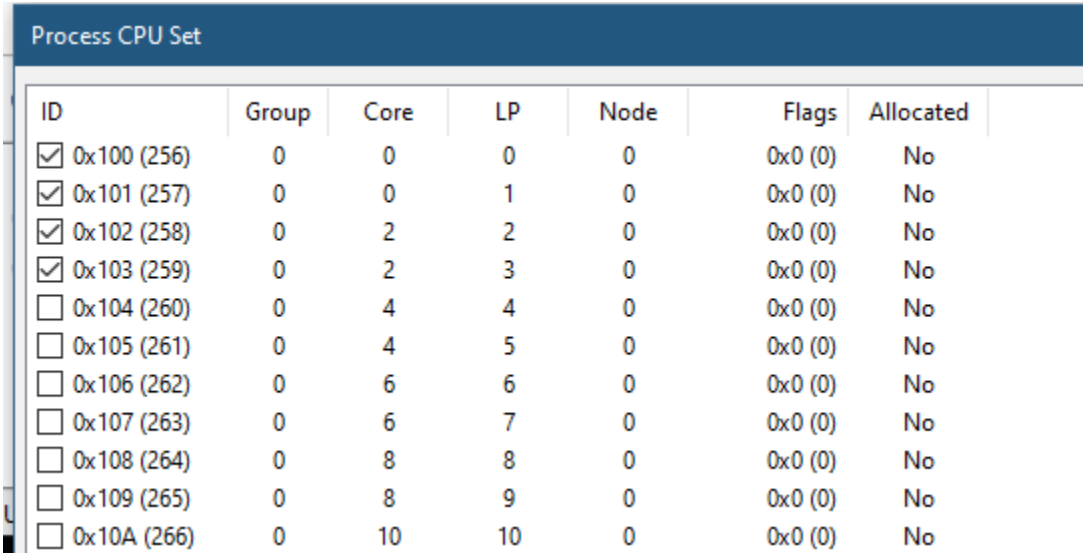


Figure 6-25: Restricted CPU sets in *CPUStress*

Next, use the *Thread/Selected CPU Sets...* menu item to set one of the worker threads to use some other CPU (figure 6-26), where CPU 10 is selected.

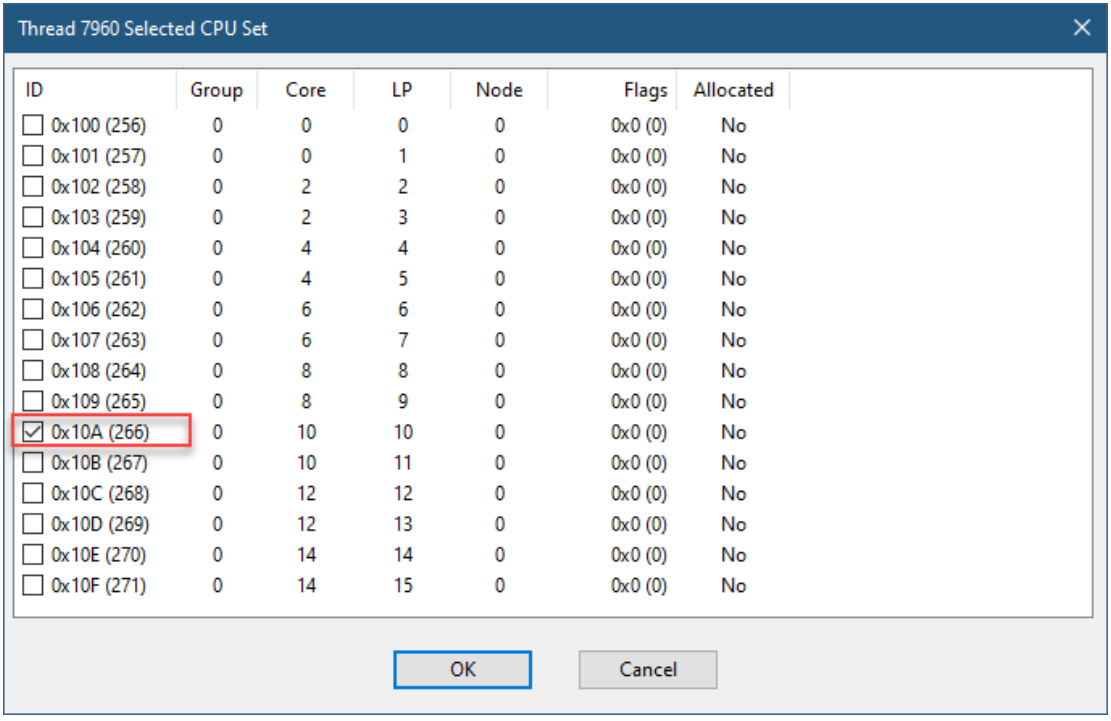


Figure 6-26: Selected CPU sets for a thread in *CPUStress*

Make sure the two threads in *CPUStress* are active. Now go back to WPR and click the *Start* button. Wait 2 or 3 seconds and click *Stop*. Wait for the processing to finish, and once it does, open the trace in *Windows Performance Analyzer* (WPA) with the *Open in WPA* button.

Windows Performance Analyzer (WPA) is an analysis tools for ETW captures. It's fairly complex and versatile, and the following information barely scratches the surface of this powerful tool. WPA is beyond the scope of this book.

When the trace opens, navigate in the left pane to *Computation* / *CPU Usage (Precise)* / *CPU Utilization by Process, Thread, Activity*. You should see something like figure 6-27.

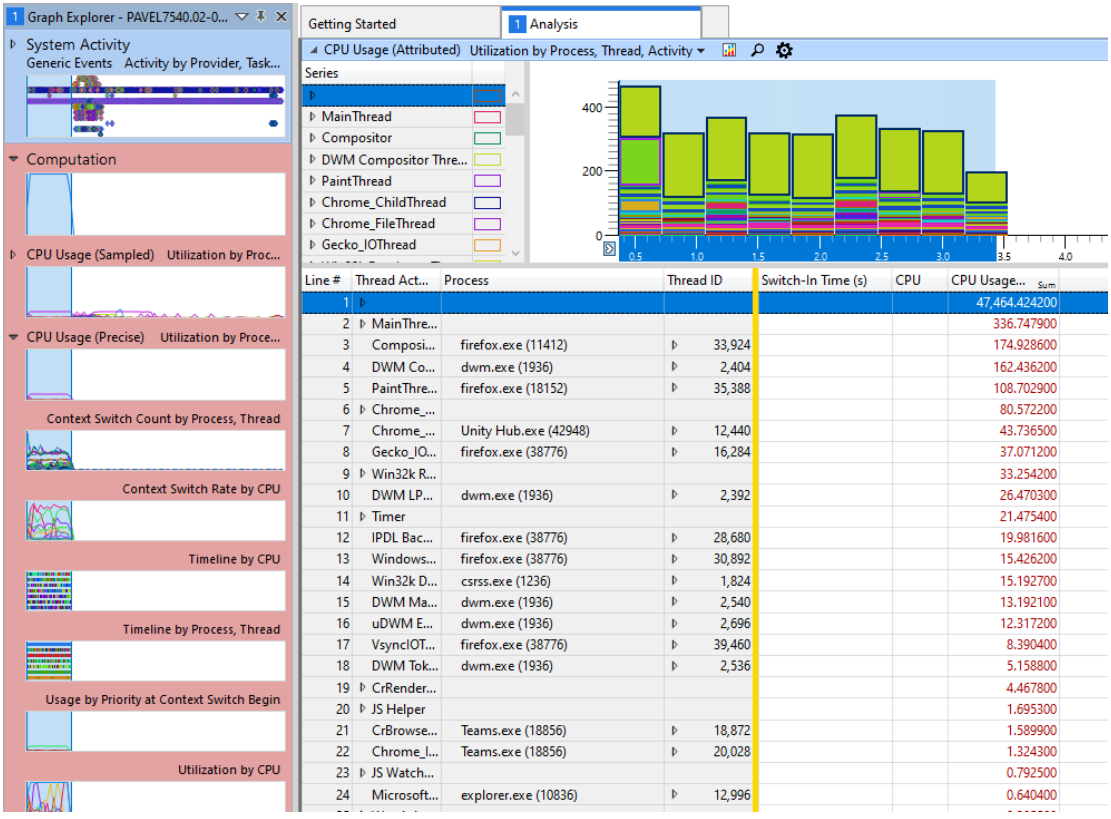


Figure 6-27: Typical UI for WPA

ETW traces are always system-wide, so we first need to filter to our process of interest - *CPUStress*. Expand the unnamed Series tree node in the top left and locate *CPUStress*. Right-click it and select *Filter to Selection*. The view should clear out, leaving *CPUStress* information only. Expand the process node, revealing thread nodes (figure 6-28).

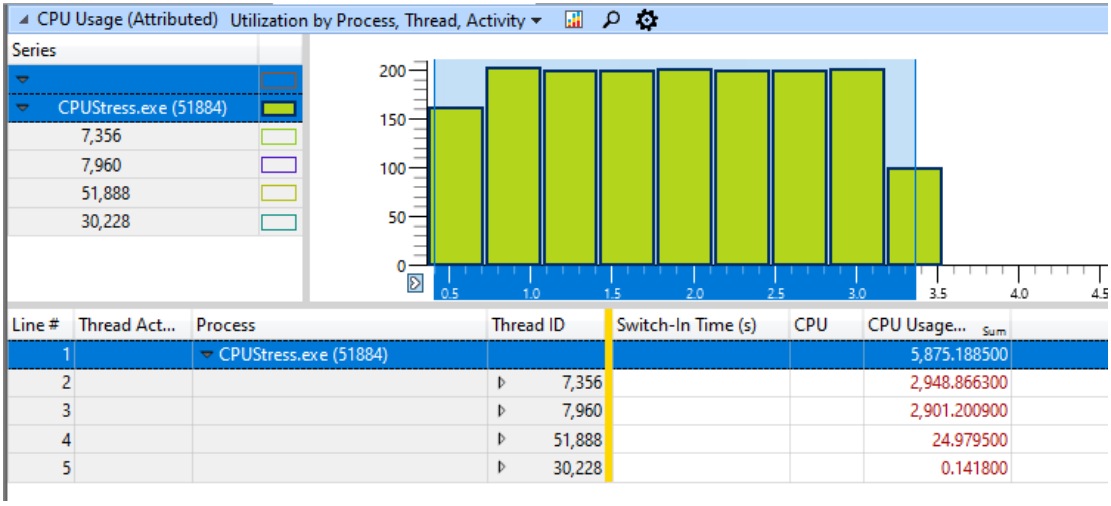


Figure 6-28: WPA filtered to *CPUStress* process

If you expand a thread which did not have its selected CPU sets altered, you should see CPU numbers 0 to 3 (the first four processors set for the process CPU set) (figure 6-29). On the other hand, expanding the thread with its own selected CPU set should only show that CPU as being utilized (figure 6-30).

Process	Thread ID	Switch-In Time (s)	CPU	CPU Usage... Sum
CPUStress.exe (51884)				5,875.188500
	7,356			2,948.866300
		0.834771600	1	317.949800
		2.413675300	3	282.046600
		2.803822300	2	216.938600
		3.154981700	2	197.705800
		1.154337900	1	120.316100
		1.707817700	3	114.882800
		0.617758100	3	108.939400
		2.038749800	3	108.928400
		1.490872600	3	108.849500
		0.509712600	2	108.026900
		1.382759900	0	108.024200
		1.274703500	1	107.989900
		1.599764000	1	107.983500
		3.020830700	2	107.965200
		2.255706000	1	107.950600
		1.822775200	2	107.950400
		1.930734900	3	107.940600
		2.695831000	3	107.920200
		2.155405300	3	100.256400
		0.779550300	0	55.144700

Figure 6-29: Thread with no assigned CPU sets

Line #	Thread Act...	Process	Thread ID	Switch-In Time (s)	CPU	CPU Usage... Sum
1		CPUStress.exe (51884)				5,875.188500
2			7,356			2,948.866300
3			7,960			2,901.200900
4				1.490907400	10	331.789300
5				0.456306800	10	270.421400
6				1.154360600	10	228.355800
7				3.128848900	10	223.825400
8				2.695765500	10	215.960100
9				1.822818900	10	215.864300
10				0.942827600	10	211.402400
11				2.161720300	10	201.937400
12				2.413667700	10	174.083400
13				2.038749600	10	108.931300
14				1.382794000	10	108.073300
15				0.834804000	10	108.022900
16				3.020802400	10	107.995200
17				0.726769300	10	107.954100
18				2.588921600	10	106.815700
19				2.911758800	10	61.990800
20				2.363697800	10	49.929800
21				2.973776300	10	46.984500
22				2.147724600	10	13.920800
23				0.453055800	10	2.118000

Figure 6-30: Thread with assigned CPU set to CPU 10

Background Mode

Some processes are naturally more important than others. For instance, if a user works with Microsoft Word, she probably expects her interaction and usage of Word to be very good. On the other hand, processes such as backup application, anti-virus scanners, search indexers and similar are not as important and should not interfere with the user's main applications.

One way for these background applications to limit their impact is to lower their CPU priority. This works, but CPU is just one type of resource used by processes. Other resources include memory and I/O. This means reducing the CPU priority of threads or the priority class of a process may not be enough to reduce the impact of such processes.

Windows offers the concept of *Background Mode*, where a thread's CPU priority drops to 4 and the *memory priority* and *I/O priority* drop as well. For example, looking at Windows Explorer in *Process Explorer* in the *Threads* view show memory and I/O priority as well as CPU priority (figure 6-31). I/O priority has a default value of *Normal" and memory priority's default value is 5 (possible values are 0 to 7).

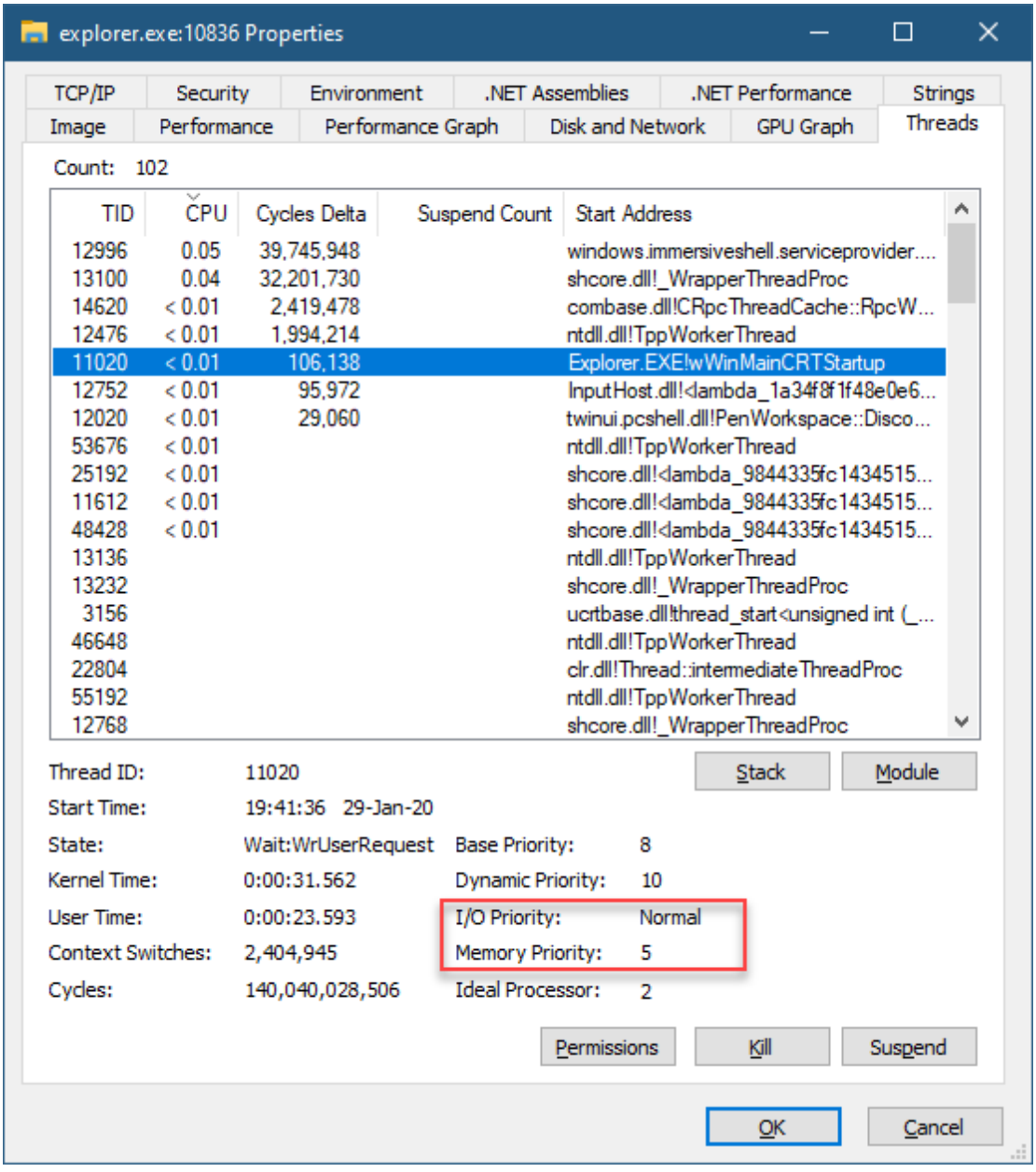


Figure 6-31: Memory and I/O priority in *Process Explorer*

The exact definition of memory priority and I/O priority are not important for this discussion. We'll discuss memory priority in a later chapter. For I/O priority - intuitively - higher levels get precedence over lower levels when accessing I/O.

As a contradictory example, examine figure 6-32, showing threads for a *SerachFilterHost.exe*

process. Notice its memory and I/O priority.

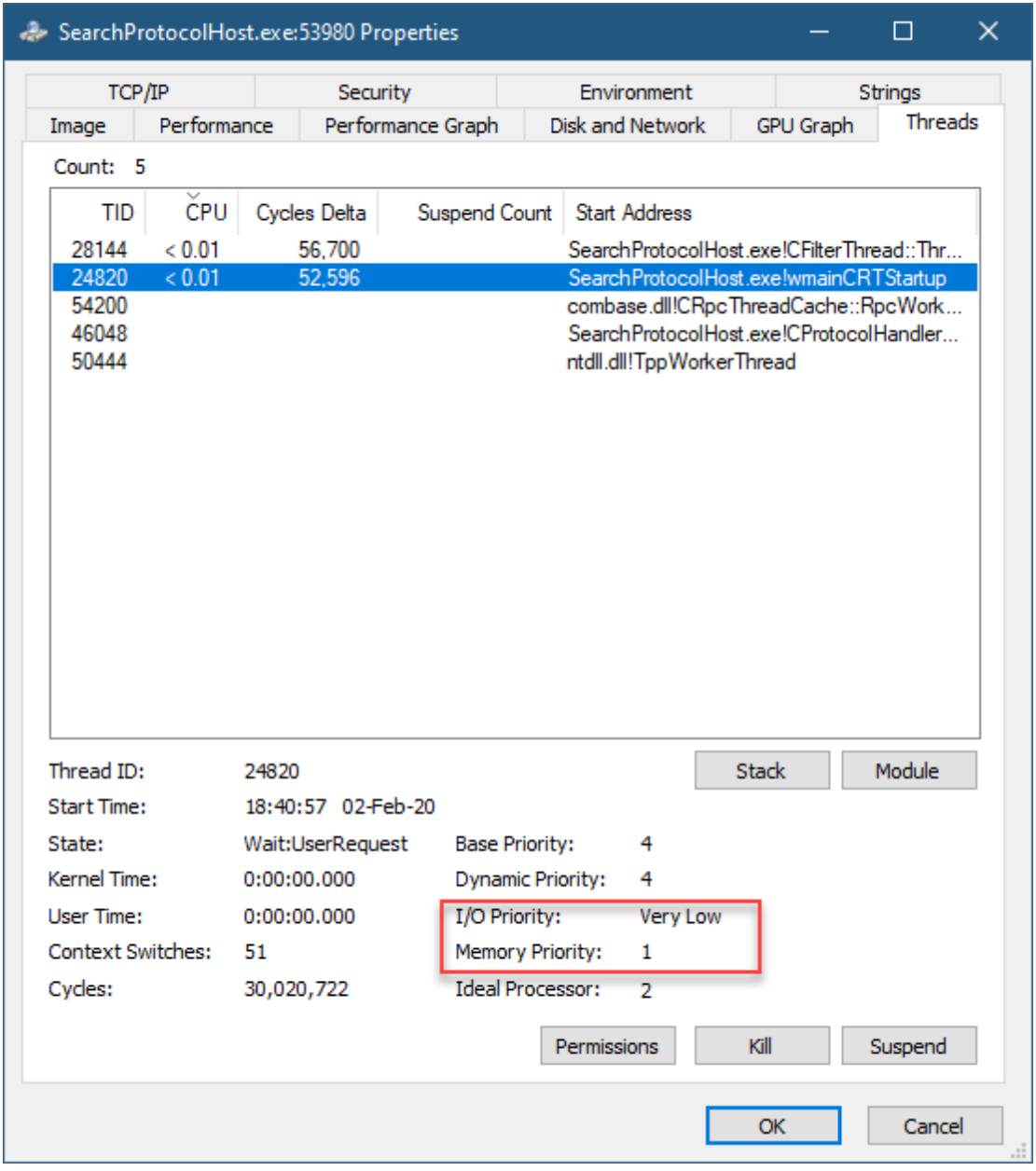


Figure 6-32: Low memory and I/O priority in *Process Explorer*

This *SearchProtocolHost.exe* process lowered its I/O and memory priority as well as its CPU priority in one swoop using a call to `SetPriorityClass` with the special `PROCESS_MODE_BACKGROUND_BEGIN` value like so:

```
::SetPriorityClass(::GetCurrentProcess(), PROCESS_MODE_BACKGROUND_BEGIN);
```

The handle to the process must point to the current process, otherwise the call fails. The background mode begins for all threads in the process until the complementary call is made with `PROCESS_MODE_BACKGROUND_END`:

```
::SetPriorityClass(::GetCurrentProcess(), PROCESS_MODE_BACKGROUND_END);
```

Similarly, the call can be made on a thread basis with the standard `SetThreadPriority` with the special values `THREAD_MODE_BACKGROUND_BEGIN` and `THREAD_MODE_BACKGROUND_END`. Here too, the thread handle must reference the current thread for the call to succeed.

The fact that the above calls require the current process/thread mean that a thread or process cannot be "forced" into background mode; rather, the thread or process itself should be a "good citizen" and enter background mode willingly.

> *Process Explorer* does allow forcing a process into background mode. Right-click a process and select *Set Priority/Background*.

Priority Boosts

As we've seen in the section "Scheduling Basics", the priority is the determining factor where scheduling is concerned. However, Windows employs several tweaks to priority called *priority boosts*. These temporary increase in priority is designed to make scheduling a little more "fair" in some sense, or to provide a better experience for the user. In this section, I'll discuss some of the common priority boost reasons. In any case, don't depend on these boosts, as they may be removed and new ones may appear in future versions of Windows.

Remember that threads in the real-time range (priorities 16 to 31) never have their priority boosted.

Completing I/O Operations

When a thread issues a synchronous I/O operation, it enters a wait state until the operation completes. Once it completes, the device driver responsible for the I/O operation has an

opportunity to boost the requesting thread's priority to increase its chances of running sooner, now that the operation finally completes. The priority boost (if applied) increases the thread's priority by the amount at the driver's discretion, and the priority decays by one level every quantum that the thread manages to run, until the priority drops back to its base level. Figure 6-33 shows a conceptual view of this process.

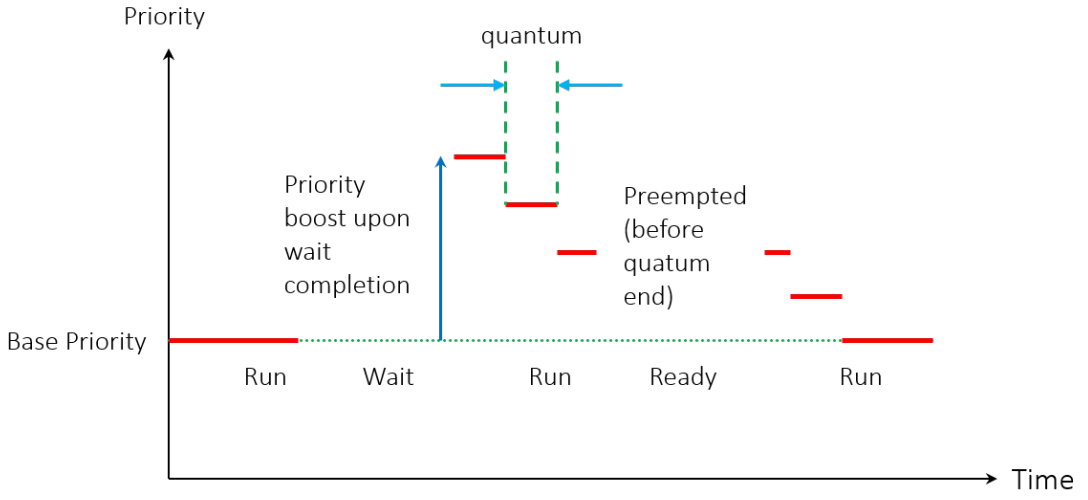


Figure 6-33: Thread priority boost and decay

Foreground Process

There is always an active window on a system - the one usually with a caption in a different color. This window was created by a thread, which is part of a process. This process is sometimes referred to as the *Foreground Process*. In a foreground process on a system configured with short quantums (the default for client versions of Windows): when a thread completes a wait on a kernel object, it gets a +2 boost to its priority. This priority decays after a single quantum to its base level.

GUI Thread Wakeup

When a thread that has a user interface receives a windows message (typically its call to `GetMessage` returns), it gets a +2 boost to improve its chances of running sooner. This priority decays after a single quantum to its base level.

Starvation Avoidance

Threads that are in the *ready* state for at least 4 seconds, get a large boost to priory 15 for a single quantum of execution, after which the priority drops back to its original level. This allows threads in low priority to make some forward progress even on a relatively busy system.

Other Aspects of Scheduling

This section looks at other aspects of scheduling not yet discussed.

Suspend and Resume

A thread can be created in a suspended state by specifying the `CREATE_SUSPENDED` flag as part of `CreateThread`. This allows the caller to prepare the thread for execution, such as manipulating its priority or affinity before actually executing useful code. Using this flag creates the thread with a suspend count of 1. More generally, a thread can be suspended by calling `SuspendThread`:

```
DWORD WINAPI SuspendThread(_In_ HANDLE hThread);
```

The thread handle must have the `THREAD_SUSPEND_RESUME` access mask for the call to succeed. The function increments the suspend count of the thread, and suspends its execution. The function returns the previous suspend count of the thread or `(DWORD)-1` if it fails. There is a maximum suspend count a thread can have defined as `MAXIMUM_SUSPEND_COUNT` (defined as 127). A thread can suspend itself, but it cannot resume itself.

Once suspended, the thread can be resumed with `ResumeThread`:

```
DWORD WINAPI ResumeThread(_In_ HANDLE hThread);
```

`ResumeThread` decrements the thread's suspend count, and if it's zero, it becomes eligible for execution. This function is required if a thread has been created with `CREATE_SUSPENDED`.

Generally, suspending a thread is a bad idea, since there is no way to know where exactly the suspension occurs. For example, the thread might have acquired a lock that other threads are waiting on. Suspending such a thread before it releases the lock causes a deadlock, as other threads contending for the lock will wait indefinitely.

Thread synchronizations and locks are discussed in the next chapter.

Suspending and Resuming a Process

A process is not scheduled, threads do. However, it may be desirable at times to suspend all threads in a process at once. This functionality is used for example in the case of UWP processes, which get all their threads suspended when the application goes to the background. The Windows API does not provide a function for this. It's technically possible to iterate over all threads in a process and call `SuspendThread`, but it's risky at best. A new thread might start while iterating, which would likely miss that thread.

UWP process suspension builds on an undocumented job object feature called *Deep Freeze*.

The native API however, does provide the (undocumented) `NtSuspendProcess` function defined like so:

```
NTSTATUS NtSuspendProcess(_In_ HANDLE hProcess);
```

Although undocumented, it's been around forever, so it's fairly safe to use. If you do use it, don't forget to add `extern "C"` to the function definition to let the linker know it's a C function. Also, add the *NtDll* import library to the project's linker input libraries or in source code like so:

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

The inverse function exists as well, aptly named `NtResumeProcess`:

```
NTSTATUS NtResumeProcess(_In_ HANDLE hProcess);
```

> *Process Explorer* offers a right-click operation on a process to suspend/resume it. It uses the above functions internally.

Sleeping and Yielding

A thread can voluntarily relinquish the remainder of its quantum by entering sleep:

```
void Sleep(_In_ DWORD dwMilliseconds);
```

The thread goes into a wait state for **approximately** the number of milliseconds requested. A value of zero is valid, and causes the scheduler to execute the next thread in the queue which has the same priority. If there is none, the thread continues execution. Another legal value is `INFINITE` which causes the thread to sleep forever; this is hardly ever useful.

The accuracy of the sleep interval actually depends on whether the internal timer resolution has be altered. Normally it should be the same as a clock tick used for scheduling, but usually it's much less because some other thread requested it. The output of the *clockres.exe Sysinternals* utility shows the current timer interval, that affects (among other things) sleep time accuracy.

As an alternative to calling `Sleep(0)`, a thread can call `SwitchToThread`:

```
BOOL SwitchToThread(void);
```

`SwitchToThread` tells the scheduler to schedule the next ready thread, even if its priority is lower than the current thread. The function returns `TRUE` if the scheduler is able to comply; otherwise, the thread continues execution and the function returns 'FALSE'.

Summary

In this chapter we looked at various aspects of scheduling. From priorities and how they can be set, to simple single CPU scheduling, up to multiprocessor considerations, with affinities and CPU sets. In the next chapter, we'll look at thread synchronization, where threads must coordinate their efforts and the various ways they can do so.

Chapter 7: Thread Synchronization (Intra-Process)

In an ideal threading world, threads would go about their business not bothering other threads. In reality, threads must at times synchronize with each other. The canonical example is accessing a shared data structure, such as a dynamic array. If one thread attempts to insert an item into the array, no other thread should manipulate the same array or even read from it. It might be the case that the threads do so at different times, but since it's all about timing, they may do so at the same time. This would lead to data corruption or some exception. To mitigate that, threads sometimes need to synchronize their work.

Windows provides a rich set of primitives that aid in achieving this (and other) synchronization. In this chapter, we'll examine the synchronization mechanisms available to user mode developers through the Windows API for synchronizing threads within a single process. In the next chapter, we'll look at more synchronization primitives that can be used to synchronize threads running in different processes.

In this chapter:

Synchronization Basics

The classic synchronization is about avoiding a *data race*. A *data race* occurs when two or more threads access the same memory location and at least one of them is writing to that location. Reading concurrently from the same location is never a problem. But once a write enters the picture, all bets are off. Data may become corrupted, reads may become torn (some of the data is read before the change and some after the change). This is where synchronization is required.

In chapter 5, we saw an example application that parallelizes primes numbers calculation. That particular algorithm (fork/join) did not require any synchronization, except for waiting for all threads to complete. This is ideal, as performance can be improved by throwing more CPUs at the problem (at least up to a point). The need to synchronize is not fun, as it reduces performance by definition, since some operations must be performed sequentially rather than concurrently. In fact, the speedup that can be gained by adding more threads/CPUs to a problem depends on the percentage of the code that can be parallelized. This is described nicely by *Amdahl's Law*:

$$\text{Speedup Limit} = \frac{1}{1-p}$$

> A more thorough discussion of Amdahl's Law can be found on Wikipedia: https://en.wikipedia.org/wiki/Amdahl%27s_law

Where p is the percentage of code that can be parallelized. For example, if 80% of code can be parallelized, then the maximum speedup possible is 5, no matter how many processors are thrown into the problem.

Most synchronization-related operations require threads to wait upon some condition, until it is safe to proceed, preventing a data race. In the following sections, we'll look at various synchronization options provided by the Windows API, from the simplest to more complex.

Atomic Operations

Some operations that seem so simple and quick are not actually thread safe. Even a simple C variable increment (`x++`) is not thread or multiprocessor safe. Consider, for example, two threads running concurrently on two processors that perform an increment to the same memory location (figure 7-1).

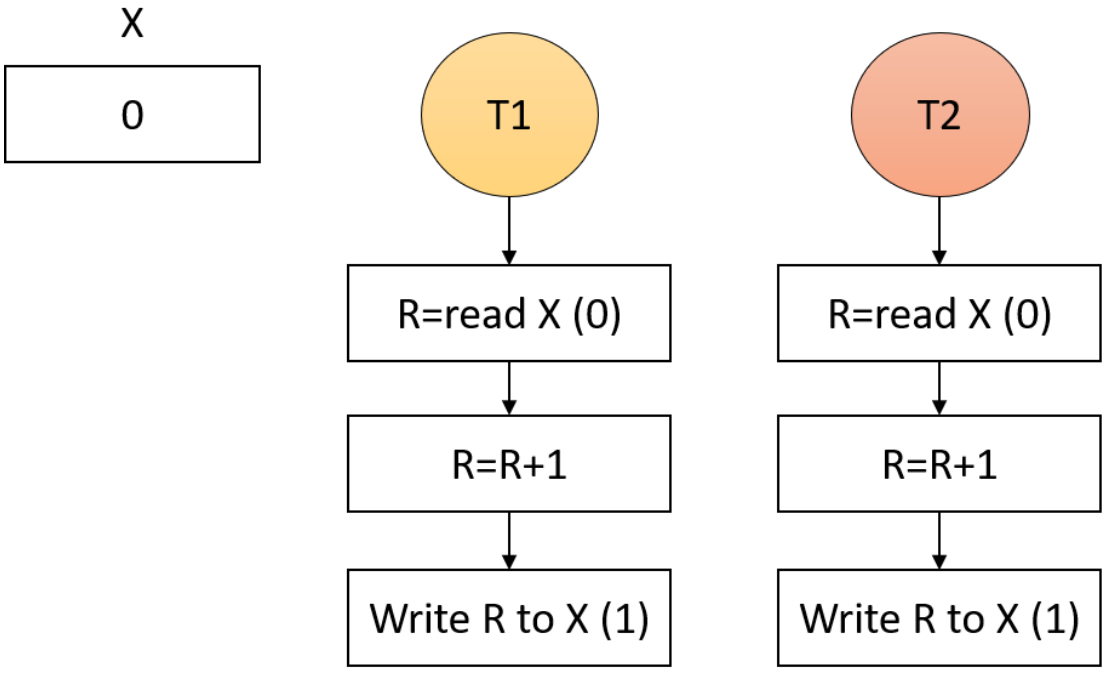


Figure 7-1: Simple increment by multiple threads

Even a simple increment involves a read and write. In figure 7-1, each thread may read the initial value (0) into CPU registers. Each thread increments its processor's register and then writes back the result. The final result written to X is 1 instead of 2. This diagram is a gross simplification, as there are other factors at play, such as CPU caches. But even ignoring that, this is clearly a data race. In fact, one of the threads (say T2) may be preempted (for example after R is incremented), and while T1 continues incrementing X, once T2 receives CPU time, it writes back 1 to X, effectively killing all increments done by thread T1.

The Simple Increment Application

The Simple Increment application shown in figure 7-2, uses multiple threads to do one thing: increment a single memory location. The program allows selecting the count of threads to run concurrently and the number of increments each thread should perform. Once the *Run* button is clicked, operations are on the way. When done, the actual result and the expected result, along with the time it took to execute.

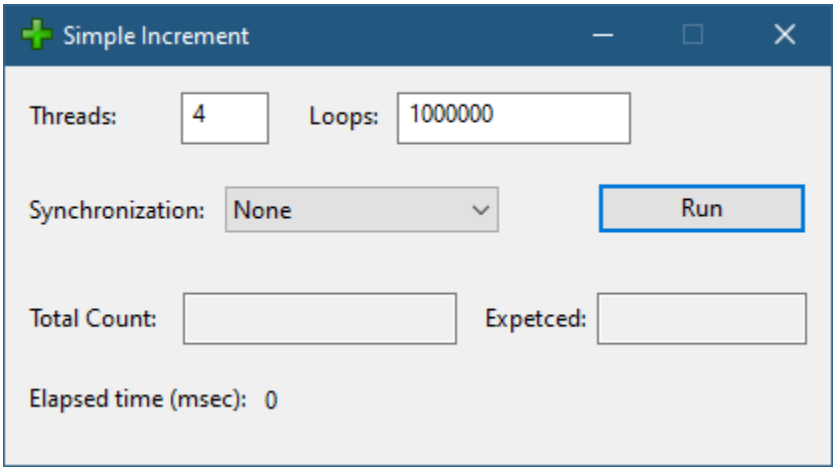


Figure 7-2: The *Simple Increment* application

The *Synchronization* combobox allows selecting how to synchronize the increments. The first (and default) option ("None") is to simply use the ++ operator on the shared memory location - no synchronization at all. Clicking *Run* with the default options shows something like figure 7-3.

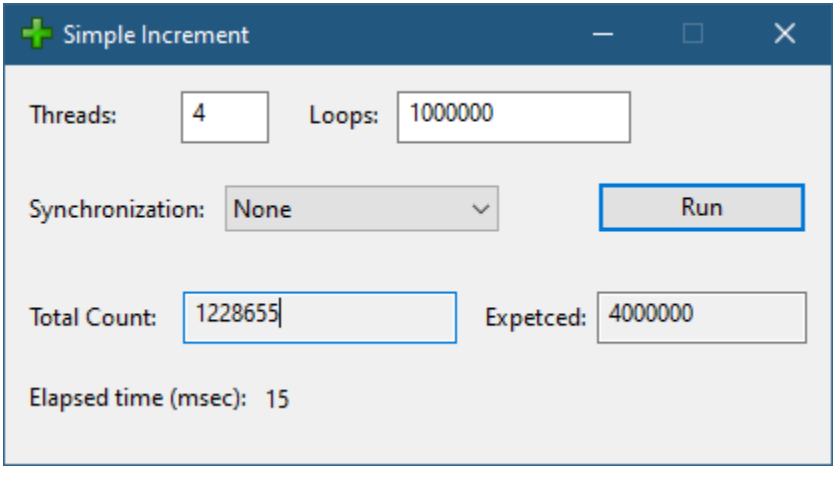


Figure 7-3: *Simple Increment* application with no synchronization

Notice the final result is nowhere near the expected result. This is because of the lack of synchronization, causing increments to become "lost". You can click *Run* again and you'll get a different result. This is the nature of synchronization issues. The following is the piece of code responsible for doing the multi-threaded increments with a simple ++ operations (in *MainDlg.cpp*):

```
void CMainDlg::DoSimpleCount() {
    auto handles = std::make_unique<HANDLE[]>(m_Threads);
    for (int i = 0; i < m_Threads; i++) {
        handles[i] = ::CreateThread(nullptr, 0, [](auto param) {
            return ((CMainDlg*)param)->IncSimpleThread();
            }, this, 0, nullptr);
    }
    ::WaitForMultipleObjects(m_Threads, handles.get(), TRUE, INFINITE);
    for (int i = 0; i < m_Threads; i++)
        ::CloseHandle(handles[i]);
}

DWORD CMainDlg::IncSimpleThread() {
    for (int i = 0; i < m_Loops; i++)
        m_Count++;
    return 0;
}
```

As we've seen in chapter 5, passing information to a thread is done via the `PVOID` parameter of `CreateThread`. However, in many cases it's more convenient to have the thread function be an instance function, rather than static or global. A useful trick is to pass `this` as the parameter and use it to call into an instance function, where the entire object state is available. This allows the `IncSimpleThread` function to be an instance one, rather than static.

You may be wondering why not just capture the `this` pointer and use the data members directly? Unfortunately, API functions can only use non-capturing lambda functions. This is why this trick is needed.

In the above code, `m_Threads` is the number of threads, `m_Loops` is the number of iterations to do and `m_Count` is the shared memory location being incremented.

This is clearly an artificial example, where millions of increments are performed on the same memory location, which exposes the bug easily. In real applications, these increments are much less frequent, which means any synchronization bugs are less likely to occur, and in fact may be missed by developers and QA, only to be discovered on customer's machines.

The `Interlocked` Family of Functions

The solution to the above synchronization issue is in performing the increment as an *atomic operation*, so that any increment is isolated from other increments and any other access to the same memory location using other `Interlocked` functions. This atomic operation and

other similar operations are exposed in the Windows API though a set of functions with the `Interlocked` prefix. In the simple increment case, that's `InterlockedIncrement`:

```
unsigned InterlockedIncrement(unsigned volatile *Addend);
```

This performs an atomic increment, and as a bonus, returns the new value in addition to actually changing the memory location. Behind the covers, this is not a true function, but rather a *compiler intrinsic* that issues a special instruction to the CPU to perform this operation atomically. This is great, since leveraging the hardware is always going to be faster than software. Also, since there is no explicit "lock" object used, no deadlock is possible with these functions.

Back to the *Simple Increment* application, the second synchronization method in the combobox sets the increment method to `InterlockedIncrement`, used in the `InterlockedThread` function:

```
DWORD CMainDlg::IncInterlockedThread() {
    for (int i = 0; i < m_Loops; i++)
        ::InterlockedIncrement((unsigned*)&m_Count);
    return 0;
}
```

Figure 7-4 shows an example run with the *Interlocked* option selected in the synchronization combobox.

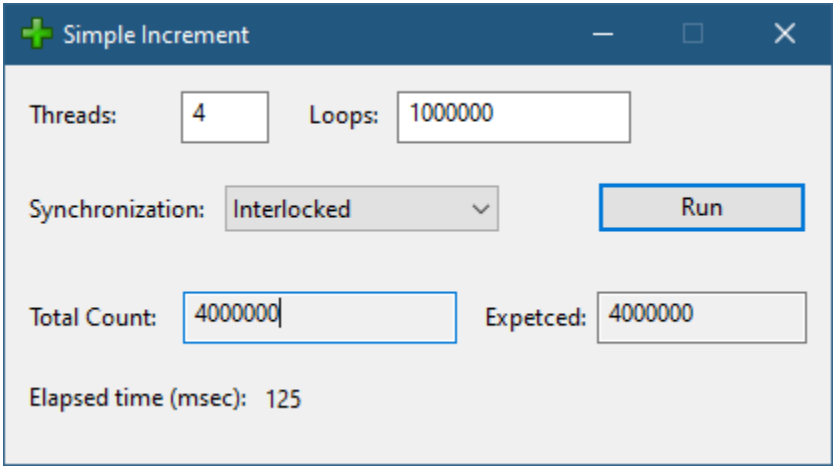


Figure 7-4: *Simple Increment* application with *Interlocked* synchronization

Other simple functions in the same vein include `InterlockedDecrement`, `InterlockedAdd`, `InterlockedExchange`, `InterlockedAnd`, `InterlockedOr`, `InterlckedXor`, `InterlockedExchangePointer`, `InterlockedCompareExchange` and a few others. These exist for 64 bit and

16 bit values as well, with the suffixes `64` and `16`, respectively, added to the function names (e.g. `InterlockedInrement64`).

There are also extended functions, such as `InterlockedAndAcquire`, `InterlockedAndRelease` and `InterlockedAndNoFence` (and similar for other operations). These specialized versions also specify the acquire/release semantics on the memory location. Discussing these variants is out of scope for this book, and you should use the "standard" functions, which are the safest, unless you know what you're doing. More information on fences and acquire/release semantics can be found on the web. One of the best is the talk "Atomic<> Weapons"[1] by Herb Sutter. You can also (or alternatively) watch my (somewhat abridged) session "Concurrency and the C++ Memory Model"[2].

The `InterlockedCompareExchange` function is mostly used when doing *lock free programming*, a paradigm that uses CPU intrinsics to avoid using any locks in software. This topic is out of scope for this book, as it's not Windows-specific. However, the Windows API offers a lock-free, singly linked list implementation. These functions use the `SLIST_HEADER` union as a linked list header and `SLIST_ENTRY` structures as the entries that can be added/removed to/from the list, atomically.

Both types are fully defined in the SDK headers, but only `SLIST_ENTRY` should be marginally interesting to look at:

```
typedef struct DECLSPEC_ALIGN(16) _SLIST_ENTRY {
    struct _SLIST_ENTRY *Next;
} SLIST_ENTRY, *PSLIST_ENTRY;
```

`SLIST_ENTRY` and `SLIST_HEADER` must be aligned on a 16-byte boundary, indicated by decorating the types with the `__declspec(align(16))` VC++ compiler attribute. Stack-based or static allocation of these types will work correctly, but more often than not you'll need to dynamically allocate `SLIST_ENTRY`. The C runtime offers the `_aligned_malloc` function that guarantees a specified alignment of memory allocation.

Clearly, it's a classic single list entry item. But where is the actual data? The expectation is that your data item includes, as a first entry, `SLIST_ENTRY` itself. This ensures the alignment requirement is met for the `SLIST_ENTRY`. The following example shows a data item type suitable to be stored in the described linked list:

[1] https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2
[2] https://www.youtube.com/watch?v=NZ_ncor_Lj0

```
struct MyDataItem {
    SLIST_ENTRY Entry;
    int MyValue;
    //...
};
```

Since the operations on the list must be lock-free, and the list is singly linked, it actually implements a stack. There is no thread-safe way to add items at the tail of the list. This is why the main operations on the list use the names "Push" and "Pop" - terms used in stack-based data structures.

Table 7-1 lists the functions available for lock-free singly linked list manipulation.

Table 7-1: Functions for working with singly linked lists

Function	Description
`InitializeSListHead`	Initializes the list's head to an empty list.
`InterlockedPushEntrySList`	Inserts an item in the front of the list.
`InterlockedPopEntrySList`	Removes the item in the front of the list.
`InterlockedPushListSListEx`	Inserts multiple items in from of the list.
`InterlockedFlushSList`	Removes all items from the list, returning the item that was in front (if any).
`QueryDepthSList`	Returns the count of items in the list. This function is not thread safe and is best avoided. It's much better to keep track of the item count yourself (with `InterlockedIncrement` / `InterlockedDecrement`).

Critical Sections

The `Interlocked` family of functions is great with simple cases such as integer increments. However, for other operations, a more general mechanism is required. A *critical section* is a classic synchronization mechanism based on one thread at most acquiring a lock. Once a thread acquired a specific lock, no other thread can acquire the same lock until the thread that acquired it in the first place releases it. Only then, one (and only one) of the waiting threads can acquire the lock. This means that at any given moment, no more than one thread has acquired the lock. This idea is depicted in figure 7-5.

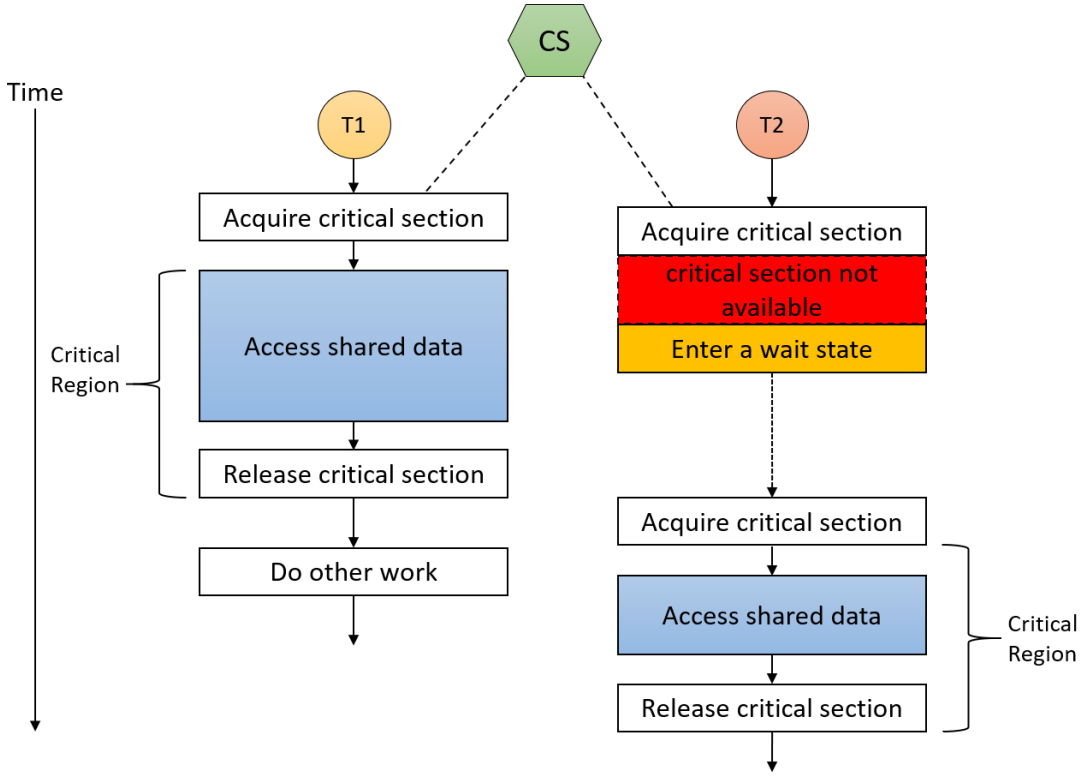


Figure 7-5: Synchronization with a critical section

The thread that acquired the lock is also its *owner*, which means two things:

1. The owner thread is the only thread that can release the critical section.
2. If the owner thread attempts to acquire the critical section a second time (recursively), it succeeds automatically, incrementing an internal counter. This means the owner thread now has to release the critical section the same number of times to truly release it.

> The code between acquire and release of the lock is called a *critical region*.

The critical section itself is represented by the `CRITICAL_SECTION` structure, in itself a `typedef` to another structure, `RTL_CRITICAL_SECTION`. Although the structure is fully defined, you should treat it as opaque. Initializing the critical section must be done with one of the following functions:

```
void InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);

BOOL InitializeCriticalSectionAndSpinCount(
  LPCRITICAL_SECTION lpCriticalSection,
  DWORD              dwSpinCount);

BOOL InitializeCriticalSectionEx(
  LPCRITICAL_SECTION lpCriticalSection,
  DWORD              dwSpinCount,
  DWORD              Flags);
```

The initialization of a critical section involves setting its members to some initial values. This is why the first function returns *void*.

The second and third variants set a *spin count* for the critical section. The idea is that if a critical section cannot be acquired by a thread, it should enter a wait state because some other thread holds the critical section. However, entering a wait state requires the thread to transition to kernel mode, which is not cheap. A compromise could be to spin a small amount of time because it's likely that the current owner of the critical section will release it very soon, and so a transition to the kernel can be avoided. The spin count maximum value is `0x00ffffff` (the remaining hex digits are used internally as flags).

What should the spin count be? It's difficult to answer that categorically, because it depends on the actual processor type and other hardware factors. The default spin count is 2000 (used by `InitializeCriticalSection`).

If the system has a single processor (or the process image file has the "Single CPU" flag in its PE header), the spin count is always set to zero. This makes sense, since another thread would never be able to release the critical section while this thread is spinning, since there are no more processors.

The last initialization function adds a flags parameter. Several are defined in the headers, but only one is documented - `CRITICAL_SECTION_NO_DEBUG_INFO` (`0x01000000`), which specifies that the critical section structure should not allocate an extra debug structure that can help with diagnosing critical section issues.

When the critical section is not needed anymore, call `DeleteCriticalSection`:

```
void DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Once the critical section is initialized, it can be acquired and released by threads using the following functions:

```
void EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);

void LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

`EnterCriticalSection` attempts to acquire the critical section and only returns when it does. If the calling thread is already the owner of the critical section, it continues immediately. Conversely, `LeaveCriticalSection` releases an already acquired critical section.

Curiously enough, any thread can call `LeaveCriticalSection` (not just the current owner thread) and succeed. I would expect the function to throw an exception (since it returns `void`). But it just releases the critical section, turning the owner thread ID back to zero.

A simple example is shown in the *Simple Increment* application. If you select the synchronization mechanism as "Critical Section" in the combobox and click *Run*, the increments would now be protected by a critical section (figure 7-6).

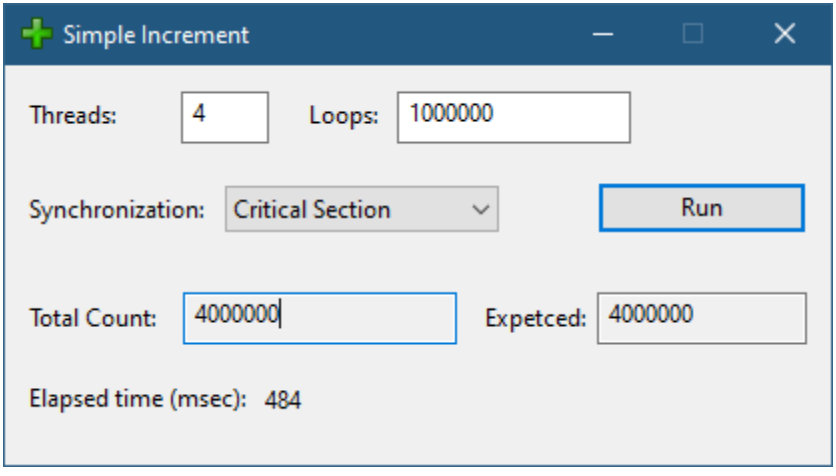


Figure 7-6: *Simple Increment* application with critical section synchronization

The code to accomplish this synchronization in *Simple Increment* is the following:

```
// m_CritSection is CRITICAL_SECTION

void CMainDlg::DoCriticalSectionCount() {
    auto handles = std::make_unique<HANDLE[]>(m_Threads);
    ::InitializeCriticalSection(&m_CritSection);

    for (int i = 0; i < m_Threads; i++) {
        handles[i] = ::CreateThread(nullptr, 0, [](auto param) {
            return ((CMainDlg*)param)->IncCriticalSectionThread();
            }, this, 0, nullptr);
    }
    ::WaitForMultipleObjects(m_Threads, handles.get(), TRUE, INFINITE);
    for (int i = 0; i < m_Threads; i++)
        ::CloseHandle(handles[i]);
    ::DeleteCriticalSection(&m_CritSection);
}

DWORD CMainDlg::IncCriticalSectionThread() {
    for (int i = 0; i < m_Loops; i++) {
        ::EnterCriticalSection(&m_CritSection);
        m_Count++;
        ::LeaveCriticalSection(&m_CritSection);
    }
    return 0;
}
```

Each call to `EnterCriticalSection` must be matched by a `LeaveCriticalSection` in the same function. It's too dangerous to call some other function within the critical region that is expected to call `LeaveCriticalSection`. To avoid possible bugs, always use these pair of functions within the same

The `EnterCriticalSection` waits for the critical section to be available as long as it takes. There is no way to specify a timeout, but there is a way to inspect the critical section, if it's free - acquire it; otherwise, continue execution. This is exactly what `TryEnterCriticalSection` does:

```
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Locks and RAII

As we've seen in the previous section, `EnterCriticalSection` and `LeaveCriticalSection` are natural pairs. It would be unfortunate to "forget" calling `LeaveCriticalSection`, for example

by returning from the function before the call. This mistake is easy to make, and even if there is no such bug, it forces the developer to think about it and make sure that any future modifications to that function don't break the pair of calls.

It would be much better to have code that automatically calls `LeaveCriticalSection` no matter what, without the code having to worry about it. There are two ways to get this behavior: termination handlers and the C++ *Resource Acquisition Is Initialization* (RAII) idiom.

Termination handlers will be discussed in more detail in a chapter 23 (in part 2), but here are the essentials:

```
CRITICAL_SECTION cs;

void DoWork() {
    ::EnterCriticalSection(&cs);
    __try {
        // manipulate shared resource
    }
    __finally {
        ::LeaveCriticalSection(&cs);
    }
}
```

The `__try` and `__finally` keywords are two Microsoft-specific keywords extending the C language for the sake of running the code in the `__finally` block when leaving the `__try` block, no matter what. Even a `return` statement within the `__try` block would first call the `__finally` block and only then return from the function.

If you're using C (and not C++), then termination handlers are your best option. When working with C++, it's better (and more convenient) to leverage constructors and destructors that can execute code automatically when objects are constructed and destroyed, respectively. (This is known as the RAII idiom in C++)

For a critical section, a RAII class could look something like this:

```
// AutoCriticalSection.h

struct AutoCriticalSection {
    AutoCriticalSection(CRITICAL_SECTION& cs);
    ~AutoCriticalSection();

    // delete copy ctor, move ctor, assignment operators
    AutoCriticalSection(const AutoCriticalSection&) = delete;
    AutoCriticalSection& operator=(const AutoCriticalSection&) = delete;
    AutoCriticalSection(AutoCriticalSection&&) = delete;
    AutoCriticalSection& operator=(AutoCriticalSection&&) = delete;

private:
    CRITICAL_SECTION& _cs;
};

// AutoCriticalSection.cpp

AutoCriticalSection::AutoCriticalSection(CRITICAL_SECTION& cs) : _cs(cs) {
    ::EnterCriticalSection(&_cs);
}

AutoCriticalSection::~AutoCriticalSection() {
    ::LeaveCriticalSection(&_cs);
}
```

> The code is part of the *ThreadingHelpers* project in this chapter's code samples.

The constructor acquires the critical section and the destructor releases it. Using it in the *Simple Increment* application could be done like so:

```
DWORD CMainDlg::IncCriticalSectionThread() {
    for (int i = 0; i < m_Loops; i++) {
        AutoCriticalSection locker(m_CritSection);
        m_Count++;
    }
    return 0;
}
```

While we're on the subject of RAII, it may be a good idea to wrap the critical section itself in a RAII class, so that critical section initialization and deletion is automatic as well. Here is a possible implementation:

```
// CriticalSection.h

class CriticalSection : public CRITICAL_SECTION {
public:
    explicit CriticalSection(DWORD spinCount = 0, DWORD flags = 0);
    ~CriticalSection();

    void Lock();
    void Unlock();
    bool TryLock();
};

// CriticalSection.cpp

CriticalSection::CriticalSection(DWORD spinCount, DWORD flags) {
    ::InitializeCriticalSectionEx(this, (DWORD)spinCount, flags);
}

CriticalSection::~CriticalSection() {
    ::DeleteCriticalSection(this);
}

void CriticalSection::Lock() {
    ::EnterCriticalSection(this);
}

void CriticalSection::Unlock() {
    ::LeaveCriticalSection(this);
}
```

```
bool CriticalSection::TryLock() {
    return ::TryEnterCriticalSection(this);
}
```

The `Lock`, `Unlock` and `TryLock` are not required, but may be helpful in some scenarios.

The derivation from `CRITICAL_SECTION` allows passing `CriticalSection` (pointer or reference) if a `CRITICAL_SECTION` is required. Alternatively, a `CRITICAL_SECTION` member could be embedded inside the structure, with an operator that can implicitly cast `CriticalSection` to `CRITICAL_SECTION`. I leave this implementation as an exercise for the reader.

Deadlocks

Working with critical sections seems simple enough. Even if we work with the various RAII wrappers, there is still a danger of deadlocks. A classic deadlock occurs when thread A that owns lock 1 (e.g. a critical section) waits for lock 2 that is owned by thread B, while thread B is waiting for lock 1.

The way to avoid deadlocks is theoretically easy: always acquire the locks in the same order. This means that every thread that needs more than one lock should always acquire the locks in the same order. This guarantees deadlock-free execution (at least not because of these locks). The practical issue is how to enforce the ordering; without writing any code, it's a matter of documenting the order so that future code continues to adhere to the rules. An alternative option is to write a "multi-lock" wrapper that always acquires locks in the same order. A simple way to accomplish that is to order the acquisition by the lock's address in memory.

Write such a multi-lock wrapper for critical sections.

The MD5 Calculator Application

The *MD5Calculator* application demonstrates the use of critical sections that is more interesting (and complex) than *Simple Increment* (we'll also modify it in later sections). The application calculates the MD5 hash of image files (EXEs and DLLs) as they are loaded by processes. Since processes typically use many common DLLs, it's better to cache the results of hashes that have already been calculated. The application has several challenges:

- Show a responsive user interface while calculations are done in the background.

- Get notified of new images (DLLs/EXEs) that are loaded by any process on the system.
- Manage a cache of files with their MD5 hashes.

Figure 7-7 shows the main screen of the application before any activity.

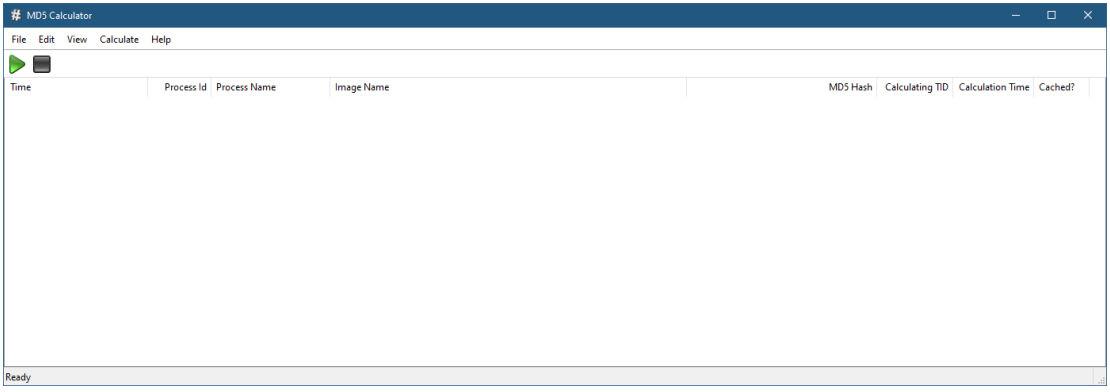


Figure 7-7: The *MD5 Calculator* application

By default, no caching for MD5 hashes is done. Clicking the green *Go* button (or the *Calculate* / *Go* menu item), starts capturing image loads and calculates hashes from scratch for each image file, even if duplicates are loaded (figure 7-8).

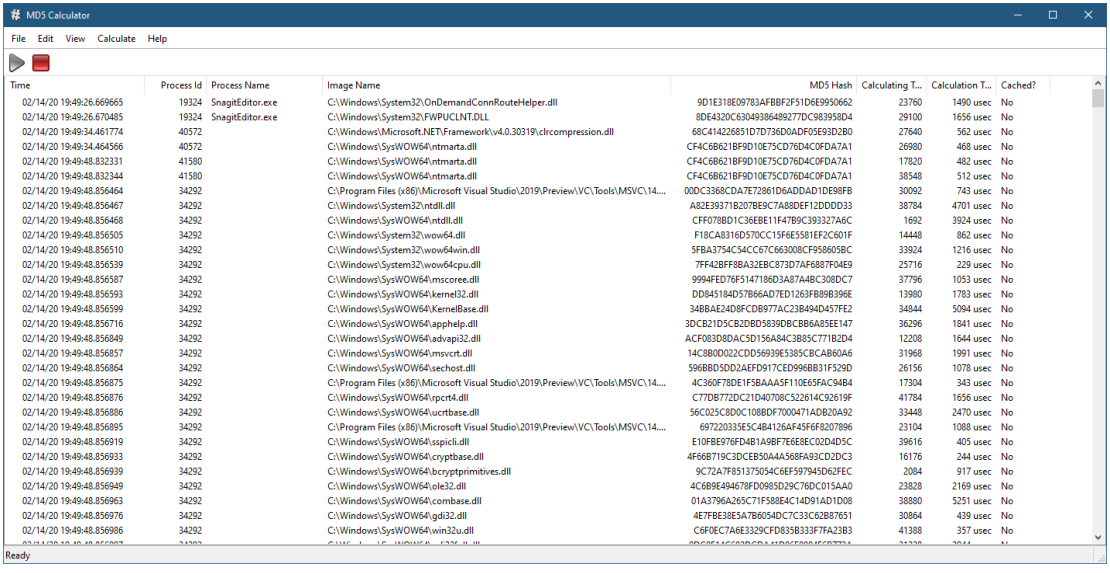


Figure 7-8: The *MD5 Calculator* application running with no caching

You can launch a new process, such as *Notepad* and watch its modules load and reflected at the bottom of the list view display. Click the *Stop* button to stop capturing image loads. You can use the *Edit* / *Clear* menu item to clear the display.

Now you can use the *Calculate* / *Use Cache* menu item to toggle cache usage. Now click *Go* again. Notice that after some calculations, the cache starts to be useful and the "Cached?" column shows more "Yes" items when the hash value can be satisfied by the cache (figure 7-9).

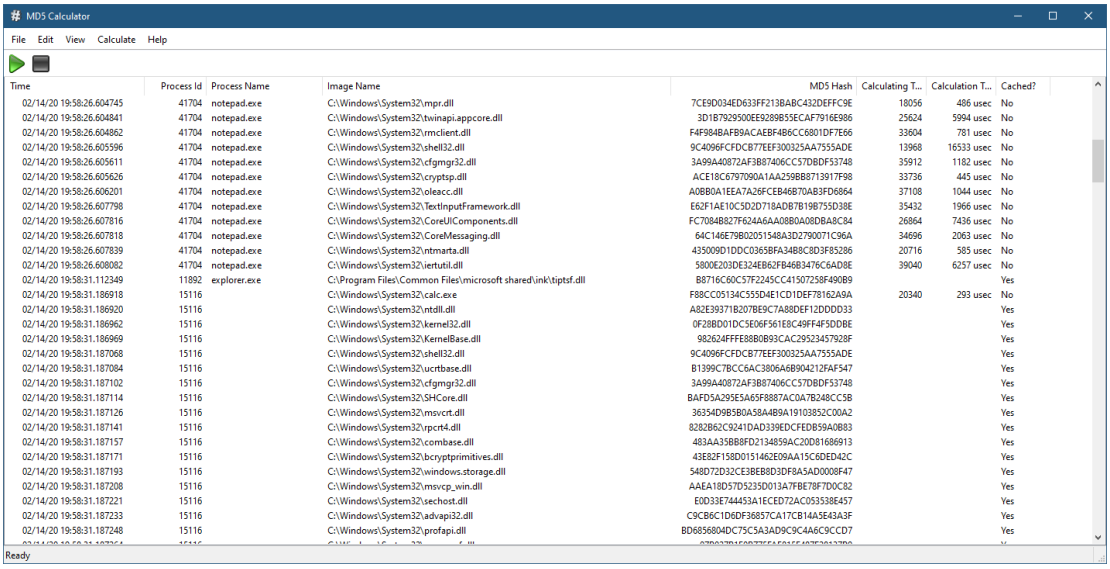


Figure 7-9: The *MD5 Calculator* application running with caching

Let's walk through the most important parts of the application.

Calculating MD5 Hash

Calculating the MD5 hash of any buffer can be done using the Windows cryptographic API. A simple class named `MD5Calculator` is used to make the calculation (part of its own static library project named *HashCalc*):

```
// MD5Calculator.h

class MD5Calculator {
public:
    static std::vector<uint8_t> Calculate(PCWSTR path);
};

// MD5Calculator.cpp

#include <wincrypt.h>
#include "MD5Calculator.h"
#include <wil\resource.h>
```

```
std::vector<uint8_t> MD5Calculator::Calculate(PCWSTR path) {
    std::vector<uint8_t> md5;

    wil::unique_hfile hFile(::CreateFile(path, GENERIC_READ, FILE_SHARE_READ,
        nullptr, OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, nullptr));
    if (!hFile)
        return md5;

    wil::unique_handle hMemMap(::CreateFileMapping(hFile.get(), nullptr,
        PAGE_READONLY, 0, 0, nullptr));
    if (!hMemMap)
        return md5;

    wil::unique_hcryptprov hProvider;
    if (!::CryptAcquireContext(hProvider.addressof(), nullptr, nullptr,
        PROV_RSA_FULL, CRYPT_VERIFYCONTEXT))
        return md5;

    wil::unique_hcrypthash hHash;
    if(!::CryptCreateHash(hProvider.get(), CALG_MD5, 0, 0, hHash.addressof()))
        return md5;

    wil::unique_mapview_ptr<BYTE> buffer((BYTE*)::MapViewOfFile(hMemMap.get(),
        FILE_MAP_READ, 0, 0, 0));
    if (!buffer)
        return md5;

    auto size = ::GetFileSize(hFile.get(), nullptr);
    if (!::CryptHashData(hHash.get(), buffer.get(), size, 0))
        return md5;

    DWORD hashSize;
    DWORD len = sizeof(DWORD);
    if (!::CryptGetHashParam(hHash.get(), HP_HASHSIZE, (BYTE*)&hashSize,
        &len, 0))
        return md5;

    md5.resize(len = hashSize);
    ::CryptGetHashParam(hHash.get(), HP_HASHVAL, md5.data(), &len, 0);
```

```
    return md5;
}
```

A detailed description of the cryptographic API is beyond the scope of this book. However, the procedure to calculate a hash is fairly straightforward. There are a few preparatory steps, before calling the actual hashing function, `CryptHashData`. This function accepts a handle representing the hashing algorithm, the buffer to hash and the size of the buffer to hash.

Here are the steps, in order, performed by `MD5Calculator::Calculate`:

1. The file in question is opened with `CreateFile` (discussed in detail in chapter 11). The file is opened for read access, with the optional flag `FILE_FLAG_SEQUENTIAL_SCAN` that is a hint to the file system that the reading is going to be sequential.
2. The contents of the file must be placed in a memory buffer to be usable by the hashing function. One way would be to allocate a buffer the size of the file and use `ReadFile` to read its contents into the buffer. A better approach is to use a *memory-mapped file* (discussed in chapter 14), that can map a file's contents to memory (see item 5 below), without any need to allocate or read anything. The `CreateFileMapping` function is used to create the file mapping object based on the file handle (first argument).
3. `CryptAcquireContext` is called to get back a cryptographic provider handle based a provider (`PROV_RSA_FULL` in our case).
4. The call to `CryptCreateHash` returns a handle to a specific hash algorithm (`CALG_MD5` for MD5).
5. Calling `MapViewOfFile` maps the file contents to memory, returning a pointer. This is wrapped by the WIL `unique_mapview_ptr<>` that calls `UnmapViewOfFile` when the variable goes out of scope.
6. Now everything is ready to call `CryptHashData` to calculate the hash.
7. All that's left to do is retrieve the hash size and hash data itself. Both are accomplished with calls to `CryptGetHashParam`: the first with `HP_HASHSIZE` to get the hash size (always 16 bytes for MD5, but the code remains generic).
8. The buffer for the result is allocated by calling `resize` on the `vector` of bytes. Then a second call to `CryptGetHashParam` is made with `HP_HASHVAL` to get the actual hash.

The Hash Cache

The cache itself is encapsulated in the `HashCache` class defined like so:

```
using Hash = std::vector<uint8_t>;

class HashCache {
public:
    HashCache();

    bool Add(PCWSTR path, const Hash& hash);
    const Hash Get(PCWSTR path) const;
    bool Remove(PCWSTR path);
    void Clear();

private:
    mutable CriticalSection _lock;
    std::unordered_map<std::wstring, Hash> _cache;
};
```

The cache is managed with an `unordered_map<>` object from the C++ standard library that maps a file path to its hash. The hash itself is stored as a `vector` of bytes, although for MD5 I could have just used an array of 16 bytes. Since accessing the cache may be done by multiple threads, the `unordered_map<>` must be protected from concurrent access. Here I'm using a critical section. The implementation is fairly straightforward, protecting each operation with the critical section:

```
HashCache::HashCache() {
    _cache.reserve(512);
}

bool HashCache::Add(PCWSTR path, const Hash& hash) {
    AutoCriticalSection locker(_lock);
    auto it = _cache.find(path);
    if (it == _cache.end()) {
        _cache.insert({ path, hash });
        return true;
    }
    return false;
}

const Hash HashCache::Get(PCWSTR path) const {
    AutoCriticalSection locker(_lock);
    auto it = _cache.find(path);
    return it == _cache.end() ? Hash() : it->second;
}
```

```
bool HashCache::Remove(PCWSTR path) {
    AutoCriticalSection locker(_lock);
    auto it = _cache.find(path);
    if (it != _cache.end()) {
        _cache.erase(it);
        return true;
    }
    return false;
}

void HashCache::Clear() {
    AutoCriticalSection locker(_lock);
    _cache.clear();
}
```

The code uses the RAII `AutoCriticalSection` class, defined earlier to acquire and release with no need for termination handlers (`__try` / `__finally`).

The main view class (`CView`) holds an instance of `HashCache` named `m_Cache`, that is used if cache usage is enabled (`m_UseCache` member).

Image Loads Notifications

The next, relatively independent, piece of the puzzle is getting notifications about image loads. One powerful way to get these notifications from user mode is to utilize *Event Tracing for Windows* (ETW). ETW is a mechanism existed since Windows 2000 that allows system components and other applications to generate rich events that can be consumed in real-time or logged to a file and analyzed later. A basic ETW architecture is shown in figure 7-10.

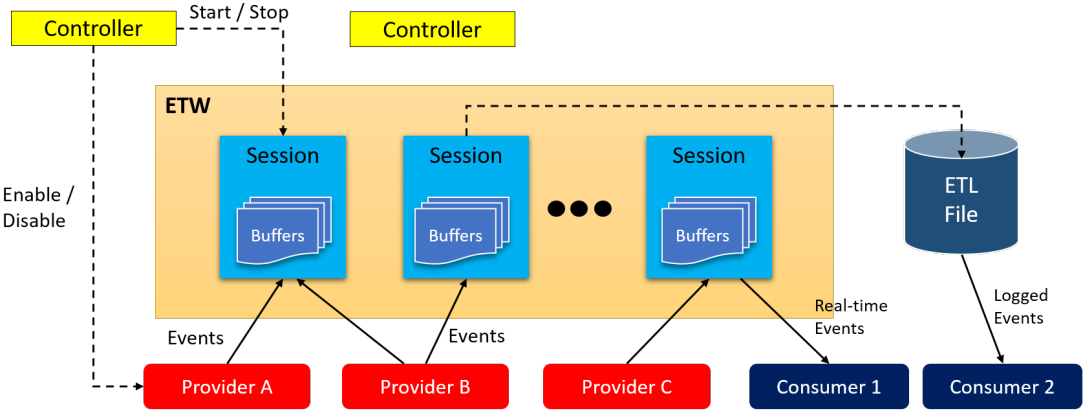


Figure 7-10: ETW Architecture

The main pieces of ETW are the following:

- Providers generate events.
- Sessions encapsulate one or more providers plus some configuration. Events are captured when the session starts until it's stopped.
- Controllers enable or disable providers, and start and stop sessions.
- Consumers consume events in real-time or to a file (*ETL* - event tracing log). In a typical case, a controller is also a consumer.

> Complete treatment of ETW is beyond the scope of this book.

In our case, we need to use the kernel provider that can send a bunch of events, one of them is image loads. The `TraceManager` class encapsulates working with the ETW infrastructure. It's defined like so:

```
class TraceManager final {
public:
    ~TraceManager();

    bool Start(std::function<void(PEVENT_RECORD)> callback);
    bool Stop();

private:
    void OnEventRecord(PEVENT_RECORD rec);
    DWORD Run();

private:
    TRACEHANDLE _handle{ 0 };
    TRACEHANDLE _hTrace{ 0 };
    EVENT_TRACE_PROPERTIES* _properties;
    std::unique_ptr<BYTE[]> _propertiesBuffer;
    EVENT_TRACE_LOGFILE _traceLog = { 0 };
    wil::unique_handle _hProcessThread;
    std::function<void(PEVENT_RECORD)> _callback;
};
```

The interface exposed by `TraceManager` is fairly simple. Once constructed, a session is started by calling `Start` and stopped by calling `Stop`. The private `Run` method is the one that starts the session running with its own thread (more on that in a moment). The `OnEventRecord` function is a callback invoked for every generated event. The various private data members are mostly concerned with building an ETW session and managing it. Let's look at the implementation.

The destructor just calls `Stop`:

```
TraceManager::~TraceManager() {
    Stop();
}
```

`Start` is a bulky function that sets up the ETW session appropriately and then initiates processing. It accepts a callback from an interested consumer to be called for each event. Its first major task is calling `StartTrace`, that configures and starts a session:

```
bool TraceManager::Start(std::function<void(PEVENT_RECORD)> cb) {
    _callback = cb;

    if (_handle || _hTrace)
        return true;

    auto size = sizeof(EVENT_TRACE_PROPERTIES) + sizeof(KERNEL_LOGGER_NAME);
    _propertiesBuffer = std::make_unique<BYTE[]>(size);
    ::memset(_propertiesBuffer.get(), 0, size);

    _properties = reinterpret_cast<EVENT_TRACE_PROPERTIES*>(_propertiesBuffer.g\
et());
    _properties->EnableFlags = EVENT_TRACE_FLAG_IMAGE_LOAD;
    _properties->Wnode.BufferSize = (ULONG)size;
    _properties->Wnode.Guid = SystemTraceControlGuid;
    _properties->Wnode.Flags = WNODE_FLAG_TRACED_GUID;
    _properties->Wnode.ClientContext = 1;
    _properties->LogFileMode = EVENT_TRACE_REAL_TIME_MODE;
    _properties->LoggerNameOffset = sizeof(EVENT_TRACE_PROPERTIES);

    auto error = ::StartTrace(&_handle, KERNEL_LOGGER_NAME, _properties);
    if (error != ERROR_SUCCESS && error != ERROR_ALREADY_EXISTS)
        return false;
```

First, it checks if a session is already in progress, and if so, it simply returns. Otherwise, it saves the callback in a data member and goes on to prepare the `EVENT_TRACE_PROPERTIES` structure. The important parts are setting `EnableFlags` to `EVENT_TRACE_FLAG_IMAGE_LOAD`, which specifies that image-related events are of interest, and `LogFileMode` to `EVENT_TRACE_REAL_TIME_MODE` to indicate a real-time session is requested.

> Check the documentation of `StartTrace` for the full details.

ETW sessions are unique in the sense that they can outlive a process. This means `StartTrace` can fail, but if the last error is `ERROR_ALREADY_EXISTS` then the session is already running, and we can just tap into it as a consumer.

Next, we need to set up the consumer, by calling `OpenTrace`:

```
    _traceLog.Context = this;
    _traceLog.LoggerName = KERNEL_LOGGER_NAME;
    _traceLog.ProcessTraceMode = PROCESS_TRACE_MODE_EVENT_RECORD |
        PROCESS_TRACE_MODE_REAL_TIME;
    _traceLog.EventRecordCallback = [](PEVENT_RECORD record) {
        ((TraceManager*)record->UserContext)->OnEventRecord(record);
    };
    _hTrace = ::OpenTrace(&_traceLog);
    if (!_hTrace)
        return false;
```

The callback for each event is set in the `EventRecordCallback` member of the `EVENT_TRACE_LOGFILE` structure (`_traceLog`). It uses the `UserContext` member as the `this` pointer (set earlier in the `Context` member) to invoke an instance function of the class. This function (`OnEventRecord`) will invoke the callback that was passed in earlier in `Start`.

The consumer is now in place, so the last operation needed is to start processing events. To that end, a separate thread is created, because the `ProcessTrace` function is a blocking one, and we don't want the caller to be blocked when calling `Start`:

```
    _hProcessThread.reset(::CreateThread(nullptr, 0, [](auto param) {
        return ((TraceManager*)param)->Run();
        }, this, 0, nullptr));

    return true;
}
```

The `Run` member function just calls `ProcessTrace`:

```
DWORD TraceManager::Run() {
    auto error = ::ProcessTrace(&_hTrace, 1, nullptr, nullptr);
    return error;
}
```

As mentioned, the `OnEventRecord` function invokes the client's callback:

```
void TraceManager::OnEventRecord(PEVENT_RECORD rec) {
    if (_callback)
        _callback(rec);
}
```

Finally, the `Stop` function closes and stops the trace:

```
bool TraceManager::Stop() {
    if (_hTrace) {
        ::CloseTrace(_hTrace);
        _hTrace = 0;
    }
    if (_handle) {
        ::StopTrace(_handle, KERNEL_LOGGER_NAME, _properties);
        _handle = 0;
    }
    return true;
}
```

The main frame class (`CMainFrm`) holds a `TraceManager` instance. It calls `Start` and `Stop` when the appropriate menu / tool bar items are selected:

```
LRESULT CMainFrame::OnStartTrace(WORD, WORD, HWND, BOOL&) {
    m_TraceManager.Start([this](auto record) {
        m_view.OnEvent(record);     // call the view
        });

    // UI updates omitted...

    return 0;
}

LRESULT CMainFrame::OnStopTrace(WORD, WORD, HWND, BOOL&) {
    m_TraceManager.Stop();

    // UI updates omitted

    return 0;
}
```

Event Parsing

An ETW event is provided to the event callback using the `EVENT_RECORD` structure that contains everything about the specific event. Here is its definition:

```
typedef struct _EVENT_RECORD {
    EVENT_HEADER        EventHeader;          // Event header
    ETW_BUFFER_CONTEXT  BufferContext;        // Buffer context
    USHORT              ExtendedDataCount;    // Number of extended
                                              // data items
    USHORT              UserDataLength;       // User data length
    PEVENT_HEADER_EXTENDED_DATA_ITEM          // Pointer to an array of
                        ExtendedData;         // extended data items
    PVOID               UserData;             // Pointer to user data
    PVOID               UserContext;          // Context from OpenTrace
} EVENT_RECORD, *PEVENT_RECORD;
```

ETW event properties can include strings (ANSI and Unicode), numbers, custom structures, and some other special types. Everything is stored in a binary blob as part of `EVENT_RECORD` starting at `UserData` address. To get to the various properties and values, some parsing is required. The `EventParser` class is a helper class for parsing properties. It stores each parsed property in an `EventProperty` structure defined like so:

```
struct EventProperty {
    EventProperty(EVENT_PROPERTY_INFO& info);

    std::wstring Name;
    BYTE* Data;
    ULONG Length;
    EVENT_PROPERTY_INFO& Info;

    template<typename T>
    T GetValue() const {
        static_assert(std::is_pod<T>() && !std::is_pointer<T>());
        return *(T*)Data;
    }

    PCWSTR GetUnicodeString() const;
    PCSTR GetAnsiString() const;
};
```

Each property has a name (`Name` member), a pointer to the actual data (`Data`), a length (`Length`) and a reference to the original structure describing the property (`Info`). The `GetValue<>` template function retrieves property values for simple POD ("plain old data") types, such as numeric types. The `static_assert` statement instructs the compiler to reject complex types, as they would produce a wrong value.

> `static_assert` was introduced in C++ 11 and enhanced in C++ 14.

`GetUnicodeString` and `GetAnsiString` return the data as their corresponding string types.

The `EventParser` class is declared like so:

```
class EventParser {
public:
    EventParser(PEVENT_RECORD record);

    PTRACE_EVENT_INFO GetEventInfo() const;
    PEVENT_RECORD GetEventRecord() const;
    const EVENT_HEADER& GetEventHeader() const;
    const std::vector<EventProperty>& GetProperties() const;
    const EventProperty* GetProperty(PCWSTR name) const;    // lookup by name

    DWORD GetProcessId() const;

    static std::wstring GetDosNameFromNtName(PCWSTR name);

private:
    std::unique_ptr<BYTE[]> _buffer;
    PTRACE_EVENT_INFO _info{ nullptr };
    PEVENT_RECORD _record;
    mutable std::vector<EventProperty> _properties;
};
```

An `EventParser` instance takes an `EVENT_RECORD` as its input. Everything about the event is stored there somewhere. `EventParser`'s task is to extract the required information. Here is the implementation except for `GetDosNameFromNtName`, which we'll deal with separately:

```
EventProperty::EventProperty(EVENT_PROPERTY_INFO& info) : Info(info) {
}

EventParser::EventParser(PEVENT_RECORD record) : _record(record) {
    ULONG size = 0;
    auto error = ::TdhGetEventInformation(record, 0, nullptr, _info, &size);
    if (error == ERROR_INSUFFICIENT_BUFFER) {
        _buffer = std::make_unique<BYTE[]>(size);
        _info = reinterpret_cast<PTRACE_EVENT_INFO>(_buffer.get());
        error = ::TdhGetEventInformation(record, 0, nullptr, _info, &size);
    }
    ::SetLastError(error);
}

PTRACE_EVENT_INFO EventParser::GetEventInfo() const {
    return _info;
}

PEVENT_RECORD EventParser::GetEventRecord() const {
    return _record;
}

const EVENT_HEADER& EventParser::GetEventHeader() const {
    return _record->EventHeader;
}

const std::vector<EventProperty>& EventParser::GetProperties() const {
    if (!_properties.empty())
        return _properties;

    _properties.reserve(_info->TopLevelPropertyCount);
    auto userDataLength = _record->UserDataLength;
    BYTE* data = (BYTE*)_record->UserData;

    for (ULONG i = 0; i < _info->TopLevelPropertyCount; i++) {
        auto& prop = _info->EventPropertyInfoArray[i];
        EventProperty property(prop);
        property.Name.assign((WCHAR*)((BYTE*)_info + prop.NameOffset));
        auto len = prop.length;
        property.Length = len;
        property.Data = data;
```

```
        data += len;
        userDataLength -= len;

        _properties.push_back(std::move(property));
    }

    return _properties;
}

const EventProperty* EventParser::GetProperty(PCWSTR name) const {
    for (auto& prop : GetProperties())
        if (prop.Name == name)
            return &prop;
    return nullptr;
}

DWORD EventParser::GetProcessId() const {
    return _record->EventHeader.ProcessId;
}

PCWSTR EventProperty::GetUnicodeString() const {
    return (PCWSTR)Data;
}

PCSTR EventProperty::GetAnsiString() const {
    return (PCSTR)Data;
}
```

The constructor calls `TdhGetEventInformation` to get the basic event details and an array of properties, all from the `EVENT_RECORD`. The call is made twice: the first time with a length of zero to get the required length, then, after allocating the required buffer, makes a second call to retrieve the actual data.

The `Tdh` functions require the header *<tdh.h>* and linking against *tdh.lib*. As mentioned earlier, a detailed discussion of these functions is outside the scope of this book.

`GetProperties` does the hard work of walking each property in the event, extracting the important information and encapsulating it in an `EventProperty` instance. The `GetProperty` helper function returns a property given its name (if any).

Putting it All Together

Now that we have all the individual pieces, we can start integrating them into an actual application. The main view class (`CView`) holds the data items that are actually displayed. These take the form of `EventData` structures defined like so (in *view.h*):

```
struct EventData {
    CString FileName;
    ULONGLONG Time;
    DWORD ProcessId;
    Hash MD5Hash;
    DWORD CalculatingThreadId;
    DWORD CalculationTime;
    bool Cached : 1;
    bool CalcDone : 1;
};
```

The view stores a `vector` of these items, a critical section to protect access to the `vector`, the cache itself, and whether it should be used:

```
class CView... {
//...
private:
    std::vector<EventData> m_Events;
    HashCache m_Cache;
    CriticalSection m_EventsLock;
    bool m_UseCache{ false };
};
```

When an event comes in, the callback `OnEvent` is invoked. The callback needs to grab the event details, store them in a new `EventData` object and go ahead and calculate the MD5 hash (or use a cached result). First, it filters out unwanted events:

```
void CView::OnEvent(PEVENT_RECORD record) {
    EventParser parser(record);
    // ID 10 is a load image event
    if (parser.GetEventHeader().EventDescriptor.Opcode != 10)
        return;
```

Remember the ETW trace only enables the image load type of events. As it turns out, there are actually four events that may be sent. Only one of them (with an opcode of 10) is an image load.

Refer to this reference[3] for the full details.
Here is the structure for the Image load set of events:

```
[EventType(10, 2, 3, 4), EventTypeName("Load", "Unload", "DCStart", "DCEnd")]
class Image_Load : Image {
  uint32 ImageBase;
  uint32 ImageSize;
  uint32 ProcessId;
  uint32 ImageCheckSum;
  uint32 TimeDateStamp;
  uint32 Reserved0;
  uint32 DefaultBase;
  uint32 Reserved1;
  uint32 Reserved2;
  uint32 Reserved3;
  uint32 Reserved4;
  string FileName;
};
```

> The above format is known as *simplified MOF* (Managed Object Format). It's used with ETW and *Windows Management Instrumentation* (WMI).

Once the proper event is received (Opcode=10), the interesting property (*FileName*) is retrieved and used to fill a new `EventData` instance, before initiating computation of the MD5 hash by posting a custom message to the view:

```
    auto fileName = parser.GetProperty(L"FileName");
    if (fileName) {
        EventData data;
        data.FileName = parser.GetDosNameFromNtName(
            fileName->GetUnicodeString()).c_str();
        data.ProcessId = parser.GetProcessId();
        data.Time = parser.GetEventHeader().TimeStamp.QuadPart;
        data.CalcDone = false;
        size_t size;
        {
            AutoCriticalSection locker(m_EventsLock);
```

[3] https://docs.microsoft.com/en-us/windows/win32/etw/image-load

```
            m_Events.push_back(std::move(data));
            size = m_Events.size();
        }
        int index = static_cast<int>(size - 1);

        // initiate work from the UI thread

        PostMessage(WM_START_CALC, index, size);
    }
}
```

There are a few points worth noting:

- The critical section protecting the view's data is held for as short time as possible. This is achieved by using an artificial block so that the critical section can be released as soon as the block is exited.
- Initiating the calculation is not done in this function, as it's called by the `TraceManager`'s thread, which should be let go as soon as possible to be available to process the next event. Instead, calling `PostMessage` causes a message to be asynchronously sent to the window, to be handled by the UI thread, allowing the current function to return.

The last interesting detail of the above code is the use `GetDosNameFromNtName`. The file name provided by the ETW event is in "device format", which is the native NT format looking something like this: "DeviceHarddiskVolume3SomeDirectorySomeFile.dll". The reason internal device names have this format will be discussed in chapter 11. For now, this kind of path should be translated to something like "c:SomeDirectorySomeFile.dll" to be usable by the `CreateFile` API used for the hash calculation. The static `EventParser::GetDosNameFromNtName` is used to translate back to a Win32 device name:

```
std::wstring EventParser::GetDosNameFromNtName(PCWSTR name) {
    static std::vector<std::pair<std::wstring, std::wstring>> deviceNames;
    static bool first = true;

    if (first) {
        auto drives = ::GetLogicalDrives();
        int drive = 0;
        while (drives) {
            if (drives & 1) {
                // drive exists
                WCHAR driveName[] = L"X:";
                driveName[0] = (WCHAR)(drive + 'A');
```

```
                WCHAR path[MAX_PATH];
                if (::QueryDosDevice(driveName, path, MAX_PATH)) {
                    deviceNames.push_back({ path, driveName });
                }
            }
            drive++;
            drives >>= 1;
        }
        first = false;
    }

    for (auto& [ntName, dosName] : deviceNames) {
        if (::_wcsnicmp(name, ntName.c_str(), ntName.size()) == 0)
            return dosName + (name + ntName.size());
    }
    return L"";
}
```

The function uses a static `vector` of string pairs, each mapping an NT device name to a drive letter. The first time the function is called, it gets all the driver letters (`GetLogicalDrives`) as a bitmask, where bit 0 corresponds to drive A, bit 1 to drive B, bit 2 to driver C, and so on, and for each drive queries for its NT device name with `QueryDosDevice` (see chapter 11 for more on `QueryDosDevice`).

With the passed-in path at hand, the device name is searched in the vector and its corresponding driver letter is extracted. Finally, the rest of the path is appended to the extracted drive letter and returned to the caller.

Reader Writer Locks

Using critical sections to protect shared data from concurrent access works well, but it's a pessimistic mechanism - it allows one thread at most to access the shared data. In some scenarios, where some threads read the data, and other threads write the data, an optimization can be made: If one thread reads the data, there is no reason to prevent other threads that only read the data from doing it concurrently. This is exactly the role of the "Single Writer Multiple Readers" mechanism.

The Windows API provides the `SRWLOCK` structure that represents such a lock (S is for "Slim"). its definition is as follows:

```
typedef RTL_SRWLOCK SRWLOCK, *PSRWLOCK;
```

What about that `RTL_SRWLOCK`? Here goes:

```
typedef struct _RTL_SRWLOCK {
    PVOID Ptr;
} RTL_SRWLOCK, *PRTL_SRWLOCK;
```

Clearly, this is just an opaque data that should be treated as such.

Initializing an `SRWLOCK` is accomplished with `InitializeSRWLock`:

```
void InitializeSRWLock(_Out_ PSRWLOCK SRWLock);
```

Alternatively, the structure can be initialized statically by assigning it to `SRWLOCK_INIT` macro, which just zeros out the structure. Curiously enough, there is no "delete" for the `SRWLOCK`; this is because all of its internal information is packed into that pointer-sized cell.

With an initialized `SRWLOCK`, threads can attempt to acquire the shared or exclusive lock using the following functions:

```
void AcquireSRWLockShared    (_InOut_ PSRWLOCK SRWLock);
void AcquireSRWLockExclusive (_InOut_ PSRWLOCK SRWLock);
```

If the relevant lock cannot be acquired, the thread enters a wait state. Once acquired, the thread can make forward progress and access the shared resource as specified, meaning it's up to the thread not to do the "wrong" access. For example, if a thread acquires the shared lock, it must not modify the shared data.

Once the work is done, the thread uses the associated release function:

```
void ReleaseSRWLockShared    (_Inout_ PSRWLOCK SRWLock);
void ReleaseSRWLockExclusive (_Inout_ PSRWLOCK SRWLock);
```

SRW locks store very little state, so their flexibility is limited:

- A shared lock owner cannot directly upgrade its lock to an exclusive one. It must first release its shared lock and then contend for the exclusive lock.
- An exclusive owner cannot recursively acquire a lock; this causes a deadlock.
- There is no guarantee that the first thread acquiring a lock is the first to receive it. As the documentation states: "SRW locks are neither fair nor FIFO".

Assuming these limitations are acceptable, performance gains are possible if most operations on the data are read rather than write.

RAII Wrappers

As with critical sections, it's convenient to have RAII wrappers for SRWLOCKs. Here are three classes, one for wrapping SRWLOCK and the others for acquiring/releasing:

```
class ReaderWriterLock : public SRWLOCK {
public:
    ReaderWriterLock();
    ReaderWriterLock(const ReaderWriterLock&) = delete;
    ReaderWriterLock& operator=(const ReaderWriterLock&) = delete;

    void LockShared();
    void UnlockShared();

    void LockExclusive();
    void UnlockExclusive();
};

struct AutoReaderWriterLockExclusive {
    AutoReaderWriterLockExclusive(SRWLOCK& lock);
    ~AutoReaderWriterLockExclusive();

private:
    SRWLOCK& _lock;
};

struct AutoReaderWriterLockShared {
    AutoReaderWriterLockShared(SRWLOCK& lock);
    ~AutoReaderWriterLockShared();

private:
    SRWLOCK& _lock;
};
```

The implementation is fairly straightforward:

```
ReaderWriterLock::ReaderWriterLock() {
    ::InitializeSRWLock(this);
}

void ReaderWriterLock::LockShared() {
    ::AcquireSRWLockShared(this);
}

void ReaderWriterLock::UnlockShared() {
    ::ReleaseSRWLockShared(this);
}

void ReaderWriterLock::LockExclusive() {
    ::AcquireSRWLockExclusive(this);
}

void ReaderWriterLock::UnlockExclusive() {
    ::ReleaseSRWLockExclusive(this);
}

AutoReaderWriterLockExclusive::AutoReaderWriterLockExclusive(SRWLOCK& lock)
    : _lock(lock) {
    ::AcquireSRWLockExclusive(&_lock);
}

AutoReaderWriterLockExclusive::~AutoReaderWriterLockExclusive() {
    ::ReleaseSRWLockExclusive(&_lock);
}

AutoReaderWriterLockShared::AutoReaderWriterLockShared(SRWLOCK& lock)
    : _lock(lock) {
    ::AcquireSRWLockShared(&_lock);
}

AutoReaderWriterLockShared::~AutoReaderWriterLockShared() {
    ::ReleaseSRWLockShared(&_lock);
}
```

> These wrappers are part of the *Threadinghelpers* project.

MD5 Calculator 2

With the MD5 Calculator, we can replace some of the critical sections for SRW locks to potentially improve concurrency, as multiple read operations may be happening at the same time. For example, the hash cache use of a critical section can be replaced with an `SRWLOCK`:

```
class HashCache {
public:
    HashCache();

    bool Add(PCWSTR path, const Hash& hash);
    const Hash Get(PCWSTR path) const;
    bool Remove(PCWSTR path);
    void Clear();

private:
    mutable ReaderWriterLock _lock;
    std::unordered_map<std::wstring, Hash> _cache;
};
```

And the implementation:

```
bool HashCache::Add(PCWSTR path, const Hash& hash) {
    AutoReaderWriterLockExclusive locker(_lock);
    auto it = _cache.find(path);
    if (it == _cache.end()) {
        _cache.insert({ path, hash });
        return true;
    }
    return false;
}

const Hash HashCache::Get(PCWSTR path) const {
    AutoReaderWriterLockShared locker(_lock);
    auto it = _cache.find(path);
```

```
    return it == _cache.end() ? Hash() : it->second;
}

bool HashCache::Remove(PCWSTR path) {
    AutoReaderWriterLockExclusive locker(_lock);
    auto it = _cache.find(path);
    if (it != _cache.end()) {
        _cache.erase(it);
        return true;
    }
    return false;
}

void HashCache::Clear() {
    AutoReaderWriterLockExclusive locker(_lock);
    _cache.clear();
}
```

Similar modifications can be made for the `CView` class.

> The above changes can be found in the *MD5Calculator2* project.

Lastly, SRW locks support *Try* variants for acquiring the locks:

```
BOOLEAN TryAcquireSRWLockExclusive (_Inout_ PSRWLOCK SRWLock);

BOOLEAN TryAcquireSRWLockShared    (_Inout_ PSRWLOCK SRWLock);
```

These functions return `TRUE` if the specified lock is acquired, and `FALSE` otherwise. If it is acquired, the normal associated *Release* function must be eventually called.

Condition Variables

Condition variables are another synchronization mechanism, providing the capability to wait on a critical section or SRW lock until some condition occurs. A classic example of using condition variables is a producer/consumer scenario. Suppose some threads produce data items and place them in a queue. Each thread does whatever work is needed to produce the items. At the same

time, other threads act as consumers - each removing an item from the queue and processes it in some way (figure 7-11).

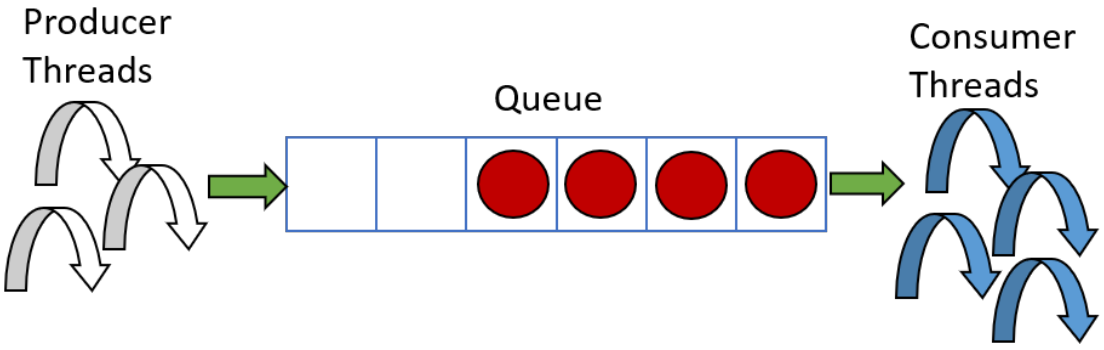


Figure 7-11: Producer/consumer

If items are produced faster than consumers can process, then the queue is non-empty, and consumers continue working. On the other hand, if consumer threads process all items, they should go into a wait state until new items are produced, in which case they should be awoken. This is exactly the behavior provided by condition variables. Consumer threads that have nothing to do (the queue is empty) should not spin, checking periodically if the queue becomes non-empty, because this consumes CPU cycles for no good reason. Condition variables allow an efficient wait (that consumes no CPU) until threads are awakened (typically by producer threads).

A condition variable is represented by a `CONDITION_VARIABLE` opaque structure, very similar to `SRWLOCK`. It must be initialized by calling `InitializeConditionVariable`:

```
void InitializeConditionVariable(_Out_ PCONDITION_VARIABLE ConditionVariable);
```

As with `SRWLOCK`, a static initialization is possible by setting the `CONDITION_VARIABLE` to `CONDITION_VARIABLE_INIT`.

A condition variable is always associated with a critical section or SRW lock. When a thread needs to wait until a condition variable is signaled, it first must acquire the critical section/SRW lock and then call the associated sleep function:

```
BOOL SleepConditionVariableCS(
    _Inout_ PCONDITION_VARIABLE ConditionVariable,
    _Inout_ PCRITICAL_SECTION CriticalSection,
    _In_ DWORD dwMilliseconds);

BOOL SleepConditionVariableSRW(
    _Inout_ PCONDITION_VARIABLE ConditionVariable,
    _Inout_ PSRWLOCK SRWLock,
    _In_ DWORD dwMilliseconds,
    _In_ ULONG Flags);
```

The thread calling one of the above `Sleep*` functions must have first acquired the associated synchronization object exactly once. The function releases the synchronization object and waits on the condition variable, atomically. While waiting, the thread may be woken by calling one of the wake functions on the condition variable:

```
VOID WakeConditionVariable    (_Inout_ PCONDITION_VARIABLE ConditionVariable);

VOID WakeAllConditionVariable (_Inout_ PCONDITION_VARIABLE ConditionVariable);
```

`WakeConditionVariable` wakes a single thread (no guarantee on which thread it is if multiple threads are sleeping on the condition variable), while `WakeAllConditionVariable` wakes all threads waiting on the condition variable.

Once woken, the thread re-acquires the synchronization object and continues execution. At this time, the thread should recheck the condition for which it was waiting, and if not satisfied, call the `Sleep*` function again. This can happen as another thread might have woken up just before this one and did some work that made the condition false again. The operations of such a thread are illustrated in figure 7-12 (using a critical section).

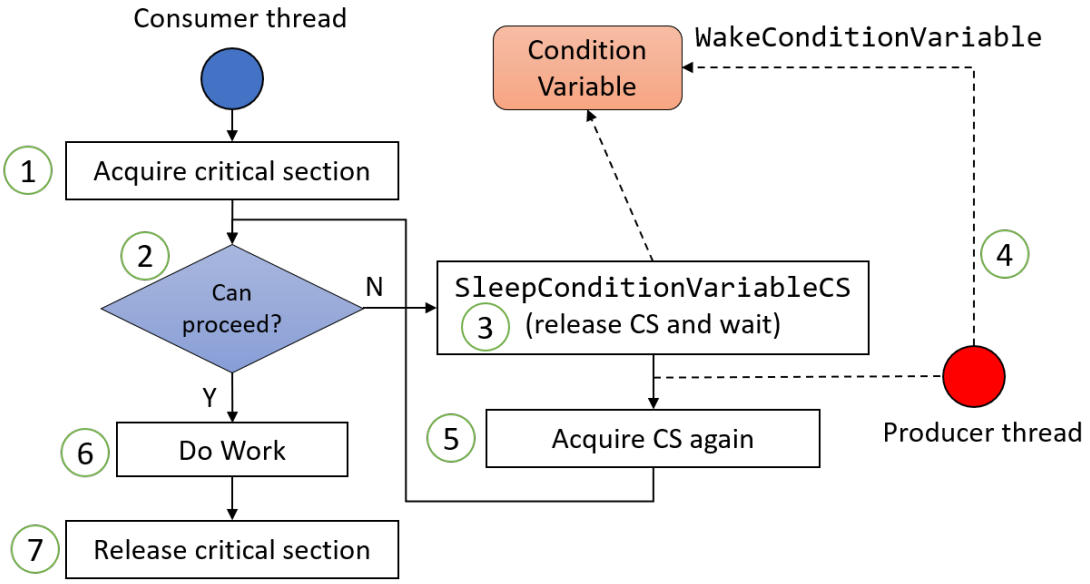


Figure 7-12: Consumer thread operation with a condition variable

The steps involved in figure 7-12 are the following:

1. The consumer thread acquires the critical section.
2. The thread checks if it's OK to proceed. For example, it may check if the queue it's supposed to work on is not empty.
3. If it is empty, the thread calls `SleepConditionVariableCS`, which releases the critical section (so another thread can acquire it) and goes to sleep (wait state).
4. At some point, a producer thread will wake the consumer thread by calling `WakeConditionVariable` because, for example, it added a new item to the queue.
5. `SleepConditionVariableCS` returns, acquires the critical section and goes back to check if it's OK to proceed. If not, it resumes waiting.
6. Now that it's OK to proceed, the thread can do its work (such as removing an item from the queue). The critical section is still held.
7. Finally, the work is done and the critical section must be released.

Back to the `Sleep*` functions: These functions return `TRUE` if successful, meaning the thread has woken up with the synchronization primitive acquired. If these return `FALSE`, it means a possible error occurred. If the `dwMillisecond` parameter is not `INFINITE`, it signifies an error. If the time interval is finite, a `FALSE` might indicate the thread was not woken up during that interval. In this case, `GetLastError` returns `ERROR_TIMEOUT`.

For SRW locks, the `Flags` parameter indicates if this should be an exclusive acquire or not. Passing zero means exclusive access, while passing `CONDITION_VARIABLE_LOCKMODE_SHARED` means shared access.

The Queue Demo Application

The *Queue Demo* application demonstrates the use of a condition variable to wake consumer threads accessing a shared queue with producer threads. Once launched, it allows selecting the number of producer and consumer threads (figure 7-13).

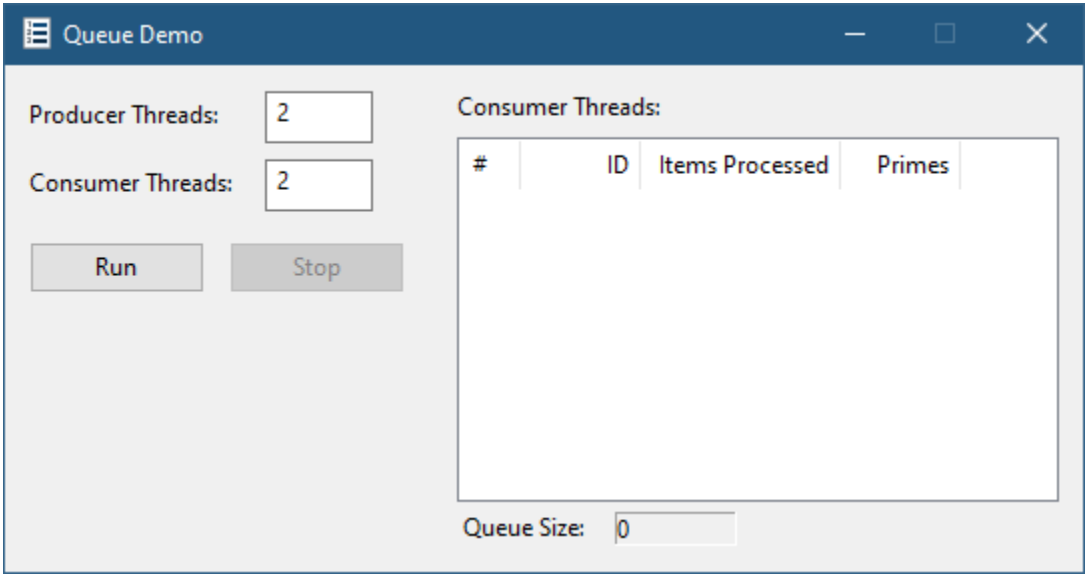


Figure 7-13: The *Queue Demo* application

Click the *Run* button to start the action. Producer threads produce items (which are numbers) and push them into a queue. Consumer threads pop items from the queue and check if the numbers are prime or not. If the queue is empty, a consumer thread sleeps on a condition variable until woken up by a producer thread. The current queue size is shown at the bottom and is updated periodically.

If the producer threads produce more items than the consumer threads can handle, the queue size will increase, as the consumers try to catch up. Clicking *Stop* stops the producers, allowing the consumers to catch up and drain the queue. If, on the other hand, the consumers are "quicker" (perhaps because there are many of them), the queue size will be mostly zero, as consumers are quick enough to pick up any new item and process it before a new one appears in the queue. While working, some statistics are displayed for the consumer threads (figure 7-14). You'll have to "play" with the thread counts to get interesting behavior.

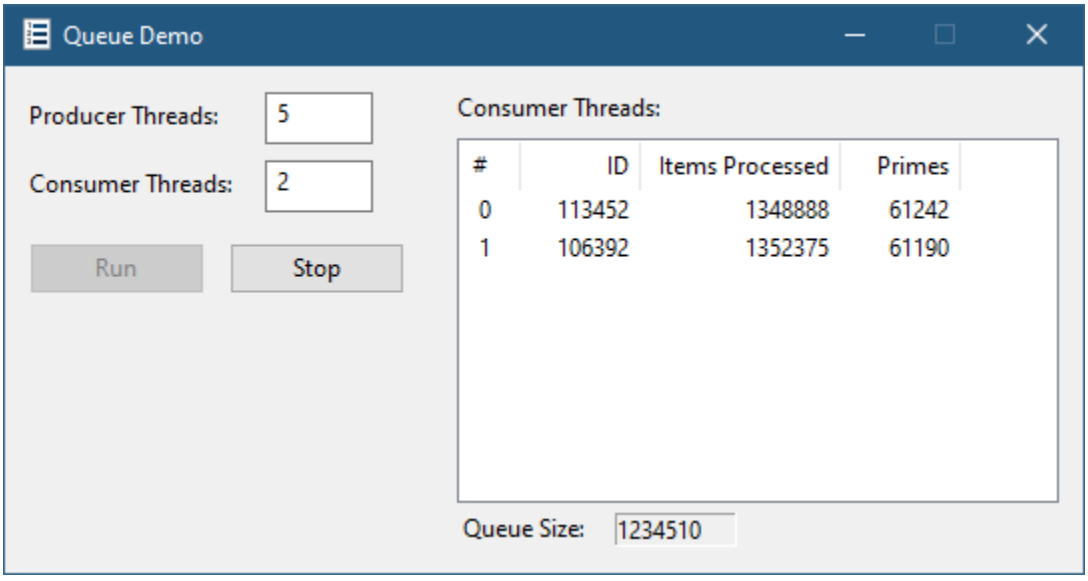


Figure 7-14: The *Queue Demo* application at work

The `CMainDlg` class defines the following nested types:

```
struct ConsumerThreadData {
    unsigned ItemsProcessed{ 0 };
    unsigned Primes{ 0 };
    wil::unique_handle hThread;
};

struct WorkItem {
    unsigned Data;
    bool IsPrime;
};
```

`ConsumerThreadData` is the data structure manipulated by a consumer thread. There is one such object per consumer thread. It stores a handle to the thread, and the count of items processed and prime numbers found. Each work item stored in the queue is made up of a the number to determine if it's prime and a result (which is not used in the application directly).

Based on these structures and the application's requirements, the following data members are stored:

```
std::queue<WorkItem> m_Queue;    // the queue
CriticalSection m_QueueLock;     // the critical section protecting the queue
CONDITION_VARIABLE m_QueueCondVar;
std::vector<wil::unique_handle> m_ProducerThreads;
std::vector<ConsumerThreadData> m_ConsumerThreads;
wil::unique_handle m_hAbortEvent;
static CMainDlg* m_pThis;        // simplifies access to this
```

Producer threads just store their handles, but they could potentially store more state, similar to the consumer threads. The `CriticalSection` class is the wrapper from the *ThreadingHelpers* project to simplify working with it. The `m_hAbortEvent` is an event kernel object handle that is used to signal producer and consumer threads to stop running. Event kernel objects are discussed in detail in the next chapter. As an alternative for this application, a volatile boolean variable could have been used instead. Finally, The `m_pThis` static member is used to refer to the only dialog instance to simplify accessing instance methods for thread functions.

The `CMainDlg::OnInitDialog` function performs single initialization of the dialog box in terms of controls, but also initializes `m_pThis` and the abort event:

```
LRESULT CMainDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL&) {
    m_pThis = this;

    m_hAbortEvent.reset(::CreateEvent(nullptr, TRUE, FALSE, nullptr));
    //...
```

Clicking the *Run* button causes a call to `OnRun` which just calls `Run` to do the actual work. The function starts by getting the number of consumer and producer threads and performs some sanity checks:

```
void CMainDlg::Run() {
    int consumers = GetDlgItemInt(IDC_CONSUMERS);
    if (consumers < 1 || consumers > 64) {
        DisplayError(L"Consumer threads must be between 1 and 64");
        return;
    }
    int producers = GetDlgItemInt(IDC_PRODUCERS);
    if (producers < 1 || producers > 64) {
        DisplayError(L"Producer threads must be between 1 and 64");
        return;
    }
```

> There is nothing special about the number 64 in the above code. Larger numbers can easily be selected if desired.

Next, some initialization should be done for this run:

```
bool abort = false;
::ResetEvent(m_hAbortEvent.get());
::InitializeConditionVariable(&m_QueueCondVar);
m_ThreadList.DeleteAllItems();
```

The abort event is reset and the condition variable is initialized. The list view showing the consumer threads is cleared of any existing items. Now it's time to create the consumer threads:

```
m_ConsumerThreads.clear();
m_ConsumerThreads.reserve(consumers);

for (int i = 0; i < consumers; i++) {
    ConsumerThreadData data;
    data.hThread.reset(::CreateThread(nullptr, 0, [](auto p) {
        return m_pThis->ConsumerThread(PtrToLong(p));
        }, LongToPtr(i), 0, nullptr));
    if (!data.hThread) {
        abort = true;
        break;
    }
    m_ConsumerThreads.push_back(std::move(data));
}
if (abort) {
    ::SetEvent(m_hAbortEvent.get());
    return;
}
```

Each consumer thread is created with a normal `CreateThread`, pointing the thread function to the instance function `ConsumerThread` with a value indicating the index of the consumer thread in the consumer thread array.

You might think it would be easier to just pass the thread function a pointer to the `ConsumerThreadData` instance (`data` in the code above). This will lead to a crash or corruption, because the data is on the stack and then will be copied to the `vector` (and so moved to the heap) making the pointer a garbage one. In this case, I decided to pass in the index as it's not going to change.

If, for some reason, thread creation fails, the loop is aborted and the event is set to cause all previously created producer threads to abort.

Next, producer threads are created in a similar fashion:

```
m_ProducerThreads.clear();
m_ProducerThreads.reserve(producers);

for (int i = 0; i < producers; i++) {
    wil::unique_handle hThread(::CreateThread(nullptr, 0, [](auto p) {
        return m_pThis->ProducerThread();
        }, this, 0, nullptr));
    if (!hThread) {
        DisplayError(L"Failed to create producer thread. Aborting");
        abort = true;
        break;
    }
}

if (abort) {
    ::SetEvent(m_hAbortEvent.get());
    return;
}
```

Producer threads call the `ProducerThread` instance function. Contrary to consumer threads, these don't require any special context, since they just generate pseudo-random numbers. The last part of the `Run` function is adding the consumer threads basic information to the list view and start a simple timer used to periodically update the queue size:

```
CString text;
for (int i = 0; i < (int)m_ConsumerThreads.size(); i++) {
    const auto& t = m_ConsumerThreads[i];
    text.Format(L"%2d", i);
    int n = m_ThreadList.InsertItem(i, text);
    m_ThreadList.SetItemText(n, 1,
        std::to_wstring(::GetThreadId(t.hThread.get())).c_str());
}

GetDlgItem(IDC_RUN).EnableWindow(FALSE);
GetDlgItem(IDC_STOP).EnableWindow(TRUE);

SetTimer(1, 500, nullptr);
```

Here is a producer thread's code:

```
DWORD CMainDlg::ProducerThread() {
    for (;;) {
        if (::WaitForSingleObject(m_hAbortEvent.get(), 0) == WAIT_OBJECT_0)
            break;

        WorkItem item;
        item.IsPrime = false;
        LARGE_INTEGER li;
        ::QueryPerformanceCounter(&li);
        item.Data = li.LowPart;
        {
            AutoCriticalSection locker(m_QueueLock);
            m_Queue.push(item);
        }
        ::WakeConditionVariable(&m_QueueCondVar);

        // sleep a little bit from time to time
        if ((item.Data & 0x7f) == 0)
            ::Sleep(1);
    }
    return 0;
}
```

The code uses an infinite loop which is only broken out of if the abort event is signaled. Then a `WorkItem` instance is prepared, and the number generated uses the low 32 bit of whatever

`QueryPerformanceCounter` returns. This choice is completely arbitrary in this example. Next, the thread acquires the critical section to prevent synchronization issues on the work items queue, since it's accessed concurrently by producers and consumers (and even the UI thread may need access). The queue itself is the standard C++ `std::queue<>` class, but any other queue implementation will do.

Once a new item is added to the queue, the thread signals the condition variable to wake up a thread that is waiting on it by calling `WakeConditionVariable`. The last bit of code before looping to push the next item is a possible sleep to delay the thread a bit.

The consumer threads' code is shown below:

```
DWORD CMainDlg::ConsumerThread(int index) {
    auto& data = m_ConsumerThreads[index];
    auto tick = ::GetTickCount64();

    for (;;) {
        WorkItem value;
        {
            bool abort = false;
            AutoCriticalSection locker(m_QueueLock);
            while (m_Queue.empty()) {
                if (::WaitForSingleObject(m_hAbortEvent.get(), 0) == WAIT_OBJEC\
T_0) {
                    abort = true;
                    break;
                }
                ::SleepConditionVariableCS(&m_QueueCondVar, &m_QueueLock, INFIN\
ITE);
            }
            if (abort)
                break;

            ATLASSERT(!m_Queue.empty());
            value = m_Queue.front();
            m_Queue.pop();
        }

        {
            // do the actual work

            bool isPrime = IsPrime(value.Data);
```

```
            if (isPrime) {
                value.IsPrime = true;
                ::InterlockedIncrement(&data.Primes);
            }
            ::InterlockedIncrement(&data.ItemsProcessed);
        }
        auto current = ::GetTickCount64();
        if (current - tick > 600) {
            PostMessage(WM_UPDATE_THREAD, index);
            tick = current;
        }
    }

    PostMessage(WM_UPDATE_THREAD, index);

    return 0;
}
```

The function starts by getting a reference to the data structure for this consumer thread. Then an infinite loop is constructed. The critical section is acquired for the queue and then an inner while loop is entered if the queue is empty. If it's empty, the thread has nothing to do, so it calls `SleepConditionVariableCS` to enter a wait state until awakened by another thread using condition variable. Before waiting, it releases the critical section. When it wakes up (because a producer called `WakeConditionVariable`), it automatically acquires the critical section again (see figure 7-12), and must recheck the condition (queue empty), because it's possible that another consumer thread woke up a bit earlier and popped the last item from the queue. Also, condition variables may be prone to spurious wake ups, which is another reason to recheck the condition.

If the queue is not empty, the consumer can go ahead and remove the first item in the queue (at this time the critical section is still held) by calling `m_Queue.front()` to get the item and `m_Queue.pop()` to drop it from the queue.

The critical section is then released (scope ended with `AutoCriticalSection`) and then actual work on the item is performed by calling the helper `IsPrime` function. The relevant counters maintained by this thread are incremented if needed. The increments are performed with `InterlockedIncrement` because the UI thread accesses these values possibly concurrently. Finally, a message is posted to the window to update the statistics of this thread every 600 milliseconds or so.

The `WM_UPDATE_THREAD` application-defined message receives the consumer thread index and updates the number of items processed and the number of primes calculated:

```
LRESULT CMainDlg::OnUpdateThread(UINT, WPARAM index, LPARAM, BOOL&) {
    auto& data = m_ConsumerThreads[index];
    int n = (int)index;

    CString text;
    text.Format(L"%u", ::InterlockedAdd((LONG*)&data.ItemsProcessed, 0));
    m_ThreadList.SetItemText(n, 2, text);
    text.Format(L"%u", ::InterlockedAdd((LONG*)&data.Primes, 0));
    m_ThreadList.SetItemText(n, 3, text);

    return 0;
}
```

Accessing the counters may be done concurrently with respect to the consumer thread, so to prevent possible torn reads, `InterlockedAdd` with zero is used to mitigate that. Reading the counters directly would probably be fine too, but it may depend on the values' alignment in memory and on the target processor, so it's better to be safe than sorry.

The other UI update is the periodic queue size update by using a timer:

```
LRESULT CMainDlg::OnTimer(UINT, WPARAM id, LPARAM, BOOL&) {
    if (id == 1) {
        size_t size;
        {
            AutoCriticalSection locker(m_QueueLock);
            size = m_Queue.size();
        }
        SetDlgItemInt(IDC_QUEUE_SIZE, (unsigned)size, FALSE);
    }
    return 0;
}
```

The timer ID used is 1, checked by the `if` statement. Any access to the queue should be done under the protection of the critical section, and then the queue size is read before updating the UI.

Lastly, clicking the *Stop* button calls the `OnStop` function which simply calls `Stop`:

```
void CMainDlg::Stop() {
    // signal threads to abort

    ::SetEvent(m_hAbortEvent.get());
    ::WakeAllConditionVariable(&m_QueueCondVar);

    // update UI

    GetDlgItem(IDC_RUN).EnableWindow(TRUE);
    GetDlgItem(IDC_STOP).EnableWindow(FALSE);
}
```

The function sets the abort event to cause all producers to exit their infinite loop. The condition variable is used to wake all consumer threads, so they can drain the queue if any items are left in it.

Write a RAII wrapper for condition variables.

Change the *Queue Demo* application to use an SRW lock instead of a critical section.

Waiting on Address

Windows 8 and Server 2012 adds another synchronization mechanism, that allows a thread to wait efficiently until the value at some address changes to a desired value. Then it can wake up and proceed with its work. It's certainly possible to use other synchronization mechanisms to achieve a similar effect, such as using a condition variable, but waiting on address is more efficient and is not prone to deadlocks since no critical sections (or other software synchronization primitives) are used directly.

A thread can enter a wait state until a certain value appears on a "monitored" data by calling `WaitOnAddress`:

```
BOOL WaitOnAddress(
    _In_ volatile VOID* Address,
    _In_ PVOID CompareAddress,
    _In_ SIZE_T AddressSize,
    _In_opt_ DWORD dwMilliseconds);
```

The functions in this section require linking against the *synchronization.lib* import library.

The function checks if the value at `*Address` is the same as in `*CompreAddress`. If they are different, the call returns immediately with a value of `TRUE`. Otherwise, the thread enters a wait state. The values size to compare is specified in the `AddressSize` parameter, and it must be 1, 2, 4 or 8. The last parameter indicates the time to wait, which can be `INFINITE` to wait however long it takes.

> Internally, the kernel keeps waited on addresses in a hash table, keyed by the address.

Some other thread may change the value in `*Address`. Unfortunately, this does not automatically cause the waiting thread to wake. Instead, the thread that made the change must call one of the "wake" functions:

```
VOID WakeByAddressSingle (_In_ PVOID Address);

VOID WakeByAddressAll    (_In_ PVOID Address);
```

With the former function, Any threads waiting on the specified address are woken, while the latter wakes just one thread. Spurious wakes are also possible with this mechanism, so the woken thread should recheck if the value is indeed the expected one. If not, the thread should go back to waiting by calling `WaitOnAddress` again (typically done with a loop).

A typical code could look something like this:

```
DWORD undesiredValue = 0;
DWORD actualValue = 0;

void Thread1() {
    // set undesiredValue as appropriate

    while(actualValue == undesiredValue) {
        ::WaitOnAddress(&actualValue, &undesiredValue, sizeof(DWORD), INFINITE);
    }

    // actualValue != undesiredValue
}

void Thread2() {
    //...
    actualValue++;
```

```
    ::WakeByAddressSingle(&actualValue);
}
```

Synchronization Barriers

Another synchronization primitive introduced in Windows 8 is a *synchronization barrier*. This object allows synchronizing threads that need to get to a certain point in their work before they all can continue. For example, suppose there are several parts of a system, each of which needs to be initialized in two phases, before the main application code can continue. One simple way to accomplish that is to call each initialization function sequentially:

```
void RunApp() {
    // phase 1

    InitSubsystem1();
    InitSubsystem2();
    InitSubsystem3();
    InitSubsystem4();

    // phase 2

    InitSubsystem1Phase2();
    InitSubsystem2Phase2();
    InitSubsystem3Phase2();
    InitSubsystem4Phase2();

    // go ahead and run main application code...
}
```

This works, but if each of the initializations can be done concurrently, so that each initialization is carried out by a different thread. Each thread must not continue to phase 2 initialization until all other threads are done with phase 1. It is, of course, possible to implement such a scheme by using a combination of other synchronization primitives, but a synchronization barrier already exists for this kind of purpose.

A synchronization barrier is represented by the `SYNCHRONIZATION_BARRIER` opaque structure that must be initialized with `InitializeSynchronizationBarrier`:

```
BOOL InitializeSynchronizationBarrier(
    _Out_ LPSYNCHRONIZATION_BARRIER lpBarrier,
    _In_ LONG lTotalThreads,
    _In_ LONG lSpinCount);
```

`lTotalThreads` is the total number of threads that need to reach the barrier before they all can continue. The `lSpinCount` parameter allows setting a spin count for the threads arriving at the barrier before entering a wait state (if the barrier is not yet released). A value of `-1` sets a default spin.

> The documentation states that the default spin is 2000. However, as far as I can tell the spin count is not currently used.

Once initialized, threads that need to wait at the barrier call `EnterSynchronizationBarrier`:

```
BOOL EnterSynchronizationBarrier(
    _Inout_ LPSYNCHRONIZATION_BARRIER lpBarrier,
    _In_ DWORD dwFlags);
```

Specifying the `SYNCHRONIZATION_BARRIER_FLAGS_SPIN_ONLY` flag causes the thread to spin until the barrier is released. This should only be used if the anticipated time for barrier release is small. The opposite flag, `SYNCHRONIZATION_BARRIER_FLAGS_BLOCK_ONLY`, specifies that no spinning should be done if the barrier is not released yet and then thread should enter a wait state. The last flag, `SYNCHRONIZATION_BARRIER_FLAGS_NO_DELETE` is a possible optimization that tells the API to skip some synchronization required when deleting the barrier. If specified, all threads entering the barrier must specify this flag.

The function returns `TRUE` for a single thread only once the barrier is released, and `FALSE` for all other threads. In the previously described scenario, here is one of the initialization functions running in a separate thread:

```
DWORD WINAPI InitSubSystem1(PVOID p) {
    auto barrier = (PSYNCHRONIZATION_BARRIER)p;

    // phase 1
    printf("Subsystem 1: Starting phase 1 initialization (TID: %u)...\n",
        ::GetCurrentThreadId());
    // do work...
    printf("Subsystem 1: Ended phase 1 initialization...\n");
    ::EnterSynchronizationBarrier(barrier, 0);

    printf("Subsystem 1: Starting phase 2 initialization...\n");
    // do work
    printf("Subsystem 1: Ended phase 2 initialization...\n");

    return 0;
}
```

After phase 1 initialization is complete, `EnterSynchronizationBarrier` is called to wait until all other threads complete their phase 1 initialization. The main function can be written like so:

```
SYNCHRONIZATION_BARRIER sb;
InitializeSynchronizationBarrier(&sb, 4, -1);

LPTHREAD_START_ROUTINE functions[] = {
    InitSubSystem1, InitSubSystem2, InitSubSystem3, InitSubSystem4
};

printf("System initialization started\n");
HANDLE hThread[4];
int i = 0;
for (auto f : functions) {
    hThread[i++] = ::CreateThread(nullptr, 0, f, &sb, 0, nullptr);
}
::WaitForMultipleObjects(_countof(hThread), hThread, TRUE, INFINITE);
printf("System initialization complete\n");
// close thread handles...
```

Running this piece of code produces output like the following:

```
System initialization started
Subsystem 1: Starting phase 1 initialization (TID: 79480)...
Subsystem 2: Starting phase 1 initialization (TID: 104836)...
Subsystem 3: Starting phase 1 initialization (TID: 32556)...
Subsystem 4: Starting phase 1 initialization (TID: 86268)...
Subsystem 2: Ended phase 1 initialization...
Subsystem 3: Ended phase 1 initialization...
Subsystem 1: Ended phase 1 initialization...
Subsystem 4: Ended phase 1 initialization...
Subsystem 4: Starting phase 2 initialization...
Subsystem 3: Starting phase 2 initialization...
Subsystem 1: Starting phase 2 initialization...
Subsystem 2: Starting phase 2 initialization...
Subsystem 3: Ended phase 2 initialization...
Subsystem 1: Ended phase 2 initialization...
Subsystem 4: Ended phase 2 initialization...
Subsystem 2: Ended phase 2 initialization...
System initialization complete
```

Finally, a synchronization barrier should be deleted with `DeleteSynchronizationBarrier`:

```
BOOL DeleteSynchronizationBarrier(_Inout_ LPSYNCHRONIZATION_BARRIER lpBarrier);
```

It's OK to call `DeleteSynchronizationBarrier` immediately after calling `EnterSynchronizationBarrier` because the function waits until all threads reached the barrier before being deleted, unless all threads use the flag `SYNCHRONIZATION_BARRIER_FLAGS_NO_DELETE` so that the delete function does not make that guarantee. This could be useful if the barrier is never deleted.

What About the C++ Standard Library?

Similar to the same-named section from chapter 5, the C++ standard library provides synchronization primitives that can be used as alternatives to the Windows API, especially for cross-platform code. As usual, customization on these objects is very limited (if any). Examples include:

- `std::mutex` that acts like a critical section without the support for recursive acquisition.
- `std::recusrsive_mutex` that acts just like a critical section (supports recursive acquisition).
- `std::shared_mutex` that is similar to an SRW lock.

- `std::condition_variable` is a conditional variable equivalent.
- Others

Clearly, some things may be missing from C++, such as waiting on address and synchronization barriers. However, these could be added to a future standard. In any case, all the C++ standard library types work within the same process only. There is no way to use these across processes.

Exercises

1. Create a thread-safe implementation of a stack data structure. If a pop operation cannot succeed, block the thread until data is available (use a condition variable).

Summary

In this chapter, we looked at various synchronization mechanisms available through the Windows API. Common to all these is the ability to synchronize in some sense between threads in the same process. In the next chapter, we'll extend the synchronization primitives available by leveraging kernel objects that can also synchronize between threads in different processes.

Chapter 8: Thread Synchronization (Inter-Process)

The previous chapter described various synchronization mechanisms with one common factor: they can be used to synchronize between threads running in the same process. This chapter complements these mechanisms with others that are based on kernel objects, which by their very nature are part of system space, and so can naturally be shared between processes, and thus used by threads running in (potentially) different processes. This does not mean these mechanisms are useless in a same process scenario - far from it. But they do have that special capability which the mechanisms described in chapter 7 don't.

In this chapter:

- **Dispatcher Objects**
- **The Mutex**
- **The Semaphore**
- **The Event**
- **The Waitable Timer**
- **Other Wait Functions**

Dispatcher Objects

Chapter 2 deals extensively with kernel objects and handles. The following lists the most important points related to kernel objects. For more details, refer to chapter 2.

- Kernel objects reside in system (kernel) space and are theoretically accessible from any process, provided that process can obtain a handle to the requested object.
- Handles are process relative.
- There are three ways to share objects across processes: handle inheritance, name, and handle duplication.

Some kernel objects are more specialized, called *dispatcher objects* or *waitable objects*. Such objects can be in one of two states: *signaled* or *non-signaled*. The meaning of signaled and non-signaled depends on the type of object. Table 8-1 summarizes the meaning of these states for common dispatcher objects.

Table 8-1: Common dispatcher objects and their signaled/non-signaled meaning

Object type	Signaled	Non-Signaled
Process	Exited/Terminated	Running
Thread	Exited/Terminated	Running
Job	End of job time reached	Limit not reached or not set
Mutex	Free (unowned)	Owned
Semaphore	Count is above zero	Count is zero
Event	Event is set	Event is not set
File	I/O operations completed	I/O operation in progress or not started
Waitable Timer	Timer count has expired	Timer count has not expired
I/O Completion	Asynchronous I/O operation completed	I/O operation has not completed

> Files and I/O completion ports are discussed in chapter 11. Waitable timer, mutex, Semaphore and Event objects are discussed later in this chapter.

Waiting for an object to become signaled is usually accomplished by one of the following two functions (except for I/O completion port which has its own waiting function, discussed in chapter 11):

```
DWORD WaitForSingleObject(
    _In_ HANDLE hHandle,
    _In_ DWORD dwMilliseconds);

DWORD WaitForMultipleObjects(
    _In_ DWORD nCount,
    _In_ CONST HANDLE* lpHandles,
    _In_ BOOL bWaitAll,
    _In_ DWORD dwMilliseconds);
```

`WaitForSingleObject` accepts a handle to a dispatcher object, where the handle must have the `SYNCHRONIZE` access right. The timeout parameter indicates how long to wait at most for the object to become signaled. The value can be zero, which means no waiting should occur either way. Conversely, the value can be set to `INFINITE`, which means the thread is willing to wait for however long it takes until the object becomes signaled. There are possible four return values from `WaitForSingleObject`:

- `WAIT_OBJECT_0` - the wait is over because the object became signaled before the timeout expired.
- `WAIT_TIMEOUT` - the object did not become signaled in the time the thread was waiting. This value will never be returned if the timeout is `INFINITE`.
- `WAIT_FAILED` - the function failed for some reason. Call the usual `GetLastError` to see why.
- `WAIT_ABANDONED` - the wait is on a mutex object and the mutex has become abandoned. The meaning of an abandoned mutex will be discussed in the "Mutex" section later in this chapter.

The extended `WaitForMultipleObjects` function allows waiting on one or more handles. The function expects an array of handles as the second parameter, with the first parameter indicating the count of handles. This number of handles is limited to `MAXIMUM_WAIT_OBJECTS` (64). The third argument specifies whether the thread should wait until all objects become signaled at once (`TRUE`) or that just one (any) becomes signaled (`FALSE`).

The function's return values include `WAIT_TIMEOUT` and `WAIT_FAILED`, just like `WaitForSingleObject`, which mean the same thing. If `bWaitAll` is `TRUE` (wait for all objects), the return value is between `WAIT_OBJECT_0` and `WAIT_OBJECT_0+count-1`, where `count` is the number of handles. If the return value is between `WAIT_ABANDONED_0` and `WAIT_ABANDONED_0+count-1`, it means all object are signaled and at least one mutex is abandoned.

If `bWaitAll` is `FALSE`, the return value (if not timeout or error) indicates which object (index in the array) was in the signaled state, offset by `WAIT_OBJECT_0` or `WAIT_ABANDONED_0`. For example, returning `WAIT_OBJECT_0 + 3` means the fourth object was signaled and the wait was over because of that. If more than one object is signaled, the smallest index is the one returned.

Succeeding a Wait

If a wait function succeeds because the object or objects became signaled, the thread is awakened and can resume execution. Does the object just signaled remains in the signaled state? It depends on the type of the object. Some objects remain in their signaled state, such as processes and threads. Once a process exits or terminates, it becomes signaled and remains so for the rest of its life (while there are open handles to that process).

Some types of objects may change their signaled state after a successful wait. For example, a successful wait on a mutex turns it back to the non-signaled state (the reason will be evident in the next section where mutexes are discussed). Another object that exhibits special behavior when signaled is an auto-reset event. When signaled, it releases one thread (and one only), and when that happens its state flips to non-signaled automatically.

What happens if multiple threads wait for the same mutex, and it becomes signaled? Only one of the threads can acquire the mutex before it flips back to the non-signaled state. Behind the scenes waiting threads for an object are stored in a first-in-first-out (FIFO) queue, so the first thread in the queue is the one woken up (regardless of its priority). However, this behavior should not be relied upon. Some internal mechanisms may remove a thread from waiting (for example if it's suspended, such as with a debugger), and then when the thread resumes, it will be pushed to the back of the queue. So the simple rule here is that there is no way to know for sure which thread will wake up first. And in any case, this algorithm can change at any time in a future version of Windows.

The Mutex

The first kernel object type we'll examine is the mutex. The mutex (short for "mutual exclusion") provides similar functionality to the critical section discussed in chapter 7. Its purpose is the same: protect shared data from concurrent access. Only one thread at a time can acquire the mutex successfully, and proceed to access the shared data. All other threads waiting for the mutex must continue to wait until the mutex is released by the acquiring thread.

Creating a mutex object requires calling the `CreateMutex` or `CreateMutexEx` functions:

```
HANDLE CreateMutex(
    _In_opt_ LPSECURITY_ATTRIBUTES lpMutexAttributes,
    _In_     BOOL bInitialOwner,
    _In_opt_ LPCTSTR lpName);

HANDLE CreateMutexEx(
    _In_opt_ LPSECURITY_ATTRIBUTES lpMutexAttributes,
    _In_opt_ LPCTSTR lpName,
    _In_ DWORD dwFlags,
    _In_ DWORD dwDesiredAccess);
```

The first parameter to both functions is the usual `SECURITY_ATTRIBUTES` pointer, typically set to `NULL`. If `bInitialOwner` is set to `TRUE`, `CreateMutex` will attempt to acquire the mutex by calling `WaitForSingleObject` until it can be acquired, before returning. The same can be achieved

with `CreateMutexEx` by specifying the flag `CREATE_MUTEX_INITIAL_OWNER` in the `dwFlags` parameter. If a new object is created, then this acquisition succeeds immediately.

The `lpName` parameter allows setting a name for the mutex. If the mutex object with the same name exists (and no security restrictions apply), the functions open a handle to the existing mutex. If the name exists, but the object is not a mutex, the functions fail.
Finally, the extended function allows specifying the desired access mask for the mutex. This is mostly useful when opening an existing mutex, possibly requesting a weaker access mask than `MUTEX_ALL_ACCESS`, which is what is requested by default with `CreateMutex`.

An existing mutex can be opened by name with `OpenMutex`:

```
HANDLE OpenMutexW(
    _In_ DWORD dwDesiredAccess,
    _In_ BOOL bInheritHandle,
    _In_ LPCWSTR lpName);
```

If the named mutex doesn't exist, the function fails and returns `NULL`. If two (or more) threads want to synchronize using the same-named mutex, it's simpler (and avoids a race condition) to call `CreateMutex` or `CreateMutexEx`: the first thread (whoever that may be) creates the object, and subsequent callers get new handles to the existing object.

As usual with kernel objects, any open handles should be eventually closed with `CloseHandle`.

Calling `WaitForSingleObject` on a mutex causes the thread to wait until it becomes signaled, which means free, or unowned by any other thread. Once acquired, the mutex transitions to the non-signaled state atomically, preventing any other thread from acquiring it. Once the mutex' owner is done working with the shared data, it calls `ReleaseMutex` to release it from its ownership, making it signaled again:

```
BOOL ReleaseMutex(_In_ HANDLE hMutex);
```

The *Simple Increment* application from chapter 7 can be configured to use a mutex as a synchronization primitive (figure 8-1). Notice it takes considerably longer to perform the count correctly. This is because synchronizing using a mutex (like all kernel objects) requires a user mode to kernel mode transition, which is not free. The example is contrived, of course, so the difference seems huge compared to a critical section. In practice, it's not that bad.

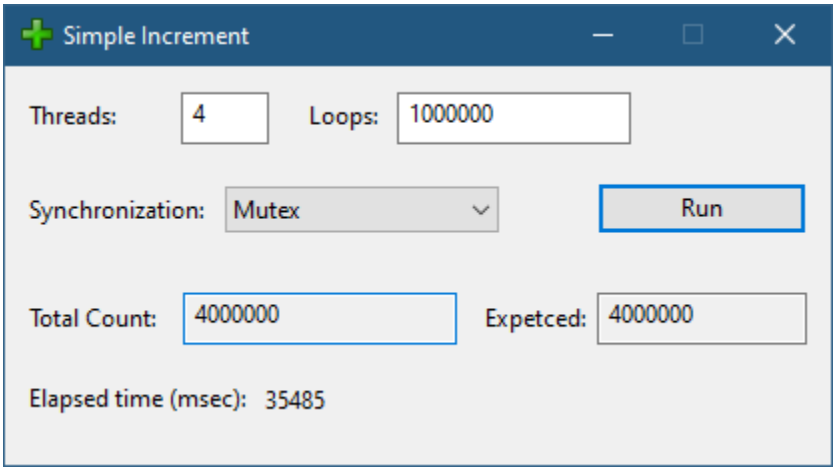


Figure 8-1: Simple increment with a mutex

The code that uses the mutex for this looped increment is the following:

```
void CMainDlg::DoMutexCount() {
    auto handles = std::make_unique<HANDLE[]>(m_Threads);
    m_hMutex = ::CreateMutex(nullptr, FALSE, nullptr);

    for (int i = 0; i < m_Threads; i++) {
        handles[i] = ::CreateThread(nullptr, 0, [](auto param) {
            return ((CMainDlg*)param)->IncMutexThread();
            }, this, 0, nullptr);
    }
    ::WaitForMultipleObjects(m_Threads, handles.get(), TRUE, INFINITE);
    for (int i = 0; i < m_Threads; i++)
        ::CloseHandle(handles[i]);
    ::CloseHandle(m_hMutex);
}

DWORD CMainDlg::IncMutexThread() {
    for (int i = 0; i < m_Loops; i++) {
        ::WaitForSingleObject(m_hMutex, INFINITE);
        m_Count++;
        ::ReleaseMutex(m_hMutex);
    }
    return 0;
}
```

Mutexes can be acquired recursively (by the same thread), causing an internal counter to be

incremented. This means the same number of calls to `ReleaseMutex` is needed to actually free the mutex. Calling `ReleaseMutex` by a thread that does not own the mutex, fails.

The Mutex Demo Application

The *MutexDemo* application shows how threads, running in different processes can synchronize access to a shared file, so that only one thread can access the file at the same time. Since multiple processes are involved, a critical section cannot be used.

To test things out, open two command windows and navigate to the directory where *MutexDemo.exe* resides. Alternatively, you can run from Visual Studio, but you would have to set a command-line argument in the project's properties (figure 8-2). The argument should be some non-existent file path.

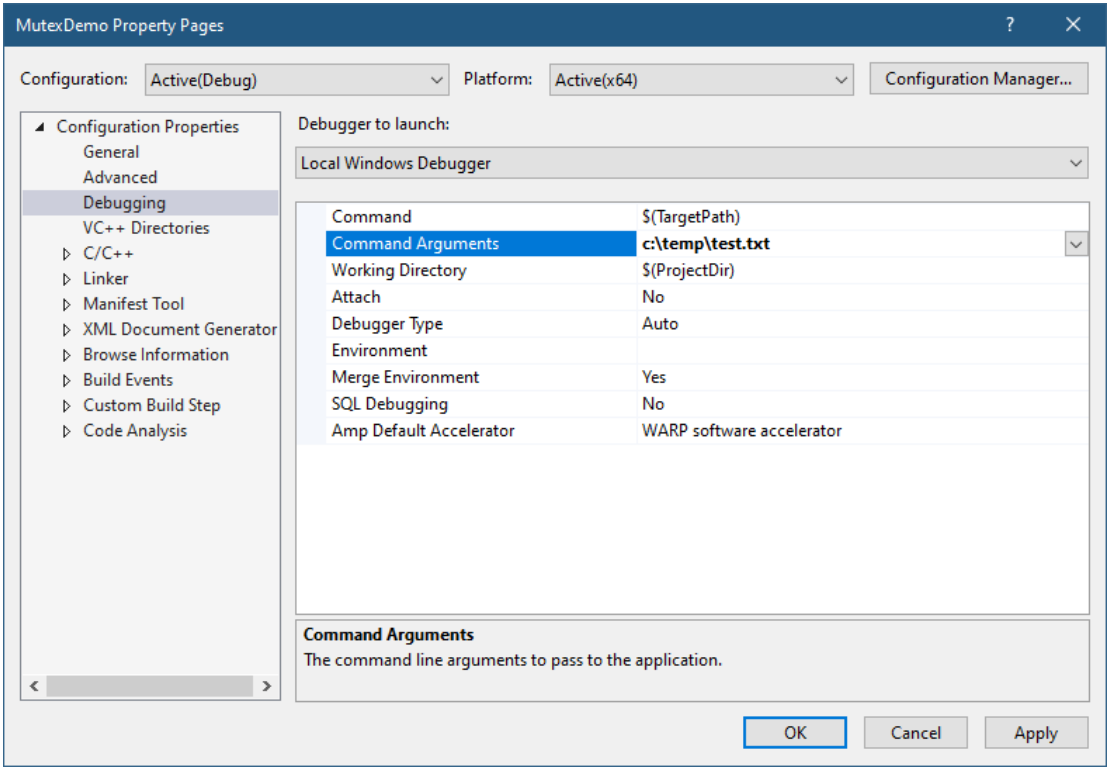


Figure 8-2: Setting a command line argument

Run the application from both command windows, pointing to the same file. You should see output like the following in each command window:

```
Process 25092. Mutex handle: 0x9C
Press any key to begin...
```

The process IDs will be different, and the most likely the handle values as well. These are handles to the same mutex object. To verify that, open *Process Explorer* and locate the two process instances. In each one locate the mutex (its name is "ExampleMutex"). Notice the handle values correspond to the values printed (figure 8-3).

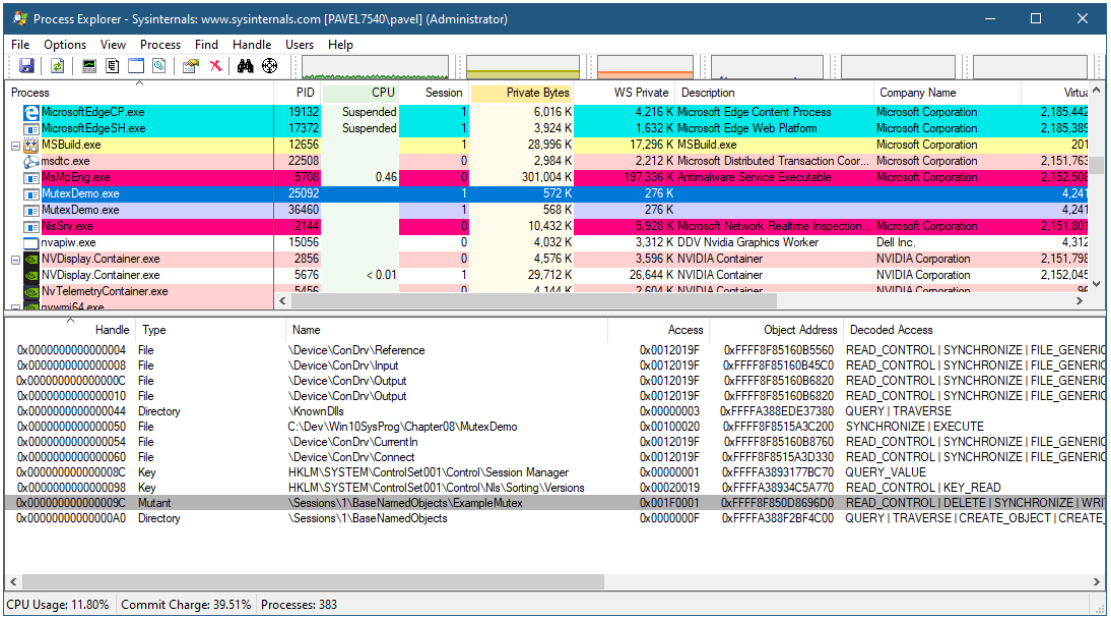


Figure 8-3: One of the mutex handles in *Process Explorer*

Now double-click the handle and verify the handle count for the mutex is 2 (figure 8-4). Also, note it's non-signaled (*Held: FALSE*).

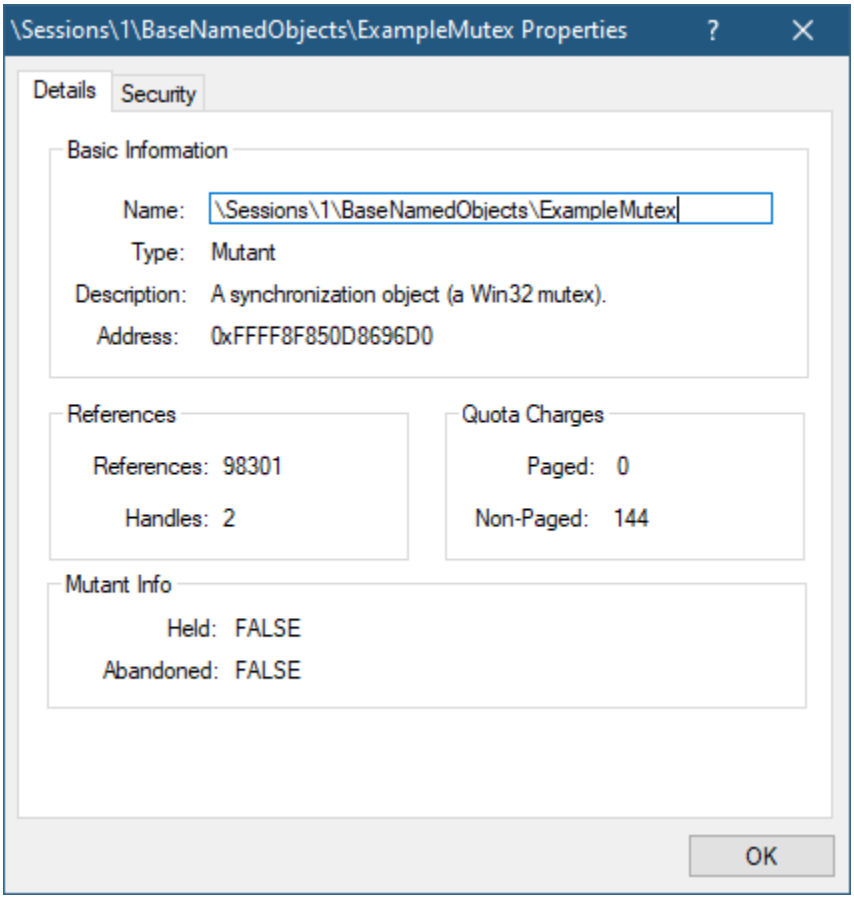


Figure 8-4: Mutex properties in *Process Explorer*

Now quickly hit any key in both console windows. Threads from these processes will now use the mutex to synchronize access to the file. Each thread appends a line to the file with a string holding the process ID.

Once the processes complete execution, you can open the file in a text editor. You should see something like the following:

```
This is text from process 25092
This is text from process 25092
This is text from process 25092
This is text from process 25092
This is text from process 25092
This is text from process 36460
This is text from process 25092
This is text from process 36460
This is text from process 25092
This is text from process 36460
This is text from process 36460
This is text from process 25092
This is text from process 36460
This is text from process 25092
This is text from process 36460
...
```

The total number of lines should be 200. Each process should have written exactly 100 lines with its own process ID.

The `main` function creates/opens the named mutex, prints the process ID and mutex handle, and then waits for a user to press a key:

```
int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 2) {
        printf("Usage: MutexDemo <file>\n");
        return 0;
    }

    HANDLE hMutex = ::CreateMutex(nullptr, FALSE, L"ExampleMutex");
    if (!hMutex)
        return Error("Failed to create/open mutex");

    printf("Process %d. Mutex handle: 0x%X\n", ::GetCurrentProcessId(), HandleT\
oULong(hMutex));
    printf("Press any key to begin...\n");
    _getch();
```

The `Error` function provides a simple error display we encountered several times before:

```
int Error(const char* text) {
    printf("%s (%d)\n", text, ::GetLastError());
    return 1;
}
```

Once a key is pressed, a loop is executed 100 times, acquires the mutex, accesses the file and, releases the mutex, all within the same iteration:

```
    printf("Working...\n");

    for (int i = 0; i < 100; i++) {
        // insert some randomness
        ::Sleep(::GetTickCount() & 0xff);

        // acquire the mutex
        ::WaitForSingleObject(hMutex, INFINITE);

        // write to the file
        if (!WriteToFile(argv[1]))
            return Error("Failed to write to file");

        ::ReleaseMutex(hMutex);
    }

    ::CloseHandle(hMutex);
    printf("Done.\n");

    return 0;
}
```

The `WriteToFile` function opens the file, sets the file pointer to the end of the file, writes the text to the file and closes the file:

```
bool WriteToFile(PCWSTR path) {
    HANDLE hFile = ::CreateFile(path, GENERIC_WRITE, FILE_SHARE_READ,
        nullptr, OPEN_ALWAYS, 0, nullptr);
    if (hFile == INVALID_HANDLE_VALUE)
        return false;

    ::SetFilePointer(hFile, 0, nullptr, FILE_END);
    char text[128];
    sprintf_s(text, "This is text from process %d\n", ::GetCurrentProcessId());
    DWORD bytes;
    BOOL ok = ::WriteFile(hFile, text, (DWORD)strlen(text), &bytes, nullptr);
    ::CloseHandle(hFile);

    return ok;
}
```

> The files API is discussed in detail in chapter 11.

You can run concurrently as many *QueueDemo* processes as you like - the shared file will not be corrupted. The fact that this demo uses the same executable is irrelevant - this works the same way if used from different executables. The important part is the shared mutex.

Change the mutex name to `NULL` and repeat the experiment. Do you understand the result?

Abandoned Mutex

What happens if a thread that owns a mutex exits or terminates (for whatever reason)? Since the owner of the mutex is the only one that can release the mutex, this may cause a deadlock, where other threads waiting for the mutex will never acquire it. This kind of mutex is called *abandoned mutex*, literally abandoned by its owner thread.

Fortunately, the kernel is aware of mutex ownership, and so releases an abandoned mutex explicitly if it sees that a thread terminated while holding a mutex (or more than one if that's the case). This causes the next thread which successfully acquires the mutex to get back `WAIT_ABANDONED` rather than `WAIT_OBJECT_0` from its `WaitForSingleObject` call. This means

the thread acquires the mutex normally, but the special return value is used as a hint to indicate the previous owner did not release the mutex before terminating. This usually indicates a bug that should be investigated further.

Write RAII wrappers for a mutex.

The Semaphore

The semaphore is the first synchronization kernel object that we examine that has no direct counterpart within the intra-process primitives examined in chapter 7. The purpose of a semaphore is to limit something, in a thread-safe way, of course.

A semaphore is initialized with a current and a maximum count. As long as its current count is above zero, it's in the signaled state. Whenever a thread calls `WaitForSingleObject` on a semaphore and it's in the signaled state, the semaphore's count is decremented and the thread is allowed to proceed. Once the semaphore count reaches zero, it becomes non-signaled, and any threads attempting to wait on it will block.

Conversely, a thread that wants to "release" one of the semaphore counts (or more), calls `ReleaseSemaphore`, causing the semaphore's count to increase and set it to the signaled state again.

Let's rewind a bit and look at how creating a semaphore with one of the following functions:

```
HANDLE CreateSemaphore(
    _In_opt_ LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    _In_ LONG lInitialCount,
    _In_ LONG lMaximumCount,
    _In_opt_ LPCTSTR lpName);

HANDLE CreateSemaphoreEx(
    _In_opt_ LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    _In_ LONG lInitialCount,
    _In_ LONG lMaximumCount,
    _In_opt_ LPCTSTR lpName,
    _Reserved_ DWORD dwFlags,
    _In_ DWORD dwDesiredAccess);
```

A semaphore can have a name, just like a mutex. This allows easy sharing between threads running in different processes. The unique parameters in the above creation functions are the

initial count and maximum count of the semaphore. These are typically set to the same value, indicating whatever is limited by the semaphore (such as a queue) has now zero elements. The extended function allows specifying the desired access mask (which is `SEMAPHORE_ALL_ACCESS` by default when calling `CreateSemaphore`). The `dwFlags` parameter is currently unused and must be set to zero.

Acquiring a count of a semaphore is done with the usual waiting functions. Releasing semaphore count(s) is accomplished with `ReleaseSemaphore`:

```
BOOL ReleaseSemaphore(
    _In_ HANDLE hSemaphore,
    _In_ LONG lReleaseCount,
    _Out_opt_ LPLONG lpPreviousCount);
```

The function allows specifying the count to release (i.e. how many to add to the current semaphore's count). This value is typically 1 but can be higher. The last argument allows retrieving the new semaphore count. It's also possible to specify zero for the release count and just get back the current semaphore's count. Of course, any such retrieval is a potential race condition, because between the retrieval and acting on the result, the semaphore's count might have changed by another thread.

Is a semaphore with a maximum count of one equivalent to a mutex? Think about that for a moment. The answer is a definite **no**. The reason is that a semaphore does not have any concept of ownership. Any thread can acquire one of its counts, and any thread can call `ReleaseSemaphore`. The semaphore's purpose is very different from a mutex'. This "free style" behavior can cause deadlocks if not careful, but in most cases, it's a good thing, as we'll see in the next code example.

As with other named objects, a handle to an existing semaphore can be obtained by name with `OpenSemaphore`:

```
HANDLE OpenSemaphore(
    _In_ DWORD dwDesiredAccess,
    _In_ BOOL bInheritHandle,
    _In_ LPCTSTR lpName);
```

The Queue Demo Application

The *Queue Demo* application shown in figure 8-5 is based on the application of the same name from chapter 7. This time, a semaphore is added so that the queue of work items is limited by a

specified amount. In the chapter 7 version, the queue could have grown indefinitely if producer threads generated data much faster than consumer threads could process. This is problematic because the memory consumption of the queue may be prohibitive and even cause out of memory situations in extreme cases (especially with 32-bit processes).

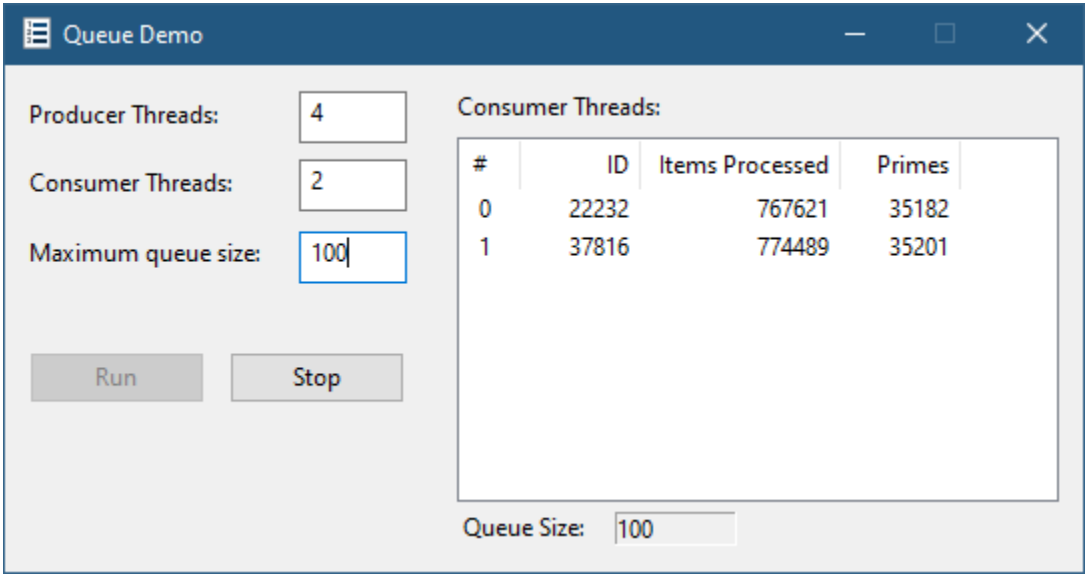


Figure 8-5: Enhanced *Queue Demo* application

This is where the semaphore comes in. In the original application, a producer thread that has new data would just push it into the queue and wake a consumer thread if needed with the condition variable. This time, a producer thread first calls `WaitForSingleObject` on the semaphore. If the semaphore's count is above zero, i.e. signaled, meaning the queue is not full and it goes ahead with pushing an item. On the consumer side, a consumer thread pops an item off the queue and then calls `ReleaseSemaphore` to indicate the queue now has one less item.

First, the semaphore is created in `CMainDlg::Run` based on the specified value in the dialog:

```
//...
int queueSize = GetDlgItemInt(IDC_MAX_QUEUE_SIZE);
if (queueSize < 10 || queueSize > 100000) {
    DisplayError(L"Maximum queue size must be between 10 and 100000");
    return;
}

// create semaphore

m_hQueueSem.reset(::CreateSemaphore(nullptr, queueSize, queueSize, nullptr));
```

`m_hQueueSem` is a new data member holding a smart handle to the semaphore (`wil::unique_handle`), ensuring `CloseHandle` is called when the semaphore goes out of scope or reset again.

Here is the modified producer code:

```
DWORD CMainDlg::ProducerThread() {
    for (;;) {
        if (::WaitForSingleObject(m_hAbortEvent.get(), 0) == WAIT_OBJECT_0)
            break;

        // wait if needed to make sure queue is not full
        ::WaitForSingleObject(m_hQueueSem.get(), INFINITE);

        WorkItem item;
        item.IsPrime = false;
        LARGE_INTEGER li;
        ::QueryPerformanceCounter(&li);
        item.Data = li.LowPart;
        {
            AutoCriticalSection locker(m_QueueLock);
            m_Queue.push(item);
        }
        ::WakeConditionVariable(&m_QueueCondVar);

        // sleep a little bit from time to time
        if ((item.Data & 0x7f) == 0)
            ::Sleep(1);
    }
    return 0;
}
```

The only addition is the call to `WaitForSingleObject` on the semaphore. The consumer threads code changes as well by releasing a single semaphore of the semaphore once an item has been removed from the queue:

```
DWORD CMainDlg::ConsumerThread(int index) {
    auto& data = m_ConsumerThreads[index];
    auto tick = ::GetTickCount64();

    for (;;) {
        WorkItem value;
        {
            bool abort = false;
            AutoCriticalSection locker(m_QueueLock);
            while (m_Queue.empty()) {
                if (::WaitForSingleObject(m_hAbortEvent.get(), 0) == WAIT_OBJEC\
T_0) {
                    abort = true;
                    break;
                }
                ::SleepConditionVariableCS(&m_QueueCondVar, &m_QueueLock, INFIN\
ITE);
            }
            if (abort)
                break;

            ATLASSERT(!m_Queue.empty());
            value = m_Queue.front();
            m_Queue.pop();

            ::ReleaseSemaphore(m_hQueueSem.get(), 1, nullptr);
        }

        // rest of code omitted...

    return 0;
}
```

You can look at a semaphore basic properties using *Process Explorer*'s handle view. Figure 8-6 shows the semaphore from the *Queue Demo* application while it's running (*Process Explorer* does not update the object's state automatically, though).

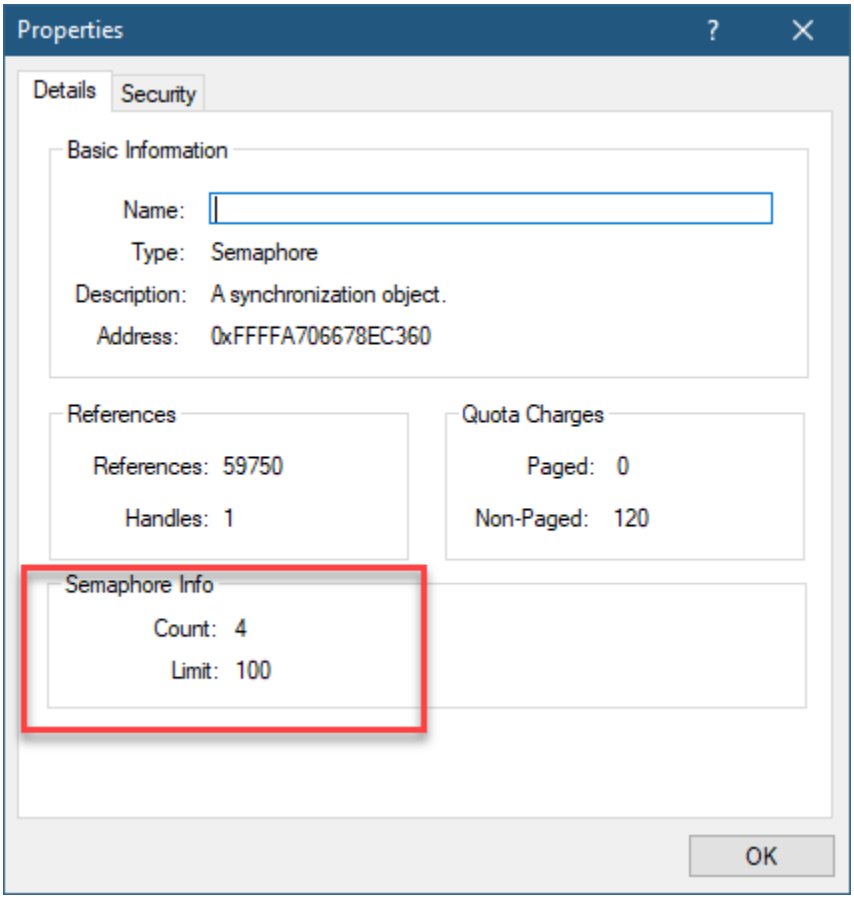


Figure 8-6: Semaphore's properties in *Process Explorer*

The flexibility of the semaphore is evident: producer threads wait to get one count from the semaphore, and consumer threads release count(s) - not the same threads.

Write RAII wrappers for a semaphore.

The Event

The event is in one sense the simplest of the synchronization primitives - it's just a flag that can be set (signaled state) or reset (non-signaled). Being a (possibly named) kernel object gives it the flexibility to work within a single process or across processes. We've already used an event in the *Queue Demo* application. Now we'll discuss it in detail.

A complexity associated with events is the fact there are two types of events: manual-reset and auto-reset. Table 8-2 summarizes their properties, which will be elaborated on next.

Table 8-2: Event type differences

Event type	Kernel name	Effect of `SetEvent`
Manual reset	Notification	puts the event in the signaled state, and releases all threads waiting on it (if any). The event remains in the signaled state
Auto reset	Synchronization	a single thread is released from wait, and then the event goes back automatically to the non-signaled state

> The kernel type name in table 8-2 is useful with tools such as *Process Explorer* which provide an event's type name using kernel terminology.

Creating an event object is no different from other object types, accomplished with one of the following functions:

```
HANDLE CreateEvent(
    _In_opt_ LPSECURITY_ATTRIBUTES lpEventAttributes,
    _In_ BOOL bManualReset,
    _In_ BOOL bInitialState,
    _In_opt_ LPCTSTR lpName);

HANDLE CreateEventEx(
    _In_opt_ LPSECURITY_ATTRIBUTES lpEventAttributes,
    _In_opt_ LPCTSTR lpName,
    _In_ DWORD dwFlags,
    _In_ DWORD dwDesiredAccess);
```

The functions specify the type of event required to create. Once the decision is made, it cannot be changed. `CreateEvent` uses the `bManualReset` parameter to indicate a manual-reset event (`TRUE`) or an auto-reset event (`FALSE`). With `CreateEventEx`, the type of event is specified with the `dwFlags` parameter, where setting it to `CREATE_EVENT_MANUAL_RESET` signifies a manual-reset event.

The second decision to be made is the initial state of the event (signaled or non-signaled). `CreateEvent` allows specifying the initial state in the `bInitialState` parameter. With `CreateEventEx`, another flag is used to indicate an initial state of signaled - `CREATE_EVENT_INITIAL_SET`.

As with the extended create functions for mutex and semaphore, `CreateEventEx` allows specifying the access mask for the new event handle (`EVENT_ALL_ACCESS` by default with `CreateEvent`). Similarly to the former objects, if a name is specified and an event with that name already exists, (and barring security constraints), another handle to the event is opened, and the event type and initial state are ignored.

Remember, the way to differentiate a new from an existing object is by calling `GetLastError` and checking for `ERROR_ALREADY_EXISTS`.

Similarly to the other objects, an event can be opened by name with `OpenEvent`:

```
HANDLE OpenEvent(
    _In_ DWORD dwDesiredAccess,
    _In_ BOOL bInheritHandle,
    _In_ LPCTSTR lpName);
```

Working with Events

An event's state can be explicitly changed with the `SetEvent` and `ResetEvent` functions:

```
BOOL SetEvent(_In_ HANDLE hEvent);      // signaled
BOOL ResetEvent(_In_ HANDLE hEvent);    // non-signaled
```

Let's look at an example scenario. Suppose there are several processes running that are part of the same system. Suppose further that one of the processes, let's call it the *controller*, needs to signal all other processes part of this application to shutdown gracefully. How can that be accomplished?

Events provide the perfect candidate. Processes are isolated and so no direct communication is possible. In most cases, that isolation is a good thing. In some cases, some information must be sent between processes. In this case the information to send is simple: a single bit would suffice to indicate a shutdown operation is required.

This flow synchronization can be accomplished with a manual-reset event. All processes create a named event with a predefined name such as "ShudownEvent":

```
// notice a manual-reset event
HANDLE hShutdown = ::CreateEvent(nullptr, TRUE, FALSE, L"ShutdownEvent");
```

Since the object is named, only the first process actually creates it and the rest get handles to the existing object. Calling `CreateEvent` is convenient and does not require any synchronization for "who creates the event first" - it simply doesn't matter.

Next, each process except the *controller* need to wait on the event somewhere. When the event becomes signaled, each process initiates its own shutdown procedure:

```
::WaitForSingleObject(hShutdown, INFINITE);
// object is signaled, initiate shutdown...
```

The *controller* process just needs to set the event when shutdown is required:

```
::SetEvent(hShutdown);
// initiate own shutdown...
```

A manual reset event is required here because setting the event should wake all threads waiting on it, which is what is required in this scenario.

A question that may come up is where exactly the participating processes wait for the event? The simplest solution is to create a thread whose sole purpose is to do the waiting. This is not ideal, as threads should not be created to just wait. There is a better alternative which is using the thread pool, which we'll examine in the next chapter.

An auto-reset event's behavior is different. Calling `SetEvent` changes it to the signaled state. If no threads are waiting on it, it remains in the signaled state, until at least one thread waits on it. Then, a single thread is released and the event goes back to the non-signaled state automatically. To wake another waiting thread, another call to `SetEvent` is required.

The *Queue Demo* application shows a common example of using an event to indicate to producer and consumer threads it's time to abort. The event is first created in `CMainDlg::OnInitDialog` as a manual-reset event, since multiple threads need to be notified with a single call that it's time to exit:

```
m_hAbortEvent.reset(::CreateEvent(nullptr, TRUE, FALSE, nullptr));
```

The producer code checks if the event is signaled without waiting:

```
DWORD CMainDlg::ProducerThread() {
    for (;;) {
        if (::WaitForSingleObject(m_hAbortEvent.get(), 0) == WAIT_OBJECT_0)
            break;
//...
```

If the event is signaled it aborts immediately by breaking out of the infinite loop. This call is made at every iteration so the producer can exit as soon as possible. Consumer threads work in a similar manner.

> You may be wondering why not just use a Boolean variable, check if it's true and exit if it becomes true. The main issue is that the compiler (and possibly the CPU as well) will optimize away the variable as having a value that does not change, since the compiler has no idea it may change from a different thread. A possible solution is to mark the variable `volatile` to prevent any optimizations and force the CPU to access the actual value. Even that is not bulletproof. Generally, these options are best avoided as this situation is a data race - multiple threads accessing the same memory where at least one is writing. In any case, there is no way to simulate the behavior of an auto-reset event with simple variables, let alone coordinate between threads in different processes.

The final function for working with events is `PulseEvent`:

```
BOOL PulseEvent(_In_ HANDLE hEvent);
```

The purpose of `PulseEvent` is to set the event momentarily and if no thread is currently waiting, reset the event. The documentation states: "This function is unreliable and should not be used. It exists mainly for backward compatibility.", and goes on to give an example of why using this function is not a good idea.

Avoid using `PulseEvent`.

The Waitable Timer

The Windows API provides access to several timers with different semantics and programming models. Here are the main ones:

- For windowing scenarios, the `SetTimer` API provides a timer that works by posting `WM_TIMER` messages to the calling thread's message queue. This timer is suitable for GUI applications, since the timer message can be handled on the UI thread.
- The Windows multimedia API provides a multimedia timer created with `timeSetEvent` that calls a callback function on a separate thread at priority 15. The timer can be one shot or periodic and can be very precise (its resolution can be set by the function). A value of zero for the resolution requests the highest resolution the system can provide. Here is a simple example using a multimedia timer:

```
#include <mmsystem.h>

#pragma comment(lib, "winmm")

void main() {
    auto id = ::timeSetEvent(
        1000,           // interval (msec)
        10,             // resolution (msec)
        OnTimer,        // callback
        0,              // user data
        TIME_PERIODIC); // periodic or one shot
    ::Sleep(10000);
    ::timeKillEvent(id);
}

void CALLBACK OnTimer(UINT id, UINT, DWORD_PTR userData, DWORD_PTR, DWORD_PTR) {
    printf("Timer struck at %u\n", ::GetTickCount());
}
```

The timer we'll focus on in this section, is a *waitable timer*, which is a kernel object and so deserves to be in this chapter. A waitable timer becomes signaled when its due time arrives.

Creating a waitbale timer is accomplished by one of two functions, somewhat similar to previously encountered functions:

```
HANDLE CreateWaitableTimer(
    _In_opt_ LPSECURITY_ATTRIBUTES lpTimerAttributes,
    _In_ BOOL bManualReset,
    _In_opt_ LPCTSTR lpTimerName);

HANDLE CreateWaitableTimerEx(
    _In_opt_ LPSECURITY_ATTRIBUTES lpTimerAttributes,
    _In_opt_ LPCTSTR lpTimerName,
    _In_ DWORD dwFlags,
    _In_ DWORD dwDesiredAccess);
```

A waitable timer can have a name, just like mutexes, semaphores and events. There are two variants of waitable timers, similar to events: manual-reset timers (`bManualReset` is `TRUE` or `dwFlags` is `CREATE_WAITABLE_TIMER_MANUAL_RESET`) or auto-reset, also called *synchronization* timer (`bManualReset` is `FALSE` or `dwFlags` is zero). Finally, `CreateWaitableTimerEx` can specify an explicit access mask for the returned handle (`TIMER_ALL_ACCESS` by default with `CreateWaitableTimer`).

As waitable timers can be named, an existing timer can be opened by name with `OpenWaitableTimer`:

```
HANDLE OpenWaitableTimer(
    _In_ DWORD dwDesiredAccess,
    _In_ BOOL bInheritHandle,
    _In_ LPCTSTR lpTimerName);
```

Creating a timer is the first step of using it, with the all-important `SetWaitableTimer(Ex)` functions:

```
typedef VOID (CALLBACK *PTIMERAPCROUTINE)(
    _In_opt_ LPVOID lpArgToCompletionRoutine,
    _In_     DWORD dwTimerLowValue,
    _In_     DWORD dwTimerHighValue);

BOOL SetWaitableTimer(
    _In_ HANDLE hTimer,
    _In_ const LARGE_INTEGER* lpDueTime,
    _In_ LONG lPeriod,
    _In_opt_ PTIMERAPCROUTINE pfnCompletionRoutine,
    _In_opt_ LPVOID lpArgToCompletionRoutine,
    _In_ BOOL fResume);
```

```
BOOL SetWaitableTimerEx(
    _In_ HANDLE hTimer,
    _In_ const LARGE_INTEGER* lpDueTime,
    _In_ LONG lPeriod,
    _In_opt_ PTIMERAPCROUTINE pfnCompletionRoutine,
    _In_opt_ LPVOID lpArgToCompletionRoutine,
    _In_opt_ PREASON_CONTEXT WakeContext,
    _In_ ULONG TolerableDelay);
```

The first five parameters to both functions are the same, so we'll tackle those first. The `lpDueTime` parameter signifies when the time should expire. It's given as a `LARGE_INTEGER` structure, which is nothing but a glorified 64-bit number, which convenient access to its two 32-bit parts if desired. The number stored is signed, with different meanings to positive vs. negative numbers:

- A positive number indicates absolute time in 100-nano second units measured from January 1, 1601, midnight, UTC (*Universal Coordinated Time* also known as GMT).
- A negative number indicates relative time in 100-nano second units.

The actual resolution of timer depends on the hardware and is not in the range of 100 nano seconds.

Let's start with the most common case, a relative interval. An interval in milliseconds is fairly common, and converting between 100nsec (10 to -7th power) to msec (10 to the -3rd power), means multiplying by 10000. An interval of 10 msec can be set by initializing a `LARGE_INTEGER` like so:

```
LARGE_INTEGER interval;
interval.QuadPart = -10000 * 10;
```

Absolute time is more tricky, as zero time is in a simingly weird time in the distant past. The best thing is to work with helper functions provided by the Windows API to get to the desired value. For example, suppose the timer should expire at 17:30:00 on March 10, 2020 (UTC) (the date I'm writing these lines), the following snippet helps in computing the correct value:

```
SYSTEMTIME st = { 0 };
st.wYear = 2020;
st.wMonth = 3;
st.wDay = 10;
st.wHour = 17;
st.wMinute = 30;

FILETIME ft;
::SystemTimeToFileTime(&st, &ft);
LARGE_INTEGER dueTime;
dueTime.QuadPart = *(LONGLONG*)&ft;
```

The `FILETIME` structure is identical to a `LARGE_INTEGER`, but curiously enough does not have the single 64-bit data member, only two 32-bit values; this is why the last line forcefully reads it as a 64-bit value. Also, the term "file time" commonly used in file system times uses the same time measurement.
If local time is needed as a basis, rather than UTC, the `TzSpecificLocalTimeToSystemTime` before calling `SystemTimeToFileTime` like so:

```
FILETIME ft;
::TzSpecificLocalTimeToSystemTime(nullptr, &st, &st);
::SystemTimeToFileTime(&st, &ft);
```

`TzSpecificLocalTimeToSystemTime` uses the current time zone by default (the first `NULL` value), taking into account Daylight Saving Time (DST), if active.

If the due time is absolute and specifies a value in the past, the timer is signaled immediately.

The third parameter to `SetWaitableTimer(Ex)` indicates whether the timer is one-shot or periodic. Specifying zero makes it one-shot; otherwise, it's the period in milliseconds. Note this works in both absolute and relative due times.

The fourth parameter is an optional function pointer that should be called when the timer is signaled (expires). The parameter can be `NULL`, in which case the normal wait functions can be used to get an indication of when the timer expires. If the value is non-`NULL`, the function is not called immediately when the timer expires; instead, the function is wrapped in an *Asynchronous Procedure Call* (APC) and attached to the thread that called `SetWaitableTimer(Ex)`.

An APC is a callback destined to a particular thread, and so must be executed by that thread only. In the case of a waitable timer, the APC is added to the queue of APCs for the thread that called

`SetWaitableTimer(Ex)`. The tricky part is that this APC does not execute immediately. That would be too dangerous, as the thread may be doing something at the time the timer expires, forcefully diverging it to the APC's callback can have unintended consequences. What if the thread has acquired a critical section at the time of the call? In general, the APC would have no clue where the thread was in its execution.

This all means that APCs are pushed at the end of the particular thread's queue, but in order to run them the thread must enter an *alertable state*. In this state, the thread first checks if any APCs have accumulated in its APC queue, and if so, runs all of them now in sequence before resuming execution of the code following its entering the alertable state.

How does a thread enter an alertable state? There are several functions that can accomplish that, the simplest being `SleepEx`:

```
DWORD SleepEx(
  _In_ DWORD dwMilliseconds,
  _In_ BOOL  bAlertable);
```

`SleepEx` is a superset of the familiar `Sleep` function. In fact, the `Sleep` function is implemented by calling `SleepEx` with the `bAlertable` argument set to `FALSE`. Calling `SleepEx` with `bAletrable` set to `TRUE` puts the thread to sleep in an alertable state for the duration of the sleep. If, during the sleep period, APCs appear in the thread's queue, they are executed immediately and the sleep is over. If any APC was already present when the call to `SleepEx` is made, no sleep occurs at all.

The following example shows how to set up a waitable timer that calls a callback every second by putting the thread in an infinite sleep in an alertable state (*SimpleTimer* project in the code samples for this chapter):

```
void CALLBACK OnTimer(void* param, DWORD low, DWORD high) {
    printf("TID: %u Ticks: %u\n", ::GetCurrentThreadId(), ::GetTickCount());
}

int main() {
    auto hTimer = ::CreateWaitableTimer(nullptr, TRUE, nullptr);
    LARGE_INTEGER interval;
    interval.QuadPart = -10000 * 1000LL;
    ::SetWaitableTimer(hTimer, &interval, 1000, OnTimer, nullptr, FALSE);
    printf("Main thread ID: %u\n", ::GetCurrentThreadId());

    while (true)
        ::SleepEx(INFINITE, TRUE);
```

```
    // we'll never get here
    return 0;
}
```

Running this yields something like the following:

```
Main thread ID: 32024
TID: 32024 Ticks: 19648406
TID: 32024 Ticks: 19649406
TID: 32024 Ticks: 19650421
TID: 32024 Ticks: 19651421
TID: 32024 Ticks: 19652437
TID: 32024 Ticks: 19653437
TID: 32024 Ticks: 19654453
...
```

Notice the thread that called `SetWaitableTimer` is the same one that executes the callback. The code uses `SleepEx` with an infinite timeout, as the thread has nothing to do except for running the timer callback. The infinite `while` loop is necessary, otherwise after the first callback runs, the sleep is over and the program would exit. The loop keeps the thread alive, just waiting for APCs to appear.

Another useful option with `SleepEx` is to use a timeout of zero. This can be thought of a simple "garbage collection", where a thread from time to time calls `SleepEx(0, TRUE)` to run any APCs that may have accumulated, but the thread does not wish to wait at all.

`SleepEx` is simple enough, but other scenarios require more flexibility. Other functions that allow a thread to wait in an alertable state include extended versions of the classic functions, discussed in the next section.

Other forms of using APCs are covered in chapter 11.

Windows supports three types of APCs: user mode APCs, kernel-mode APCs and special kernel-mode APCs. The former is the one discussed in this book. The kernel variants are (obviously) available for kernel-mode callers only (and are in fact *not* documented in the WDK), and in any case are not in the scope of this book. See my book "Windows Kernel Programming", and various online resources for a detailed description of kernel-mode APCs.

The fifth parameter to `SetWaitableTimer(Ex)` is a user-defined value passed as-is to the callback function if provided. The callback itself receives this value as its first argument, and also two 32-bit values that comprise a 64-bit value in the absolute format described earlier indicating the time the timer has been triggered.

The sixth (and last) parameter to `SetWaitableTimer` specifies if timer expiration should trigger waking the system if it was in a power conservation state (such as *connected standby*).

This concludes `SetWaitableTimer`. The extended function (available from Windows 7 and Server 2008 R2) has two more parameters. The first (sixth) is an optional pointer to a `REASON_CONTEXT` structure defined like so:

```
typedef struct _REASON_CONTEXT {
    ULONG Version;
    DWORD Flags;
    union {
        struct {
            HMODULE LocalizedReasonModule;
            ULONG LocalizedReasonId;
            ULONG ReasonStringCount;
            LPWSTR *ReasonStrings;
        } Detailed;
        LPWSTR SimpleReasonString;
    } Reason;
} REASON_CONTEXT, *PREASON_CONTEXT;
```

The structure can provide additional context for the timer request. It's also used with power requests. This helps in logging in case the timer causes the system to wake from a low power state. Passing `NULL` indicates there is no specific context. See the documentation for `REASON_CONTEXT` for the exact details.

The last parameter to `SetWaitableTimerEx` is a tolerance value in milliseconds for the timer's expiration. This relates to a feature called *coalescing timers* introduced in Windows 7. Suppose you have two timers, one that expires in 100 msec and the other in 105 msec. Normally, a CPU would have to wake up after 100 msec and signal the first timer, go to sleep, and then wake after 5 msec to signal the second timer. If, however, the second (or first) timer has requested a tolerance of (say) 10 msec, the system would wake the CPU just once and signal both timers in one stroke, because the application indicated it's OK to signal the timer at some tolerance interval from the exact time. Specifying zero (which is what `SetWaitableTimer` does internally) means there is no tolerance, and the application wants the best accuracy at the possible expanse of more power consumption. Otherwise, the tolerance is taken into consideration with respect to other timers on the system.

We are not done with timers yet - timers can also be handled more conveniently by utilizing the thread pool, which we'll look at in the next chapter.

Lastly, a timer can be canceled after a successful call to `SetWaitableTimer(Ex)` (but before the timer expires) with `CancelWaitableTimer`:

```
BOOL CancelWaitableTimer(_In_ HANDLE hTimer);
```

Other Wait Functions

The classic wait functions `WaitForSingleObject` and `WaitForMultipleObjects` are the most common to use. However, there are other variants, which we look it in this section.

Waiting in Alertable State

Extended versions of the common functions exist that accept an extra parameter to indicate whether to wait in an alertable state:

```
DWORD WaitForSingleObjectEx(
    _In_ HANDLE hHandle,
    _In_ DWORD dwMilliseconds,
    _In_ BOOL bAlertable);

DWORD WaitForMultipleObjectsEx(
    _In_ DWORD nCount,
    _In_reads_(nCount) CONST HANDLE* lpHandles,
    _In_ BOOL bWaitAll,
    _In_ DWORD dwMilliseconds,
    _In_ BOOL bAlertable);
```

Passing `FALSE` for `bAlertable` is the same as calling the original function. With `bAlertable` set to `TRUE`, any APCs attached to the calling thread are executed in sequence and then the wait is over. The return value in such a case is `WAIT_IO_COMPLETION`. If that's returned and the thread still wants to wait on the object(s), it can call the wait function again.

Other functions exist with the same pattern, providing an option for waiting in an alertable state, described in the next sections.

Waiting on GUI Threads

GUI threads should not generally use the `WaitForSingleObject` or `WaitForMultipleObjects` (or their extended variants) with an `INFINITE` timeout if the wait could be long. The problem is that if the object(s) in question do not become signaled for a long time, all UI activity managed by that thread will freeze, producing the dreaded "Not Responding" status in *Task Manager*, with the windows created by that thread becoming faded and unresponsive, and "Not Responding" added to their title bar. This is a very bad user experience and should be avoided at all costs.

In many cases, GUI threads don't need to wait for kernel objects, but in some cases it's unavoidable. Fortunately, there is a solution: the `MsgWaitForMultipleObject(Ex)` functions:

```
DWORD
WINAPI
MsgWaitForMultipleObjects(
    _In_ DWORD nCount,
    _In_ CONST HANDLE *pHandles,
    _In_ BOOL fWaitAll,
    _In_ DWORD dwMilliseconds,
    _In_ DWORD dwWakeMask);

WINUSERAPI
DWORD
WINAPI
MsgWaitForMultipleObjectsEx(
    _In_ DWORD nCount,
    _In_ CONST HANDLE *pHandles,
    _In_ DWORD dwMilliseconds,
    _In_ DWORD dwWakeMask,
    _In_ DWORD dwFlags);
```

The functions wait on one or more object normally, but also for UI messages destined for the calling thread, whose type is specified by the `dwWakeMask`parameter. The simplest is `QS_ALLEVENTS` that causes the functions to return with the value `WAIT_OBJECT_0+nCount` whenever any message appears in the thread's message queue. In this case, the thread should pump messages and resume waiting. Here is an example for waiting on an event object and pumping messages in between:

```
void WaitWithMessages(HANDLE hEvent) {
    while (::MsgWaitForMultipleObjects(1, &hEvent, FALSE, INFINITE, QS_ALLEVENT\
S)
        == WAIT_OBJECT_0 + 1) {
        MSG msg;
        while (::PeekMessage(&msg, nullptr, 0, 0, PM_REMOVE)) {
            ::TranslateMessage(&msg);
            ::DispatchMessage(&msg);
        }
    }
}
```

> Check out the other state mask (`QS_`) values in the documentation.

The extended function drops the `fWaitAll` parameter from the original function, and instead provides an extra `dwFlags` parameter that can be zero or include one or more the following flags:

- `MWMO_ALERTABLE` - the function waits in an alertable state, as described in the previous section.
- `MWMO_INPUTAVAILABLE` - the function returns if input exists in the message queue, even if that input has already been seen with `PeekMessage` or `GetMessage`. Without this flag, only new input causes the function to return.
- `MWMO_WAITALL` - the function returns if all the objects are signaled and input is available, all at the same time.

Waiting for an Idle GUI Thread

The `WaitForInputIdle` function can be used to wait until a GUI thread in the specified process is ready to process messages:

```
DWORD WINAPI WaitForInputIdle(
  _In_ HANDLE hProcess,
  _In_ DWORD  dwMilliseconds);
```

The function is most useful for a parent that creates a child process and wants to interact with its UI thread (typically the first thread). Since process creation on the child's process side is asynchronous, the parent has no way of knowing when exactly the thread is ready to receive messages. Posting messages to the thread too early (before the thread's message queue is ready) will cause the messages to be lost.

Here is a typical code that accomplishes this:

```
PROCESS_INFORMATION pi;
//...
::CreateProcess(..., &pi);
// error handling omitted
::WaitForInputIdle(pi.hProcess, INFINITE);
// UI thread is ready, post some message to the main thread
::PostThreadMessage(pi.dwThreadId, WM_USER, 0, 0);
//...
```

Signaling and Waiting Atomically

The last waiting function we'll look at in this chapter is `SignalObjectAndWait`:

```
DWORD SignalObjectAndWait(
    _In_ HANDLE hObjectToSignal,
    _In_ HANDLE hObjectToWaitOn,
    _In_ DWORD dwMilliseconds,
    _In_ BOOL bAlertable);
```

The `hObjectToSignal` can be an event, semaphore or mutex only, each signaled with its own function: `SetEvent`, `ReleaseSemaphore` (with a count of 1), and `ReleaseMutex`, respectively. `hObjectToWaitOn` can point to any waitable object. The function combines signaling one object and atomically waiting on another one.

The first benefit of this function is its efficiency. Instead of something like the following:

```
::SetEvent(hEvenet1);
::WaitForSingleObject(hEvent2, INFINITE);
```

`SignalObjectAndWait` combines these two functions so that only a single transition to kernel mode is required:

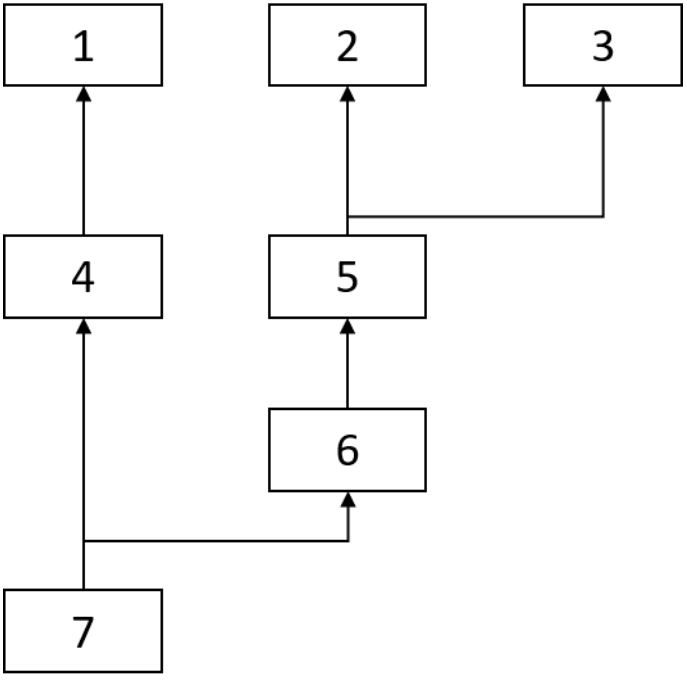
```
::SignalObjectAndWait(hEvent1, hEvent2, INFINITE, FALSE);
```

The second benefit is the fact that the two operations are atomic, meaning no other thread can observe the signaled state of the signaled object before the thread enters a wait state on the other object. In some edge cases involving the `PulseEvent` function, `SignalObjectAndWait` provides a reliable solution.

The former warning applies - don't use `PulseEvent`.

Exercises

1. Create a system that can run multiple work items concurrently, but some of them may have dependencies on other work items. As a concrete example, think of compiling projects in Visual Studio. Some projects depend on other projects, so they must be processed in order. Here is an example of a project hierarchy (read: project 4 depends on 1, project 5 depends on projects 2 and 3, and so on). The goal is to compile all projects as quickly as possible while adhering to the dependencies. use event objects for flow synchronization.



Example project dependencies

Summary

In this chapter, we looked at the dispatcher objects commonly used for thread synchronization. In the next chapter, we'll examine thread pools, a common alternative to explicitly creating threads.

Chapter 9: Thread Pools

In the last few chapters, we saw how to create and manage threads. Although this works, there are cases where it's overkill. Sometimes we need to perform some bounded operation on a different thread, but always creating a new thread has its overhead. Threads are not free: the kernel has its structures that manage the information of a thread, a thread has user-mode and kernel-mode stacks, and the creation of a thread itself takes time. If the thread is expected to be relatively short-lived, the extra overhead becomes significant.

The thread pools discussed in this chapter have no relationship to the .NET or .NET Core thread pool, which is available in managed processes. The CLR / CoreCLR has its own implementation of a thread pool.

In this chapter:

Why Use a Thread Pool?

Windows provides thread pools, a mechanism that allows sending operations to be performed by some thread from a pool of threads. The advantages of using a thread pool over manually creating and managing threads are the following:

- No explicit thread creation or termination is done by client code - the thread pool manager handles that.
- Completed operations do not destroy the worker thread - it returns to the pool to service another request.
- The number of threads in the pool can grow and shrink dynamically based on work items load.

Windows 2000 was the first Windows version to provide support for thread pooling. It offered a single thread pool per process. Starting from Windows Vista, the thread pool API was significantly enhanced, including the addition of private thread pools, which means more than one thread pool can exist in a process.

> We'll describe the newer API only, as there are no good reasons to use the old API unless you're targetting pre-Vista versions of Windows.

The services of thread pooling are used internally by some Windows functions and third-party libraries, so you might find that thread pool threads exist even if you did not explicitly use any thread pool-related APIs. If you run a simple application such as *Notepad*, which does not require more than one thread, you may find, when looking at the process in *Process Explorer*, that it has several threads, some of them starting with the function `ntdll!TppWorkerThread` (figure 9-1). This is the starting function for a thread-pool thread. If you allow the *Notepad* process to linger for a while, you might find that after some time of inactivity, the thread pool threads are gone (figure 9-2).

> "Tpp" is short for *Thread Pool Private*, i.e. a private (not exported) function related to thread pooling. The kernel object responsible for managing thread pools is called *TpWorkerFactory*.

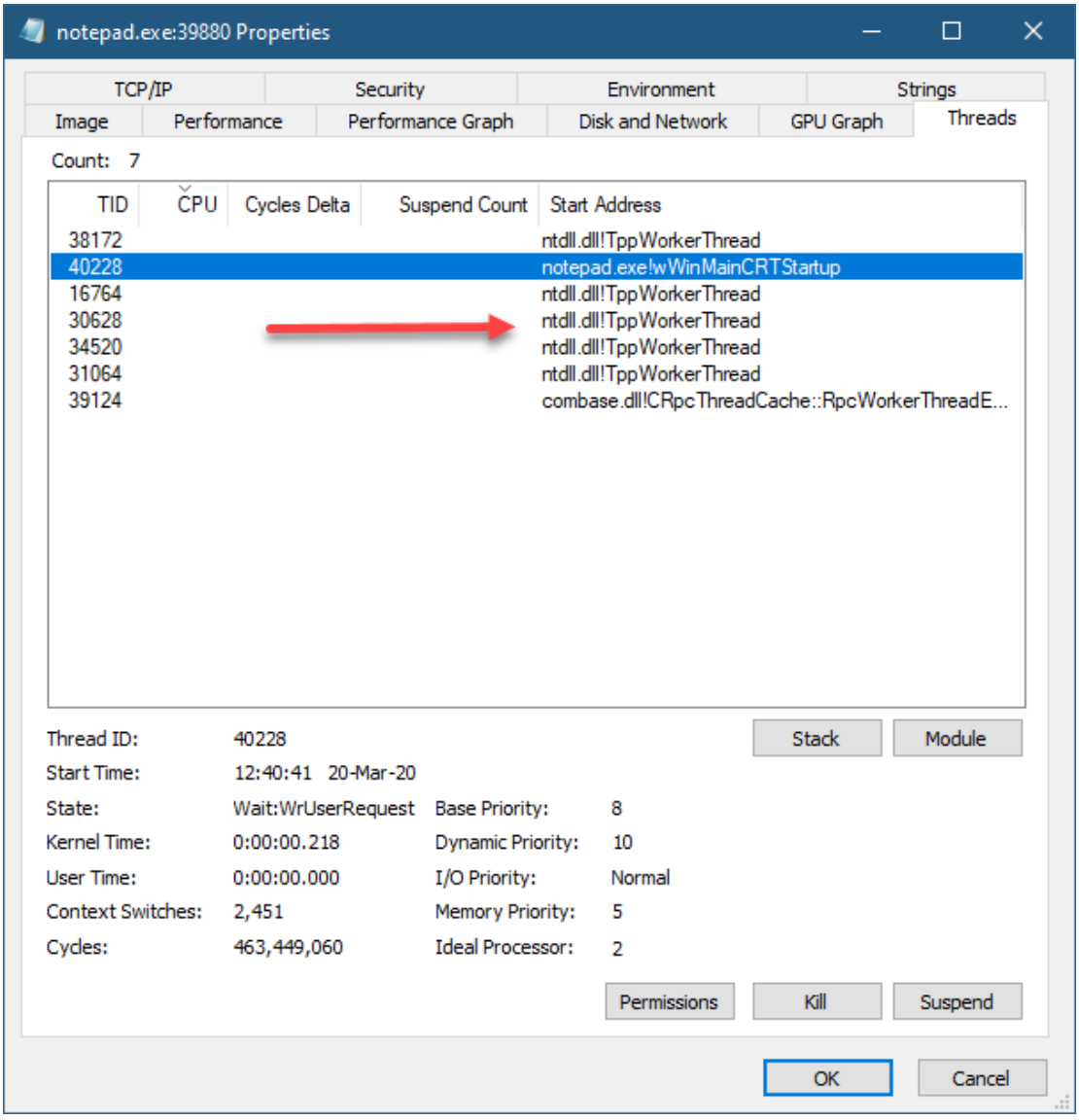


Figure 9-1: Thread pool threads in *Notepad*

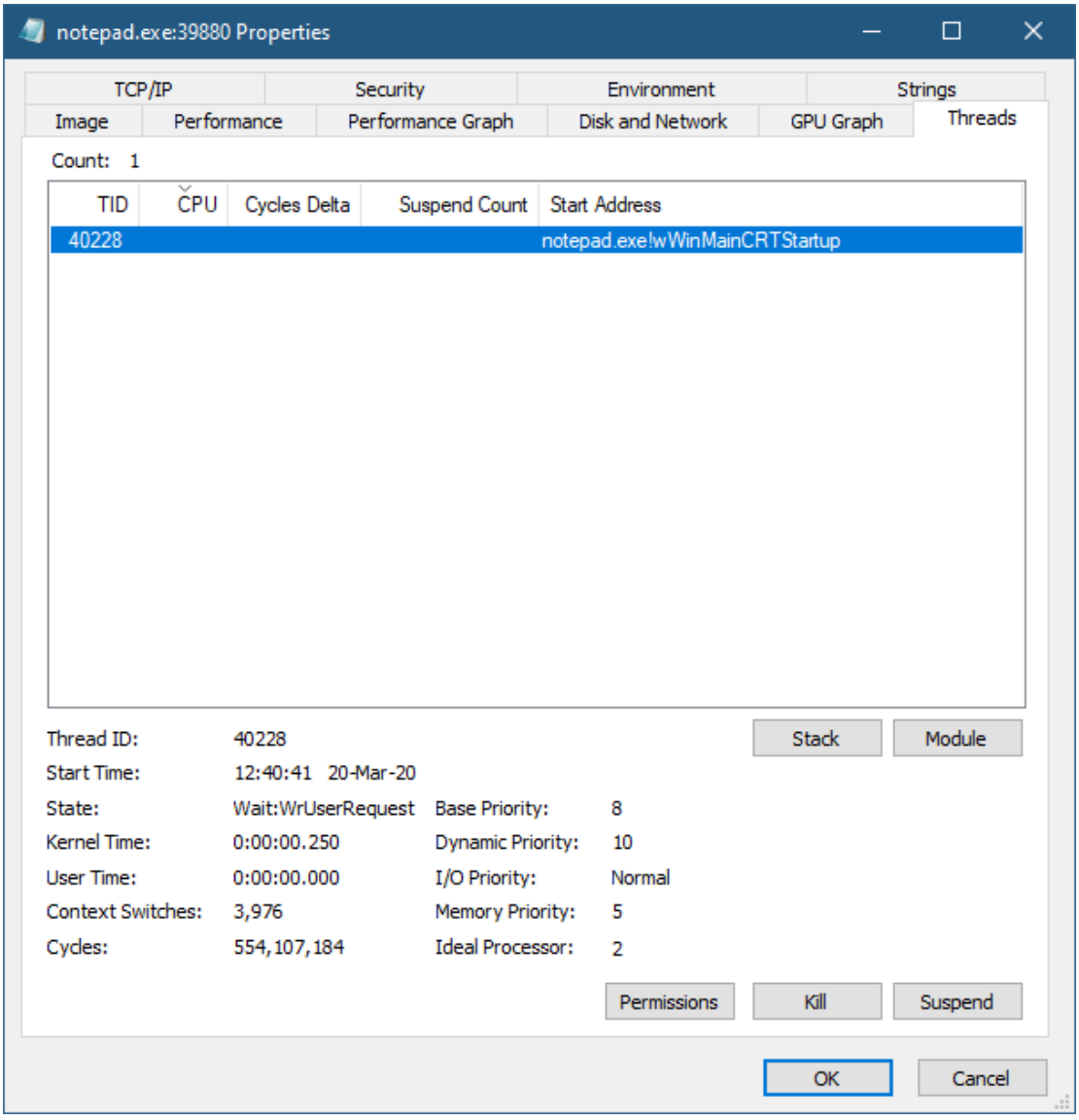


Figure 9-2: No thread pool threads in *Notepad*

You can get a sense of the number of thread pools in the system by running my *Object Explorer* tool, that can be downloaded from https://github.com/zodiacon/AllTools or https://github.com/zodiacon/ObjectExplorer/Releases. When opened, the *Object Types* is shown. Sort by name and look for *TpWorkerFactory* (figure 9-3). Note the number of such objects in the system.

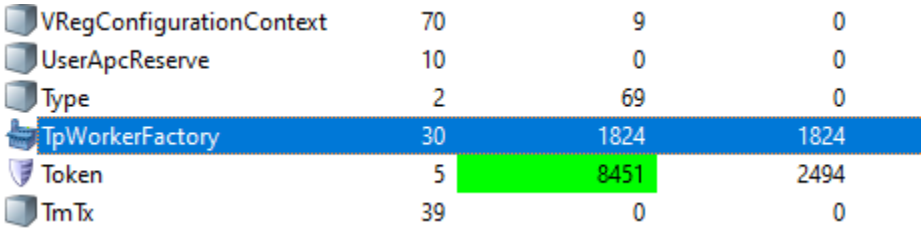


Figure 9-3: The number of *TpWorkerFactory* objects shown in *Object Explorer*

You can view more details by right-clicking the *TpWorkerFactory* object and selecting *All Objects* to view all objects of this type with some details (figure 9-4). You can get a sense of how many thread pools exist in processes.

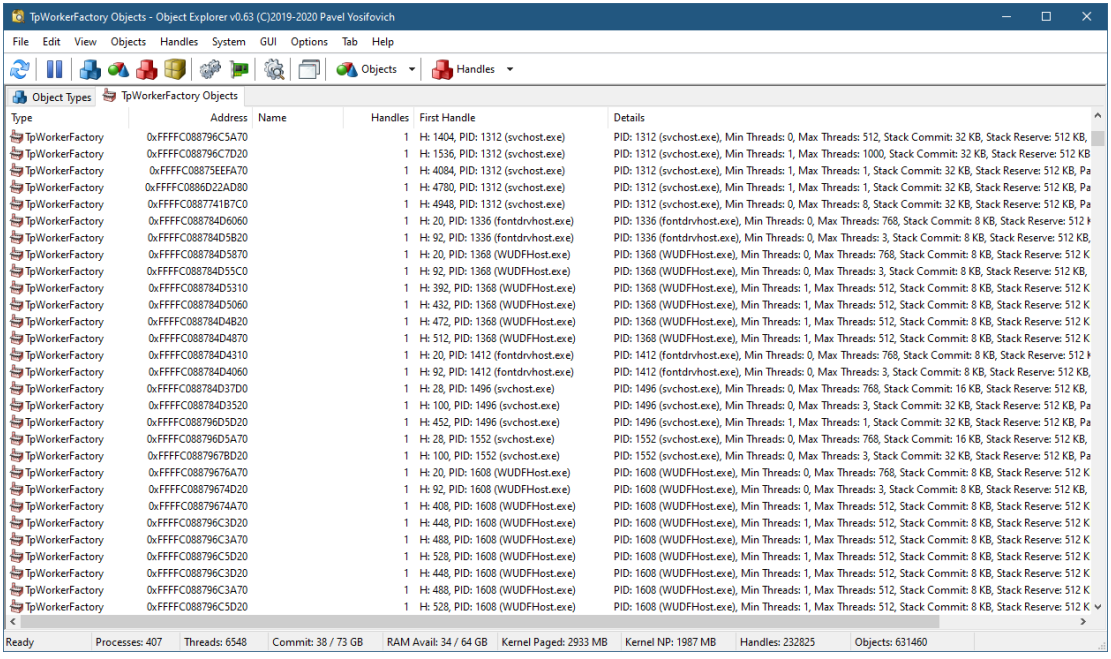


Figure 9-4: *TpWorkerFactory* objects in the system

Thread Pool Work Callbacks

The simplest API for submitting a work item for the thread pool is `TrySubmitThreadPoolCallback`:

```
typedef VOID (NTAPI *PTP_SIMPLE_CALLBACK)(
    _Inout_     PTP_CALLBACK_INSTANCE Instance,
    _Inout_opt_ PVOID                 Context);

BOOL TrySubmitThreadpoolCallback(
    _In_ PTP_SIMPLE_CALLBACK pfns,
    _Inout_opt_ PVOID pv,
    _In_opt_ PTP_CALLBACK_ENVIRON pcbe);
```

`TrySubmitThreadPoolCallback` sets up a callback provided as the first parameter, to be called by the thread pool. The second parameter allows specifying a context value that is passed as-is to the callback function. The last optional parameter, related to a *callback environment* can be set to `NULL`. We'll take a look at the callback environment later in this chapter.

The callback function itself is invoked with two parameters, where the second is the context provided to `TrySubmitThreadPoolCallback`. The first parameter of type `PTP_CALLBACK_INSTANCE` is an opaque pointer than represents this callback instance. We'll discuss this parameter later in this chapter as well.

The function returns `TRUE` on success, which should be the case most of the time, unless Windows is in extreme memory pressure. Once submitted, the callback executes as soon as possible by a thread pool thread. There is no built-in way to cancel the request once submitted. There is also no direct way of knowing when the callback finished execution. It is possible, of course, to add our own mechanism, such as signaling an event object from the callback and waiting for it on a different thread.

The Simple Work Application

The *Simple Work* sample application, shown in figure 9-5 when it's executed allows submitting work items to the thread pool with `TrySubmitThreadPoolCallback` while watching the thread under which each callback is executed. At the same time, the application shows the number of threads in the process.

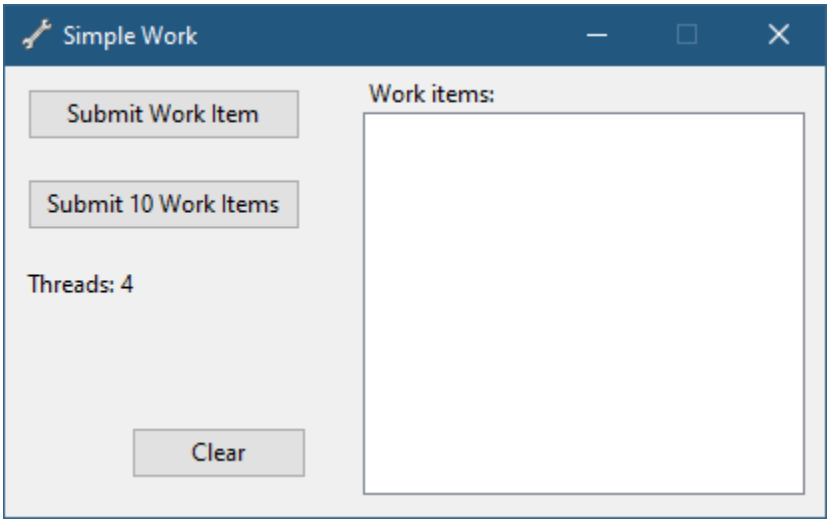


Figure 9-5: The *Simple Work* Application

When the application starts up, the number of threads should be low, usually 1 or 4. Click the *Submit Work Item* button to submit a single work item. If the number of threads is above one, it's likely to stay with the same number, as at least one thread pool thread is already alive and can pick up the request (figure 9-6).
Clicking the same button a few more times will initiate more work items, and the number of threads should increase as the thread pool "senses" a higher load.

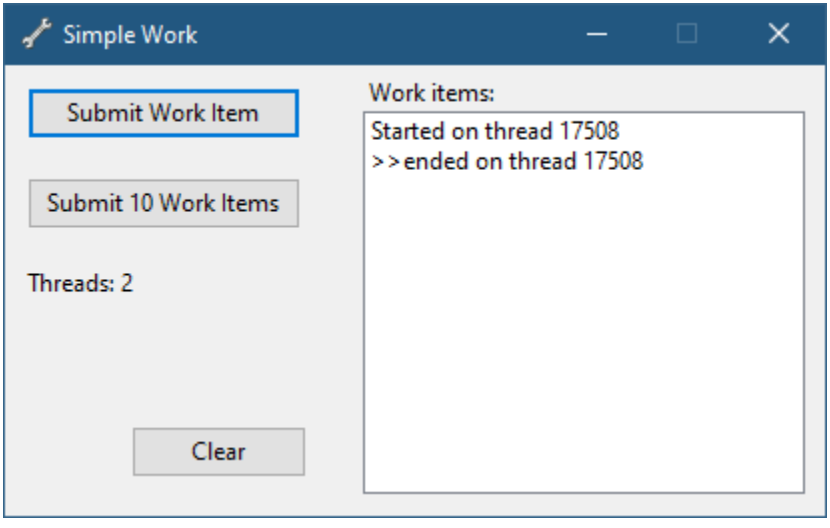


Figure 9-6: A single work item submitted to the thread pool

Now click *Submit 10 Work Items* button several times and watch the thread count go up substantially (figure 9-7).

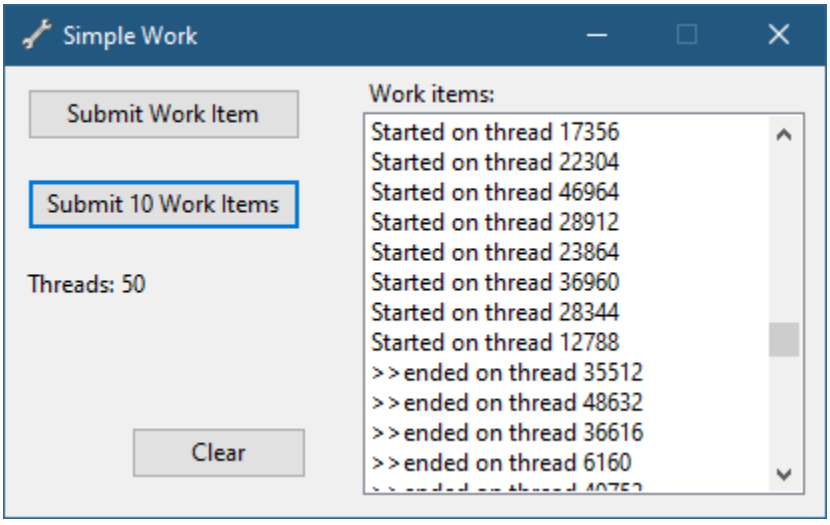


Figure 9-7: Many work items submitted to the thread pool

If you don't submit many items, simply wait some time, the thread count will start to drop. Given enough time, the thread count will drop to 1, leaving only the main UI thread alive (figure 9-8). This application shows the dynamic nature of the thread pool.

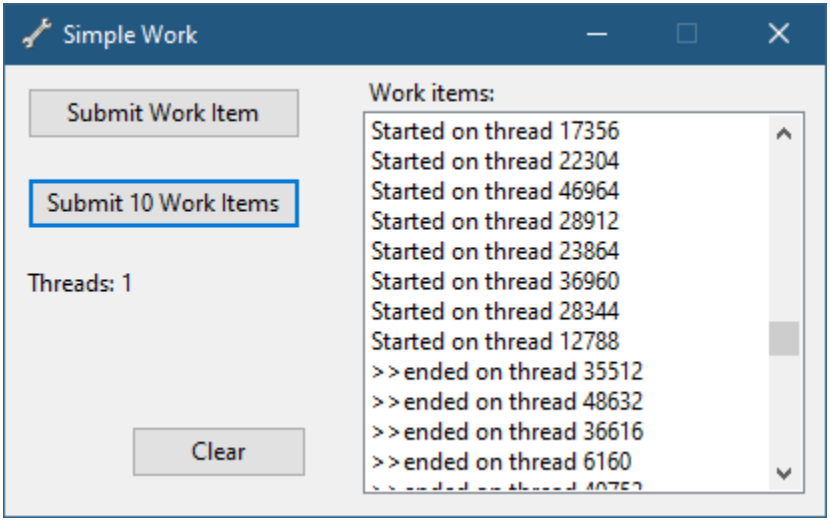


Figure 9-8: *Simple Work* after some time of inactivity

The "submit" buttons simply call `TrySubmitThreadPoolCallback`:

```
LRESULT CMainDlg::OnSubmitWorkItem(WORD, WORD wID, HWND, BOOL&) {
    if(!::TrySubmitThreadpoolCallback(OnCallback, this, nullptr))
        AtlMessageBox(*this, L"Failed to submit work item callback",
            IDR_MAINFRAME, MB_ICONERROR);

    return 0;
}

LRESULT CMainDlg::OnSubmit10WorkItems(WORD, WORD, HWND, BOOL&) {
    for (int i = 0; i < 10; i++) {
        if (!::TrySubmitThreadpoolCallback(OnCallback, this, nullptr)) {
            AtlMessageBox(*this, L"Failed to submit work item callback",
                IDR_MAINFRAME, MB_ICONERROR);
            break;
        }
    }
    return 0;
}
```

The context argument is set to `this`, so that the static callback function (`OnCallback`) has access to the dialog box object. The callback simulates work by sleeping a bit, indicating on which thread it's invoked by posting messages to the dialog box with `PostMessage`:

```
void CMainDlg::OnCallback(PTP_CALLBACK_INSTANCE instance, PVOID context) {
    auto dlg = (CMainDlg*)context;

    // post message indicating start
    dlg->PostMessage(WM_APP + 1, ::GetCurrentThreadId());

    // simulate work...
    ::Sleep(10 * (::GetTickCount() & 0xff));

    // post message indicating end
    dlg->PostMessage(WM_APP + 2, ::GetCurrentThreadId());
}
```

The messages are mapped by the normal WTL message map to functions that are invoked by the UI (main) thread. This is because any messages posted (or sent) to a window are always put in the message queue of the window's creator thread, and only that thread is allowed to retrieve messages from the queue and handle them. Here is the handler for the custom `WM_APP + 1` message:

```
LRESULT CMainDlg::OnCallbackStart(UINT, WPARAM wParam, LPARAM, BOOL&) {
    CString text;
    text.Format(L"Started on thread %d", wParam);
    m_List.AddString(text);

    return 0;
}
```

The second message is identical except for the text itself.

Getting the number of threads in the current process is surprisingly tricky, as there is no documented (or undocumented, for that matter) API to get the value directly. The *Tool help* API discussed in chapter 3 is used here to locate the current process, where the number of threads is provided as part of the `PROCESSENTRY32` structure. The code is part of a `WM_TIMER` message handler that is called every 2 seconds to update the current thread count:

```
auto hSnapshot = ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if (hSnapshot == INVALID_HANDLE_VALUE)
    return 0;

PROCESSENTRY32 pe;
pe.dwSize = sizeof(pe);

::Process32First(hSnapshot, &pe);
// skip idle process

auto pid = ::GetCurrentProcessId();
ULONG threads = 0;
while (::Process32Next(hSnapshot, &pe)) {
    if (pe.th32ProcessID == pid) {
        threads = pe.cntThreads;
        break;
    }
}
::CloseHandle(hSnapshot);

CString text;
text.Format(L"Threads: %u\n", threads);
SetDlgItemText(IDC_THREADS, text);
```

Controlling a Work Item

Using `TrySubmitThreadpoolCallback` is fairly straightforward, but sometimes you need more control. For example, you may want to know when the callback completes, or you may want to cancel the work item if some condition is satisfied. For these cases, you can create a thread pool work item explicitly. The API to achieve this is `CreateThreadPoolWork`:

```
PTP_WORK CreateThreadpoolWork(
    _In_ PTP_WORK_CALLBACK pfnwk,
    _Inout_opt_ PVOID pv,
    _In_opt_ PTP_CALLBACK_ENVIRON pcbe);
```

The function looks similar to `TrySubmitThreadpoolCallback`, with two differences. The first is the return value, which is an opaque `PTP_WORK` pointer representing the work item, or `NULL` on failure. The second difference is the callback prototype that looks like this:

```
typedef VOID (CALLBACK *PTP_WORK_CALLBACK)(
    _Inout_     PTP_CALLBACK_INSTANCE Instance,
    _Inout_opt_ PVOID                 Context,
    _Inout_     PTP_WORK              Work);
```

There is a third argument to the callback, which is the work item object returned originally from `CreateThreadpoolWork`. Once a work item object is created, it can be submitted (possibly multiple times) by calling `SubmitThreadpoolWork`:

```
VOID SubmitThreadpoolWork(
    _Inout_ PTP_WORK Work
);
```

Notice the function returns `void`, implying that it cannot fail. This is because if `CreateThreadpoolWork` succeeds, there is no way `SubmitThreadpoolWork` can fail. On the other hand, each call to `TrySubmitThreadpoolCallback` can potentially fail.

More than one call to `SubmitThreadpoolWork` is allowed using the same work object. The potential downside is that all these submissions use the same callback and the same context, as these can only be provided at work object creation. Once submitted, some control over the submitted callbacks is available with `WaitForThreadpoolWorkCallbacks`:

```
void WaitForThreadpoolWorkCallbacks(
    _Inout_ PTP_WORK pwk,
    _In_    BOOL     fCancelPendingCallbacks);
```

The first parameter is the work object returned from `CreateThreadpoolWork`. If `fCancelPendingCallbacks` is `FALSE`, the calling thread enters a wait state until all callbacks submitted through the work item have completed. If no callback has been submitted yet, the function returns immediately.

If `fCancelPendingCallbacks` is `TRUE`, the function cancels any submitted callbacks that have not started execution yet. The function never cancels a callback in progress - that would not make sense, as the only way to do that is to forcefully terminate the thread pool thread, which is a bad idea. The calling thread waits for all currently executing callbacks to complete before its wait os ended.

Finally, the thread pool work object must be be eventually freed with `CloseThreadpoolWork`:

```
void CloseThreadpoolWork(_Inout_ PTP_WORK pwk);
```

Modify the *Simple Work* application to use `CreateThreadpoolWork` and related functions described in this section. (The solution is the project called *SimpleWork2*, located with the other samples for this chapter.)

The *Windows Implementation Library* (WIL) has handles for thread pool work objects: `wil::unique_threadpool_work`, `unique_threadpool_work_nocancel`, and `unique_threadpool_work_nowait`. The "nowait" variant just closes the work object when it goes out of scope. The first two variants call `WaitForThreadpoolWorkCallbacks` to wait for all pending callbacks to complete, with the cancellation argument set to `TRUE` in the former and `FALSE` in the latter.

The MD5 Calculator Application

The MD5 calculation application from chapter 7 creates a new thread for every new calculation needed. This is inefficient, and the thread pool can be used here as an alternative. The code to be replaced is in `CView::OnStartCalc`, where currently a thread is created for each required calculation:

```
// spawn a thread to do the actual calculation
auto data = new CalcThreadData;
data->View = this;
data->Index = (int)index;

auto hThread = ::CreateThread(nullptr, 0, [](auto param) {
    auto data = (CalcThreadData*)param;
    auto view = data->View;
    auto index = data->Index;
    delete data;
    return view->DoCalc(index);
    }, data, 0, nullptr);
if (!hThread) {
    AtlMessageBox(nullptr, L"Failed to create worker thread!",
        IDR_MAINFRAME, MB_ICONERROR);
    return 0;
}

::CloseHandle(hThread);
```

The simplest way to replace this code is by using `TrySubmitThreadpoolCallback`, like so:

```
auto data = new CalcThreadData;
data->View = this;
data->Index = (int)index;

if (!::TrySubmitThreadpoolCallback([](auto instance, auto param) {
    auto data = (CalcThreadData*)param;
    auto view = data->View;
    auto index = data->Index;
    delete data;
    view->DoCalc(index);
    }, data, nullptr)) {
    AtlMessageBox(nullptr, L"Failed to submit thread pool work!",
        IDR_MAINFRAME, MB_ICONERROR);
    return 0;
}
```

Although both `CreateThread` and `TrySubmitThreadpoolCallback` can potentially fail, this is less likely to happen with `TrySubmitThreadpoolCallback` as it requires fewer resources than spawning a new thread.

The other option is to use a full work object with `CreateThreadpoolWork`. However, this has fewer benefits in this case, because we need a different context for each work item submission, so its main advantage is the ability to wait and possibly cancel pending operations. Still, let's do this with the help of WIL's wrappers:

```
auto data = new CalcThreadData;
data->View = this;
data->Index = (int)index;

wil::unique_threadpool_work_nowait work(::CreateThreadpoolWork(
    [](auto instance, auto param, auto work) {
    auto data = (CalcThreadData*)param;
    auto view = data->View;
    auto index = data->Index;
    delete data;
    view->DoCalc(index);
    }, data, nullptr));
if(!work) {
    AtlMessageBox(nullptr, L"Failed to submit thread pool work!",
        IDR_MAINFRAME, MB_ICONERROR);
    return 0;
}
::SubmitThreadpoolWork(work.get());
```

We select the `unique_threadpool_work_nowait` variant since we don't want to wait for the pending operation when the work object goes out of scope, which happens at the end of the function. In this example, there is no real benefit in using the manually created work item, but other cases may benefit from this approach.

Thread Pool Wait Callbacks

In chapter 8 in the section "Working with Events", we looked at an example where several processes need to wait on a common event ("shutdown"), and for this purpose, each process creates a thread that does the waiting. As mentioned in that section, this is inefficient - threads should do actual work rather than waiting. A better approach is to let a thread pool wait for the event. At first glance, it seems to be the same thing: isn't the thread pool thread going to just wait? And if so, what's the difference with respect to an application-created thread?

The difference is that the same thread pool thread can be waiting for multiple objects, submitted by the application and possibly other Windows APIs and libraries. Each such thread can wait for

up to 64 objects at the same time using the familiar `WaitForMultipleObjects`. If more than 64 waits are required, another thread from the pool can be launched for this purpose.

To create a thread pool wait object, call `CreateThreadpoolWait`:

```
typedef VOID (NTAPI *PTP_WAIT_CALLBACK)(
    _Inout_     PTP_CALLBACK_INSTANCE Instance,
    _Inout_opt_ PVOID                 Context,
    _Inout_     PTP_WAIT              Wait,
    _In_        TP_WAIT_RESULT        WaitResult);  // just a DWORD

PTP_WAIT CreateThreadpoolWait(
    _In_ PTP_WAIT_CALLBACK pfnwa,
    _Inout_opt_ PVOID pv,
    _In_opt_ PTP_CALLBACK_ENVIRON pcbe);
```

The pattern should be apparent at this point, as the function and callback are very similar to `CreateThreadpoolWork`. The parameters to `CreateThreadpoolWait` are the same with a minor twist to the callback function. It provides an extra argument, which specifies why the callback was invoked; that is, the callback is invoked when a wait operation for an object is over, so `WaitResult` specifies why it was over. Possible values include `WAIT_OBJECT_0` to indicate the object was signaled and `WAIT_TIMEOUT` indicating the timeout expired without the object being signaled.

`CreateThreadpoolWait` returns an opaque `PTP_WAIT` pointer that represents the wait object. Now actual wait requests can be submitted with `SetThreadpoolWait`:

```
VOID SetThreadpoolWait(
    _Inout_ PTP_WAIT pwa,
    _In_opt_ HANDLE h,
    _In_opt_ PFILETIME pftTimeout);
```

Apart from the wait object, the function accepts the handle to wait on, and the timeout argument specifies how long to wait. The format of this value is the same discussed in chapter 8 in relation to waitable timers: a negative number indicates the relative time in 100nsec units, and a positive number indicates absolute time counted from January 1st, 1601, midnight UTC, in 100nsec units. Refer to chapter 8 for the full discussion of how to specify this value. A `NULL` pointer indicates infinite wait.

Contrary to thread pool work object, calling `SetThreadpoolWait` with the same wait object (`PTP_WAIT`) causes cancellation of the current wait (if it has not already executed) and replaces the wait with the (possibly) new handle and timeout. Setting the handle to `NULL` stops queuing new callbacks.

For our example of a shutdown event, the wait could be accomplished by code such as the following:

```
void ConfigureWait() {
    HANDLE hShutdown = ::CreateEvent(nullptr, TRUE, FALSE, L"ShutdownEvent");
    auto wait = ::CreateThreadpoolWait(OnWaitSatisfied, nullptr, nullptr);
    ::SetThreadpoolWait(wait, hShutdown, nullptr);
    // continue running normally...
}

void OnWaitSatisfied(PTP_CALLBACK_INSTANCE instance, PVOID context,
    PTP_WAIT wait, TP_WAIT_RESULT) {
    // Since the wait request specified infinite time, the fact that we're here
    // means the event was signaled
    DoShutdown();       // initiate shutdown
}
```

An extended set function is available as well:

```
BOOL SetThreadpoolWaitEx(
    _Inout_ PTP_WAIT pwa,
    _In_opt_ HANDLE h,
    _In_opt_ PFILETIME pftTimeout,
    _Reserved_ PVOID Reserved);
```

The function is essentially identical to `SetThreadpoolWait`, except the return value. Instead of `void`, it returns `TRUE` in means a wait was active and has now been replaced. If `FALSE` is returned, it means a callback was executed for the previously registered handle or about to be executed.

Similarly to a thread pool work item, some control is possible with `WaitForThreadpoolWaitCallbacks`:

```
VOID WaitForThreadpoolWaitCallbacks(
    _Inout_ PTP_WAIT pwa,
    _In_ BOOL fCancelPendingCallbacks);
```

`WaitForThreadpoolWaitCallbacks` has essentially the same semantics and behavior as `WaitForThreadpoolWorkCallbacks`.

Finally, the Wait item needs to be freed with `CloseThreadpoolWait`:

```
VOID CloseThreadpoolWait(_Inout_ PTP_WAIT pwa);
```

Thread Pool Timer Callbacks

In chapter 8, we looked at a waitable timer kernel object that can be used to initiate operations when it expires, optionally periodically. However, initiating operations was not very convenient. It required some wait operation (which now we know can be accomplished with the thread pool), or running a callback as an APC on the thread that called `SetWaitableTimer`. The thread pool provides yet another service that calls a callback directly (from the pool) after a specified period elapses, and optionally periodically.

The semantics of the relevant functions is very similar to what we've seen already. Here is the thread pool timer object creation function:

```
typedef VOID (CALLBACK *PTP_TIMER_CALLBACK)(
    _Inout_     PTP_CALLBACK_INSTANCE Instance,
    _Inout_opt_ PVOID                 Context,
    _Inout_     PTP_TIMER             Timer);

PTP_TIMER CreateThreadpoolTimer(
    _In_ PTP_TIMER_CALLBACK pfnti,
    _Inout_opt_ PVOID pv,
    _In_opt_ PTP_CALLBACK_ENVIRON pcbe);
```

The function parameters should be self-explanatory at this point. `CreateThreadpoolTimer` returns an opaque pointer representing the timer object. To initiate an actual timer, call `SetThreadpoolTimer`:

```
VOID SetThreadpoolTimer(
    _Inout_ PTP_TIMER pti,
    _In_opt_ PFILETIME pftDueTime,
    _In_ DWORD msPeriod,
    _In_opt_ DWORD msWindowLength);
```

The `pftDueTime` parameter specifies the time for expiration, with the aforementioned format used with `SetWaitableTimer` and `SetThreadpoolWait`, event though it's typed as `FILETIME`. If this parameter is `NULL`, it causes the timer object to stop queuing new expiration requests (but those already queued will be invoked when expiration is due). The `msPeriod` is the requested period, in milliseconds. If zero is specified, the timer is one-shot. The last parameter

is used similarly to the last argument to `SetWaitableTimerEx` - an acceptable tolerance (in milliseconds), so that timer coalescing can occur to conserve power.

Calling the function a second time with the same timer object cancels the callback and replaces the timer with the new information.

An extended function exists, similar to the wait object variant:

```
BOOL SetThreadpoolWaitEx(
    _Inout_ PTP_WAIT pwa,
    _In_opt_ HANDLE h,
    _In_opt_ PFILETIME pftTimeout,
    _Reserved_ PVOID Reserved);
```

As with the wait object, the only difference compared to the non-`Ex` function is the return value. The function returns `TRUE` if a previous timer was in effect and now has been replaced. It returns `FALSE` otherwise.

To determine whether there is a timer set on the timer object, call `IsThreadpoolTimerSet`:

```
BOOL WINAPI IsThreadpoolTimerSet(_Inout_ PTP_TIMER pti);
```

As you might expect, waiting and possible cancellation of the timer is possible with `WaitForThreadpoolTimerCallbacks`:

```
VOID WaitForThreadpoolTimerCallbacks(
    _Inout_ PTP_TIMER pti,
    _In_ BOOL fCancelPendingCallbacks);
```

Lastly, the timer pool object should be closed:

```
VOID CloseThreadpoolTimer(_Inout_ PTP_TIMER pti);
```

The Simple Timer Sample

The *Simple Timer* sample from chapter 8 can be re-written to use a thread pool timer object instead of using a waitable timer. Here is the complete code:

```
void CALLBACK OnTimer(PTP_CALLBACK_INSTANCE inst, PVOID context, PTP_TIMER time\
r) {
    printf("TID: %u Ticks: %u\n", ::GetCurrentThreadId(), ::GetTickCount());
}

int main() {
    auto timer = ::CreateThreadpoolTimer(OnTimer, nullptr, nullptr);
    if (!timer) {
        printf("Failed to create a thread pool timer (%u)", ::GetLastError());
        return 1;
    }

    static_assert(sizeof(LONG64) == sizeof(FILETIME), "something weird!");

    LONG64 interval;
    interval = -10000 * 1000LL;
    ::SetThreadpoolTimer(timer, (FILETIME*)&interval, 1000, 0);

    printf("Main thread ID: %u\n", ::GetCurrentThreadId());

    ::Sleep(10000);
    ::WaitForThreadpoolTimerCallbacks(timer, TRUE);
    ::CloseThreadpoolTimer(timer);

    return 0;
}
```

The `static_assert` keyword makes a sanity check so that a 64-bit integer can be treated the same as a `FILETIME` structure. The timer is set for a one second interval with a period of one second. The call to `Sleep` just causes the main thread to wait, all the while timer callbacks are invoked as specified.

Thread Pool I/O Callbacks

I/O callbacks handled by the thread pool are used to service asynchronous I/O operations. This is discussed in chapter 11, “File and Device I/O”.

Thread Pool Instance Operations

There are two parameters we have not described yet - the callback environment and the instance parameter provided to callbacks. In this section, we'll look at the instance parameter, and in the next section we'll examine the callback environment.

The opaque instance parameter (typed as `PTP_CALLBACK_INSTANCE`) is used with several functions callable from the callback itself. We'll start with `CallbackMayRunLong`:

```
BOOL CallbackMayRunLong(_Inout_ PTP_CALLBACK_INSTANCE pci);
```

Calling this function provides a hint to the thread pool that this callback may be long running, so the thread pool should consider this thread not part of the thread pool thread limits and spawn a new thread to service the next request, since this one is unlikely to end any time soon. The function returns `TRUE` to indicate that the thread pool is able to spawn a new thread for the next request. It returns `FALSE` if it cannot do so at this time. The long-running flag is still applied to the instance just the same.

The next set of functions request the thread pool to perform a certain operation before the callback really ends and the thread can go back to the pool:

```
VOID SetEventWhenCallbackReturns(
    _Inout_ PTP_CALLBACK_INSTANCE pci,
    _In_ HANDLE evt);
VOID ReleaseSemaphoreWhenCallbackReturns(
    _Inout_ PTP_CALLBACK_INSTANCE pci,
    _In_ HANDLE sem,
    _In_ DWORD crel);
VOID ReleaseMutexWhenCallbackReturns(
    _Inout_ PTP_CALLBACK_INSTANCE pci,
    _In_ HANDLE mut);
VOID LeaveCriticalSectionWhenCallbackReturns(
    _Inout_ PTP_CALLBACK_INSTANCE pci,
    _Inout_ PCRITICAL_SECTION pcs);
VOID FreeLibraryWhenCallbackReturns(
    _Inout_ PTP_CALLBACK_INSTANCE pci,
    _In_ HMODULE mod);
```

The first four functions are fairly straightforward, calling the following functions before the callback returns: `SetEvent`, `ReleaseSemaphore` (with a count of the `crel` parameter), `ReleaseMutex`, `LeaveCriticalSection`, respectively. This may not look like big deal - can't

the callback function provided by the client just call the appropriate function? It can, but that would require the relevant object/handle to be passed via the context parameter in the original function which may be inconvenient, and even problematic as the context can only be set once when the related thread pool is created.

The last function in the list, `FreeLibraryWhenCallbackReturns` calls `FreeLibrary` to unload a dynamic link library (DLL) at the end of the callback. It has the extra benefit of being able to unload a DLL from which the callback itself originated. Calling `FreeLibrary` by the callback itself is fatal, as the function would unload its own code, resulting in a memory access violation once `FreeLibrary` returns. Having the thread pool call `FreeLibray` solves the problem, since the caller is not part of the DLL.

The last function that works on an instance parameter is `DisassociateCurrentThreadFromCallback`:

```
void DisassociateCurrentThreadFromCallback(_Inout_ PTP_CALLBACK_INSTANCE pci);
```

Calling this function tells the thread pool that this callback finished its important work, so other threads that may be waiting on functions such as `WaitForThreadpoolWorkCallbacks` to satisfy their wait, even though the callback is technically still executing. if the callback is part of a cleanup group (described in the next section), this function does not change that association.

The Callback Environment

Each of the thread pool object creation functions has a last parameter of type `PTP_CALLBACK_ENVIRON`, called a *callback environment*. Its purpose is to allow further customization when using thread pool functions. The structure itself is defined in the *<winnt.h>* header, but you should consider the structure as opaque and only manipulate it through the functions described in this section.

To initialize a callback environment to a clean state, call `InitializeThreadpoolEnvironment`:

```
VOID InitializeThreadpoolEnvironment(_Out_ PTP_CALLBACK_ENVIRON pcbe);
```

The callback environment API functions are implemented inline by calling other functions from *ntdll.dll*. For example, `InitializeThreadpoolEnvironment` calls `TpInitializeCallbackEnviron`, also implemented in the same header file. The actual effect of a callback environment is only evident once it's passed to thread pool functions.

Back to `InitializeThreadpoolEnvironment` - the function currently zeroes all members except the `Version` field that is set to `3` (Windows 7 and later) or `1` (Vista). Eventually, the environment should be destroyed with `DestroyThreadpoolEnvironment`:

```
VOID DestroyThreadpoolEnvironment(_Inout_ PTP_CALLBACK_ENVIRON);
```

Currently, this function does nothing, but this may change in future versions of Windows, so it's a good practice to call it once the environment object is no longer needed.

Once a callback environment is initialized, a set of functions can be called to customize various members of the environment structure. Table 9-1 summarizes the functions and their meaning.

Table 9-1: Functions for manipulating a callback environment

Function	Description (effective for the callbacks associated with the callback environment)
`SetThreadpoolCallbackPool`	Sets the thread pool object to use for the callbacks
`SetThreadpoolCallbackPriority`	Sets the priority of the callbacks
`SetThreadpoolCallbackRunsLong`	Sets a hint that the callbacks are long-running
`SetThreadpoolCallbackLibrary`	Indicates the callbacks are part of a DLL, synchronizing some parts of DLL handling
`SetThreadpoolCallbackCleanupGroup`	Associates the callback with a cleanup group (described later in this chapter)

The relationship between a callback environment, a thread pool and various thread pool items is illustrated in figure 9-9.

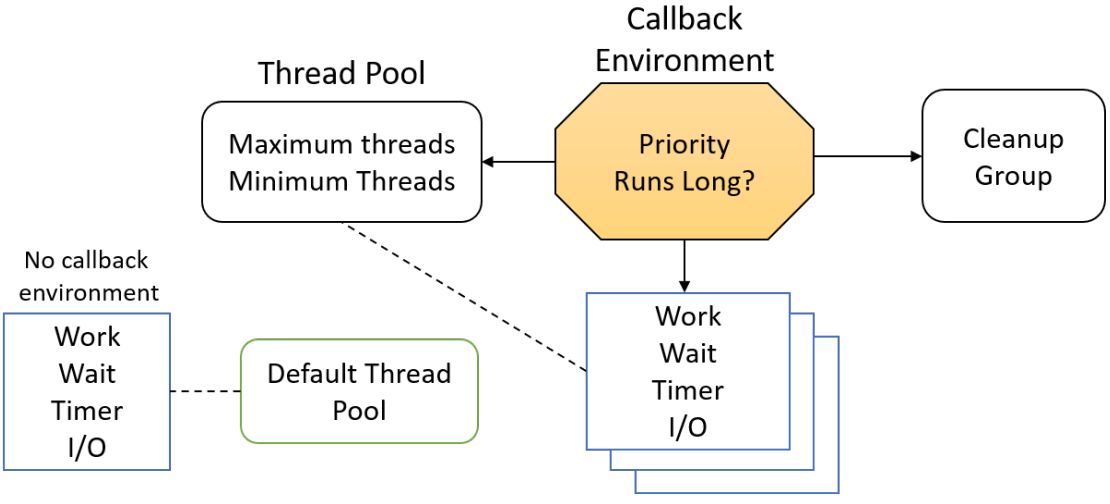


Figure 9-9: Relationships between various thread pool entities

Some functions require a more detailed discussion. `SetThreadpoolCallbackPool` sets a differ-

ent thread pool than the process default. The next section shows how to create thread pools.

`SetThreadpoolCallbackPriority` provides the opportunity to set a priority for callbacks relative to other callback running on the same thread pool:

```
VOID SetThreadpoolCallbackPriority(
    _Inout_ PTP_CALLBACK_ENVIRON pcbe,
    _In_    TP_CALLBACK_PRIORITY Priority);
```

The priority parameter can be one of the values from the `TP_CALLBACK_PRIORITY` enumeration:

```
typedef enum _TP_CALLBACK_PRIORITY {
    TP_CALLBACK_PRIORITY_HIGH,
    TP_CALLBACK_PRIORITY_NORMAL,
    TP_CALLBACK_PRIORITY_LOW,
} TP_CALLBACK_PRIORITY;
```

Higher priority callbacks are guaranteed to start before lower priority ones. This provides some level of flexibility within the same thread pool.

> This function was added in Windows 7 and Server 2008 R2.

`SetThreadpoolCallbackLibrary` is called to let the thread pool know the callback is part of a DLL, and so it should keep the DLL loaded in the process as long as there are callbacks for that environment. It also helps in preventing a deadlock if other threads try to acquire the *Loader Lock* (the loader lock is discussed in the chapter "Dynamic Link Libraries").

Private Thread Pools

By default, the process has a single thread pool, which cannot be destroyed. It's the pool targeted when callbacks don't have a custom callback environment. With a callback environment, it's possible to target callbacks to a different thread pool by calling `SetThreadpoolCallbackPool`:

```
VOID SetThreadpoolCallbackPool(
    _Inout_ PTP_CALLBACK_ENVIRON pcbe,
    _In_    PTP_POOL             ptpp);
```

The thread pool itself is represented by the opaque `PTP_POOL` pointer. A private thread pool is created with the `CreateThreadpool` function:

```
PTP_POOL CreateThreadpool(_Reserved_ PVOID reserved);
```

As indicated, the only parameter is reserved and should be set to `NULL`. If the function succeeds, it returns an opaque pointer used in subsequent calls. `NULL` is returned on failure, which may happen in extreme low resources conditions.

With a thread pool in hand, functions are available to customize it to some extent. The most important functions are related to the minimum and the maximum number of threads in the pool:

```
VOID SetThreadpoolThreadMaximum(
    _Inout_ PTP_POOL ptpp,
    _In_ DWORD cthrdMost);
BOOL SetThreadpoolThreadMinimum(
    _Inout_ PTP_POOL ptpp,
    _In_ DWORD cthrdMic);
```

The functions are pretty self-explanatory. The default maximum number of threads is 512 and the minimum is 0. However, these numbers should not be relied upon, so it's better to call the above functions to set appropriate values - after all this is one of the primary reasons to create private thread pools. If the minimum number of threads is greater than zero, this number of threads is created upfront, ready to process callbacks.

Curiously enough, there are no documented reciprocal functions to get the current minimum and the maximum number of threads.

> This can be achieved by calling the native API function `NtQueryInformationWorkerFactory` (see the sourec code of my *Object Explorer* tool for an example of how to do it).

Another customization supported for private thread pools is the stack sizes used for threads in that pool:

```
typedef struct _TP_POOL_STACK_INFORMATION {
    SIZE_T StackReserve;
    SIZE_T StackCommit;
}TP_POOL_STACK_INFORMATION, *PTP_POOL_STACK_INFORMATION;

BOOL SetThreadpoolStackInformation(
    _Inout_ PTP_POOL ptpp,
    _In_ PTP_POOL_STACK_INFORMATION ptpsi);
```

The default sizes come from the PE header as described in chapter 5. That is, the default of the defaults is 4KB committed memory and 1MB reserved. This may be too much or too little, so calling `SetThreadpoolStackInformation` allows better utilization of memory for thread pool threads.

Curiously enough, the stack sizes do have a query function:

```
BOOL QueryThreadpoolStackInformation(
    _In_ PTP_POOL ptpp,
    _Out_ PTP_POOL_STACK_INFORMATION ptpsi);
```

There are more ways to customize a thread pool object, but these are not (currently) exposed by the Windows API, so they will not be described here.

Lastly, a thread pool needs to be properly destroyed by calling `CloseThreadpool`:

```
VOID CloseThreadpool(_Inout_ PTP_POOL ptpp);
```

Update the *Simple Work* application to use a private thread pool and change the maximum and minimum threads, as well as stack sizes. (the solution is in the *SimpleWork3* project).

Cleanup Groups

In a heavily used thread pool application, it may be difficult to know when to close the various thread pools, work items, wait items, etc. A *cleanup group* keeps track of all callbacks associated with it so they can be closed with a single stroke, without the application having to keep track of all callbacks manually. Note that this applies to a private thread pool only, as the default thread pool cannot be destroyed.

A cleanup group is associated with a callback environment discussed in the previous section. The first step is to create a new cleanup group with `CreateThreadpoolCleanupGroup`:

```
PTP_CLEANUP_GROUP CreateThreadpoolCleanupGroup();
```

As expected, the function returns an opaque pointer representing the cleanup group. To have any effect, the cleanup group must be associated with a callback environment with `SetThreadpoolCallbackCleanupGroup`:

```
VOID SetThreadpoolCallbackCleanupGroup(
    _Inout_  PTP_CALLBACK_ENVIRON              pcbe,
    _In_     PTP_CLEANUP_GROUP                 ptpcg,
    _In_opt_ PTP_CLEANUP_GROUP_CANCEL_CALLBACK pfng);
```

The function accepts an already initialized callback environment, the cleanup group to associate with the environment and an optional callback of the following form:

```
typedef VOID (CALLBACK *PTP_CLEANUP_GROUP_CANCEL_CALLBACK)(
    _Inout_opt_ PVOID ObjectContext,
    _Inout_opt_ PVOID CleanupContext);
```

The optional callback is invoked if the cleanup group is canceled (discussed shortly). Every time functions such as `CreateThreadpoolWork`, `CreateThreadpoolWait`, etc. are called, they are tracked by the associated cleanup group. When you want to clean up everything, call `CloseThreadpoolCleanupGroupMembers`:

```
VOID CloseThreadpoolCleanupGroupMembers(
    _Inout_ PTP_CLEANUP_GROUP ptpcg,
    _In_ BOOL fCancelPendingCallbacks,
    _Inout_opt_ PVOID pvCleanupContext);
```

The function waits for all outstanding callbacks to complete. This saves you from calling the various close functions for the items created for this pool. If `fCancelPendingCallbacks` is `TRUE`, all callbacks that have not started yet are canceled, and the callback provided to `SetThreadpoolCallbackCleanupGroup` (if not `NULL`) is called for each canceled item. The callback is called with the original context argument set in the `CreateThreadPool*` (work, wait, timer, I/O) and the cleanup context provided in `SetThreadpoolCallbackCleanupGroup` in the last parameter.

Once the wait and optional cancellation are done, the thread pool and cleanup group can be closed gracefully with `CloseThreadpool` and `CloseThreadpoolCleanupGroup`:

```
VOID CloseThreadpoolCleanupGroup(_Inout_ PTP_CLEANUP_GROUP ptpcg);
```

Exercises

1. Use the thread pool to implement the Mandelbrot exercises from chapter 5.
2. Use the thread pool to implement the exercise from chapter 8.

Summary

Thread pools are a great mechanism to improve performance and scalability in a heavily multithreaded process. In the next chapter, we'll round up some advanced (and some not so advanced) features related to threading that didn't fit well in previous chapters.

Chapter 10: Advanced Threading

This chapter rounds off threads-related topics that didn't fit well in previous chapters.

In this chapter:

- **Thread Local Storage**
- **Remote Threads**
- **Thread Enumeration**
- **Caches and Cache Lines**
- **Wait Chain Traversal**
- **User Mode Scheduling**
- **Init Once Initialization**
- **Debugging Multithreaded Applications**

Thread Local Storage

A thread naturally has access to its stack data, and to process-wide global variables. However, it's sometimes convenient to have some storage on a thread by thread basis, but accessible in a uniform way. A classic example is the `GetLastError` function we are familiar with. Although any thread can call `GetLastError`, the result is different for each threading making the access. One way to handle that would be to store some hash table keyed by the thread ID and then find the value based on that key. That would work, but it has some drawbacks. One is that the hash table would need synchronization as multiple threads may access it concurrently. Second, searching for the correct thread is perhaps not as fast as one would hope.

Thread Local Storage (TLS) is a user-mode mechanism that allows storing data on a per-thread basis, accessible by each thread in the process, but only to its own data; the access method, however, is uniform.

The value stored for `GetLastError` is not stored in TLS; it's stored as part of the *Thread Environment Block* (TEB) structure that is maintained for each thread, but it is the same idea.

Another classic example of TLS usage is in the C/C++ standard library. In the days when the C standard library was conceived back in the early 1970s, there was no concept of multithreading. So the C runtime maintains a set of global variables as state for certain operations. For example, the following classic C code attempts to open a file and deal with possible errors:

```
FILE* fp = fopen("somefile.txt", "r");
if(fp == NULL) {
    // something went wrong
    switch(errno) {
        case ENOENT:    // no such file
            //
            break;

        case EFBIG:
            //
            break;
        //
    }
}
```

Any I/O error is reflected in the global `errno` variable. But in a multithreaded application, this is a problem. Imagine thread 1 making an I/O function call, causing `errno` to change. Before it can check its value, thread 2 makes an I/O call as well, changing `errno` again. This causes thread 1 to examine a value produced because of thread 2's activity.

The net result of this is that `errno` cannot be a global variable. And so today, `errno` is not a variable, but rather a macro, that calls a function, `errno()`, that uses thread-local storage to retrieve the value for the current thread. Similarly, the implementation of I/O functions such as `fopen` store the error result to the current thread using TLS.

> The same idea applies to other global variables maintained by the C/C++ runtime library.

Dynamic TLS

The Windows API provides 4 functions for TLS usage. The first function allocates a slot for every thread in the process (and future threads):

```
DWORD TlsAlloc();
```

The function returns an available slot index and zeros all the corresponding cells for all existing threads to zero. If the function returns `TLS_OUT_OF_INDEXES` (defined as `0xffffffff`), it means the function failed and all available slots are allocated. The number of slots that are guaranteed to be available is defined as `TLS_MINIMUM_AVAILABLE` (currently 64). This may seem a lot, but that's not necessarily the case. TLS is fairly useful with DLLs, where a DLL may want to store some piece of information on a per-thread basis, so it would allocate a lot when loaded and use it when required. If you take a look at a typical process, the number of DLLs can be easily higher than 100. In practice, the number of available slots is higher than 64. If you try allocating slots until failure, you can get a sense of how many slots are available:

```
int slots = 0;
while (true) {
    DWORD slot = ::TlsAlloc();

    if (slot == TLS_OUT_OF_INDEXES) {
        printf("Out of TLS indices!\n");
        break;
    }
    slots++;
}
printf("Allocated: %d\n", slots);
```

Running this on my Windows 10 version 2004 in basic console application yields 1084 slots. If you examine the actual slot values, some are already allocated.

> Internally, 64 slots are allocated beforehand, and so are always available. If more are requested, 1024 more are allocated if possible (remember each thread needs its own TLS array). However, you should not rely on this number or behavior.

Each cell in TLS is a pointer-sized value, so the best practice here is to use a single slot, and allocate dynamically whatever structure is needed to hold all the information that you would need to store in TLS, and just store the pointer to the data in the slot itself.

Once an index is available, two functions are used to store or retrieve a value to/from a slot:

```
BOOL TlsSetValue(
    _In_ DWORD dwTlsIndex,
    _In_opt_ PVOID pTlsValue);

PVOID TlsGetValue(_In_ DWORD dwTlsIndex);
```

The functions are fairly straightforward to use. A thread calling these functions can only access its own value in the specific slot index. There is no direct way to access another thread's TLS slot - that would defeat the purpose. This also implies that no synchronization is ever necessary when accessing TLS since only a single thread can access the same address in memory. The TLS arrays are illustrated in figure 10-1.

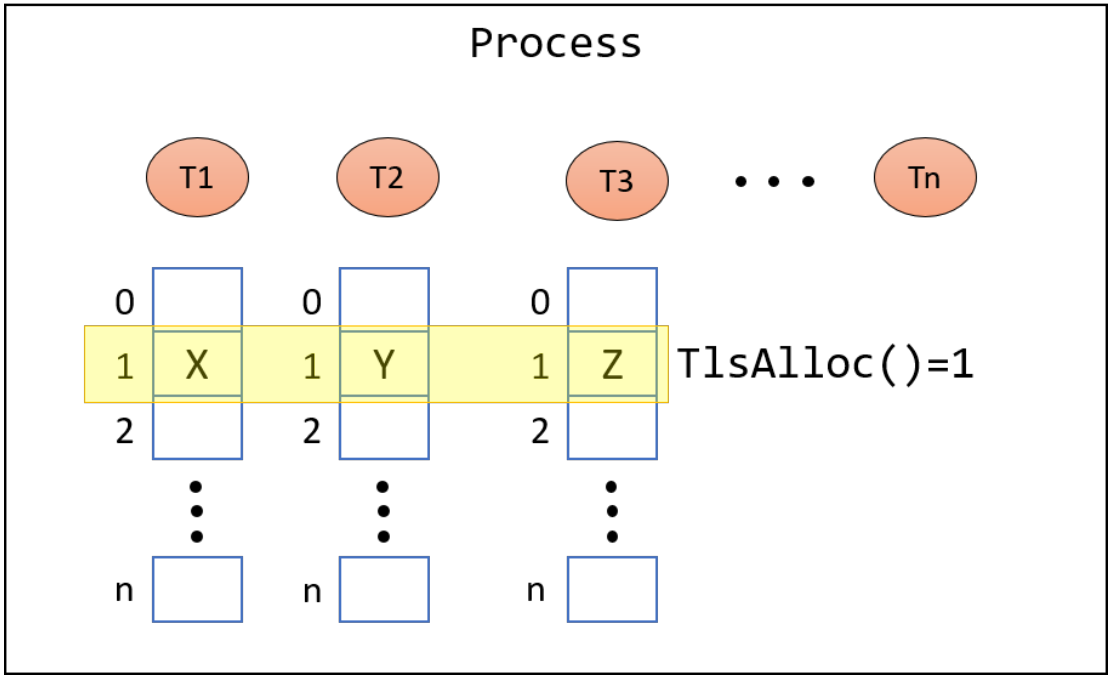


Figure 10-1: Thread Local Storage

Eventually, the TLS index allocated by `TlsAlloc` needs to be freed with `TlsFree`:

```
BOOL TlsFree(_In_ DWORD dwTlsIndex);
```

One non-standard use for TLS is to pass arguments to functions without actually passing an argument. For example, suppose there is a function that already exists and its prototype cannot change. If that function needs an additional context when called, how can you pass along an extra argument? TLS is a nice solution to the problem. The only thing that needs sharing is the TLS index allocated for this purpose.

Here is a more concrete example. Let's assume we have a class named `Transaction` that manages a transaction of some kind. Operations may be part of a transaction, but they may be invoked without a transaction. How can we model such a constraint? Perhaps the obvious answer is to add a `Transaction*` parameter to each function in the system, so that each function can make decisions based on whether it's part of a transaction or not.

However, this could be problematic if the functions already exist. Adding another parameter is non-trivial and could have a ripple effect. In some cases it cannot be done without breaking most code if virtual functions are involved, where any change to their signature could have a cascading effect that can get out of control.

Of course, an alternative exists by (again) managing some hash table of transaction objects keyed by thread Id, but this is inefficient (managing a hash table and requires locking) - a transaction is single threaded, it propagates on a thread by thread basis.

TLS offers an elegant solution. Here is an example `Transaction` class declaration:

```
class Transaction {
public:
    Transaction();
    ~Transaction();

    static Transaction* GetCurrent();

    void AddLog(PCWSTR text);
    void AddError(PCWSTR text);

private:
    int _errors = 0;
    // requires C++ 17 compiler
    inline static DWORD _tlsIndex = TLS_OUT_OF_INDEXES;
};

// pre C++ 17 compiler
DWORD Transaction::_tlsIndex = TLS_OUT_OF_INDEXES;
```

Whenever a transaction is scope, function can get it from TLS indirectly by calling `Transaction::GetCurrent()`. If the return value is `NULL`, there is no transaction. Otherwise, there is a transaction and the code can use it. Here is a conceptual implementation of the `Transaction` class:

```
Transaction::Transaction() {
    if (_tlsIndex == TLS_OUT_OF_INDEXES)
        _tlsIndex = ::TlsAlloc();

    ::TlsSetValue(_tlsIndex, this);
}

Transaction::~Transaction() {
    if (_errors == 0) {
        // commit transaction
    }
    else {
        // abort/rollback transaction
    }
    ::TlsSetValue(_tlsIndex, nullptr);
}

Transaction* Transaction::GetCurrent() {
    if (_tlsIndex == TLS_OUT_OF_INDEXES)
        return nullptr;

    return static_cast<Transaction*>(::TlsGetValue(_tlsIndex));
}

void Transaction::AddError(PCWSTR) {
    _errors++;
    // more code
}

void Transaction::AddLog(PCWSTR) {
    // more code
}
```

The constructor allocates a TLS index (just once). Then it sets its own address as the value in the TLS slot. The destructor decides whether to commit or abort the transaction and then sets the TLS value to `NULL`, indicating there is no transaction.

`GetCurrent` simply retrieves the value in the TLS slot and casts it to a `Transaction*`. Here is an example code using this class:

```
bool DoWork() {
    // more code
}

void f1() {
    auto tn = Transaction::GetCurrent();
    if (tn)
        tn->AddLog(L"f1 working");

    if (!DoWork()) {
        if (tn)
            tn->AddError(L"Failed in DoWork");
        else
            printf("Failed in DoWork");
    }
}

void do_something() {
    Transaction t;
    f1();
}
```

> This idea, sometimes called *ambient transaction* is used in the .NET Framework with the `TransactionScope` class.

Static TLS

Thread local storage is also available in a simpler form, by using a Microsoft extension keyword on a global or static variable or by using C++ 11 or higher compiler. Let's examine both options.

The Microsoft-specific specifier `__declspec(thread)` can be used to designate a thread local variable like so:

```
__declspec(thread) int counter;
```

The variable counter is now thread-local. Each thread has its own value. Similarly, with C++ 11 or later, this can be done in a cross-platform manner with the `thread_local` keyword like so:

```
thread_local int counter;
```

The result is the same. This TLS is "static" in the sense that it does not need any allocation, and it cannot be destroyed. Internally, the compiler bundles up all the thread-local variables into one chunk and stores the information in the PE in a section named `.tls`. The loader (NTDLL) that reads this information when the process starts up calls `TlsAlloc` to allocate a slot and allocates dynamically for each thread that starts up a memory block that contains all the thread-local variables. This is possible since every user-mode thread starts in an NTDLL-provided function before the "real" function that is passed to `CreateThread` is invoked.

Figure 10-2 shows the TLS data in a PE file where the following line was compiled:

```
thread_local int counter =5;
```

The value "5" can be clearly seen in the binary data of the TLS section. Also, it appears there is more TLS data that is used as part of some Windows DLLs that were not created by the application directly.

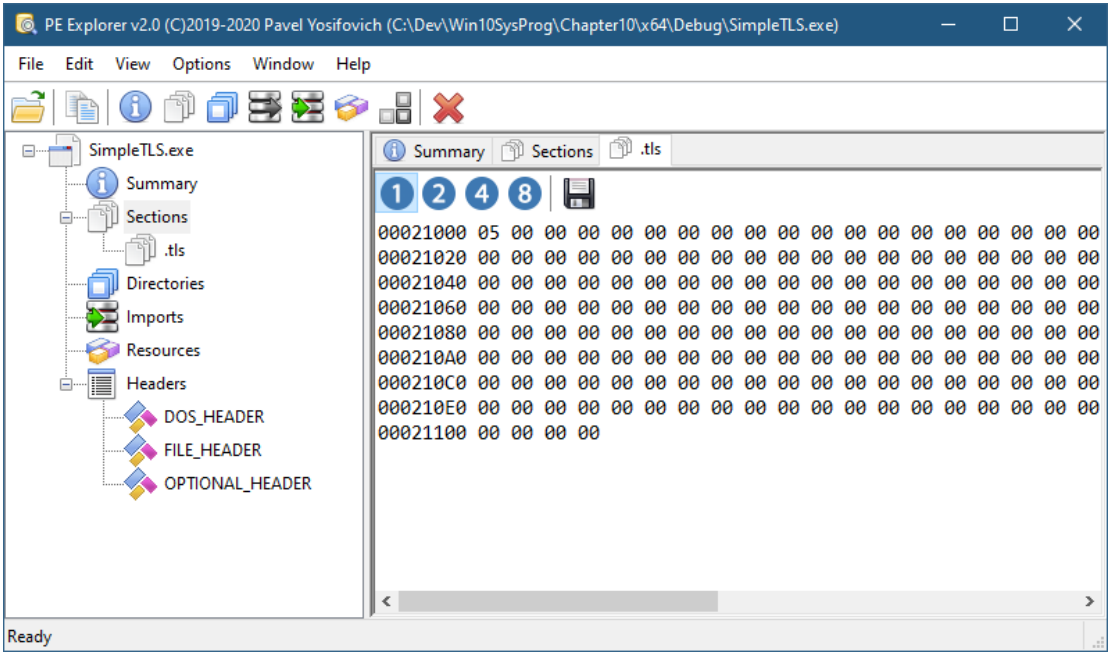


Figure 10-2: TLS in a PE file

Remote Threads

The `CreateThread` function we used numerous times creates a thread in the current process. There are some cases, however, that one process may wish to create a thread in another process. The canonical example for this use is by a debugger. When a forceful breakpoint is required, such as when a user presses a "Break" button, the debugger creates a thread in the target process and points it to a `DebugBreak` function (or a CPU intrinsic that issues a break instruction), causing the process to break and the debugger to be notified.

The functions that allows this are `CreateRemoteThread` and `CreateRemoteThreadEx`:

```
HANDLE WINAPI CreateRemoteThread(
    _In_      HANDLE                 hProcess,
    _In_      LPSECURITY_ATTRIBUTES  lpThreadAttributes,
    _In_      SIZE_T                 dwStackSize,
    _In_      LPTHREAD_START_ROUTINE lpStartAddress,
    _In_      LPVOID                 lpParameter,
    _In_      DWORD                  dwCreationFlags,
    _Out_opt_ LPDWORD                lpThreadId);

HANDLE CreateRemoteThreadEx(
    _In_      HANDLE                       hProcess,
    _In_opt_  LPSECURITY_ATTRIBUTES        lpThreadAttributes,
    _In_      SIZE_T                       dwStackSize,
    _In_      LPTHREAD_START_ROUTINE       lpStartAddress,
    _In_opt_  LPVOID                       lpParameter,
    _In_      DWORD                        dwCreationFlags,
    _In_opt_  LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList,
    _Out_opt_ LPDWORD                      lpThreadId);
```

`CreateRemoteThread` adds just one parameter compared to `CreateThread` - the first - the handle to the target process. This handle must have quite an access mask - `PROCESS_CREATE_THREAD`, `PROCESS_QUERY_INFORMATION`, `PROCESS_VM_OPERATION`, `PROCESS_VM_WRITE`, and `PROCESS_VM_READ`. This makes sense, since creating a thread in another process is a very invasive operation.

> The handle can also be `GetCurrentProcess()`, which makes the function identical to `CreateThread`.

The most interesting parameter is the function pointer itself (`lpStartAddress`). The address of the function is relative to the target process, meaning the code the thread needs to execute should already be there somehow. One idea is to use a function that is guaranteed to be in the target process and in a known address. Windows API functions are generally of this type. Since the Windows subsystem DLLs (*kernel32.dll*, *kernelbase.dll*, *user32.dll*, etc. and certainly *ntdll.dll*) are mapped to the same address in all processes, the address obtained from the calling process can be used in the target process as well.

`CreateRemoteThreadEx` adds another parameter compared to `CreateRemoteThread`: an attribute list. This is the same attribute lists discussed in chapter 3. Some attributes are related to processes (chapter 3), but there are some attributes that are related to threads, and this is the way to specify them. Note that this is just as relevant for local threads as it is for remote threads. Refer back to table 3-9 for the list of attributes, some op which are for threads.

The *Breakin* Application

The *breakin* sample application does a remote break-in into a process using a remote thread calling the `DebugBreak` function, similarly to how a debugger would do it.

> Technically, a function already exists to perform this action - `DebugBreakProcess`.

The first step is to get a process ID from the command line and open a strong-enough handle:

```
int main(int argc, const char* argv[]) {
    if (argc < 2) {
        printf("Usage: breakin <pid>\n");
        return 0;
    }

    int pid = atoi(argv[1]);

    auto hProcess = ::OpenProcess(PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMA\
TION |
        PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE,
        FALSE, pid);
    if (!hProcess)
        return Error("Failed to open process");
```

There is nothing new in the above code. The important part is to request the minimal access mask for `CreateRemoteThread` to succeed.

Next comes the call to `CreateRemoteThread`. We're using the fact that *kernel32.dll* is mapped to the same address in each process, so the address of the `DebugBreak` function is the same in each process. This means we can locate this function in this process and instruct the remote thread to use the same function address:

```
    auto hThread = ::CreateRemoteThread(hProcess, nullptr, 0,
        (LPTHREAD_START_ROUTINE)::GetProcAddress(
            ::GetModuleHandle(L"kernel32"), "DebugBreak"),
        nullptr, 0, nullptr);
    if (!hThread)
        return Error("Failed to create remote thread");

    printf("Remote thread created successfully!\n");

    ::CloseHandle(hThread);
    ::CloseHandle(hProcess);

    return 0;
}
```

`GetModuleHandle` returns the address of a loaded module in this process (*kernel32.dll*), and `GetProcAddress` retrieves a function's address. These functions and others related to DLLs are discussed in detail in chapter 15.

There is yet another hidden assumption in this code, which is about the parameters of a thread's function. A standard thread's function looks like this:

```
DWORD WINAPI ThreadFunction(PVOID param);
```

This means any function we ask the remote thread (or any thread for that matter) to run must have this prototype or something "close enough". In this case, "close enough" works because `DebugBreak` does not accept anything, so we can pass along a `NULL` as the `param` value, and the return type is not going to be used anyway, so that's OK as well. If the function would require more than one parameter, that would be problematic as there would be no easy way to pass these along.

We can test the *breakin* application by running some process (e.g. *notepad*), attaching with some debugger, and letting the process run freely. Then, we can use *breakin* to force a breakpoint. The result should be a breakpoint in the process with the debugger regaining control.

What would happen if we do this with a process that is not being debugged? Try it and find out!

A more useful use for `CreateRemoteThread(Ex)` is injecting a DLL into a target process. Since this requires dealing with virtual memory and DLLs, we'll postpone this example usage until chapter 15.

Thread Enumeration

We've seen in chapter 3 several ways to enumerate the processes running on a system. What about threads? The *tool help* function `CreateToolhelp32Snapshot` offers a flag, `TH32_SNAPTHREAD`, that enumerates all threads in the system.

```
HANDLE CreateToolhelp32Snapshot(
    DWORD dwFlags,
    DWORD th32ProcessID);
```

The snapshot contains all threads in all processes - there is no way to specify a specific process ID. The second parameter to `CreateToolhelp32Snapshot` does include a process ID, but this works only when enumerating modules or heaps.

Once a snapshot is created, you can go over the threads in the snapshot by calling `Thread32First` once, and then iterating by calling `Thread32Next` until it returns false:

```
BOOL Thread32First(
    HANDLE hSnapshot,
    LPTHREADENTRY32 lpte);

BOOL Thread32Next(
    HANDLE hSnapshot,
    LPTHREADENTRY32 lpte);
```

Both functions return information in a `THREADENTRY32` structure defined like so:

```
typedef struct tagTHREADENTRY32 {
    DWORD   dwSize;     // must be set before calls
    DWORD   cntUsage;
    DWORD   th32ThreadID;       // this thread
    DWORD   th32OwnerProcessID; // Process this thread is associated with
    LONG    tpBasePri;          // base priority
    LONG    tpDeltaPri;     // not used
    DWORD   dwFlags;        // not used
} THREADENTRY32;
```

The *thlist* Application

The *thlist* application lists all threads in a particular process or all threads in the system depending on whether a process ID was provided on the command line. The function also lists the image name of the process each thread belongs to.

The heart of the application is a helper function, `EnumThreads` that returns a vector of `ThreadInfo` structures defined like so:

```
struct ThreadInfo {
    DWORD Id;                   // thread ID
    DWORD Pid;                  // process ID
    int Priority;               // thread priority
    FILETIME CreateTime;        // thread create time
    DWORD CPUTime;              // thread CPU time (msec)
    std::wstring ProcessName;   // process image name
};
```

The thread creation time and CPU time is not provided by the `THREADENTRY32` structure. These require opening a handle to the thread, and if successful - obtaining the information. A thread can be opened much like a process by providing its ID and a requested access mask to `OpenThread`:

```
HANDLE OpenThread(
    _In_ DWORD dwDesiredAccess,
    _In_ BOOL bInheritHandle,
    _In_ DWORD dwThreadId);
```

The `EnumThreads` function begins by creating a snapshot for processes and threads:

```
std::vector<ThreadInfo> EnumThreads(int pid) {
    std::vector<ThreadInfo> threads;

    HANDLE hSnapshot = ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS | TH32CS_S\
NAPTHREAD, 0);
    if (hSnapshot == INVALID_HANDLE_VALUE)
        return threads;
```

The snapshot contains processes and threads. First, we enumerate the processes so we can create a mapping between a process ID and its associated information. Then it would be easy to locate a process when iterating over threads:

```
PROCESSENTRY32 pe;
pe.dwSize = sizeof(pe);

std::unordered_map<DWORD, PROCESSENTRY32> processes;
processes.reserve(512);

::Process32First(hSnapshot, &pe);

// skip idle process

while (::Process32Next(hSnapshot, &pe)) {
    processes.insert({ pe.th32ProcessID, pe });
}
```

For each process discovered, an entry is inserted into an `unordered_map<>`, so processes can be looked up faster in the second stage. Process ID 0 (the idle process) is skipped since it's not a real process and is not of particular interest (`Process32First` is called, but the result is not processed).

Now comes thread enumeration itself. The `THREADENTRY32` structure is initialized with the proper size:

```
threads.reserve(4096);

THREADENTRY32 te;
te.dwSize = sizeof(te);
```

Next, thread iteration takes place. For each thread, the corresponding process is looked up in the `unordered_map<>` to locate the process image name:

```
::Thread32First(hSnapshot, &te);

do {
    if (te.th32OwnerProcessID > 0 && (pid == 0 || te.th32OwnerProcessID == pid)\
) {
        ThreadInfo ti;
        ti.Id = te.th32ThreadID;
        ti.Pid = te.th32OwnerProcessID;
        ti.Priority = te.tpBasePri;
        ti.ProcessName = processes[ti.Pid].szExeFile;
```

The `if` statement checks that if a process ID was specified (non zero), only threads in that process are processed; otherwise, all threads are processed.

Next, we need to get some more thread information that is not part of the enumeration:

```
auto hThread = ::OpenThread(THREAD_QUERY_LIMITED_INFORMATION, FALSE, ti.Id);
if (hThread) {
    FILETIME user, kernel, exit;
    ::GetThreadTimes(hThread, &ti.CreateTime, &exit, &kernel, &user);
    ti.CPUTime = DWORD((*(ULONGLONG*)&kernel + *(ULONGLONG*)&user) / 10000000);
    ::CloseHandle(hThread);
}
else {
    ti.CPUTime = 0;
    ti.CreateTime.dwHighDateTime = ti.CreateTime.dwLowDateTime = 0;
}
```

`THREAD_QUERY_LIMITED_INFORMATION` is the needed access mask to retrieve superficial information about a thread, such as its creation and execution times, using `GetThreadTimes`:

```
BOOL GetThreadTimes(
    _In_ HANDLE hThread,
    _Out_ LPFILETIME lpCreationTime,
    _Out_ LPFILETIME lpExitTime,
    _Out_ LPFILETIME lpKernelTime,
    _Out_ LPFILETIME lpUserTime);
```

The function is similar to `GetProcessTimes` we've used before, but works on a thread basis. The creation time and exit times are in the usual 100nsec units from January 1, 1601, UTC, and the kernel and user execution times are measured in 100nsec units. The above code divides the sum of kernel and user times by 10 million, to reduce the number to units of seconds.

All that remains is to add the `ThreadInfo` object to the vector and continue iteration:

```
            threads.push_back(std::move(ti));
        }
    } while (::Thread32Next(hSnapshot, &te));

    ::CloseHandle(hSnapshot);

    return threads;
}
```

The `main` function calls `EnumThreads` and presents the information with various format manipulations:

```
int main(int argc, const char* argv[]) {
    DWORD pid = 0;
    if (argc > 1)
        pid = atoi(argv[1]);

    auto threads = EnumThreads(pid);
    printf("%6s %6s %5s %18s %11s %s\n", "TID", "PID", "Pri", "Started",
        "CPU Time", "Process Name");
    printf("%s\n", std::string(60, '-').c_str());

    for (auto& t : threads) {
        printf("%6d %6d %5d %18ws %11ws %ws\n", t.Id, t.Pid, t.Priority,
            t.CreateTime.dwLowDateTime + t.CreateTime.dwHighDateTime == 0 ?
            L"(Unknown)" : (PCWSTR)CTime(t.CreateTime).Format(L"%x %X"),
            (PCWSTR)CTimeSpan(t.CPUTime).Format(L"%D:%H:%M:%S"),
            t.ProcessName.c_str());
    }

    return 0;
}
```

Running the application without any parameters dumps all threads in the system. Running it with a process ID limits its output to that process:

```
C:\>thlist.exe 11740
   TID    PID   Pri            Started    CPU Time Process Name
-------------------------------------------------------------
 11744  11740     8  03/22/20 12:12:08  0:00:02:06 explorer.exe
 11904  11740     8  03/22/20 12:12:08  0:00:00:27 explorer.exe
 13280  11740     9  03/22/20 12:12:10  0:00:17:14 explorer.exe
 11936  11740     8  03/22/20 12:12:11  0:00:00:27 explorer.exe
 11716  11740     8  03/22/20 12:12:11  0:00:00:32 explorer.exe
 ...
  5080  11740     8  03/25/20 11:14:36  0:00:00:00 explorer.exe
 17064  11740     8  03/25/20 11:14:36  0:00:00:00 explorer.exe
 41084  11740     8  03/25/20 11:14:37  0:00:00:00 explorer.exe
 48916  11740     8  03/25/20 11:14:44  0:00:00:00 explorer.exe
```

> The native APIs for process enumerations described in chapter 3 also provide the capability to enumerate threads in each process.

Caches and Cache Lines

In the early days of microprocessors, the CPUs' speeds and the memory's (RAM) speeds were comparable. Then CPU speeds went up and memory speeds lagged. This leads to a situation where the CPU stalls a lot, waiting for memory to read or write a value. To compensate, a cache was introduced between the CPU and memory, as illustrated in figure 10-3.

Figure 10-3: Cache between CPU and memory

The cache is a fast memory compared to main memory, which allows the CPU to stall less. Naturally, the cache is not nearly as large as main memory, but its existence is essential in today's systems. The importance of the cache cannot be overestimated.

The *SumMatrix* project compares summing a matrix in two ways as shown here:

```
long long SumMatrix1(const Matrix<int>& m) {
    long long sum = 0;
    for (int r = 0; r < m.Rows(); ++r)
        for (int c = 0; c < m.Columns(); ++c)
            sum += m[r][c];

    return sum;
}

long long SumMatrix2(const Matrix<int>& m) {
    long long sum = 0;
    for (int c = 0; c < m.Columns(); ++c)
        for (int r = 0; r < m.Rows(); ++r)
            sum += m[r][c];

    return sum;
}
```

The `Matrix<>` class is a simple wrapper over a one-dimensional array. From an algorithmic perspective, the time it takes to sum the matrix elements in both functions should be the same. After all - the code goes over all the matrix' elements once. But the practical result may be surprising. Here is a run on my machine with various matrix sizes and the time it takes to sum the elements (everything is single-threaded):

```
Type                Size                 Sum         Time (nsec)
-----------------------------------------------------------------
Row major       256 X 256          2147516416          34 usec
Col Major       256 X 256          2147516416          81 usec

Row major       512 X 512         34359869440         130 usec
Col Major       512 X 512         34359869440         796 usec

Row major     1024 X 1024        549756338176         624 usec
Col Major     1024 X 1024        549756338176        3080 usec

Row major     2048 X 2048       8796095119360        2369 usec
Col Major     2048 X 2048       8796095119360       43230 usec

Row major     4096 X 4096     140737496743936        8953 usec
Col Major     4096 X 4096     140737496743936      190985 usec
```

```
Row major      8192 X 8192     2251799847239680       35258 usec
Col Major      8192 X 8192     2251799847239680     1035334 usec

Row major    16384 X 16384    36028797153181696      142603 usec
Col Major    16384 X 16384    36028797153181696     4562040 usec
```

The differences are extreme, and they are due to caches. When a CPU reads data, it doesn't read a single integer or whatever it was instructed to read, but reads an entire cache line (typically 64 bytes) and places it in its internal cache. Then when reading the next integer in memory, no memory access is required because the integer is already present in the cache. This is optimal and the way `SumMatrix1` works - it traverses memory linearly.

`SumMatrix2` on the other hand, reads an integer (along with the rest of the cache line), and the next integer is further away, on a different cache line (for all but the smallest of matrices), which requires reading another cache line, possibly throwing away data that may be needed soon, making things even worse.

Technically, there are 3 cache levels implemented in most CPUs. The closer the cache to the processor, the faster it is and the smaller it is. Figure 10-4 shows a typical cache configuration for a 4-core CPU (with Hyperthreading technology).

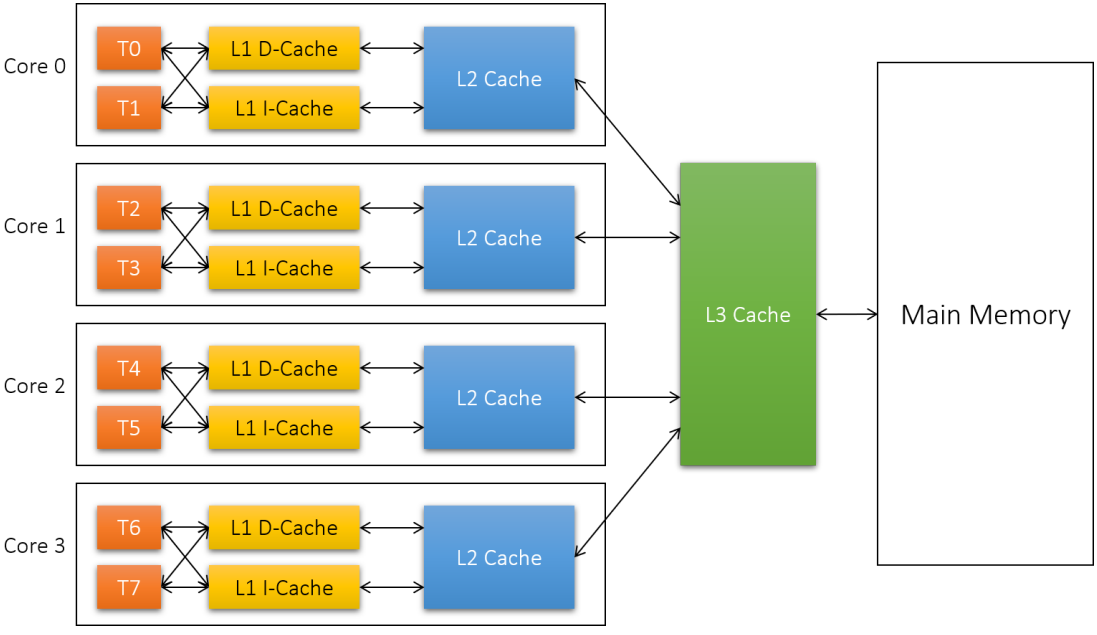


Figure 10-4: Cache levels in CPUs

Level 1 cache is made up of a data cache (D-cache) and instruction cache (I-cache) and is per logical processor. Then there is cache level 2, which is shared by logical processors that are part

of the same core. Finally, level 3 cache is system-wide. The sizes of these caches are rather small, roughly 3 orders of magnitude smaller than main memory. The cache sizes on a system are easily visible in *Task Manager* in the *Performance* / *CPU* tab, as shown in figure 10-5.

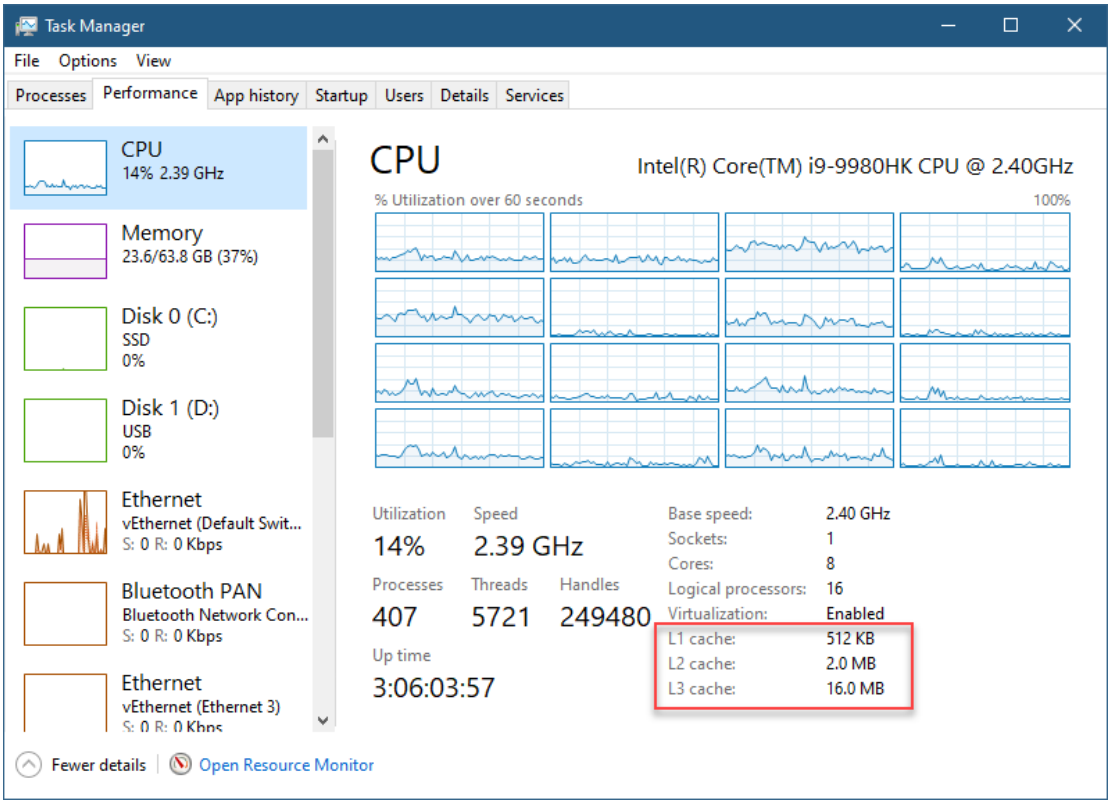


Figure 10-5: Cache sizes in *Task Manager*

In figure 10-5, level 3 cache size is 16 MB (system-wide). Level 2 cache size is 2 MB, but that includes all cores. Since this system has 8 cores, each level 2 cache is really `2MB/8=256KB`. Similarly, level 1 cache size is 512 KB, spread over 16 logical processors makes each cache `512KB/16=32KB`. The bottom line is that cache sizes are small compared to the main memory size (in the gigabytes).

> There is yet another important cache not shown by *Task Manager*, called *Translation Lookaside Buffer* (TLB). This is a CPU cache dedicated to a fast translation of virtual to physical addresses. We'll discuss this cache further in chapter 12.

Let's look at another example where caches and cache lines play an important (even crucial) part.

The *FalseSharing* project demonstrates going over a large array, counting the number of even numbers in the array. This is done with multiple threads - each thread is assigned a contiguous part of the array. The counts themselves are placed in another array, each cell modified by the corresponding thread. Figure 10-6 shows this arrangement with 4 threads.

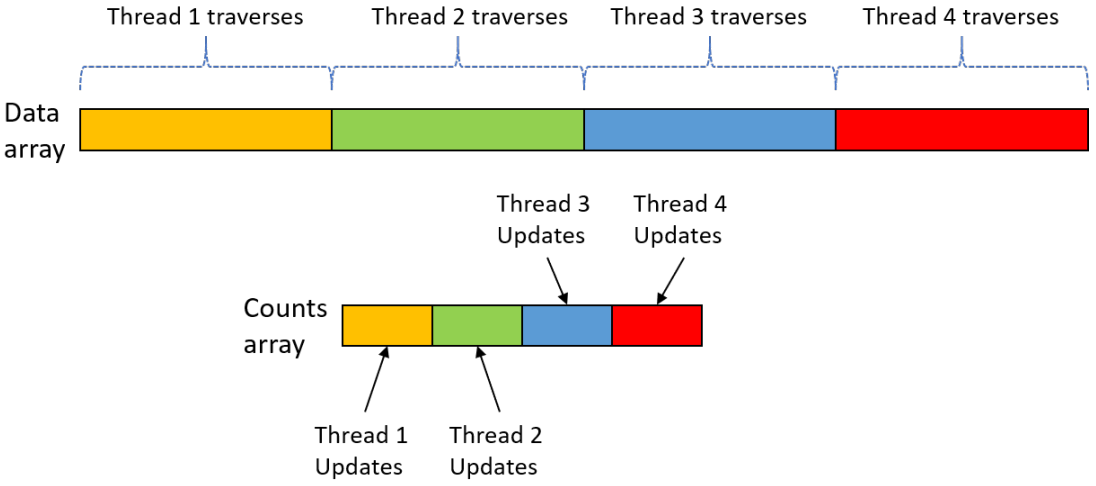


Figure 10-6: The *False Sharing* application

Here is the first version of counting even numbers:

```
using namespace std;

struct ThreadData {
    long long start, end;
    const int* data;
    long long* counters;
};

long long CountEvenNumbers1(const int* data, long long size, int nthreads) {
    auto counters_buffer = make_unique<long long[]>(nthreads);
    auto counters = counters_buffer.get();
    auto tdata = make_unique<ThreadData[]>(nthreads);

    long long chunk = size / nthreads;
    vector<wil::unique_handle> threads;
    vector<HANDLE> handles;

    for (int i = 0; i < nthreads; i++) {
        long long start = i * chunk;
```

```
        long long end = i == nthreads - 1 ? size : ((long long)i + 1) * chunk;
        auto& d = tdata[i];
        d.start = start;
        d.end = end;
        d.counters = counters + i;
        d.data = data;

        wil::unique_handle hThread(::CreateThread(nullptr, 0, [](auto param) ->\
 DWORD {
            auto d = (ThreadData*)param;
            auto start = d->start, end = d->end;
            auto counters = d->counters;
            auto data = d->data;

            for (; start < end; ++start)
                if (data[start] % 2 == 0)
                    ++(*counters);
            return 0;
            }, tdata.get() + i, 0, nullptr));

        handles.push_back(hThread.get());
        threads.push_back(move(hThread));
    }

    ::WaitForMultipleObjects(nthreads, handles.data(), TRUE, INFINITE);

    long long sum = 0;
    for (int i = 0; i < nthreads; i++)
        sum += counters[i];
    return sum;
}
```

`CountEvenNumbers1` accepts a pointer to the data, its size and the number of threads to use for partitioning the data array. It then allocates `counters_buffer`, which is the buffer where each thread should increment its own designated cell. Then an array of `ThreadData` is allocated (`tdata`), so that each thread can receive its own parameters for iteration. Next, two `vector` objects are created to hold the thread handles. One holds them as `wil::unique_handle` so they are automatically closed when the object goes out of scope. The second `vector` (`handles`) holds the handles as plain `HANDLE` instances, so it can be passed directly to `WaitForMultipleObjects`. The chunk size is calculated as well.

Now starts the loop that prepares the data for each thread and calls `CreateThread` to start threads

rolling. The chunking is done almost identically as was demonstrated in chapter 5 in the primes calculation application. Let's take a closer look at the thread's loop:

```
for (; start < end; ++start)
    if (data[start] % 2 == 0)
        ++(*counters);
```

For each even number, the contents of the `counters` pointer is incremented by one. Note that there is no data race here - each thread has its own cell, so the final results should be correct. The problem with this code is the fact that when a single count is written by some thread, a full cache line is written, which invalidates any caches looking at this memory on other processors, forcing them to refresh their cache by reading from main memory again, which we already know is slow. This situation is referred to as *false sharing*.

The alternative is not to write over cells that are sharing cache lines with other threads, at least not very often. Here is the code inside the thread block in the function `CountEvenNumbers2`, which is identical in all other respects:

```
auto d = (ThreadData*)param;
auto start = d->start, end = d->end;
auto data = d->data;
size_t count = 0;

for (; start < end; ++start)
    if (data[start] % 2 == 0)
        count++;
*(d->counters) = count;
return 0;
}, tdata.get() + i, 0, nullptr));
```

The major difference is keeping the count in a local variable (`count`) and only writing to the cell in the result array just once - when the loop is done. Since `count` is on the thread's stack, and stacks are at least 4 KB in size, they cannot possibly be on the same cache line as other `count` variables in other threads. This improves performance considerably. Of course, generally using a local variable is likely to be faster than accessing memory indirectly because it's easier for the compiler to cache this variable in a register. But the real impact is the avoidance of sharing a cache line between threads.

The `main` function tests both implementations with various numbers of threads, like so:

```
const long long size = (1LL << 32) - 1;     // just a large number
cout << "Initializing data..." << endl;

auto data = make_unique<int[]>(size);
for (long long i = 0; i < size; i++)
    data[i] = (unsigned)i + 1;

auto processors = min(8, ::GetActiveProcessorCount(ALL_PROCESSOR_GROUPS));

cout << "Option 1" << endl;
for (WORD i = 1; i <= processors; ++i) {
    auto start = ::GetTickCount64();
    auto count = CountEvenNumbers1(data.get(), size, i);
    auto end = ::GetTickCount64();
    auto duration = end - start;
    cout << setw(2) << i << " threads " << "count: " << count << " time: "
        << duration << " msec" << endl;
}

cout << endl << "Option 2" << endl;
for (WORD i = 1; i <= processors; ++i) {
    auto start = ::GetTickCount64();
    auto count = CountEvenNumbers2(data.get(), size, i);
    auto end = ::GetTickCount64();
    auto duration = end - start;
    cout << setw(2) << i << " threads " << "count: " << count << " time: "
        << duration << " msec" << endl;
}
```

You'll have to run this as a 64-bit process since it allocates about 16 GB of memory.

The function uses up to 8 threads (just to limit running time). In a well-behaved program, we would expect almost a linear improvement, because no synchronization is used or needed. But, in the false sharing variation (the first), the improvement by adding threads is less noticeable, and makes things worse with some thread values:

```
Initializing data...
Option 1
 1 threads count: 2147483647 time: 4843 msec
 2 threads count: 2147483647 time: 3391 msec
 3 threads count: 2147483647 time: 2468 msec
 4 threads count: 2147483647 time: 2125 msec
 5 threads count: 2147483647 time: 2453 msec
 6 threads count: 2147483647 time: 1906 msec
 7 threads count: 2147483647 time: 2109 msec
 8 threads count: 2147483647 time: 2532 msec

Option 2
 1 threads count: 2147483647 time: 4046 msec
 2 threads count: 2147483647 time: 2313 msec
 3 threads count: 2147483647 time: 1625 msec
 4 threads count: 2147483647 time: 1328 msec
 5 threads count: 2147483647 time: 1062 msec
 6 threads count: 2147483647 time: 953 msec
 7 threads count: 2147483647 time: 859 msec
 8 threads count: 2147483647 time: 855 msec
```

Notice in the false sharing option (1), in some cases more threads degrade performance. In the optimal case (2), there is constant improvement.

Wait Chain Traversal

In chapters 7 and 8, we've looked at various synchronization primitives used to synchronized thread activity. One caveat of synchronization is the possibility of deadlocks. If a deadlock does occur in a non-trivial application, it's not easy to discover where the deadlock is. There are some techniques used with debuggers that can help locate such deadlocks. In this section, we'll look at a programmatic technique that can identify a wide range of deadlocks, called *Wait Chain Traversal* (WCT).

The WCT API provides the ability to traverse a *wait chain* starting from a thread of interest. A wait chain contains an alternating sequence of threads and objects. Each thread in the stream waits for an object that follows it, which is owned by the following thread in the chain, and so on. For example, a thread may be an owner of a critical section, and wait for a mutex, that is owned by another thread - this is an example of a wait chain.

Wait chain analysis is able to track chains involving the following objects:

- Critical sections

- Mutexes (including across processes)
- *Asynchronous Local Procedure Calls* (ALPC) - the internal inter-process communication mechanism used by Windows components
- `SendMessage` - The `SendMessage` API is synchronous, and if called from a thread that is not the window's owner causes the thread to block
- Wait operations on threads and processes
- *Component Object Model* (COM) cross-apartment calls (see chapter 18 for more on COM)
- Sockets and *Simple Message Block* (SMB) operations

To start a wait chain analysis, a handle to a WCT session has to be opened with `OpenThreadWaitChainSession`:

```
HWCT OpenThreadWaitChainSession (
    _In_ DWORD Flags,
    _In_opt_ PWAITCHAINCALLBACK callback);
```

The function opens a session for WCT and specifies whether the session should synchronous or asynchronous. Specifying zero for `Flags` sets a synchronous session. This means the thread that performs the analysis is blocked until the analysis is complete. Specifying `WCT_ASYNC_OPEN_FLAG` (1) indicates an asynchronous session, in which case the `callback` parameter should point to a function that is invoked when the analysis is complete. If successful, `OpenThreadWaitChainSession` returns an opaque handle to the WCT session. Otherwise, `NULL` is returned.

We'll work with the simpler synchronous session, and describe the differences for asynchronous sessions later.

Once a WCT handle is in place, a wait chain analysis is invoked for a specific thread with `GetThreadWaitChain`:

```
BOOL GetThreadWaitChain (
    _In_ HWCT WctHandle,
    _In_opt_ DWORD_PTR Context,
    _In_ DWORD Flags,
    _In_ DWORD ThreadId,
    _Inout_ LPDWORD NodeCount,
    _Out_writes_(*NodeCount) PWAITCHAIN_NODE_INFO NodeInfoArray,
    _Out_ LPBOOL IsCycle);
```

`WctHandle` is the handle received from `OpenThreadWaitChainSession`. `Context` is an optional value that is passed as-is to the callback provided to `OpenThreadWaitChainSession` in case of an asynchronous session. `Flags` indicates which out-of-process should be considered, as described in table 10-1.

Table 10-1: Flags for `GetThreadWaitChain`

Flag	Description
`WCT_OUT_OF_PROC_FLAG` (1)	Without this flag, no out-of-process analysis is attempted
`WCT_OUT_OF_PROC_COM_FLAG` (2)	Required for COM analysis
`WCT_OUT_OF_PROC_CS_FLAG` (4)	Required for critical section analysis
`WCT_NETWORK_IO_FLAG` (8)	Required for sockets/SMB analysis
`WCTP_GETINFO_ALL_FLAGS`	Combines all the previous flags

`ThreadId` is the thread from which to begin wait chain analysis. If the thread is a member of a process with a higher integrity level (see chapter 16 for more information), the analysis may fail. Running with admin rights and enabling the Debug privilege can help with getting access to such processes.

`NodeCount` points to the number of analysis nodes the function is willing to take. On return, it specifies how many actual nodes were written. The maximum analysis depth, which is the maximum nodes that can be returned is defined by `WCT_MAX_NODE_COUNT` (currently defined as 16). `NodeInfoArray` is the output array of node objects. Finally, the last output parameter, `IsCycle`, indicates whether there is a cycle in the analysis, returning `TRUE` if there is a deadlock.

Each analysis node is of type `WAITCHAIN_NODE_INFO`, defined like so:

```
typedef struct _WAITCHAIN_NODE_INFO {
    WCT_OBJECT_TYPE ObjectType;
    WCT_OBJECT_STATUS ObjectStatus;

    union {
        struct {
            WCHAR ObjectName[WCT_OBJNAME_LENGTH];
            LARGE_INTEGER Timeout;     // Not implemented
            BOOL Alertable;            // Not implemented
        } LockObject;

        struct {
            DWORD ProcessId;
            DWORD ThreadId;
            DWORD WaitTime;
            DWORD ContextSwitches;
        } ThreadObject;
    };
```

```
} WAITCHAIN_NODE_INFO, *PWAITCHAIN_NODE_INFO;
```

Each node represents one object in the chain. A chain always starts with a thread object and then is followed (if the thread is waiting on something) by an object, then by a thread (if that thread owns the object), etc.

The `ObjectType` and `ObjectStatus` are always valid. Here are their definitions:

```
typedef enum _WCT_OBJECT_TYPE {
    WctCriticalSectionType = 1,
    WctSendMessageType,
    WctMutexType,
    WctAlpcType,
    WctComType,
    WctThreadWaitType,
    WctProcessWaitType,
    WctThreadType,
    WctComActivationType,
    WctUnknownType,
    WctSocketIoType,
    WctSmbIoType,
    WctMaxType
} WCT_OBJECT_TYPE;

typedef enum _WCT_OBJECT_STATUS {
    WctStatusNoAccess = 1,          // ACCESS_DENIED for this object
    WctStatusRunning,               // Thread status
    WctStatusBlocked,               // Thread status
    WctStatusPidOnly,               // Thread status
    WctStatusPidOnlyRpcss,          // Thread status
    WctStatusOwned,                 // Dispatcher object status
    WctStatusNotOwned,              // Dispatcher object status
    WctStatusAbandoned,             // Dispatcher object status
    WctStatusUnknown,               // All objects
    WctStatusError,                 // All objects
    WctStatusMax
} WCT_OBJECT_STATUS;
```

`ObjectType` indicates what object is represented by the current node. If it's a thread (`WctThreadType`), then the `ThreadObject` part of the anonymous union provides more infor-

mation: thread ID, the process ID of this thread, the time it spent waiting and the number of context switches it incurred.

If the object type is not a thread, it can be a lock object (such as a critical section or a mutex), or something else (such as a send message call or COM). In the case of a lock object, the `LockObject` part of the union is valid, and provides the object's name (if any).

`ObjectStatus` indicates the status of the object described by the node as can be seen in the above `WCT_OBJECT_STATUS` enum definition.

All that's left to do at this point is to go over the returned array of nodes, performing some form of analysis with the information.

Finally, when WCT is no longer needed, the session must be closed with `CloseThreadWaitChainSession`:

```
VOID CloseThreadWaitChainSession(_In_ HWCT WctHandle);
```

The Deadlock Detector Application

With the WCT information in hand, it's not too difficult to build a tool that can identify deadlocks. The *DeadlockDetector* project does just that. Figure 10-7 shows the application's window when launched.

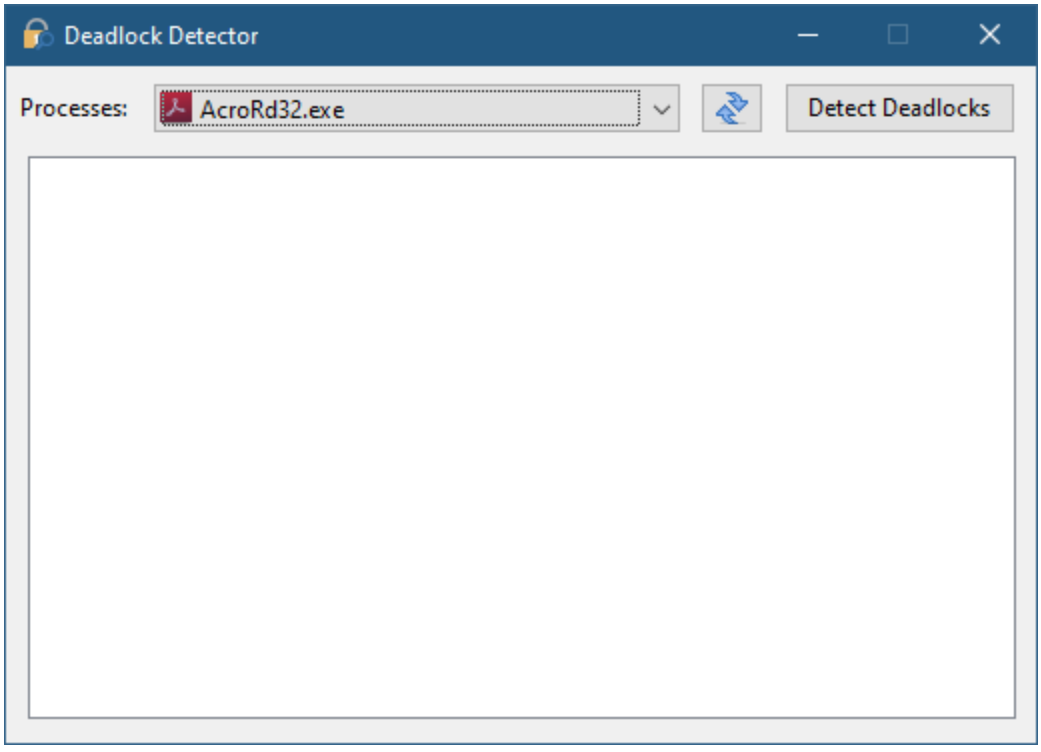


Figure 10-7: *Deadlock Detector* at launch

The *Processes* Combobox allows selecting a process for analysis. Clicking *Detect Deadlocks* enumerates all threads in the selected process, and then perform wait chain analysis for each thread in the process, displaying the result in a tree view, where each root node is a thread. Figure 10-8 shows the application detecting deadlocks with one of the sample applications, *SimpleDeadlock1*.

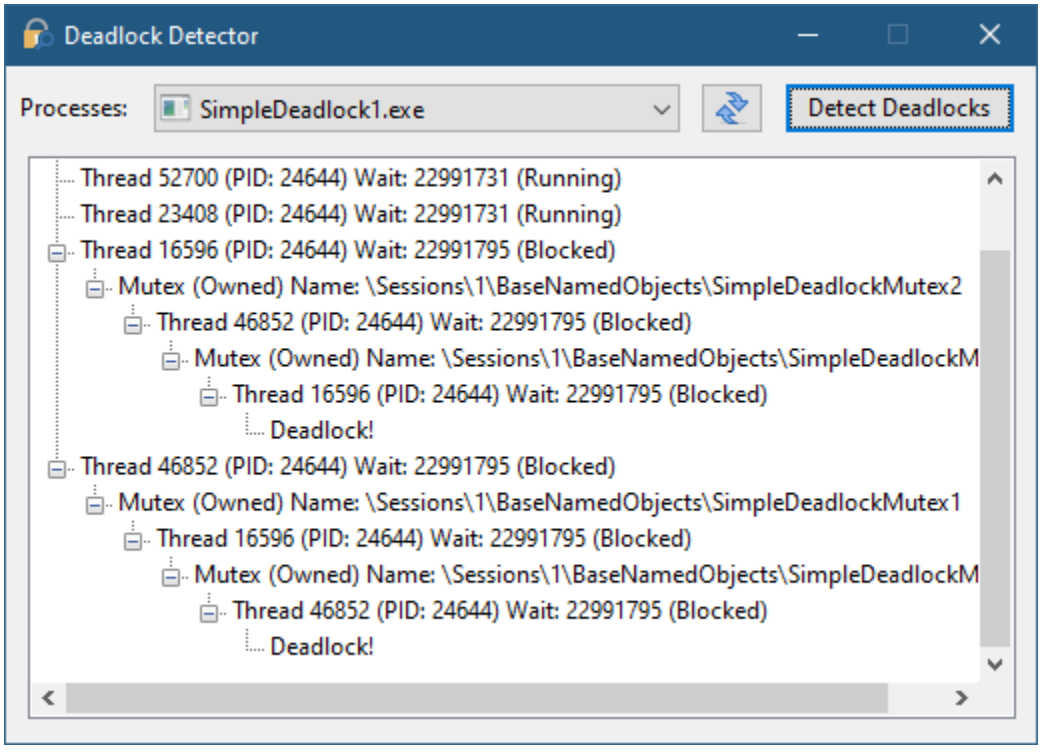


Figure 10-8: *Deadlock Detector* detects mutex deadlock

The application starts by enumerating processes with the *Toolhelp* functions, as we've done numerous times, so I will not repeat the code here. The Combobox is then populated with the results.

Once the user clicks *Detect Deadlocks*, the message handler starts by opening a WCT session for synchronous analysis:

```
LRESULT CMainDlg::OnDetect(WORD, WORD wID, HWND, BOOL&) {
    auto hWct = ::OpenThreadWaitChainSession(0, nullptr);
    if (hWct == nullptr) {
        AtlMessageBox(*this, L"Failed to open WCT session", IDR_MAINFRAME, MB_I\
CONERROR);
        return 0;
    }
```

Then, the selected process ID is extracted from the selected item in the Combobox, and a call is made to enumerate the threads in that process:

```
auto pid = (DWORD)m_ProcCombo.GetItemData(m_ProcCombo.GetCurSel());
auto threads = EnumThreads(pid);
```

The thread enumeration function returns a `vector<DWORD>`, a vector of all thread IDs in the process. The code used for thread enumeration is very similar to the code used in the section "Thread Enumeration", earlier in this chapter.

At this point, we need to start a loop over all thread IDs, and perform the analysis for each thread:

```
    m_Tree.DeleteAllItems();

    int failures = 0;
    for (auto& tid : threads) {
        if (!DoWaitChain(hWct, tid))
            failures++;
    }
    if (failures == threads.size()) {
        AtlMessageBox(*this, L"Failed to analyze wait chain. (try running eleva\
ted)",
            IDR_MAINFRAME, MB_ICONEXCLAMATION);
    }
    ::CloseThreadWaitChainSession(hWct);
    return 0;
}
```

If all threads fail analysis, it means the target process is not accessible, and an error message is displayed to that effect. The `DoWaitChain` function initiates the analysis for a specific thread ID:

```
bool CMainDlg::DoWaitChain(HWCT hWct, DWORD tid) {
    WAITCHAIN_NODE_INFO nodes[WCT_MAX_NODE_COUNT];
    DWORD nodeCount = WCT_MAX_NODE_COUNT;
    BOOL cycle;
    auto success = ::GetThreadWaitChain(hWct, 0, WCTP_GETINFO_ALL_FLAGS, tid,
        &nodeCount, nodes, &cycle);
    if(success) {
        ParseThreadNodes(nodes, nodeCount, cycle);
    }
    return success;
}
```

The function allocates the maximum-sized array for nodes on the stack, and then initializes `nodeCount` to that maximum. Next, `GetThreadWaitChain` is invoked to do the actual analysis.

if successful, the returned node chain is processed by calling `ParseThreadNodes`. Why might `GetThreadWaitChain` fail? Apart from the already mentioned reason (access denied), it's also possible that the thread in question has exited, since thread enumeration is based on a snapshot of currently executing threads, and it is possible for some of the threads to terminate in the meantime. Of course, the opposite could be true as well - new threads may have been created in the process that are not currently being analyzed. This is usually not a big deal, since new threads are less likely to cause issues. In any case, the analysis can be repeated, with threads re-enumerated.

`ParseThreadNodes` starts by defining textual representations for the object types and status types:

```
void CMainDlg::ParseThreadNodes(const WAITCHAIN_NODE_INFO* nodes, DWORD count,
    bool cycle) {
    static PCWSTR objectTypes[] = {
        L"Critical Section",
        L"Send Message",
        L"Mutex",
        L"ALPC",
        L"COM",
        L"Thread Wait",
        L"Process Wait",
        L"Thread",
        L"COM Activation",
        L"Unknown",
        L"Socket",
        L"SMB",
    };

    static PCWSTR statusTypes[] = {
        L"No Access",
        L"Running",
        L"Blocked",
        L"PID only",
        L"PID only RPCSS",
        L"Owned",
        L"Not Owned",
        L"Abandoned",
        L"Unknown",
        L"Error"
    };
```

Then, a loop over the actual nodes returned starts, processing each object, and adding its information to the tree as child nodes. A `switch` statement is used to distinguish between three object types: thread, lock object, and all the rest. For each one the available information is extracted and placed in the tree:

```
HTREEITEM hCurrentNode = TVI_ROOT;
CString text;

for (DWORD i = 0; i < count; i++) {
    auto& node = nodes[i];
    auto type = node.ObjectType;
    auto status = node.ObjectStatus;

    switch (type) {
        case WctThreadType:
            text.Format(L"Thread %u (PID: %u) Wait: %u (%s)",
                node.ThreadObject.ThreadId, node.ThreadObject.ProcessId,
                node.ThreadObject.WaitTime, statusTypes[status - 1]);
            break;

        case WctCriticalSectionType:
        case WctMutexType:
        case WctThreadWaitType:
        case WctProcessWaitType:
            // waitable objects
            text.Format(L"%s (%s) Name: %s",
                objectTypes[type - 1], statusTypes[status - 1],
                node.LockObject.ObjectName);
            break;

        default:
            // other objects
            text.Format(L"%s (%s)", objectTypes[type - 1],
            statusTypes[node.ObjectStatus - 1]);
            break;
    }
    auto hOld = hCurrentNode;
    hCurrentNode = m_Tree.InsertItem(text, hCurrentNode, TVI_LAST);
    m_Tree.Expand(hOld, TVE_EXPAND);
}
```

Lastly, if there is a cycle (i.e. deadlock), another leaf node is added with the text “Deadlock!”:

```
if (cycle) {
    m_Tree.InsertItem(L"Deadlock!", hCurrentNode, TVI_LAST);
    m_Tree.Expand(hCurrentNode, TVE_EXPAND);
}
```

There is one final detail worth mentioning. For COM analysis, a special registration is required so that COM infrastructure is connected to WCT. The following piece of code is used in `OnInitDialog` to perform that connection:

```
auto comLib = ::GetModuleHandle(L"ole32");
if (comLib) {
    ::RegisterWaitChainCOMCallback(
        (PCOGETCALLSTATE)::GetProcAddress(comLib, "CoGetCallState"),
        (PCOGETACTIVATIONSTATE)::GetProcAddress(comLib, "CoGetActivationState")\
);
}
```

Asynchronous WCT Sessions

If the session is configured as asynchronous, the callback function provided to `OpenThreadWaitChainSession` must have the following prototype:

```
typedef VOID (CALLBACK *PWAITCHAINCALLBACK) (
    HWCT WctHandle,
    DWORD_PTR Context,
    DWORD CallbackStatus,
    LPDWORD NodeCount,
    PWAITCHAIN_NODE_INFO NodeInfoArray,
    LPBOOL IsCycle);
```

The call to `OpenThreadWaitChainSession` always returns `FALSE`, but of `GetLastError` returns `ERROR_IO_PENDING`, this means the call was successful in initiating the analysis on a WCT-managed thread. Once the analysis is complete, the callback is invoked with the results. Most of the arguments are similar to those for `OpenThreadWaitChainSession`. except `CallbackStatus`, which indicates whether the analysis succeeded or the reason for the failure. `ERROR_SUCCESS` (0) means success, and there are a few reasons for failure. The most common failure is `ERROR_ACCESS_DENIED`, which as mentioned earlier may be averted by running with admin rights and enabling the Debug privilege.

> The *DeadlockDetector* project contains a function to enable the Debug privilege in *DeadlockDetector.cpp* (such a function was also used in chapter 3).

User Mode Scheduling

Chapter 6 discussed scheduling in detail. The kernel scheduler is responsible for determining which thread should run on which processor, and make the context switches when needed. In some extreme cases, this is not as efficient as it can be. It would be advantageous in some scenarios to be able to control scheduling from user-mode rather than kernel mode. These decisions should not require a switch from user mode to kernel mode, as these switches are not cheap.

In the past (and still supported), Windows provided *fibers*, which attempted to provide a user-mode scheduling mechanism. However, fibers were not recognized by the kernel, which caused a number of issues, such as Thread Local Storage not properly propagated, Thread Environment Block structures not aligned with the currently executing fiber and more. Fibers should not be used today, and so they are not described in this book.

Starting with Windows 7 and Windows 2008 R2, Windows supports an alternative mechanism called *User Mode Scheduling* (UMS), where a user-mode thread becomes a scheduler of sorts and can schedule threads from user mode without the need to have a user-mode/kernel-mode transition. The mechanism is known to the kernel, so the drawbacks of fibers are not present with UMS, since real threads are used rather than fibers sharing a thread.

Unfortunately, crafting a real system that uses UMS is not trivial, to say the least, and so a deep description of UMS is not attempted in this book. Rather, Microsoft provided (since 2010) a library called *Concurrency Runtime*, abbreviated *Concrt* and pronounced "concert", that uses UMS behind the covers to provide efficient use of threads when concurrent execution is required.

The *Primes Counter* application from chapter 3 is used as an example. In that application, we created a number of threads (that was a parameter to the application), and split the work of calculating prime numbers between the threads, allowing each one to count the number of threads in its chunk, and then finally summing the results from all threads.

One of the issues we faced is the fact that it was difficult to partition the work fairly so that each thread has roughly the same workload. Otherwise, threads that are done early cause a CPU to be idle. We tried to compensate by having more threads, but that has its cost too in terms of memory and context switches.

In some cases, the concurrency runtime provides nice solutions with very simple code compared to the hardship of partitioning, thread creation, and management, that was done in the application.

The application from chapter 3 is also available in the projects for this chapter, with the addition of doing the same work of counting prime numbers in a range, but this time using *concrt*.

The first thing required is an `#include`:

```
#include <ppl.h>
```

ppl.h provides the convenience functions, one of which we'll use in a moment. It includes the "real" workhorse of *concrt*, *concrt.h*.

In the `main` function, after the existing calculation code is done, we use *concrt* like so:

```
count = 0;
concurrency::parallel_for(from, to + 1, [&count](int n) {
    if (IsPrime(n))
        ::InterlockedIncrement((unsigned*)&count);
    });
auto end = ::GetTickCount64();
printf("Using concrt: Primes: %d, Elapsed: %u msec\n", count, (ULONG)(end - sta\
rt));
```

The `parallel_for` function does what it implies: a `for` loop, parallelized automatically. You just specify the initial value and the final value plus one, and a function to run for each iteration (here provided as a lambda). This code runs concurrently in some way. Note there is no indication for how many threads to create or how to manage them. The only caveat here is the required synchronization on the `count` shared variable. In the original partitioning scheme, this was not needed, as each thread was incrementing its own counter.

Here is an example run on my system:

```
C:\>PrimesCounter.exe 3 20000000 16
Thread 1 created (3 to 1250001). TID=42576
Thread 2 created (1250002 to 2500000). TID=30636
Thread 3 created (2500001 to 3749999). TID=16944
...
Thread 15 created (17499989 to 18749987). TID=32580
Thread 16 created (18749988 to 20000000). TID=55440
Thread  1 Count:   96468. Execution time:  515 msec
Thread  2 Count:   86603. Execution time:  796 msec
...
Thread 15 Count:   74745. Execution time: 1906 msec
Thread 16 Count:   74475. Execution time: 1906 msec
Total primes: 1270606. Elapsed: 1985 msec
Using concrt: Primes: 1270606, Elapsed: 1640 msec
```

It doesn't matter how many threads I throw at the problem, the *concrt* version always outperforms the manual partitioning. And it does so with very few lines of code, and much more efficiently - it never creates more threads than the number of logical processors on the system.

SQL Server uses UMS to improve its performance since it's a highly multithreaded server application.

Init Once Initialization

One common pattern in many applications is having a singleton object. In some views, it's an *anti-pattern*, but this debate is out of scope for this book. The fact is, singletons are useful and sometimes necessary. One common requirement for a singleton is to be initialized just once. In a multithreaded application, multiple threads may access the singleton at the same time initially, but the singleton must be initialized just once. How can that be achieved?

There are several well-known algorithms that, if properly implemented, can get the job done. This is not a general algorithm book, so these are not described here. The Windows API however, provides a built-in way to call a function with the guarantee that it is called just once.

In C++ 11 and later, static variables in functions are guaranteed to be initialized just once. Also, C++ 11 has the `std::call_once` function, that does the same thing dynamically.

The *One-Time Initializing* API is available starting from Windows 8 and Server 2012. The simplest version is synchronous initialization, which is described here. For the asynchronous version, consult the official documentation.

A variable of type `INIT_ONCE` is used to control on-time initialization. It must be initialized (pun not intended) in a static manner - global or static variable, so that *its* initialization is guaranteed to be just once. The simplest way to initialize it is by setting its value to `INIT_ONCE_STATIC_INIT` like so:

```
INIT_ONCE init = INIT_ONCE_STATIC_INIT;
```

`INIT_ONCE` is just a wrapper for an opaque pointer. The alternative is to initialize it with `InitOnceInitialize`:

```
VOID InitOnceInitialize(_Out_ PINIT_ONCE InitOnce);
```

Then, when the initialization is needed call `InitOnceExecuteOnce`:

```
BOOL InitOnceExecuteOnce(
    _Inout_ PINIT_ONCE InitOnce,
    _In_ __callback PINIT_ONCE_FN InitFn,
    _Inout_opt_ PVOID Parameter,
    _Outptr_opt_result_maybenull_ LPVOID* Context);
```

The `InitOnce` parameter is the one initialized previously, controlling this initialization. `InitFn` is the function that is guaranteed to be called just once. `Parameter` is an optional context value to be passed into the initialization function, and `Context` is an optional result from the initialization function. The function itself must have the following prototype:

```
BOOL (WINAPI *PINIT_ONCE_FN) (
    _Inout_ PINIT_ONCE InitOnce,
    _Inout_opt_ PVOID Parameter,
    _Outptr_opt_result_maybenull_ PVOID *Context);
```

if the callback returns `TRUE`, the initialization is considered successful. Otherwise, it's considered failed, and `FALSE` returns to `InitOnceExecuteOnce`. `Parameter` is the value passed in from `InitOnceExecuteOnce`. The context can return some value, but it must have the rightmost `INIT_ONCE_CTX_RESERVED_BITS` (2) bits zeroed, which means that if it's an address, it must be 4-bytes aligned.

Debugging Multithreaded Applications

Writing correct and efficient multithreaded applications is hard. Once bugs creep in - and they will - debugging becomes important, and is much more difficult than single-threaded application. Covering all aspects of debugging is beyond the scope of this book. Instead, I would like to mention some tips and helpers available in Visual Studio that could be useful, especially for multithreaded applications.

> Debugging is a big topic, and Visual Studio is not the only debugger in town. In fact, in production environments, other low-level and low-footprint debuggers, such as *WinDbg* are used. This section is only about Visual Studio debugging, typically performed on the developer's workstation.

Breakpoints

When a breakpoint is set in a multithreaded part of the code, any thread triggers it, and all threads are suspended as a result. Performing debugger step operations resume *all* threads, not just the thread you may be interested in isolating. One way to isolate a thread is to freeze all other threads using the *Threads* window, as shown in figure 10-9.

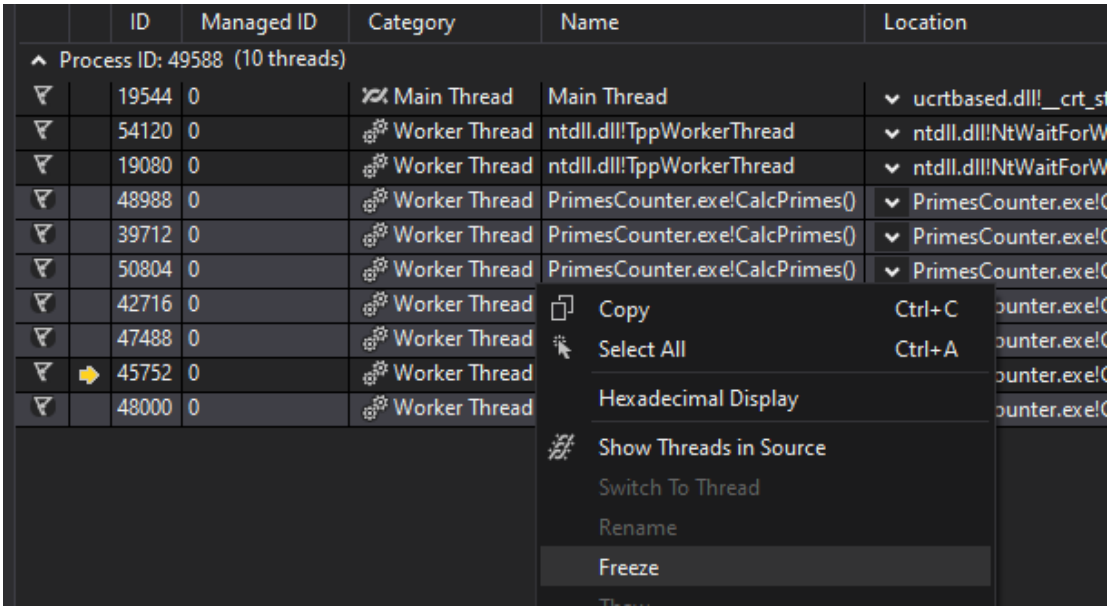


Figure 10-9: Context menu options for the *Threads* debugger window

The *Show Threads in Source* option shown in figure 10-9, adds a thread icon in the source code where one or more threads are when a breakpoint hits.

Breakpoints can be conditional. One of the possible conditions is a thread ID or thread name, allowing you to break only on a specific thread (or threads) of interest (figure 10-10).

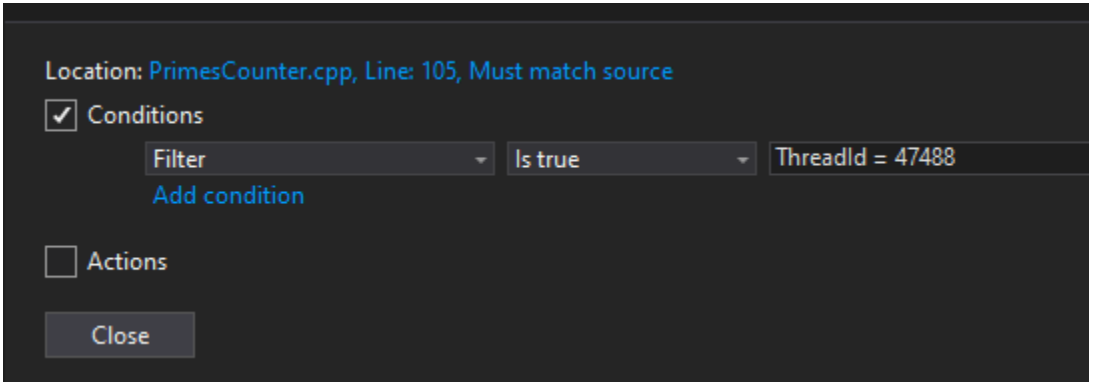


Figure 10-10: Breakpoint thread filter

Parallel Stacks

The *Parallel Stacks* window (accessible from the *Debug* / *Windows* / *Parallel Stacks* menu) shows a graphical view of how threads are spawned from other threads, giving a nice visual

representation of all threads running in the process (figure 10-11).

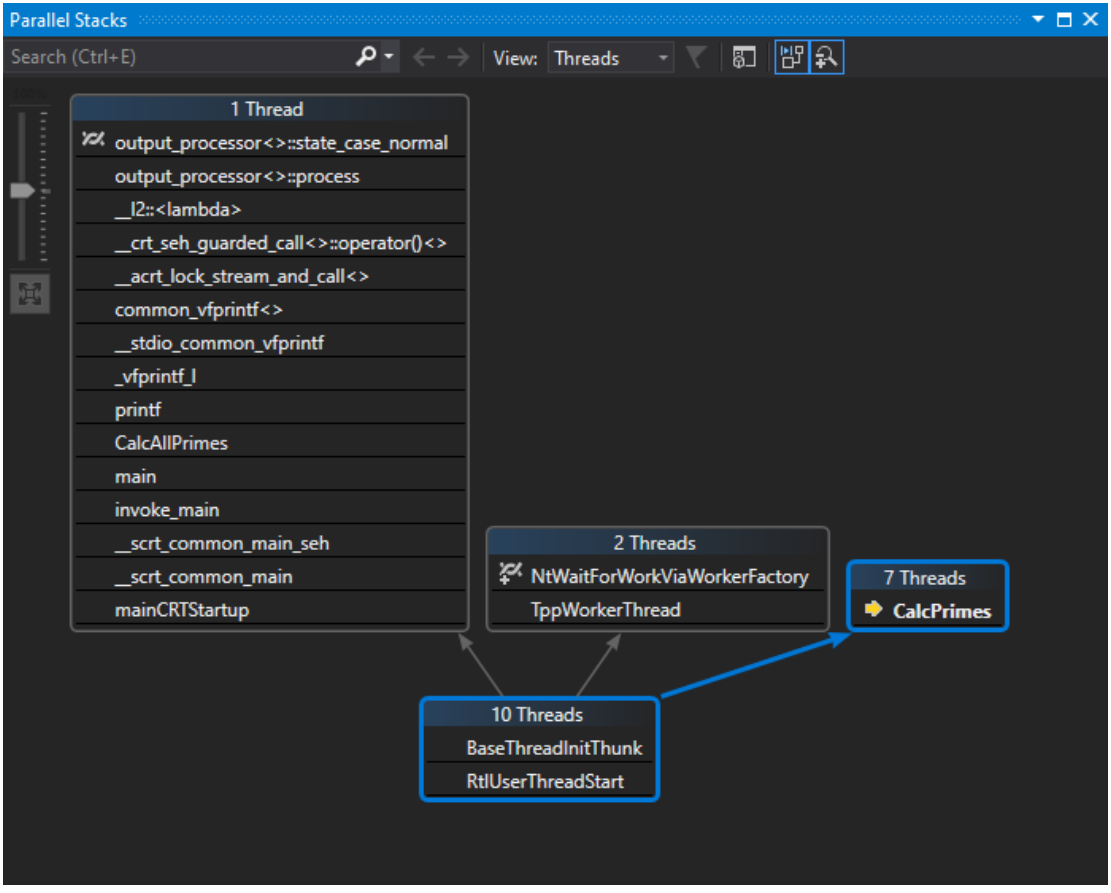


Figure 10-11: *Parallel Stacks* window

Parallel Watch

The *Parallel Watch* window, accessible in a similar manner as *Parallel Stacks*, shows a variable or expression of choice that is used by multiple threads running the same code, each having its own copy of the variable (figure 10-12).

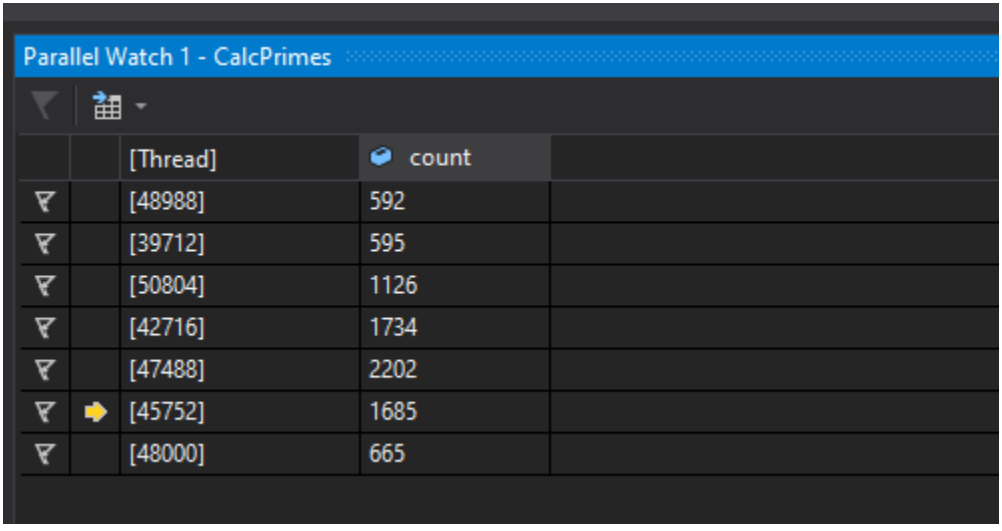


Figure 10-12: *Parallel Watch* window

Thread Names

Chapter 5 described thread names, available in Windows 10 with the `SetThreadDescription` function. A thread's name can also be directly changed in the *Threads* window during debugging, making it easier to follow specific threads. The names get reset once the debugging session is over, so it's better to call `SetThreadDescription` on well-known threads in code so that their names are ready when starting debugging.

Exercises

1. Create a console application that calculates the Mandelbrot set with *concrt*. Compare your results with the same exercise from chapter 5.

Summary

In this chapter, various threading-related topics were presented. In the next chapter, we'll leave threads behind, and look at file and device I/O operations.

Chapter 11: File and Device I/O

In previous chapters, we used threads in various ways to perform CPU-bound work. However, not all operations are CPU related. Some require communication with files or other devices, commonly known as I/O operations. These operations do not require CPU usage until the operations complete, at which point thread code continues processing with the result of the I/O operation.

In this chapter, we'll examine I/O operations, both synchronous and asynchronous, and look at how threads can pick up I/O results efficiently to continue processing.

In this chapter:

- **The I/O System**
- **The `CreateFile` Function**
- **Synchronous I/O**
- **Asynchronous I/O**
- **I/O Completion Ports**
- **I/O Cancellation**
- **Devices**
- **Pipes and Mailslots**
- **Transactional NTFS**
- **File Search and Enumeration**
- **NTFS Streams**

The I/O System

The main purpose of the I/O system is to abstract access to physical and logical devices. Accessing a file in any file system should not be different than accessing a serial port, a USB camera or a printer. The I/O System is comprised of multiple components, some in user mode and most in kernel mode. The most important pieces are shown in figure 11-1.

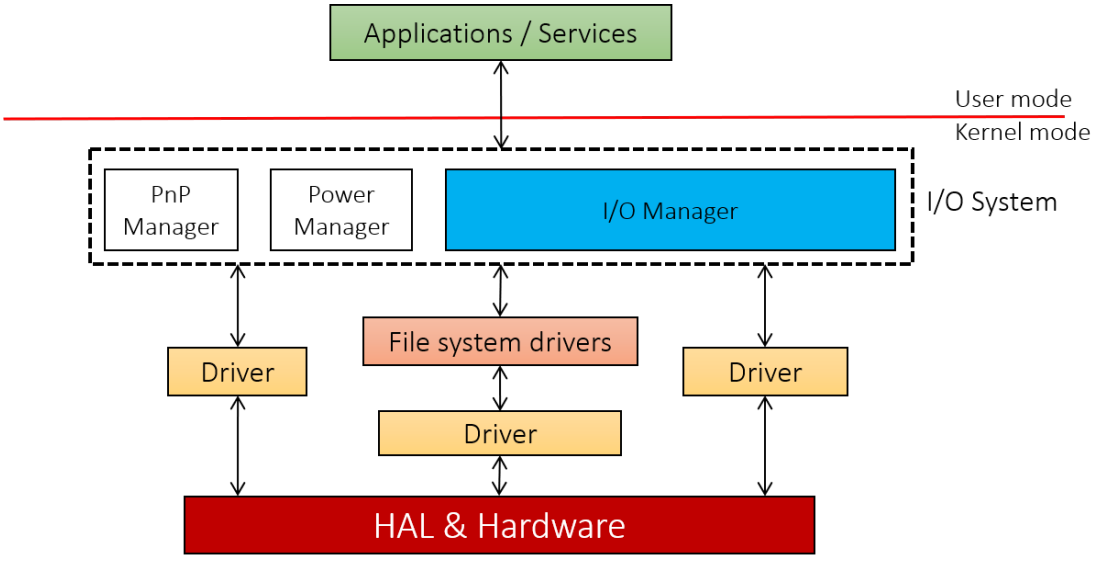


Figure 11-1: Important parts of the I/O System

User-mode processes call into the I/O system using various Windows APIs, which will be examined in this chapter. All file and device operations on the kernel side are initiated by the I/O manager. A request, such as read or write, is handled by creating a kernel structure called *I/O Request Packet* (IRP), filling in the details of the request and then passing it to the appropriate device driver. For real files, this goes to a file system driver, such as NTFS. This process is not essentially different than normal system calls, as shown in figure 11-2.

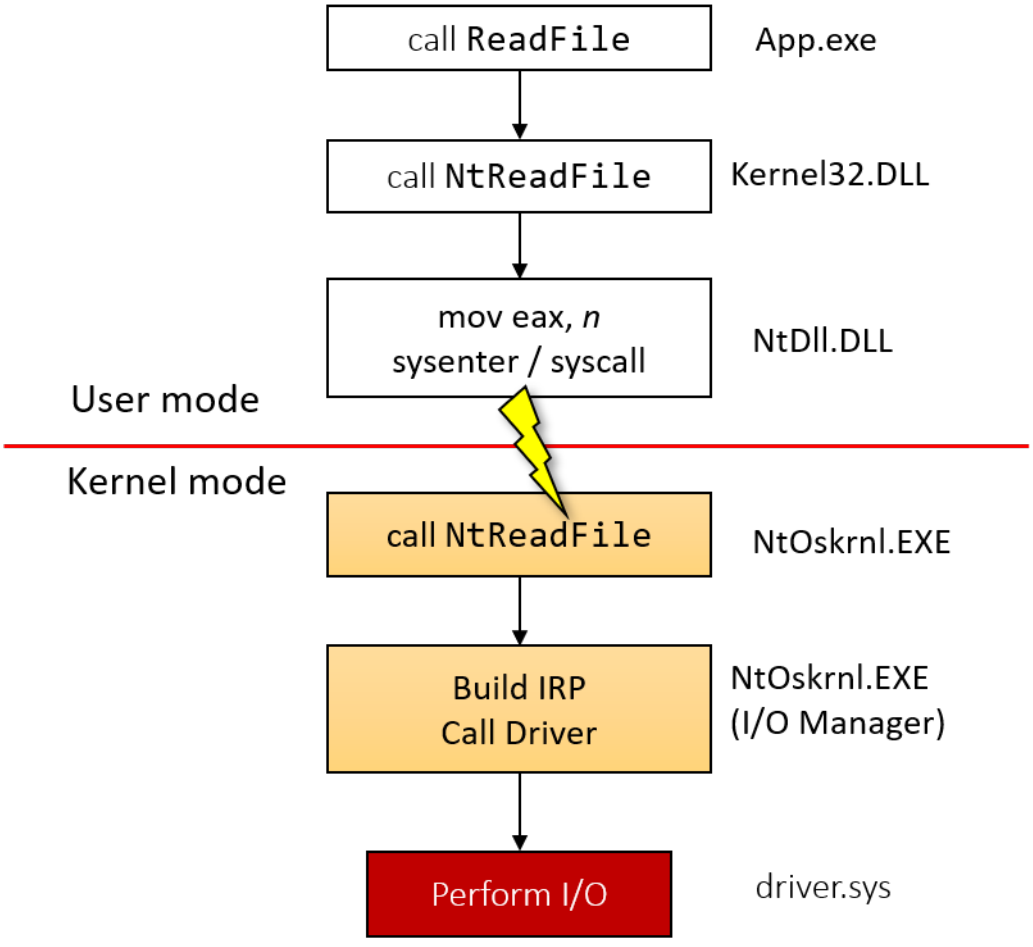


Figure 11-2: Read I/O operation call flow

As far as the kernel is concerned, I/O operations are always asynchronous. This means that a driver should initiate the operation and return as soon as possible so that the calling thread can regain control. The original caller, however, can choose to make the call synchronous. In that case, the I/O manager waits on behalf of the caller until the operation is done. This flexibility is very convenient from the client's perspective.

The `CreateFile` Function

The `CreateFile` function is the entry point to the world of I/O operations. The function name itself is somewhat misleading. The term "File" used in `CreateFile` is short for "File Object", which is the abstraction used by the kernel to represent a connection to a device, whether that device happens to be a file in a file system or not. Here is the prototype of `CreateFile`:

```
HANDLE CreateFile(
    _In_ LPCTSTR lpFileName,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwShareMode,
    _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _In_ DWORD dwCreationDisposition,
    _In_ DWORD dwFlagsAndAttributes,
    _In_opt_ HANDLE hTemplateFile);
```

Windows 8 and Server 2012 added a similar function, `CreateFile2`, defined like so:

```
typedef struct _CREATEFILE2_EXTENDED_PARAMETERS {
    DWORD dwSize;
    DWORD dwFileAttributes;
    DWORD dwFileFlags;
    DWORD dwSecurityQosFlags;
    LPSECURITY_ATTRIBUTES lpSecurityAttributes;
    HANDLE hTemplateFile;
} CREATEFILE2_EXTENDED_PARAMETERS, *PCREATEFILE2_EXTENDED_PARAMETERS;

HANDLE CreateFile2(
    _In_ LPCWSTR lpFileName,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwShareMode,
    _In_ DWORD dwCreationDisposition,
    _In_opt_ PCREATEFILE2_EXTENDED_PARAMETERS pCreateExParams);
```

`CreateFile2` is very similar to `CreateFile`, but is usable from UWP applications as well as desktop applications. `CreateFile`, on the other hand, cannot be called from UWP applications. Note that `CreateFile2` is Unicode only, whereas `CreateFile` has the usual `CreateFileA` and `CreateFileW` variants. `CreateFile2` also supports one new flag (`FILE_FLAG_OPEN_REQUIRING_OPLOCK`) that cannot be provided to `CreateFile`.

> Opportunistic locks (Oplocks) are beyond the scope of this chapter.

The `lpFileName` parameter indicates which file or device name to create or open. This is not necessarily a "file name", as indicated by the parameter's name. It is a symbolic link into the executive's object manager namespace, with some parsing rules added. Table 11-1 shows some common filename patterns, some of which are elaborated further in the following paragraphs.

Table 11-1: Common file name arguments for `CreateFile`

Filename format	Example	Description
x:*dir1**dir2**file*	c:\mydir\myfile.txt	Full path to a file/directory in the filesystem
..*dir1**file*	..\mydir\myfile.txt	Relative path to the file/directory system (.. means parent directory)
*dir1**dir2**file*	mydir1\mydir2\myfile.txt	Relative path to a file/directory from current directory
file	myfile.txt	File in the file system in the current directory
*server**share**dir1**dir2**file*	\myserver\myshare\mydir\myfile.txt	File/directory in a share on another machine
server\pipe*pipename*	\myserver\pipe\mypipe	Named pipe client
server\mailslot*mailslotname*	\myserver\mailslot\mymailslot	Mailslot client
\.*devicename*	\.\kobjexp	Device symbolic link name
builtin	com1	Old DOS names are treated as symbolic links rather than files in the current directory

The "symbolic link" is the fundamental value provided for the file name in `CreateFile`. Even something like *"c:"* that may look very "fundamental" is, in fact, a symbolic link. To view these symbolic link, we can check out the *WinObj* tool from *Sysinternals* we met briefly in chapter 2, or my own *Object Explorer* tool. Figure 11-3 shows *WinObj* with the *Global??* object directory selected. Figure 11-4 shows the same directory in *Object Explorer* (select *Objects* / *Object Manager Namespace* to open that view). The selected directory is the symbolic links directory.

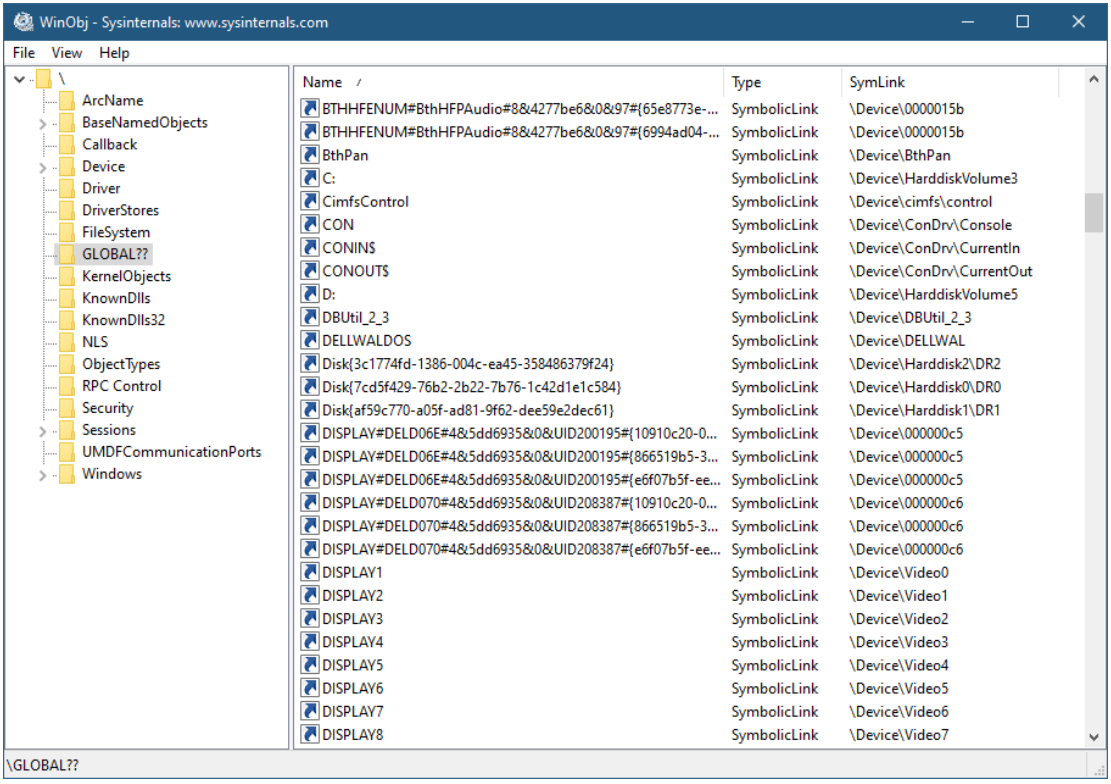


Figure 11-3: Symbolic links in *WinObj*

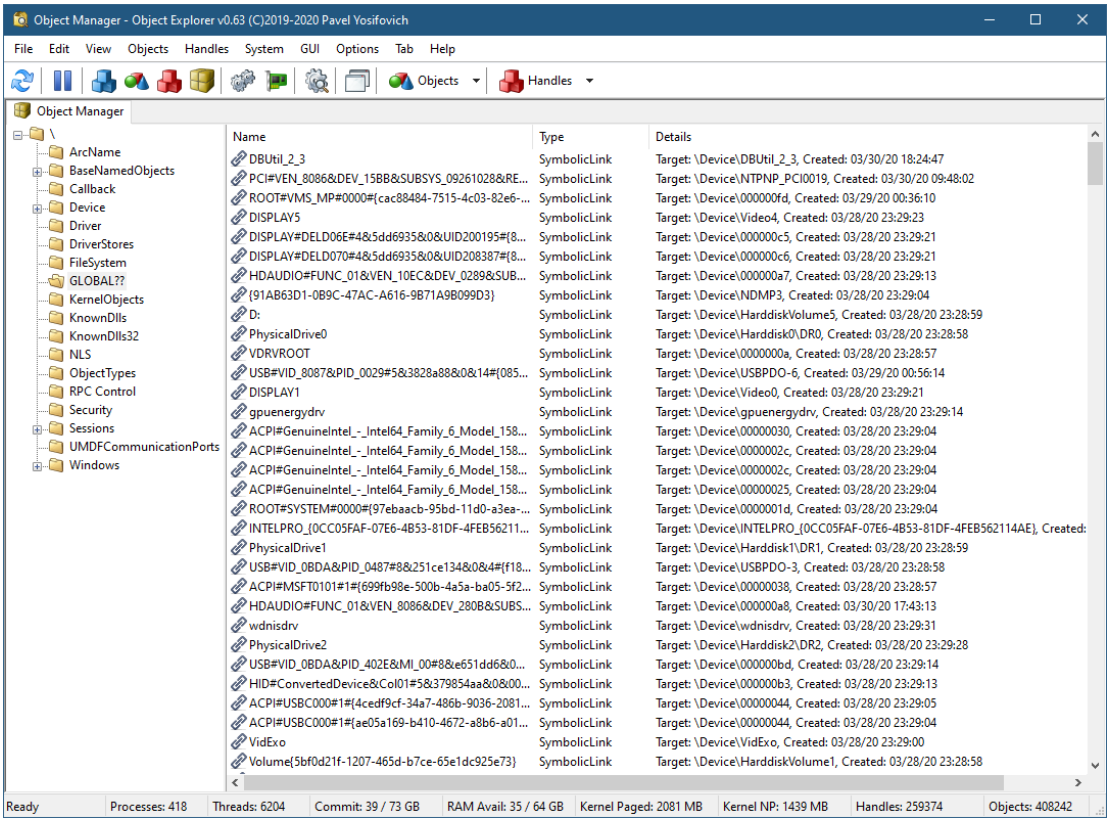


Figure 11-1: Symbolic links in *Object Explorer*

Every name in the list is a symbolic link, which is a candidate to `CreateFile` with the prefix "\\.\". Some symbolic links do not require this prefix, such as "C:". Notice that "C:" is indeed a symbolic link, pointing to something like "Device\HarddiskVolume3", which can be found under the *Device* object manager directory.

Some of the symbolic links look nice, like "C:", "PhysicalDrive0", "PIPE" and more, while others look like a jumble of numbers with GUIDs. These links are mostly for hardware devices. The subsection "Communicating with devices" later in this chapter provides further details.

Next, we'll examine the rest of the parameters to `CreateFile` before discussing some of the common "files" that can be accessed with the function.

The `dwDesiredAccess` parameter is used to specify the access mask required for accessing the file object. In most cases, you'll use one or more of the generic access rights: `GENERIC_READ` (to read data from the file/device), `GENERIC_WRITE` (to write data to the file/device), or both (`GENERIC_READ | GENERIC_WRITE`). You can also specify zero, if only very superficial information is to be accessed, such as a file's timestamp or size. You can also use more fine-grained access masks relevant to the file or device being accessed. For example, `FILE_READ_DATA` is a specific

access mask for files, requesting reading their contents. However, reading a file's attribute requires another access mask - `FILE_READ_ATTRIBUTES`. `GENERIC_READ`, being a generic access mask, is mapped to specific access masks, that for files include both `FILE_READ_DATA` and `FILE_READ_ATTRIBUTES`.

The `SYNCHRONIZE` and `FILE_READ_ATTRIBUTES` access masks are always requested regardless of whether they are specified explicitly or not.

> More information on access masks and generic mappings are provided in chapter 16 ("Security").

The `dwShareMode` parameter specifies the sharing mode the file/device should be opened with. This is mostly used with a file system file or directory. If the file/directory is not open, the caller specifies how she allows sharing the object with other `CreateFile` calls. For example, if the initial caller allows sharing as read-only, subsequent callers cannot open the object with `GENERIC_WRITE` access. If the file/directory is already open when another `CreateFile` call comes in for the same object, the share mode is ignored. Table 11-2 lists the possible sharing modes.

Table 11-2: Sharing modes for `CreateFile`

Sharing mode	Description
`0`	Object is opened with exclusive access. No other `CreateFile` call can succeed.
`FILE_SHARE_READ`	Object is allowed to be opened with `GENERIC_READ` access by subsequent callers.
`FILE_SHARE_WRITE`	Object is allowed to be opened with `GENERIC_WRITE` access by subsequent callers.
`FILE_SHARE_DELETE`	Object is allowed to be opened with `DELETE` access by subsequent callers. The file will be deleted when all handles are closed.
Combinations of the above	Combines the meanings of each flag

The next argument, `lpSecurityAttributes` is the standard `SECURITY_ATTRRIBUTES` discussed in chapter 2.

The `dwCreationDisposition` parameter specifies how to create or open file system objects (files and directories). For other devices, the flag should always be set to `OPEN_EXISTING`. The possible

values and their meanings are described in table 11-3.

Table 11-3: Creation disposition for `CreateFile`

Value	File exists	File does not exist
`CREATE_NEW` (1)	`CreateFile` fails	A new file is created
`CREATE_ALWAYS` (2)	Overwrites the existing file	Creates a new file
`OPEN_EXISTING` (3)	Opens the file	`CreateFile` fails
`OPEN_ALWAYS` (4)	Opens the file (no overwrite)	Creates the file
`TRUNCATE_EXISTING` (5)	Opens the file and truncates it to zero size	‘CreateFile fails

The `dwFlagsAndAttributes` parameter allows setting three separate flags/values that can be combined by the normal OR operator:

- Various flags affecting the operations that are to be performed once the file object is created (table 11-4)
- File attributes on the resulting file in case a new file is created in the file system (table 11-5)
- Quality of service flags for named pipe clients, if `SECURITY_SQOS_PRESENT` flag is present as well. Named pipes are discussed in chapter 18 (in part 2).

Table 11-4: Standard flags for `dwFlagsAndAttributes`

Flag (`FILE_FLAG_*`)	Description
`WRITE_THROUGH`	Forces any write operations flush data to disk (and written to the cache as well)
`NO_BUFFERING`	Forces operations to go directly to disk (no caching is used)
`SEQUENTAIL_SCAN`	A hint to the file system that a sequential read is is the typical operation on the file, may have positive effects on performance
`RANDOM_ACCESS`	A hint to the file system that random access to the file is expected
`DELETE_ON_CLOSE`	Indicates the file should be deleted when the last handle to it is closed
`OVERLAPPED`	Opens the file/device for asynchronous operations (see later in this chapter)
`BACKUP_SEMANTICS`	Required flag to open a handle to a directory rather than a file. The flag allows callers with the *Backup* or *Restore* privileges to open any file, regardless of security settings on the file
`POSIX_SEMANTICS`	Requests file name lookup is case sensitive. This flag does not seem to be respected in recent versions of Windows

Table 11-4: Standard flags for `dwFlagsAndAttributes`

Flag (`FILE_FLAG_*`)	Description
`OPEN_REPARSE_POINT`	Ignores the normal processing of the *reparse point* (if any) and opens the file for normal access (reparse points are beyond the scope of this chapter)
`OPEN_NO_RECALL`	A hint to the file system that a remote file will not necessarily be read to local storage
`SESSION_AWARE`	(Windows 8+) Device is opened with session awareness. This allows session 0 to open to access devices that are session-aware. This flag also requires a Registry entry to enable this check. The value *IoEnableSessionZeroAccessCheck* must be set to 1 in *HKLM\System\CurrentControlSet\SessionManager\I/O System*

Table 11-5: File attributes for `dwFlagsAndAttributes`

File attribute (`FILE_ATTRIBUTE_`)	Description
`NORMAL` or none	Normal file (if used, must be without any of the following attributes)
`HIDDEN`	File is hidden
`ARCHIVE`	File should be archived. It has no actual effect, but applications that perform file backup use it as a marker
`ENCRYPTED`	Contents of the file are encrypted
`READONLY`	File is read-only, and cannot be opened for write access
`SYSTEM`	File is a system file, to be used by the OS and system components only
`OFFLINE`	Actual storage of the file is elsewhere. This flag should not be set arbitrarily
`TEMPORARY`	A hint to the file system and cache manager that the file is used for temporary storage. The system tries to avoid writing the file's data to storage, as the file is expected to be deleted shortly. Good to combine with `FILE_FLAG_DELETE_ON_CLOSE`
`NOT_CONTENT_INDEXED`	The file will not be indexed by the indexing service

The defined attribute list is longer than those shown in table 11-5, but these extras cannot be set with `CreateFile`, and must be set with different APIs (depending on the attribute in question), described later in this chapter.

Careful use of the cache-related flags (`FILE_FLAG_WRITE_THROUGH`, `FILE_FLAG_NO_BUFFERING`,

FILE_FLAG_SEQUENTAIL_SCAN, FILE_FLAG_RANDOM_ACCESS) for files can have performance benefits. For FILE_FLAG_NO_BUFFERING, however, there are some requirements to make it work:

- Read/write access size is required to be a multiple of the volume sector size. This size can be discovered by calling GetDiskFreeSpace.
- The buffers used in read/write operations must be aligned on the physical sector size boundary. For these kinds of buffers, the VirtualAlloc function is recommended for allocation purposes, as it's always page (4 KB)-aligned (see chapter 13 for more on VirtualAlloc) or use the C runtime _aligned_malloc function. The physical sector size may be different from the logical sector size provided by GetDiskFreeSpace. To get the physical sector size, a DeviceIoControl call is needed on the volume with the IOCTL_STORAGE_QUERY_PROPERTY control code (see later in this chapter for more on DeviceIoControl).

If caching is used (the normal case), you can call FlushFileBuffers to force flushing data to the file:

```
BOOL FlushFileBuffers(_In_ HANDLE hFile);
```

The last parameter to CreateFile, hTemplateFile, is an optional file handle to copy attributes from in case a new file is being created. if specified, the handle must have the GENERIC_READ access mask. If an existing file is opened, this parameter is ignored.

CreateFile returns a handle to the created file object, or INVALID_HANDLE_VALUE if it fails (one of the few *Create* functions that do not return NULL on failure). As usual, GetLastError provides more information about the error.

In the following sections, we'll examine various aspects of working with files and devices and elaborate on some of the flags described in this section.

Working with Symbolic Links

As we've seen, CreateFile internally works by parsing symbolic links. These links are available to query with QueryDosDevice:

```
DWORD QueryDosDevice(
    _In_opt_ LPCTSTR lpDeviceName,
    _Out_    LPTSTR lpTargetPath,
    _In_     DWORD ucchMax);
```

The function works in two modes: if `lpDeviceName` is not `NULL`, the function looks up the symbolic link and returns its target (if any) in `lpTargetPath`. If `lpDeviceName` is `NULL`, it returns all symbolic links in `lpTargetPath`, separated by `'\0'`, so they can be iterated if desired, calling `QueryDosDevice` for each symbolic link. The last entry, in this case, has an extra `'\0'` to indicate the end of the list. The function returns the number of characters written to the target buffer, or zero if the function fails. If the function fails because the target buffer is too small, `GetLastError` returns `ERROR_INSUFFICIENT_BUFFER`.

The `symlinks` application allows querying symbolic links using `QueryDosDevice`. If no arguments are passed in, the application dumps all symbolic links and their targets. If an argument is provided, the application dumps only those symbolic links that have the provided string in their names.

The first step is to allocate a large enough buffer to read all symbolic links. This is needed since the application needs to either dump all of them or search through them for matches. Either way, all the symbolic links need to be read:

```
#include <memory>
#include <string>
#include <set>

using namespace std;

int wmain(int argc, wchar_t* argv[]) {
    auto size = 1 << 14;
    unique_ptr<WCHAR[]> buffer;
    for (;;) {
        buffer = make_unique<WCHAR[]>(size);
        if (0 == ::QueryDosDevice(nullptr, buffer.get(), size)) {
            if (::GetLastError() == ERROR_INSUFFICIENT_BUFFER) {
                size *= 2;
                continue;
            }
            else {
                printf("Error: %d\n", ::GetLastError());
                return 1;
            }
        }
        else
            break;
    }
```

A loop is constructed that allocates an array of characters with `std::make_unique<>` and then

`QueryDosDevice` is called with a `NULL` `lpDeviceName` to bring in all symbolic links. If the buffer is too small, the size is doubled, and the attempt is retried.

Once successful, the returned list must be iterated, calling `QueryDosDevice` again for each symbolic link that matches the provided command-line argument, or all of them. The application takes care of sorting the results by symbolic link using `std::set` (which inserts its elements into a binary search tree sorted by the element) with a custom comparer so the comparison is case insensitive (which is not the default for `std::wstring`):

```
if (argc > 1) {
    // convert argument to lowercase
    ::_wcslwr_s(argv[1], ::wcslen(argv[1]) + 1);
}
auto filter = argc > 1 ? argv[1] : nullptr;

// simplify stored type
using LinkPair = pair<wstring, wstring>;

struct LessNoCase {
    bool operator()(const LinkPair& p1, const LinkPair& p2) const {
        return ::_wcsicmp(p1.first.c_str(), p2.first.c_str()) < 0;
    }
};

// sorted by LessNoCase
set<LinkPair, LessNoCase> links;
WCHAR target[512];

for (auto p = buffer.get(); *p; ) {
    wstring name(p);

    auto locase(name);
    ::_wcslwr_s((wchar_t*)locase.data(), locase.size() + 1);
    if (filter == nullptr || locase.find(filter) != wstring::npos) {
        ::QueryDosDevice(name.c_str(), target, _countof(target));
        // add pair to results
        links.insert({ name, target });
    }

    // move to next item
    p += name.size() + 1;
}
```

```
// print results
for (auto& link : links) {
    printf("%ws = %ws\n", link.first.c_str(), link.second.c_str());
}
```

Here are example runs with some common symbolic links:

```
C:\>symlinks.exe c:
C: = \Device\HarddiskVolume3

c:\> symlinks.exe pipe
PIPE = \Device\NamedPipe

c:\>symlinks.exe nul
NUL = \Device\Null

c:\>symlinks con
CimfsControl = \Device\cimfs\control
CON = \Device\ConDrv\Console
CONIN$ = \Device\ConDrv\CurrentIn
CONOUT$ = \Device\ConDrv\CurrentOut
...
PartmgrControl = \Device\PartmgrControl
PciControl = \Device\PciControl
...
UVMLiteController = \Device\UVMLiteController0x1
VolMgrControl = \Device\VolMgrControl
```

The opposite function to `QueryDosDevice` exists as well, which allows defining new symbolic links:

```
BOOL DefineDosDevice(
  _In_     DWORD   dwFlags,
  _In_     LPCTSTR lpDeviceName,
  _In_opt_ LPCTSTR lpTargetPath);
```

`lpDeviceName` is the symbolic link's name and `lpTargetPath` is the target of the link. For example, making the following call sets up a new logical drive to point to an existing directory:

```
::DefineDosDevice(0, L"s:", L"c:\\Windows\\System32");
```

This is the same effect provided by the built-in Windows tool *subst.exe*:

```
c:>subst s: c:\windows\system32
```

After one of the previous calls, you can open *Explorer* and see the new drive appearing just like any other drive.
However, going back to *WinObj* or *Object Explorer*, the new symbolic link does not appear in the *Global??* object manager directory; that would be too powerful an operation. Instead, it's part of the *logon session* associated with the calling process' token. Figure 11-5 shows where such symbolic links are stored.

> The mapping affects the global namespace if the caller is running under the *LocalSystem* account.

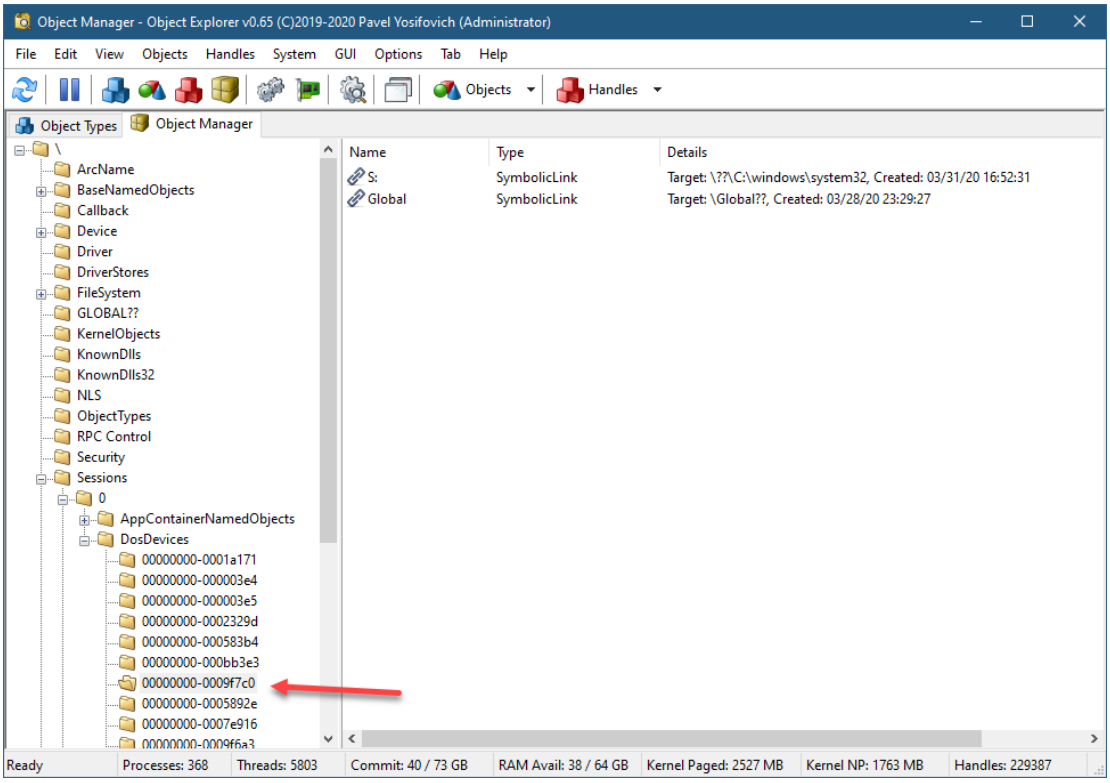


Figure 11-5: Symbolic links created with `DefineDosDevice`

The directory name pointed to by the arrow in figure 11-5 is the logon session ID. This is discussed further in chapter 16 ("Security").

The `dwFlags` parameter to `DefineDosDevice` can be zero or a combination of the values shown in table 11-6.

Table 11-6: Flags to `DefineDosDevice`

Flag	Description
`DDD_NO_BROADCAST_SYSTEM`	Prevent the function from broadcasting a `WM_SETTINGSCHANGE` message. This message normally allows applications such as *Explorer* to update their state
`DDD_RAW_TARGET_PATH`	The target path is interpreted as a native path (something like *\Device\Harddiskvolume3\MyDir*) instead of a Win32 path (*c:\MyDir*)
`DDD_REMOVE_DEFINITION`	Removes the symbolic link mapping. Typically `lpTargetPath` is set to `NULL` to delete the name provided in `lpDeviceName`. Otherwise, the target name is looked up for removal

Table 11-6: Flags to `DefineDosDevice`

Flag	Description
`DDD_EXACT_MATCH_ON_REMOVE`	Only valid with the previous flag. Performs an exact match on the target path (rather than a partial match)

Path Length

The file name length to `CreateFile` is traditionally limited to `MAX_PATH` characters, defined as 260. With the Unicode version of the function (`CreateFileW`), that path can be extended to about 32767 characters by prepending the path with *\\?* (e.g. *\\?\c:\MyDir\MyFile.txt*). Each part within the path is limited to 255 characters.

Remember that as part of a C/C++ string, each backslash must be escaped with another backslash. So "c:\temp" must be written "c:\\temp". An alternative with C++ 11 is to use the string literal feature. This is done by prepending `R` to the string before the quotes and then put the path surrounded by parenthesis. Example: `R"(c:\temp\file.txt)"` or with Unicode: `LR"(c:\temp\file.txt)"`. An optional delimiter sequence can appear before the `(` and must be the same after `)`, but for paths this is rarely needed.

Extending the path with *\\?* is also supported for *Universal Naming Convention* (UNC) paths as well. The prefix changes to *\\?\UNC*.

Windows 10 version 1607 and Windows Server 2016 added a new feature that allows breaking out of these path length limitations. This is an opt-in feature that requires two settings:

- A global registry value named *LongPathsEnabled* at *HKLM\System\CurrentControlSet\Control\FileSystem* must be set to 1 (`DWORD` value). The first time an I/O function is called for a process, this value is read and cached for the lifetime of the process. This means any change to this value will be noticed for new processes only. If this value is changed and the system is rebooted, all processes are guaranteed to see the new value.

- The specific executable must include the *longPathAware* element in its manifest and set to `true`. This is the full section in the manifest XML file:

```
<application xmlns="urn:schemas-microsoft-com:asm.v3">
    <windowsSettings
        xmlns:ws2="http://schemas.microsoft.com/SMI/2016/WindowsSettings">
        <ws2:longPathAware>true</ws2:longPathAware>
    </windowsSettings>
</application>
```

The Registry is a machine-wide setting and so requires admin access to change.

Most of the functions in the Windows API dealing with paths can cope with the possibly very long paths.

Be careful when using long paths, as most built-in applications do not have this setting in their manifest. For example, *Windows Explorer* doesn't have it. With very long paths, *Explorer* will not be able to handle such paths.

Directories

The `CreateFile` function can open a handle to an existing directory if the flag `FILE_FLAG_BACKUP_SEMANTICS` is specified in the `dwFlagsAndAttribute` argument. To create a directory, a separate function is needed:

```
BOOL CreateDirectory(
    _In_ LPCTSTR lpPathName,
    _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes);

BOOL CreateDirectoryEx(
    _In_     LPCTSTR lpTemplateDirectory,
    _In_     LPCTSTR lpNewDirectory,
    _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

The `lpPathName` parameter of `CreateDirectory` and `lpNewDirectory` of `CreateDirectoryExW` specifies the path of the new directory (this can be a full path or relative path, but all components of the path except the new directory must exist before the call). An optional `SECURITY_ATTRIBUTES` pointer can be supplied to set the security descriptor for the new directory (see chapter 16 for more information). Finally, the `lpTemplateDirectory` parameter to `CreateDirectoryEx` allows specifying an existing directory from which some properties of the new directory are copied, such as compression and encryption settings.

Files

Once a file handle is open, there is some basic information about the file that can be queried. Probably the most common is the file's size:

```
DWORD GetFileSize(
    _In_ HANDLE hFile,
    _Out_opt_ LPDWORD lpFileSizeHigh);

BOOL GetFileSizeEx(
    _In_ HANDLE hFile,
    _Out_ PLARGE_INTEGER lpFileSize);
```

File sizes are 64-bit, implying Windows can handle extremely large files. The maximum file size in practice is much more limited than the theoretical 2 to the 64th power (16 EB) bytes and depends on the actual disk size, the file system and some attributes, such as whether the file is compressed or sparse (discussed later in this chapter). Still, files larger than 32-bit size (4 GB) are fairly common, and code should generally expect such sizes unless there is a compelling reason to assume otherwise.

`GetFileSize` returns the low 32-bit of the file size as its return value, and the high 32-bit value in the `lpFileSizeHigh` parameter, if specified. If `lpFileSizeHigh` is `NULL`, the high 32-bit value is not returned. The function returns `INVALID_FILE_SIZE` in case of an error, defined as `0xffffffff`.

`GetFileSizeEx` is simpler, as it returns the file size in a `LARGE_INTEGER` structure, which we met before. This function returns the usual boolean to indicate success or failure.

The file size returned by the above two functions is the logical file size, which may be different than the physical file size. For example, if the file is compressed or sparse, its actual size on the disk is likely to be smaller. For such files, there is a dedicated function that can be used to query the actual file size on disk:

```
DWORD GetCompressedFileSize(
    _In_ LPCTSTR lpFileName,
    _Out_opt_ LPDWORD lpFileSizeHigh);
```

`GetCompressedFileSize` accepts the file name rather than a handle, and returns the requested size in the same format as `GetFileSize`.

Another piece of basic information about a file relates to its creation, modification and access times. `GetFileTime` retrieves these values:

```
BOOL GetFileTime(
    _In_ HANDLE hFile,
    _Out_opt_ LPFILETIME lpCreationTime,
    _Out_opt_ LPFILETIME lpLastAccessTime,
    _Out_opt_ LPFILETIME lpLastWriteTime
    );
```

The returned times are in the usual units of 100nsec from January 1, 1601. A `NULL` pointer can be provided for any of the times to indicate that a particular result is of no interest to the caller.

File attributes can be retrieved with `GetFileAttributes` or `GetFileAttributesEx`:

```
DWORD GetFileAttributes(_In_ LPCTSTR lpFileName);

BOOL GetFileAttributesEx(
    _In_ LPCTSTR lpFileName,
    _In_ GET_FILEEX_INFO_LEVELS fInfoLevelId,
    _Out_ LPVOID lpFileInformation);
```

`GetFileAttributes` accepts a file name and returns its attributes. These attributes include those in table 11-5 and can include additional attributes that are not legal to set in a `CreateFile` call. These additional values are listed in table 11-7.

Table 11-7: More file attributes

File attribute (`FILE_ATTRIBUTE_`)	Description
`DIRECTORY` or none	Directory (rather than a file)
`REPARSE_POINT`	The file has an associated reparse point
`COMPRESSED`	The file is compressed
`SPARSE_FILE`	The file is a sparse file
`INTEGRITY_STREAM`	(Windows 8+) Directory or data stream is configured with integrity (ReFS only)*
`NO_SCRUB_DATA`	(Windows 8+) The data stream should not be read by the data integrity scanner (ReFS and storage spaces only)*

* A detailed discussion of the attributes `FILE_ATTRIBUTE_INTEGRITY_STREAM` and `FILE_ATTRIBUTE_NO_SCRUB_DATA` are beyond the scope of this book.

`GetFileAttributesEx` currently accepts a single "level" with the value `GetFileExInfoStandard`, and returns a `WIN32_FILE_ATTRIBUTE_DATA` structure defined like so:

```
typedef struct _WIN32_FILE_ATTRIBUTE_DATA {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
    DWORD nFileSizeHigh;
    DWORD nFileSizeLow;
} WIN32_FILE_ATTRIBUTE_DATA, *LPWIN32_FILE_ATTRIBUTE_DATA;
```

In addition to the file attributes discussed earlier, the function returns the file's creation time, last access time, last write time, and size.

To get even more information for a file, call `GetFileInformationByHandle`:

```
BOOL GetFileInformationByHandle(
    _In_ HANDLE hFile,
    _Out_ LPBY_HANDLE_FILE_INFORMATION lpFileInformation);
```

The function takes a handle to a file, rather than a file path, and returns a superset of `WIN32_FILE_ATTRIBUTE_DATA` called `BY_HANDLE_FILE_INFORMATION`:

```
typedef struct _BY_HANDLE_FILE_INFORMATION {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
    DWORD dwVolumeSerialNumber;
    DWORD nFileSizeHigh;
    DWORD nFileSizeLow;
    DWORD nNumberOfLinks;
    DWORD nFileIndexHigh;
    DWORD nFileIndexLow;
} BY_HANDLE_FILE_INFORMATION, *PBY_HANDLE_FILE_INFORMATION;
```

The function returns several pieces of information, some of which are also part of `WIN32_FILE_ATTRIBUTE_DATA`. The extras include the volume serial number, the number of links to the file (can be more than 1 on NTFS if there are hard links to the file), and the file's index (64 bit, provided as two 32-bit numbers). The times are given in the usual 100nsec units since January

1, 1601. The file index is unique to a file on a particular volume. This means combining the file index with the volume number gives an identity for a file on a particular machine. This can be used for comparing two file handles, indicating whether they point to the same file or not.

If you're only interested in a file's attributes, `GetFileInformationByHandle` is faster than `GetFileAttributes` or `GetFileAttributesEx`, because it uses an already open handle. The other functions need to open the file, get the information and close it. Always prefer working with a handle (if you have it) than using a file path.

To get even more information about a file, call `GetFileInformationByHandleEx`:

```
BOOL GetFileInformationByHandleEx(
    _In_  HANDLE hFile,
    _In_  FILE_INFO_BY_HANDLE_CLASS FileInformationClass,
    _Out_ LPVOID lpFileInformation,
    _In_  DWORD dwBufferSize
);
```

The function has quite a few pieces of information that can be retrieved, where the information requested is provided by the `FILE_INFO_BY_HANDLE_CLASS` enumeration:

```
typedef enum _FILE_INFO_BY_HANDLE_CLASS {
    FileBasicInfo,
    FileStandardInfo,
    FileNameInfo,
    FileRenameInfo,
    FileDispositionInfo,
    FileAllocationInfo,
    FileEndOfFileInfo,
    FileStreamInfo,
    FileCompressionInfo,
    FileAttributeTagInfo,
    FileIdBothDirectoryInfo,
    FileIdBothDirectoryRestartInfo,
    FileIoPriorityHintInfo,
    FileRemoteProtocolInfo,
    FileFullDirectoryInfo,
    FileFullDirectoryRestartInfo,
#if (_WIN32_WINNT >= _WIN32_WINNT_WIN8)
    FileStorageInfo,
```

```
    FileAlignmentInfo,
    FileIdInfo,
    FileIdExtdDirectoryInfo,
    FileIdExtdDirectoryRestartInfo,
#endif
#if (_WIN32_WINNT >= _WIN32_WINNT_WIN10_RS1)
    FileDispositionInfoEx,
    FileRenameInfoEx,
#endif
#if (NTDDI_VERSION >= NTDDI_WIN10_19H1)
    FileCaseSensitiveInfo,
    FileNormalizedNameInfo,
#endif
    MaximumFileInfoByHandleClass
} FILE_INFO_BY_HANDLE_CLASS, *PFILE_INFO_BY_HANDLE_CLASS;
```

That's quite a list. The conditional compilation shows which values have been added in Windows 8 / Server 2012 and later, in Windows 10 version 1607 and Server 2016, and Windows 10 version 1903.

For each enumeration value, the documentation describes the structure associated with the value.

Setting File Information

`GetFileAttributes` has a complementary function to set file attributes:

```
BOOL SetFileAttributes(
    _In_ LPCTSTR lpFileName,
    _In_ DWORD dwFileAttributes);
```

The attributes that can be set by these function are: `FILE_ATTRIBUTE_ARCHIVE`, `FILE_ATTRIBUTE_HIDDEN`, `FILE_ATTRIBUTE_NORMAL`, `FILE_ATTRIBUTE_NOT_CONTENT_INDEXED`, `FILE_ATTRIBUTE_OFFLINE`, `FILE_ATTRIBUTE_READONLY`, `FILE_ATTRIBUTE_SYSTEM`, and `FILE_ATTRIBUTE_TEMPORARY`.

Other attributes that can be changed with other APIs include:

- `FILE_ATTRIBUTE_COMPRESSED` - call `DeviceIoControl` with `FSCTL_SET_COMPRESSION` control code (see later in this chapter for more on `DeviceIoControl`).
- `FILE_ATTRIBUTE_ENCRYPTED` - if not created as such, call the `EncryptFile` to encrypt the file's current contents, and set the attribute.

- `FILE_ATTRIBUTE_REPARSE_POINT` - call `DeviceIoControl` with the

`FSCTL_SET_REPARSE_POINT` control code to associate the file with a reparse point.

- `FILE_ATTRIBUTE_SPARSE_FILE` - call `DeviceIoControl` with the `FSCTL_SET_SPARSE` control code to turn the file into a sparse one. A sparse file is expected to contain mostly zeros, so it can use less disk space.

Changing the times associated with a file is a matter of calling `SetFileTime`:

```
BOOL SetFileTime(
    _In_ HANDLE hFile,
    _In_opt_ CONST FILETIME* lpCreationTime,
    _In_opt_ CONST FILETIME* lpLastAccessTime,
    _In_opt_ CONST FILETIME* lpLastWriteTime);
```

The handle must have the `FILE_WRITE_ATTRIBUTES` to allow these changes. The caller can specify `NULL` for each value it does not wish to change.

To set other information pieces for a file, the complement of `GetFileInformationByHandleEx` can be used:

```
BOOL SetFileInformationByHandle(
    _In_ HANDLE hFile,
    _In_ FILE_INFO_BY_HANDLE_CLASS FileInformationClass,
    _In_reads_bytes_(dwBufferSize) LPVOID lpFileInformation,
    _In_ DWORD dwBufferSize);
```

The `FileInformationClass` parameter is of the same enumeration given to `GetFileInformationByHandleEx`. However, only a small subset of information classes can be used for setting data: `FileBasicInfo`, `FileRenameInfo`, `FileDispositionInfo`, `FileAllocationInfo`, `FileEndOfFileInfo`, and `FileIoPriorityHintInfo`.

Synchronous I/O

When calling `CreateFile` and not specifying `FILE_FLAG_OVERLAPPED` as part of the `dwFlagsAndAttributes` parameter, the file object is created for synchronous I/O only. This is the simplest to work with, so we'll tackle synchronous I/O first.

The main functions to perform I/O are `ReadFile` and `WriteFile`, which work with any file object (not necessarily pointing to a file system file):

```
BOOL ReadFile(
    _In_ HANDLE hFile,
    _Out_ LPVOID lpBuffer,
    _In_ DWORD nNumberOfBytesToRead,
    _Out_opt_ LPDWORD lpNumberOfBytesRead,
    _Inout_opt_ LPOVERLAPPED lpOverlapped);

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

These functions work for synchronous and asynchronous I/O. `lpBuffer` is the buffer from which data is to be read (`WriteFile`) or the buffer to which data is to be written (`ReadFile`). For `ReadFile`, `nNumberOfBytesToRead` indicates how many bytes to read into the buffer, and for `WriteFile`, `nNumberOfBytesToWrite` indicates how many bytes to write.
The number of bytes actually read/written is returned in `lpNumberOfBytesRead` (read) or `lpNumberOfBytesWritten` (write). It can be smaller than the requested number of bytes or even zero. Note that you cannot pass `NULL` for these arguments with synchronous I/O, or else you will get an access violation when the function attempts to dereference the `NULL` pointer.

The last parameter, `lpOverlapped`, is required to be non-`NULL` for asynchronous operations, but for Synchronous I/O it should be `NULL`.

These functions are synchronous, which means the calling thread is now blocked (goes into a wait state) until the operation is complete and the data has been transferred. The functions return `FALSE` on failure, with `GetLastError` providing the exact error encountered.

The following example shows how to create a new file and write some data to it:

```
HANDLE hFile = ::CreateFile(LR"(c:\temp\mydata.txt)",
    GENERIC_WRITE,      // access
    0,                  // sharing (exclusive)
    nullptr,            // SECURITY_ATTRIBUTES
    CREATE_NEW,         // creation disposition
    0,                  // flags and attributes
    nullptr);           // template file
if(hFile != INVALID_HANDLE_VALUE) {
    char text[] = "Hello from Windows!";
    DWORD bytes;
    ::WriteFile(hFile, text, ::strlen(text), &bytes, nullptr);
```

```
    ::CloseHandle(hFile);
}
```

The next example reads all bytes in a file:

```
HANDLE hFile = ::CreateFile(LR"(c:\temp\mydata.txt)",
    GENERIC_READ,       // access
    FILE_SHARE_READ,    // sharing
    nullptr,            // SECURITY_ATTRIBUTES
    OPEN_EXISTING,      // creation disposition
    0,                  // flags and attributes
    nullptr);           // template file
if(hFile != INVALID_HANDLE_VALUE) {
    // assume file size is less than 4GB
    DWORD size = ::GetFileSize(hFile, nullptr);
    auto buffer = std::make_unique<char[]>(size + 1);
    DWORD bytes;
    if(::ReadFile(hFile, buffer.get(), size, &bytes, nullptr)) {
        // assume data is expected to be ASCII text
        buffer[bytes] = '\0';   // add string terminator
        printf("%s\n", buffer.get());
    }
    ::CloseHandle(hFile);
}
```

Each file object opened for synchronous access maintains an internal file pointer, that is automatically advanced with each I/O operation. For example, if a file is opened and a read operation of 10 bytes is performed, the file pointer advances by 10 bytes after the operation completes. If another read for 10 bytes is issued, it reads bytes 10 to 19, and the file pointer advances to position 20 in the file.

For sequential reads and writes, that's great. In some cases, however, you may want to jump forward or back and read/write from a different position. This can be accomplished with one of the following functions:

```
DWORD SetFilePointer(
    _In_ HANDLE hFile,
    _In_ LONG lDistanceToMove,
    _Inout_opt_ PLONG lpDistanceToMoveHigh,
    _In_ DWORD dwMoveMethod);

BOOL SetFilePointerEx(
    _In_ HANDLE hFile,
    _In_ LARGE_INTEGER liDistanceToMove,
    _Out_opt_ PLARGE_INTEGER lpNewFilePointer,
    _In_ DWORD dwMoveMethod);
```

The functions move the internal file pointer to the desired position. `SetFilePointerEx` is easier to use, since it allows a full 64-bit offset to be provided in the `liDistanceToMove` parameter. `SetFilePointer` accepts the low 32-bit of the offset in `lDistanceToMove` and optionally the high 32-bit in `lpDistanceToMoveHigh`. Both functions attempt to return the previous file pointer: `SetFilePointerEx` in `lpNewFilePointer`, and `SetFilePointer` in the return value (low 32-bit) and `lpDistanceToMoveHigh` (if not `NULL` - the high 32-bit).

The offset to move, however, is not necessarily the offset from the start of the file. The last parameter, `dwMoveMethod`, indicates how to interpret the provided offset:

- `FILE_BEGIN` (0) - from the beginning of the file
- `FILE_CURRENT` (1) - from the current file position
- `FILE_END` (2) - from the end of the file

You can query the current file pointer without moving by specifying a zero distance to move, and `FILE_CURRENT` move method. You can move to the end of the file by specifying zero for the offset and `FILE_END` for the method.

Multiple file objects open on the same file are different and not synchronized in anything, including their file pointer. Each has its own and they don't affect each other.

Calling `SetFilePointer(Ex)` with an offset that is beyond the file's current size, extends the file to that size. Conversely, trimming a file can be done by calling `SetEndOfFile` after setting the file pointer to the required size:

```
BOOL SetEndOfFile(_In_ HANDLE hFile);
```

Asynchronous I/O

As describes at the beginning of this chapter, the Windows I/O system is asynchronous in nature. Once a device driver issues a request to its controlled hardware (such a disk drive), the driver does not need to wait for the operation to complete. Instead, it marks the request as "pending" and returns to its caller. The thread is now free to perform other operations while the I/O is in progress.

After some time the I/O operation completes by the hardware device. The device issues a hardware interrupt that causes a driver-supplied callback to run and complete the pended request.

Using synchronous I/O is simple and easy, and in many cases good enough. However, if lots of requests are to be served, it's inefficient to create a thread per request that would initiate an I/O operation and wait for it to complete; this approach does not scale well. Asynchronous I/O provides a solution, where a thread initiates a request, and then goes back to serve the next request, and so on, since I/O operations operate concurrently while the CPU executes other code. The only wrinkle in this simplistic model is how a thread is notified of an I/O operation completion. As we'll soon see, there are a few techniques provided by Windows to deal with this.

Requesting asynchronous operations must start with the original `CreateFile` call (which, by the way, is always synchronous). The `FILE_FLAG_OVERLAPPED` flag must be specified as part of `dwFlagsAndAttributes` parameter. This opens the file/device in asynchronous mode.

One of the consequences of a file opened for asynchronous access is that there is no file pointer anymore. This means every operation must somehow provide an offset from the start of the file to perform the operation (the size is not a problem since it's part of the read/write call). This is one of the tasks of the `OVERLAPPED` structure, that must be passed as the last argument to `ReadFile` and `WriteFile`:

```
typedef struct _OVERLAPPED {
    ULONG_PTR Internal;
    ULONG_PTR InternalHigh;
    union {
        struct {
            DWORD Offset;
            DWORD OffsetHigh;
        };
        PVOID Pointer;
    };
```

```
    HANDLE hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

This structure contains three distinct pieces of information:

- `Internal` and `InternalHigh` are as named, used by the I/O manager and should not be written to, although their usage is described below.
- `Offset` and `OffsetHigh` are the offsets to set, indicating where the operation is to start within the file. The `Pointer` member of the union is an alternative to these fields that is somewhat easier to work with if a 64-bit offset is required.
- `hEvent` is a handle to a kernel event object, that, if non-`NULL`, is signaled by the I/O manager when the operation completes.

Technically, `Internal` holds the I/O operation's error code. For an asynchronous operation in progress, it holds `STATUS_PENDING`, which is the kernel equivalent of `STILL_ACTIVE` (0x103). In fact, Windows defines a macro, `HasOverlappedIoCompleted` that takes advantage of this fact. Here is its definition:

```
#define HasOverlappedIoCompleted(lpOverlapped) \
    (((DWORD)(lpOverlapped)->Internal) != STATUS_PENDING)
```

The `InternalHigh` member stores the number of bytes transferred once the operation completes.

In an asynchronous operation, the `ReadFile` or `WriteFile` calls normally return `FALSE`, since the operation is not yet complete, it just started. In some cases, the underlying device driver might decide to perform the operation synchronously, in which case the functions return `TRUE`. This should happen rarely with `ReadFile` and `WriteFile` targeted at a file system file.

If the functions return `FALSE`, then calling `GetLastError` returns `ERROR_IO_PENDING`, it means everything is working as planned - the operation is underway and the thread can continue doing something else. If a different error is returned, then this is a real error - the operation has not started.

Both `ReadFile` and `WriteFile` have a returned bytes parameter that returns the number of bytes transferred in the case of a synchronous operation. With an asynchronous operation, this makes no sense, since the operation has not yet completed. Although you can provide an address of a `DWORD` and just disregard the result, it's better to specify `NULL` for this argument to avoid any confusion.

Given the above information, the following code example opens a file for asynchronous access, performs a read operation, does some more processing while the operation is underway, and then waits for the operation to complete:

```
HANDLE hFile = ::CreateFile(LR"(c:\temp\mydata.txt)", GENERIC_READ,
    FILE_SHARE_READ, nullptr, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, nullptr);
if (hFile != INVALID_HANDLE_VALUE) {
    // initialize OVERLAPPED
    OVERLAPPED ov = { 0 };      // offset is zero
    ov.hEvent = ::CreateEvent(nullptr, TRUE, FALSE, nullptr);

    BYTE buffer[1 << 12];   // 4KB
    BOOL ok = ::ReadFile(hFile, buffer, sizeof(buffer), nullptr, &ov);
    if(!ok) {
        if (::GetLastError() != ERROR_IO_PENDING) {
            // some real error occurred...
            return;
        }
        else {
            // do some other work...

            // wait for the operation to complete
            ::WaitForSingleObject(ov.hEvent, INFINITE);
            ::CloseHandle(ov.hEvent);
        }
    }
    // do something with the result...
    ::CloseHandle(hFile);
}
```

There are few things to note about the above code:

- The `OVERLAPPED` instance must survive as long as the I/O operation is in progress. In the above code it's allocated on the stack, and the caller thread waits for the operation to complete, so the instance is guaranteed to survive. In more complex cases, it may need to be allocated dynamically.
- The event is waited on by the original caller, but this is not a requirement; any thread can wait on the event, including a thread pool thread (as demonstrated in chapter 9).

Once the operation completes, how can you tell how many bytes were transferred? We can't get it from the original `ReadFile`/`WriteFile` call. We can get it from the `InternalHigh` member of the `OVERLAPPED` structure, but there is a dedicated function to get this information:

```
BOOL GetOverlappedResult(
    _In_ HANDLE hFile,
    _In_ LPOVERLAPPED lpOverlapped,
    _Out_ LPDWORD lpNumberOfBytesTransferred,
    _In_ BOOL bWait);
```

`GetOverlappedResult` accepts the file handle and the `OVERLAPPED` structure for the particular operation you're interested in (you can initiate several operations from the same file handle, each having a different `OVERLAPPED` instance). `lpNumberOfBytesTransferred` returns the number of bytes actually transferred.

The final parameter, `bWait`, specifies whether to wait for the operation to complete (`TRUE`) before reporting the result or not. If the operation already completed, then it doesn't matter. if the operation is still in progress and `bWait` is `TRUE`, the caller thread waits until the operation is complete and the function returns `TRUE`. If `bWait` is `FALSE` and the operation is still in progress, the function returns `FALSE` and `GetLastError` returns `ERRO_IO_INCOMPLETE`.

An extended function, `GetOverlappedResultEx`, provides more flexibility while waiting for the operation to complete:

```
BOOL GetOverlappedResultEx(
    _In_ HANDLE hFile,
    _In_ LPOVERLAPPED lpOverlapped,
    _Out_ LPDWORD lpNumberOfBytesTransferred,
    _In_ DWORD dwMilliseconds,
    _In_ BOOL bAlertable);
```

The function allows setting a timeout limit for waiting (`dwMilliseconds`), and also the ability to wait in an alertable state (`bAlertable`) (see chapter 7 for more on alertable state).

Windows provides several ways to deal with asynchronous I/O completion. We've just seen one of these, using an event object. Table 11-8 summarizes the available options.

Table 11-8: Asynchronous completion handling options

Mechanism	Remarks
Waiting on the file handle	Easy to use, but limited to a single operation
Waiting on an event in the `OVERLAPPED` structure	Easy to use. Any thread can wait on the event
Using `ReadFileEx` and `WriteFileEx` with a callback	Callback queued as APC to caller thread, meaning it's the only thread that can process the result
I/O completion port	Not as easy as the others, but flexible and powerful

The first option in table 11-8 indicates that a file handle is a waitable object, and becomes signaled when an asynchronous operation completes. This works fine if there is only one such request in progress at a time. If more than one request is in progress, the file handle is signaled when the first completes, but there is no guarantee of which request that would be.

Generally, I/O operations can complete out of order, since it's the driver's prerogative in how to schedule the actual requests. You should never rely on some order of I/O operation completion.

Since waiting on a file handle is not ideal, you can tell the system not to bother signaling it:

```
BOOL SetFileCompletionNotificationModes(
    _In_ HANDLE FileHandle,
    _In_ UCHAR Flags);
```

One of the flags is `FILE_SKIP_SET_EVENT_ON_HANDLE`, which is the one needed to tell the I/O manager to skip signaling the file object.

The second option of using an event object tucked in the `OVERLAPPED` structure works well, provided that each operation is associated with its own event. Any thread can wait on the event, including thread pool threads.

`ReadFileEx` and `WriteFileEx`

The third option for responding to completed I/O operation is by using an extended version of `ReadFile` or `WriteFile`:

```
BOOL ReadFileEx(
    _In_    HANDLE hFile,
    _Out_   LPVOID lpBuffer,
    _In_    DWORD nNumberOfBytesToRead,
    _Inout_ LPOVERLAPPED lpOverlapped,
    _In_    LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);

BOOL WriteFileEx(
    _In_    HANDLE hFile,
    _In_    LPCVOID lpBuffer,
    _In_    DWORD nNumberOfBytesToWrite,
    _Inout_ LPOVERLAPPED lpOverlapped,
    _In_    LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

The functions are identical to their non-Ex counterparts, except for an additional argument, which is a function pointer that must have the following prototype:

```
typedef VOID (WINAPI *LPOVERLAPPED_COMPLETION_ROUTINE)(
    _In_    DWORD dwErrorCode,
    _In_    DWORD dwNumberOfBytesTransfered,
    _Inout_ LPOVERLAPPED lpOverlapped);
```

One the face of it, this mechanism looks close to perfect: a callback is invoked when the asynchronous I/O operation completes. Unfortunately, the callback is wrapped in an *Asynchronous Procedure Call* (APC) and queued to the original thread that called `ReadFileEx` or `WriteFileEx`. This means this particular thread is the only one which can invoke the callback by getting into an alertable state, at least occasionally. Refer to chapter 8 for more on APCs and alertable state.

Once invoked, the callback function provides the error code of the operation (`ERROR_SUCCESS` if all is well), the number of bytes transferred, and the original pointer to the `OVERLAPPED` structure. This last bit is very convenient, because if the structure was allocated dynamically (highly likely), the code can free the structure within the callback - the I/O manager does not touch the structure again at this point.

Manually Queued APC

It's worthwhile describing another function related to APCs in general. It's the ability to queue an APC to a target thread:

```
DWORD QueueUserAPC(
    _In_ PAPCFUNC pfnAPC,
    _In_ HANDLE hThread,
    _In_ ULONG_PTR dwData);
```

The function queues an APC to the target thread represented by the `hThread` parameter, that must have the `THREAD_SET_CONTEXT` access mask. `pfnAPC` is a function pointer that must have the following prototype:

```
typedef VOID (WINAPI *PAPCFUNC)(_In_ ULONG_PTR Parameter);
```

`dwData` is the value sent to the APC function in the `Parameter` parameter.

This is still an APC, and so the target thread must enter an alertable state if the APC callback is to be executed. This means queuing APCs to arbitrary threads is problematic; the caller should know

in advance that the target thread is likely to be in alertable state in the near future. Otherwise, these APCs queue up, up to some limit, but never executed.

One simple use of `QueueUserAPC` is to implement a simple queue served by a particular thread, without the need to create and manage any data structures. The thread that runs the work items need to be in an alertable state all the time, unless instructed to exit:

```
DWORD WorkThread(PVOID param) {
    // assume an event handle is passed in to signal thread exit
    HANDLE hEvent = (HANDLE)param;

    for(;;) {
        if(::WaitForSingleObjectEx(hEvent, INFINITE, TRUE) == WAIT_OBJECT_0)
            break;
        // if the wait is over and the event is not set, this means one or more\
 APCs
        // executed. Just continue waiting again for more APCs or an exit signal
    }
    return 0;
}
```

The thread is created by the following code:

```
HANDLE hEvent = ::CreateEvent(nullptr, TRUE, FALSE, nullptr);
HANDLE hThread = ::CreateThread(nullptr, 0, WorkThread, (PVOID)hEvent, 0, nullp\
tr);
```

At this point, a work item can be queued by using `QueueUserAPC`:

```
::QueueUserAPC(hThread, SomeFunction, SomeData);
```

Finally, when the thread should be torn down, just set the event, and optionally wait for the thread to exit, perhaps finishing up APCs still in the queue:

```
::SetEvent(hEvent);
::WaitForSingleObject(hEvent, INFINITE);
```

I/O Completion Ports

I/O completion ports deserve their own major section, since they are useful not just for handling asynchronous I/O. We met them briefly when discussing jobs in chapter 4 - a job can be associated

with an I/O completion port, to receive notifications related to the job. In this section, we'll focus on an I/O completion's usage for handling I/O completions.

An I/O completion port is associated with a file object (could be more than one). It encapsulates a queue of requests, and a list of threads that can serve these requests once completed. Whenever an asynchronous operation completes, one of the threads that waits on the completion ports should wake up and handle the completion, possibly initiating the next request.

The first step is creating an I/O completion port object and associating it with one or more file handles. This is a task for `CreateIoCompletionPort`:

```
HANDLE CreateIoCompletionPort(
    _In_ HANDLE FileHandle,
    _In_opt_ HANDLE ExistingCompletionPort,
    _In_ ULONG_PTR CompletionKey,
    _In_ DWORD NumberOfConcurrentThreads);
```

The function can perform two distinct operations, possibly combining the two. It can do any of the following:

- Create an I/O completion port not associated with any file objects.
- Associate an existing completion port to a file object.
- Combine the above two operations in a single call.

> This function is the only one in the *Create* kernel object functions that does not accept a `SECURITY_ATTRIBUTES` structure. This is because a completion port is always local to the process that created it. Technically, duplicating such a handle to another process succeeds, but the new handle is unusable.

Creating a new completion port without associating it with any file needs only the last argument, like so:

```
HANDLE hNewCP = ::CreateIoCompletionPort(INVALID_HANDLE_VALUE, nullptr, 0,
    NumberOfConcurrentThreads);
```

The number of concurrent threads indicates the maximum threads that can handle an I/O completion through this I/O completion port. Specifying zero sets the number to the number of logical processors on the system. We'll see the effect of this parameter a bit later.

Once a I/O completion port object is created, it can be associated with one or more file objects (handles). For each file handle, a completion key is specified, which is application-defined. The file object must have been opened with the `FILE_FLAG_OVERLAPPED`. Here is an example of adding a file object to a completion port:

```
const int Key = 1;
HANDLE hFile = ::CreateFile(..., FILE_FLAG_OVERLAPPED, ...);
HANDLE hOldCP = ::CreateIoCompletionPort(hFile, hNewCP, Key, 0);
assert(hOldCP == hNewCP);
```

There is no real need to capture the returning handle in this case, since the existing completion port handle was specified as the second argument. The above `assert` just makes it clearer.

> Remember that "file" is not necessarily a file in a file system. It could be a pipe, a socket, or a device, for example.

The above code can be repeated with other file objects, all associated with the completion port. A somewhat simplistic diagram of a completion port is depicted in figure 11-6. We'll see what bound threads are and how all this works in a moment.

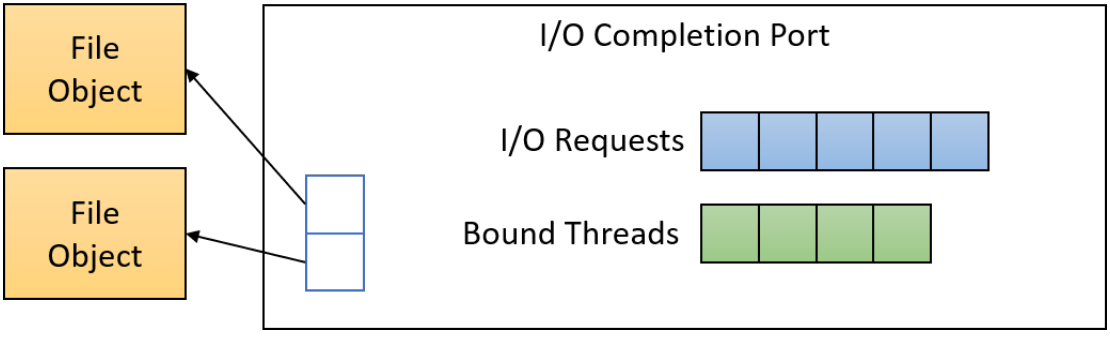


Figure 11-6: I/O Completion Port components

The I/O completion port's purpose is to allow processing of completed I/O operations by worker threads, where "worker" here can mean any threads that are bound to the completion port. A thread becomes bound to a completion port when it calls `GetQueuedCompletionStatus`:

```
BOOL GetQueuedCompletionStatus(
    _In_ HANDLE CompletionPort,
    _Out_ LPDWORD lpNumberOfBytesTransferred,
    _Out_ PULONG_PTR lpCompletionKey,
    _Out_ LPOVERLAPPED* lpOverlapped,
    _In_ DWORD dwMilliseconds);
```

The call puts the thread into a wait state until an asynchronous I/O operation initiated with one of the file objects associated with the completion port completes, or the timeout elapsed. Typically `dwMilliseconds` is set to `INFINITE`, meaning the thread has nothing to do until an I/O operation completes. If the operation that wakes up the thread completed successfully, `GetQueuedCompletionStatus` returns `TRUE` and the out parameters are filled with the number of bytes transferred, the completion key originally associated with the file handle and the `OVERLAPPED` structure pointer that was used for the request.

If some error occurres, the return value is `FALSE`, and `GetLastError` returns the error code. If the timeout is not infinite, the return value is still `FALSE` but `GetLastError` returns `WAIT_TIMEOUT` and the `OVERLAPPED` pointer is set to `NULL`.

Any number of threads can call `GetQueuedCompletionStatus` to wait until a completed packet arrives. The I/O completion port will not allow more than the maximum threads specified at the port's creation to succeed the call at the same time. However, if a thread for which `GetQueuedCompletionStatus` succeeded, and that thread, while processing the completion operation enters a wait state for whatever reason (`SuspendThread`, `WaitForSingleObject`. etc.), the completion port will allow another thread to have its `GetQueuedCompletionStatus` call end its wait. This means that periodically, the number of threads handling completion packets can be higher than the original specified maximum. However, only the maximum number of threads will be "runnable" - that is, not in a wait state.

Once a thread calls `GetQueuedCompletionStatus` the first time, it becomes bound to the completion port, until the thread exits, the completion port is closed, or the thread calls `GetQueuedCompletionStatus` on a different completion port.

If multiple threads wait for completion packets, the thread that gets the next one is the last one executed, i.e. it's a Last In First Out (LIFO) queue (technically a stack). This is beneficial in cases where the rate of completed operations is relatively low, allowing the same thread or few threads to do the processing. This benefits potentially less context switches, and definitely better use of CPU caches.

A thread can also request to dequeue multiple I/O completions with an extended version of `GetQueuedCompletionStatus`:

```
BOOL GetQueuedCompletionStatusEx(
    _In_  HANDLE CompletionPort,
    _Out_ LPOVERLAPPED_ENTRY lpCompletionPortEntries,
    _In_  ULONG ulCount,
    _Out_ PULONG ulNumEntriesRemoved,
    _In_  DWORD dwMilliseconds,
    _In_  BOOL fAlertable);
```

The function fills an array of `OVERLAPPED_ENTRY` structures, no more than `ulCount`. Each such structure looks like this:

```
typedef struct _OVERLAPPED_ENTRY {
    ULONG_PTR lpCompletionKey;
    LPOVERLAPPED lpOverlapped;
    ULONG_PTR Internal;
    DWORD dwNumberOfBytesTransferred;
} OVERLAPPED_ENTRY, *LPOVERLAPPED_ENTRY;
```

The structure includes the three output arguments for a single completion entry, provided for a single completion by `GetQueuedCompletionStatus`. The `Internal` member is just that, and should not be touched. Back to `GetQueuedCompletionStatusEx` - the number of actual entries returned is provided by the `ulNumEntriesRemoved` parameter. Also, the function allows the waiting to be in an alertable state, if so desired.

It is possible to manually post a completion packet to an I/O completion port by calling `PostQueuedCompletionStatus`, which makes these objects more generic, and not really just about I/O. This is exactly how the notifications work for a job object.

```
BOOL PostQueuedCompletionStatus(
    _In_ HANDLE CompletionPort,
    _In_ DWORD dwNumberOfBytesTransferred,
    _In_ ULONG_PTR dwCompletionKey,
    _In_opt_ LPOVERLAPPED lpOverlapped);
```

Aside from the completion port, the function accepts the three parameters that will be later extracted by `GetQueuedCompletionStatus(Ex)`. Usually, the completion key is used for distinguishing the type of notification. Alos, for non-I/O operation, there is little meaning for the `OVERLAPPED` structure, and so `NULL` is typically passed in that case.

The *Bulk Copy* Application

The *BuldCopy* application, whose main window is shown in figure 11-7 is an example of how to put all the pieces together to create an application that asynchronously copies multiple files, where each file can be customized to be copied to a selected destination.

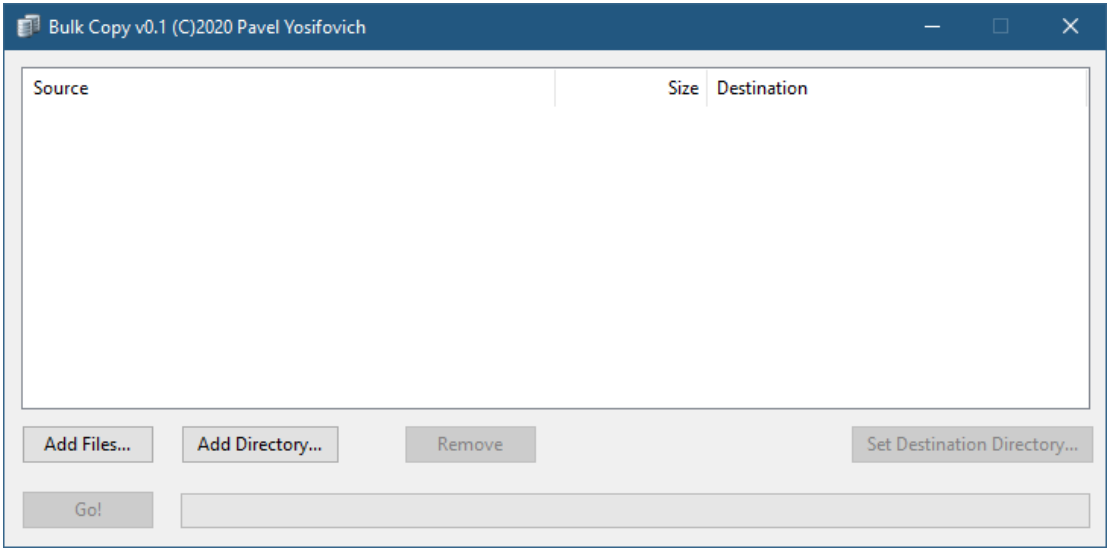


Figure 11-7: The initial window of the *Bulk Copy* application

Source files can be added with the *Add Files...* button (multiple files allowed). Then the *Set Destination Directory...* button is used to select a destination for the copy (for all source files or some of them). An example with three source files is shown in figure 11-8.

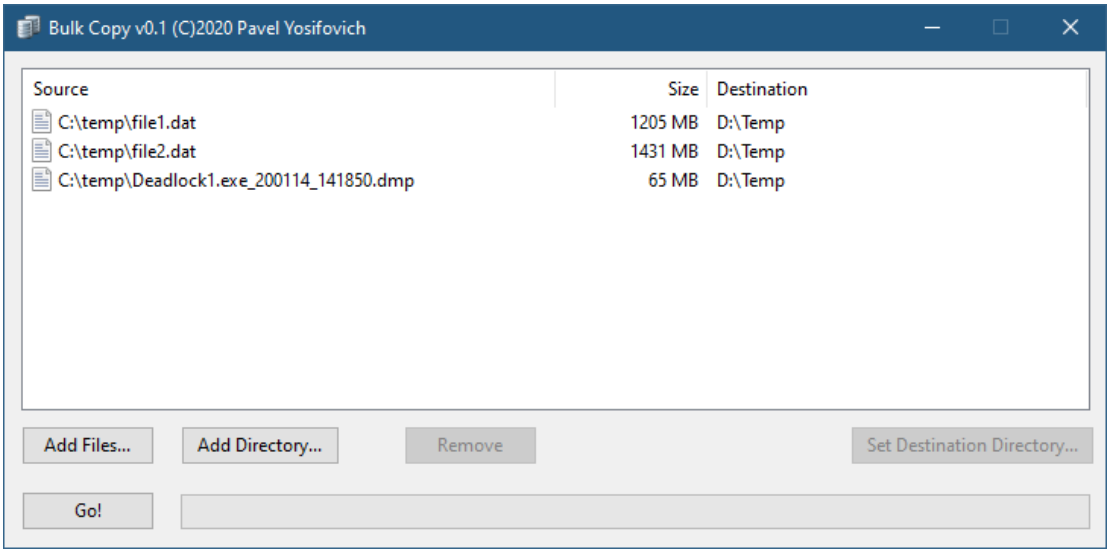


Figure 11-8: Files added in *Bulk Copy*

Now the `Go!` button becomes enabled, which allows performing the copy operations. A progress bar at the bottom of the dialog gives indication of, well, progress (figure 11-9).

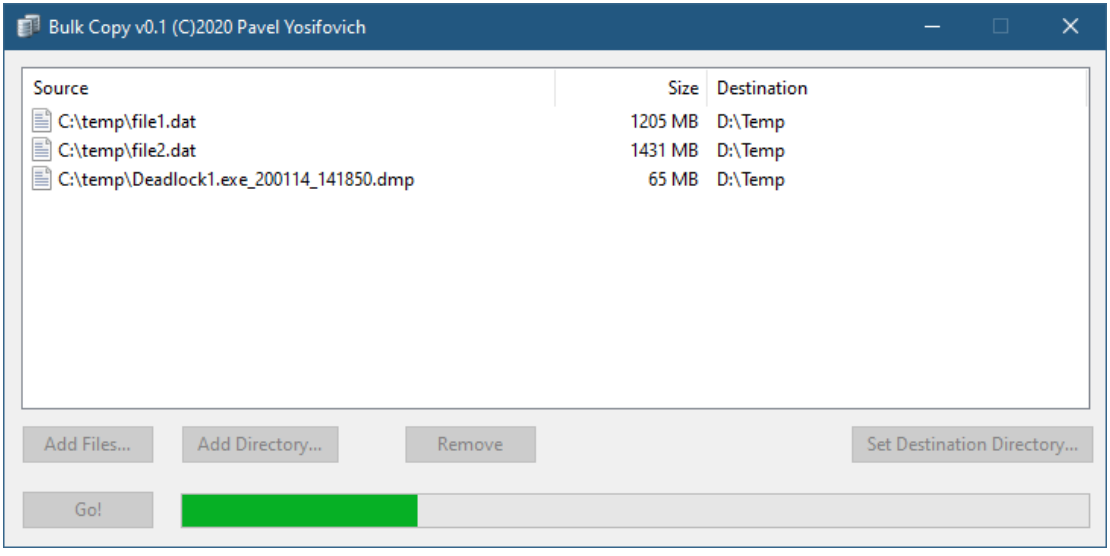


Figure 11-9: Copy operations in progress *Bulk Copy*

When all copying is done, the application shows an "All Done!" message box.

The *Add Directory...* button allows adding a directory, but the application does not implement copy operations for the files in the directory. This is left as an exercise for the reader.

Copying files is much more complex than merely reading from one file and writing to another. More elements actually need to be copied, such as the security descriptor and NTFS streams (see later in this chapter on NTFS streams).

Adding files to the list view is not particularly interesting, except the retrieval of the file size for each file. Here is the complete message handler:

```
LRESULT CMainDlg::OnAddFiles(WORD, WORD wID, HWND, BOOL&) {
    CMultiFileDialog dlg(nullptr, nullptr,
        OFN_FILEMUSTEXIST | OFN_ALLOWMULTISELECT,
        L"All Files (*.*)\0*.*\0", *this);
    dlg.ResizeFilenameBuffer(1 << 16);

    if (dlg.DoModal() == IDOK) {
        CString path;
        int errors = 0;
        dlg.GetFirstPathName(path);
        do {
            wil::unique_handle hFile(::CreateFile(path, 0,
                FILE_SHARE_READ, nullptr, OPEN_EXISTING,
                0, nullptr));
            if (!hFile) {
                errors++;
                continue;
            }
            LARGE_INTEGER size;
            ::GetFileSizeEx(hFile.get(), &size);
            int n = m_List.AddItem(m_List.GetItemCount(), 0, path, 0);
            m_List.SetItemText(n, 1, FormatSize(size.QuadPart));
            m_List.SetItemData(n, (DWORD_PTR)Type::File);
        } while (dlg.GetNextPathName(path));
        m_List.EnsureVisible(m_List.GetItemCount() - 1, FALSE);
        UpdateButtons();
        if (errors > 0)
            AtlMessageBox(*this, L"Some files failed to open",
                IDR_MAINFRAME, MB_ICONEXCLAMATION);
    }
    return 0;
}
```

First, a multi-file open dialog box is created and presented to the user. Each file is opened to get

its file size with `GetFileSizeEx`. Notice that the access mask provided is zero, because as noted earlier, `SYNCHRONIZE` and `FILE_READ_ATTRIBUTES` are always requested, and these attributes include the file size. The file is then added to the list view along with its size (formatted by a little helper function, `FormatSize`).

Setting the destination path(s) is not interesting from a Windows API perspective, as it's all UI-related. The real work starts once the *Go!* button is clicked.

Each source/destination pair, along with handles to these files is stored in a helper structure defined like so:

```
struct FileData {
    CString Src;
    CString Dst;
    wil::unique_handle hDst, hSrc;
};
```

To keep the handles alive while I/O is being processed, these structures are held in a member of the dialog class (in *MainDlg.h*):

```
std::vector<FileData> m_Data;
```

The first thing the Go button handler does, is constructing this vector without yet opening the files:

```
LRESULT CMainDlg::OnGo(WORD, WORD wID, HWND, BOOL&) {
    // transfer list data to vector
    m_Data.clear();
    int count = m_List.GetItemCount();
    m_Data.reserve(count);
    for (int i = 0; i < count; i++) {
        if (m_List.GetItemData(i) != (DWORD_PTR)Type::File) {
            // folders not yet implemented
            continue;
        }

        FileData data;
        m_List.GetItemText(i, 0, data.Src);
        m_List.GetItemText(i, 2, data.Dst);
        m_Data.push_back(std::move(data));
    }
```

The code extracts the file names from the list view and fills in the `FileData` structures, adding them to the vector.

The UI thread should not be bound to any I/O completion port, because that would cause the UI to be unresponsive while the thread waits for completion packets, so a new thread is created to serve the I/O completion port. Here is the rest of the `OnGo` function:

```
    // create a worker thread
    auto hThread = ::CreateThread(nullptr, 0, [](auto param) {
        return ((CMainDlg*)param)->WorkerThread();
        }, this, 0, nullptr);
    // error handling ommitted
    ::CloseHandle(hThread);

    // update UI state
    m_Progress.SetPos(0);
    m_Running = true;
    UpdateButtons();

    return 0;
}
```

The real job is performed by the `WorkerThread` function. Its first task is to create a new I/O completion port, unassociated with any file handles at this time:

```
DWORD CMainDlg::WorkerThread() {
    wil::unique_handle hCP(::CreateIoCompletionPort(
        INVALID_HANDLE_VALUE, nullptr, 0, 0));
    ATLASSERT(hCP);
    if (!hCP) {
        PostMessage(WM_ERROR, ::GetLastError());
        return 0;
    }
```

The various read and write operations that follow will be done in chunks, which are set to 64 KB (you can experiment with other chunk sizes).

```
const int chunkSize = 1 << 16;  // 64 KB
```

The I/O operations are architected as shown in figure 11-10.

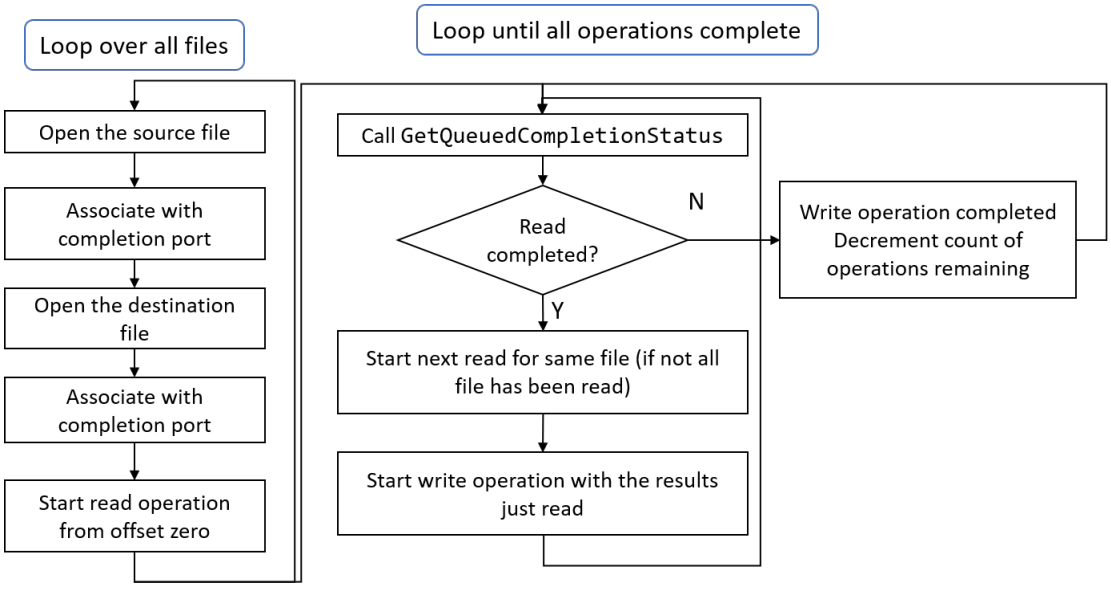


Figure 11-10: Handling I/O operations

First, we loop over all file pairs, and open each source and destination. For the source, its size is queried:

```
LONGLONG count = 0;
for (auto& data : m_Data) {
    // open source file for async I/O
    wil::unique_handle hSrc(::CreateFile(data.Src, GENERIC_READ,
        FILE_SHARE_READ, nullptr, OPEN_EXISTING, FILE_FLAG_OVERLAPPED,
        nullptr));
    if (!hSrc) {
        PostMessage(WM_ERROR, ::GetLastError());
        continue;
    }

    // get file size
    LARGE_INTEGER size;
    ::GetFileSizeEx(hSrc.get(), &size);
```

The destination file may or may not exist. We need to open or create it, and then set its final size:

```
// create the target file and set final size
CString filename = data.Src.Mid(data.Src.ReverseFind(L'\\'));
wil::unique_handle hDst(::CreateFile(data.Dst + filename, GENERIC_WRITE, 0,
    nullptr, OPEN_ALWAYS, FILE_FLAG_OVERLAPPED, nullptr));
if (!hDst) {
    PostMessage(WM_ERROR, ::GetLastError());
    continue;
}

::SetFilePointerEx(hDst.get(), size, nullptr, FILE_BEGIN);
::SetEndOfFile(hDst.get());
```

Extending the file now to its final size is important, because file extension is always done synchronously, so it's best to get it over with in one stroke.

Now we can associate both files with the completion port and save the handles in the `FileData` structure:

```
ATLVERIFY(hCP.get() == ::CreateIoCompletionPort(hSrc.get(), hCP.get(),
    (ULONG_PTR)Key::Read, 0));
ATLVERIFY(hCP.get() == ::CreateIoCompletionPort(hDst.get(), hCP.get(),
    (ULONG_PTR)Key::Write, 0));

data.hSrc = std::move(hSrc);
data.hDst = std::move(hDst);
```

A simple enum is used to identify the source file's completion key (`Key::Read`) vs. the destination file's completion key (`Key::Write`), since we need to know for each operation whether it's a read or write, and that's one easy way of propagating that information, since every file is used for read or write only.

`ATLVERIFY` is similar to `assert`, but is compiled in Release build as well as Debug. Using `assert` or `ATLASSERT` would remove the entire instruction from the compiled binary. These asserts just verify that adding to an existing completion port returns back the same handle.

Now it's time to initiate the first read request for the source file. We need some context information to be accessible for each operation. One trick we can use is to derive a class from `OVERLAPPED` and add any context we need. The pointer would be available after each successful call to `GetQueuedCompletionStatus` and we can then just cast to the full type. Here is the derived data structure (defined in *MainDlg.h*):

```
struct IOData : OVERLAPPED {
    HANDLE hSrc, hDst;
    std::unique_ptr<BYTE[]> Buffer;
    ULONGLONG Size;
};
```

We need the handles to source and destination files, the buffer to read or write, and the size of the files. This will allow us to determine whether the file is read in its entirety, or not.

With this structure in hand, we can build the first read operation:

```
    auto io = new IOData;
    io->Size = size.QuadPart;
    io->Buffer = std::make_unique<BYTE[]>(chunkSize);
    io->hSrc = data.hSrc.get();
    io->hDst = data.hDst.get();
    ::ZeroMemory(io, sizeof(OVERLAPPED));
    auto ok = ::ReadFile(io->hSrc, io->Buffer.get(), chunkSize, nullptr, io);
    ATLASSERT(!ok && ::GetLastError() == ERROR_IO_PENDING);
    count += (size.QuadPart + chunkSize - 1) / chunkSize;
}
```

The structure is allocated dynamically because it must survive until the operation completes. In this particular application we could have created these data structures statically because all is done in a single function, but I chose to use dynamic allocation to demonstrate this pattern that is necessary if the application would have been architected differently. The offsets inside the `OVERLAPPED` part of the structure are zeroed (start of file), the buffer allocated, the size is copied from the file size, and the operation is underway with `ReadFile` and the structure pointer.

Finally, the local `count` variable is updated with the number of chunks required to read (and write) the entire file. This will help in determining when all operations are done.

At this point, a number of read operations are underway, based on the number of files. Now it's time to wait for I/O completion notifications and act accordingly:

```
PostMessage(WM_PROGRESS_START, count);  // update UI

while (count > 0) {
    DWORD transferred;
    ULONG_PTR key;
    OVERLAPPED* ov;
    BOOL ok = ::GetQueuedCompletionStatus(hCP.get(), &transferred, &key,
        &ov, INFINITE);
    if (!ok) {
        PostMessage(WM_ERROR, ::GetLastError());
        count--;
        delete ov;
        continue;
    }
```

Once `GetQueuedCompletionStatus` returns successfully, we need to examine the completed packet:

```
    // get actual data object
    auto io = static_cast<IOData*>(ov);

    if (key == (DWORD_PTR)Key::Read) {
        // check if need another read
        ULARGE_INTEGER offset = { io->Offset, io->OffsetHigh };
        offset.QuadPart += chunkSize;
        if (offset.QuadPart < io->Size) {
            auto newio = new IOData;
            newio->Size = io->Size;
            newio->Buffer = std::make_unique<BYTE[]>(chunkSize);
            newio->hSrc = io->hSrc;
            newio->hDst = io->hDst;
            ::ZeroMemory(newio, sizeof(OVERLAPPED));
            newio->Offset = offset.LowPart;
            newio->OffsetHigh = offset.HighPart;
            auto ok = ::ReadFile(newio->hSrc, newio->Buffer.get(), chunkSize,
                nullptr, newio);
            auto error = ::GetLastError();
            ATLASSERT(!ok && error == ERROR_IO_PENDING);
        }

        // read done, initiate write to the same offset in the target file
```

```
            // offset is the same, just a different file
            io->Internal = io->InternalHigh = 0;
            ok = ::WriteFile(io->hDst, io->Buffer.get(), transferred, nullptr, ov);
            auto error = ::GetLastError();
            ATLASSERT(!ok && error == ERROR_IO_PENDING);
        }
        else {
            // write operation complete
            count--;
            delete io;
            PostMessage(WM_PROGRESS);
        }
    }
```

If a read completes, we check if another read is needed, and if so, a new `IOData` object is allocated and filled appropriately with the next chunk in the file. Because a read was completed, a write operation is initiated for writing the returned buffer to the destination file.

If it was a write that completed, we decrement the count of operations remaining, free the `IOData` object, and send an update message to the UI. This loop continues until all I/O operations are complete.

Lastly, the UI can be updated, all vector of file data cleared (closing the handles to all the files), and the thread can exit gracefully, while also closing the I/O completion port.

Currently, read operations are constructed twice: the first read and all the rest. Use `PostQueuedCompletionStatus` to post a custom notification so that the initial read is constructed just as subsequent reads.

Add an option to limit the number of concurrent I/O operations that are in progress. Currently, this number is based on the number of files, which could be very big.

Using the Thread Pool for I/O Completion

In chapter 9, we've seen the use and benefits of thread pools. One set of functions we omitted was related to I/O operations. Now it's time to fill in this gap. In the *Bulk Copy* application, we create a dedicated thread that called `GetQueuedCompletionStatus`, and handled I/O completions. This service is also offered by the thread pool, so no explicit thread needs to be created, and the scaling of the thread pool can be used to have more than one thread handling completions. To get things rolling, call `CreateThreadpoolIo` to create a completion port behind the covers and associate it with a file handle:

```
PTP_IO CreateThreadpoolIo(
    _In_ HANDLE hFile,
    _In_ PTP_WIN32_IO_CALLBACK pfnio,
    _Inout_opt_ PVOID pv,
    _In_opt_ PTP_CALLBACK_ENVIRON pcbe);
```

`hFile` is the file handle (already open for asynchronous I/O) to be associated with the internal I/O completion port. `pfnio` is a callback that is invoked by a thread pool thread whenever the internal call to `GetQueuedCompletionStatus` returns. The `pv` parameter is an application-defined value passed as-is to the callback function. Finally, `pcbe` is an optional callback environment, described in chapter 9. The function returns an opaque pointer to the I/O thread pool object.

The callback must have the following prototype:

```
typedef VOID (WINAPI *PTP_WIN32_IO_CALLBACK)(
    _Inout_     PTP_CALLBACK_INSTANCE Instance,
    _Inout_opt_ PVOID                 Context,
    _Inout_opt_ PVOID                 Overlapped,
    _In_        ULONG                 IoResult,
    _In_        ULONG_PTR             NumberOfBytesTransferred,
    _Inout_     PTP_IO                Io);
```

The function provides the standard instance value (`Instance` parameter) as other thread pool callbacks, as well as the context provided to `CreateThreadpoolIo`. The next three arguments are related to the I/O operation: The `OVERLAPPED` pointer, the result code (`ERROR_SUCCESS` if all goes well) and the number of bytes transferred. The last parameter is the I/O thread pool object that was returned from `CreateThreadpoolIo`. As we shall soon see, it's convenient to it here.

To get the thread pool completion infrastructure going, it is necessary to call `StartThreadpoolIo` before every asynchronous operation:

```
VOID StartThreadpoolIo(_Inout_ PTP_IO pio);
```

If a `ReadFile` or `WriteFile` call returns an error (return value is `FALSE` and `GetLastError` is not `ERROR_IO_PENDING`), the thread pool I/O must be cancelled with `CancelThreadpoolIo`:

```
VOID CancelThreadpoolIo(_Inout_ PTP_IO pio);
```

Similarly to other thread pool APIs we met, a thread can wait and/or cancel pending I/O operations:

```
VOID WaitForThreadpoolIoCallbacks(
    _Inout_ PTP_IO pio,
    _In_ BOOL fCancelPendingCallbacks
    );
```

Finally, the thread pool I/O object needs to be closed:

```
VOID CloseThreadpoolIo(_Inout_ PTP_IO pio);
```

The *Bulk Copy 2* Application

The *BulkCopy2* project is identical to the *BulkCopy* project in terms of functionality, but it uses the thread pool to respond to I/O completions. In this section, we'll look at the code changes to make it work.

First, since we are using the thread pool, creating a dedicated thread is unnecessary - that's what the thread pool is for. The `OnGO` function that responds to the Go button click, calls a function named `StartCopy` to initiate the copy operations. It starts by iterating over all file pairs, opening the files, and setting the final size in the destination:

```
void CMainDlg::StartCopy() {
    m_OperationCount = 0;

    for (auto& data : m_Data) {
        // open source file for async I/O
        wil::unique_handle hSrc(::CreateFile(data.Src, GENERIC_READ, FILE_SHARE\
_READ,
            nullptr, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, nullptr));
        if (!hSrc) {
            PostMessage(WM_ERROR, ::GetLastError());
            continue;
        }

        // get file size
        LARGE_INTEGER size;
        ::GetFileSizeEx(hSrc.get(), &size);

        // create target file and set final size
        CString filename = data.Src.Mid(data.Src.ReverseFind(L'\\'));
        wil::unique_handle hDst(::CreateFile(data.Dst + filename, GENERIC_WRITE\
, 0,
```

```
            nullptr, OPEN_ALWAYS, FILE_FLAG_OVERLAPPED, nullptr));
        if (!hDst) {
            PostMessage(WM_ERROR, ::GetLastError());
            continue;
        }

        ::SetFilePointerEx(hDst.get(), size, nullptr, FILE_BEGIN);
        ::SetEndOfFile(hDst.get());
```

We are not creating an I/O completion port explicitly. Instead, we are using the thread pool to create two thread pool I/O objects and associate them with the two files. The `FileData` structure has been extended to store these handles:

```
struct FileData {
    CString Src;
    CString Dst;
    wil::unique_handle hDst, hSrc;
    wil::unique_threadpool_io tpSrc, tpDst;
};
```

Notice the use of `wil::unique_threadpool_io` that will call `CloseThreadpoolIo` when the object goes out of scope.

Continuing inside `StartCopy`, we create the thread pool I/O objects:

```
data.tpDst.reset(::CreateThreadpoolIo(hDst.get(), WriteCallback,
    this, nullptr));
data.tpSrc.reset(::CreateThreadpoolIo(hSrc.get(), ReadCallback,
    data.tpDst.get(), nullptr));
```

For the write operations, `this` is passed as the context argument. For the read operations, the write I/O pool object is passed in. We'll see why this is required when we implement the read and write callbacks. The last piece of business in this loop is starting the first read operation, very similarly to the *Bulk Copy* application, with the addition of `StartThreadpoolIo` to get the thread pool I/O mechanism going:

```
        data.hSrc = std::move(hSrc);
        data.hDst = std::move(hDst);

        // initiate first read operation
        auto io = new IOData;
        io->Size = size.QuadPart;
        io->Buffer = std::make_unique<BYTE[]>(chunkSize);
        io->hSrc = data.hSrc.get();
        io->hDst = data.hDst.get();
        ::ZeroMemory(io, sizeof(OVERLAPPED));

        ::StartThreadpoolIo(data.tpSrc.get());
        auto ok = ::ReadFile(io->hSrc, io->Buffer.get(), chunkSize, nullptr, io\
);

        ATLASSERT(!ok && ::GetLastError() == ERROR_IO_PENDING);
        ::InterlockedAdd64(&m_OperationCount,
            (size.QuadPart + chunkSize - 1) / chunkSize);
    }

    PostMessage(WM_PROGRESS_START, (WPARAM)m_OperationCount);
}
```

`StartCopy` runs quickly, starting read operations for all files and then returns to pumping UI messages. The rest of the work is done by the two static callbacks registered in `CreateThreadpoolIo`. Here is the read callback:

```
void CMainDlg::ReadCallback(PTP_CALLBACK_INSTANCE Instance, PVOID Context,
    PVOID Overlapped, ULONG IoResult, ULONG_PTR Transferred, PTP_IO Io) {
    if (IoResult == ERROR_SUCCESS) {
        auto io = static_cast<IOData*>(Overlapped);
        ULARGE_INTEGER offset = { io->Offset, io->OffsetHigh };
        offset.QuadPart += chunkSize;
        if (offset.QuadPart < io->Size) {
            auto newio = new IOData;
            newio->Size = io->Size;
            newio->Buffer = std::make_unique<BYTE[]>(chunkSize);
            newio->hSrc = io->hSrc;
            newio->hDst = io->hDst;
            ::ZeroMemory(newio, sizeof(OVERLAPPED));
            newio->Offset = offset.LowPart;
```

```
            newio->OffsetHigh = offset.HighPart;
            ::StartThreadpoolIo(Io);
            auto ok = ::ReadFile(newio->hSrc, newio->Buffer.get(), chunkSize,
                nullptr, newio);
            auto error = ::GetLastError();
            ATLASSERT(!ok && error == ERROR_IO_PENDING);
        }

        // read done, initiate write to the same offset in the target file
        io->Internal = io->InternalHigh = 0;
        auto writeIo = (PTP_IO)Context;
        ::StartThreadpoolIo(writeIo);
        auto ok = ::WriteFile(io->hDst, io->Buffer.get(),
            (ULONG)Transferred, nullptr, io);
        auto error = ::GetLastError();
        ATLASSERT(!ok && error == ERROR_IO_PENDING);
    }
}
```

The code is very similar to the original application for handling read completions. Calling `StartThreadpoolIo` is mandatory before initiating new requests, which shows why the last parameter (`PTP_IO`) is convenient to have. Since ending a read operation needs to start a write operation, the context that was passed in is the `PTP_IO` object of the destination, allowing calling the correct `StartThreadpoolIo` for write. As an alternative, the thread pool I/O objects could just have easily be tucked in the `IOData` structure.

The write callback is simpler, as it just needs to update the operation count (in a thread-safe manner), and free the completed operation:

```
void CMainDlg::WriteCallback(PTP_CALLBACK_INSTANCE Instance, PVOID Context,
    PVOID Overlapped, ULONG IoResult, ULONG_PTR Transferred, PTP_IO Io) {
    if (IoResult == ERROR_SUCCESS) {
        auto pThis = static_cast<CMainDlg*>(Context);
        pThis->PostMessage(WM_PROGRESS);
        auto io = static_cast<IOData*>(Overlapped);
        delete io;
        if (0 == InterlockedDecrement64(&pThis->m_OperationCount)) {
            pThis->PostMessage(WM_DONE);
        }
    }
}
```

I/O Cancellation

Once an I/O operation is underway, how can it be canceled? The Windows API provides some options in this regard.
The obvious need for I/O cancellation is related to asynchronous operations. For that, there are two functions:

```
BOOL CancelIo(_In_ HANDLE hFile);

BOOL CancelIoEx(
    _In_ HANDLE hFile,
    _In_opt_ LPOVERLAPPED lpOverlapped);
```

`CancelIo` attempts to cancel all asynchronous operations initiated through the provided file handle by the calling thread. For more fine-grained control, `CancelIoEx` can be used with a specific `OVERLAPPED` structure representing the operation to cancel.

In any case, canceling an I/O operation is not guaranteed to succeed. The canceling operation itself is implemented by the device driver responsible for the operation. Some drivers (especially for devices) don't support cancellation at all. Even if the driver supports cancelation, it may not be able to do so for every operation. For example, if an operation is currently being processed by the hardware, it may be too late to cancel. You should think of the cancel APIs as requesting cancellation, with no guarantees.

I/O operations are also canceled in the following scenarios:

- When a file handle is closed, all pending I/O operations are canceled (except if the file handle is associated with a completion port).
- When a thread exits, all pending I/O operations issued by the thread are canceled, except requests made to file handles that are associated with I/O completion ports.

If an I/O operation is canceled successfully, the return value from `GetLastError` (or the error result provided by a thread pool callback) is `ERROR_OPERATION_ABORTED`.

What about synchronous operations? Clearly, the thread that initiated the request cannot cancel it, since it's waiting for the I/O to complete. A different thread, can, however, attempt cancelation with `CancelSynchronousIo`:

```
BOOL CancelSynchronousIo(_In_ HANDLE hThread);
```

The thread handle must have the `PROCESS_TERMINATE` access mask. As stated before, cancelation is not guaranteed. If it is canceled, the wait of the original thread completes, the operation returns `FALSE` and `GetLastError` returns `ERROR_OPERATION_ABORTED`.

Devices

Working with devices (that is, non file-system files), is essentially no different than working with file-system files. The `ReadFile` and `WriteFile` functions work for any device, including asynchronously, although not all devices support read and write operations. For devices in particular, there is yet another function for performing I/O operations - `DeviceIoControl`:

```
BOOL DeviceIoControl(
    _In_  HANDLE hDevice,
    _In_  DWORD dwIoControlCode,
    _In_  LPVOID lpInBuffer,
    _In_  DWORD nInBufferSize,
    _Out_ LPVOID lpOutBuffer,
    _In_ DWORD nOutBufferSize,
    _Out_opt_ LPDWORD lpBytesReturned,
    _Inout_opt_ LPOVERLAPPED lpOverlapped);
```

`DeviceIoControl` is a general-purpose function that allows sending a request, defined by a control code (`dwIoControlCode`) with optional two buffers, one designated "input" and the other "output". The function returns the number of bytes written to the output buffer (if any) (in `lpBytesReturned`) and accepts an optional `OVERLAPPED` structure if the request is to be performed asynchronously.

As an example, consider the idea of a sparse file. A sparse file should contain mostly zeros, so the file system can store it in less space than just storing all bytes normally. Compression can provide a similar effect, but these formats are not the same. To transform a file to become sparse, a `DeviceIoControl` call is needed with the `FSCTL_SET_SPARSE` I/O control code. The documentation for this control code indicates what should the input and output buffer contain. In the case of `FSCTL_SET_SPARSE`, the input buffer should point to the following structure:

```
typedef struct _FILE_SET_SPARSE_BUFFER {
    BOOLEAN SetSparse;
} FILE_SET_SPARSE_BUFFER;
```

A very simple structure, indicating whether to turn on or off the sparse file feature. There is no output buffer for this operation. Making a file sparse can be done like so:

```
FILE_SET_SPARSE_BUFFER buffer;
buffer.SetSparse = TRUE;
DWORD bytes;
::DeviceIoControl(hFile, FSCTL_SET_SPARSE, &buffer, sizeof(buffer),
    nullptr, 0, &bytes, nullptr);
```

Once a file is sparse, zeros must be written explicitly with another control code, `FSCTL_SET_ZERO_DATA`, like so:

```
FILE_ZERO_DATA_INFORMATION buffer;
buffer.FileOffset.QuadPart = 100;
buffer.BeyondFinalZero.QuadPart = 1 << 20;
::DeviceIoControl(hFile, FSCTL_SET_ZERO_DATA, &buffer, sizeof(buffer),
    nullptr, 0, &bytes, nullptr);
```

Many other standard control codes exist for various types of devices, check out the documentation for more information.

`CreateFile` works with any symbolic link, those shown in figures 11-3 and 11-4. For example, there are symbolic links called "PhysicalDrive0" and perhaps others, which is a way to open a drive's sectors directly, without looking at it through the file system's lens.

The *DumpDrive* application displays raw bytes from a disk, starting from a desired sector. The `main` function starts by parsing command-line arguments:

```
int main(int argc, const char* argv[]) {
    if (argc < 4) {
        printf("Usage: DumpDrive <index> <offset in sectors> <size in sectors>\\
n");
        return 0;
    }

    WCHAR path[] = L"\\\\.\\PhysicalDriveX";
    path[::wcslen(path) - 1] = argv[1][0];

    auto offset = atoll(argv[2]) * 512;
    auto size = atol(argv[3]) * 512;
```

The offset and size must be a multiple of a sector's size. Otherwise, later `ReadFile` calls will fail with `ERROR_INVALID_PARAMETER`. The above code assumes 512 bytes per sector. It's better not to assume, and get the actual size programmatically. The exercises at the end of this section will point you in the right direction.

Next, we need to open the drive, move the file pointer to the required offset and perform the read:

```
    HANDLE hDevice = ::CreateFile(path, GENERIC_READ, FILE_SHARE_READ
        | FILE_SHARE_WRITE, nullptr, OPEN_EXISTING, 0, nullptr);
    if (hDevice == INVALID_HANDLE_VALUE)
        return Error("Failed to open Physical drive");

    LARGE_INTEGER fp;
    fp.QuadPart = offset;
    if (!::SetFilePointerEx(hDevice, fp, nullptr, FILE_BEGIN))
        return Error("Failed in SetFilePointerEx");

    auto buffer = std::make_unique<BYTE[]>(size);
    DWORD bytes;
    if (!::ReadFile(hDevice, buffer.get(), size, &bytes, nullptr))
        return Error("Failed to read data");

    DisplayData(offset, buffer.get(), bytes);

    ::CloseHandle(hDevice);

    return 0;
}
```

The fact that we're working with a device rather than a file-system file does not change the fundamental way we write code. The path to `CreateFile` is what's important. `DisplayData` is a simple function that dumps hex bytes to the console:

```
void DisplayData(long long offset, const BYTE* buffer, DWORD bytes) {
    const int bytesPerLine = 16;
    for (DWORD i = 0; i < bytes; i += bytesPerLine) {
        printf("%16X: ", offset + i);
        for (int b = 0; b < bytesPerLine; b++) {
            printf("%02X ", buffer[i + b]);
        }
        printf("\n");
    }
}
```

Here is a truncated example running it on my physical drive 1:

```
c:\>DumpDrive 1 0 2
   0: 33 C0 8E D0 BC 00 7C FB 50 07 50 1F FC BE 1B 7C
  10: BF 1B 06 50 57 B9 E5 01 F3 A4 CB BD BE 07 B1 04
  20: 38 6E 00 7C 09 75 13 83 C5 10 E2 F4 CD 18 8B F5
  30: 83 C6 10 49 74 19 38 2C 74 F6 A0 B5 07 B4 07 8B
  40: F0 AC 3C 00 74 FC BB 07 00 B4 0E CD 10 EB F2 88
  50: 4E 10 E8 46 00 73 2A FE 46 10 80 7E 04 0B 74 0B
  60: 80 7E 04 0C 74 05 A0 B6 07 75 D2 80 46 02 06 83
  70: 46 08 06 83 56 0A 00 E8 21 00 73 05 A0 B6 07 EB
...
 120: 32 E4 8A 56 00 CD 13 EB D6 61 F9 C3 49 6E 76 61
 130: 6C 69 64 20 70 61 72 74 69 74 69 6F 6E 20 74 61
 140: 62 6C 65 00 45 72 72 6F 72 20 6C 6F 61 64 69 6E
 150: 67 20 6F 70 65 72 61 74 69 6E 67 20 73 79 73 74
...
 3E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 3F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Add code to dynamically determine the size of a sector. Use the `IOCTL_DISK_GET_DRIVE_GEOMETRY` control code to query the disk for its geometry.

Other uses of symbolic links are for "software drivers", those that do not manage any hardware, but need to do things that cannot be done in user mode. A canonical example is the driver for *Process Explorer*, that must expose a symbolic link, so that *Process Explorer* itself (the driver's client) can open a handle to the device and make `DeviceIoControl` calls to it, requesting various services, based on a communication protocol established by the driver and known to *Process Explorer*.

If you run *Process Explorer* at least once with admin rights, you'll find the symbolic link name "ProcExp152" using tools such as *WinObj* or *ObjectExplorer*. This means *Process Explorer* opens a handle to its device using code like the following:

```
HANDLE hDevice = ::CreateFile(L"\\\\.\\ProcExp152", GENERIC_READ | GENERIC_WRIT\
E, 0, nullptr,
    OPEN_EXISTING, 0, nullptr);
```

And then calls `DeviceIoControl` when needed.

My own tool, *ObjectExplorer*, also uses a kernel driver, with a symbolic link named "KObjExp". I use a similar `CreateFile` call to communicate with my device driver.

Another interesting set of symbolic links look like nonsense strings that could not have been selected by humans; indeed, these were generated by the kernel to (at least) ensure uniqueness. These weird-looking symbolic links are used for hardware device names. For example, if you want to access a camera connected to your computer, how would you do it? There is no symbolic link called "Camera1" or something similar, since this kind of string has a few limitations:

- What if there are two cameras or more?
- Devices can be plugged out and then back in - do the numeric values persist somehow?
- Is English special in any way - why the word "Camera"? Every driver can come up with its own name.
- How can camera devices be enumerated if names can be anything?
- Some devices may have multiple "personalities". For example, a printer device may also be a scanner.

> Some classic names from the DOS era are still maintained for compatibility, such as PRN, LPT, COM, NUL and so on.

Behind the scenes, devices expose *device interfaces*, which you can think of as being similar to software interfaces. Each interface represents some functionality. For example, a printer device can "implement" a printing interface and a scanning interface. With these interfaces, you can search "printers" or "scanners".

Device interfaces are represented by GUIDs, and many are defined by Microsoft and can be found in the documentation. This means we need to use an API to locate a device, and part of the information returned by the API, is the device's symbolic link that we can pass as-is to `CreateFile`.

The *EnumDevices* application shows an example of how to enumerate devices based on a device interface, and locate the device symbolic link. The heart of the application is the `EnumDevices` function, that accepts a `GUID` for the requested device interface, and performs the enumeration. Each device information is returned in the following structure:

```
struct DeviceInfo {
    std::wstring SymbolicLink;
    std::wstring FriendlyName;
};
```

Device enumeration begins by building a *device information set* (infoset) with `SetupDiGetClassDevs`:

```
std::vector<DeviceInfo> EnumDevices(const GUID& guid) {
    std::vector<DeviceInfo> devices;

    auto hInfoSet = ::SetupDiGetClassDevs(&guid, nullptr, nullptr,
        DIGCF_PRESENT | DIGCF_INTERFACEDEVICE);
    if (hInfoSet == INVALID_HANDLE_VALUE)
        return devices;
```

> Detailed discussion of the `SetupDi*` APIs is beyond the scope of this book.

"SetupDi" is short for "Setup Device Interface".

The first parameter to `SetupDiGetClassDevs` is the device GUID, which requires the last flags parameter to include `DIGCF_INTERFACEDEVICE`. The other flag specified (`DIGCF_PRESENT`) indicates that only connected devices should be enumerated.

Once an infoset is created, it can be enumerated with several enumeration functions, in this case what we need is `SetupDiEnumDeviceInterfaces`. If it returns `FALSE`, it means there are no more devices (or some other error occurred):

```
devices.reserve(4);

SP_INTERFACE_DEVICE_DATA data = { sizeof(data) };
SP_DEVINFO_DATA ddata = { sizeof(ddata) };
BYTE buffer[1 << 12];
for (DWORD i = 0; ; i++) {
    if (!::SetupDiEnumDeviceInterfaces(hInfoSet, nullptr, &guid, i, &data))
        break;
```

The enumeration returns a `SP_INTERFACE_DEVICE_DATA` structure that can be used to query for the symbolic link:

```
if (::SetupDiGetDeviceInterfaceDetail(hInfoSet, &data, details,
    sizeof(buffer), nullptr, &ddata)) {
    DeviceInfo info;
    info.SymbolicLink = details->DevicePath;
```

Finally, we can get a "friendly name" for the device, and add the device to the vector:

```
            if(::SetupDiGetDeviceRegistryProperty(hInfoSet, &ddata,
                SPDRP_DEVICEDESC, nullptr, buffer, sizeof(buffer), nullptr))
                info.FriendlyName = (WCHAR*)buffer;

            devices.push_back(std::move(info));
        }
    }
    ::SetupDiDestroyDeviceInfoList(hInfoSet);

    return devices;
}
```

The `DisplayDevices` function takes a collection of `DeviceInfo` instances, displays the information, and attempts to open a handle with `CreateFile`:

```
void DisplayDevices(const std::vector<DeviceInfo>& devices, const char* name) {
    printf("%s\n%s\n", name, std::string(::strlen(name), '-').c_str());
    for (auto& di : devices) {
        printf("Symbolic link: %ws\n", di.SymbolicLink.c_str());
        printf("  Name: %ws\n", di.FriendlyName.c_str());
        auto hDevice = ::CreateFile(di.SymbolicLink.c_str(), GENERIC_READ,
            FILE_SHARE_READ | FILE_SHARE_WRITE,
            nullptr, OPEN_EXISTING, 0, nullptr);
        if (hDevice == INVALID_HANDLE_VALUE)
            printf("  Failed to open device (%d)\n", ::GetLastError());
        else {
            printf("  Device opened successfully!\n");
            ::CloseHandle(hDevice);
        }
    }
    printf("\n");
}
```

The `main` function uses some GUIDs from various header files to enumerate some types of devices:

```
#define INITGUID
#include <Wiaintfc.h>
#include <Ntddvdeo.h>
#include <devpkey.h>
#include <Ntddkbd.h>

int main() {
    auto devices = EnumDevices(GUID_DEVINTERFACE_IMAGE);
    DisplayDevices(devices, "Image");

    // now in one stroke
    DisplayDevices(EnumDevices(GUID_DEVINTERFACE_MONITOR), "Monitor");
    DisplayDevices(EnumDevices(GUID_DEVINTERFACE_DISPLAY_ADAPTER),
        "Display Adapter");
    DisplayDevices(EnumDevices(GUID_DEVINTERFACE_DISK), "Disk");
    DisplayDevices(EnumDevices(GUID_DEVINTERFACE_KEYBOARD), "keyboard");

    return 0;
}
```

Here is an example run (truncated) on my machine:

```
...
Monitor
-------
Symbolic link: \\?\display#deld06e#4&5dd6935&0&uid200195#{e6f07b5f-ee97-4a90-b0\
76-33f57bf4eaa7}
  Name: Generic PnP Monitor
  Device opened successfully!
Symbolic link: \\?\display#deld070#4&5dd6935&0&uid208387#{e6f07b5f-ee97-4a90-b0\
76-33f57bf4eaa7}
  Name: Generic PnP Monitor
  Device opened successfully!

Display Adapter
---------------
Symbolic link: \\?\pci#ven_8086&dev_3e9b&subsys_09261028&rev_02#3&11583659&0&10\
#{5b45201d-f2f2-4f3b-85bb-30ff1f953599}
  Name: Intel(R) UHD Graphics 630
  Failed to open device (5)
Symbolic link: \\?\pci#ven_10de&dev_1f36&subsys_09261028&rev_a1#4&13a74b11&0&00\
```

```
08#{5b45201d-f2f2-4f3b-85bb-30ff1f953599}  Name: NVIDIA Quadro RTX 3000
  Failed to open device (5)
Symbolic link: \\?\root#basicdisplay#0000#{5b45201d-f2f2-4f3b-85bb-30ff1f953599}
  Name: Microsoft Basic Display Driver
  Failed to open device (5)

Disk
----
Symbolic link: \\?\scsi#disk&ven_nvme&prod_pm981a_nvme_sams#4&9bd8d03&0&020000#\
{53f56307-b6bf-11d0-94f2-00a0c91efb8b}
  Name: Disk drive
  Device opened successfully!
Symbolic link: \\?\usbstor#disk&ven_wd&prod_elements_10b8&rev_1012#575836314134\
344e39393230&0#{53f56307-b6bf-11d0-94f2-00a0c91efb8b}
  Name: Disk drive
  Device opened successfully!

keyboard
--------
Symbolic link: \\?\hid#vid_1532&pid_021e&mi_01&col01#9&5ed78c5&0&0000#{884b96c3\
-56ef-11d1-bc8c-00a0c91405dd}
  Name: Razer Ornata Chroma
  Failed to open device (5)
Symbolic link: \\?\hid#vid_044e&pid_1212&col01&col02#7&1551398c&0&0001#{884b96c\
3-56ef-11d1-bc8c-00a0c91405dd}
  Name: HID Keyboard Device
  Failed to open device (5)
...
```

> The prefix "\\?\" is identical to "\\.\".

Pipes and Mailslots

Two types of devices are worth mentioning in this section - pipes and mailslots. Pipes is uni- or bi-directional (also referred to as half-duplex and full-duplex) communication mechanism, that works across processes and across machines on the network. Mailslots is a uni-directional communication mechanism, that works locally or over the network.

You can view existing pipes and mailslots with *Object Explorer*. Select *Pipes...* or *Mailslots...* from the *Objects* menu. There are a lot of open pipes on a typical system (figure 11-11).

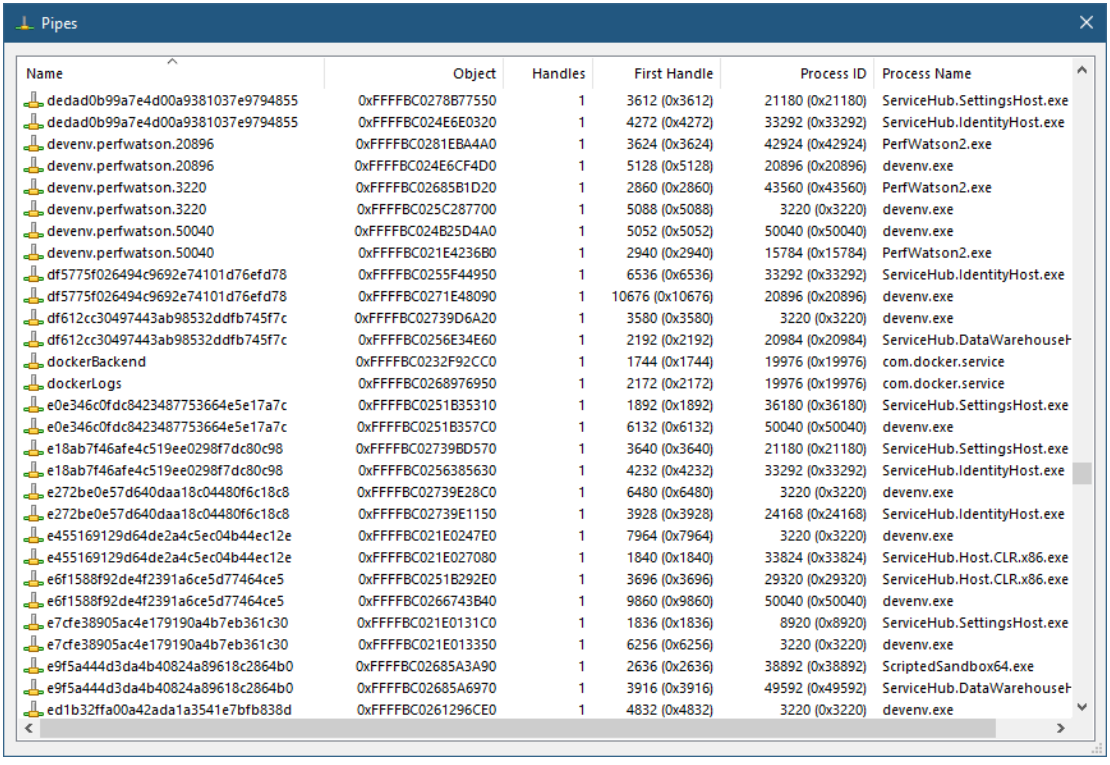


Figure 11-11: Pipes on a system (*Object Explorer*)

Table 11-1 showed an example of a path related to a named pipe and to a mailslot. The `CreateFile` function is used by a named pipe/mailslot client. For the server endpoint, other functions need to be used.

Pipes

Pipes come in two variants - anonymous and named. Anonymous pipes is a simple uni-direction communication mechanism that is limited to the local machine. An anonymous pipe pair is created with `CreatePipe`:

```
BOOL CreatePipe(
    _Out_ PHANDLE hReadPipe,
    _Out_ PHANDLE hWritePipe,
    _In_opt_ LPSECURITY_ATTRIBUTES lpPipeAttributes,
    _In_ DWORD nSize);
```

`CreatePipe` creates handles for the two ends of the pipe. A classic example of using anonymous pipes is to redirect input and/or output to another process. This allows one process to feed data to another process, while the other process has no idea, and doesn't really care, it just uses the standard handle(s) for input/output.

The *SimpleRedirect* application shows an example of redirecting the output handle to the *EnumDevices* application from the previous section. Instead of the output of *EnumDevices* going to its console, it will go to the *SimpleRedirect* process.

The application window is a simple dialog with a big edit box (figure 11-12). Clicking *Create and Redirect* creates the pipes, the *EnumDevices* process and performs the redirection. The result is the text written by that process (figure 11-13).

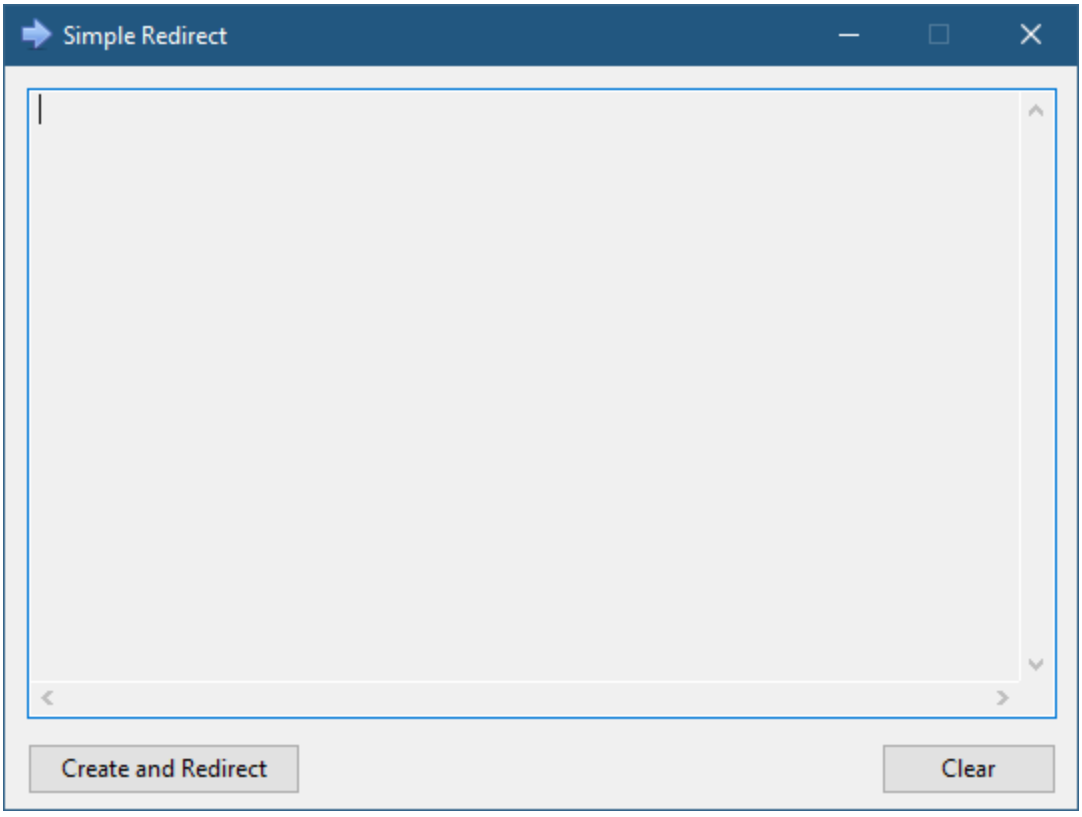


Figure 11-12: ***Simple Redirect*** at launch

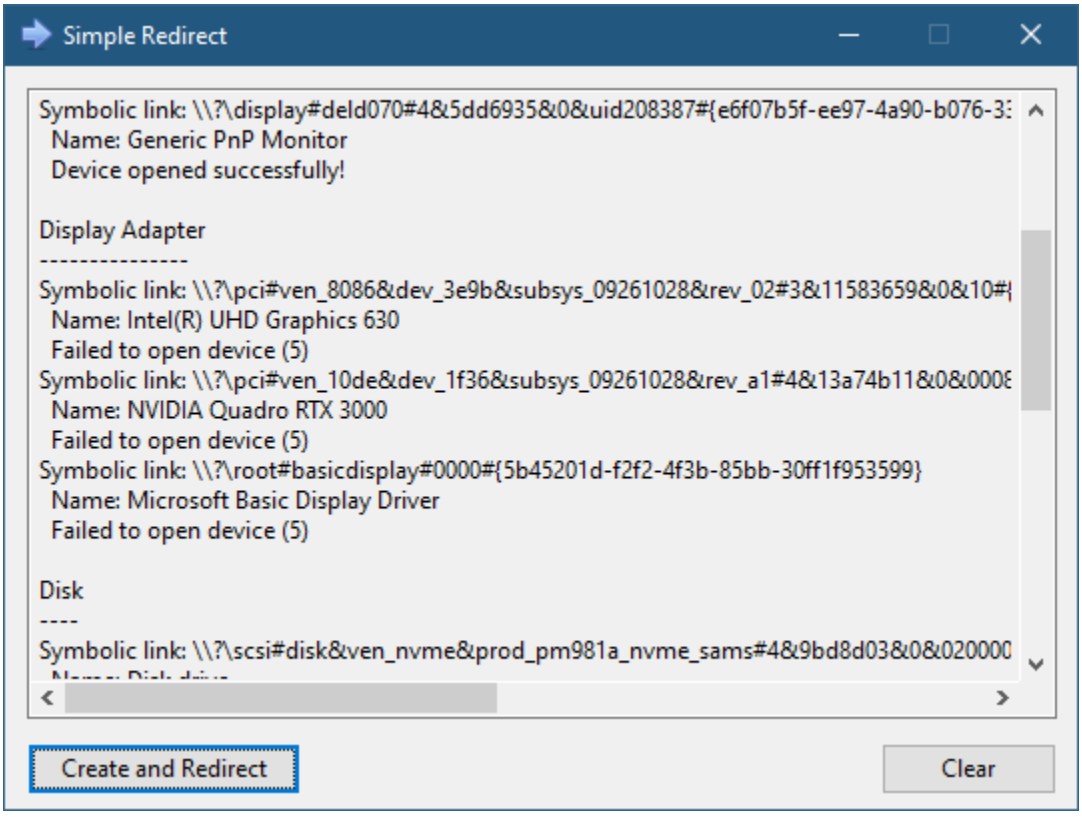


Figure 11-13: Redirected text in *Simple Redirect*

The basic idea is to create an anonymous pipe and share its write end with the *EnumDevices* process. This way, anything the *EnumDevices* process writes, the read end of the pipe can be used to read it. To make it work, the write and of the pipe must be attached to the standard output of the *EnumDevices* process, so any standard output calls (such as `printf`) will be available through the pipe. This arrangement is depicted in figure 11-14.

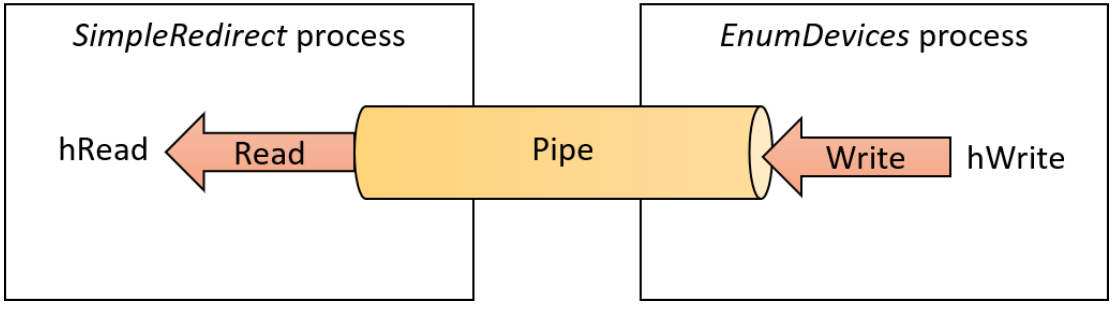


Figure 11-14: Anonymous pipe with redirection

The trick is to pass the write handle to the new process, which is done with process handle

inheritance, as described in chapter 3.

The `CMainDlg::OnRedirect` function does the work of creating the pipe and using it. First it creates the pipe:

```
LRESULT CMainDlg::OnRedirect(WORD, WORD wID, HWND, BOOL&) {
    wil::unique_handle hRead, hWrite;
    if (!::CreatePipe(hRead.addressof(), hWrite.addressof(), nullptr, 0))
        return Error(L"Failed to create pipe");
```

Next, the write handle needs to be shared with the new process (yet to be created), so it must be made inheritable:

```
::SetHandleInformation(hWrite.get(), HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
```

Now the *EnumDevices* process can be created, by calling the `CreateOtherProcess` helper function (discussed shortly). Then the local write handle is not needed anymore, so can be closed:

```
if (!CreateOtherProcess(hWrite.get()))
    return Error(L"Failed to create process");

// local write handle not needed anymore
hWrite.reset();
```

All that's left to do is read data from the read end of the pipe and use the data:

```
char buffer[1 << 12] = { 0 };
DWORD bytes;
CEdit edit(GetDlgItem(IDC_TEXT));
ATLASSERT(edit);

while (::ReadFile(hRead.get(), buffer, sizeof(buffer), &bytes, nullptr) && byte\
s > 0) {
    CString text;
    edit.GetWindowText(text);
    text += CString(buffer);
    edit.SetWindowText(text);
    ::memset(buffer, 0, sizeof(buffer));
}
```

It's a normal `ReadFile` call that is repeated as long as there is data in written to the pipe from the other end.

To make this work properly, the new process needs to be created with handle inheritance and proper flags to make the inherited handle be used as standard output:

```
bool CMainDlg::CreateOtherProcess(HANDLE hOutput) {
    PROCESS_INFORMATION pi;
    STARTUPINFO si = { sizeof(si) };
    si.hStdOutput = hOutput;
    si.dwFlags = STARTF_USESTDHANDLES;

    WCHAR path[MAX_PATH];
    ::GetModuleFileName(nullptr, path, _countof(path));
    *::wcsrchr(path, L'\\') = L'\0';
    ::wcscat_s(path, L"\\EnumDevices.exe");

    BOOL created = ::CreateProcess(nullptr, path, nullptr, nullptr, TRUE,
        CREATE_NO_WINDOW, nullptr, nullptr, &si, &pi);
    if (created) {
        ::CloseHandle(pi.hProcess);
        ::CloseHandle(pi.hThread);
    }

    return created;
}
```

The `STARTUPINFO` structure is initialized by setting the `hStdOutput` member to the value of the write handle. This works because inherited handles have the same value in the new process. The flag `STARTF_USESTDHANDLES` ensures the standard handles are picked up automatically by the new process.

To locate the *EnumDevices* executable, the code assumes it's in the same directory as the current executable. `GetModuleFileName` with a `NULL` first argument returns the full executable path of the current process. Then, the filename part is replaced with "EnumDevices".

Finally, `CreateProcess` is called with the handle inheritance flag set to `TRUE` (fifth argument). The returned handles are closed properly as they are not really needed. Adding the `CREATE_NO_WINDOW` flag is a nice touch that prevents a console window popping up for the new process.

Named pipes and mailslots are discussed in their own chapter (in Part 2).

Transactional NTFS

The Windows executive has a component called *Kernel Transaction Manager* (KTM), that provides support for using transactions for file (and registry) operations. For files, it's sometimes referred to as *Transactional NTFS* (TxF). A transaction is a set of operations that adhere to the so-called *ACID* properties:

- Atomicity - either all operations in a transaction succeed, or all operations fail.
- Consistency - the file system will always be in a consistent state.
- Isolation - multiple transaction in progress do not affect each other.
- Durability - system failure should not cause the transaction to violate the earlier properties.

Microsoft's documentation has been warning developers for some years now not to rely on the KTM, and look for other mechanisms to get similar results. The warning indicates that transaction support for file and registry operations may be removed in a future Windows version. Why? My guess is not many developers use this powerful facility. In any case, it hasn't happened yet, and in my opinion the alternatives listed in the documentation are not real replacements for the KTM.

This section provides a quick introduction to TxF. To get started with transactional operations, call `CreateTransaction` to create a new transaction:

```
HANDLE CreateTransaction (
    _In_opt_ LPSECURITY_ATTRIBUTES lpTransactionAttributes,
    _In_opt_ LPGUID UOW,            // must be NULL
    _In_opt_ DWORD CreateOptions,
    _In_opt_ DWORD IsolationLevel,  // must be 0
    _In_opt_ DWORD IsolationFlags,  // must be 0
    _In_opt_ DWORD Timeout,
    _In_opt_ LPTSTR Description);
```

The transaction APIs have their own `#include` (<*ktmw32.h*>) and import library (*ktmw32.lib*).

The function has some unused parameters. `lpTransactionAttributes` is the standard `SECURITY_ATTRIBUTES` structure. `CreateOptions` can be zero or `TRANSACTION_DO_NOT_PROMOTE` to

prevent promoting the transaction to a distributed one. If `Timeout` is provided which is not zero or `INFINITE`, the transaction will abort after the specified time in milliseconds elapses. Otherwise, there is no timeout for the transaction. The last parameter is an optional human-readable string describing the transaction.

The function returns a handle to the new transaction, or `INVALID_HANDLE_VALUE` if it fails.

With a transaction handle in hand, several file related functions accept a transaction handle, such as `CreateFileTransacted`:

```
HANDLE CreateFileTransacted(
    _In_       LPCTSTR lpFileName,
    _In_       DWORD dwDesiredAccess,
    _In_       DWORD dwShareMode,
    _In_opt_   LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _In_       DWORD dwCreationDisposition,
    _In_       DWORD dwFlagsAndAttributes,
    _In_opt_   HANDLE hTemplateFile,
    _In_       HANDLE hTransaction,
    _In_opt_   PUSHORT pusMiniVersion,
    _Reserved_ PVOID  lpExtendedParameter); // NULL
```

The function is an extended version of `CreateFile`. The file name must reference a local file, otherwise the function fails and `GetLastError` returns `ERROR_TRANSACTIONS_UNSUPPORTED_REMOTE`. `hTransaction` is the transaction handle obtained from `CreateTransaction`.

The `pusMiniVersion` parameter should be `NULL` if the file is opened for read access only. If opened for write access, it indicates what kind of view the file should present to clients during the transaction (defined in *txfw32.h*):

- `TXFS_MINIVERSION_COMMITTED_VIEW` - view based on the last commit.
- `TXFS_MINIVERSION_DIRTY_VIEW` - dirty view as its being modified by the transaction.
- `TXFS_MINIVERSION_DEFAULT_VIEW` - committed for a transaction that does not modify the file, dirty otherwise.

> A custom miniversion view can also be created by using the `FSCTL_TXFS_CREATE_MINIVERSION` I/O control code (check the documentation).

The handle returned by `CreateFileTransacted` can be passed to normal I/O access functions, such as `ReadFile` and `WriteFile`. This means that once a file object is created transacted, all the other operations on the file remain exactly the same.

Similarly to `CreateFileTransacted`, there are other functions that can work as part of a transaction: `CopyFileTransacted`, `CreateHardLinkTransacted`, `DeleteFileTransacted`, `CreateDirectoryTransacted` and more.

Once all operations complete successfully, the transaction can be committed with `CommitTransaction`:

```
BOOL CommitTransaction(_In_ HANDLE TransactionHandle);
```

If something went wrong with the various operations in the transaction, you can request all operations to roll back:

```
BOOL RollbackTransaction(_In_ HANDLE TransactionHandle);
```

Transaction handles are closed normally with `CloseHandle`.

> The kernel object type for a transaction is *TmTx*.

Every transaction has an ID that can be retrieved with `GetTransactionId`:

```
BOOL GetTransactionId (
    _In_  HANDLE TransactionHandle,
    _Out_ LPGUID TransactionId);
```

The returned GUID can be used to open a handle to an existing transaction with `OpenTransaction`:

```
HANDLE OpenTransaction (
    _In_ DWORD dwDesiredAccess,
    _In_ LPGUID TransactionId);
```

> Transactions are implemented behind the covers with the *Common Log File System* (CLFS) logging facility.

File Search and Enumeration

Sometimes there is a need to search or enumerate for files and directories. Fortunately, the file management APIs provide several functions to accomplish such a fit. To start enumeration/search, call either `FindFirstFile` or `FindFirstFileEx`:

```
HANDLE FindFirstFileW(
    _In_ LPCTSTR lpFileName,
    _Out_ LPWIN32_FIND_DATA lpFindFileData);

HANDLE FindFirstFileEx(
    _In_  LPCTSTR lpFileName,
    _In_  FINDEX_INFO_LEVELS fInfoLevelId,
    _Out_ LPVOID lpFindFileData,
    _In_  FINDEX_SEARCH_OPS fSearchOp,
    _Reserved_ LPVOID lpSearchFilter,
    _In_  DWORD dwAdditionalFlags);
```

Both functions accepts a file name to start the search with. This can be any path specification and include wildcards. Examples include *c:\temp*.png* and *c:\mydir\file??.txt*.

Each result is returned with the `WIN32_FIND_DATA` structure defined like so:

```
typedef struct _WIN32_FIND_DATA {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
    DWORD nFileSizeHigh;
    DWORD nFileSizeLow;
    DWORD dwReserved0;
    DWORD dwReserved1;
    _Field_z_ TCHAR  cFileName[ MAX_PATH ];
    _Field_z_ TCHAR  cAlternateFileName[ 14 ];
} WIN32_FIND_DATA, *PWIN32_FIND_DATA, *LPWIN32_FIND_DATA;
```

The structure gives the basic properties of a file.

The extended function has an `fInfoLevelId` parameter to indicate what information to return. Using `FindExInfoStandard` is equivalent to the non-extended function. The alternative value, `FindExInfoBasic`, does not return the short file name (in `cAlternateFileName`), which speeds up the search operation, and is recommended, since short filenames are rarely (if at all) needed.

The `fSearchOp` parameter's only useful value is `FindExSearchLimitToDirectories`, that hints to locate directories only. Not all file systems support this hint, so don't rely on it.

The last parameter to the extended function, `dwAdditionalFlags` provides some more customization for the search:

- `FIND_FIRST_EX_CASE_SENSITIVE` - searches are case sensitive.
- `FIND_FIRST_EX_LARGE_FETCH` - uses larger buffer for searches, which can increase performance at the expense of more memory usage.
- `FIND_FIRST_EX_ON_DISK_ENTRIES_ONLY` - skips files that are not resident (such as virtualized files common in services such as *OneDrive*).

The functions return a search handle, which is `INVALID_HANDLE_VALUE` if an error occurs.

With a valid handle, the first search match is available. To progress to the next match, call `FindNextFile`:

```
BOOL FindNextFile(
    _In_ HANDLE hFindFile,
    _Out_ LPWIN32_FIND_DATA lpFindFileData);
```

The function returns the next match, or `FALSE` if there are no matches.

When you're done with the search, call `FindClose` to close the search handle:

```
BOOL FindClose(_Inout_ HANDLE hFindFile);
```

NTFS Streams

The NTFS file system supports *file streams*, which are essentially files within a file. Normally, we use the default data stream, but others can be created and used. These are essentially hidden from normal view and don't show up in standard tools such as *Windows Explorer*.

A familiar example of such a case is when downloading some types of files from the web, and when selecting *Properties* in *Explorer*, shows something like figure 11-15.

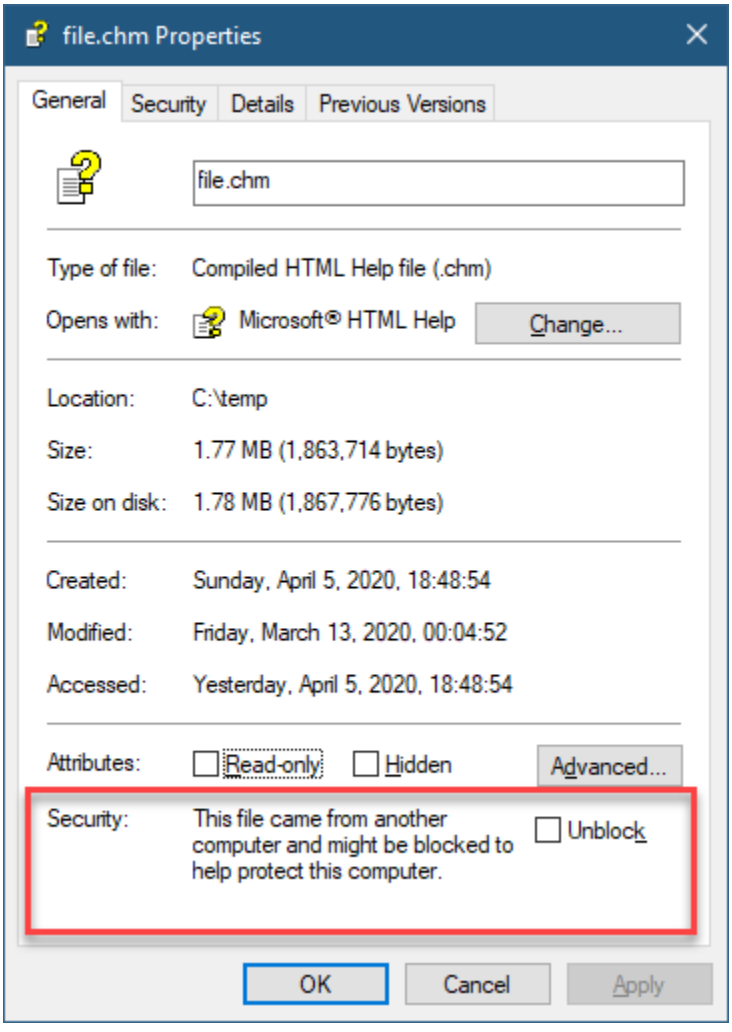


Figure 11-15: Properties of a "blocked" file

How does *Explorer* "know" this file came from a different machine? The secret is in an NTFS stream that is within the file. The *streams* command-line tool from *Sysinternals* can identify such streams. Here is the output for the file in figure 11-15:

```
C:\>streams -nobanner file.chm
C:\file.chm:
   :Zone.Identifier:$DATA       26
```

There is a hidden NTFS stream named "Zone.Identifier" within the file and it's 26 bytes long. *Streams* does not show the contents of NTFS streams, but my tool *NTFS Streams* does (figure 11-16).

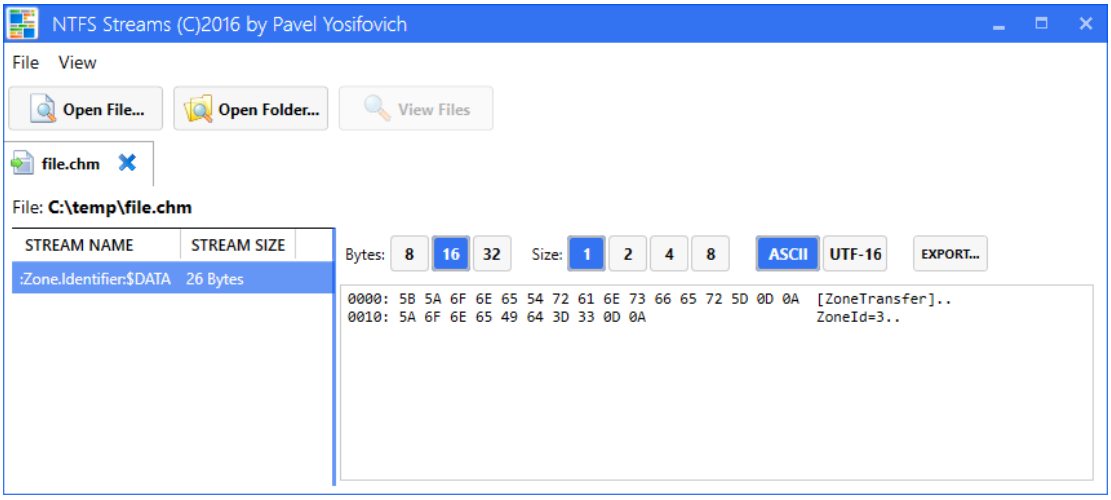


Figure 11-16: Stream contents in *NTFS Streams*

You can find *NTFS Streams* in my Gitub repository https://github.com/zodiacon/AllTools or https://github.com/zodiacon/NtfsStreams.

The *HTML Help* (*hh.exe*) Windows application looks for this stream and if found does not parse the HTML.

The *Unblock* check box in figure 11-15 deletes the stream, allowing *HTML Help* to work normally.

How can we create such hidden streams? The normal `CreateFile` function can be used, where the filename is appended with a colon and the stream's name. Here is an example:

```
HANDLE hFile = ::CreateFile(L"c:\\temp\\myfile.txt:mystream", GENERIC_WRITE, 0,
    nullptr, CREATE_NEW, 0, nullptr);
char text[] = "Hello from a hidden stream!";
DWORD bytes;
::WriteFile(hFile, text, ::strlen(text), &bytes, nullptr);
::CloseHandle(hFile);
```

Here is some interaction with the new file:

```
C:\temp>dir myfile.txt
 Volume in drive C is OS
 Volume Serial Number is 9010-6C18

 Directory of C:\temp

06-Apr-20  12:11                 0 myfile.txt
               1 File(s)              0 bytes
               0 Dir(s)  904,581,414,912 bytes free

C:\temp>streams -nobanner myfile.txt
C:\temp\myfile.txt:
        :mystream:$DATA 27
```

The file is shown as having zero size! Clearly, this is not the case, as there is a hidden stream inside that can be arbitrarily long. *NTFS Streams* will show the contents of the stream.

> The "$DATA" suffix is the default stream reparse point. Custom reparse points can be created that are handled in special ways by file system filter drivers. This is beyond the scope of this book.

You may be wondering how *Streams* and *NTFS Streams* work. Two functions exist for enumerating streams within a file, very similar to the search functions from the previous section:

```
HANDLE WINAPI FindFirstStream(
  _In_       LPCTSTR            lpFileName,
  _In_       STREAM_INFO_LEVELS InfoLevel,
  _Out_      LPVOID             lpFindStreamData,
  _Reserved_ DWORD              dwFlags);

BOOL FindNextStreamW(
  _In_ HANDLE hFindStream,
  _Out_ LPVOID lpFindStreamData);
```

The functions allow enumerating all streams in a file. Each returned value in the `lpFindStreamData` parameter is actually the following structure:

```
typedef struct _WIN32_FIND_STREAM_DATA {
  LARGE_INTEGER StreamSize;
  WCHAR         cStreamName[MAX_PATH + 36];
} WIN32_FIND_STREAM_DATA, *PWIN32_FIND_STREAM_DATA;
```

It provides the stream size and its name.

Write an equivalent tool to *Streams*.

Summary

This chapter was all about I/O operations. We saw how to work with files and devices, both synchronously and asynchronously. There are more file-related APIs that we didn't cover in this chapter, for which you can find more information in the documentation. Examples of functionality not discussed in this chapter include file operations (copy, move, etc.), file links (soft and hard), file locking, and file encryption and decryption.

In the next chapter, we'll venture into a new area that no application or operating system can do without - memory management.

Chapter 12: Memory Management Fundamentals

Memory is a fundamental building block of any computer system. In the old days, using memory was relatively simple, as an application just allocated physical memory directly, used it, freed it, and that was it. Modern operating systems manage *virtual memory*, a term that has some unfortunate connotations. In this chapter we'll introduce all major concepts related to memory - both virtual and physical.

In this chapter:

- **Basic Concepts**
- **Process Address Space**
- **Memory Counters**
- **Process Memory Map**
- **Page Protection**
- **Enumerating Address Space Regions**
- **Sharing Memory**
- **Page Files**
- **WOW64**
- **Virtual Address Translation**

Basic Concepts

Today's modern Intel/AMD processors started as very modest in terms of memory. The original 8086/8088 processors supported just 1 MB of memory (physical, as there was nothing else at the time). Each access to memory was a combination of a segment address and an offset, which was needed because these processors worked with 16-bit values internally, but the memory access requires 20 bits (1 MB). A segment register's value (16-bit) was multiplied by 16 (0x10) and then

an offset was added to reach an address in the 1 MB range. This mode of working is now called *Real Mode*, and is still the mode in which today's Intel/AMD processors wake up in.

With the introduction of the 80386 processor, virtual memory was born as it's essentially used today, including the ability to access memory linearly (with the segment registers just set to zero) by using offsets only. This makes memory access much more convenient. Virtual memory means every memory access needs to be translated to where the physical address is. This mode is referred to as *Protected Mode*. In protected mode, there is no way to access physical memory directly - only via a mapping from a virtual address to a physical address. This mapping must be prepared upfront by the operating system's Memory Manager, since the CPU expects this mapping to be present.

> On 64-bit systems, *protected mode* is called *Long Mode*, but it's essentially the same mechanism, extended to 64-bit addresses.

The mapping between virtual and physical addresses, as well as the management of memory blocks on the OS level are performed in chunks called *pages*. This is necessary, since it's not possible to manage every single byte - the management structure will be much larger than that byte. There are two supported page sizes, with a third size supported on Windows 10 and Server 2016 on x64 systems. Table 12-1 lists the page sizes for all architectures Windows supports.

Table 12-1: Page sizes

Architecture	Small (normal) page	Large page	Huge page
x86	4 KB	2 MB	N/A
x64	4 KB	2 MB	1 GB
ARM	4 KB	4 MB	N/A
ARM64	4 KB	2 MB	N/A

Small (normal) pages are the default, and the term "page" used throughout this chapter (and the next ones) means a small or normal page, which is 4 KB on all architectures. If a different page size is mentioned, it will be accompanied by an explicit prefix of "large" or "huge".

Process Address Space

Each process has its own linear, virtual, private address space. The address space starts at address zero and ends at some maximum value, based on the OS bitness (32 or 64) and the process bitness,

as we shall soon see. The important part here is "private". For example, stating that there is some data at address `0x100000` requires another question answered: in which process? Every process has an address `0x100000`, but that address may be mapped to a different physical address, to a file, or to nothing at all. This conceptual mapping is illustrated in figure 12-1, where two processes are mapping some of their pages to physical memory (RAM), some of their pages to disk, and still others are unmapped.

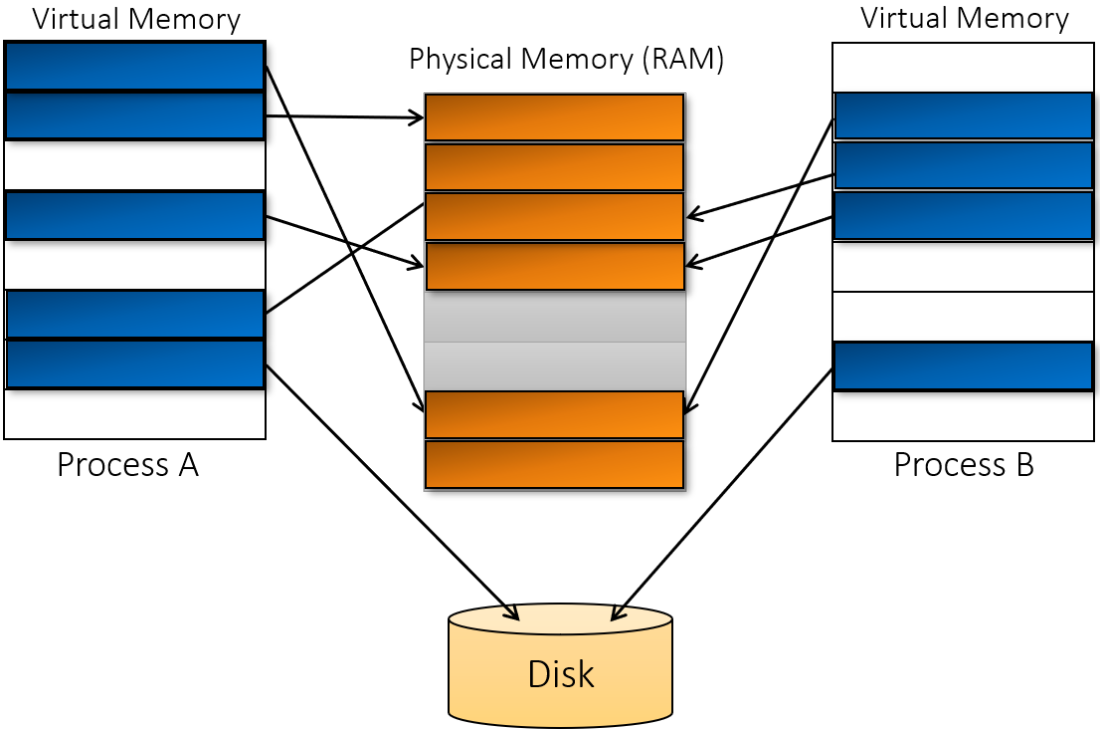


Figure 12-1: Virtual address mapping

A process can directly access memory in its own address space. This means a process cannot accidentally or maliciously read or write to another process' address space simply by manipulating a pointer. It is possible to access memory of another process, but that requires calling a function (`ReadProcessMemory` or `WriteProcessMemory`, discussed later in this chapter) with a strong-enough handle to the target process.

The address space of a process is referred to as *virtual*. This refers to the fact that the address space is just that: a space for potential memory mappings. Every process starts with very modest usage of its virtual address space - the executable is mapped as well as *NtDll.Dll*. Then the loader (part of *NtDll*) allocates some basic structures within the process address space, such as the default process heap (discussed in the next chapter), the *Process Environment Block* (PEB), the *Thread Environment Block* (TEB) for the first thread in the process. Most of the address space is empty.

Page States

Each page in virtual memory can be in one of three states: free, committed and reserved. A free page is unmapped, and so trying to access that page causes an access violation exception. Most of a process' address space starts as free.

A *committed* page is the opposite of free - this is a mapped page, to RAM or to a file, and accessing that page should succeed (barring any conflicting page protection, discussed later in this chapter). If the page is in RAM, the CPU accesses the data directly and moves on. If the page is not in RAM (at least as far as the tables the CPU consults tell it), the CPU raises an exception called *page fault*, which is captured by the memory manager. If the page does indeed reside on disk, the memory manager brings it back to RAM, fixes the translation table to point to the new address in RAM, and instructs the CPU to try again. The net result is that the access succeeds from the point of view of the calling thread. if indeed I/O was involved, the access is slower, but the calling thread does not need to know about this or do anything special about it - it works transparently.

Technically, accessing a free page causes a page fault as well. In this case, the memory manager concludes that there is nothing behind the given address and raises an access violation exception.

Committed memory is what normally is called "allocated" memory. Calling C/C++ memory allocation functions, such `malloc`, `calloc`, `operator new`, etc. always commit memory (if they succeed, of course).

> We'll discuss memory allocation APIs in depth in the next chapter.

The last page state is somewhere between free and committed called *reserved*. A reserved page is similar to free, in the sense that accessing that page causes an access violation - there is nothing there. The reserved page may be committed later on. A reserved page range ensures that normal memory allocations do not use that range, because it's reserved for another purpose. We've seen this idea in the way a thread's stack is managed. Since a thread's stack can grow, and must be contiguous in virtual memory, a range of pages is reserved so that other allocations happening in the process don't use the reserved address range.

Table 12-2 summarizes the pages states.

Table 12-2: Page states

Page state	Meaning	If accessed
Free	Unallocated page	Access violation exception
Committed	Allocated page	Success (assuming no page protection restriction)
Reserved	Unallocated page, reserved for future use	Access violation exception

Address Space Layout

In this section, we'll examine the address layout of processes on 32-bit and 64-bit systems. Table 12-3 summarizes the address space sizes.

Table 12-3: Process virtual address size

OS Type	Process type	`LARGEADDRESSAWARE` clear	`LARGEADDRESSAWARE` set
32-bit booted w/o increase UVA	32-bit	2 GB	2 GB
32-bit booted w/ increase UVA	32-bit	2 GB	2 GB to 3 GB
64-bit (Windows 8.1+)	32-bit	2 GB	4 GB
64-bit (Windows 8.1+)	64-bit	2 GB	128 TB
64-bit (up to Windows 8)	32-bit	2 GB	4 GB
64-bit (up to Windows 8)	64-bit	2 GB	8 TB

The `LARGEADDRESSAWARE` is a linker flag that can be specified when building an executable, and is stored as part of the PE header. It can also be set later, without access to the source code, using a PE-editing tool, such as the *editbin.exe* command-line tool, available in the Windows SDK. What is the purpose of this flag?

If the executable is signed, however, changing this flag (or any other flag for that matter) will invalidate the signature.

Originally (before Windows NT 4), 32-bit processes could get 2 GB of address space only. 2 GB requires 31 bits to represent an address, so the *most significant bit* (MSB) is always zero. Starting with NT 4, 32-bit Windows could boot with a 3 GB address space per process. However, some

developers could take advantage of the fact that any address they use has its MSB set to zero, and use the free bit for some applicative purpose. Then, if such a process is given more than 2 GB, where the MSB might be 1, the process will fail in some way, as it will mask off the MSB before accessing memory. Setting the `LARGEADDRESSAWARE` bit states that the developers for that executable did not mess around with the MSB of addresses, so the process can accepts addresses larger than 2 GB (`0x80000000`) without any issue.

> This bit only affects executables, not DLLs. DLLs must always work correctly and never assume anything about the address values they are given.

You typically set this bit in Visual Studio, in the project properties / *Linker* / *System* (figure 12-2). The default is "No" for 32-bit configurations and "Yes" for 64-bit configurations. For 32-bit executables, assuming you don't assume anything special about addresses, there is almost no downside to setting the flag to "Yes".

One downside does exist: if your process leaks memory, it will have more address space to leak it, which means the system will consume more memory because of your process.

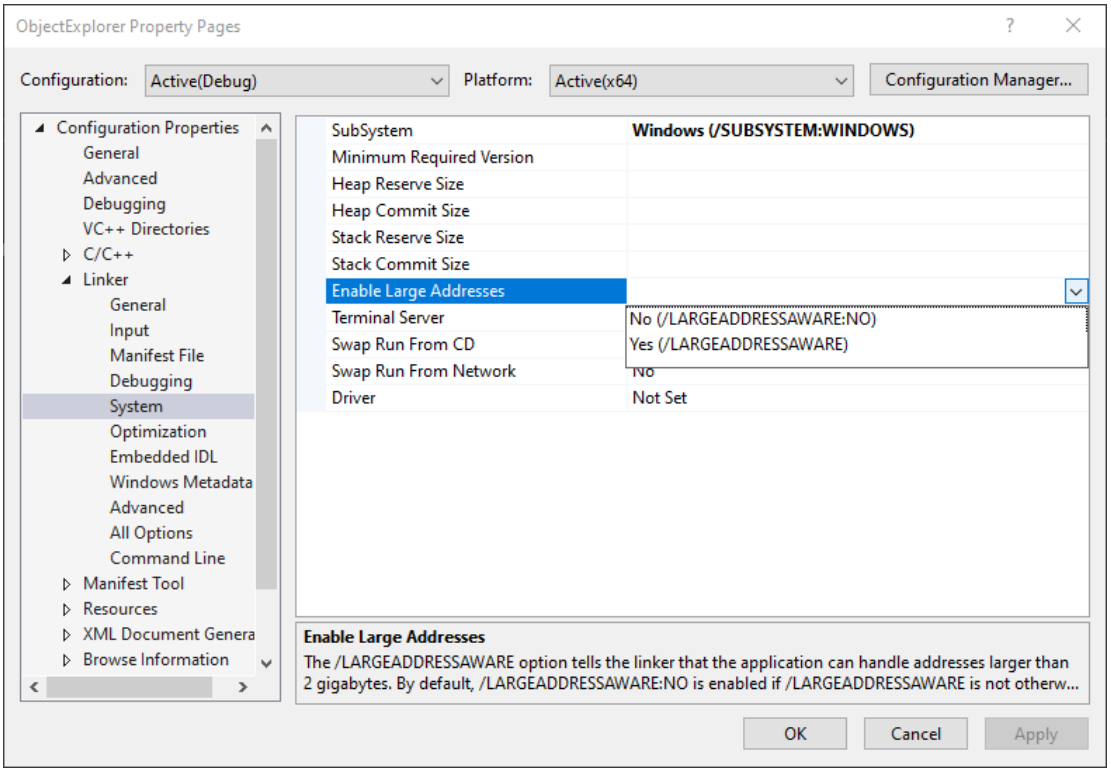


Figure 12-2: `LARGEADDRESSAWARE` flag in *Visual Studio*

You can use the *Dumpbin.exe* command-line tool to view information about a PE, including the state of the `LARGEADDRESSAWARE` bit. Here is an example with *Explorer.exe*:

```
C:\>dumpbin /headers c:\windows\explorer.exe
Microsoft (R) COFF/PE Dumper Version 14.26.28720.3
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file c:\windows\explorer.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
            8664 machine (x64)
               8 number of sections
        4D818882 time date stamp
```

```
          0 file pointer to symbol table
          0 number of symbols
         F0 size of optional header
         22 characteristics
              Executable
              Application can handle large (>2GB) addresses
...
```

As mentioned in chapter 5, there are several graphic tools that can show this information as well, such as my own *PE Explorer V2* (figure 12-3).

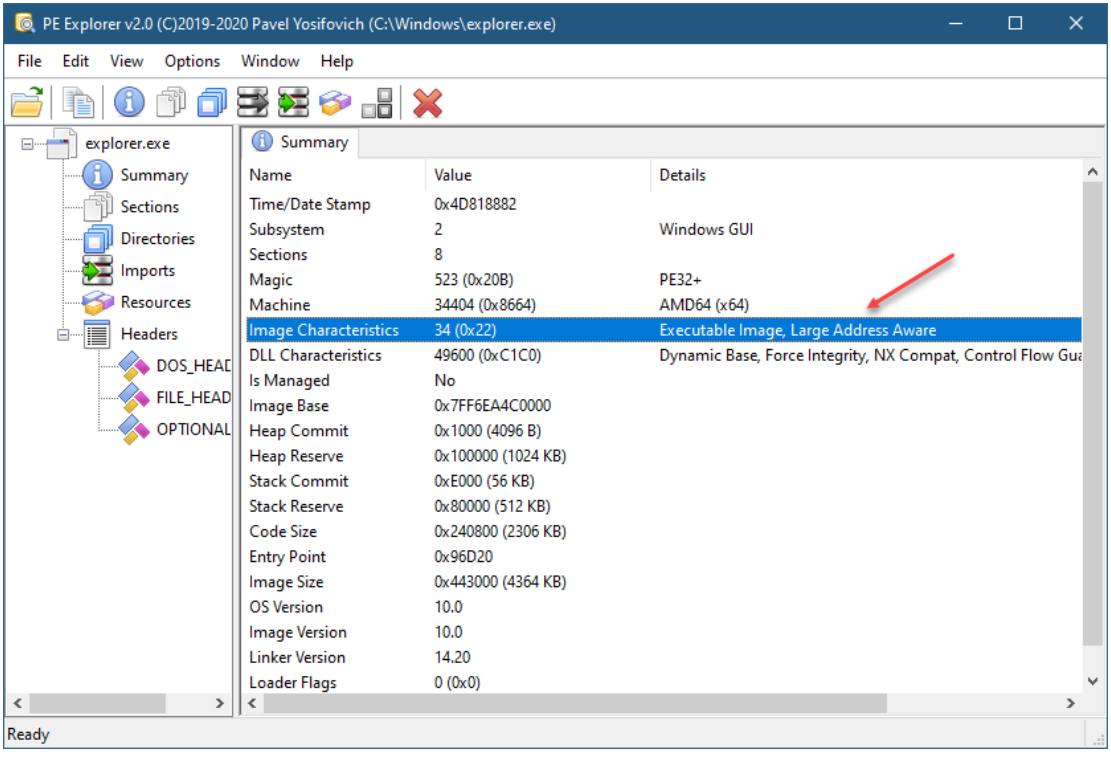


Figure 12-3: *PE Explorer V2* showing PE properties

32-bit Systems

On 32-bit systems, two variants exits, listed in table 12-3 and shown graphically in figure 12-4.

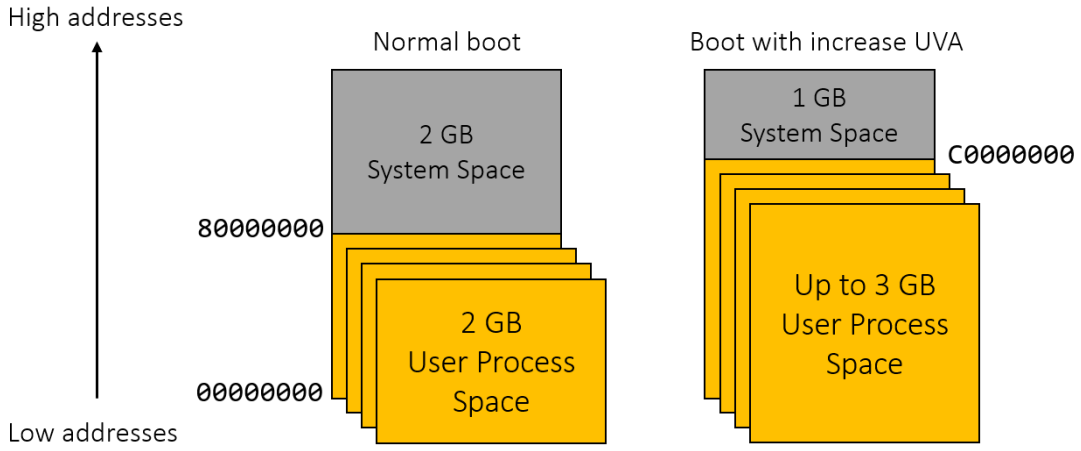


Figure 12-4: 32-bit address space layout

32-bit means 4 GB, which may have left you wondering why processes get only 2 GB. Figure 12-2 solves the mystery: the upper 2 GB are *system space* (also called *kernel space*). This is where the operating system kernel itself resides, with all kernel device drivers, and the memory they're consuming in terms of code and data.

> Notice that system space is a singleton - there us just one system, just one kernel. This means addresses in system space are absolute, rather than relative; they mean the same thing from every process context.

If the system boots with the "increase user virtual address" option, the system has to make do with an address range of 1 GB only, and user processes get 2 GB to 3 GB (anything about 2 GB requires having the `LARGEADDRESSAWARE` flag in their PE header).

Booting a 32-bit system with the "increase UVA" option can be done by running the following in an elevated command window, then rebooting the system:

```
c:\>bcdedit /set increaseuserva 3072
```

The number is the amount of user address space in MB, which can be from 2048 (2 GB default) up to 3072 (3 GB).

To remove this option, use `bcdedit /deletevalue increaseuserva`.

64-bit Systems

The 32-bit option to increase user address up to 3 GB is nice, but not nearly enough. This was a step gap until true 64-bit operating systems are available. Today, most (if not all) desktop Windows versions are 64-bit, not just servers. 64-bit systems offer several advantages, the first of which is a radically increased address space.

The theoretical limit for 64 bits is 2 to the 64th power, or 16 EB (Giga, Tera, Peta, Exa). This is a literally astronomical address range, that seems unreachable in today's systems. To make use of such an address space you would have to have RAM plus paging files close to this number, which is still far away from today's systems. In fact, most modern processors support only 48 bits of virtual and physical addresses. This means that the maximum address range that is possible to get is 2 to the 48th power, or 256 TB. This is why on a 64-bit system each process can have 128 TB of address space range, where the other 128 TB are for system space.

The *Sunny Cove* Intel microarchitecture supports 57 bits of virtual address space and 52 bits of physical address space. This means address space with such processors will be 64 PB per process and 64 PB for system space!

The transition to 64-bit systems was mostly painless because of the ability to run 32-bit x86 processes on a 64-bit x64 system without any change to the original binary. This will be discussed further in the section "WOW64", later in this chapter. 32-bit executables that have the `LARGEADDRESSAWARE` bit get 4 GB of address space on a 64-bit system. This makes sense, as the transition from 3 GB to 4 GB does require extra bits, so a process that can handle 3 GB can certainly handle 4 GB.

A canonical example of an executable that leverages this capability is Visual Studio (*devenv.exe*). Visual Studio is a 32-bit process, and since developers use 64-bit systems, Visual Studio gets 4 GB of address space. Some people claim it allows Visual Studio to leak more memory :)

There was some pain in the transition from 32-bit to 64-bit. The pain was in converting device drivers from 32-bit to 64-bit. There is no "WOW64" layer in the kernel. Complex kernel drivers, such as display drivers had stability issues in the early 64-bit days. Fortunately, these are all behind us now.

Figure 12-5 shows the address space layouts on 64-bit systems, for 32-bit and 64-bit processes.

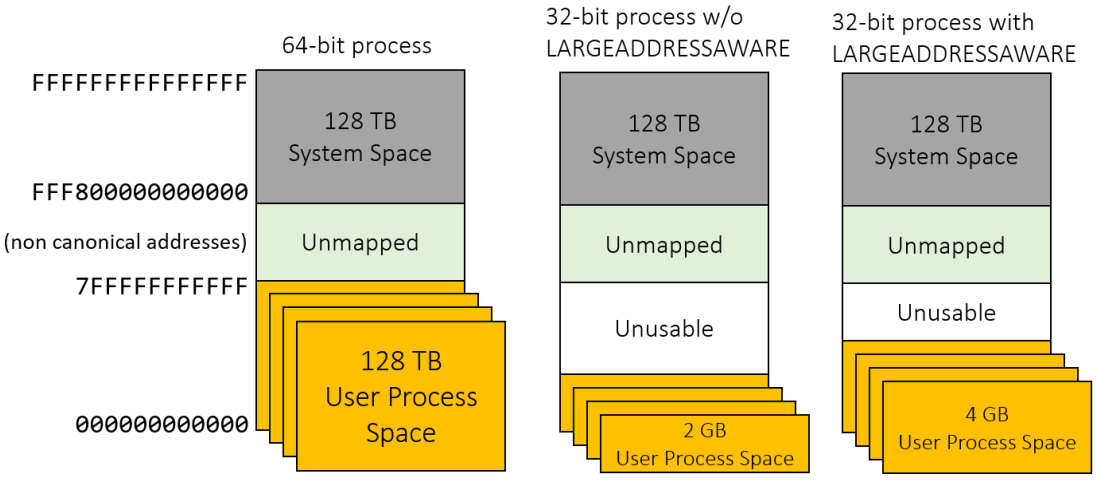


Figure 12-5: Address layouts on 64-bit systems

64-bit versions of Windows 8 and earlier only support 8 TB of user address space and 8 TB for system space. This was due to an implementation detail in the kernel that was fixed in Windows 8.1.

64-bit systems are not all rose gardens, however. 64-bit processes can leak memory to the point of bringing the system to its knees, as the address space is practically unlimited - RAM plus page files will be full before a 64-bit process' address space runs out. Also, address translation requires an extra level compared to 32-bit systems, which could be slower if the *Translation Lookaside Buffer* (TLB) cache is not used effectively (see later in this chapter).

Address Space Usage

We've seen in broad strokes the amount of virtual memory space available to various types of processes. However, not the entire user address space range is usable. To get a sense of

what is and is not usable, we can call the `GetSystemInfo` function, and its sister function, `GetNativeSystemInfo`:

```
VOID GetSystemInfo(_Out_ LPSYSTEM_INFO lpSystemInfo);
VOID GetNativeSystemInfo(_Out_ LPSYSTEM_INFO lpSystemInfo);
```

Both functions return a `SYSTEM_INFO` structure defined like so:

```
typedef struct _SYSTEM_INFO {
    union {
        DWORD dwOemId;          // Obsolete, do not use
        struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    };
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD_PTR dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;      // obsolete
    DWORD dwAllocationGranularity;
    WORD wProcessorLevel;
    WORD wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

The structure contains some system-level information, including the minimum and maximum usable address for user-mode processes. The `GetSystemInfo` function takes into account the calling process bitness. A 32-bit process on a 64-bit system can only "see" 32-bit values. `GetNativeSystemInfo` allows looking at the "real" values - up to a point. For a 32-bit process on a 32-bit system and for a 64-process on a 64-bit system, the functions are identical.

The *sysinfo* application displays some of the information available in a `SYSTEM_INFO` structure. It starts simply by calling `GetSystemInfo`:

```
SYSTEM_INFO si;
::GetSystemInfo(&si);
DisplaySystemInfo(&si, "System information");
```

Whatever is returned from `GetSystemInfo` is passed to the helper function, `DisplaySystemInfo` to display the results.

If the current process is 32-bit on a 64-bit system (WOW64 process), a second call is made to `GetNativeSystemInfo` to display more accurate information:

```
BOOL isWow = FALSE;
if (sizeof(void*) == 4 && ::IsWow64Process(::GetCurrentProcess(), &isWow) && is\
Wow) {
    ::GetNativeSystemInfo(&si);
    printf("\n");
    DisplaySystemInfo(&si, "Native System information");
}
```

The question is how to check if a process is a WOW64 process. The functions `IsWow64Process` and the newer `IsWow64Process2` can help:

```
BOOL IsWow64Process(
    _In_ HANDLE hProcess,
    _Out_ PBOOL Wow64Process);

// Windows 10+ only
BOOL IsWow64Process2(
    _In_ HANDLE hProcess,
    _Out_ USHORT* pProcessMachine,
    _Out_opt_ USHORT* pNativeMachine);
```

`IsWow64Process` returns `TRUE` in `Wow64Process` if it's a WOW64 process. It's important to note that the function sets `Wow64Process` to `FALSE` if running on a 32-bit system.

The newer `IsWow64Process2` function provides more information about the processor powering the process and the native processor on the machine. `pProcessMachine` returns one of the `IMAGE_FILE_MACHINE_*` constants defined in *<winnt.h>*. If the value is `IMAGE_FILE_MACHINE_UNKNOWN`, it means the process is not WOW. If `pNativeMachine` is not `NULL`, it returns the native machine identifier from the same list.

The code in *sysinfo* checks if the current process is 32-bit (`sizeof(void*) == 4`) and is a WOW64 process. If both are true, then the native system is different from the current process and so a call to `GetNativeSystemInfo` is merited.

Task Manager provides a column named *Platform* in the *Details* tab, that shows the "bitness" of each process.

`DisplaySystemInfo` is mostly straightforward, displaying most of the information from a `SYSTEM_INFO` instance:

```
const char* GetProcessorArchitecture(WORD arch) {
    switch (arch) {
        case PROCESSOR_ARCHITECTURE_AMD64: return "x64";
        case PROCESSOR_ARCHITECTURE_INTEL: return "x86";
        case PROCESSOR_ARCHITECTURE_ARM: return "ARM";
        case PROCESSOR_ARCHITECTURE_ARM64: return "ARM64";
    }
    return "Unknown";
}

void DisplaySystemInfo(const SYSTEM_INFO* si, const char* title) {
    printf("%s\n%s\n", title, std::string(::strlen(title), '-').c_str());
    printf("%-24s%s\n", "Processor Architecture:",
        GetProcessorArchitecture(si->wProcessorArchitecture));
    printf("%-24s%u\n", "Number of Processors:", si->dwNumberOfProcessors);
    printf("%-24s0x%llX\n", "Active Processor Mask:",
        (DWORD64)si->dwActiveProcessorMask);
    printf("%-24s%u KB\n", "Page Size:", si->dwPageSize >> 10);
    printf("%-24s0x%p\n", "Min User Space Address:",
        si->lpMinimumApplicationAddress);
    printf("%-24s0x%p\n", "Max User Space Address:",
        si->lpMaximumApplicationAddress);
    printf("%-24s%u KB\n", "Allocation Granularity:",
        si->dwAllocationGranularity >> 10);
}
```

Here is an example output on a 64-bit system when compiling the app to 64-bit:

```
System information
------------------
Processor Architecture: x64
Number of Processors:   16
Active Processor Mask:  0xFFFF
Page Size:              4 KB
Min User Space Address: 0x0000000000010000
Max User Space Address: 0x00007FFFFFFEFFFF
Allocation Granularity: 64 KB
```

Notice the lowest usable address is `0x10000`, meaning the first 64 KB of virtual address space are unusable. These are traditionally used to catch `NULL` pointers. Similarly, the upper 64 KB of address space, just before system space starts, are unusable as well. In short, it means the usable

address space range is 128 KB less than perhaps expected. For 64-bit processes, this is completely unnoticeable.

Running the same application as a 32-bit executable on a 64-bit system yields the following:

```
System information
------------------
Processor Architecture: x86
Number of Processors:   16
Active Processor Mask:  0xFFFF
Page Size:              4 KB
Min User Space Address: 0x00010000
Max User Space Address: 0x7FFEFFFF
Allocation Granularity: 64 KB

Native System information
-------------------------
Processor Architecture: x64
Number of Processors:   16
Active Processor Mask:  0xFFFF
Page Size:              4 KB
Min User Space Address: 0x00010000
Max User Space Address: 0xFFFEFFFF
Allocation Granularity: 64 KB
```

We can see that the same 64 KB from the bottom and top are unusable. The native system information is accurate in terms of processor architecture (x64), while the "local" version claims the processor is x86. In terms of addresses, the upper limit in the native output assumes the entire 4 GB of 32-bit address space is available, which it is if the executable has the `LARGEADDRESSAWARE` linker flag set.

In all these cases, the page size reported is 4 KB (the small page size is the one reported). Also of note is something called *allocation granularity*, which, as we'll see later, is the granularity of allocations made with the `VirtualAlloc` family of functions. It is currently 64 KB on all Windows architectures and versions.

Here is another output on 32-bit Windows 8.1 with 4 logical processors:

```
System information
------------------
Processor Architecture: x86
Number of Processors:   4
Active Processor Mask:  0xF
Page Size:              4 KB
Min User Space Address: 0x00010000
Max User Space Address: 0x7FFEFFFF
Allocation Granularity: 64 KB
```

The memory-related values are identical to the WOW64 version, since this system was not booted in "increase UVA" mode.

Memory Counters

Developers often want to understand how their processes are doing in terms of memory usage. Is the process consuming a lot of memory? Is there perhaps a memory leak? What about the system itself? Windows provides many counters related to memory, and making sense of them is very important, as some have somewhat cryptic names, and in some cases different tools call the same counters by different names.

The first tool developers usually use to get a sense of what's going on on a system is *Task Manager*. Its *Performance* tab with the *Memory* sub-tab selected, shows system memory-related information. Figure 12-6 shows a snapshot with added annotations.

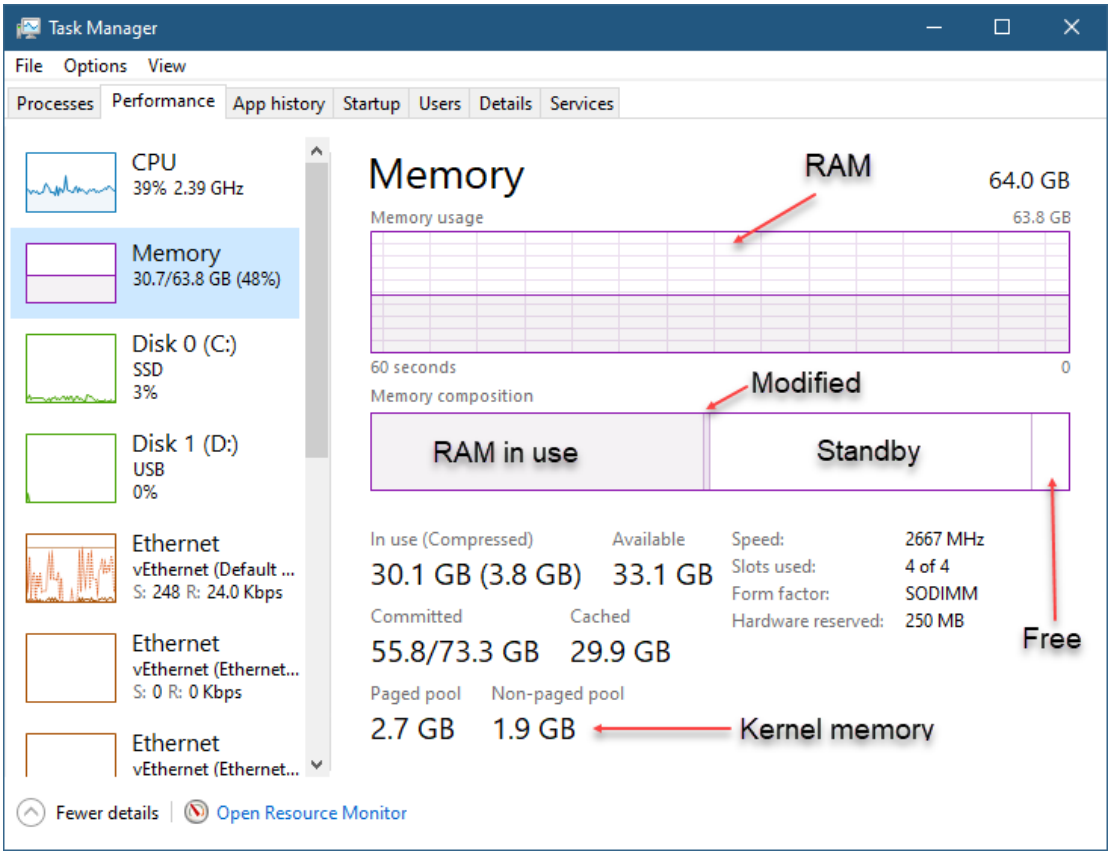


Figure 12-6: *Task Manager*'s *Performance*/*Memory* information

Table 12-4 summarizes the pieces of information shown in figure 12-6.

Table 12-4: Memory information in *Task Manager*

Name	Description
Memory usage graph	Shows the physical memory (RAM) consumption for the past 60 seconds
In Use	The current physical memory in Use
(Compressed)	The amount of compressed memory (see the sidebar "Memory Compression")
Committed / Commit Limit	Total committed memory / committed memory limit before page file expansion
Memory Composition - Modified	Memory that is not yet written to disk
Memory Composition - Free	Free pages (most of them are zero pages)

Table 12-4: Memory information in *Task Manager*

Name	Description
Cached	Memory that can be repurposed if required (Standby + Modified)
Available	Available physical memory (Standby + Free)
Paged pool / Non-paged pool	Kernel pools memory usage

Memory Compression

Memory compression was added in Windows 10, as a way to save memory by compressing not-currently needed memory, especially useful for UWP processes when they go to the background, as they don't consume CPU, so that any private physical memory used by these processes can be given away. Instead, the memory is compressed, still leaving free pages for other processes. When the process wakes up, the memory is quickly decompressed and ready to be used, avoiding I/O to a page file.

In the first two versions of Windows 10, the compressed memory was stored in the user-mode address space of the *system* process. This was too visible in tools, so starting with Windows 10 version 1607, a special process, *Memory Compression* (a minimal process), is the one holding on to the compressed memory. Furthermore, *Task Manager* explicitly does not show this process at all. Other tools, such as *Process Explorer*, show this process normally.

The *Memory Composition* bar in figure 12-6 indicates, in broad strokes, how physical pages are managed internally. The "In use" part are pages currently considered part of processes and the system's *working set*. The *Standby* pages is memory that has its backup stored on disk, but the relation to the owning process is still preserved. If the process now touches one of these pages, they immediately go back to its working set (become "in use"). If such pages were immediately thrown to the "free" pile of pages, an I/O would have been needed to get the page(s) back to RAM.

The *Modified* part represents pages whose contents has not yet been written to a backing store (typically a page file), so these cannot be discarded. If the number of modified pages becomes to large, or the standby and free page count becomes too small, modified pages will be written to their backing file, and they will move to the *standby* state.

All these transitions and management is geared towards reducing I/O. A more precise view of these physical page list management is available in the *System Information* view in *Process Explorer*, in the *Memory* tab, shown in figure 12-7. (Use *View* / *System Information...* menu to open it.)

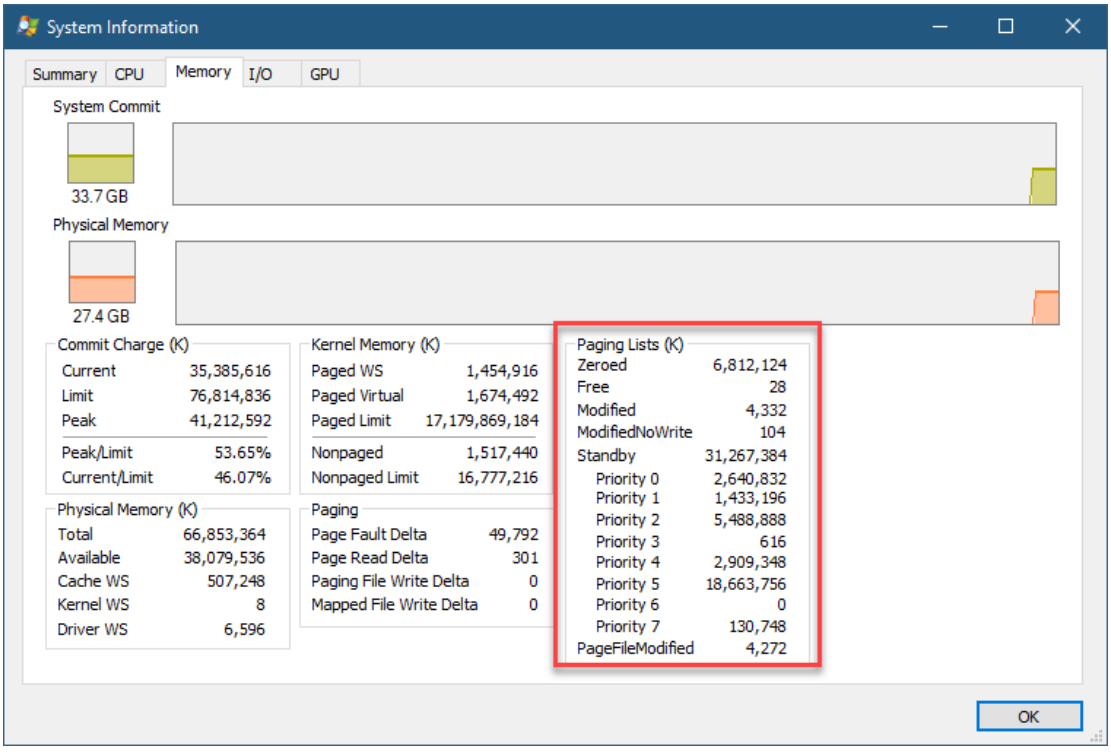


Figure 12-7: *System Information* for memory in *Process Explorer*

The *paging lists* part in figure 12-7 details the various lists used by the executive's Memory Management to manage physical pages. *Zeroed* pages are pages containing zeros only, and these are the majority compared to *Free* pages, which contain garbage. A special executive thread called the *Zero page thread*, that runs in priority 0 (the only thread that has this priority), is the one zeroing out free pages. The reason zero pages are important, is to satisfy a security requirement, where allocated memory cannot ever include data once belonging to another process, even if that process no longer exists. The *Free* part in the memory composition in figure 12-6 includes the free and zero pages combined.

Another interesting part of figure 12-7 is the fact that there is no single standby list of pages, but eight of them, based on priority. This is known as *Memory Priority*, and can be seen in *Process Explorer* on a thread by thread basis, although this is also a process property, inherited by default by each thread.

Memory priority is used when pages from the standby list need to be moved to become free pages, because processes or the system need physical memory. The question is, which pages should be "let go" first (and lose their connection to the original process)? A simple approach is to use a FIFO queue, where the first page to be removed form a process' working set is the first to become free. However, this is too simplistic. Suppose a process works a lot in the background, such as anti-malware or a backup application. These processes obviously use memory, but they

are not as important as the applications the user is working with directly. So if physical memory is needed, their standby pages should be the first to go, even if they were relatively recently used. This is where memory priority comes in.

The default memory priority is 5. In chapter 6, we looked at *background mode* for processes and threads, where the CPU priority is reduced to 4, and the memory priority is reduced to 1, making standby pages used by that process more likely to be reused before processes with a higher memory priority.

Sometimes, you may want to change the memory priority without entering *background mode*. Windows 8 and later provide this capability with the functions `SetProcessInformation` for a process-wide default, or `SetThreadInformation` on a thread by thread basis:

```
BOOL SetProcessInformation(
    _In_ HANDLE hProcess,
    _In_ PROCESS_INFORMATION_CLASS ProcessInformationClass,
    _In_ LPVOID ProcessInformation,
    _In_ DWORD ProcessInformationSize);

BOOL SetThreadInformation(
    _In_ HANDLE hThread,
    _In_ THREAD_INFORMATION_CLASS ThreadInformationClass,
    _In_ LPVOID ThreadInformation,
    _In_ DWORD ThreadInformationSize);
```

These functions are fairly generic, accepting several possible values for the `PROCESS_INFORMATION_CLASS` and `THREAD_INFORMATION_CLASS` enumerations. For memory priority, the enums are `ProcessMemoryPriority` and `ThreadMemoryPriority`, where the value of the priority is between 1 and 5. This means only lowering memory priority is permitted.

> Figure 12-7 shows priorities 6 and 7. These are used by services such as *superfetch* that attempt to load code and data before the processes that use this memory even start. Such pages should remain in RAM for as long as possible, since they may serve more than a single process.

For a process, the handle must have the `PROCESS_SET_INFORMATION` access mask. For a thread, the handle must have the `THREAD_SET_INFORMATION` access mask.

Here is an example where the current thread reduces its own memory priority to 2:

```
DWORD priority = 2;
::SetThreadInformation(::GetCurrentThread(), ThreadMemoryPriority,
    &priority, sizeof(priority));
```

Naturally, the complementary functions exist as well:

```
BOOL GetProcessInformation(
    _In_ HANDLE hProcess,
    _In_ PROCESS_INFORMATION_CLASS ProcessInformationClass,
    _Out_writes_bytes_(ProcessInformationSize) LPVOID ProcessInformation,
    _In_ DWORD ProcessInformationSize);
BOOL GetThreadInformation(
    _In_ HANDLE hThread,
    _In_ THREAD_INFORMATION_CLASS ThreadInformationClass,
    _Out_writes_bytes_(ThreadInformationSize) LPVOID ThreadInformation,
    _In_ DWORD ThreadInformationSize);
```

The above "set" functions are thin wrappers over the native `NtSetInformationProcess` and `NtSetInformationThread` functions. Similarly, the "get" functions are thin wrappers around `NtQueryInformationProcess` and `NtQueryInformationThread`.

Process Memory Counters

Process-related memory counters in *Task Manager* are somewhat confusing. The first issue with *Task Manager*, is the default memory counter shown in the *Details* tab: *Memory (private working set)* or *Memory (active private working set)* (the latter appeared in Windows 10 version 1903). Let's dissect these terms:

- Working Set - physical memory used by the process
- Private - memory private to the process (not shared)
- Active - does not include UWP processes that are in the background

The problem with these counters is the *Working Set* part. They indicate the private memory that is currently in RAM. However, this is an unstable counter, that may go up and down depending on the process activity. If you're trying to determine how much memory is committed (allocated) by the process or if a process leaks memory, these are **not** the counters to look at.

The fact that these counters show private memory only is generally a good thing, since shared memory (such as used by DLL code) is constant, so there is little anyone can do about it. Private memory is the memory controlled by the process.

So what is the correct counter to look at? It's *Commit Size*. To make things more confusing, *Process Explorer* and *Performance Monitor* call this counter *Private Bytes*. Figure 12-8 shows *Task Manager*, which *Commit Size* and *Active Private Working Set* side by side, sorted by commit size.

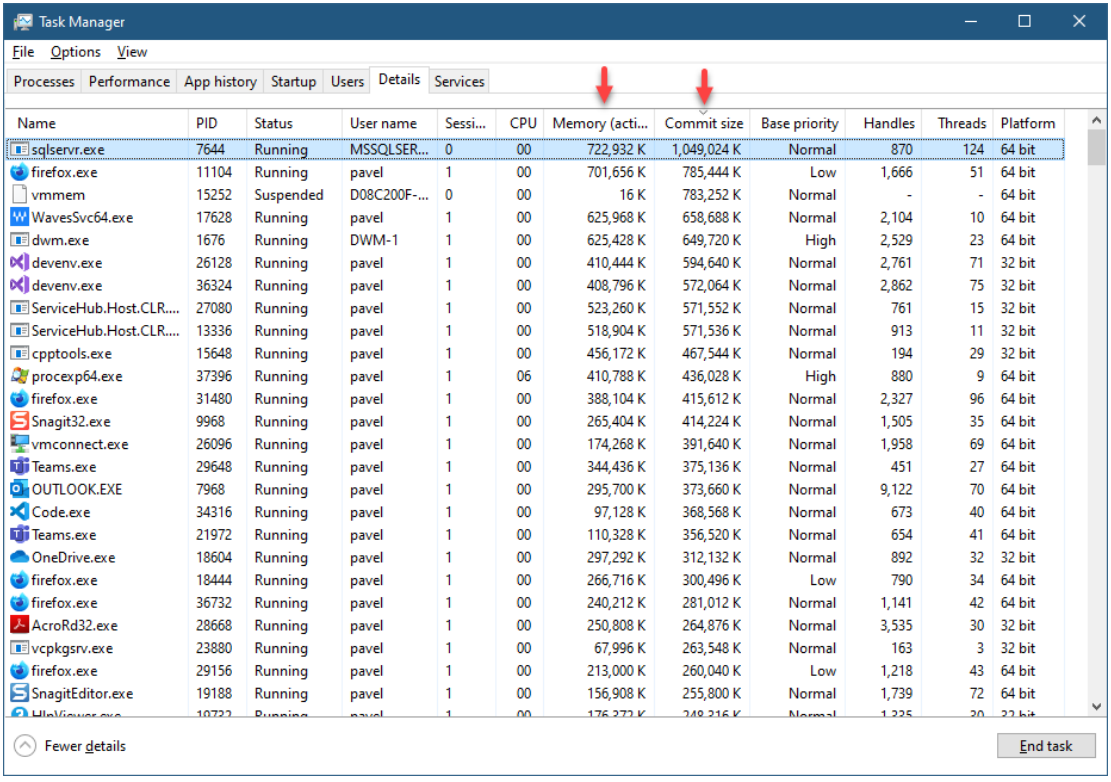


Figure 12-8: ***Task Manager***

Commit size is also about private memory, so it's on equal footing as *private working set*. The difference is the memory that is *not* in the working set. If both counters are close, it means either the process is fairly active, and works with most of its memory, or that Windows is not low on free memory, so the memory manager is not quick to remove pages from working sets.

In some cases, the difference between the two counters can be quite large. In figure 12-8, the process *Code* (PID 34316) has most of its committed memory not part of its working set. This is why looking at *private working set* counter can be misleading. It looks like that process consumes roughly 97 MB, but it actually consumes about 368 MB of memory. Granted, currently in RAM it only uses 97 MB, but the committed memory does consume page tables (used for mapping the committed memory), and that memory counts against the system's commit limit (shown in

figure 12-6).

The bottom line: use the *Commit size* column in *Task Manager* to determine the memory consumption of a process. It does not include shared memory, but this is not important (in most cases).

With *Process Explorer*, the equivalent of *Commit size* is *Private Bytes*. Both *Task Manager* and *Process Explorer* contain more columns related to memory (*Procss Explorer* has more than *Task Manager*). One column in particular, has no close equivalent, and that is the *Virtual Size* column, shown in figure 12-9.

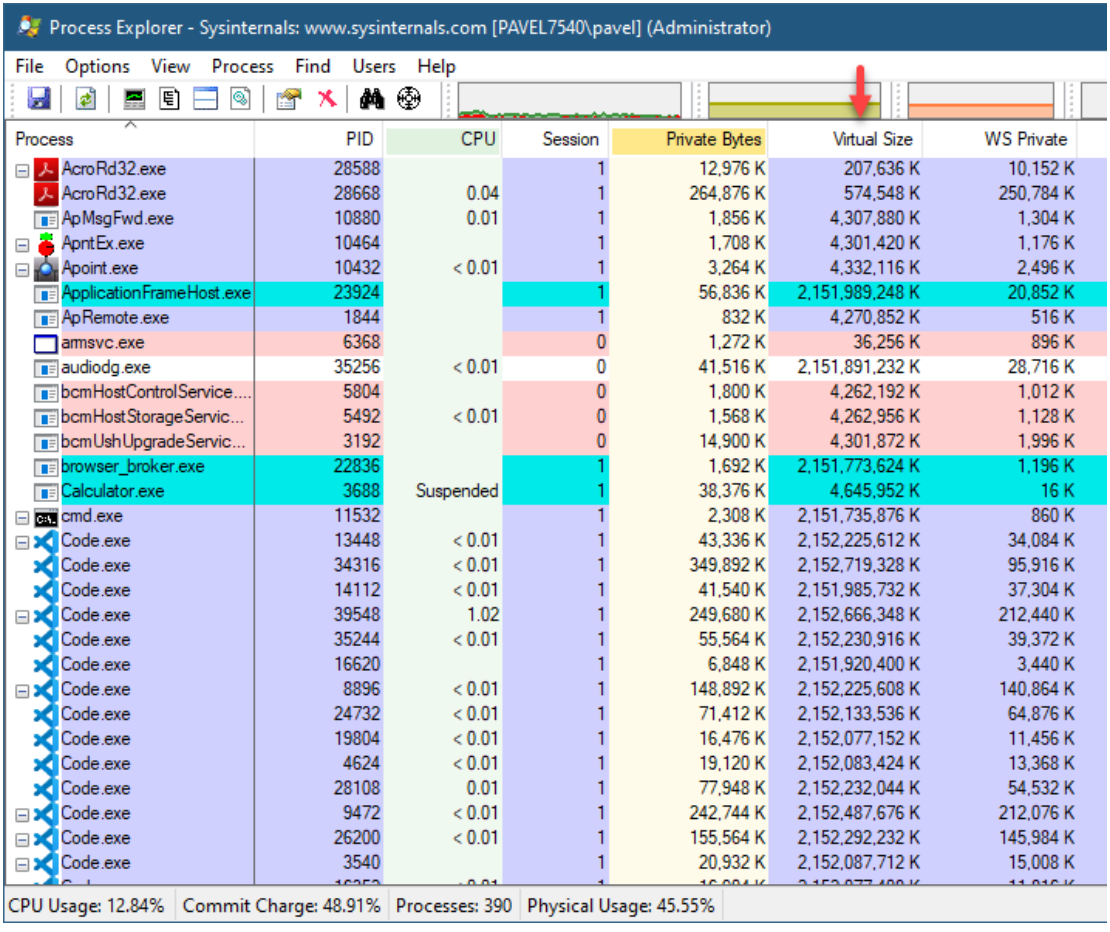


Figure 12-9: Virtual Size column in *Process Explorer*

The *virtual Size* column counts all pages that are not in the free state - that is, committed and reserved. This is essentially the amount of address space consumed by a process. For 64-bit processes where the potential address space is 128 TB, this matters very little. For 32 bit processes, this may be an issue. Even if the committed memory is not too high, having large

reserved memory regions limits the available address space for new allocations, which can cause allocation failures even though the system as a whole may have plenty of free memory.

The previously described counters don't include reserved memory, and for good reason. Reserved memory costs very little, since from the CPU's perspective it's the same as free - no page tables are needed to describe reserved memory. In fact, starting from Windows 8.1, reserved memory costs even less.

Some of the numbers in the *Virtual Size* column in figure 12-9 may seem somewhat alarming. Several processes seem to have a virtual size of roughly 2 TB. The *Private Bytes* column shows much smaller numbers, which means most of the memory size described by *Virtual Size* is reserved. The real reason some processes have this huge reserved chunk is because of a Windows 10 security feature, called *Control Flow Guard* (CFG). You can add the CFG column in *Process Explorer*, and you'll see the tight correlation between a process that supports CFG and the huge ~2 TB reserved region.

> We will look at CFG more closely in chapter 16 (in Part 2), "Security".

Some of the global memory information is available by calling `GlobalMemoryStatusEx`:

```
typedef struct _MEMORYSTATUSEX {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    DWORDLONG ullTotalPhys;
    DWORDLONG ullAvailPhys;
    DWORDLONG ullTotalPageFile;
    DWORDLONG ullAvailPageFile;
    DWORDLONG ullTotalVirtual;
    DWORDLONG ullAvailVirtual;
    DWORDLONG ullAvailExtendedVirtual;  // always zero
} MEMORYSTATUSEX, *LPMEMORYSTATUSEX;

BOOL GlobalMemoryStatusEx(_Inout_ LPMEMORYSTATUSEX lpBuffer);
```

The function name is somewhat misleading - only some of the members of `MEMORYSTATUSEX` refer to system-wide information. Other members are related to the calling process.
The `dwLength` member of `MEMORYSTATUSEX` must be filled with the size of the structure prior to the call. Here is a drill down of the members:

- `dwMemoryLoad` - a number between 0 and 100 indicating the physical memory load on the system (in percent)
- `ullTotalPhys` - total system physical memory (in bytes)
- `ullAvailPhys` - available physical memory in bytes (sum of standby, free and zero lists)
- `ullTotalPageFile` - commit size in bytes for the system or the calling process (not directly related to a page file), whichever is smaller
- `ullAvailPageFile` - maximum bytes the calling process can commit
- `ullTotalVirtual` - virtual address size of the calling process
- `ullAvailVirtual` - available address space in the calling process (free pages)

There is another function that fills in the gaps for system-wide information (#include *<psapi.h>*):

```
typedef struct _PERFORMANCE_INFORMATION {
    DWORD cb;
    SIZE_T CommitTotal;
    SIZE_T CommitLimit;
    SIZE_T CommitPeak;
    SIZE_T PhysicalTotal;
    SIZE_T PhysicalAvailable;
    SIZE_T SystemCache;
    SIZE_T KernelTotal;
    SIZE_T KernelPaged;
    SIZE_T KernelNonpaged;
    SIZE_T PageSize;
    DWORD HandleCount;
    DWORD ProcessCount;
    DWORD ThreadCount;
} PERFORMANCE_INFORMATION, *PPERFORMANCE_INFORMATION;

BOOL GetPerformanceInfo (
    PPERFORMANCE_INFORMATION pPerformanceInformation,
    DWORD cb);
```

`GetPerformanceInfo` returns system-wide information only, unrelated to the calling process (but on 64-bit Windows, the values may be wrong if called from a 32-bit process, as `SIZE_T` is 32-bit in a 32-bit process). `cb` show be set to `sizeof(PERFORMANCE_INFORMATION)`.

Keep in mind that the values in `PERFORMANCE_INFORMATION` related to memory are in pages, rather than bytes. The structure is kind enough to provide the page size (which as we know is 4 KB on all supported architectures).

The structure also returns the kernel memory size, and the current number of processes, threads, and handles.

Build an application that shows as many values as possible from *Task Manager*'s *Performance* / *Memory* tab and update the values every second. Correlate with *Task Manager*.

Process Memory Map

A process' address space must contain everything that is used by the process in terms of memory: The executable's code and global data, DLLs code and global data, Threads stacks, heaps (discussed in the next chapter), and any other memory committed and/or reserved by the process. Figure 12-10 shows a typical example of a virtual address space of a process.

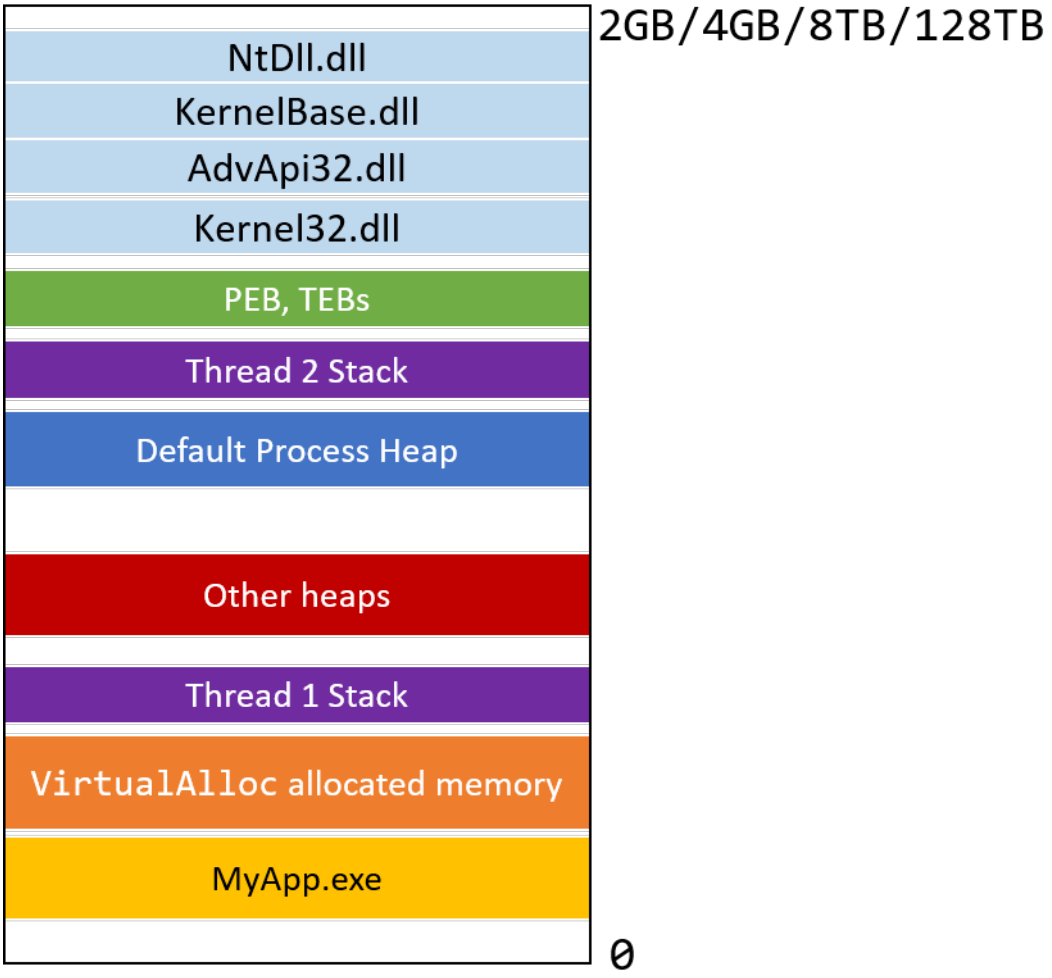


Figure 12-10: Process virtual address space example

You can probably guess the meaning of some of the parts in figure 12-10. We will discuss heaps and the `VirtualAlloc` function in the next chapter. Figure 12-10 is just a minimal example. A typical process loads dozens of DLLs and may use many threads. Frameworks such as .NET have their own DLLs and heaps, but all these seemingly different things are made of the same "stuff".

To view the actual memory map of a process, you can use the *VMMap* tool from *Sysinternals*. When *VMMap* is launched, it immediately shows a process selection dialog box, where you can select the process of interest (figure 12-11). The *Show All Processes* button allows launching *VMMap* with admin rights, allowing access to more processes. However, *VMMap* is still limited to user mode access, and cannot open protected processes.

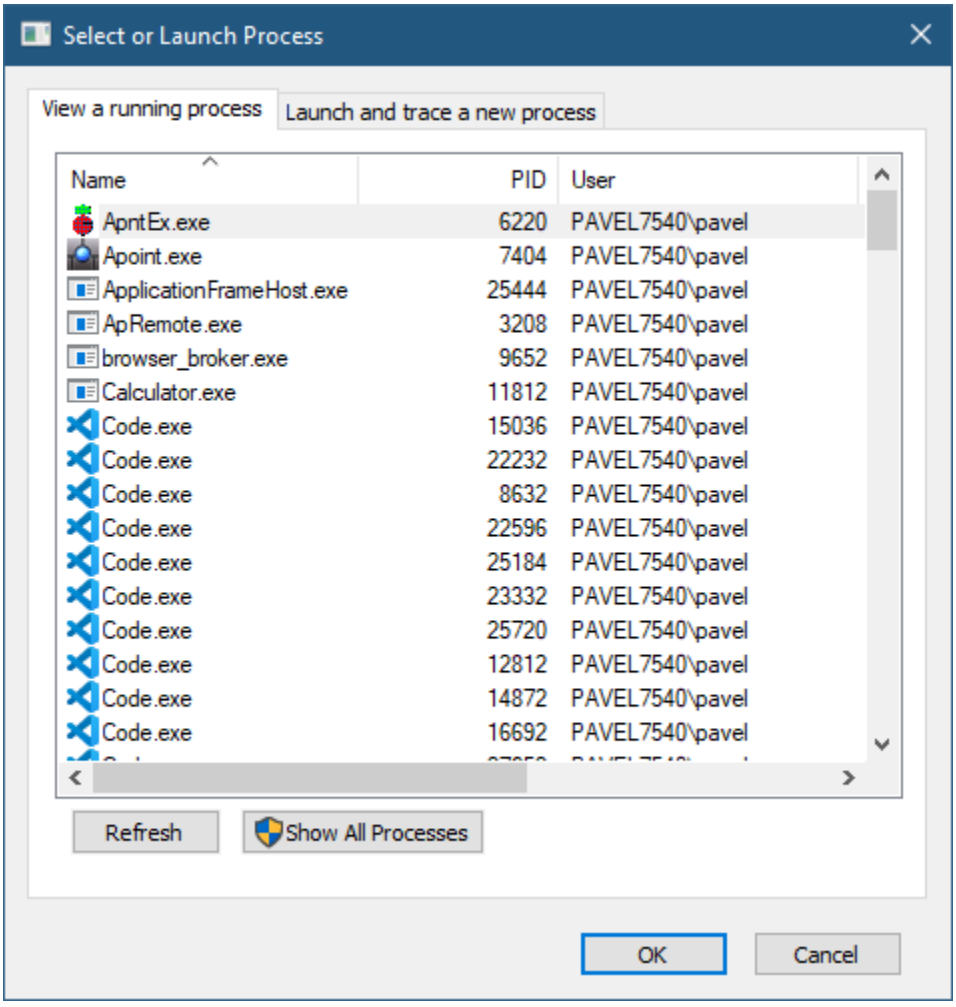


Figure 12-11: Selecting a process in *VMMap*

Once a process is selected, the main view of *VMMap* is filled by three distinct horizontal sections (figure 12-12 shows an instance of *Explorer.exe*).

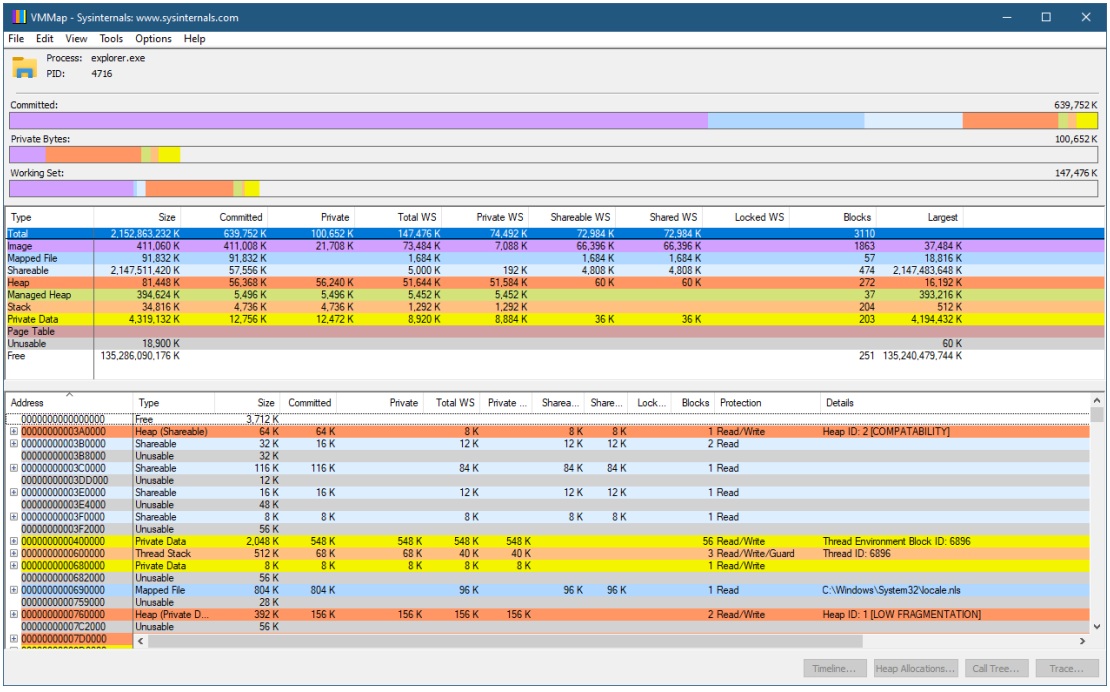


Figure 12-12: Process open in *VMMap*

The top section shows three counters:

- Committed memory - total committed memory in the process (including private and shared pages)
- Private Bytes - private committed memory
- Working set - total working set (physical memory used by private and shared pages)

Each counter is accompanied by a histogram of sorts, showing the type of memory regions contained within that counter. The region types are shown on the second section. Table 12-5 summarizes the types of regions shown by *VMMap*.

Table 12-5: Region types in *VMMap*

Type	Description
Image	Mapped images (EXEs and DLLs)
Mapped File	Mapped files (other than images)
Shareable	Memory-mapped files backed up by a page file
Heap	Memory used by heaps
Managed Heap	Memory managed by the .NET runtime (CLR or CoreCLR)
Stack	Memory for threads stacks
Private Data	Generic memory allocated with `VirtualAlloc`
Unusable	Memory blocks that cannot be used (smaller than the 64 KB allocation granularity)
Free	Free pages

The lower section shows the regions according to the currently selected region type. You can sort by any column, and dig into a region by expanding *Address* nodes, exposing the blocks within. The node itself was made by a single call to `VirtualAlloc` that reserved a chunk of memory. Then, blocks within that reserved region may have been committed, or left reserved. Each block has a common page state and protection (discussed in the next section).

The *Details* column provides more information, if available, on a reserved region or a block. *VMMap* uses various techniques to display useful information about a region or block. The simplest is calling the `GetMappedFileName` function to retrieved the file that is mapped (if any) at some address:

```
DWORD GetMappedFileName(
  _In_  HANDLE hProcess,
  _In_  LPVOID lpv,
  _Out_ LPTSTR lpFilename,
  _In_  DWORD  nSize
);
```

The process handle must have the `PROCESS_QUERY_INFORMATION` access mask bit. Given the address in the `lpv` parameter, the function returns the file name in `lpFileName` (if any). The return value from the function is the number of characters copied to `lpFileName` or zero if the function fails. The only wrinkle in this faction is that it returns the file name in NT device form (*\Device\harddiskVolume3\...*), which may require conversion to Win32 form for use with APIs such as `CreateFile`. You can use the technique from chapter 7 to do the conversion.

You can probably think of other ways that *VMMap* can extract details. For example, thread stacks can be reported by enumerating threads in the process (shown in chapter 10), and then using the native `NtQueryInformationThread` to retrieve the undocumented *Thread Environment Block* (TEB) structure, in which the stack sizes of a thread are stored.

If you need to get a summary view of a process memory usage, the PSAPI function `GetProcessMemoryInfo` can help:

```
BOOL GetProcessMemoryInfo(
    HANDLE Process,
    PPROCESS_MEMORY_COUNTERS ppsmemCounters,
    DWORD cb);
```

The function accepts a process handle that must have the `PROCESS_VM_READ` access mask and either `PROCESS_QUERY_INFORMATION` or `PROCESS_QUERY_LIMITED_INFORMATION`. The current process handle (`GetCurrentProcess`) is a natural candidate, as it has a full access mask.

The information is returned in one of two structures, based on the size passed in the `cb` parameter:

```
typedef struct _PROCESS_MEMORY_COUNTERS {
    DWORD cb;
    DWORD PageFaultCount;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    SIZE_T QuotaPeakPagedPoolUsage;
    SIZE_T QuotaPagedPoolUsage;
    SIZE_T QuotaPeakNonPagedPoolUsage;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
} PROCESS_MEMORY_COUNTERS;
typedef PROCESS_MEMORY_COUNTERS *PPROCESS_MEMORY_COUNTERS;

typedef struct _PROCESS_MEMORY_COUNTERS_EX {
    DWORD cb;
    DWORD PageFaultCount;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    SIZE_T QuotaPeakPagedPoolUsage;
    SIZE_T QuotaPagedPoolUsage;
    SIZE_T QuotaPeakNonPagedPoolUsage;
    SIZE_T QuotaNonPagedPoolUsage;
```

```
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
    SIZE_T PrivateUsage;
} PROCESS_MEMORY_COUNTERS_EX;
typedef PROCESS_MEMORY_COUNTERS_EX *PPROCESS_MEMORY_COUNTERS_EX;
```

The extended structure has two extra members compared to the original. Here is a rundown of the members (all memory sizes are in bytes):

- `cb` - size of the structure
- `PageFaultCount` - number of *page fault* exceptions that occurred in the process
- `PeakWorkingSetSize` - peak physical memory consumption
- `WorkingSetSize` - current physical memory consumption
- `QuotaPeakPagedPoolUsage` - peak paged pool usage caused by the process
- `QuotaPagedPoolUsage` - current paged pool usage caused by the process
- `QuotaPeakNonPagedPoolUsage` - peak non-paged pool usage caused by the process
- `QuotaNonPagedPoolUsage` - current non-paged pool usage caused by the process
- `PagefileUsage` - current commit size of the process (private committed memory) (Windows 8+)
- `PeakPagefileUsage` - peak commit size for the process
- `PrivateUsage` - same as `PagefileUsage` (works for Windows 7 and earlier as well)

The kernel's page pool and non-paged pool values require further discussion. The kernel has two basic memory pool types: paged pool, which holds memory that can be paged to disk, and non-paged pool, which by its definition is always resident in RAM, and is never paged out to disk. These memory pools are used by the kernel and device drivers, and their current sizes is visible in tools such as *Task Manager*. However, user-mode processes also contribute to these pools indirectly by using certain APIs. For example, creating any kernel object uses paged pool for the handle itself (16 bytes on 64-bit systems, see chapter 2).

The pools values can also be viewed by *Task Manager*, by adding the appropriate columns (*Paged pool* and *NP pool*).

As a quick example, you can run the command-line tool *TestLimit* from *Sysinternals* to create as many handles as possible:

```
C:\>testlimit -h

Testlimit v5.24 - test Windows limits
Copyright (C) 2012-2015 Mark Russinovich
Sysinternals - www.sysinternals.com

Process ID: 35288

Creating handles...
Created 16711496 handles. Lasterror: 1450
```

Looking at *Task Manager* for the *TestLimit.exe* process, shows the image in figure 12-13.

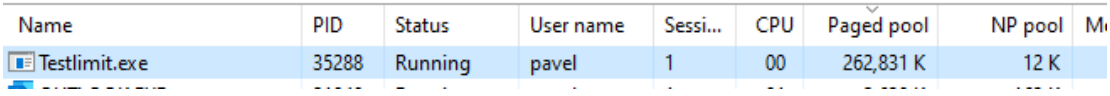


Figure 12-13: Paged pool usage with *TestLimit*

Notice the paged pool size for the process is about 256 MB. This makes sense, as around 16 M handles were created, each consuming 16 bytes.

Page Protection

Every committed page in a process virtual address space has protection flags. These can be set with the `VirtualAlloc` or `VirtualProtect` functions, discussed in the next chapter. Table 12-6 shows the page protection attributes, from which one can be specified for a committed page. Any access that violates a page's protection causes an access violation exception.

Table 12-6: Basic protection flags

Protection flag	Description
`PAGE_NOACCESS`	page is inaccessible
`PAGE_READONLY`	read access only
`PAGE_READWRITE`	read and write access
`PAGE_WRITECOPY`	copy on write access (discussed in the section "Sharing Memory")
`PAGE_EXECUTE`	execute access
`PAGE_EXECUTE_READ`	execute and read access
`PAGE_EXECUTE_READWRITE`	all possible access

Table 12-6: Basic protection flags

Protection flag	Description
PAGE_EXECUTE_WRITECOPY	execute access and copy on write

Apart from the above values, there are a few protection constants that can be added, listed in table 12-7.

Table 12-7: Optional protection flags

Protection flag	Description
PAGE_GUARD	a guard page. Any access causes a page guard exception
PAGE_NOCACHE	non-cachable page. Should only be used when memory is accessed by a kernel driver and the driver requires it
PAGE_WRITECOMBINE	An optimization that some kernel drivers are able to use. Should not be generally used
PAGE_TARGETS_INVALID	(Windows 10+) page is an invalid target for CFG (see chapter 16 for more on CFG)
PAGE_TARGETS_NO_UPDATE	(Windows 10+) do not update CFG information while protection is being changed with VirtualProtect (see chapter 16)

Page protection is set initially when calling VirtualAlloc for new allocations, and can be changed by calling VirtualProtect for exiting pages. We'll look at both these functions in the next chapter.

We've seen how guard pages are used to extend a thread's stack (chapter 5), but guard pages can also be used as a generic mechanism to detect when memory is being accessed. If such exception is raised because of a guard page access, the PAGE_GUARD flag is automatically removed, so it does not cause further exceptions when accessing the same page. You can use VirtualProtect to re-establish the guard page, if required.

Enumerating Address Space Regions

How does *VMMap* get information about the various regions? The fundamental function that returns this data is VirtualQuery for the current process, or VirtualQueryEx for any process for which a strong-enough handle can be obtained:

```
SIZE_T VirtualQuery(
    _In_opt_ LPCVOID lpAddress,
    _Out_ PMEMORY_BASIC_INFORMATION lpBuffer,
    _In_ SIZE_T dwLength);

SIZE_T VirtualQueryEx(
    _In_ HANDLE hProcess,
    _In_opt_ LPCVOID lpAddress,
    _Out_ PMEMORY_BASIC_INFORMATION lpBuffer,
    _In_ SIZE_T dwLength);
```

The process handle to `VirtualQueryEx` must have the `PROCESS_QUERY_INFORMATION` access mask. This explains why protected processes are off-limits, as this access mask is not attainable from user mode. Apart from the process handle, both functions work the same way. `lpAddress` is the address for which information is requested. The address is always rounded down to the nearest page boundary. The functions return a `MEMORY_BASIC_INFORMATION` that describes the region included in the `lpAddress` parameter:

```
typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    SIZE_T RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

The `dwLength` parameter should be set to the size of the `MEMORY_BASIC_INFORMATION` structure. The function returns the number of bytes written to the buffer (`sizeof(MEMORY_BASIC_INFORMATION)` on success) or zero if the function fails.

> `dwLength` and the return value are typed as `SIZE_T`, which is unusual and unnecessary, as the value is at most the size of `MEMORY_BASIC_INFORMATION`. It should have been typed as a simple `DWORD`.

The members of `MEMORY_BASIC_INFORMATION` are as follows:

- `BaseAddress` - the address of the start of this block. It's equal to the `lpAddress` parameter if no rounding down was necessary

- `AllocationBase` - the original region base address allocated with `VirtualAlloc`. `BaseAddress` is contained within this region. `AllocationBase` is how *VMMap* chunks blocks into regions.
- `AllocationProtect` - the original page protection specified in the `VirtualAlloc` call (see previous section).
- `RegionSize` - the size of this block. The size spans pages while the state (`State` member - committed, reserved, free), protection (`Protect` member), and type (`Type` member - private, image, mapped) are the same.
- `State` - either `MEM_COMMIT`, `MEM_FREE` or `MEM_RESERVED`
- `Protect` - current protection flags
- `Type` - the types of allocation for a committed region - `MEM_IMAGE` (mapped DLL or EXE), `MEM_MAPPED` (mapped file that is not a PE), or `MEM_PRIVATE` (private data)

Some members have meaning in certain cases only. For example, `Type` and `Protect` have meaning only if `State` is `MEM_COMMITTED`. `AllocationProtect` and `AllocationBase` have no meaning if `State` is `MEM_FREE`.

The *Simple VMMap* Application

The *SimpleVMMap* console application puts `QueryVirtualEx` to good use and lists the blocks for an input process with the details provided by `MEMORY_BASIC_INFORMATION` and, for mapped image regions, the path to the mapped file.
The heart of the application is fairly simple. Most of the work is about displaying the information properly.

The `main` function accepts a process ID, and if not provided uses the current process as the target:

```
int main(int argc, const char* argv[]) {
    DWORD pid;
    if (argc == 1) {
        printf("No PID specified, using current process...\n");
        pid = ::GetCurrentProcessId();
    }
    else {
        pid = atoi(argv[1]);
    }
```

Now we can open a handle to the target process, and call the workhorse of the application to display the memory map:

```
HANDLE hProcess = ::OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, pid);
if (!hProcess)
    return Error("Failed to open process");

printf("Memory map for process %d (0x%X)\n\n", pid, pid);

ShowMemoryMap(hProcess);

::CloseHandle(hProcess);
```

`ShowMemoryMap` starts from address zero, and walks the regions in a loop, calling a helper function to display each line of data for a region:

```
void ShowMemoryMap(HANDLE hProcess) {
    BYTE* address = nullptr;

    MEMORY_BASIC_INFORMATION mbi;

    DisplayHeaders();

    for (;;) {
        if (0 == ::VirtualQueryEx(hProcess, address, &mbi, sizeof(mbi)))
            break;

        DisplayBlock(hProcess, mbi);
        address += mbi.RegionSize;
    }
}
```

When `VirtualQueryEx` returns zero, this means we reached the end of the legal address space, and we're done. The address is incremented by the region size. Typing it as `BYTE*` allows easy pointer addition, and just like any pointer, automatically casts to `void*`, which is what `VirtualQueryEx` expects.

`DisplayHeaders` just displays the various headers in preparation for the actual data. `DisplayBlock` is called for each region:

```
void DisplayBlock(HANDLE hProcess, MEMORY_BASIC_INFORMATION& mbi) {
    printf("%s", mbi.AllocationBase == mbi.BaseAddress ? "*" : " ");
    printf("0x%16p", mbi.BaseAddress);
    printf(" %11llu KB", mbi.RegionSize >> 10);
    printf(" %-10s", StateToString(mbi.State));
    printf(" %-17s", mbi.State != MEM_COMMIT ? "" :
        ProtectionToString(mbi.Protect).c_str());
    printf(" %-17s", mbi.State == MEM_FREE ? "" :
        ProtectionToString(mbi.AllocationProtect).c_str());
    printf(" %-8s", mbi.State == MEM_COMMIT ? MemoryTypeToString(mbi.Type) : ""\
);
    printf(" %s\n", GetDetails(hProcess, mbi).c_str());
}
```

An asterisk is added if the address is the beginning of a reservation region (allocation base is equal to the base address). `DisplayBlock` uses more helpers to convert various values to strings suitable for display. Finally, `GetDetails` uses `GetMappedFileName` to retrieve the file name for mapped pages:

```
std::string GetDetails(HANDLE hProcess, MEMORY_BASIC_INFORMATION& mbi) {
    if (mbi.State != MEM_COMMIT)
        return "";

    if (mbi.Type == MEM_IMAGE || mbi.Type == MEM_MAPPED) {
        char path[MAX_PATH];
        if (::GetMappedFileNameA(hProcess, mbi.BaseAddress, path, sizeof(path))\
 > 0)
            return path;
    }
    return "";
}
```

Be careful not to use a 32-bit process to enumerate the address space of a 64-process, as you'll get wrong results. This is because `MEMORY_BASIC_INFORMATION` holds 32-bit pointers and sizes when compiled for 32-bit. However, there are 32-bit and 64-bit specific structures named `MEMORY_BASIC_INFORMATION32` and `MEMORY_BASIC_-INFORMATION64`. The latter could be used in such a case.

Here is an example (truncated) run:

```
c:\>SimpleVMMap 42504
emory map for process 42504 (0xA608)

 Base Address           Size       State      Protection        Alloc. Protecti\
on Type       Details
-------------------------------------------------------------------------------\
---------------------
*0x0000000000000000     2097024 KB Free
*0x000000007FFE0000           4 KB Committed  Read              Read           \
   Private
 0x000000007FFE1000          32 KB Free
*0x000000007FFE9000           4 KB Committed  Read              Read           \
   Private
 0x000000007FFEA000   920090968 KB Free
*0x000000DBDDE40000           4 KB Reserved                     Read/Write
 0x000000DBDDE41000          12 KB Committed  Read/Write/Guard  Read/Write     \
   Private
 0x000000DBDDE44000        1008 KB Committed  Read/Write        Read/Write     \
   Private
 0x000000DBDDF40000         768 KB Free
...
*0x000001FB57C40000           8 KB Committed  Read/Write        Read/Write     \
   Private
 0x000001FB57C42000          56 KB Free
*0x000001FB57C50000         804 KB Committed  Read              Read           \
   Mapped   \Device\HarddiskVolume3\Windows\System32\locale.nls
 0x000001FB57D19000          28 KB Free
*0x000001FB57D20000          68 KB Committed  Read              Read           \
   Mapped   \Device\HarddiskVolume3\Windows\System32\C_1252.NLS
 0x000001FB57D31000         124 KB Free
...
 0x00007FF5A47B0000     8779072 KB Free
*0x00007FF7BC500000           4 KB Committed  Read              Execute/WriteCo\
py Image    \Device\HarddiskVolume3\Windows\System32\cmd.exe
 0x00007FF7BC501000         196 KB Committed  Execute/Read      Execute/WriteCo\
py Image    \Device\HarddiskVolume3\Windows\System32\cmd.exe
 0x00007FF7BC532000          44 KB Committed  Read              Execute/WriteCo\
py Image    \Device\HarddiskVolume3\Windows\System32\cmd.exe
 0x00007FF7BC53D000           8 KB Committed  Read/Write        Execute/WriteCo\
py Image    \Device\HarddiskVolume3\Windows\System32\cmd.exe
 0x00007FF7BC53F000           8 KB Committed  WriteCopy         Execute/WriteCo\
```

```
py Image    \Device\HarddiskVolume3\Windows\System32\cmd.exe
...
```

More Address Space Information

If you take another look at *VMMap* (figure 12-12), you'll see that for every block *VMMap* provides information such as working set (RAM) sizes for the block (private, shared, sharable). This extra information is available with another PSAPI function, `QueryWorkingSetEx`:

```
BOOL QueryWorkingSetEx(
    _In_  HANDLE hProcess,
    _Out_ PVOID pv,
    _In_  DWORD cb);
```

The process handle must have the `PROCESS_QUERY_INFORMATION` access mask. `pv` must point to one or more structures of type `PSAPI_WORKING_SET_EX_INFORMATION`:

```
typedef union _PSAPI_WORKING_SET_EX_BLOCK {
    ULONG_PTR Flags;
    union {
        struct {
            ULONG_PTR Valid : 1;
            ULONG_PTR ShareCount : 3;
            ULONG_PTR Win32Protection : 11;
            ULONG_PTR Shared : 1;
            ULONG_PTR Node : 6;
            ULONG_PTR Locked : 1;
            ULONG_PTR LargePage : 1;
            ULONG_PTR Reserved : 7;
            ULONG_PTR Bad : 1;

#if defined(_WIN64)
            ULONG_PTR ReservedUlong : 32;
#endif
        };
        struct {
            ULONG_PTR Valid : 1;          // Valid = 0 in this format.
            ULONG_PTR Reserved0 : 14;
            ULONG_PTR Shared : 1;
            ULONG_PTR Reserved1 : 15;
            ULONG_PTR Bad : 1;
```

```
#if defined(_WIN64)
            ULONG_PTR ReservedUlong : 32;
#endif
        } Invalid;
    };
} PSAPI_WORKING_SET_EX_BLOCK, *PPSAPI_WORKING_SET_EX_BLOCK;

typedef struct _PSAPI_WORKING_SET_EX_INFORMATION {
    PVOID VirtualAddress;
    PSAPI_WORKING_SET_EX_BLOCK VirtualAttributes;
} PSAPI_WORKING_SET_EX_INFORMATION, *PPSAPI_WORKING_SET_EX_INFORMATION;
```

The structure may look complex, but it's nothing more than an address of interest (`VirtualAddress`) and a set of flags (`VirtualAttributes`). Here is the rundown of the flags:

- `Valid` - set if the page is in the process' working set. If clear, the `Invalid` part of the union should be consulted - most of the other flags are meaningless.
- `Shared` - indicates if the page is shareable. If clear, the page is private.
- `ShareCount` - if `Shared` is set, indicates the share count. If greater than 1, the page is being shared. The maximum count of this member is 7, so it should not be treated as an accurate share count.
- `Win32Protection` - basic protection flags, also available with `VirtualQuery(Ex)`.
- `Node` - the NUMA node this page is part of.
- `Locked` - if set, locked in physical memory (see next chapter for more on locked pages.)
- `LargePage` - if set, this is a large page. (see next chapter for more on large pages.)
- `Bad` - if set, this is a bad page (from a hardware perspective). Technically, can also represent a memory enclave (Windows 10+) (see next chapter for more on enclaves.)

The *SimpleVMMap2* application is an enhancement of *SimpleVMMap*, that adds for each committed block its working set attributes (if resident).

The `DisplayBlock` function has an added call for querying a committed page range:

```
if (mbi.State == MEM_COMMIT)
    DisplayWorkingSetDetails(hProcess, mbi);
```

`DisplayWorkingSetDetails` does all the hard work:

```
void DisplayWorkingSetDetails(HANDLE hProcess, MEMORY_BASIC_INFORMATION& mbi) {
    auto pages = mbi.RegionSize >> 12;
    PSAPI_WORKING_SET_EX_INFORMATION info;
    ULONG attributes = 0;
    void* address = nullptr;
    SIZE_T size = 0;
    for (decltype(pages) i = 0; i < pages; i++) {
        info.VirtualAddress = (BYTE*)mbi.BaseAddress + (i << 12);

        if (!::QueryWorkingSetEx(hProcess, &info,
            sizeof(PSAPI_WORKING_SET_EX_INFORMATION))) {
            printf("  <<<Unable to get working set information>>>\n");
            break;
        }

        if (attributes == 0) {
            address = info.VirtualAddress;
            attributes = (ULONG)info.VirtualAttributes.Flags;
            size = 1 << 12;
        }
        else if(attributes == (ULONG)info.VirtualAttributes.Flags) {
            size += 1 << 12;
        }
        if(attributes != (ULONG)info.VirtualAttributes.Flags || i == pages - 1)\
 {
            printf("  Address: %16p (%10llu KB) Attributes: %08X %s\n",
                address, size >> 10, attributes,
                AttributesToString(
                    *(PSAPI_WORKING_SET_EX_BLOCK*)&attributes).c_str());
            size = 1 << 12;
            attributes = (ULONG)info.VirtualAttributes.Flags;
            address = info.VirtualAddress;
        }
    }
}
```

The function first calculates the number of pages in the block. Then it loops over each page and uses `QueryWorkingSetEx` to query its working set status. The only challenge in the code is not to display the state of every page, but to group all contiguous pages that share the same attributes. As long as attributes are the same, the chunk size is increased by a page and the loop continues. If the attributes change, the existing stats are displayed and the variables reset to the next values.

The last piece is the `AttributesToString` helper that returns a string representation of the attributes:

```
std::string AttributesToString(PSAPI_WORKING_SET_EX_BLOCK attributes) {
    if (!attributes.Valid)
        return "(Not in working set)";

    std::string text;
    if (attributes.Shared)
        text += "Shareable, ";
    else
        text += "Private, ";

    if(attributes.ShareCount > 1)
        text += "Shared, ";

    if (attributes.Locked)
        text += "Locked, ";
    if (attributes.LargePage)
        text += "Large Page, ";
    if (attributes.Bad)
        text += "Bad, ";
    // eliminate last command and space
    return text.substr(0, text.size() - 2);
}
```

Here is an example run:

```
C:\>SimpleVMMap2.exe 42504
Memory map for process 42504 (0xA608)

 Base Address           Size       State      Protection        Alloc. Protecti\
on Type       Details
-------------------------------------------------------------------------------\
---------------------
*0x0000000000000000     2097024 KB Free
*0x000000007FFE0000           4 KB Committed  Read              Read           \
   Private
  Address: 000000007FFE0000 (         4 KB) Attributes: 4000802F Shareable, Sha\
red
 0x000000007FFE1000          32 KB Free
```

```
*0x000000007FFE9000           4 KB Committed  Read              Read           \
   Private
  Address: 000000007FFE9000 (         4 KB) Attributes: 4000802F Shareable, Sha\
red
 0x000000007FFEA000   920090968 KB Free
*0x000000DBDDE40000           4 KB Reserved                     Read/Write
 0x000000DBDDE41000          12 KB Committed  Read/Write/Guard  Read/Write     \
   Private
  Address: 000000DBDDE43000 (         4 KB) Attributes: 00000000 (Not in workin\
g set)
...
*0x000001FB57C00000         116 KB Committed  Read              Read           \
   Mapped
  Address: 000001FB57C00000 (        56 KB) Attributes: 4000802F Shareable, Sha\
red
  Address: 000001FB57C0E000 (        16 KB) Attributes: 40008000 (Not in workin\
g set)
  Address: 000001FB57C12000 (        16 KB) Attributes: 4000802F Shareable, Sha\
red
  Address: 000001FB57C16000 (         4 KB) Attributes: 40008000 (Not in workin\
g set)
  Address: 000001FB57C17000 (        24 KB) Attributes: 4000802F Shareable, Sha\
red
...
 0x00007FF7BC559000          56 KB Committed  Read              Execute/WriteCo\
py Image    \Device\HarddiskVolume3\Windows\System32\cmd.exe
  Address: 00007FF7BC559000 (        12 KB) Attributes: 4000802F Shareable, Sha\
red
  Address: 00007FF7BC55C000 (         4 KB) Attributes: 00400000 (Not in workin\
g set)
  Address: 00007FF7BC55D000 (         4 KB) Attributes: 4000802F Shareable, Sha\
red
  Address: 00007FF7BC55E000 (        36 KB) Attributes: 40008000 (Not in workin\
g set)
...
```

Sharing Memory

Generally, processes have separate address spaces that don't mix. However, it's sometimes beneficial to be able to share memory between processes. The canonical example is DLLs. All user-mode processes need *NtDll.dll*, and most need *Kernel32.Dll*, *KernelBase.dll*, *AdvApi32.Dll*,

and many others. If each process had its own copy of the DLL in physical memory, it would quickly run out. In fact, one of the primary motivations to have DLLs in the first place, is the ability to share (at least their code). Code is by convention read-only, so can safely be shared. The same goes for executable code coming from an EXE file. If multiple processes execute based on the same image file, there is no reason not to share (at least the code). This idea is depicted in figure 12-14, where the *Kernel32.dll* is shared between two processes.

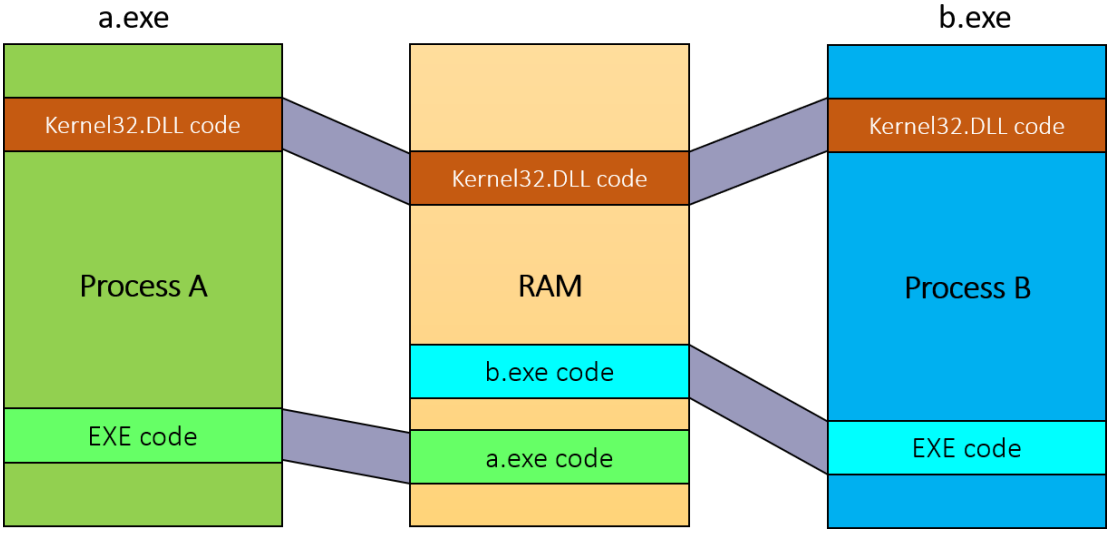


Figure 12-14: Sharing code pages

The virtual addresses of a the DLL in figure 12-14 in all processes that share it is the same. This is necessary, since not all code is relocatable. This will be discussed further in chapter 15.

What about global data? If we declare a variable in global scope like so:

```
int x;

void main() {
    x++;
    //...
}
```

And we run two instances of this executable - what would be the value of x in the second instance? The answer is 1. `x` is global to a process, not to the system. This works the same with a DLL. If a DLL declares a global variable, it's only global to each process the DLL is loaded into.

In most cases, this is what we want. This works by utilzing a page protection called *Copy on Write* (`PAGE_WRITECOPY`). The idea is that all processes using the same variable (declared in the

executable or in a DLL used by these processes) map the page this variable is located to the same physical page (figure 12-15). If a process changes the value of that variable (process A in figure 12-16), an exception is thrown, causing the memory manager to create a copy of the page and hand it to the calling process as a private page, removing the Copy-on-Write protection (page 3 in figure 12-16).

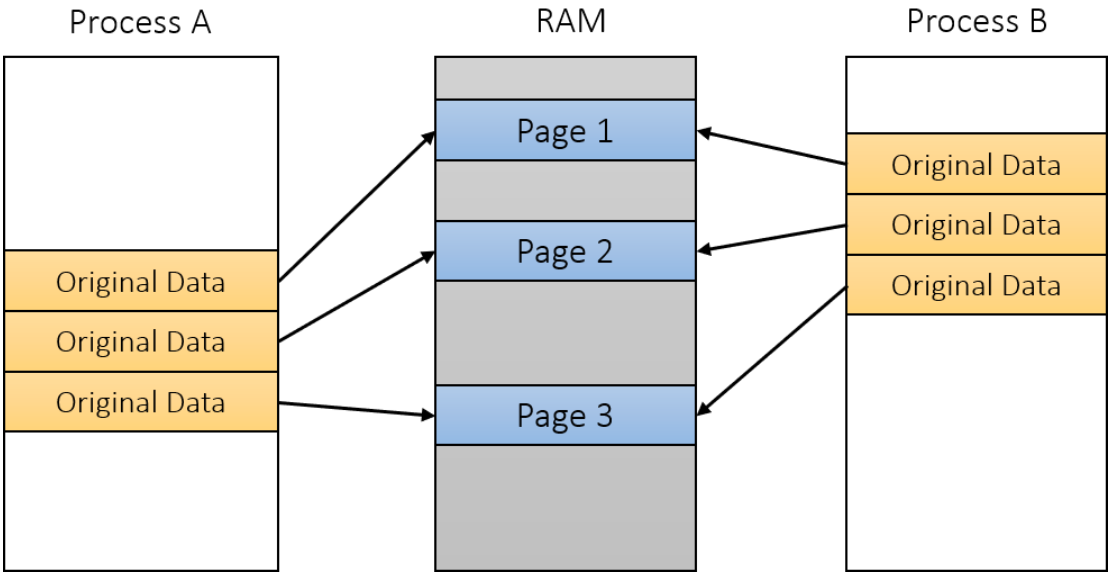


Figure 12-15: Copy on Write - before

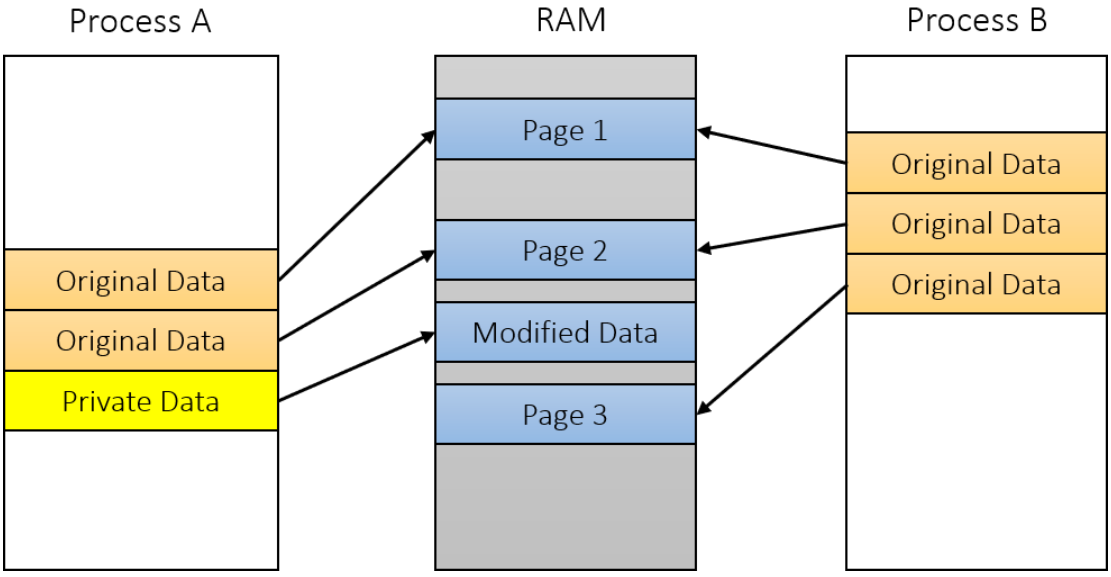


Figure 12-16: Copy on Write - after

It would have been simpler just to duplicate any global data to every process that uses it, but that would waste physical memory. If the data isn't changed, no copy needs to be made.

In some cases, sharing data between processes is desired. One relatively simple mechanism is to use global variables, but to specify that the page(s) should be protected by a normal `PAGE_READWRITE` and not `PAGE_WRITECOPY`.

This can be done by building a new data segment in the executable or DLL, and specifying its desired properties. The following shows how this can be accomplished:

```
#pragma data_seg("shared")
int x = 0;
#pragma data_seg()

#pragma comment(linker, "/section:shared,RWS")
```

The `data_seg` pragma creates a new section in the PE. Its name can be anything (up to 8 characters), called "shared" in the above code for clarity. Then, all variables that should be shared are placed in that section, and they **must be initialized explicitly**, otherwise they will not be stored in that section.

> Technically, if you have several variables, only the first needs to be explictly initialized. Still, it's best to initialize them all.

The second `#pragma` is an instruction to the linker to create the section with the attributes RWS (read, write, shared). That little "S" is the key. When the image is mapped, it will not have the `PAGE_WRITECOPY` protection, and so is shared between all processes that use the same PE.

Such variables are shared, which means concurrent access is possible. You may need to protect access to these with a mutex, for instance.

The *SimpleShare* application demonstrates the use of this technique. Figure 12-17 shows the application's dialog when it's first launched.

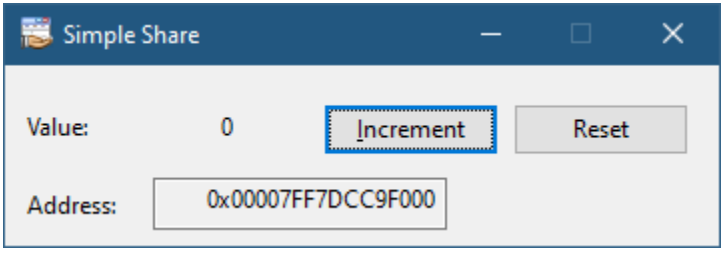


Figure 12-17: *SimpleShare* at launch

Now you can launch more instances, and click the *Increment* button to increment the value shown by 1. You'll notice all instances follow suit. Each application has a 1-second timer that just reads the current value and displays it. Figure 12-18 shows some instances with matching values.

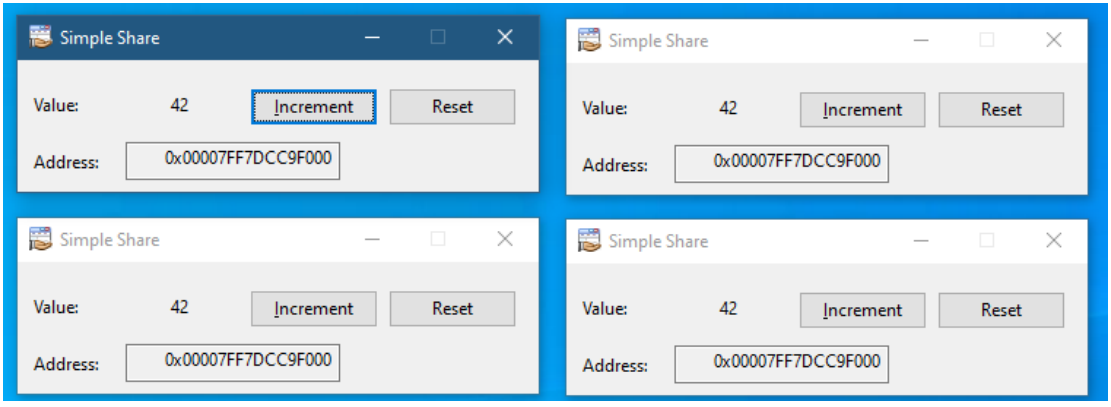


Figure 12-18: *SimpleShare* instances synchronized

The application declares a global shared variable as described above:

```
#pragma data_seg("shared")
int SharedValue = 0;
#pragma data_seg()

#pragma comment(linker, "/section:shared,RWS")
```

Every click of the *Increment* button does a simple increment of the variable:

```
LRESULT CMainDlg::OnIncrement(WORD, WORD wID, HWND, BOOL&) {
    SharedValue++;
    return 0;
}
```

The timer handler just reads the shared value and outputs it:

```
LRESULT CMainDlg::OnTimer(UINT, WPARAM id, LPARAM, BOOL&) {
    if (id == 1)
        SetDlgItemInt(IDC_VALUE, SharedValue);

    return 0;
}
```

A more general technique for sharing memory between processes is using *Memory Mapped Files*, which we'll discuss in detail in chapter 14.

Page Files

Processors can only access code and data in physical memory (RAM). If some executable is launched, Windows maps the executable's code and data (and *NTdll.dll*) into the process' address space. Then, the process' first thread starts execution. This causes the code it executes (first in *NtDll.dll* and then the executable) to be mapped to physical memory and loaded from disk so that the CPU can execute it.

Suppose that process's threads are all in a wait state, perhaps the process has a user interface, and the user minimized the application's window and didn't work with the application for a while. Windows can repurpose the RAM used by the executable for other processes that need it. Now suppose the user restores the application's window - Windows now must bring back the application's code into RAM. Where would the code be read from? The executable file itself.

This means executables and DLLs are their own backup. In fact, Windows creates a *Memory Mapped File* for executables and DLLs (which also explains why such files cannot be deleted since there is at least one open handle to the files).

What about data? If some data is not accessed for a long time (or Windows is low on free memory), the memory manager can write the data to disk - to a *page file*. A page file is used as backup for private, committed memory. Using a page file is not required - Windows can function just fine without one. But this reduces the amount of memory that can be committed at a time.

Furthermore, Windows supports up to 16 page files. They must be in different disk partitions and are named *pagefile.sys*, located at the root's partition (the files are hidden by default). Having more than one page file may be beneficial if one partition is too full, or another partition is a separate physical disk, which can increase I/O throughput.

Windows on ARM devices only supports 2 page files.

> Windows 8 and later have another special page file named *Swapfile.sys* that is used in some scenarios by UWP processes.

The commit limit shown in *Task Manager* (figure 12-6) is essentially the amount of RAM plus the current size of all page files. Each page file can have an initial size and a maximum size. If the system reaches its commit limit, the page files are increased to their configured maximum value, so the commit limit is now increased (at the possible performance degradation because of

more I/O). If the committed memory drops below the original commit limit, the page file sizes will reduce back to their initial sizes.

Configuring the page file(s) sizes is possible by going to the System properties, and then selecting *Advanced System settings*, then selecting *Settings* in the *Performance* section, then selecting the *Advanced* tab, and finally selecting *Change...* in the *Virtual Memory* section, resulting in the dialog in figure 12-19. Before clicking the last button, notice the current size of the paging files is displayed near the button.

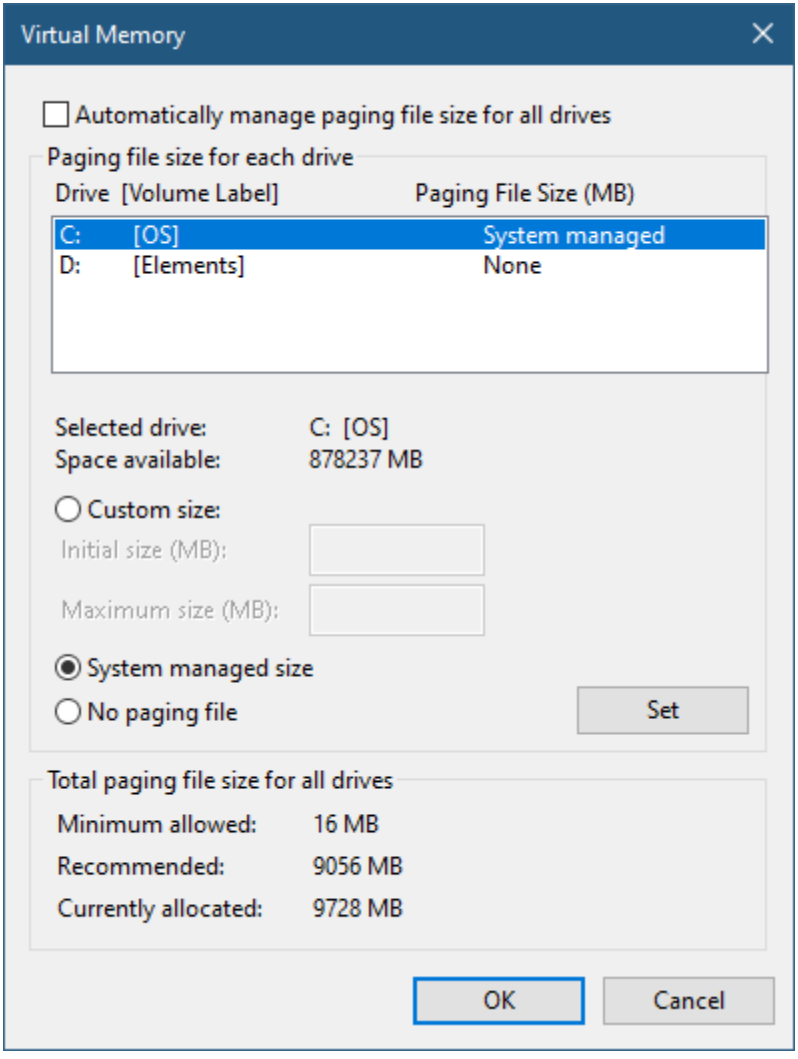


Figure 12-19: Page files configuration dialog

Normally, the top checkbox is checked, telling Windows to take care of the page file sizes automatically. Starting with Windows 10, this is my recommended choice. Earlier Windows

versions used a heuristic where the initial page file size is 1 x RAM size and the maximum size is 3 x RAM (Windows 8+ capped it at 32 GB). The problem with these heuristics is that the amount of RAM a system has is not really related to the actual work done by the user.

For example, suppose some user requires 40 GB of committed memory for her work. If the machine has 8 GB of RAM, then the page file size should be set to around 32 GB. If, on the other hand, the machine has 32 GB of RAM, only 8 GB of page file size is needed. If that system has 64 GB of RAM, no page file is needed at all!

Of course having more RAM is beneficial, since it reduces the likelihood of page file usage, but the page file size has nothing to do with the amount of RAM on the system.

Windows 10 uses a much better scheme for determining the required page file sizes if "automatically manage" is selected. It tracks committed memory usage for the past 14 days, and adjusts the size accordingly, which is of course related to what the actual user is doing, regardless of the system RAM size.

In any case, the "automatically manage" checkbox can be unchecked, allowing configuration to be made with custom initial and maximum size, or remove the paging file entirely.

> The maximum page file size is 16 TB, except on ARM, where it's limited to 4 GB.

The page file configuration is stored (like most things in Windows) in the registry at *HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles*

WOW64

The *Windows on Windows 64* (WOW64) is a software layer that allows 32-bit x86 executables to run on an x64 64-bit system without any change. In this section, we'll take a look at how this works and the implications not already discussed earlier in this chapter.

On a 64-bit Windows (x64), you'll find two sets of DLLs and executables. The native (64-bit) images are stored in the *System32* directory (e.g. *c:\Windows\System32*), while the set of 32-bit images are stored in the *SysWow64* directory (e.g. *c:\Windows\SysWow64*). This is required because a fundamental rule Windows enforces is that a 32-bit process cannot load a 64-bit DLL and vice versa. This makes sense, since pointer sizes are different and address ranges are different - this just cannot work correctly.

> The exception to this rule is that a DLL that only contains resources (strings, bitmap, etc.) with no code can be loaded by any process.

The net result of these restrictions is that a 32-bit executable must link and load 32-bit DLLs. This is why there is a separate directory (*SysWow64*) that contains all the Windows-provided 32-bit DLLs.

> The *SysWow64* directory also contains 32-bit versions of standard applications, such as *Notepad*, *mspaint*, *cmd*, etc.

The problem with the use of 32-bit DLLs is that the kernel is still 64-bit, which means any system call must be called as 64-bit, normally provided by the 64-bit NtDll. Furthermore, the standard 32-bit NtDll on a 32-bit system invokes a system call directly, which cannot work. This means that a 64-bit system has a special 32-bit NtDll that **does not** invoke system calls. Instead it calls into some helper DLLs that provide the necessary system call translation (changing pointer sizes and other arguments), and then calling on the real 64-bit *NtDll*. This architecture is shown in figure 12-20.

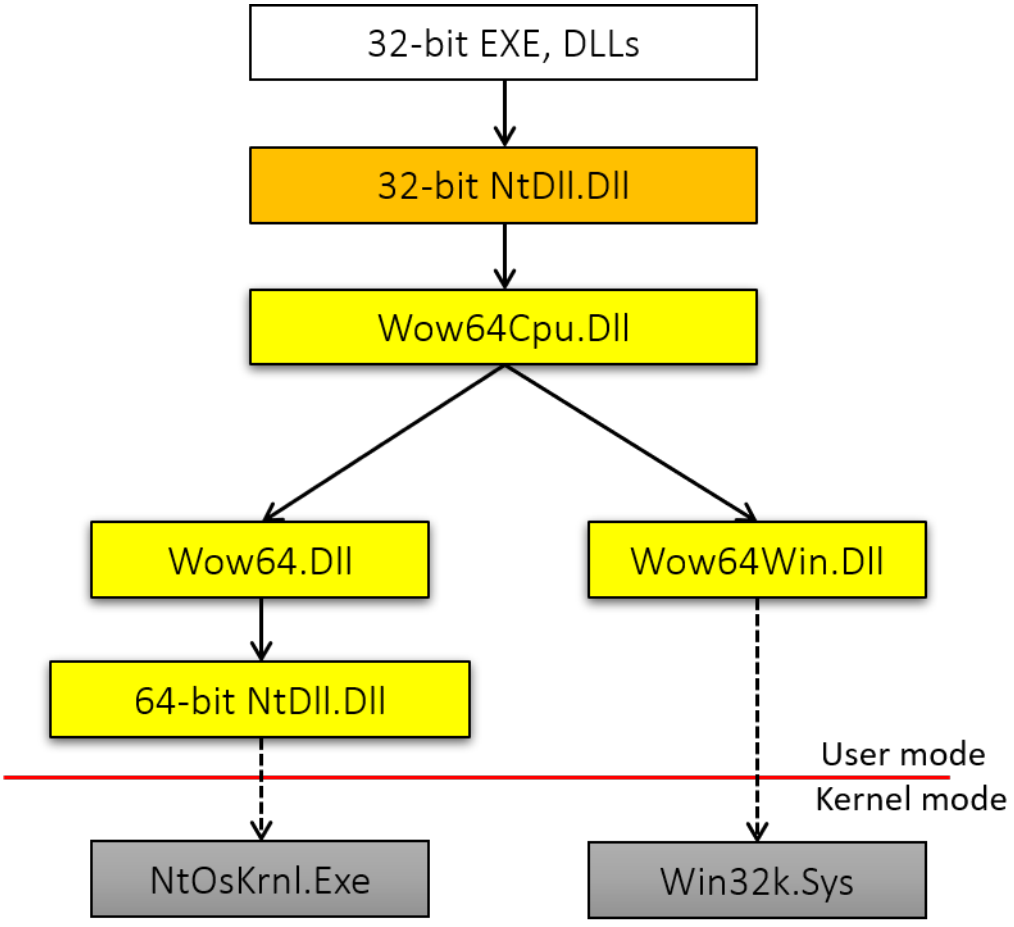


Figure 12-20: WOW64 architecture

The fact that 32-bit and 64-bit DLLs are loaded into the same process may be confusing. From the kernel's perspective, there is no such thing as a true 32-bit process. The 32-bit code has no idea there is more to the 4 GB maximum address space it can see. It's like having two-dimensional creatures living on a table, that have no idea there is a third dimension.

Figure 12-21 shows a screenshot from *Process Explorer* showing the two versions of *NtDll* in the same process. One is from *System32* and loaded in a high address, well above 4 GB, and the other from *SysWow64*, loaded below 2 GB (the *Base* column indicates the address into which an image is loaded).

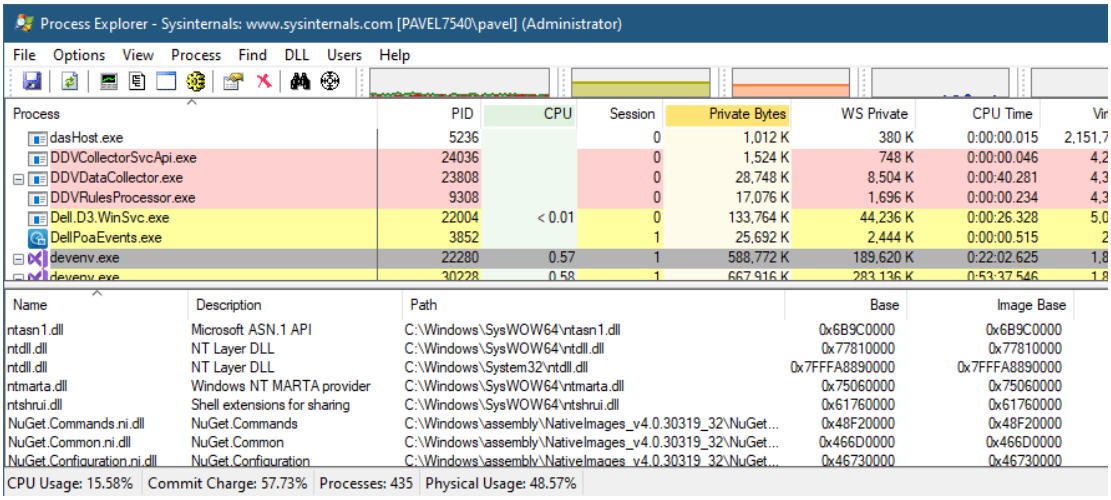


Figure 12-21: Two *NtDll.Dll* images loaded into a 32-bit process

> You can also find the three translation-related DLLs in the process address space as well.

There are other changes for 32-bit WOW64 processes. There are two stacks per thread, along with two *Thread Environment Block* structures per thread. One is while the thread is in 32-bit mode, and the other for when the thread moves to the 64-bit environment when the "translation layer" DLLs are invoked. Although interesting from an architecture perspective, these changes should not affect the way code executes.

There are some APIs that do not work in a WOW64 process. The *Address Windowing Extension* (AWE) and the functions `ReadFileScatter` and `WriteFileGather`. Fortunately, these are fairly rare, so unlikely to be problematic in practice.

WOW64 Redirections

What happens if a 32-bit WOW64 process calls `GetSystemDirectory`? or perhaps loads a DLL directly from a path like *c:\Windows\System32\ws2.dll*? As discussed earlier, a 32-bit process cannot load a 64-bit DLL. But the executable has no idea it's running on a 64-bit system - this is the point of WOW64.

Windows provides *file system redirection*, so that any attempt to get to *System32* is automatically and transparently redirected to *Syswow64*. This works with explicit paths as well as function calls

such as `GetSystemDirectory`. A similar redirection occurs when accessing the *Program Files* directory - it's redirected to *Program Files (x86)*.

A thread can opt-out of this redirection temporarily by calling `Wow64DisableWow64FsRedirection`:

```
BOOL Wow64DisableWow64FsRedirection(_Out_ PVOID* OldValue);
```

The `OldValue` parameter is an opaque value that should be passed to `Wow64RevertWow64FsRedirection` to re-enable redirection:

```
BOOL Wow64RevertWow64FsRedirection(_In_ PVOID OldValue);
```

Disabling redirection can be useful for an application that is aware of WOW64 and does I/O operations that need to see things as they are. The application may have been written as 32-bit for convenience, allowing it to run unchanged on 32-bit and 64-bit systems.

Disabling redirection only works for the current thread. Other threads in the process are unaffected, unless they too request disabling file system redirection.

To access the real *System32* without a redirection, use the virtual path *c:\Windows\Sysnative*.

Another form of redirection automatically employed by the WOW64 layer is for certain Registry keys. These will be discussed in chapter 17.

Virtual Address Translation

In this last section of this chapter, we'll look at the basics of how virtual addresses are translated into physical addresses. This section is strictly optional, and can be skipped entirely. A detailed discussion of the translation tables is beyond the scope of this book; see chapter 5 for more on that in the "Windows Internals 7th edition, Part 1" book.

The translation itself is automatic, so when a CPU sees an instruction like:

```
mov eax, [100000H]
```

It knows that the address `0x100000` is virtual rather than physical (since the CPU is configured to run in *protected mode* / *long mode*). The CPU must now look at tables that were prepared beforehand by the memory manager, that describe where that page is in RAM, if at all. If it's not in RAM (marked by a zero in a Valid bit checked by the CPU in the translation tables), it raises a *page fault* exception, to be handled appropriately by the memory manager. The basic components involved in address translation are shown in figure 12-22.

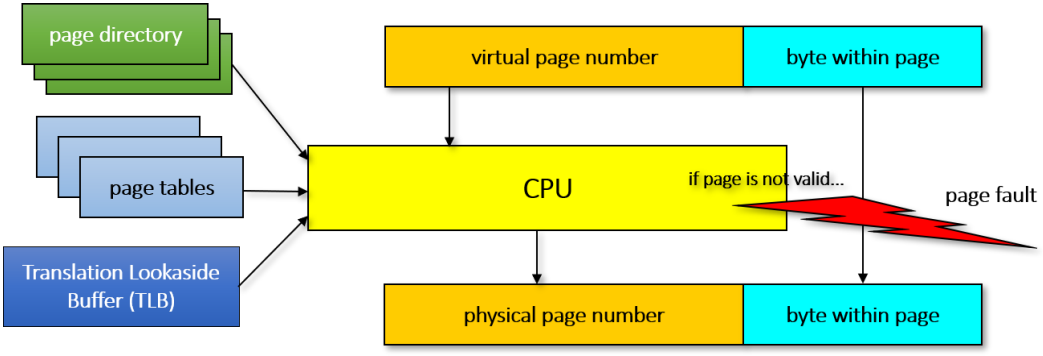


Figure 12-22: Virtual address translation

The CPU is provided with a virtual address as input, and should output (and use) a physical address. Since everything works in terms of pages, the lower 12 bit of an address (the offset within a page) are never translated, and pass as-is to the final address.

The CPU needs context for translation. Each process has a single initial structure that always resides in RAM. For 32-bit systems, it's called *Page directory pointer table*, and for 64-bit systems it's *page map level 4* (these is the Intel terminology). From this initial structure, other structures are used including page directories and finally page tables (the leaves of the translation "tree"). Page table entries are the ones pointing to the physical page address (if the valid bit is set). When a page is moved to the page file, the memory manager marks the corresponding page table entry as invalid, so that the next time the CPU encounters that page, it will raise a *page fault* exception.

Finally, the *Translation Lookaside Buffer* (TLB) is a cache of recently translated pages, so accessing these pages does not require going through multiple levels of structures for translation purposes. This cache is relatively small and is very important from a practical perspective. This emphasizes some of the things we looked at in chapter 10 related to caching and contiguous memory: working with the same range of memory address at close times is great for utilizing the TLB cache.

Summary

In this chapter, we began our journey into the world of virtual and physical memory. We looked at process' address space, page states and more. In the next chapter (and next book), we'll learn how to use memory-related APIs effectively in our applications.