3.1 Windows System Architecture

Four basic types of user-mode processes

Core subsystem Dynamic-Link Libraries (DLLs): kernel32, Advapi32, User32,Gdi32

System support library, Ntdll.dll

Windows Subsystem

Windows Subsystem for Linux

Windows Executive

Core Windows components

Windows driver layered model approach

System processes found on a Windows 10 system

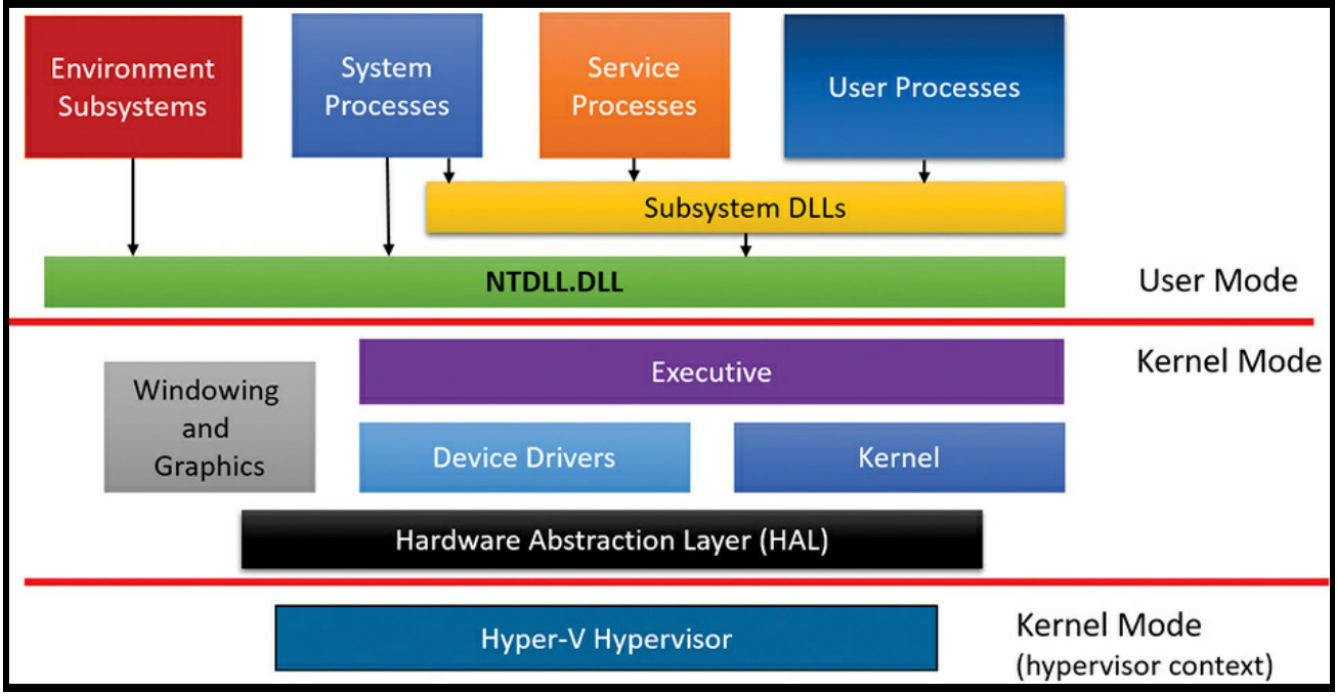
What is a minimal process

An overview of the Virtualization-Based Security architecture

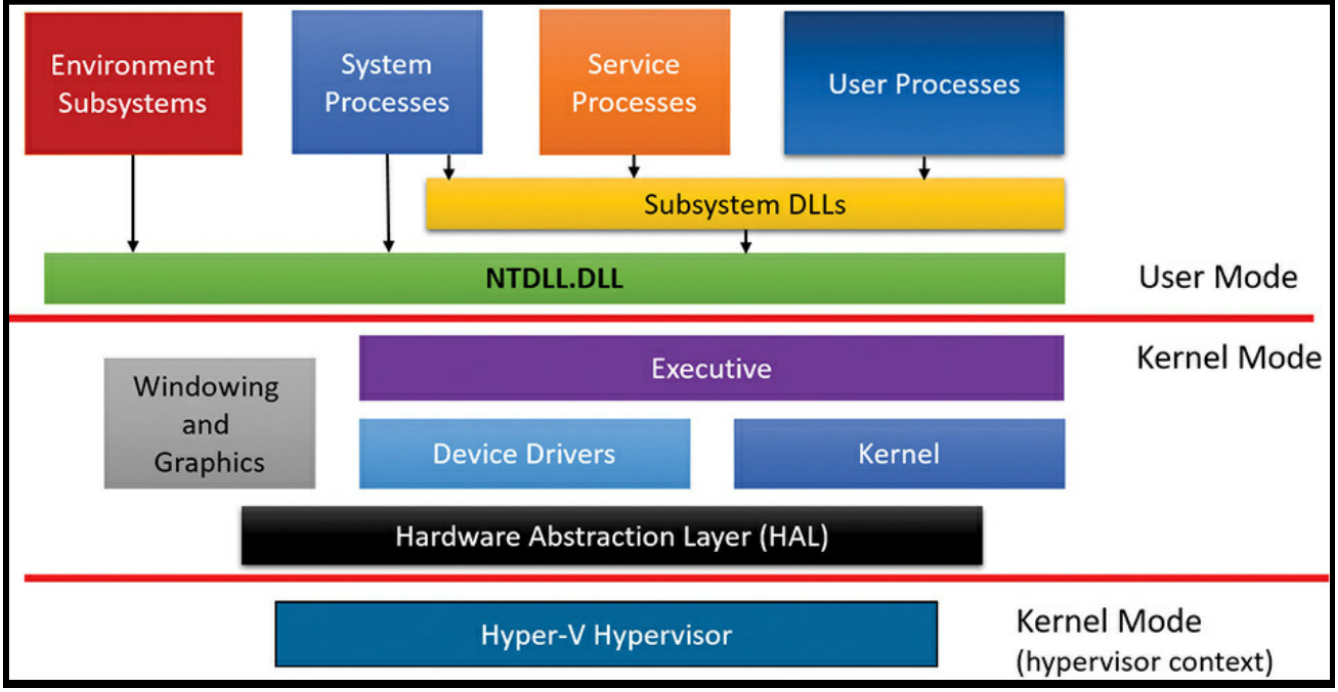
Boot process

Architecture Overview

With this brief overview of the design goals and packaging of Windows, let's take a look at the key system components that make up its architecture. A simplified version of this architecture is shown in Figure 2-1. Keep in mind that this diagram is basic. It doesn't show everything. For example, the networking components and the various types of device driver layering are not shown.



With this brief overview of the design goals and packaging of Windows, let's take a look at the key system components that make up its architecture. A simplified version of this architecture is shown in Figure 2-1. Keep in mind that this diagram is basic. It doesn't show everything. For example, the networking components and the various types of device driver layering are not shown.



Four Basic User Processes

User processes These processes can be one of the following types: Windows 32-bit or 64-bit (Windows Apps running on top of the Windows Runtime in Windows 8 and later are included in this category), Windows 3.1 16-bit, MS-DOS 16-bit, or POSIX 32-bit or 64-bit. Note that 16-bit applications can be run only on 32-bit Windows, and that POSIX applications are no longer supported as of Windows 8.

Service processes These are processes that host Windows services, such as the Task Scheduler and Print Spooler services. Services generally have the requirement that they run independently of user logons. Many Windows server applications, such as Microsoft SQL Server and Microsoft Exchange Server, also include components that run as services. Chapter 9, "Management mechanisms," in Part 2 describes services in detail.

System processes These are fixed, or hardwired, processes, such as the logon process and the Session Manager, that are not Windows services. That is, they are not started by the Service Control Manager.

Environment subsystem server processes These implement part of the support for the OS environment, or personality, presented to the user and programmer. Windows NT originally shipped with three environment subsystems: Windows, POSIX, and OS/2. However, the OS/2 subsystem last shipped with Windows 2000 and POSIX last shipped with Windows XP. The Ultimate and Enterprise editions of Windows 7 client as well as all of the server versions of Windows 2008 R2 include support for an enhanced POSIX subsystem called Subsystem for UNIX-based Applications (SUA). The SUA is now discontinued and is no longer offered as an optional part of Windows (either client or server).

Kernel Components

Executive The Windows executive contains the base OS services, such as memory management, process and thread management, security, I/O, networking, and inter-process communication.

The Windows kernel This consists of low-level OS functions, such as thread scheduling, interrupt and exception dispatching, and multiprocessor synchronization. It also provides a set of routines and basic objects that the rest of the executive uses to implement higher-level constructs.

Device drivers This includes both hardware device drivers, which translate user I/O function calls into specific hardware device I/O requests, and non-hardware device drivers, such as file system and network drivers.

The Hardware Abstraction Layer (HAL) This is a layer of code that isolates the kernel, the device drivers, and the rest of the Windows executive from platform-specific hardware differences (such as differences between motherboards).

The windowing and graphics system This implements the graphical user interface (GUI) functions (better known as the Windows USER and GDI functions), such as dealing with windows, user interface controls, and drawing.

The hypervisor layer This is composed of a single component: the hypervisor itself. There are no drivers or other modules in this environment. That being said, the hypervisor is itself composed of multiple internal layers and services, such as its own memory manager, virtual processor scheduler, interrupt and timer management, synchronization routines, partitions (virtual machine instances) management and inter-partition communication (IPC), and more.

Core System Files

File Name	Components
Ntoskrnl.exe	Executive and kernel
Hal.dll	HAL
Win32k.sys	Kernel-mode part of the Windows subsystem (GUI)
Hvix64.exe (Intel), Hvax64.exe (AMD)	Hypervisor
.sys files in \SystemRoot\System32\Drivers	Core driver files, such as Direct X, Volume Manager, TCP/IP, TPM, and ACPI support
Ntdll.dll	Internal support functions and system service dispatch stubs to executive functions
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Core Windows subsystem DLLs

File Name	Components
Ntoskrnl.exe	Executive and kernel
Hal.dll	HAL
Win32k.sys	Kernel-mode part of the Windows subsystem (GUI)
Hvix64.exe (Intel), Hvax64.exe (AMD)	Hypervisor
.sys files in \SystemRoot\System32\Drivers	Core driver files, such as Direct X, Volume Manager, TCP/IP, TPM, and ACPI support
Ntdll.dll	Internal support functions and system service dispatch stubs to executive functions
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Core Windows subsystem DLLs

Windows Subsystem Overview

Although Windows was designed to support multiple independent environment subsystems, from a practical perspective, having each subsystem implement all the code to handle windowing and display I/O would result in a large amount of duplication of system functions that, ultimately, would negatively affect both system size and performance. Because Windows was the primary subsystem, the Windows designers decided to locate these basic functions there and have the other subsystems call on the Windows subsystem to perform display I/O. Thus, the SUA subsystem calls services in the Windows subsystem to perform display I/O.

As a result of this design decision, the Windows subsystem is a required component for any Windows system, even on server systems with no interactive users logged in. Because of this, the process is marked as a critical process (which means if it exits for any reason, the system crashes).

Windows Subsystem

The Windows subsystem consists of the following major components:

For each session, an instance of the environment subsystem process (Csrss.exe) loads four DLLs (Basesrv.dll, Winsrv.dll, Sxssrv.dll, and Csrsrv.dll) that contain support for the following:

- Various housekeeping tasks related to creating and deleting processes and threads
- Shutting down Windows applications (through the ExitWindowsEx API)
- Containing .ini file to registry location mappings for backward compatibility
- Sending certain kernel notification messages (such as those from the Plug-and-Play manager) to Windows applications as Window messages (WM_DEVICECHANGE)
- Portions of the support for 16-bit virtual DOS machine (VDM) processes (32-bit Windows only) • Side-by-Side (SxS)/Fusion and manifest cache support
- Several natural language support functions, to provide caching

Windows Subsystem for Linux

Under this model, the idea of a Pico provider is defined, which is a custom kernel-mode driver that receives access to specialized kernel interfaces through the PsRegisterPicoProvider API. The benefits of these specialized interfaces are two-fold:

They allow the provider to create Pico processes and threads while customizing their execution contexts, segments, and store data in their respective EPROCESS and ETHREAD structures (see Chapter 3 and Chapter 4 for more on these structures).

They allow the provider to receive a rich set of notifications whenever such processes or threads engage in certain system actions such as system calls, exceptions, APCs, page faults, termination, context changes, suspension/resume, etc.

With Windows 10 version 1607, one such Pico provider is present: Lxss.sys and its partner Lxcore.sys. As the name suggests, this refers to the Windows Subsystem for Linux (WSL) component, and these drivers make up the Pico provider interface for it.

Windows Executive

The Windows executive is the upper layer of Ntoskrnl.exe. (The kernel is the lower layer.) The executive includes the following types of functions:

Functions that are exported and callable from user mode

Device driver functions that are called through the DeviceIoControl function

Functions that can be called only from kernel mode that are exported and documented in the WDK

Functions that are exported and can be called from kernel mode but are not documented in the WDK

Functions that are defined as global symbols but are not exported

Functions that are internal to a module that are not defined as global symbols

Windows Executive

The executive contains the following major components:

Configuration manager implements and manages the system registry.

Process Manager creates and terminates processes and threads.

Security Reference Monitor (SRM) enforces security policies on the local computer.

I/O Manager, implements device-independent I/O and is responsible for dispatching to the appropriate device drivers for further processing.

Plug and Play (PnP) manager determines which drivers are required to support a particular device and loads those drivers.

Power manager coordinate power events and generate power management I/O notifications to device drivers.

Windows Management Instrumentation (WMI) routines These routines enable device drivers to publish performance and configuration information and receive commands from the user-mode WMI service.

Memory manager implements virtual memory, a memory management scheme that provides a large private address space for each process that can exceed available physical memory.

Cache manager improves the performance of file-based I/O by causing recently referenced disk data to reside in main memory for quick access.

System Processes

System processes The following system processes appear on every Windows 10 system. One of these (Idle) is not a process at all, and three of them—System, Secure System, and Memory Compression—are not full processes because they are not running a user-mode executable. These types of processes are called minimal processes and are described in Chapter 3.

Idle process This contains one thread per CPU to account for idle CPU time.

System process This contains the majority of the kernel-mode system threads and handles.

Secure System process This contains the address space of the secure kernel in VTL 1, if running. **Memory Compression process** This contains the compressed working set of user-mode processes, as described in Chapter 5.

Session manager (Smss.exe).

Windows subsystem (Csrss.exe).

Session 0 initialization (Wininit.exe).

Logon process (Winlogon.exe).

Service Control Manager (Services.exe) and the child service processes it creates such as the system-supplied generic service-host process (Svchost.exe).

Local Security Authentication Service (Lsass.exe), and if Credential Guard is active, the Isolated Local Security Authentication Server (Lsaiso.exe).

Useful Tools

Process Explorer

Dependency Walker

Registry Editor

Task Manager

3.2 Build Environment and Frameworks

Visual Studio project/solution hierarchy

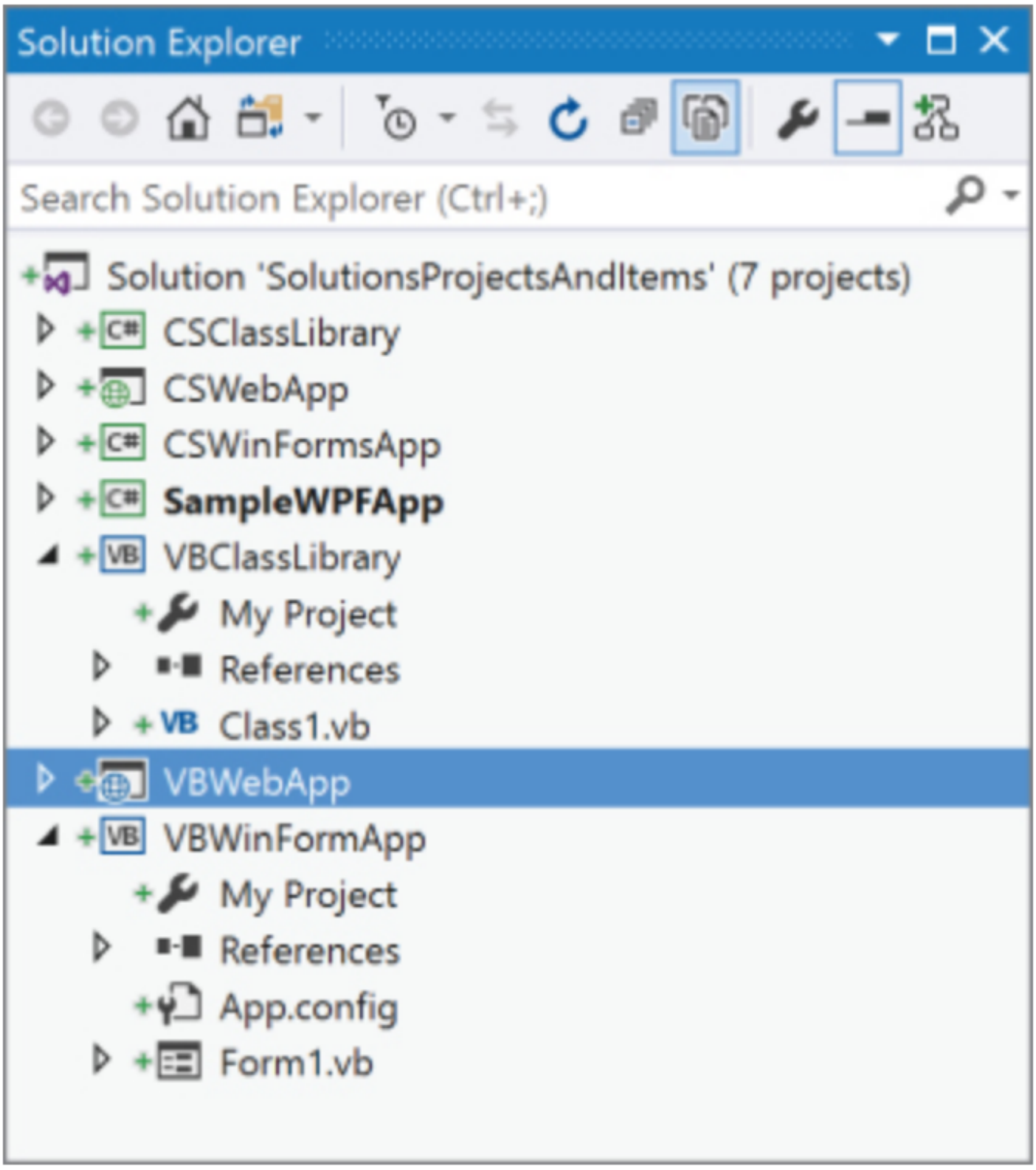
Managed Code

Microsoft Intermediate Language (MSIL)

Common Language Runtime (CLR)

dotNET Framework

Visual Studio Solution Hierarchy

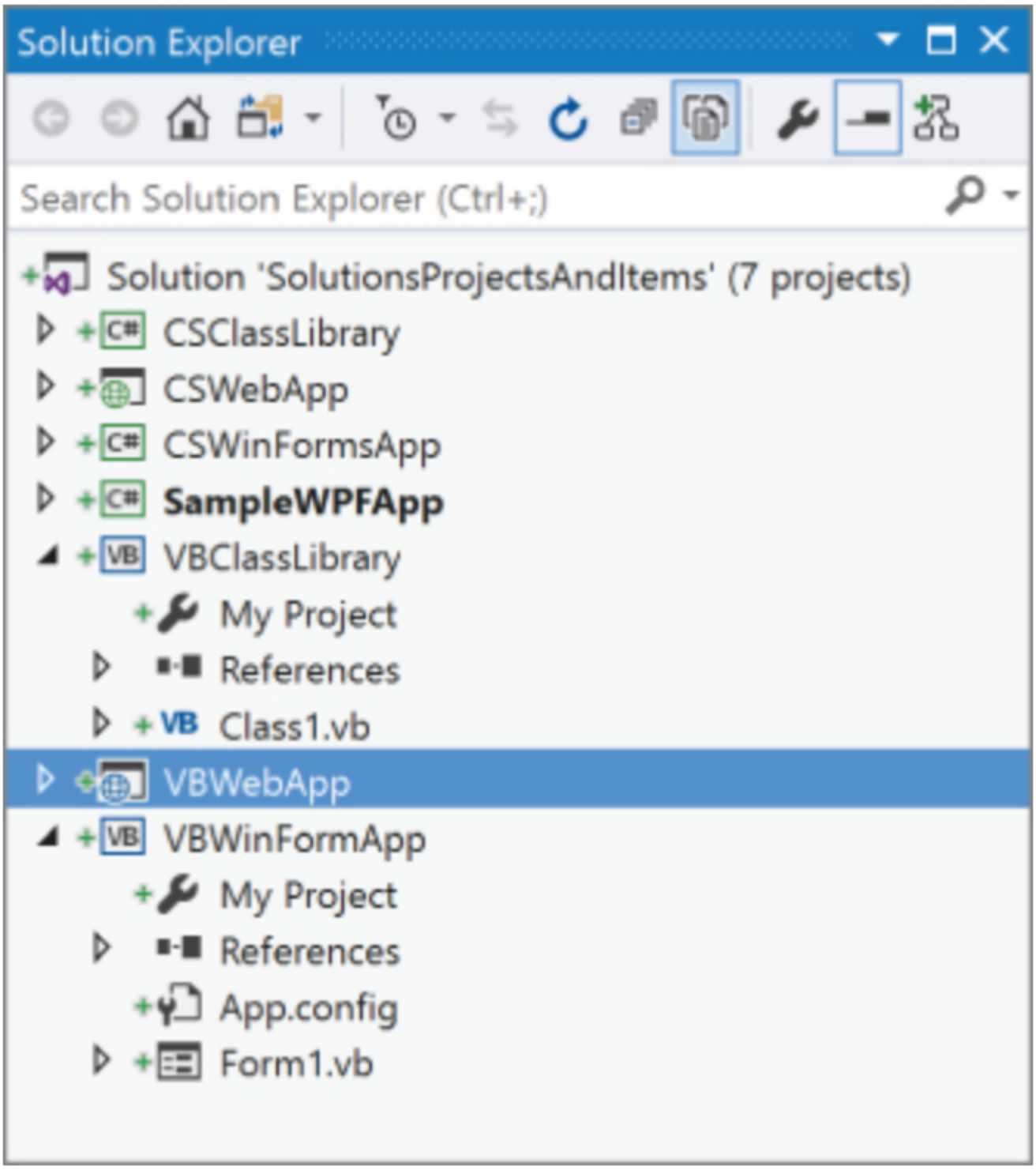


A solution can contain many different projects. Each project can contain source code as well as other files like photos, icons, and libraries.

To build a specific project you need to set it as a target. When working on a project, **adding a file to it's folder location on windows isn't the same as adding it within visual studio**. If you want to add files you should place them in the correct folder as well as manually add it within visual studio. This is because visual studio uses references to files.

What is convenient about visual studio is it will automatically create its own build process so you don't have to code and list everything like a tedious linux makefile.

Visual Studio Solution Hierarchy



In addition to tracking which files are contained within an application, solution and project files can record other information, such as how a particular file should be compiled, project settings, resources, and much more. Visual Studio 2017 includes nonmodal dialog for editing project properties, whereas solution properties still open in a separate window. As you might expect, the project properties are those properties pertaining only to the project in question, such as assembly information and references, whereas solution properties determine the overall build configurations for the application.

Managed Code

To put it very simply, managed code is just that: code whose execution is managed by a runtime. In this case, the runtime in question is called the Common Language Runtime or CLR, regardless of the implementation (for example, Mono, .NET Framework, or .NET Core/.NET 5+). CLR is in charge of taking the managed code, compiling it into machine code and then executing it. On top of that, runtime provides several important services such as automatic memory management, security boundaries, type safety etc.

Contrast this to the way you would run a C/C++ program, also called "unmanaged code". In the unmanaged world, the programmer is in charge of pretty much everything. The actual program is, essentially, a binary that the operating system (OS) loads into memory and starts. Everything else, from memory management to security considerations are a burden of the programmer.

https://docs.microsoft.com/en-us/dotnet/standard/managed-code

C++ is unmanaged because it does not produce any intermediate language and has more direct control of memory management.

Microsoft Intermediate Language

What is "Intermediate Language" (or IL for short)? It is a product of compilation of code written in high-level .NET languages. Once you compile your code written in one of these languages, you will get a binary that is made out of IL. It is important to note that the IL is independent from any specific language that runs on top of the runtime; there is even a separate specification for it that you can read if you're so inclined.

Once you produce IL from your high-level code, you will most likely want to run it. This is where the CLR takes over and starts the process of Just-In-Time compiling, or JIT-ing your code from IL to machine code that can actually be run on a CPU. In this way, the CLR knows exactly what your code is doing and can effectively manage it.

Common Language Runtime

.NET provides a run-time environment, called the common language runtime, that runs the code and provides services that make the development process easier.

Compilers and tools expose the common language runtime's functionality and enable you to write code that benefits from this managed execution environment. Code that you develop with a language compiler that targets the runtime is called managed code. Managed code benefits from features such as cross-language integration, cross-language exception handling, enhanced security, versioning and deployment support, a simplified model for component interaction, and debugging and profiling services.

https://docs.microsoft.com/en-us/dotnet/standard/clr

Managed Execution Process

The managed execution process includes the following steps:

Choosing a compiler. To obtain the benefits provided by the common language runtime, you must use one or more language compilers that target the runtime.

Compiling your code to MSIL. Compiling translates your source code into Microsoft intermediate language (MSIL) and generates the required metadata.

Compiling MSIL to native code. At execution time, a just-in-time (JIT) compiler translates the MSIL into native code. During this compilation, code must pass a verification process that examines the MSIL and metadata to find out whether the code can be determined to be type safe.

Running code. The common language runtime provides the infrastructure that enables execution to take place and services that can be used during execution.

https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process

.Net Framework

NET is a free, cross-platform, open source developer platform for building many different types of applications.

You can build with C#, F#, and Visual Basic code.

NET applications are written in the C#, F#, or Visual Basic programming language. Code is compiled into a language-agnostic Common Intermediate Language (CIL). Compiled code is stored in assemblies—files with a .dll or .exe file extension.

When an app runs, the CLR takes the assembly and uses a just-in-time compiler (JIT) to turn it into machine code that can execute on the specific architecture of the computer it is running on.

https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework

.Net Framework

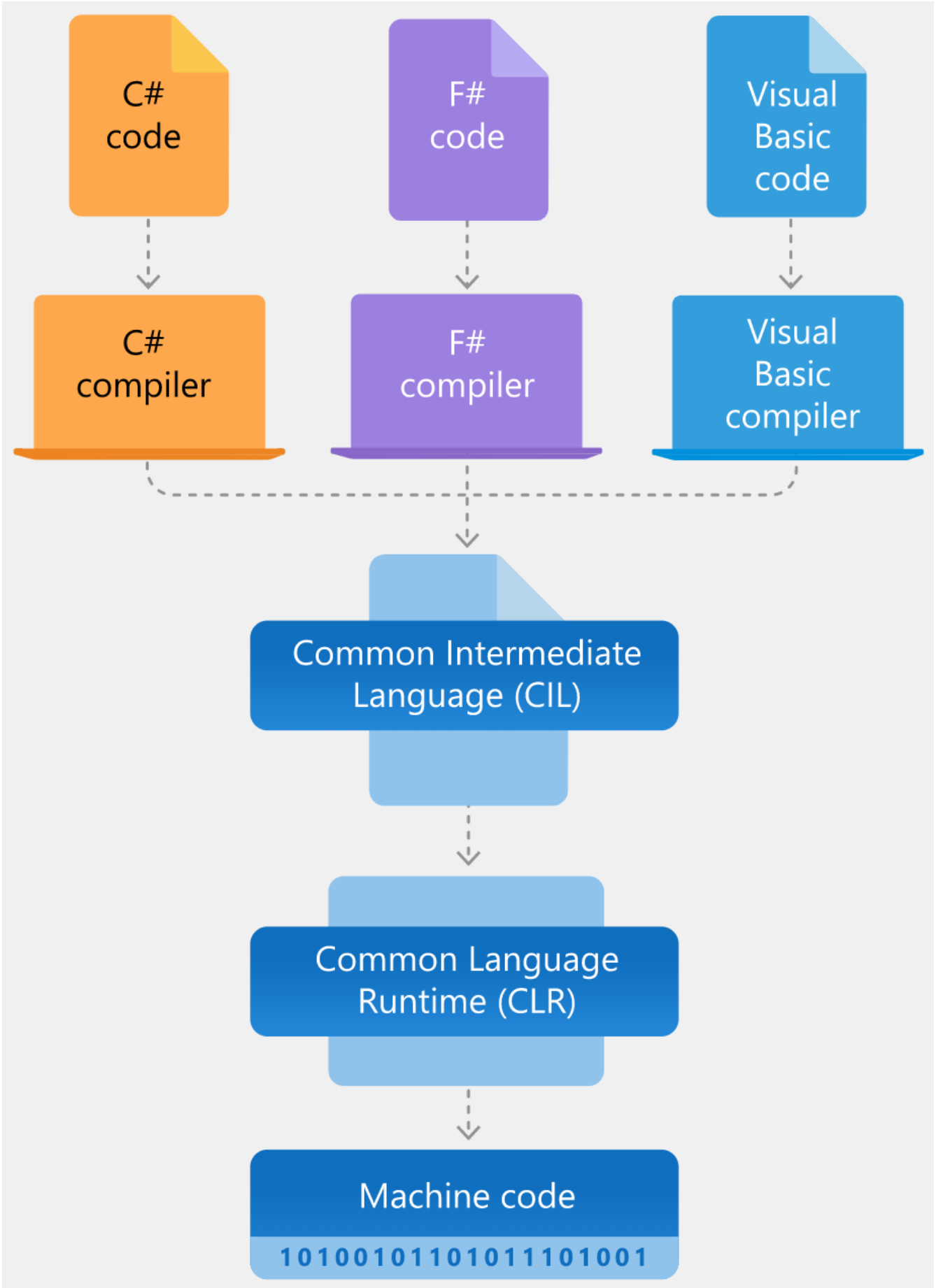


Figure 1. NET Framework is a managed execution environment for Windows that provides a variety of services to its running apps. It consists of two major components: the common language runtime (CLR), which is the execution engine that handles running apps, and the .NET Framework Class Library, which provides a library of tested, reusable code that developers can call from their own apps.

3.3 Error Handling

Return Types

Retrieve Errors

Display Error Messages

Windows Via C/C++ 5th Edition - Chapter 1

Return Types

Table 1-1. Common Return Types for Windows Functions

Data Type	Value to Indicate Failure
VOID	This function cannot possibly fail. Very few Windows functions have a return type of VOID.
BOOL	If the function fails, the return value is 0; otherwise, the return value is nonzero. Avoid testing the return value to see if it is TRUE: it is always best to test this return value to see if it is different from FALSE.
HANDLE	If the function fails, the return value is usually NULL; otherwise, the HANDLE identifies an object that you can manipulate. Be careful with this one because some functions return a handle value of INVALID_HANDLE_VALUE, which is defined as –1. The Platform SDK documentation for the function will clearly state whether the function returns NULL or INVALID_HANDLE_VALUE to indicate failure.
PVOID	If the function fails, the return value is NULL; otherwise, PVOID identifies the memory address of a data block.
LONG/DWORD	This is a tough one. Functions that return counts usually return a LONG or DWORD. If for some reason the function can't count the thing you want counted, the function usually returns 0 or –1 (depending on the function). If you are calling a function that returns a LONG/DWORD, please read the Platform SDK documentation carefully to ensure that you are properly checking for potential errors.

Error Codes

WinError.h header file contains the list of Microsoft-defined error codes.

When a Windows function fails, you should call GetLastError right away because the value is very likely to be overwritten if you call another Windows function. Notice that a Windows function that succeeds might overwrite this value with ERROR_SUCCESS.

If I detect an error in an application I've written, I might want to show the text description to the user. Windows offers a function that converts an error code into its text description. This function is called FormatMessage:

```
DWORD FormatMessage( DWORD dwFlags, LPCVOID pSource, DWORD dwMessageId, DWORD
dwLanguageId, PTSTR pszBuffer, DWORD nSize, va_list *Arguments);
```

Visual Studio also ships with a small utility called Error Lookup. You can use Error Lookup to convert an error code number into its textual description.

Custom Error Codes

To indicate failure, simply set the thread's last error code and then have your function return FALSE, INVALID_HANDLE_VALUE, NULL, or whatever is appropriate. To set the thread's last error code, you simply call VOID SetLastError(DWORD dwErrCode); passing into the function whatever 32-bit number you think is appropriate. I try to use codes that already exist in WinError.h— as long as the code maps well to the error I'm trying to report. If you don't think that any of the codes in WinError.h accurately reflect the error, you can create your own code. The error code is a 32-bit number that is divided into the fields shown in Table 1-2.

Table 1-2. Error Code Fields

Bits:	31–30	29	28	27–16	15–0
Contents	Severity	Microsoft/customer	Reserved	Facility code	Exception code
Meaning	0 = Success 1 = Informational 2 = Warning 3 = Error	0 = Microsoft-defined code 1 = customer-defined code	Must be 0	The first 256 values are reserved by Microsoft	Microsoft/customer-defined code

These fields are discussed in detail in Chapter 24. For now, the only important field you need to be aware of is in bit 29. Microsoft promises that all error codes it produces will have a 0 in this bit. If you create your own error codes, you must put a 1 in this bit. This way, you're guaranteed that your error code will never conflict with a Microsoft-defined error code that currently exists or is created in the future. Note that the Facility field is large enough to hold 4096 possible values. Of these, the first 256 values are reserved for Microsoft; the remaining values can be defined by your own application.

3.3 Exercise

Build a simple windows console application that writes to a text file. Then try to open that text file with error handling then copy it to another file. Use example code in Windows System Programming book in chapter 1.

Run the program a second time and get the message for the error code and display it to the user with the FormatMessage function. Also see: https://stackoverflow.com/questions/2812760/print-tchar-on-console

3.4 Characters and Strings

Code Pages

American National Standards Institute (ANSI)

American Standard Code for Information Interchange (ASCII)

Unicode Transformation Format 7 (UTF-7)

Unicode Transformation Format 8 (UTF-8)

Unicode Transformation Format 16 (UTF-16)

Multi-byte and double-byte character sets

Unicode (UTF-16-LE)

How wchar_t can be a type or define

Character and string types

Different strings

Character type macros

Properly converting different types of characters and strings

Code Pages

Contain specific encoding for foreign languages or for data storage and communication encoding.

Not consistent between computers, do not rely on it.

Recommended to use Unicode.

See https://docs.microsoft.com/en-us/windows/win32/intl/code-page-identifiers

ANSI & ASCII

The American National Standards Institute (ANSI) is a private, non-profit organization that administers and coordinates the U.S. voluntary standards and conformity assessment system

Each letter is represented by one 8-bit character in ASCII.

American Standard Code for Information Interchange is character encoding for the English alphabet. 95 Printable characters. From decimal codes 32 to 126.

ASCII codes 0 to 31 and 127 are control characters. Tabs, linefeed, etc.

See ansi.org or ascii-code.net

Unicode

Fulfills the need for an international standard to display character sets for foreign languages.

ASCII is limited to only English.

UTF-16 is recommended for most applications for compatibility and efficiency. Where each character is 2 bytes (16 bits).

UTF-8 uses different rule-sets of groups of 1 byte, 2 bytes, 3 bytes, and 4 bytes to encode a character. Popular outside of Windows but less efficient than UTF-16 implementation.

UTF-7 was an obsolete standard originally intended for more efficient emails.

See www.unicode.org/charts/

Multibyte Encoding

The problem is that some languages and writing systems (Japanese kanji being a classic example) have so many symbols in their character sets that a single byte, which offers no more than 256 different symbols at best, is just not enough. So double-byte character sets (DBCSs) were created to support these languages and writing systems. In a double-byte character set, each character in a string consists of either 1 or 2 bytes. With kanji, for example, if the first character is between 0x81 and 0x9F or between 0xE0 and 0xFC, you must look at the next byte to determine the full character in the string. Working with double-byte character sets is a programmer's nightmare because some characters are 1 byte wide and some are 2 bytes wide. Fortunately, you can forget about DBCS and take advantage of the support of Unicode strings supported by Windows functions and the C run-time library functions.

Nasarre, Christophe; Richter, Jeffrey. Windows via C/C++ (Developer Reference) (Kindle Locations 855-861). Pearson Education. Kindle Edition.

UTF-16-LE

Windows uses UTF-16 little endian by default.

You can use byte order mark (BOM) at the start of a file to find what type of unicode and encoding.

See: https://stackoverflow.com/questions/13499920/what-unicode-encoding-utf-8-utf-16-other-does-windows-use-for-its-unicode-da

3.4 Exercise

Convert from ASCII to UTF-16 with windows functions.

Convert from UTF-16 to ASCII.

Use CompareString to create a basic search for text on a console application. Search for word "gold" in treasure_island.txt See standard library string and find. Also see: https://www.codeproject.com/articles/2995/the-complete-guide-to-c-strings-part-i-win32-chara

3.5 Windows Kernel Objects

Kernel Objects and its Lifecycle

Handle

Sharing Objects

Kernel Objects and Lifecycle

As a Windows software developer, you create, open, and manipulate kernel objects regularly. The system creates and manipulates several types of kernel objects, such as access token objects, event objects, file objects, file-mapping objects, I/ O completion port objects, job objects, mailslot objects, mutex objects, pipe objects, process objects, semaphore objects, thread objects, waitable timer objects, and thread pool worker factory objects. The free WinObj tool from Sysinternals (located at http:// www.microsoft.com/ technet/ sysinternals/ utilities/ winobj.mspx) allows you to see the list of all the kernel object types. Notice that you have to run it as Administrator through Windows Explorer to be able to see the list on the next page.

Nasarre, Christophe; Richter, Jeffrey. Windows via C/C++ (Developer Reference) (Kindle Locations 1573-1579). Pearson Education. Kindle Edition.

Kernel Objects

Kernel Objects are given unique handles that usually are controlled by the process that created them. In Sharing Kernel Objects Across Process Boundaries, there are three methods to do so.

The easiest way to determine whether an object is a kernel object is to examine the function that creates the object. Almost all functions that create kernel objects have a parameter that allows you to specify security attribute information.

Handles

When a process first initializes, its handle table is empty. When a thread in the process calls a function that creates a kernel object, such as CreateFileMapping, the kernel allocates a block of memory for the object and initializes it. The kernel then scans the process' handle table for an empty entry. Because the handle table in Table 3-1 is empty, the kernel finds the structure at index 1 and initializes it. The pointer member will be set to the internal memory address of the kernel object's data structure, the access

Handles

```
HANDLE CreateThread( PSECURITY_ATTRIBUTES psa, size_t dwStackSize,
LPTHREAD_START_ROUTINE pfnStartAddress, PVOID pvParam, DWORD dwCreationFlags, PDWORD
pdwThreadId);

HANDLE CreateFile( PCTSTR pszFileName, DWORD dwDesiredAccess, DWORD dwShareMode,
PSECURITY_ATTRIBUTES psa, DWORD dwCreationDisposition, DWORD
dwFlagsAndAttributes,HANDLE hTemplateFile);

HANDLE CreateFileMapping( HANDLE hFile, PSECURITY_ATTRIBUTES psa, DWORD flProtect,
DWORD dwMaximumSizeHigh,DWORD dwMaximumSizeLow, PCTSTR pszName);

HANDLE CreateSemaphore( PSECURITY_ATTRIBUTES psa, LONG lInitialCount, LONG
lMaximumCount, PCTSTR pszName);
```

Handles

All functions that create kernel objects return process-relative handles that can be used successfully by any and all threads that are running in the same process. This handle value should actually be divided by 4 (or shifted right two bits to ignore the last two bits that are used internally by Windows) to obtain the real index into the process' handle table that identifies where the kernel object's information is stored. So when you debug an application and examine the actual value of a kernel object handle, you'll see small values such as 4, 8, and so on. Remember that the meaning of the handle is undocumented and is subject to change.

Whenever you call a function that accepts a kernel object handle as an argument, you pass the value returned by one of the Create* functions. Internally, the function looks in your process' handle table to get the address of the kernel object you want to manipulate and then manipulates the object's data structure in a well-defined fashion.

If you pass an invalid handle, the function returns failure and GetLastError returns 6 (ERROR_INVALID_HANDLE). Because handle values are actually used as indexes into the process' handle table, these handles are process-relative and cannot be used successfully from other processes. And if you ever tried to do so, you would simply reference the kernel object stored at the same index into the other process' handle table, without any idea of what this object would be.

Handles

When your process terminates, the system automatically scans the process' handle table. If the table has any valid entries (objects that you didn't close before terminating), the system closes these object handles for you. If the usage count of any of these objects goes to zero, the kernel destroys the object.

So your application can leak kernel objects while it runs, but when your process terminates, the system guarantees that everything is cleaned up properly.

Sharing Objects (Inherit)

Through parent-child inherited object handles. First, when the parent process creates a kernel object, the parent must indicate to the system that it wants the object's handle to be inheritable. To create an inheritable handle, the parent process must allocate and initialize a SECURITY_ATTRIBUTES structure and pass the structure's address to the specific Create function.

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof( sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE; // Make the returned handle inheritable.
HANDLE hMutex = CreateMutex(& sa, FALSE, NULL);
```

Sharing Objects (Inherit)

The next step to perform when using object handle inheritance is for the parent process to spawn the child process. This is done using the CreateProcess function:

```
BOOL CreateProcess(
    PCTSTR pszApplicationName,
    PTSTR pszCommandLine,
    PSECURITY_ATTRIBUTES psaProcess,
    PSECURITY_ATTRIBUTES psaThread,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    PVOID pvEnvironment,
    PCTSTR pszCurrentDirectory,
    LPSTARTUPINFO pStartupInfo,
    PPROCESS_INFORMATION pProcessInformation);
```

Sharing Objects (Naming)

The second method available for sharing kernel objects across process boundaries is to name the objects. Many— though not all— kernel objects can be named. For example, all of the following functions create named kernel objects:

```
HANDLE CreateMutex( PSECURITY_ATTRIBUTES psa, BOOL bInitialOwner, PCTSTR pszName);

HANDLE CreateEvent( PSECURITY_ATTRIBUTES psa, BOOL bManualReset, BOOL
bInitialState,PCTSTR pszName);
```

Sharing Objects (Naming)

```
HANDLE CreateSemaphore( PSECURITY_ATTRIBUTES psa, LONG lInitialCount, LONG
lMaximumCount, PCTSTR pszName);

HANDLE CreateWaitableTimer( PSECURITY_ATTRIBUTES psa, BOOL bManualReset, PCTSTR
pszName);

HANDLE CreateFileMapping( HANDLE hFile, PSECURITY_ATTRIBUTES psa, DWORD flProtect,
DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow, PCTSTR pszName);

HANDLE CreateJobObject( PSECURITY_ATTRIBUTES psa, PCTSTR pszName);
```

Sharing Objects (Naming)

All these functions have a common last parameter, pszName. When you pass NULL for this parameter, you are indicating to the system that you want to create an unnamed (anonymous) kernel object. When you create an unnamed object, you can share the object across processes by using either inheritance (as discussed in the previous section) or DuplicateHandle (discussed in the next section). To share an object by name, you must give the object a name.

If you don't pass NULL for the pszName parameter, you should pass the address of a zero-terminated string name. This name can be up to MAX_PATH characters long (defined as 260). Unfortunately, Microsoft offers no guidance for assigning names to kernel objects. For example, if you attempt to create an object called "JeffObj," there's no guarantee that an object called "JeffObj" doesn't already exist. To make matters worse, all these objects share a single namespace even though they don't share the same type. Because of this, the following call to CreateSemaphore always returns NULL— because a mutex already exists with the same name:.

Sharing Objects (Naming)

```
HANDLE hMutex = CreateMutex( NULL, FALSE, TEXT(" JeffObj"));

HANDLE hSem = CreateSemaphore( NULL, 1, 1, TEXT(" JeffObj"));

DWORD dwErrorCode = GetLastError();
```

Sharing Objects (Naming)

Now that you know how to name an object, let's see how to share objects this way. Let's say that Process A starts up and calls the following function:

```
HANDLE hMutexProcessA = CreateMutex( NULL, FALSE, TEXT(" JeffMutex"));
```

This function call creates a new mutex kernel object and assigns it the name "JeffMutex". Notice that in Process A's handle, hMutexProcessA is not an inheritable handle— and it doesn't have to be when you're only naming objects. Some time later, some process spawns Process B. Process B does not have to be a child of Process A; it might be spawned from Windows Explorer or any other application. The fact that Process B need not be a child of Process A is an advantage of using named objects instead of inheritance. When Process B starts executing, it executes the following code:

```
HANDLE hMutexProcessB = CreateMutex( NULL, FALSE, TEXT(" JeffMutex"));
```

When Process B's call to CreateMutex is made, the system first checks to find out whether a kernel object with the name "JeffMutex" already exists. Because an object with this name does exist, the kernel then checks the object type.

Sharing Objects (Naming)

the name "JeffMutex" is also a mutex, the system then makes a security check to see whether the caller has full access to the object. If it does, the system locates an empty entry in Process B's handle table and initializes the entry to point to the existing kernel object. If the object types don't match or if the caller is denied access, CreateMutex fails (returns NULL).

3.5 Exercise

Share a file kernel object between two processes. One program waits until you close it so that the kernel object stays alive. The other will add some text and close.

3.6 Console Input/Output

Standard Devices

Console Handles

Console APIs

Standard Devices

Like UNIX, a Windows process has three standard devices for input, output, and error reporting. UNIX uses well-known values for the file descriptors (0, 1, and 2), but Windows requires HANDLEs and provides a function to obtain them for the standard devices.

HANDLE GetStdHandle (DWORD nStdHandle)

Return: A valid handle if the function succeeds; INVALID_HANDLE_VALUE otherwise. Useful to

manipulate the console cursor position and clear the screen. With **SetConsoleCursorPosition**

Console Handles

nStdHandle must have one of these values:

- STD_INPUT_HANDLE
- STD_OUTPUT_HANDLE
- STD_ERROR_HANDLE

The standard device assignments are normally the console and the keyboard. Standard I/O can be redirected. GetStdHandle does not create a new or duplicate handle on a standard device. Successive calls in the process with the same device argument return the same handle value. Closing a standard device handle makes the device unavailable for future use within the process. For this reason, the examples often obtain a standard device handle but do not close it.

Useful Functions

CreateFile

CopyFile

DeleteFile

MoveFile

GetCurrentDirectory

CreateDirectory

Console I/O

Console I/O can be performed with **ReadFile** and **WriteFile**, but it is simpler to use the specific console I/O functions, **ReadConsole** and **WriteConsole**. The principal advantages are that these functions process generic characters (TCHAR) rather than bytes, and they also process characters according to the console mode, which is set with the SetConsoleMode function.

Console I/O

A process can have only one console at a time. Applications such as the ones developed so far are normally initialized with a console. In many cases, such as a server or GUI application, however, you may need a console to display status or debugging information. There are two simple parameterless functions for this purpose.

```
BOOL FreeConsole (VOID)

BOOL AllocConsole (VOID)
```

FreeConsole detaches a process from its console. Calling AllocConsole then creates a new one associated with the process's standard input, output, and error handles. AllocConsole will fail if the process already has a console; to avoid this problem, precede the call with FreeConsole.

Console I/O

The **ConsolePrompt** function is a useful utility that prompts the user with a specified message and then returns the user's response. There is an option to suppress the response echo. The function uses the console I/O functions and generic characters. **PrintStrings** and **PrintMsg** are the other entries in this module; they can use any handle but are normally used with standard output or error handles. The first function allows a variable-length argument list, whereas the second one allows just one string and is for convenience only. PrintStrings uses the va_start, va_arg, and va_end functions in the Standard C library to process the variable-length argument list.

Exercise 3.6

Get the CatFile program to run. It will 'cat' multiple files to screen. Use what you can and remove any code that does not work. The original author did not include the Everything.h file. Preferably remove the code related to dashS "-s" option. also see https://docs.microsoft.com/en-us/cpp/cpp/main-function-command-line-args?view=msvc-160

```
#include "Everything.h"
/* cat [options] [files] Only the -s option, which suppresses error
reporting if one of the files does not exist */
#define BUF_SIZE 0x200

static VOID CatFile(HANDLE, HANDLE);
int _tmain(int argc, LPTSTR argv[])
{
    HANDLE hInFile, hStdIn = GetStdHandle(STD_INPUT_HANDLE);
    HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
    BOOL dashS;
    int iArg, iFirstFile;

    /* iFirstFile is the argv[] index of the first input file. */
    iFirstFile = Options(argc, argv, _T("s"), &dashS, NULL);
    if (iFirstFile == argc) {   /*No input files in arg list. */
        /*Use standard input. */
        CatFile(hStdIn, hStdOut);
        return 0;
    }

    /*Process each input files. */
    for (iArg = iFirstFile; iArg < argc; iArg++) {
        hInFile =
            CreateFile(argv[iArg], GENERIC_READ, 0, NULL, OPEN_EXISTING,
                   FILE_ATTRIBUTE_NORMAL, NULL);
        if (hInFile == INVALID_HANDLE_VALUE) {
            if (!dashS)
                ReportError(_T("Error: File does not exist."),
                        0, TRUE);
        } else {
            CatFile(hInFile, hStdOut);
            if (GetLastError() != 0 && !dashS)
                ReportError(_T("Cat Error."), 0, TRUE);
            CloseHandle(hInFile);
        }
    }

    return 0;
}

static VOID CatFile(HANDLE hInFile, HANDLE hOutFile)
{
    DWORD nIn, nOut;
    BYTE buffer[BUF_SIZE];
    while (ReadFile(hInFile, buffer, BUF_SIZE, &nIn, NULL)
        && (nIn != 0)
        && WriteFile(hOutFile, buffer, nIn, &nOut, NULL)) ;
    return;
}
```

3.7 Structured Exception Handling

Handling Exceptions in C vs C++

User Generated Exceptions

Abnormal Termination

Console Control Handlers

Handling Exceptions, C vs C++

Structured Exception Handling is usually confined to C programs. C++, C#, and other languages have very similar mechanisms, however, and these mechanisms build on the SEH facilities presented here.

Console control handlers, also described in this chapter, allow a program to detect external signals such as a Ctrl-C from the console or the user logging off or shutting down the system. These signals also provide a limited form of process-to-process signaling.

Try and Except Blocks

The first step in using SEH is to determine which code blocks to monitor and provide them with exception handlers, as described next. It is possible to monitor an entire function or to have separate exception handlers for different code blocks and functions. A code block is a good candidate for an exception handler in situations that include the following, and catching these exceptions allows you to detect bugs and avoid potentially serious problems.

- Detectable errors, including system call errors, might occur, and you need to recover from the error rather than terminate the program.
- There is a possibility of dereferencing pointers that have not been properly initialized or computed.
- There is array manipulation, and it is possible for array indices to go out of bounds. • The code performs floating-point arithmetic, and there is concern with zero divides, imprecise results, and overflows.
- The code calls a function that might generate an exception intentionally, because the function arguments are not correct, or for some other occurrence.

SEH uses “try” and “except” blocks.

```
__try {
    /* Block of monitored code*/
}
__except( filtered_expression ) {
    /* Exception handling block */
}
```

Filter Expression

The filter_expression in the __except clause is evaluated immediately after the exception occurs. The expression is usually a literal constant, a call to a filter function, or a conditional expression. In all cases, the expression should return one of three values.

1. EXCEPTION_EXECUTE_HANDLER—Windows executes the except block as shown in Figure 4-1 (also see Program 4-1).
2. EXCEPTION_CONTINUE_SEARCH—Windows ignores the exception handler and searches for an exception handler in the enclosing block, continuing until it finds a handler.
3. EXCEPTION_CONTINUE_EXECUTION—Windows immediately returns control to the point at which the exception occurred. It is not possible to continue after some exceptions, and inadvisable in most other cases, and another exception is generated immediately if the program attempts to do so.

User Generated Exceptions

You can raise an exception at any point during program execution using the RaiseException function. In this way, your program can detect an error and treat it as an exception.

```
Void RaiseException(
    DWORD dwExceptionCode,
    DWORD dwExceptionFlags,
    DWORD nNumberOfArguments,
    CONST DWORD *lpArguments)
```

dwExceptionCode is the user-defined code. Do not use bit 28, which is reserved and Windows clears. The error code is encoded in bits 27–0 (that is, all except the most significant hex digit). Set bit 29 to indicate a “customer” (not Microsoft) exception. Bits 31–30 encode the severity as follows, where the resulting lead exception code hex digit is shown with bit 29 set.

- 0—Success (lead exception code hex digit is 2).
- 1—Informational (lead exception code hex digit is 6).
- 2—Warning (lead exception code hex digit is A).
- 3—Error (lead exception code hex digit is E).

dwExceptionFlags is normally 0, but setting the value to EXCEPTION_NONCONTINUABLE indicates

that the filter expression should not generate EXCEPTION_CONTINUE_EXECUTION; doing so will cause an immediate EXCEPTION_NONCONTINUABLE_EXCEPTION exception.

Abnormal Termination

Termination for any reason other than reaching the end of the try block and falling through or performing a *leave statement is considered an abnormal termination. The effect of* leave is to transfer to the end of the __try block and fall through. Within the termination handler, use this function to determine how the try block terminated.

```
BOOL AbnormalTermination (VOID)
```

The return value will be TRUE for an abnormal termination or FALSE for a normal termination. Note: The termination would be abnormal even if, for example, a return statement were the last statement in the try block.

Try-finally form

```
__try {
    /* Code block. */
}
__finally {
    /* Termination handler (finally block). */
}
```

The structure is the same as for a try-except statement, but there is no filter expression. Termination handlers, like exception handlers, are a convenient way to close handles, release resources, restore masks, and otherwise restore the process to a known state when leaving a block.

Console Control Handlers Overview

Exception handlers can respond to a variety of asynchronous events, but they do not detect situations such as the user logging off or entering a Ctrl-C from the keyboard to stop a program. Use console control handlers to detect such events. The function SetConsoleCtrlHandler allows one or more specified functions to be executed on receipt of a Ctrl-C, Ctrl-break, or one of three other console-related signals. The GenerateConsoleCtrlEvent function also generates these signals, and the signals can be sent to other processes that are sharing the same console. The handlers are user-specified Boolean functions that take a DWORD argument identifying the signal. Multiple handlers can be associated with a signal, and handlers can be removed as well as added. Here is the function to add or delete a handler.

Console Control Handlers

```
BOOL SetConsoleCtrlHandler( PHANDLER_ROUTINE HandlerRoutine, BOOL Add)
```

The handler routine is added if the Add flag is TRUE; otherwise, it is deleted from the list of console control routines. Notice that the signal is not specified. The handler must test to see which signal was received. The handler routine returns a Boolean value and takes a single DWORD parameter that identifies the signal. The HandlerRoutine in the definition is a placeholder; the programmer specifies the name. Here are some other considerations when using console control handlers.

- If the HandlerRoutine parameter is NULL and Add is TRUE, Ctrl-C signals will be ignored.
- The ENABLE_PROCESSED_INPUT flag on SetConsoleMode (Chapter 2) will cause Ctrl-C to be treated as keyboard input rather than as a signal.
- The handler routine actually executes as an independent thread (see Chapter 7) within the process. The normal program will continue to operate, as shown in the next example.
- Raising an exception in the handler will not cause an exception in the thread that was interrupted because exceptions apply to threads, not to an entire process. If you wish to communicate with the interrupted thread, use a variable, as in the next example, or a synchronization method (Chapter 8).

3.7 Exercise

Create a simple program with try-except that handles a division by zero error. Ask user for two numbers to divide and handle for the division by zero error.

3.8 Windows Processes

Windows Process Subsystems

Process Attributes in User Space

Process Attributes in Kernel Space

Windows Application Programming Interface (API) for Processes

Process Permissions

User Account Control (UAC)

Processes

A process is usually defined as an instance of a running program and consists of two components:

A kernel object that the operating system uses to manage the process. The kernel object is also

where the system keeps statistical information about the process.

An address space that contains all the executable or dynamic-link library (DLL) module's code and data. It also contains dynamic memory allocations such as thread stacks and heap allocations.

Processes are inert. For a process to accomplish anything, it must have a thread that runs in its context; this thread is responsible for executing the code contained in the process' address space. In fact, a single process might contain several threads, all of them executing code "simultaneously" in the process' address space.

Windows Process Subsystems

Windows supports two types of applications: those based on a graphical user interface (GUI) and those based on a console user interface (CUI). A GUI-based application has a graphical front end. It can create windows, have menus, interact with the user via dialog boxes, and use all the standard "Windowsy" stuff. Almost all the accessory applications that ship with Windows (such as Notepad, Calculator, and WordPad) are GUI-based applications. Console-based applications are text-based such as CMD.exe command prompt.

When you use Microsoft Visual Studio to create an application project, the integrated environment sets up various linker switches so that the linker embeds the proper type of subsystem in the resulting executable. This linker switch is /SUBSYSTEM:CONSOLE for CUI applications and /SUBSYSTEM:WINDOWS for GUI applications. When the user runs an application, the operating system's loader looks inside the executable image's header and grabs this subsystem value.

Windows Process Subsystems

Table 4-1. Application Types and Corresponding Entry Points

Application Type	Entry Point	Startup Function Embedded in Your Executable
GUI application that wants ANSI characters and strings	`_tWinMain (WinMain)`	`WinMainCRTStartup`
GUI application that wants Unicode characters and strings	`_tWinMain (wWinMain)`	`wWinMainCRTStartup`
CUI application that wants ANSI characters and strings	`_tmain (Main)`	`mainCRTStartup`
CUI application that wants Unicode characters and strings	`_tmain (Wmain)`	`wmainCRTStartup`

Depending on subsystem and type of text you want to use such as ANSI or Unicode. This table provides what type of "main" entry-point function you need to use.

Creating a Process

You create a process with the CreateProcess function:

```
BOOL CreateProcess(
    PCTSTR pszApplicationName,
    PTSTR pszCommandLine,
    PSECURITY_ATTRIBUTES psaProcess,
    PSECURITY_ATTRIBUTES psaThread,
    BOOL bInheritHandles,
    DWORD fdwCreate,
    PVOID pvEnvironment,
    PCTSTR pszCurDir,
    PSTARTUPINFO psiStartInfo,
    PPROCESS_INFORMATION ppiProcInfo);
```

Also see CreateProcessAsUser and CreateProcessWithLogon

Creating a Process

When a thread calls CreateProcess, the system creates a process kernel object with an initial usage count of 1. This process kernel object is not the process itself but a small data structure that the operating system uses to manage the process— you can think of the process kernel object as a small data structure that consists of statistical information about the process. The system then creates a virtual address space for the new process and loads the code and data for the executable file and any required DLLs into the process' address space.

The system then creates a thread kernel object (with a usage count of 1) for the new process' primary thread. Like the process kernel object, the thread kernel object is a small data structure that the operating system uses to manage the thread. This primary thread begins by executing the application entry point set by the linker as the C/ C + + run-time startup code, which eventually calls your WinMain, wWinMain, main, or wmain function. If the system successfully creates the new process and primary thread, CreateProcess returns TRUE.

Exercise 3.8

Spawn a notepad.exe process from your program.

Also see:

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

https://docs.microsoft.com/en-us/windows/win32/procthread/creating-processes?redirectedfrom=MSDN

3.9 Threads in Windows

Attributes of a thread in user and kernel space

Thread Priorities

Thread Local Storage

Thread API Library

C Runtime (CRT) Library

When to use _beginthreadex

Thread Priorities

To understand the thread-scheduling algorithms, one must first understand the priority levels that Windows uses. As illustrated in Figure 4-6, Windows uses 32 priority levels internally, ranging from 0 to 31 (31 is the highest). These values divide up as follows:

Sixteen real-time levels (16 through 31)

Sixteen variable levels (0 through 15), out of which level 0 is reserved for the zero page thread

Thread Priorities

The Windows kernel always runs the highest-priority thread that is ready for execution. A thread is not ready if it is waiting, suspended, or blocked for some reason. Threads receive priority relative to their process priority classes. Process priority classes are set initially by CreateProcess (Chapter 6), and each has a base priority, with values including:

- IDLE_PRIORITY_CLASS, for threads that will run only when the system is idle.
- NORMAL_PRIORITY_CLASS, indicating no special scheduling requirements.
- HIGH_PRIORITY_CLASS, indicating time-critical tasks that should be executed immediately.
- REALTIME_PRIORITY_CLASS, the highest possible priority.

The two extreme classes are rarely used, and the normal class can be used normally, as the name suggests. Windows is not a real-time OS, and using REALTIME_PRIORITY_CLASS can prevent other essential threads from running.

Set and get the priority class with: BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriority) DWORD GetPriorityClass(HANDLE hProcess)

Real-time Priorities

You can raise or lower thread priorities within the dynamic range in any application. However, you must have the increase scheduling priority privilege (SeIncreaseBasePriorityPrivilege) to enter the Real-Time range. Be aware that many important Windows kernel-mode system threads run in the Real-Time priority range, so if threads spend excessive time running in this range, they might block critical system functions (such as in the memory manager, cache manager, or some device drivers).

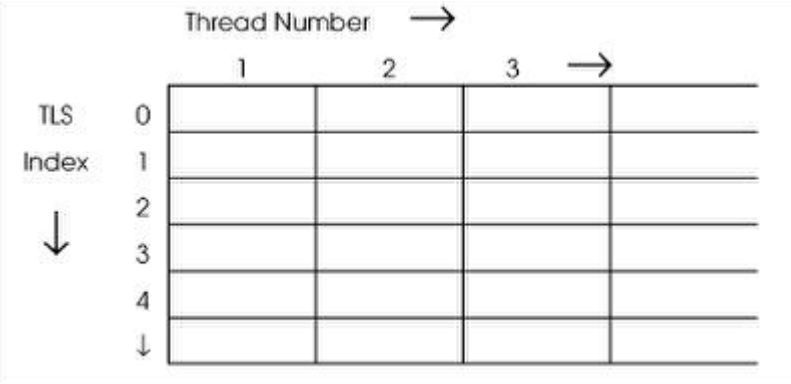
Using the standard Windows APIs, once a process has entered the Real-Time range, all its threads (even Idle ones) must run at one of the Real-Time priority levels. It is thus impossible to mix real-time and dynamic threads within the same process through standard interfaces.

Thread Local Storage

Threads may need to allocate and manage their own storage independently of and protected from other threads in the same process. One technique is to have the creating thread call CreateThread (or _beginthreadex) with lpvThreadParm pointing to a data structure that is unique for each thread. The thread can then allocate additional data structures and access them through lpvThreadParm.

Windows also provides TLS, which gives each thread its own array of pointers.Windows also provides TLS, which gives each thread its own array of pointers.

Initially, no TLS indexes (rows) are allocated, but new rows can be allocated and deallocated at any time, with at least TLS_MINIMUM_AVAILABLE (64) indexes for any process. The number of columns can change as new threads are created and old ones terminate.



Thread Local Storage

Initially, no TLS indexes (rows) are allocated, but new rows can be allocated and deallocated at any time, with at least TLS_MINIMUM_AVAILABLE (64) indexes for any process. The number of columns can change as new threads are created and old ones terminate.

The first issue is TLS index management. The primary thread is a logical place to do this, but any thread can manage thread indexes.

TlsAlloc returns the allocated index (≥ 0), with –1 (0xFFFFFFFF) if no index is available.

DWORD TlsAlloc (VOID)

An individual thread can get and set its values (void pointers) from its slot using a TLS index. LPVOID TlsGetValue(DWORD dwTlsIndex) BOOL TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue)

The programmer must ensure that the TLS index parameter is valid—that is, that it has been allocated with TlsAlloc and has not been freed.

TLS provides a convenient mechanism for storage that is global within a thread but unavailable to other threads. Normal global storage is shared by all threads. Although no thread can access another thread's TLS, any thread can call TlsFree and destroy an index for all threads, so use TlsFree carefully. BOOL TlsFree (DWORD dwIndex)

TLS is frequently used by DLLs as a replacement for global storage in a library; each thread, in effect, has its own global storage. TLS also provides a convenient way for a calling program to communicate with a DLL function, and this is the most common TLS use.

Thread API

Useful functions: CreateThread ExitThread ResumeThread SuspendThread TerminateThread GetCurrentThread GetCurrentThreadId GetThreadId OpenThread GetProcessIdOfThread

Also see:

https://docs.microsoft.com/en-us/windows/win32/procthread/using-processes-and-threads

https://docs.microsoft.com/en-us/windows/win32/procthread/process-and-thread-functions

C Runtime Library

The C runtime Library (CRT) is the part of the C Standard Library that incorporates the ISO C standard library. The Visual C libraries that implement the CRT support native code development, and both mixed native and managed code. All versions of the CRT support multi-threaded development. Most of the libraries support both static linking, to link the library directly into your code, or dynamic linking to let your code use common DLL files.

See: https://docs.microsoft.com/en-us/cpp/c-runtime-library/crt-library-features?view=msvc-160

When to use _beginthreadex

The _beginthreadex function gives you more control over how the thread is created than _beginthread does. The _endthreadex function is also more flexible. For example, with _beginthreadex, you can use security information, set the initial state of the thread (running or suspended), and get the thread identifier of the newly created thread. You can also use the thread handle that's returned by _beginthreadex with the synchronization APIs, which you cannot do with _beginthread.

It's safer to use _beginthreadex than _beginthread. If the thread that's generated by _beginthread exits quickly, the handle that's returned to the caller of _beginthread might be invalid or point to another thread. However, the handle that's returned by _beginthreadex has to be closed by the caller of _beginthreadex, so it is guaranteed to be a valid handle if _beginthreadex did not return an error.

Exercise 3.9

Spawn some threads that test if a number is prime. The main program adds these primes to it's array. Test the first 1000 numbers. Try 2 threads then more.

3.10 Synchronization Objects

Thread Synchronization Objects

Thread Synchronization Structures

Thread Synchronization APIs

Objects

Two mechanisms discussed so far allow processes and threads to synchronize with one another.

1. A thread can wait for another process to terminate by waiting on the process handle with **WaitForSingleObject** or **WaitForMultipleObjects**. A thread can wait for another thread to terminate, regardless of how the thread terminates, in the same way.
2. File locks are specifically for synchronizing file access.

Windows NT5 and NT6 provide four other objects designed for thread and process synchronization. Three of these objects—mutexes, semaphores, and events—are kernel objects that have handles. Events are also used for other purposes, such as asynchronous I/O (Chapter 14).

The fourth object, the **CRITICAL_SECTION**, is discussed first. Because of their simplicity and performance advantages, CRITICAL_SECTIONs are the preferred mechanism when they are adequate for a program's requirements.

Caution: There are risks inherent to the use of synchronization objects if they are not used properly. These risks, such as deadlocks, are described in this and subsequent chapters, along with techniques for developing reliable code. First, however, we'll show some synchronization examples in realistic situations.

Structures

Critical Code Regions are sections of code where multiple threads need to update the same memory (or variables). If those regions are not synchronized there will be incorrect computations.

An example is two threads adding amounts to a variable. If thread A adds 1 and thread B adds 2. You would expect 3 total to be added. Without synchronization you may get only 1 or 2 added.

The **volatile** storage modifier tells the compiler that this variable should always be fetched from memory and not be temporarily used as a register.

As a simple guideline, use **volatile** for any variable that is accessed by concurrent threads and is:

- Modified by at least one thread, and
- Accessed, even if read-only, by two or more threads, and correct program operation depends on the new value being visible to all threads immediately

APIs : Critical Sections

CRITICAL_SECTION Objects A critical code region, as described earlier, is a code region that only one thread can execute at a time; more than one thread executing the critical code region concurrently can result in unpredictable and incorrect results. Windows provides the CRITICAL_SECTION object as a simple “lock” mechanism for implementing and enforcing the critical code region concept.

CRITICAL_SECTION (CS) objects are initialized and deleted but do not have handles and are not shared with other processes. Declare a CS variable as a CRITICAL_SECTION. Threads enter and leave a CS, and only one thread at a time can be in a specific CS. A thread can, however, enter and leave a specific CS at multiple points in the program. To initialize and delete a CRITICAL_SECTION variable and its resources, use **InitializeCriticalSection** and **DeleteCriticalSection**, respectively.

APIs : Critical Sections

EnterCriticalSection blocks a thread if another thread is in the section, and multiple threads can wait simultaneously on the same CS. One waiting thread unblocks when another thread executes **LeaveCriticalSection**; you cannot predict which waiting thread will unblock. We say that a thread owns the CS once it returns from EnterCriticalSection, and LeaveCriticalSection relinquishes ownership. Always be certain to leave a CS; failure to do so will cause other threads to wait forever, even if the owning thread terminates. The examples use __finally blocks to leave CSs. We will often say that a CS is locked or unlocked, and entering a CS is the same as locking the CS.

APIs : Mutex

A mutex (“mutual exclusion”) object provides locking functionality beyond that of CRITICAL_SECTIONs. Because mutexes can be named and have handles, they can also be used for interprocess synchronization between threads in separate processes. For example, two processes that share memory by means of memory-mapped files can use mutexes to synchronize access to the shared memory.

Mutex objects are similar to CSs, but in addition to being process-sharable, mutexes allow time-out values and become signaled when abandoned by a terminating thread. A thread gains mutex ownership (or locks the mutex) by successfully waiting on the mutex handle (**WaitForSingleObject** or **WaitForMultipleObjects**), and it releases ownership with **ReleaseMutex**.

As a rule of thumb, use a CRITICAL_SECTION if the limitations are acceptable, and use mutexes when you have more than one process or need some other mutex capability. Also, CSs are nearly always much faster.

Windows functions are **CreateMutex**, **ReleaseMutex**, and **OpenMutex**.

Critical Section vs Mutex

As stated several times, the two lock objects, mutexes and CRITICAL_SECTIONs, are very similar and solve the same basic problems. In particular, both objects can be owned by a single thread, and other threads attempting to gain ownership will block until the object is released. Mutexes do provide greater flexibility, but with a performance penalty. In summary, these are the differences:

- Mutexes, when abandoned by a terminated thread, are signaled so that other threads are not blocked forever. This allows the application to continue execution, but an abandoned mutex almost certainly indicates a serious program bug or failure.
- Mutex waits can time out, whereas you can only poll a CS.
- Mutexes can be named and are sharable by threads in different processes.
- The thread that creates a mutex can specify immediate ownership. This is only a slight convenience, as the thread could immediately acquire the mutex with the next statement.
- CSs are almost always considerably faster than mutexes. There is more on this in Chapter 9, and Chapter 9's SRW locks provide an additional, faster option.

Exercise 3.10

Create two threads that increment a variable. Use critical sections to properly compute.

3.11 Memory Management

Windows Memory Architecture

Virtual Address partitioning on both x86 and x64

Why are the addresses 0x00000000 and Null can't be referenced.

Page translations on x86 and x64

Memory Regions

Memory Pages

Memory Protection Attributes

Memory Mapped Files

Windows Heaps

When to use individual created heaps

Memory Management API

Windows Memory Manager

The memory manager has two primary tasks:

Translating, or mapping, a process's virtual address space into physical memory so that when a thread running in the context of that process reads or writes to the virtual address space, the correct physical address is referenced. (The subset of a process's virtual address space that is physically resident is called the working set. Working sets are described in more detail in the section "Working sets" later in this chapter.)

Paging some of the contents of memory to disk when it becomes overcommitted—that is, when running threads try to use more physical memory than is currently available—and bringing the contents back into physical memory when needed.

Windows Memory Architecture

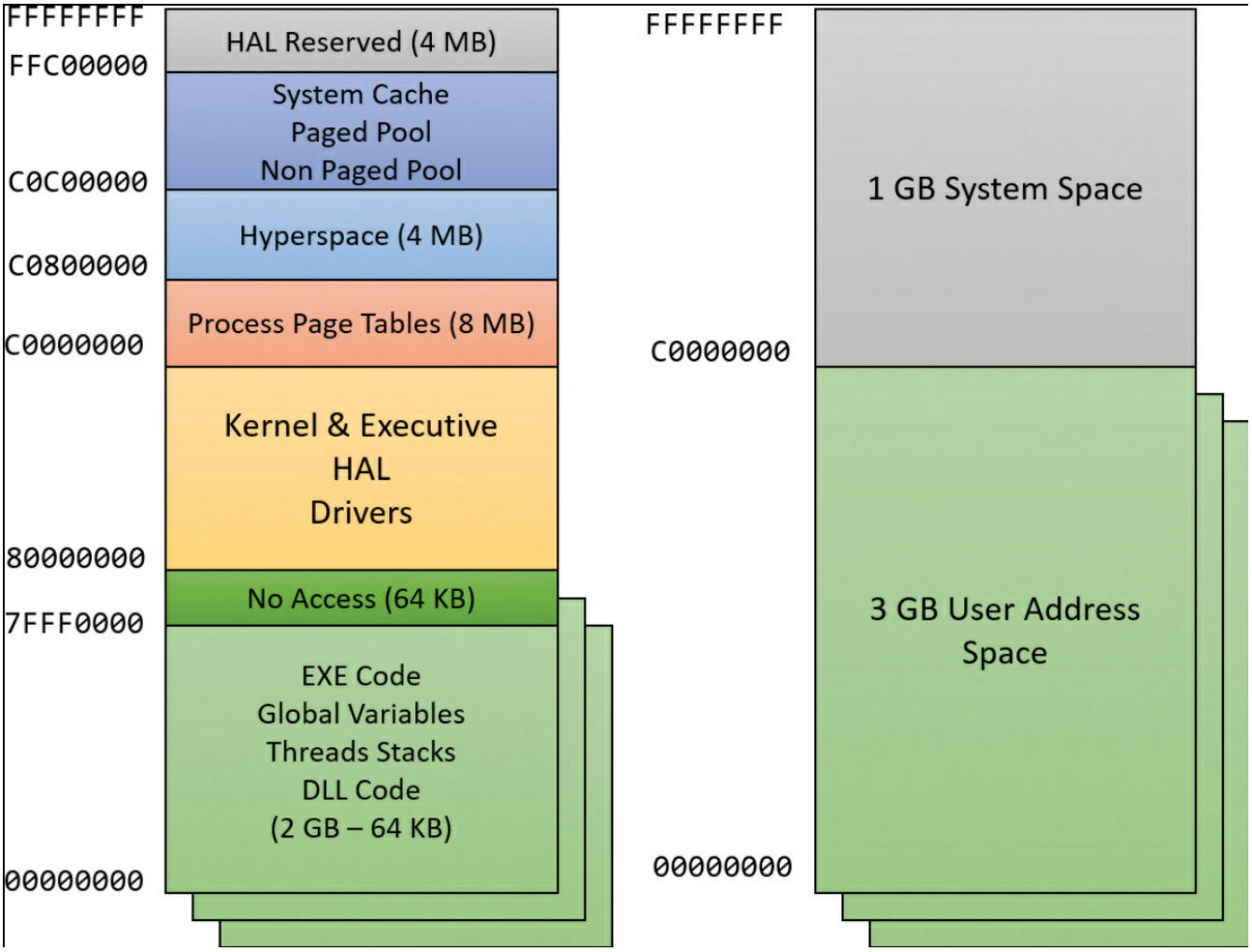
The memory manager is part of the Windows executive and therefore exists in the file Ntoskrnl.exe. It's the largest component in the executive, hinting at its importance and complexity. No parts of the memory manager exist in the HAL. The memory manager consists of the following components:

A set of executive system services for allocating, deallocating, and managing virtual memory, most of which are exposed through the Windows API or kernel-mode device driver interfaces

A translation-not-valid and access fault trap handler for resolving hardware-detected memory-management exceptions and making virtual pages resident on behalf of a process

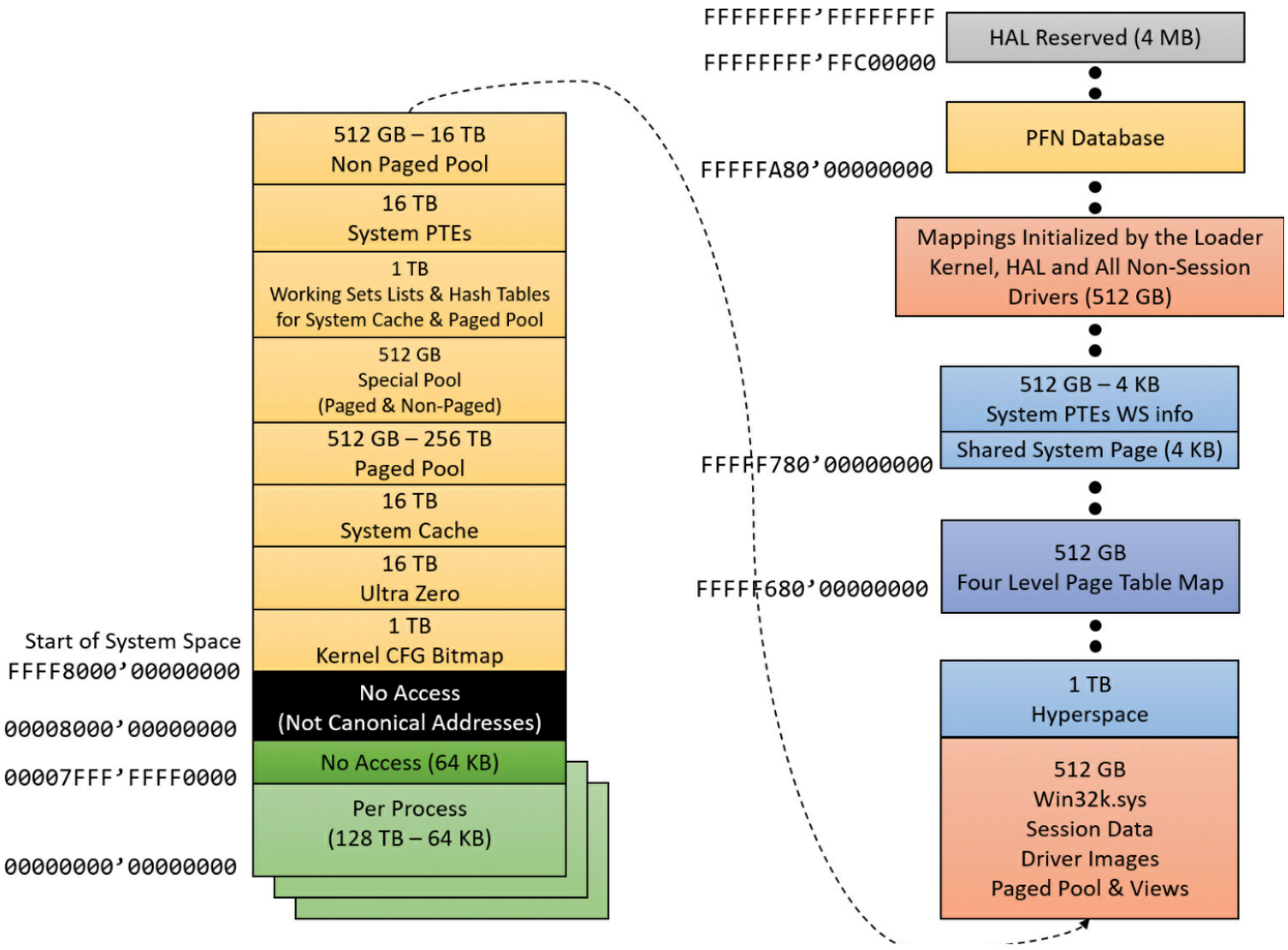
Six key top-level routines, each running in one of six different kernel-mode threads in the System process: KeBalanceSetManager, KeSwapProcessOrStack, MiModifiedPageWriter, MiMappedPageWriter, MiDereferenceSegmentThread, MiZeroPageThread.

x86 Address Space



By default, each user process on 32-bit versions of Windows has a 2 GB private address space. (The operating system takes the remaining 2 GB.) However, for x86, the system can be configured with the increaseuserva BCD boot option to permit user address spaces up to 3 GB. Two possible address space layouts are shown

x64 Address Space



The address space is divided in half, where the lower 128 TB are available as private user processes and the upper 128 TB are system space. System space is divided into several different-sized regions. The theoretical 64-bit virtual address space is 16 exabytes (EB)

Memory Management APIs

Virtual API This is the lowest-level API for general memory allocations and deallocations. It always works on page granularity. It is also the most powerful, supporting the full capabilities of the memory manager. Functions include VirtualAlloc, VirtualFree, VirtualProtect, VirtualLock, and others.

Heap API This provides functions for small allocations (typically less than a page). It uses the Virtual API internally, but adds management on top of it. Heap manager functions include HeapAlloc, HeapFree, HeapCreate, HeapReAlloc and others. The heap manager is discussed in the section “Heap manager” later in this chapter.

Local/Global APIs These are leftovers from 16-bit Windows and are now implemented using the Heap API.

Memory-mapped files These functions allow mapping files as memory and/or sharing memory between cooperating processes. Memory-mapped file functions include CreateFileMapping, OpenFileMapping, MapViewOfFile, and others.

3.12 Asynchronous IO

When should asynchronous IO be used

Overlapped IO

Completion Routines

Waitable Timers

IO Completion Ports

When Should Asynch IO Be Used?

Supporting a large number of clients without needing a thread for each client.

Thread cannot wait for I/O operation to complete.

Removing delays from the following:

- Delays caused by track and sector seek time on random access devices, such as disks
- Delays caused by the relatively slow data transfer rate between a physical device and system memory
- Delays in network data transfer using file servers, storage area networks, and so on

Overlapped I/O

The first requirement for asynchronous I/O, whether overlapped or extended, is to set the overlapped attribute of the file or other handle. Do this by specifying the **FILE_FLAG_OVERLAPPED** flag on the **CreateFile** or other call that creates the file, named pipe, or other handle. Sockets, whether created by socket or accept, have the attribute set by default. An overlapped socket can be used asynchronously in all Windows versions. Until now, overlapped structures have only been used with **LockFileEx** and as an alternative to **SetFilePointerEx** (Chapter 3), but they are essential for overlapped I/O. These structures are optional parameters on four I/O functions that can potentially block while the operation completes: ReadFile, WriteFile, TransactNamedPipe, ConnectNamedPipe

Extended I/O With Completion Routines

There is an alternative to using synchronization objects. Rather than requiring a thread to wait for a completion signal on an event or handle, the system can invoke a user-specified completion, or callback, routine when an I/O operation completes. The completion routine can then start the next I/O operation and perform any other bookkeeping.

How can the program specify the completion routine? There are no remaining ReadFile or WriteFile parameters or data structures to hold the routine's address. There is, however, a family of

extended I/O functions, identified by the Ex suffix and containing an extra parameter for the completion routine address. The read and write functions are ReadFileEx and WriteFileEx, respectively. It is also necessary to use one of five alertable wait functions:

- WaitForSingleObjectEx • WaitForMultipleObjectsEx • SleepEx • SignalObjectAndWait • MsgWaitForMultipleObjectsEx

Waitable Timers

You can always create your own timing signal using a timing thread that sets an event after waking from a Sleep call. Waitable timers are a redundant but useful way to perform tasks periodically or at specified absolute or relative times.

As the name suggests, you can wait for a waitable timer to be signaled, but you can also use a callback routine similar to the extended I/O completion routines. A waitable timer can be either a synchronization timer or a manual-reset (or notification) timer. Synchronization and manual-reset timers are comparable to auto-reset and manual-reset events; a synchronization timer becomes unsignaled after a wait completes on it, and a manual-reset timer must be reset explicitly. See **CreateWaitableTimer**, **OpenWaitableTimer**, **SetWaitableTimer**

Waitable Timer Example

```
/* chapter 14 - Timeseep. Periodic alarm. */
/* Usage: TimeBeop period (in ms). */
/* This implementation uses kernel object waitable timers".
This program uses a console control handler to catch
control C signals. */

//#include Everything.h"

static BOOL WINAPI Handler (DWORD CntrlEvent);
static VOID APIENTRY Beeper (LPVOID, DWORD, DWORD);
volatile static LONG exitFlag = 0;

HANDLE hTimer;

int _tmain (int argc, LPTSTR argv[])
{
    DWORD count = 0, period;
    LARGE_INTEGER dueTime;

    if (argc >= 2) period = _ttoi (argv [1]) * 1000;

    SetConsoleCtrlHandler(Handler, TRUE);

    dueTime.quadpart = -(LONGLONG)period * 10000;

    /* due time is negative for first time-out relative to
```

```
    current time. Period is in ms (10**-3 sec) whereas
    the due time is in 100 ns (10*%-7 sec) units to be
    consistent with a FILETIME. */

    hTimer = CreateWaitableTimer (NULL,
        /*TRUE*/,/* Not manual reset (□notification□) timer, but
        a □synchronization timer." */
        NULL);

    SetWaitableTimer (hTimer,
        &duetime /* relative time of first signal. Positive value
        would indicate an absolute time. */,
        period /* Time period in ms */,
        Beeper /* Timer function */,
        &count /* Paramoter passed to timer function */,
        FALSE);

    /* Enter the main loop */
    while (!exitFlag){
        _tprintf (_("count = Sd\n"), count);
        /* count is increased in the timer routine */
        /* Enter an alertable wait state, enabling the timer routine.
        The timer handle is a synchronization object, so you can
        also wait on it. */
        SleepEx (INFINITE, TRUE);
        /* or WaitForSingleObjectEx (hTimer, INFINITE); Beeper(...); */
    }

    _tprintf (_T("Shut Down. count = %d"), count);
    CancelWaitableTimer (hTimer);
    Closetandle (htimer) ;
    return 0;
}

/* Waitable timer callback function */
static VOID APIENTRY Beeper (LPVOID lpCount,
DWORD dwPimerLowalue, DWORD dwTimerHighValue)
{
    *(LPDWORD) lpCount = *(LPDWORD)lpcount + 1;
    _tprintf (_T("About to perform beep number: %d\n"),
        *(LPDWORD) lpCount);
    Beep (1000 /* Frequency */, 250 /* Duration (ms) */);
    return;
;

BOOL WINAPT Handler (DWORD CntriBvent)
{
    InterlockedIncrement(&exitFlag);
    _tprintf(_T("shutting Down\n"));

    return TRUE;
```

```
}
```

I/O Completion Ports

Consider a situation where you need many threads, but need to keep memory resources low. For example, thousands of clients connecting to a server.

I/O completion ports provide a solution on all Windows versions by allowing you to create a limited number of server threads in a thread pool while having a very large number of named pipe handles (or sockets), each associated with a different client. Handles are not paired with individual worker server threads; rather, a server thread can process data on any handle that has available data.

An I/O completion port, then, is a set of overlapped handles, and threads wait on the port. When a read or write on one of the handles is complete, one thread is awakened and given the data and the results of the I/O operation. The thread can then process the data and wait on the port again.

See: **CreateIoCompletionPort**, **ExistingCompletionPort**, **GetQueuedCompletionStatus**

3.12 Exercise

Create a waitable timer that periodically writes the time to a file every 30 seconds and displays the current time.

See **CreateWaitableTimer**, **OpenWaitableTimer**, **SetWaitableTimer** on MSDN documentation

3.13 Securing Windows Objects

Security Attributes

Security Descriptor

Security Descriptor Control Flags

Security Identifiers (SIDs)

Access Control Lists (ACLs)

Securing Objects APIs

Security Attributes

Nearly any object created with a **Create** system call has a security attributes parameter. Therefore, programs can secure files, processes, threads, events, semaphores, named pipes, and so on. The first step is to include a **SECURITY_ATTRIBUTES** structure in the Create call. Until now, our programs have always used a NULL pointer in Create calls or have used SECURITY_ATTRIBUTES simply to create inheritable handles (Chapter 6). In order to implement security, the important element in the

SECURITY_ATTRIBUTES structure is **lpSecurityDescriptor**, the pointer to a security descriptor, which describes the object's owner and determines which users are allowed or denied various rights.

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

Security Descriptor

A security descriptor is initialized with the function InitializeSecurityDescriptor, and it contains the following:

The owner security identifier (SID) (described in the next section, which deals with the object's owner)

The group SID

A discretionary access control list (DACL)—a list of entries explicitly granting and denying access rights. The term "ACL" without the "D" prefix will refer to DACLs in our discussion.

A system ACL (SACL), sometimes called an "audit access ACL," controls audit message generation when programs access securable objects; you need to have system administrator rights to set the SACL.

SetSecurityDescriptorOwner and SetSecurityDescriptorGroup associate SIDs with security descriptors.

Security Control Flags

Flags within the Control structure of the security descriptor, the SECURITY_DESCRIPTOR_CONTROL flags, control the meaning assigned to the security descriptor. Several of these flags are set or reset by the upcoming functions and will be mentioned as needed. **GetSecurityDescriptorControl** and **SetSecurityDescriptorControl** access these flags, but the examples do not use the flags directly.

Security Identifiers (SID)s

Windows uses SIDs to identify users and groups. The program can look up a SID from the account name, which can be a user, group, domain, and so on. The account can be on a remote system. The first step is to determine the SID from an account name.

```
BOOL LookupAccountName (
    LPCSTR lpSystemName,
    LPCSTR lpAccountName,
    PSID Sid,
    LPDWORD cbSid,
    LPTSTR ReferenceDomainName,
    LPDWORD cbReferenceDomainName,
    PSID_NAME_USE peUse)
```

Security Identifiers (SID)s

Parameters lpSystemName and lpAccountName point to the system and account names. Frequently, lpSystemName is NULL to indicate the local system.

Sid is the returned information, which is of size *cbSid. The function will fail, returning the required size, if the buffer is not large enough.

ReferencedDomainName is a string of length *cbReferencedDomainName characters. The length parameter should be initialized to the buffer size (use the usual techniques to process failures). The return value shows the domain where the name is found. The account name Administrators will return BUILTIN, whereas a user account name will return that same user name.

peUse points to a SID_NAME_USE (enumerated type) variable and can be tested for values such as SidTypeWellKnownGroup, SidTypeUser, SidTypeGroup, and so on.

Access Control Lists

Each ACL is a set (list) of access control entries (ACEs). There are two types of ACEs: one for access allowed and one for access denied.

You first initialize an ACL with InitializeAcl and then add ACEs. Each ACE contains a SID and an access mask, which specifies rights to be granted or denied to the user or group specified by the SID. FILE_GENERIC_READ and DELETE are typical file access rights.

The two functions used to add ACEs to discretionary ACLs are AddAccessAllowedAce and AddAccessDeniedAce. AddAuditAccessAce is for adding to a system ACL (SACL). Finally, remove ACEs with DeleteAce and retrieve them with GetAce.

Kernel and Private Object Security

Many objects, such as processes, threads, and mutexes, are kernel objects. To get and set kernel security descriptors, use **GetKernelObjectSecurity** and **SetKernelObjectSecurity**, which are similar to the file security functions described in this chapter. However, you need to know the access rights appropriate to an object; the next subsection shows how to find the rights.

It is also possible to associate security descriptors with private, programmer-generated objects,

such as a proprietary database. The appropriate functions are **GetPrivateObjectSecurity** and **SetPrivateObjectSecurity**. The programmer must take responsibility for enforcing access and must provide security descriptors with calls to **CreatePrivateObjectSecurity** and **DestroyPrivateObjectSecurity**.

3.14 Static and Dynamic Libraries

Difference between static and dynamic libraries

When to use each type of library

How a library is loaded dynamically

DLL injection techniques

Implicit and explicit linking

Static Libraries

The most direct way to construct a program is to gather the source code of all the functions, compile them, and link everything into a single executable image. Common functions, such as ReportError, can be put into a library to simplify the build process. This technique was used with all the sample programs presented so far, although there were only a few functions, most of them for error reporting. This monolithic, single-image model is simple, but it has several disadvantages:

The executable image may be large, consuming disk space and physical memory at run time and requiring extra effort to manage and deliver to users.

Each program update requires a rebuild of the complete program even if the changes are small or localized.

Every program in the computer that uses the functions will have a copy of the functions, possibly different versions, in its executable image. This arrangement increases disk space usage and, perhaps more important, physical memory usage when several such programs are running concurrently.

DLLs

Library functions are not linked at build time. Rather, they are linked at program load time (implicit linking) or at run time (explicit linking). As a result, the program image can be much smaller because it does not include the library functions.

DLLs can be used to create shared libraries. Multiple programs share a single library in the form of a DLL, and only a single copy is loaded into memory. All programs map the DLL code to their process address space, although each process has a distinct copy of the DLL's global variables. For example, the ReportError function was used by nearly every example program; a single DLL implementation could be shared by all the programs.

New versions or alternative implementations can be supported simply by supplying a new version of the DLL, and all programs that use the library can use the new version without modification.

The library will run in the same processes as the calling program.

When to use Static vs Dynamic

Dynamic:

- Lower memory overhead, library does not need to be loaded in ram.
- Many programs can share the same DLLs. Less storage.
- Many software components are provided as DLLs and may not be found elsewhere as source code. (proprietary format)

Static :

- The user does not need to install any type of libraries.
- Will run faster because there will not be any dynamic querying of symbols.

DLL Injection Techniques

Helpful links:

https://www.geeksforgeeks.org/static-vs-dynamic-libraries/

https://medium.com/@ozan.unal/process-injection-techniques-bc6396929740

https://www.apriorit.com/dev-blog/679-windows-dll-injection-for-api-hooks

https://medium.com/bug-bounty-hunting/dll-injection-attacks-in-a-nutshell-71bc84ac59bd

Most DLL injection can be simplified to 4 steps:

- Attach to the process.
- Allocate memory in the process.
- Copy the DLL or DLL path into that process memory with proper addresses.
- Instruct the process to execute your DLL.

Implicit Linking

Implicit or load-time linking is the easier of the two techniques. The required steps, using Microsoft Visual C++, are as follows.

1. The functions in a new DLL are collected and built as a DLL rather than, for example, a console application.
2. The build process constructs a .LIB library file, which is a stub for the actual code and is linked

into the calling program at build time, satisfying the function references. The .LIB file contains code that loads the DLL at program load time. It also contains a stub for each function, where the stub calls the DLL. This file should be placed in a common user library directory specified to the project.

3. The build process also constructs a .DLL file that contains the executable image. This file is typically placed in the same directory as the application that will use it, and the application loads the DLL during its initialization. The alternative search locations are described in the next section.
4. Take care to export the function interfaces in the DLL source.

Explicit Linking

Explicit or run-time linking requires the program to request specifically that a DLL be loaded or freed. Next, the program obtains the address of the required entry point and uses that address as the pointer in the function call. The function is not declared in the calling program; rather, you declare a variable as a pointer to a function. Therefore, there is no need for a library at link time.

The three required functions are **LoadLibrary** (or LoadLibraryEx), **GetProcAddress**, and **FreeLibrary**. Note: The function definitions show their 16-bit legacy through far pointers and different handle types. The two functions to load a library are **LoadLibrary** and **LoadLibraryEx**.

3.14 Exercise

Follow the guide in the link and attempt to create your first DLL injection. Git archive is at the end of the guide.

http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html

3.15 Management Mechanisms

Basic administrative and troubleshooting command prompt commands

Filesystem APIs

Registry architecture

Registry usage

Registry data types

WoW64 in relation to the Registry

Registry management APIs

Gaining persistence through Registry Modifications

Service applications

The Service Control Manager

Service Control Manager APIs

Component Object Model (COM)

Windows Management Instrumentation

The Common Information Model and the Managed Object Format Language

WMI implementation

Registry

If you've worked with Windows operating systems, you've probably heard about or looked at the registry. You can't talk much about Windows internals without referring to the registry because it's the system database that contains the information required to boot and configure the system, system-wide software settings that control the operation of Windows, the security database, and per-user configuration settings such as which screen saver to use.

In addition, the registry provides a window into in-memory volatile data, such as the current hardware state of the system (what device drivers are loaded, the resources they are using, and so on) as well as the Windows performance counters. The performance counters, which aren't actually in the registry, can be accessed through the registry functions.

Registry Architecture

Read the guide:

https://techcommunity.microsoft.com/t5/ask-the-performance-team/windows-architecture-registry-101/ba-p/372358

3.16 Network Programming

Windows Sockets

Berkeley Sockets vs Windows Sockets

Overlapped IO with Windows Sockets

Windows Sockets API

Windows Sockets

The Winsock API was developed as an extension of the Berkeley Sockets API into the Windows environment, and all Windows versions support Winsock. Winsock's benefits include the following.

- Porting code already written for Berkeley Sockets is straightforward.

- Windows machines easily integrate into TCP/IP networks, both IPv4 and IPv6. IPv6, among other features, allows for longer IP addresses, overcoming the 4-byte address limit of IPv4.
- Sockets can be used with Windows overlapped I/O (Chapter 14), which, among other things, allows servers to scale when there is a large number of active clients.
- Sockets can be treated as file HANDLEs for use with ReadFile, WriteFile, and, with some limitations, other Windows functions, just as UNIX allows sockets to be used as file descriptors. This capability is convenient whenever there is a need to use asynchronous I/O and I/O completion ports (Chapter 14).
- Windows provides nonportable extensions.
- Sockets can support protocols other than TCP/IP, but this chapter assumes TCP/IP. See MSDN if you use some other protocol, particularly Asynchronous Transfer Mode (ATM).

Berkeley Sockets vs Windows Sockets

Programs that use standard Berkeley Sockets calls will port to Windows Sockets, with the following important exceptions.

- You must call WSAStartup to initialize the Winsock DLL.
- You must use closesocket (which is not portable), rather than close, to close a socket.
- You must call WSACleanup to shut down the DLL.

Overlapped I/O With Windows Sockets

Overlapped I/O allows a higher number of connections without overwhelming the system.

Further details about it are from asynchronous I/O from the previous module 3.12.

3.16 Exercise

Create a simple server with the winsock API, that when connected to will say “Hello World”. Try to connect to your peers’ servers.

https://docs.microsoft.com/en-us/windows/win32/winsock/finished-server-and-client-code

3.17 Driver Programming

Windows kernel module development terminology and toolchain

IO processing

Asynchronous execution

Queues and serialization

IO Request Packet (IRP)

Filter drivers

Kernel callbacks

Hooking techniques

Patchguard

3.17 Exercise

Follow the guide and create a USB driver

https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/tutorial—write-your-first-usb-client-driver—kmdf-