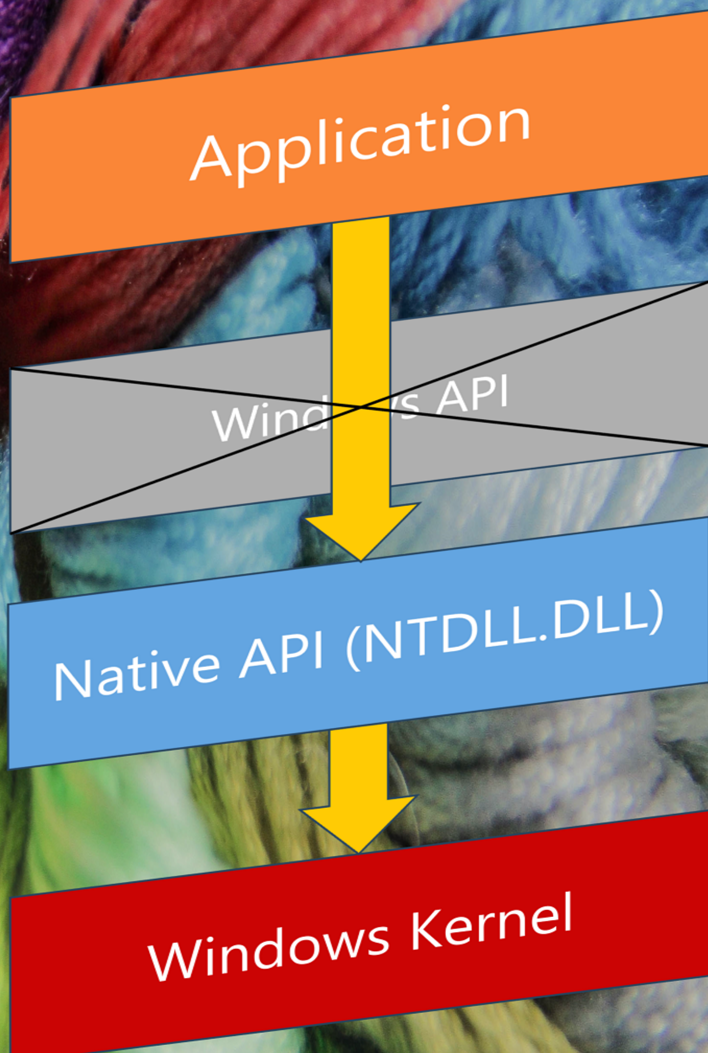
Windows Native API Programming



Pavel Yosifovich

Windows Native API Programming

Pavel Yosifovich

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

This version was published on 2023-11-17

Contents

Chapter 1: Introduction to Native API Development

In this first chapter, we'll describe the Windows system architecture as it pertains to application development, and then see where the native API enters the picture. Finally, we'll see how to add the required headers to a Visual Studio C++ application to be able to use the native API.

In this chapter:

- **Windows System Architecture**
- **What is the Native API?**
- **Getting Started**
- **Dynamic Linking to *NtDll.Dll***
- **Accessing the Native API**

1.1: Windows System Architecture

A simplified Windows system architecture is presented in figure 1-1. A user-mode process (running images such as *Notepad.exe*, *Explorer.exe*, etc.) wants to do some work. Any meaningful work (from a system's perspective) requires kernel code to be invoked. Examples include opening and working with files, allocating memory, creating threads, loading DLLs - all require kernel intervention.

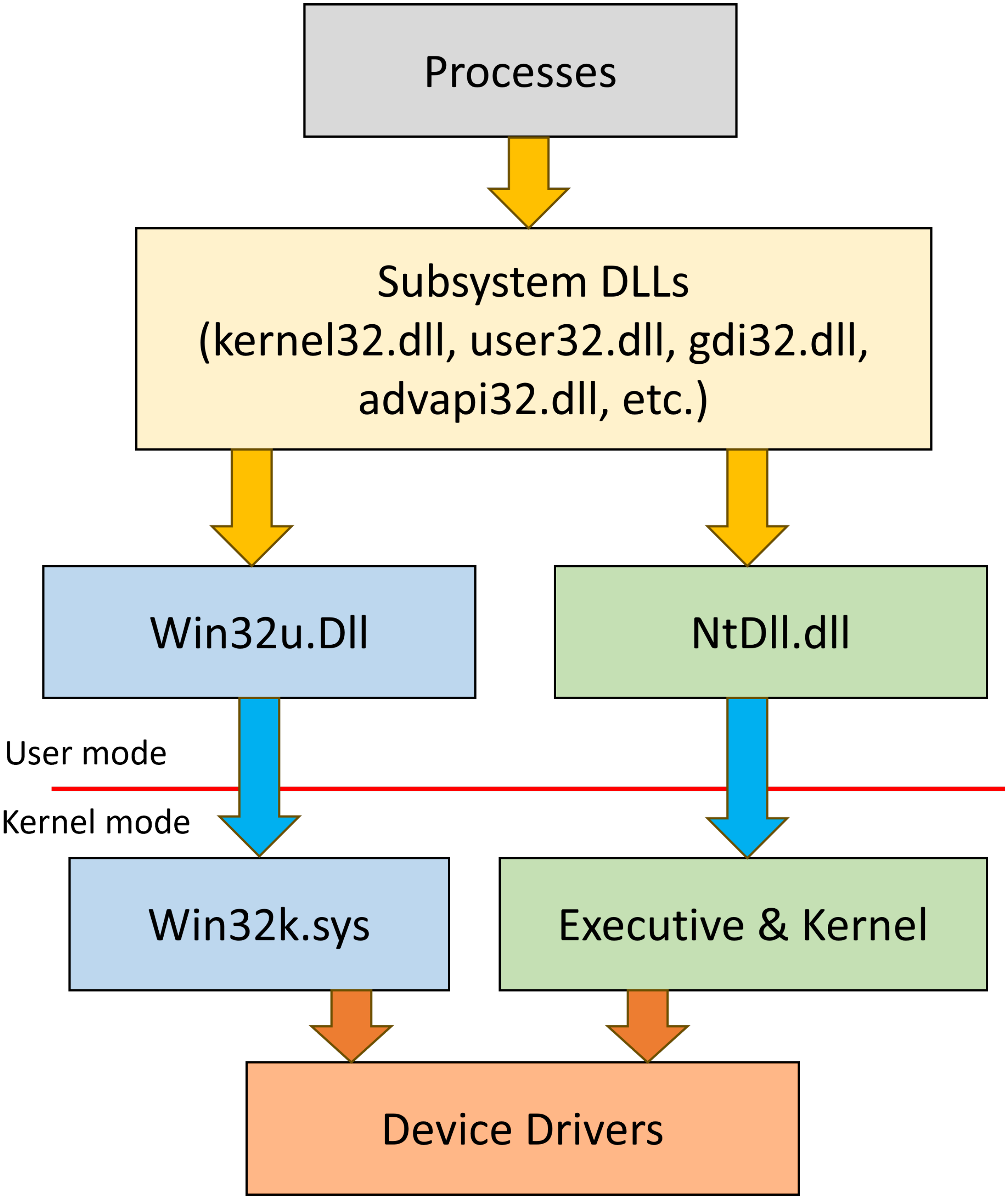


Figure 1-1: Windows System Architecture (simplified)

A typical application calls a documented Windows API, implemented in one of a set of *Subsystem DLLs*. For example, the `CreateFile` API is implemented in *kernel32.dll* (its actual implementation is in *kernelbase.dll* - *kernel32.dll* jumps there, but this detail is unimportant in practice).

> The term “Subsystem” refers to the original subsystems concept introduced in Windows NT. For a full description, consult the book *Windows Internals“, 7th edition, part 1. Suffice it to say that the only remaining subsystem (from the original OS2, POSIX, and Windows subsystems) is the Windows subsystem. This subsystem provides the APIs system developers are familiar with, such as `CreateFile`. Also note that the term “subsystem” as used here has nothing to do with the “Windows Subsystem for Linux”; refer to the same book for more information.

The `CreateFile` API will (after some parameter checks) invoke the `NtCreateFile` native API, part of *NtDll.Dll*, which is the lowest-layer DLL that is still in user-mode. This is where the native API is implemented. Its purpose is to make the transition to kernel-mode, so that the real `NtCreateFile` function will be invoked. This is accomplished (on x64 processors) by loading a value into the `EAX` register, and then invoking the `syscall` machine instruction. The value placed in the `EAX` register is the one indicating the kind of “service” required from the kernel. Here is a disassembly of the `NtCreateFile` function from *NtDll.dll* on some Windows 10 machine:

```
ntdll!NtCreateFile:
00007ffa`d138db40 mov     r10,rcx
00007ffa`d138db43 mov     eax,55h
00007ffa`d138db48 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffa`d138db50 jne     ntdll!NtCreateFile+0x15 (00007ffa`d138db55)
00007ffa`d138db52 syscall
00007ffa`d138db54 ret
00007ffa`d138db55 int     2Eh
00007ffa`d138db57 ret
```

In the above code snippet, the value `0x55` represents the `NtCreateFile` system call. Every system call has its own system call number. Behind the scenes, on the kernel side of things, this value is used as an index to a table known as *System Service Dispatch Table* (SSDT), that stores the actual system call address implementations.

> Technically, on 64-bit Windows systems, the stored values in the SSDT are not absolute addresses. Rather, a 32-bit offset from the beginning of the SSDT is stored (in the 28 most significant bits). The lower 4 bits store the number of arguments passed on the stack to the system call, so that the *System Service Dispatcher* (the common function that invokes system calls) knows how many arguments it needs to clean from the stack after the call returns.

The `jne` branch instruction just before the `syscall` is normally not taken, so that `syscall` is invoked. For compatibility reasons (at least), using `int 0x2E` still works if the tested bit happens to be clear, or is invoked directly.

All system call invocations inside *NtDll.Dll* look exactly the same, except for the number stored in `EAX`. It’s important to note that the “real” system call in the kernel has exactly the same prototype as it does in user-mode (*NtDll.Dll*). *NtDll.Dll* is used as a “trampoline” of sorts to make the quantum jump to the kernel; on the kernel side, the real implementation is invoked.

Once the system call executes within the kernel, which could mean contacting a device driver, or not, depending on the system call, the call returns back to user-mode. Note that it’s the same thread that makes the call - it starts in user-mode, transitions to the kernel when `syscall` is executed, and finally transitions back to user-mode when the opposite instruction (`sysret`) is executed.

As shown in figure 1-1, there is yet another *NtDll*-like DLL - *Win32u.Dll*. This one serves the same purpose as *NtDll.Dll* for USER and GDI (*Graphics Device Interface*) APIs. A function such as `CreateWindowEx` implemented in *User32.dll* invokes `NtUserCreateWindowEx` implemented in *Win32u.dll*. Here is the system call invocation for that:

```
win32u!NtUserCreateWindowEx:
00007ffa`cefd1eb0 mov     r10,rcx
00007ffa`cefd1eb3 mov     eax,1074h
00007ffa`cefd1eb8 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffa`cefd1ec0 jne     win32u!NtUserCreateWindowEx+0x15 (00007ffa`cefd1ec5)
00007ffa`cefd1ec2 syscall
00007ffa`cefd1ec4 ret
00007ffa`cefd1ec5 int     2Eh
00007ffa`cefd1ec7 ret
```

Not surprisingly, perhaps, the code is identical to the `NtCreateFile` code except for the system service number stashed in `EAX` - `0x1074` for `NtUserCreateWindowEx`. USER and GDI system call numbers start with `0x1000` (bit 12 set); this is intentional. On the kernel side, *Win32k.sys* is the target of these system calls. *Win32k.sys* is referred to as the “Kernel component of the Windows subsystem”, essentially implementing the windowing of the OS and the calls to use the classic GDI.

1.2: What is the Native API?

In this book, we’ll focus on the system calls provided by *NtDll.Dll*, referred to as the *Native API*, as these are more “interesting”, as they deal with the most fundamental pieces of Windows, such as processes, threads, memory, I/O, and more. From that perspective, the subsystem DLLs are bypassed, as can be seen in figure 1-2.

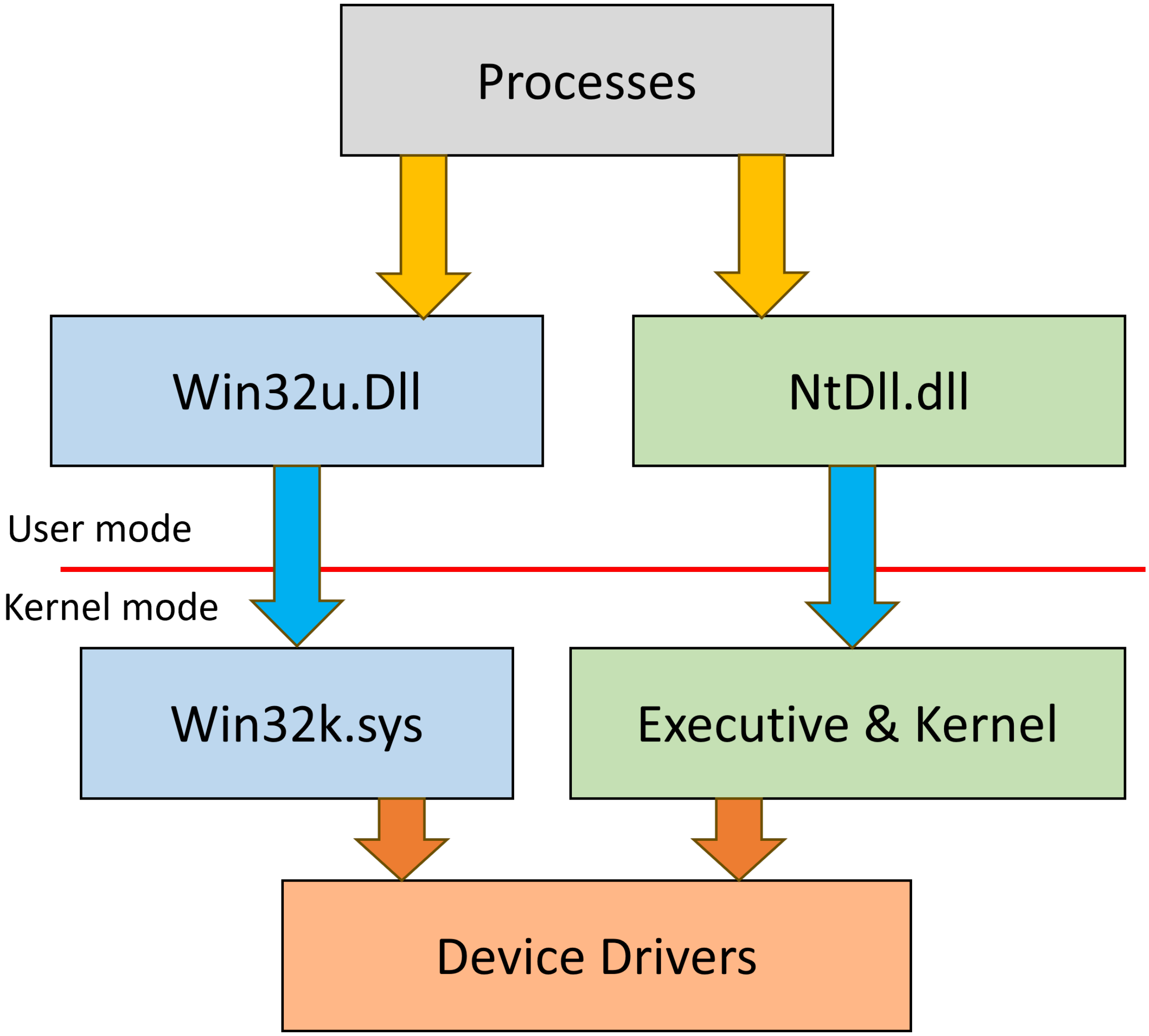


Figure 1-2: System Calls invoked directly

The Native API is mostly undocumented. Some of of it is documented indirectly in the *Windows Driver Kit* (WDK), to be used by driver developers. `NtCreateFile`, for example, happens to be documented in the WDK. Most of the native API is undocumented, however; even the prototypes are not officially provided. This begs the question: how can we use something that is not documented?

Before we tackle that question, let's consider another, perhaps more obvious question: why would we want to use the Native API in the first place? Is there something wrong with the standard, documented, Windows API?

Using the native APIs can have the following benefits:

- Performance - using the native API bypasses the standard Windows API, thus removing a software layer, speeding things up.
- Power - some capabilities are not provided by the standard Windows API, but are available with the native API.

- Dependencies - using the native API removes dependencies on subsystem DLLs, creating potentially smaller, leaner executables.
- Flexibility - in the early stages of Windows boot, native applications (those dependent on *NtDll.dll* only) can execute, while others cannot.

That said, there are potential disadvantages to using the native API:

- It's mostly undocumented, making it more difficult to use from the get go. This is where this book comes in!
- Being undocumented also means that Microsoft can make changes to the native API without warning, and no one can complain. That said, removal of capabilities from the native API or modification of existing functionality are rare. This is because some of Microsoft's own tools and applications leverage the native API.

Back to the first question: how can we get the prototypes of the native API functions? One source is the WDK, which provides a good start, although it's hardly complete. Reverse engineering can help as well, if you're so inclined. Fortunately, it's not necessary. The *winsiderss/phnt* (used to be *processhacker*) project on Github has most of the native API prototypes along with constants, enumerations, and structures - ready to go.

URL at the time of writing is https://github.com/winsiderss/phnt.

1.3: Getting Started

Let's start by creating a "hello, world" type of application to use one native API. The API in question is `NtQuerySystemInformation`, that packs a lot of functionality into its relatively simple prototype:

```
NTSTATUS NTAPI NtQuerySystemInformation(
    _In_ SYSTEM_INFORMATION_CLASS SystemInformationClass,
    _Out_writes_bytes_opt_(SystemInformationLength) PVOID SystemInformation,
    _In_ ULONG SystemInformationLength,
    _Out_opt_ PULONG ReturnLength);
```

The `NTAPI` macro is expanded to `__stdcall`, the standard calling convention used by most of the Windows API and the native API. It only means something in 32-bit processes, as in 64-bit there is just one calling convention.

We'll discuss the details of the `NtQuerySystemInformation` API in the chapter 3. For now, let's just utilize it to get some basic system information with a certain `SYSTEM_INFORMATION_CLASS` value and its associated structure. Here is the one enumeration value we'll use along with the expected structure:

```
typedef enum _SYSTEM_INFORMATION_CLASS {
    SystemBasicInformation,
} SYSTEM_INFORMATION_CLASS;

typedef struct _SYSTEM_BASIC_INFORMATION {
    ULONG Reserved;
    ULONG TimerResolution;
    ULONG PageSize;
    ULONG NumberOfPhysicalPages;
    ULONG LowestPhysicalPageNumber;
    ULONG HighestPhysicalPageNumber;
    ULONG AllocationGranularity;
    ULONG_PTR MinimumUserModeAddress;
    ULONG_PTR MaximumUserModeAddress;
    ULONG_PTR ActiveProcessorsAffinityMask;
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION, *PSYSTEM_BASIC_INFORMATION;
```

We'll add these definitions manually to a new C++ console application project created in Visual Studio. Then we'll make the call, after which we'll display some of the returned information. Here is the full `main` function:

```
int main() {
    SYSTEM_BASIC_INFORMATION sysInfo;
    NTSTATUS status = NtQuerySystemInformation(SystemBasicInformation,
        &sysInfo, sizeof(sysInfo), nullptr);
    if (status == 0) {
        //
        // call succeeded
        //
        printf("Page size: %u bytes\n", sysInfo.PageSize);
        printf("Processors: %u\n", (ULONG)sysInfo.NumberOfProcessors);
        printf("Physical pages: %u\n", sysInfo.NumberOfPhysicalPages);
        printf("Lowest  Physical page: %u\n", sysInfo.LowestPhysicalPageNumber);
        printf("Highest Physical page: %u\n", sysInfo.HighestPhysicalPageNumber);
    }
    else {
        printf("Error calling NtQuerySystemInformation (0x%X)\n", status);
    }
    return 0;
}
```

Compiling this code produces a linker error indicating that the implementation of `NtQuerySystemInformation` is nowhere to be found:

```
HelloNative.obj : error LNK2019: unresolved external symbol "long __cdecl NtQuerySys\
temInformation(enum _SYSTEM_INFORMATION_CLASS,void *,unsigned long,unsigned long *)"
 (?NtQuerySystemInformation@@YAJW4_SYSTEM_INFORMATION_CLASS@@PEAXKPEAK@Z) referenced
 in function main
```

Even though we get one error, we really have two issues here. The first is the fact that the linker has no idea where `NtQuerySystemInformation` is implemented. We know it's inside *NtDll.dll*, but the linker doesn't, so we have to tell it.

One way of doing that is by adding the import library, *ntdll.lib*, that is provided with a Visual Studio installation, to the linker's *Input* node where additional import libraries can be added (figure 1-3).

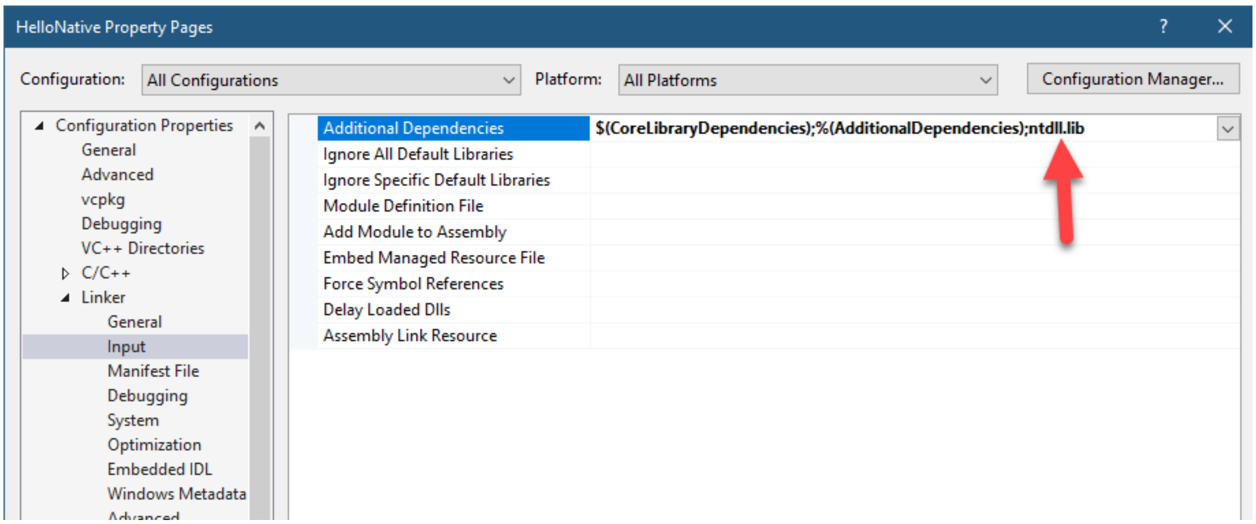


Figure 1-3: Linker's Input node in Visual Studio

Note that it's best to select *All Configurations* and *All Platforms* at the top of the dialog in figure 1-3, so we don't have to repeat this setting in every configuration and platform.

The second way to achieve the same thing is to use a `#pragma` to add the same import library in source code:

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

Even with that change (utilizing either option), we still get the same linker error. The problem is that `NtQuerySystemInformation` is defined in a C++ file, that the C++ compiler processes. The function is named based on the assumption that multiple functions with the same name can exist (function overloading) with different arguments, so the linker does not connect the function name to its "true" function in *NtDll.dll*, which is a C function. You can see the "true" name of the function is mangled by the compiler with weird symbols to capture the uniqueness of its arguments.

Fortunately, the solution is simple: decorate the function with the `extern "C"` modifier to force the compiler to treat the function as a C function (not C++):

```
extern "C" NTSTATUS NTAPI NtQuerySystemInformation(
    _In_ SYSTEM_INFORMATION_CLASS SystemInformationClass,
    _Out_writes_bytes_opt_(SystemInformationLength) PVOID SystemInformation,
    _In_ ULONG SystemInformationLength,
    _Out_opt_ PULONG ReturnLength);
```

Now everything should link successfully, and we can run the application that produces output similar to the following:

```
Page size: 4096 bytes
Processors: 16
Physical pages: 33346913
Lowest  Physical page: 1
Highest Physical page: 34142207
```

Here is the full code (found in the *HelloNative* project):

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

typedef enum _SYSTEM_INFORMATION_CLASS {
    SystemBasicInformation,
} SYSTEM_INFORMATION_CLASS;

typedef struct _SYSTEM_BASIC_INFORMATION {
    ULONG Reserved;
    ULONG TimerResolution;
    ULONG PageSize;
    ULONG NumberOfPhysicalPages;
    ULONG LowestPhysicalPageNumber;
    ULONG HighestPhysicalPageNumber;
    ULONG AllocationGranularity;
    ULONG_PTR MinimumUserModeAddress;
    ULONG_PTR MaximumUserModeAddress;
    ULONG_PTR ActiveProcessorsAffinityMask;
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION, *PSYSTEM_BASIC_INFORMATION;

extern "C" NTSTATUS NTAPI NtQuerySystemInformation(
    _In_ SYSTEM_INFORMATION_CLASS SystemInformationClass,
    _Out_writes_bytes_opt_(SystemInformationLength) PVOID SystemInformation,
    _In_ ULONG SystemInformationLength,
```

```
    _Out_opt_ PULONG ReturnLength);

#pragma comment(lib, "ntdll")

int main() {
    SYSTEM_BASIC_INFORMATION sysInfo;
    NTSTATUS status = NtQuerySystemInformation(SystemBasicInformation,
        &sysInfo, sizeof(sysInfo), nullptr);
    if (status == 0) {
        //
        // call succeeded
        //
        printf("Page size: %u bytes\n", sysInfo.PageSize);
        printf("Processors: %u\n", (ULONG)sysInfo.NumberOfProcessors);
        printf("Physical pages: %u\n", sysInfo.NumberOfPhysicalPages);
        printf("Lowest  Physical page: %u\n", sysInfo.LowestPhysicalPageNumber);
        printf("Highest Physical page: %u\n", sysInfo.HighestPhysicalPageNumber);
    }
    else {
        printf("Error calling NtQuerySystemInformation (0x%X)\n", status);
    }
    return 0;
}
```

The `#include` of *<Windows.h>* is necessary (at least for now), as it provides basic definitions like `ULONG`, `PVOID`, and other standard Windows types.

The `$(CoreLibraryDependencies)` Visual Studio variable visible in figure 1-3 includes many of the standard subsystem DLLs, but it does not include *NtDll.lib*, which is why the project failed to link. You can verify this by clicking the rightmost down arrow (figure 1-3) and selecting *Edit…* and then click the *Macros* button. You can type at the top edit box of the expanded window to filter the variable list and get to the default list of import libraries hiding under this specific variable (figure 1-4).

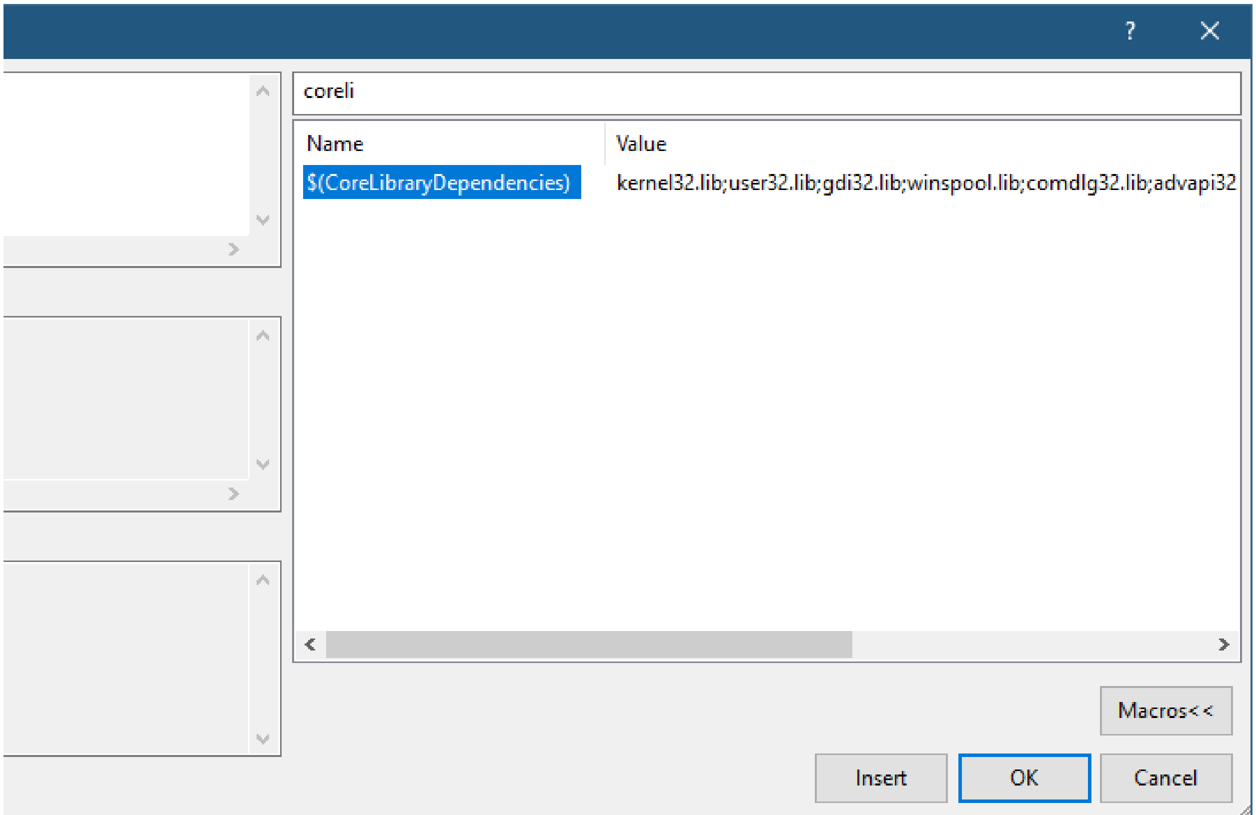


Figure 1-4: $(CoreLibraryDependencies) Visual Studio variable

1.4: Dynamic Linking to *NtDll.Dll*

The method of using an import library *NtDll.lib* works, but sometimes may not be the best way. One potential problem is using an API that is not supported on some Windows version the application targets. In that case, the application may compile and link just fine, but when it launches on a system that does not support an API that is being used - it will crash on startup.

To solve this, APIs can be bound dynamically, at runtime. If the API in question exists, we'll get a pointer to it. Otherwise, a `NULL` pointer is returned, and the application can handle the failure gracefully, without crashing.

Binding dynamically to functions is achieved with the `GetProcAddress` Windows API. Here is an example:

```
auto pNtQuerySystemInformation =
    (decltype(NtQuerySystemInformation)*)GetProcAddress(
    GetModuleHandle(L"ntdll"), "NtQuerySystemInformation");

if (pNtQuerySystemInformation) {
    SYSTEM_BASIC_INFORMATION sysInfo;
    NTSTATUS status = pNtQuerySystemInformation(SystemBasicInformation,
        &sysInfo, sizeof(sysInfo), nullptr);
//...
```

Using the `auto` and `decltype` C++11 keywords simplifies the code, as it does not require to define a function pointer. But that can be done if needed:

```
typedef NTSTATUS (NTAPI *PNtQuerySystemInformation)(
    _In_ SYSTEM_INFORMATION_CLASS SystemInformationClass,
    _Out_writes_bytes_opt_(SystemInformationLength) PVOID SystemInformation,
    _In_ ULONG SystemInformationLength,
    _Out_opt_ PULONG ReturnLength);

int main() {
    auto NtQuerySystemInformation = (PNtQuerySystemInformation)GetProcAddress(
        GetModuleHandle(L"ntdll"), "NtQuerySystemInformation");

    if (NtQuerySystemInformation) {
        SYSTEM_BASIC_INFORMATION sysInfo;
        NTSTATUS status = NtQuerySystemInformation(SystemBasicInformation,
            &sysInfo, sizeof(sysInfo), nullptr);
//...
```

In this case we can define the function pointer type with a "P" prefix, so we can use the "real" function name as the name of the variable, making the code feel perhaps slightly more "natural". Notice that `extern "C"` is no longer needed, as `GetProcAddress` always assumes an unmangled function name.

The full code for this example is in the project *HelloNative2*.

`GetModuleHandle` is used to retrieve the instance handle (just the address of) the *NtDll.Dll* DLL in the process, that is fed to `GetProcAddress`. This will always succeed because *NtDll.Dll* is mapped to every user-mode process by the kernel, and so is available at a very early stage of a process' lifetime.

Astute readers may be wondering why we're using `GetProcAddress` and `GetModuleHandle`, as one of our goals is to use native APIs instead of standard Windows APIs; we'll do that in the next chapter, because the corresponding native APIs require some more background, which is provided in the next chapter. In any case, the goal is not necessarily to remove all usage of the Windows API in all cases, but to use the native API to our advantage where it makes sense.

Ultimately, the way code binds to native APIs is up to the developer. It's also possible to combine both ways - use static binding (import library) for APIs that exist in all Windows versions being targeted, and use dynamic binding for functions that may be unavailable.

If you feel rusty as to how to bind to DLLs (statically and dynamically), look it up online, or read chapter 15 in my book "Windows 10 System Programming, Part 2".

1.5: Accessing the Native API

The richness of the native API can be viewed by looking at the exports of *NtDll.dll*. One way to see these is to use the *dumpbin* tool, provided with a Visual Studio installation:

```
C:\>dumpbin /exports c:\windows\System32\ntdll.dll
Microsoft (R) COFF/PE Dumper Version 14.37.32705.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file c:\windows\System32\ntdll.dll

File Type: DLL

  Section contains the following exports for ntdll.dll

    00000000 characteristics
    6349A4F2 time date stamp
        0.00 version
           8 ordinal base
        2435 number of functions
        2434 number of names

    ordinal hint RVA      name

          9    0 00040280 A_SHAFinal
         10    1 000410B0 A_SHAInit
         11    2 000410F0 A_SHAUpdate
         12    3 000E09C0 AlpcAdjustCompletionListConcurrencyCount
         13    4 00070780 AlpcFreeCompletionListMessage
         14    5 000E09F0 AlpcGetCompletionListLastMessageInformation
```

```
         15     6 000E0A10 AlpcGetCompletionListMessageAttributes
         16     7 000704B0 AlpcGetHeaderSize
         17     8 00070470 AlpcGetMessageAttribute
         18     9 00010A60 AlpcGetMessageFromCompletionList
         19     A 00085E00 AlpcGetOutstandingCompletionListMessageCount
...
       2439   97E 00097FF0 wcstok_s
       2440   97F 00092410 wcstol
       2441   980 000924B0 wcstombs
       2442   981 00092470 wcstoul
...
```

You can also see these with any graphical *Portable Executable* (PE) viewer, like my own *TotalPE*, by loading *NtDll.Dll* and examining the Exports data directory (figure 1-5). Notice there are 2435 exported functions on the Windows version where this screenshot was taken.

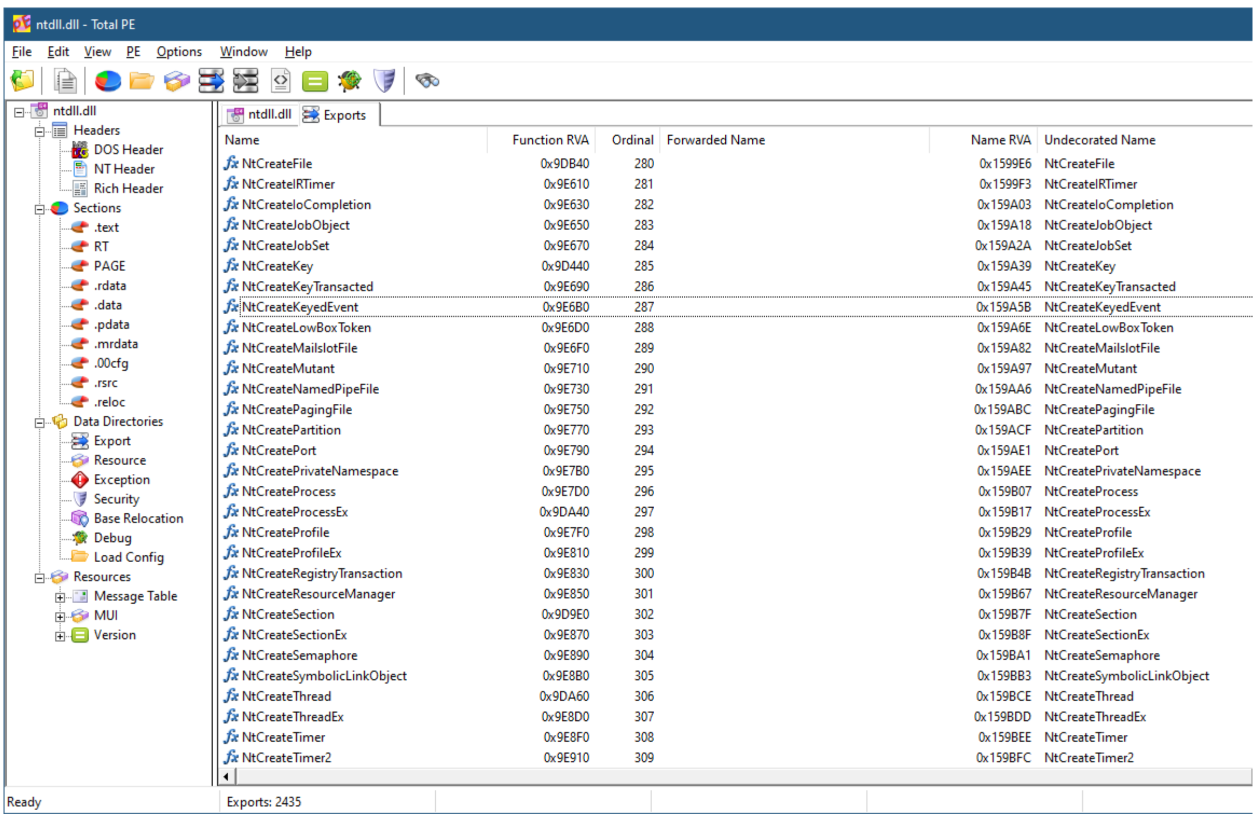


Figure 1-5: TotalPE showing NtDll.Dll exports

One way to get the prototypes of functions, structures, and enumerations, is to copy what we need from the headers provided by the winsiderss/phnt[1] project. There is one catch, however: certain definitions are common to many functions, and you would have to hunt down each and every definition and copy them as well to make sure everything compiles.

[1] https://github.com/winsiderss/phnt

An alternative approach is to clone the above repository (or add it as a submodule), and use the instructions on the *phnt* home page on usage. Basically, the following two includes should be used, without including <Windows.h>:

```
#include <phnt_windows.h>
#include <phnt.h>
```

In addition, there are a few macros you can use to let the *phnt* headers include APIs from required Windows versions. By default, only Windows 7 functions are included. Set the `PHNT_VERSION` macro to the required version selected from the following, before including the above headers:

```
#define PHNT_VERSION PHNT_WIN2K
#define PHNT_VERSION PHNT_WINXP
#define PHNT_VERSION PHNT_WS03
#define PHNT_VERSION PHNT_VISTA
#define PHNT_VERSION PHNT_WIN7
#define PHNT_VERSION PHNT_WIN8
#define PHNT_VERSION PHNT_WINBLUE
#define PHNT_VERSION PHNT_THRESHOLD
#define PHNT_VERSION PHNT_THRESHOLD2
#define PHNT_VERSION PHNT_REDSTONE
#define PHNT_VERSION PHNT_REDSTONE2
#define PHNT_VERSION PHNT_REDSTONE3
#define PHNT_VERSION PHNT_REDSTONE4
#define PHNT_VERSION PHNT_REDSTONE5
#define PHNT_VERSION PHNT_19H1
#define PHNT_VERSION PHNT_19H2
#define PHNT_VERSION PHNT_20H1
```

The names used above are the internal names used by Microsoft to represent various Windows versions. The list is likely to be extended in the future.

There is yet another way to get the *phnt* headers, by installing the *phnt* package using vcpkg[2]. *Vcpkg* is a package manager for C++. Follow the instructions on the above link to install *Vcpkg* (if you haven't done so already). Then you can install *phnt* by typing:

```
C:\vcpkg> .\vcpkg.exe install phnt:x64-windows
```

Assuming *Vcpkg* is integrated with Visual Studio (`vcpkg integrate install`), you'll be able to use the *phnt* headers without adding anything special to a project.

The following shows the code that performs the same operations as *HelloNative*, but using the *phnt* headers, assuming it has been installed with *Vcpkg*, copied to the project manually, or added as a submodule to a Git repository (*HelloNative3* project in the samples):

[2] https://github.com/microsoft/vcpkg

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

#pragma comment(lib, "ntdll")

int main() {
    SYSTEM_BASIC_INFORMATION sysInfo;
    NTSTATUS status = NtQuerySystemInformation(SystemBasicInformation,
        &sysInfo, sizeof(sysInfo), nullptr);
    if (status == STATUS_SUCCESS) {
        printf("Page size: %u bytes\n", sysInfo.PageSize);
        printf("Processors: %u\n", (ULONG)sysInfo.NumberOfProcessors);
        printf("Physical pages: %u\n", sysInfo.NumberOfPhysicalPages);
        printf("Lowest  Physical page: %u\n", sysInfo.LowestPhysicalPageNumber);
        printf("Highest Physical page: %u\n", sysInfo.HighestPhysicalPageNumber);
    }
    else {
        printf("Error calling NtQuerySystemInformation (0x%X)\n", status);
    }
    return 0;
}
```

Notice that the code can use `STATUS_SUCCESS` (0) to compare against, as it's properly defined. Also note that linking is still the responsibility of the developer. The above example uses the `#pragma` approach to add the *NtDll.lib* dependency to the project.

Install *phnt* with *Vcpkg*. Create a new C++ project, add the above code and make sure everything compiles, links, and executes successfully.

1.5.1: The <Winternl.h> Header File

There is a header provided in the Windows SDK named *<Winternl.h>*. This header contains a small subset of native APIs and definitions. It's tempting to use it, and for very simple requirements, may be good enough.

However, the definitions that exist in that file are not always complete. For example, here is how the `SYSTEM_BASIC_INFORMATION` structure is defined:

```
typedef struct _SYSTEM_BASIC_INFORMATION {
    BYTE Reserved1[24];
    PVOID Reserved2[4];
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION, *PSYSTEM_BASIC_INFORMATION;
```

This definition only provides the number of processors, and nothing else. If you try to use this header with *phnt*, many definitions will clash. Bottom line - don't use the <Winternl.h> header for any serious work with the native API.

1.6: Summary

In this chapter we looked at how system calls are invoked, provided by the native API. We added support for using native API definitions from the *phnt* project into a Visual Studio project. In the next chapter, we'll examine the foundational structures, enumerations, and constants, used with the native API, and write native applications.

Chapter 2: Native API Fundamentals

The native API uses common patterns and types throughout. This chapter will focus on these commonalities.

In this chapter:

- **Function Prefixes**
- **Errors**
- **Strings**
- **Linked Lists**
- **Object Attributes**
- **Client ID**
- **Time and Time Span**
- **Bitmaps**
- **Sample: Terminating a Process**

2.1: Function Prefixes

If you browse the exported function in *NtDll.Dll*, you'll come across several function prefixes. The most common one is `Nt` - these are system calls that are implemented as described in chapter 1.

Another prefix you'll encounter is `Zw`. This prefix is used for some functions where an `Nt` prefix exists as well. In fact, if a pair of functions have the same name except for the prefix (`Nt` vs. `Zw`) - these are the same function in practice. You can verify that by looking at their RVA (*Relative Virtual Address*) having the same value. Figure 2-1 shows *TotalPE* highlighting some of these pairs.

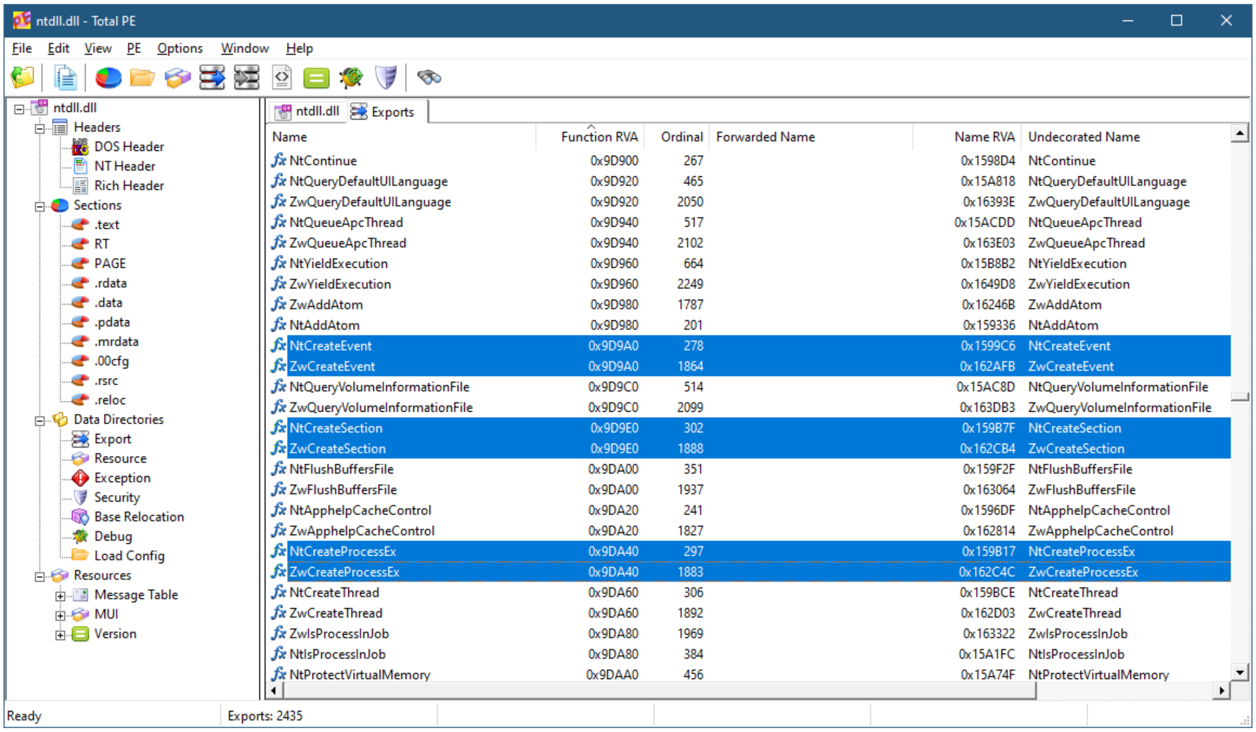


Figure 2-1: and prefixes in NtDll.Dll exports

Which prefix you use doesn't matter (if both exist), but I recommend you stick with the `Nt` prefix.

> Why are there two prefixes? It turns out that within the kernel, these are not identical functions - the `Zw` variant changes the previous thread access mode to kernel-mode before invoking the real system call (the `Nt` variant). This allows kernel code to invoke system calls directly without being considered as coming from user-mode. In user-mode, there is no distinction between these pairs of functions.

> What does `Zw` stand for? Microsoft's official documentation states that it means absolutely nothing, and this is why it was chosen. Folklore suggests it's the initials of a developer within Microsoft.

There is another common prefix within *NtDll.Dll*'s exports: `Rtl`, which stands for *Runtime Library*. There are two categories these functions are split into:

- Helper routines that do not perform any call into the kernel. Examples include working with strings, numbers, memory, data structures (bit maps, hash tables, trees). Examples: `RtlClearBits`, `RtlCompareMemory`, `RtlComputeCrc32`, `RtlCreateHashTable`.
- Convenient wrappers around `Nt` functions that make it easier to invoke certain system calls. For example, `RtlCreateUserProcessEx` does some work and then delegates to `NtCreateUserProcess`, which is the system call itself.

Finally, there are other, more specific prefixes, such as the following. Some of these are wrappers around system calls, some provide higher level functionality - it depends on the specific function.

- `Tp` - thread pool related functions.
- `Rtlp` - more runtime library functions ("p" stands for *private*) that are exported for some reason.
- `Etw` - *Event Tracing for Windows* related functions.
- `Alpc` - *Advanced (*or *Asynchronous) Local Procedure Call* related functions.
- `Dbg`, `DbgUi` - debugging related functions.
- `Csr` - *Client Server Runtime* - communication with the Windows Subsystem process (*Csrss.exe*) functions.
- `Ldr` - loader related functions.

There are a few more specialized prefixes (`MD4`, `MD5`, `Sb`, `Ship`, `Rtlx`, `Ki`, `Exp`, `Nls`, `Etwp`, `Evt`) that we may encounter later in this book.

2.2: Errors

Most native API functions return `NTSTATUS` as a direct result. This is a 32-bit signed integer, where zero indicates success (`STATUS_SUCCESS`), while negative values indicate some kind of failure.

The usual way to check for success or failure is to use the `NT_SUCCESS` macro, that returns `true` if the given status is zero (or positive).

While debugging, you may encounter an error, and would want to get a textual description of the error without searching in the headers for the particular error. Fortunately, the Visual Studio debugger supports a suffix (`,hr`) that can be appended to an error value in the *Watch* window to get a textual description. See some examples in figure 2-2.

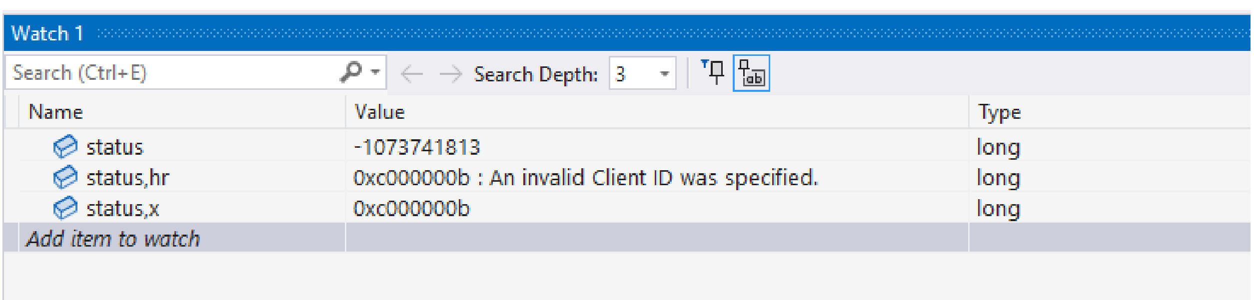


Figure 2-2: Visual Studio's Watch window

2.3: Strings

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

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

The `UNICODE_STRING` structure is a string descriptor - it describes a string, but does not necessarily own it. Here is a simplified definition of the structure:

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWCH Buffer;    // pointer to the characters
} UNICODE_STRING, *PUNICODE_STRING;
typedef const UNICODE_STRING *PCUNICODE_STRING;
```

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

Manipulating `UNICODE_STRING` structures is typically done with a set of *Rtl* functions that deal specifically with strings. Table 2-1 lists some of the common functions for string manipulation provided by the *Rtl* functions.

Table 2-1: Common ‘UNICODE_STRING‘ functions

Function	Description
`RtlInitUnicodeString`	Initializes a `UNICODE_STRING` based on an existing C-string pointer. It sets `Buffer`, then calculates the `Length` and sets `MaximumLength` to the `Length+2` to accommodate the `NULL`-terminator. Note that this function does not allocate any memory - it just initializes the internal members.
`RtlCopyUnicodeString`	Copies one `UNICODE_STRING` to another. The destination string pointer (`Buffer`) must be allocated before the copy and `MaximumLength` set appropriately.
`RtlCompareUnicodeString`	Compares two `UNICODE_STRING`s (equal, less, greater), specifying whether to do a case sensitive comparison.
`RtlCompareUnicodeStrings`	Compares two `NULL`-terminated (C-style) Unicode strings, specifying whether to do a case sensitive comparison.
`RtlEqualUnicodeString`	Compares two `UNICODE_STRING`s for equality, with case sensitivity specification.
`RtlAppendUnicodeStringToString`	Appends one `UNICODE_STRING` to another.
`RtlAppendUnicodeToString`	Appends a C-style string to a `UNICODE_STRING`.
`RtlPrefixUnicodeString`	Checks if the first string is a prefix of the second string, with optional case sensitivity.

In addition to the above functions, there are functions that work on C-string pointers. Moreover, some of the well-known string functions from the C Runtime Library are implemented within *NtDll.Dll* as well for convenience: `wcscpy_s`, `wcscat_s`, `wcslen`, `wcschr`, `strcpy`, `strcpy_s` and others. The reason for this implementation (compared to using the Visual C++ Runtime Library) will be clear in the next chapter where we discuss native applications.

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

> Don't use the non-safe functions. You can include <*dontuse.h*> to get errors for deprecated functions if you do use these in code.

Initializing a `UNICODE_STRING` is a common operation. `RtlInitUnicodeString` is a simple way to initialize it with an existing C-style Unicode string. A common case is initializing with a literal string. Here is an example:

```
UNICODE_STRING name;
RtlInitUnicodeString(&name, L"SomeString");
```

This works, but is slightly inefficient, since calculation of the string length is performed at runtime even though the string's length can be computed at compile time. To get over this slight inefficiency, the `RTL_CONSTANT_STRING` macro is provided by *phnt*, and can be used like so:

```
UNICODE_STRING name = RTL_CONSTANT_STRING(L"SomeString");
```

This, however, at the time of writing, fails in a C++ compilation because of a constness issue that the macro does not take care of. The definition of this macro in the WDK headers is more complex, and caters for constness properly, as well as for Ansi string initialization.

Here is one possible definition for this macro that forcefully casts away the constness for a Unicode string initialization:

```
#ifdef RTL_CONSTANT_STRING
    #undef RTL_CONSTANT_STRING
#endif
#define RTL_CONSTANT_STRING(s) { sizeof(s) - sizeof((s)[0]), sizeof(s), (PWSTR)s }
```

We can add a similar macro for Ansi literal strings initialization (needed in a few cases):

```
#define RTL_CONSTANT_ANSI_STRING(s) { sizeof(s)-sizeof((s)[0]), sizeof(s), (PSTR)s }
```

You can technically copy the definition of this macro from the WDK, but the above definitions do the work just fine and are simple to understand and use.

2.4: Linked Lists

The native API uses circular doubly linked lists in many of its internal data structures. For example, all modules loaded into a process are store with such linked lists in the PEB structure (see chapter 5 for more on the PEB).

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

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

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

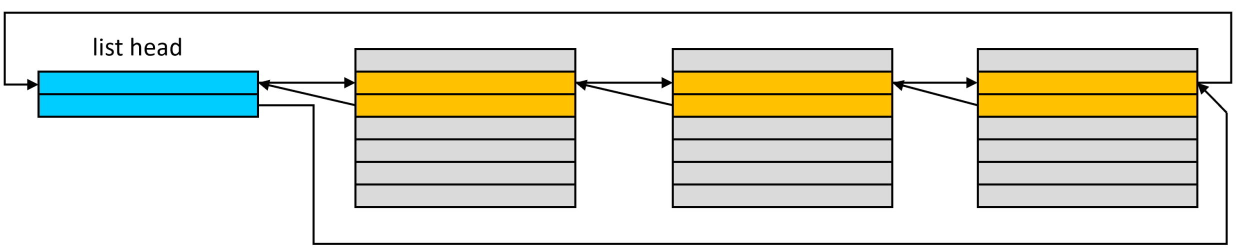


Figure 2-3: Circular linked list

One such structure is embedded inside the real structure of interest. For example, in the EPROCESS structure, the member ActiveProcessLinks is of type LIST_ENTRY, pointing to the next and previous LIST_ENTRY objects of other EPROCESS structures. The head of a list is stored separately; in the case of the process, that's PsActiveProcessHead. To get the pointer to the actual structure of interest given the address of a LIST_ENTRY can be obtained with the CONTAINING_RECORD macro.

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

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

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

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

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

Table 2-2 shows the common functions (implemented inline in the header) for working with these linked lists. All operations use constant time.

Table 2-2: Functions for working with circular linked lists

Function	Description
`InitializeListHead`	Initializes a list head to make an empty list. The forward and back pointers point to the forward pointer.
`InsertHeadList`	Insert an item to the head of the list.
`AppendTailList`	Appends one list to another.
`InsertTailList`	Insert an item to the tail of the list.
`IsListEmpty`	Check if the list is empty.
`RemoveHeadList`	Remove the item at the head of the list.
`RemoveTailList`	Remove the item at the tail of the list.
`RemoveEntryList`	Remove a specific item from the list.

The following code example shows the list of modules loaded in the current process (see chapter 5 for more details):

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

int main() {
    PPEB peb = NtCurrentPeb();
    auto& head = peb->Ldr->InLoadOrderModuleList;

    for (auto next = head.Flink; next != &head; next = next->Flink) {
        auto mod = CONTAINING_RECORD(next, LDR_DATA_TABLE_ENTRY, InLoadOrderLinks);
        printf("0x%p: %wZ\n", mod->DllBase, &mod->BaseDllName);
    }
    return 0;
}
```

NtCurrentPeb is a macro that returns the pointer to the current process' PEB. Loaded modules are stored in three separate linked lists - the example uses one of them, that stored the modules in load order (we'll examine the other lists in chapter 5). The head of the in-load-order list is stored within a member variable called Ldr (of type PPEB_LDR_DATA) in its InLoadOrderModuleList member (a LIST_ENTRY).

"%wZ" can be used to format a UNICODE_STRING in a printf-style formatting.

Since the list is circular, iterating by looking for a NULL pointer is futile - that will never happen. Instead the pointer next is initialized with the first element, moving through the loop until next becomes the same as the address of peb->Ldr->InLoadOrderModuleList - this indicates the end of iteration.

Each next pointer points to a LIST_ENTRY within the structure representing a module (LDR_DATA_TABLE_ENTRY) in the InLoadOrderLinks member. This is where the CONTAINING_RECORD macro comes in - it moves the pointer back to the beginning of the structure and performs the cast to the provided type. Using the C++ auto keyword is particularly nice here, since the type (LDR_DATA_TABLE_ENTRY*) needs no repetition.

Running this example in Debug x64 build shows something like this (*ModList* project in the source code for this chapter):

```
0x00007FF6B0B40000: ModList.exe
0x00007FF937EF0000: ntdll.dll
0x00007FF937510000: KERNEL32.DLL
0x00007FF935C50000: KERNELBASE.dll
0x00007FF8D5D50000: VCRUNTIME140D.dll
0x00007FF847280000: ucrtbased.dll
```

2.5: Object Attributes

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

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

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

- `ObjectName` is the name of the object to be created or opened, provided as a pointer to a `UNICODE_STRING`. In some cases it may be ok to set it to `NULL`, for objects that don't have names, such as processes.
- `RootDirectory` is an optional directory pointer in the object manager namespace if the name of the object is a relative one. If `ObjectName` specifies a fully-qualified name, `RootDirectory` should be set to `NULL`.
- `Attributes` allows specifying a set of flags that have effect on the operation in question. Table 2-3 shows the defined flags and their meaning.
- `SecurityDescriptor` is an optional security descriptor (`SECURITY_DESCRIPTOR`) to set on the newly created object. `NULL` indicates the new object gets a default security descriptor, based on the caller's token.
- `SecurityQualityOfService` is an optional set of attributes related to the new object's impersonation level and context tracking mode. It has no meaning for most object types. Consult the documentation for more information.

Table 2-3: Object attributes flags

Flag (**`OBJ_`**)	Description
`INHERIT` (0x02)	The returned handle should be marked as inheritable
`PERMANENT` (0x10)	The object created should be marked as permanent. Permanent objects have an additional reference count that prevents them from dying even if all handles to them are closed
`EXCLUSIVE` (0x20)	If creating an object, the object is created with exclusive access. No other handles can be opened to the object. If opening an object, exclusive access is requested, which is granted only if the object was originally created with this flag
`CASE_INSENSITIVE` (0x40)	When opening an object, perform a case insensitive search for its name. Without this flag, the name must match exactly (except for files)
`OPENIF` (0x80)	Open the object if it exists. Otherwise, fail the operation (don't create a new object)
`OPENLINK` (0x100)	If the object to open is a symbolic link object, open the symbolic link object itself, rather than following the symbolic link to its target
`KERNEL_HANDLE` (0x200)	The returned handle should be a kernel handle. This flag cannot be used from user-mode
`FORCE_ACCESS_CHECK` (0x400)	Access checks should be performed even if the object is opened in `KernelMode` access mode (not valid for user-mode)
`IGNORE_IMPERSONATED_DEVICEMAP` (0x800)	Use the process device map instead of the user's if it's impersonating (consult the documentation for more information on device maps)

Flag (**OBJ_**)	Description
DONT_REPARSE (0x1000)	Don't follow a reparse point, if encountered. Instead an error is returned (STATUS_REPARSE_POINT_ENCOUNTERED). *Reparse Points* are beyond the scope of this book.

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

2.6: Client ID

The CLIENT_ID structure is a fairly simple one, but deserves attention since it can be somewhat confusing:

```
typedef struct _CLIENT_ID {
    HANDLE UniqueProcess;
    HANDLE UniqueThread;
} CLIENT_ID, *PCLIENT_ID;
```

Its purpose is to specify a process and/or thread ID. The confusing part is that these IDs are typed as HANDLEs, rather than simple integers. The reason has to do with the fact that the kernel generates unique process and thread IDs by using a private handle table. This means the type internally is HANDLE, but it should be treated as an ID.

> Process (and thread IDs) are limited to about 26 bits, which means a 64-value is not needed to represent them.

If you have a 32-bit value that needs to be placed into a HANDLE, some casting are required for the compiler to be happy. Here is one way of doing it:

```
ULONG pid = ...;
CLIENT_ID cid;
cid.UniqueProcess = (HANDLE)(ULONG_PTR)pid;
```

The double cast is necessary for 64-bit processes - first extending the value to 64-bit and then turning it into a HANDLE (all handles are typed as void pointers).

If you're strict about "proper" c++ usage, you can use more specific casts:

```
cid.UniqueProcess = reinterpret_cast<HANDLE>(static_cast<ULONG_PTR>(pid));
```

Feel free to use such forms, I will stick with C-style casts for simplicity and for catering for C developers and beginning C++ developers.

There is yet another way to do the same thing - using simple functions/macros provided by Windows headers:

```
cid.UniqueProcess = ULongToHandle(pid);
```

You may notice there is a macro, `UlongToHandle` (notice the lowercase `l`), that just calls the inline function. Either option works.

I recommend using the inline functions/macros for these kind of conversions - it's clearer and less cluttered.

2.7: Time and Time Span

When date/time is provided by native APIs, they use a 64-bit value, typically wrapped as a `LARGE_INTEGER`:

```
typedef union _LARGE_INTEGER {
    struct {
        DWORD LowPart;
        LONG HighPart;
    };
    LONGLONG QuadPart;
} LARGE_INTEGER;
```

It's a glorified 64-bit integer, typically accessed directly with the `QuadPart` member. dates and times are conveyed in 100 nano-second units, measured from January 1, 1601 at midnight GMT. For example, the number 10000000 (10 million) represents one second after midnight on that date.

> The fact that 100 nano-second units are used does not mean that Windows is capable of such resolution, although it may be capable of that in the future. The units of measurement, however, apply.

For time spans, the same units are used, but they are relative to the time of invocation. Some APIs allow specifying either absolute time or relative time. In such a case, negative values are interpreted as relative time, while positive values are interpreted as absolute time.

An canonical example is the `NtDelayExecution` API (which is a rough equivalent of the `Sleep(Ex)` Windows API, where the time to sleep can be specified as relative or absolute. Note that `Sleep(Ex)` only support relative times in units of milliseconds.

Here is an example that sets a sleep of 100 milliseconds:

```
LARGE_INTEGER interval;
interval.QuadPart = -100 * 10000;   // 100 msec
NtDelayExecution(FALSE, &interval);
```

The following is an example that causes a sleep until March 12, 2026 at noon GMT:

```
TIME_FIELDS tf{};
tf.Year = 2026;
tf.Month = 3;
tf.Day = 12;
tf.Hour = 12;
LARGE_INTEGER interval;
RtlTimeFieldsToTime(&tf, &interval);
NtDelayExecution(FALSE, &interval);
```

This example uses the helper `TIME_FIELDS` structure, that is easier for human consumption:

```
typedef struct _TIME_FIELDS {
    CSHORT Year;            // 1601...
    CSHORT Month;           // 1..12
    CSHORT Day;             // 1..31
    CSHORT Hour;            // 0..23
    CSHORT Minute;          // 0..59
    CSHORT Second;          // 0..59
    CSHORT Milliseconds;    // 0..999
    CSHORT Weekday;         // 0..6 = Sunday..Saturday
} TIME_FIELDS, *PTIME_FIELDS;
```

To get the current time as a `LARGE_INTEGER`, use `NtQuerySystemTime`:

```
NTSTATUS NtQuerySystemTime(_Out_ PLARGE_INTEGER SystemTime);
```

Another function that provides potentially more precision is `RtlGetSystemTimePrecise`:

```
LARGE_INTEGER RtlGetSystemTimePrecise();
```

`RtlGetSystemTimePrecise` is currently not in the *phnt* headers. Add it manually if you need it.

There are other `Rtl` functions for working with times, such as `RtlCutoverTimeToSystemTime`, `RtlSystemTimeToLocalTime`, `RtlLocalTimeToSystemTime`, `RtlTimeToElapsedTimeFields`, and others.

2.8: Bitmaps

A bitmap, represented by the `RTL_BITMAP` type, provides an efficient way to store the existence or absence of something, each occupying a single bit. Functions exit to set, clear, and otherwise manipulate the bitmap. Here is `RTL_BITMAP`:

```
typedef struct _RTL_BITMAP {
    ULONG SizeOfBitMap;
    PULONG Buffer;
} RTL_BITMAP, *PRTL_BITMAP;
```

A bitmap stores its size (in bits) in the `SizeOfBitMap` member, and a pointer to the buffer, which must be 4-byte aligned where the actual bits are stored. Initializing such a bitmap is done with `RtlInitializeBitMap`:

```
VOID RtlInitializeBitMap(
    _Out_ PRTL_BITMAP BitMapHeader,
    _In_ PULONG BitMapBuffer,
    _In_ ULONG SizeOfBitMap);
```

`BitMapBuffer` is a pointer where the bits are stored, allocated by the client. `SizeOfBitMap` is the number of bits to manage. `RtlInitializeBitMap` copies the pointer to the bits and the number of bits to the provided structure. It does not initialize the bits, a job left to the caller.

The following simple APIs are used with bitmaps:

```
VOID RtlClearBit(   // clear a single bit
    _In_ PRTL_BITMAP BitMapHeader,
    _In_range_(<, BitMapHeader->SizeOfBitMap) ULONG BitNumber);
VOID RtlSetBit(     // set a single bit
    _In_ PRTL_BITMAP BitMapHeader,
    _In_range_(<, BitMapHeader->SizeOfBitMap) ULONG BitNumber);
BOOLEAN RtlTestBit(     // check the status of a bit
    _In_ PRTL_BITMAP BitMapHeader,
    _In_range_(<, BitMapHeader->SizeOfBitMap) ULONG BitNumber);
VOID RtlClearAllBits(_In_ PRTL_BITMAP BitMapHeader);
VOID RtlSetAllBits(_In_ PRTL_BITMAP BitMapHeader);
VOID RtlClearBits(  // clear a range of bits
    _In_ PRTL_BITMAP BitMapHeader,
    _In_range_(0, BitMapHeader->SizeOfBitMap - NumberToClear) ULONG StartingIndex,
    _In_range_(0, BitMapHeader->SizeOfBitMap - StartingIndex) ULONG NumberToClear);
VOID RtlSetBits(    // set a range of bits
    _In_ PRTL_BITMAP BitMapHeader,
```

```
    _In_range_(0, BitMapHeader->SizeOfBitMap - NumberToSet) ULONG StartingIndex,
    _In_range_(0, BitMapHeader->SizeOfBitMap - StartingIndex) ULONG NumberToSet);
ULONG RtlNumberOfClearBits(_In_ PRTL_BITMAP BitMapHeader);
ULONG RtlNumberOfSetBits(_In_ PRTL_BITMAP BitMapHeader);
```

These APIs are self-explanatory. The more interesting APIs relate to finding a set of one or more bits with a given value (set or clear):

```
ULONG RtlFindClearBits(
    _In_ PRTL_BITMAP BitMapHeader,
    _In_ ULONG NumberToFind,
    _In_ ULONG HintIndex);
ULONG RtlFindSetBits(
    _In_ PRTL_BITMAP BitMapHeader,
    _In_ ULONG NumberToFind,
    _In_ ULONG HintIndex);

ULONG RtlFindClearBitsAndSet(
    _In_ PRTL_BITMAP BitMapHeader,
    _In_ ULONG NumberToFind,
    _In_ ULONG HintIndex);
ULONG RtlFindSetBitsAndClear(
    _In_ PRTL_BITMAP BitMapHeader,
    _In_ ULONG NumberToFind,
    _In_ ULONG HintIndex);
```

All the above functions search for a set of bits given a hint index to begin with (`HintIndex`) and return the bit index for the beginning of the range. If no such range could be found, these functions return -1 (`0xFFFFFFFF`). The latter two functions also flip the state of the bits. Note that the hint is just that, and if the searched range cannot be found from the hinted index, the search continues from index zero.

More search APIs are available, including `RtlFindFirstRunClear`, `RtlFindNextForwardRunClear`, `RtlFindLastBackwardRunClear`, and others. They are documented in the Windows Driver Kit.

All the mentioned functions are not thread-safe. There are, however, some that are thread-safe that use Interlocked instructions for thread and CPU safety:

```
VOID RtlInterlockedClearBitRun(
    _In_ PRTL_BITMAP BitMapHeader,
    _In_range_(0, BitMapHeader->SizeOfBitMap - NumberToClear) ULONG StartingIndex,
    _In_range_(0, BitMapHeader->SizeOfBitMap - StartingIndex) ULONG NumberToClear);
VOID RtlInterlockedSetBitRun(
    _In_ PRTL_BITMAP BitMapHeader,
    _In_range_(0, BitMapHeader->SizeOfBitMap - NumberToSet) ULONG StartingIndex,
    _In_range_(0, BitMapHeader->SizeOfBitMap - StartingIndex) ULONG NumberToSet);
```

2.9: Sample: Terminating a Process

To demonstrate the usage of `CLIENT_ID` and `OBJECT_ATTRIBUTES`, we'll write a function that terminates a process given its ID. We'll compare a Windows API version against a native API version. First, the Windows API version:

```
bool KillWin32(ULONG pid) {
    auto hProcess = OpenProcess(PROCESS_TERMINATE, FALSE, pid);
    if (!hProcess)
        return false;

    auto success = TerminateProcess(hProcess, 1);
    CloseHandle(hProcess);
    return success;
}
```

`OpenProcess` is called to obtain a handle powerful enough to terminate a process (`PROCESS_TERMINATE` access mask). If successful, it calls `TerminateProcess` and finally closes the handle.

The native API version uses the system calls used behind the scenes in the example above - `NtOpenProcess` to open a process, and `NtTerminateProcess` to terminate a process. `NtOpenProcess` has the following prototype (I'll drop the `NTAPI` macro to simplify):

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

The function returns the handle to the process in the first parameter (if the call succeeds). `DesiredAccess` is the access mask requested (should be `PROCESS_TERMINATE` in our case). We must provide an `OBJECT_-ATTRIBUTES` structure, which may seem a bit odd.

Processes have no names - they have IDs. The ID is clearly provided by the last parameter. Why is `OBJECT_ATTRIBUTES` needed? The attributes flags may make a difference, and that's why its needed; the name can be set to `NULL`.

Given the above information, we can implement a native version of terminating a process:

```
NTSTATUS KillNative(ULONG pid) {
    OBJECT_ATTRIBUTES procAttr = RTL_CONSTANT_OBJECT_ATTRIBUTES(nullptr, 0);
    CLIENT_ID cid{};    // zero-out structure
    cid.UniqueProcess = ULongToHandle(pid);
    HANDLE hProcess;
    auto status = NtOpenProcess(&hProcess, PROCESS_TERMINATE, &procAttr, &cid);
    if (!NT_SUCCESS(status))
        return status;

    status = NtTerminateProcess(hProcess, 1);
    NtClose(hProcess);
    return status;
}
```

Notice that it's important to zero out the `CLIENT_ID`. Otherwise, a garbage thread ID is provided to the API, which causes it to fail, even though it's not interested in the thread ID at all. Here is the rest of the code:

```
#include <phnt_windows.h>
#include <phnt.h>
#include <stdio.h>
#include <stdlib.h>

#pragma comment(lib, "ntdll")

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

    auto pid = strtoul(argv[1], nullptr, 0);

    auto status = KillNative(pid);
    if (NT_SUCCESS(status))
        printf("Success!\n");
    else
        printf("Error: 0x%X\n", status);
```

```
    return 0;
}
```

Notice the use of the `strtoul` C function (defined in <*stdlib.h*>), that allows receiving numeric values in hexadecimal if these are prefixed with `0x`. The last parameter to that function is a radix (base), where zero indicates the function should figure it out on its own.

Similar functions exist for other integer sizes and types: `strtol`, `strtoll`, etc.

2.10: Summary

This chapter focused on common types and patterns of the native API. In the next chapter, we'll examine what native applications are, and how (and why) to write them.

Chapter 3: Native Applications

The sample applications we looked at in the previous chapters were "standard" Windows applications, that happened to use the native API. They had a dependency on *NtDll.dll*, but not just *NtDll.Dll*. An executable that depends on *NtDll.dll* only is a *Native Application*.

In this chapter:

- **Native vs. Standard Applications**
- **Building Native Applications**
- **The Main Function**
- **Simple Native Application**
- **Launching Native Applications**
- **Debugging Native Applications**

3.1: Native vs. Standard Applications

Looking at the header of a normal *Portable Executable* (PE), such as *Explorer.exe*, *Notepad.exe*, and the samples from the previous chapters show that they belong to the *Windows Subsystem*. For console applications, the value is 3 (figure 3-1), and for GUI applications that value is 2 (figure 3-2).

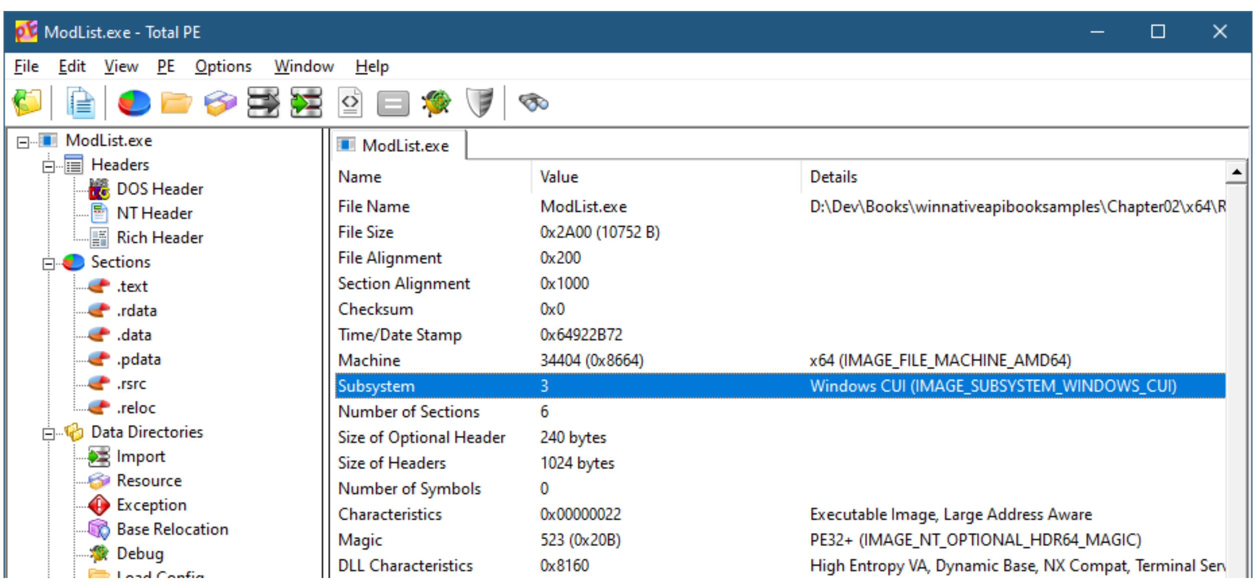


Figure 3-1: Console application subsystem value

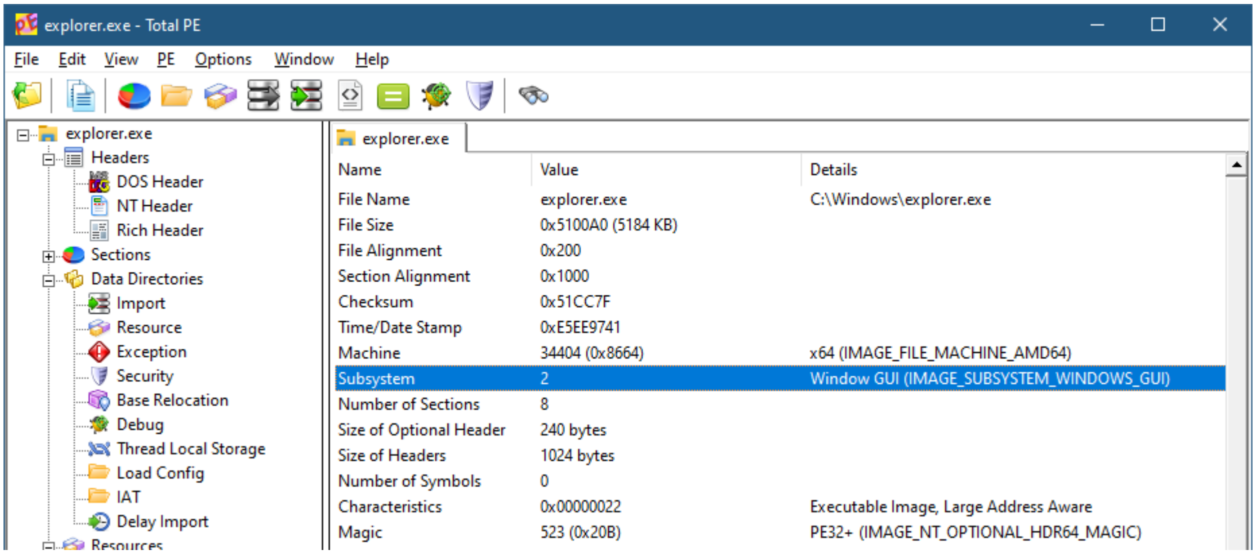


Figure 3-2: GUI application subsystem value

Although the two numbers are different, they represent the same thing: the Windows Subsystem. These are equivalent because a console application can create a graphical user interface, and a GUI application can create a console (`AllocConsole` Windows API).

If you examine the dependencies of such executables, you may or may not see *NtDll.dll* in the *Imports* PE directory, but that dependency always exists (albeit indirectly). Looking at the *Imports* directory for *Kill.exe* from chapter 2 (*Release* build) shows it has dependencies on *Kernel32.dll*, *NtDll.Dll*, and *VCRuntime140.Dll*. Selecting a specific DLL shows the imported functions at the bottom (figure 3-3).

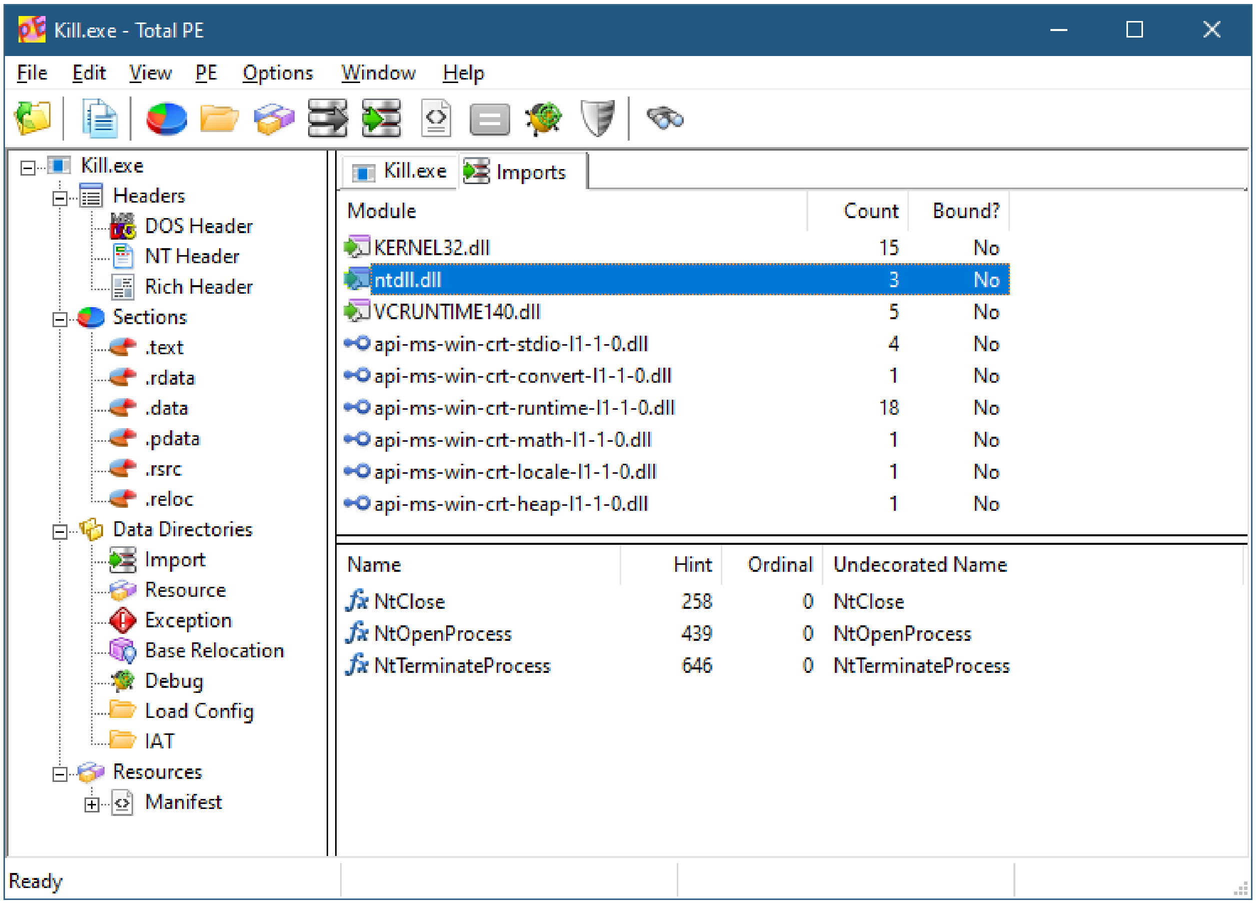


Figure 3-3: Imports of Kill.exe

A native application is one that has a sole dependency on *NtDll.Dll* and nothing else. A canonical example is *Smss.exe* (the session manager), which is the first user-mode process created in the system. Figure 3-4 shows it's part of the *Native* subsystem (value of 1), which can be described as no subsystem at all. Looking at its imports, *NtDll.Dll* is the only dependency (figure 3-5).

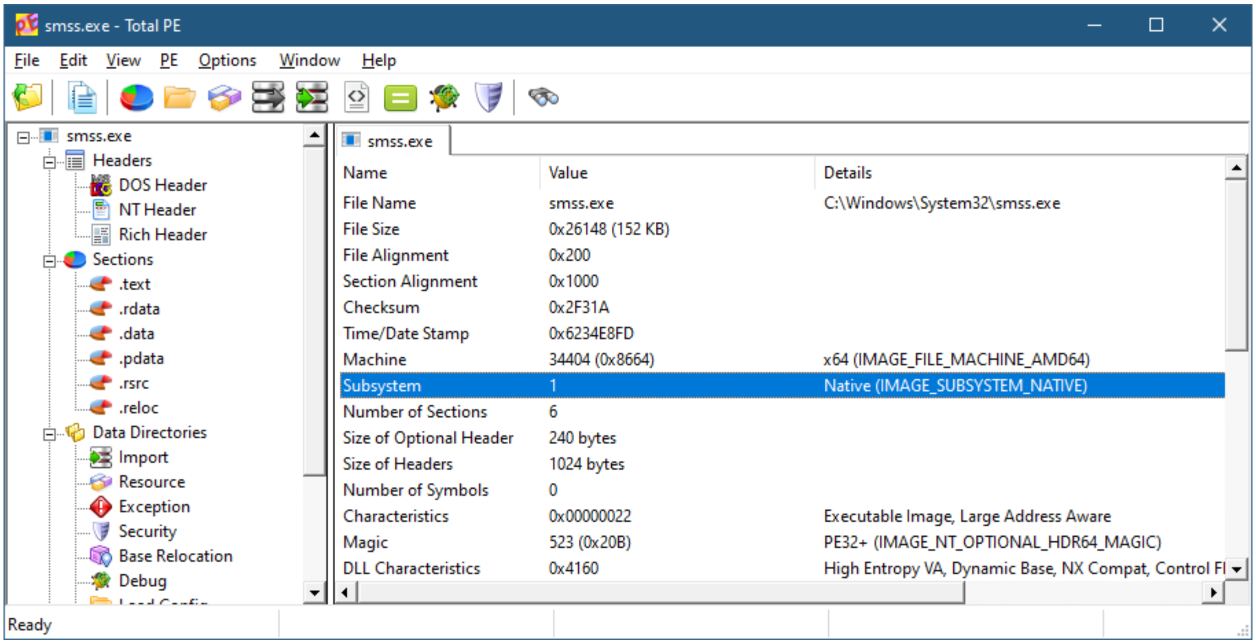


Figure 3-4: Subsystem of Smss.exe

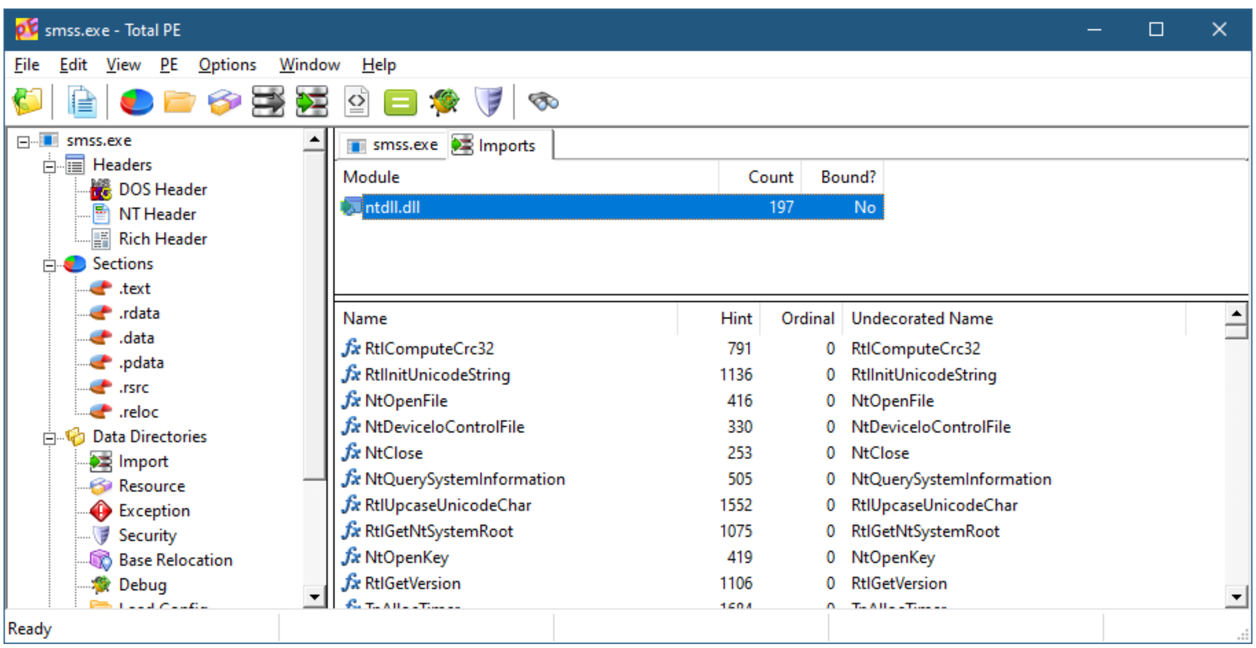


Figure 3-5: Imports of Smss.exe

Another example of a native application is *AutoChk.exe*, found in the *System32* directory. This is a native application that does some work when Windows does not shutdown cleanly, and there is a chance that hard disk integrity has been affected. It offers to check the disk(s) when Windows boots.

There is, however, another version of *AutoChk.exe* called *ChkDsk.exe*, which is a normal console Windows application, that has dependencies on the *msvcrt.dll* and other DLLs (figure 3-6). Why have two versions that do essentially the same thing?

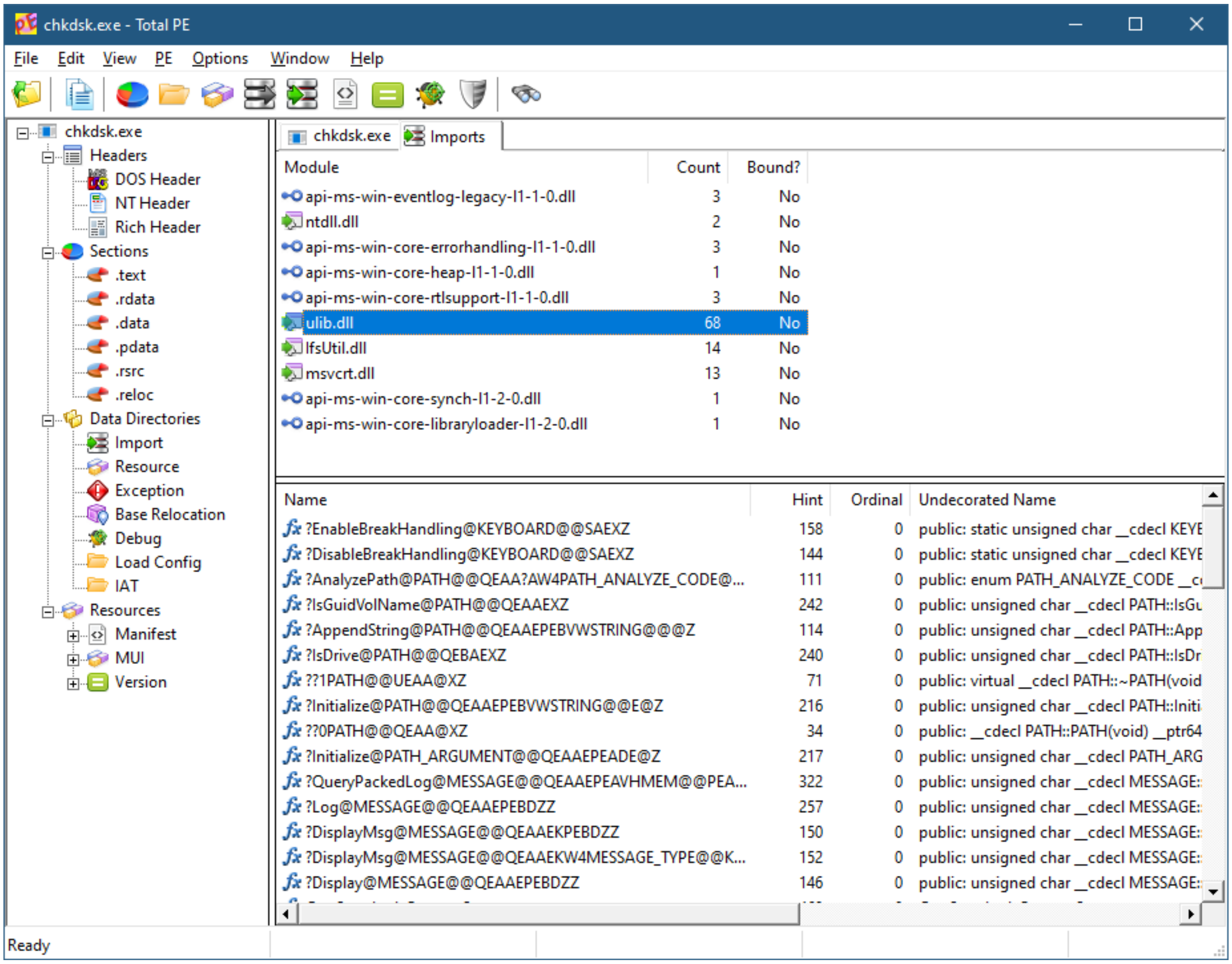


Figure 3-6: Imports of Chkdsk.exe

First, native applications cannot be run directly by normal means such as double-clicking in *Explorer* or running from a command window. Here is what you get if you try to run *AuthoChk.exe*:

```
The C:\Windows\system32\autochk.exe application cannot be run in Win32 mode.
```

In other words, the `CreateProcess` Windows API is unable to launch native applications. Second, only native applications can run early in Windows boot process. In fact, one of *Smss.exe*'s jobs is to run native applications based on a multi-string value named *BootExecute* in the Registry key *HKLM\System\CurrentControlSet\Control\Session Manager*. Figure 3-7 shows what that value looks like in *RegEdit*.

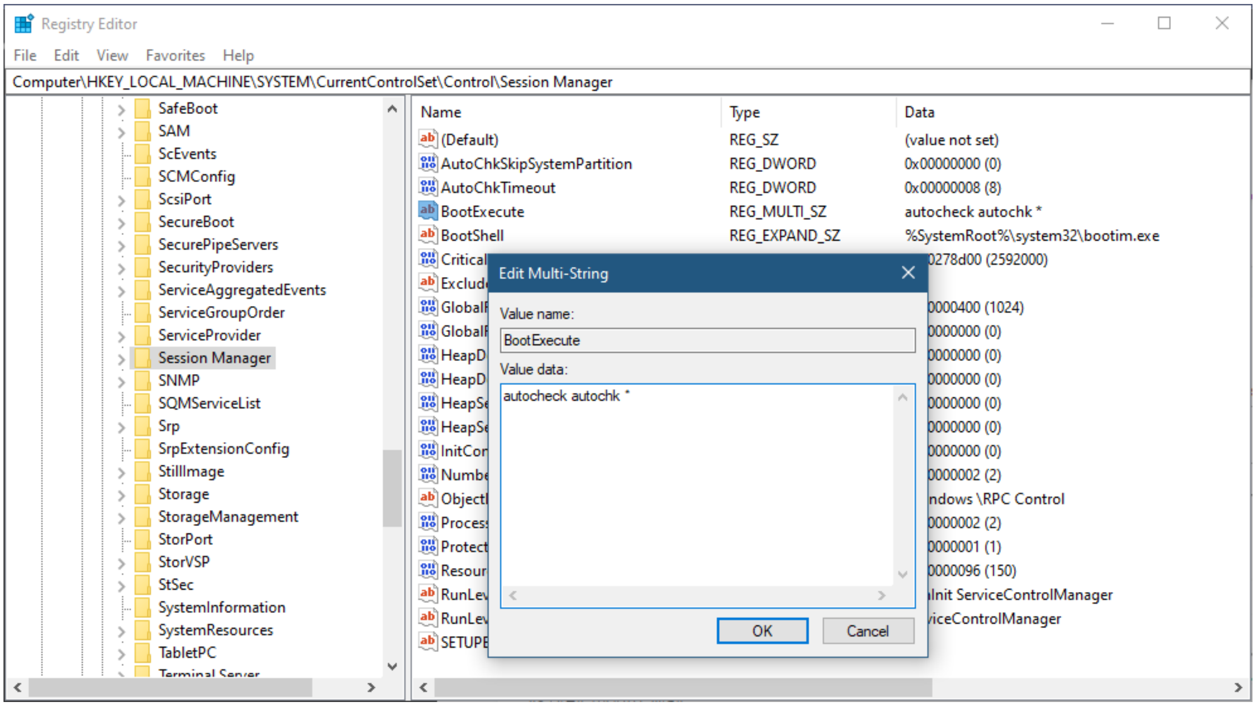


Figure 3-7: The BootExecute value

Each line in the multi-string value consists of an executable name and command line arguments. Autochk gets special treatment, and has a friendly name to identify it as such. As can be seen in figure 3-7, “autocheck” is the friendly name, and “autochk *” is the executable name and command line arguments. The executable must be in the *System32* directory - full paths are not supported. This is intentional, because writing to *System32* requires administrator privileges by default. The Registry key itself allows write access to administrator accounts only, to further limit arbitrary users from making changes to this key.

You can also see the *BootExecute* information in a somewhat friendlier manner with the *Sysinternals Autoruns* utility (figure 3-8).

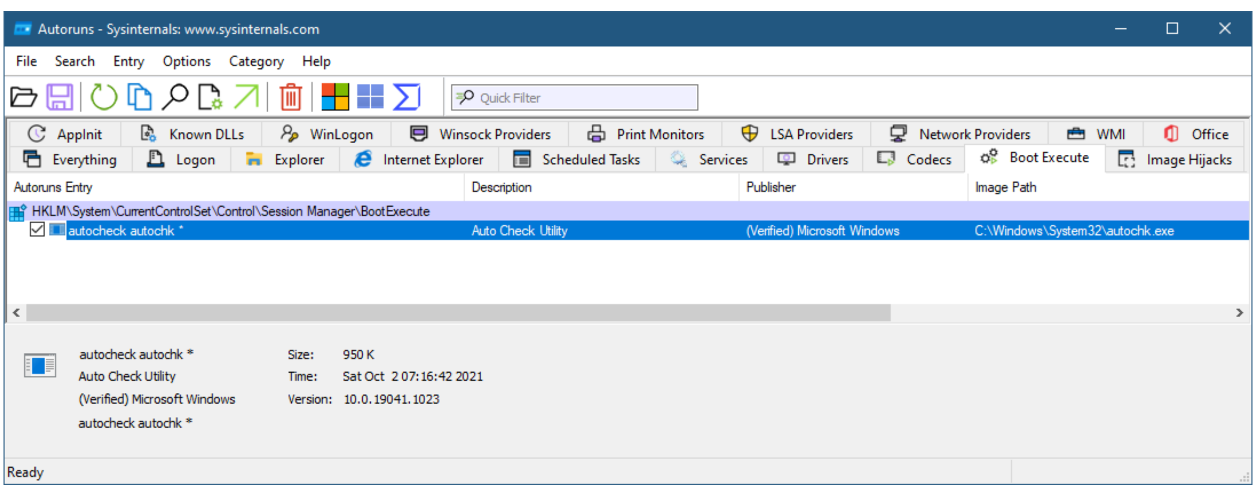


Figure 3-8: The BootExecute value in Autoruns

Any string added to *BootExecute* will be run by *Smss.exe* at system startup. Such applications, however, must be native, as the Windows subsystem process (*Csrss.exe*) has not been loaded yet.

> The Session Manager is responsible for loading *Csrss.exe* for each session. At the time *BootExecute* executables run, the Session Manager is the only user-mode process alive.

3.2: Building Native Applications

Building native applications requires removing all the "standard" dependencies, such as *Kernel32.dll*, the Visual C++ runtime - everything. Only *NtDll.dll* can remain. Here are the steps required after creating a standard console C++ application:

- Open the project's properties, select "All Platforms" and "All Configurations" in the top comboboxes, so that you don't have to repeat these procedures for every combination of platform/configuration separately.
- Navigate to the *Linker / Input* node, and set "Ignore all Default Libraries" to Yes. While you're at it, add *NtDll.lib* as a dependency (figure 3-9).

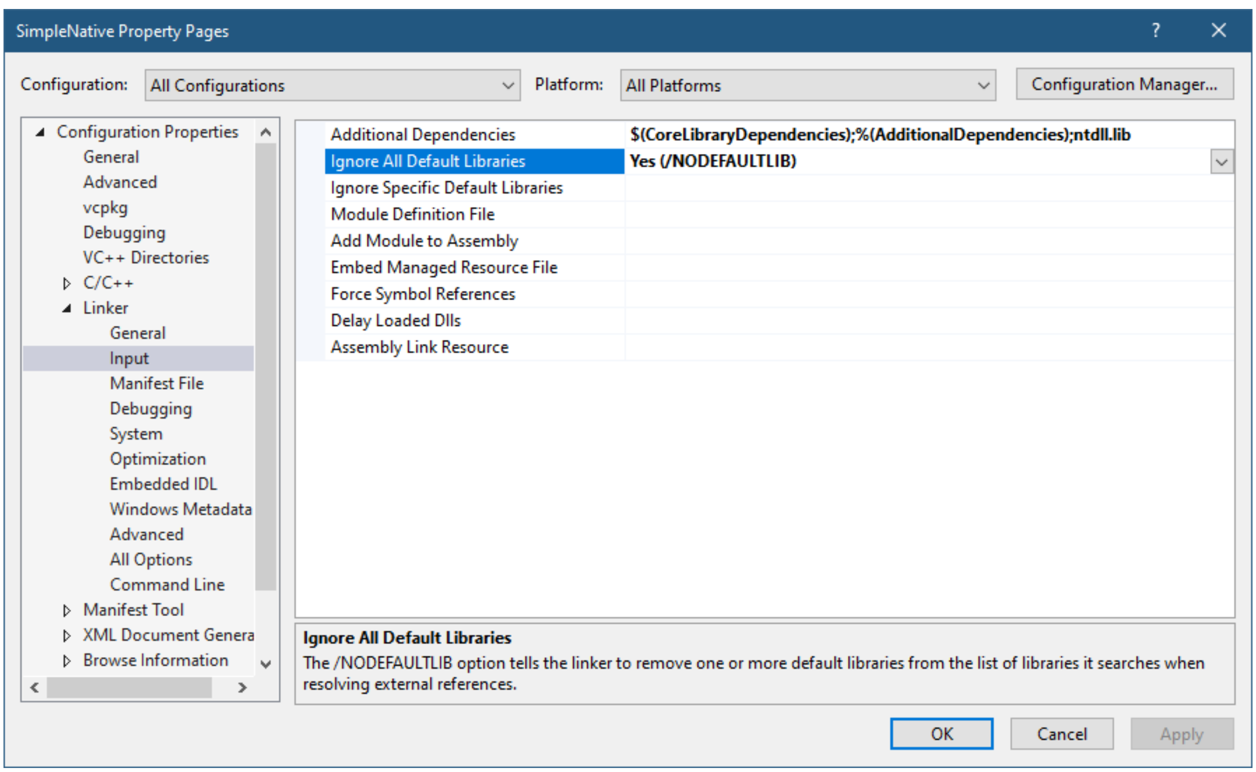


Figure 3-9: Linker Input options

- Navigate to the *Linker* / *System* node, and change the Subsystem to "Native" (figure 3-10).

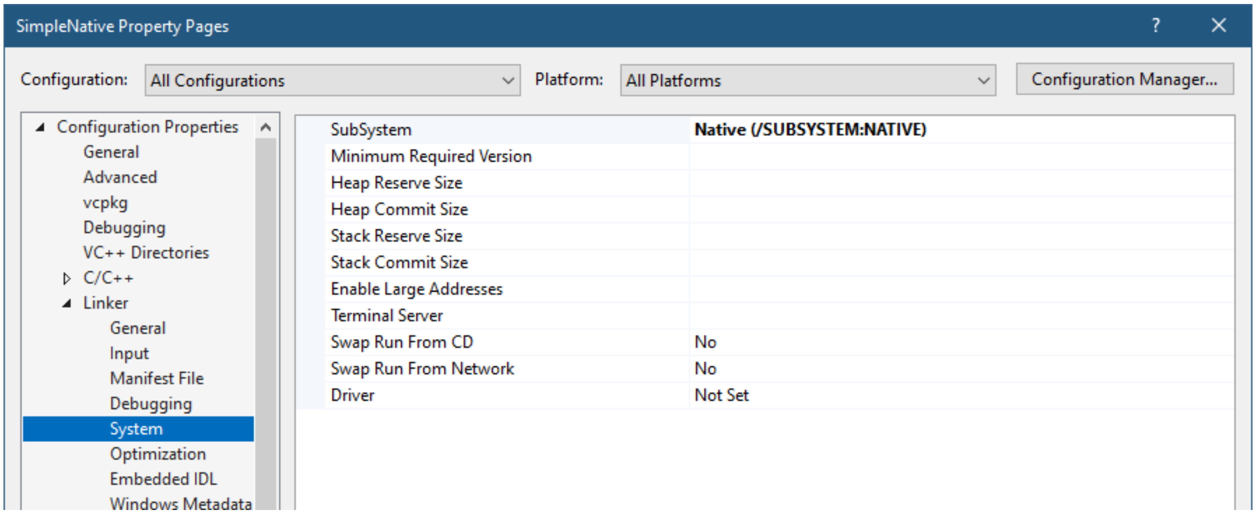


Figure 3-10: Linker System options

- Navigate to the *C/C++* / *Code Generation* node. Change "Basic Runtime Checks" to "Default", which effectively turns off runtime checks. This is required, because these checks are provided by the CRT. Change "Security Check" to "Disable Security Check". See figure 3-11.

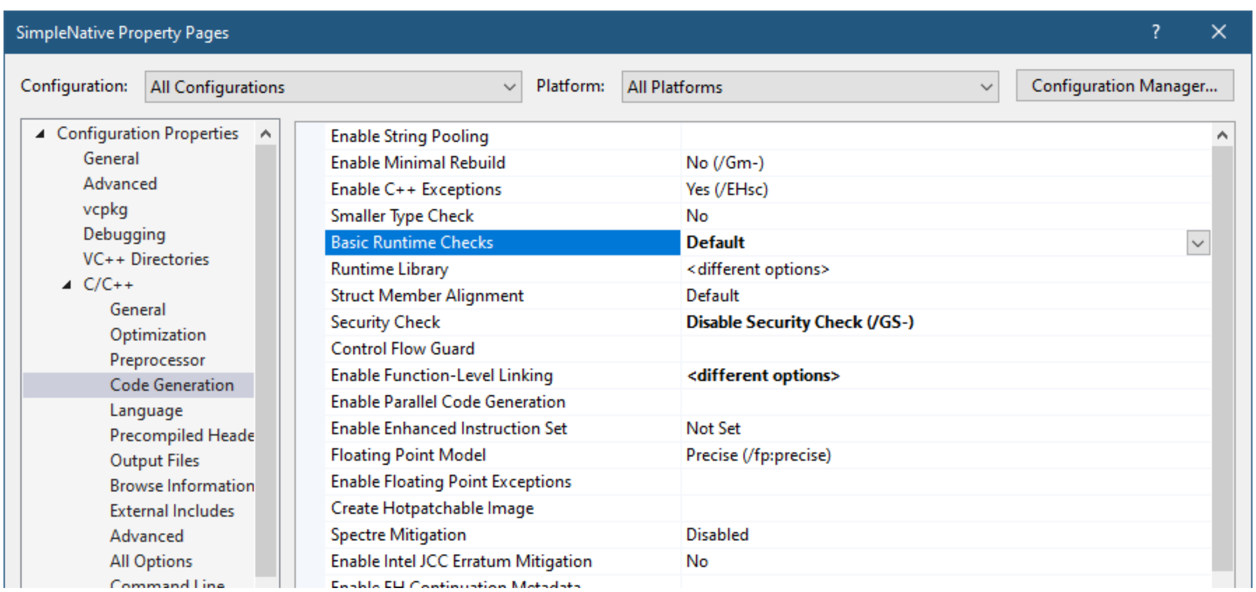


Figure 3-11: Compiler Code Generation options

- Navigate to the *C/C++* / *General* node. Change "SDL Checks" to "No". Change "Debug Information Format" to "Program Database". See figure 3-12.

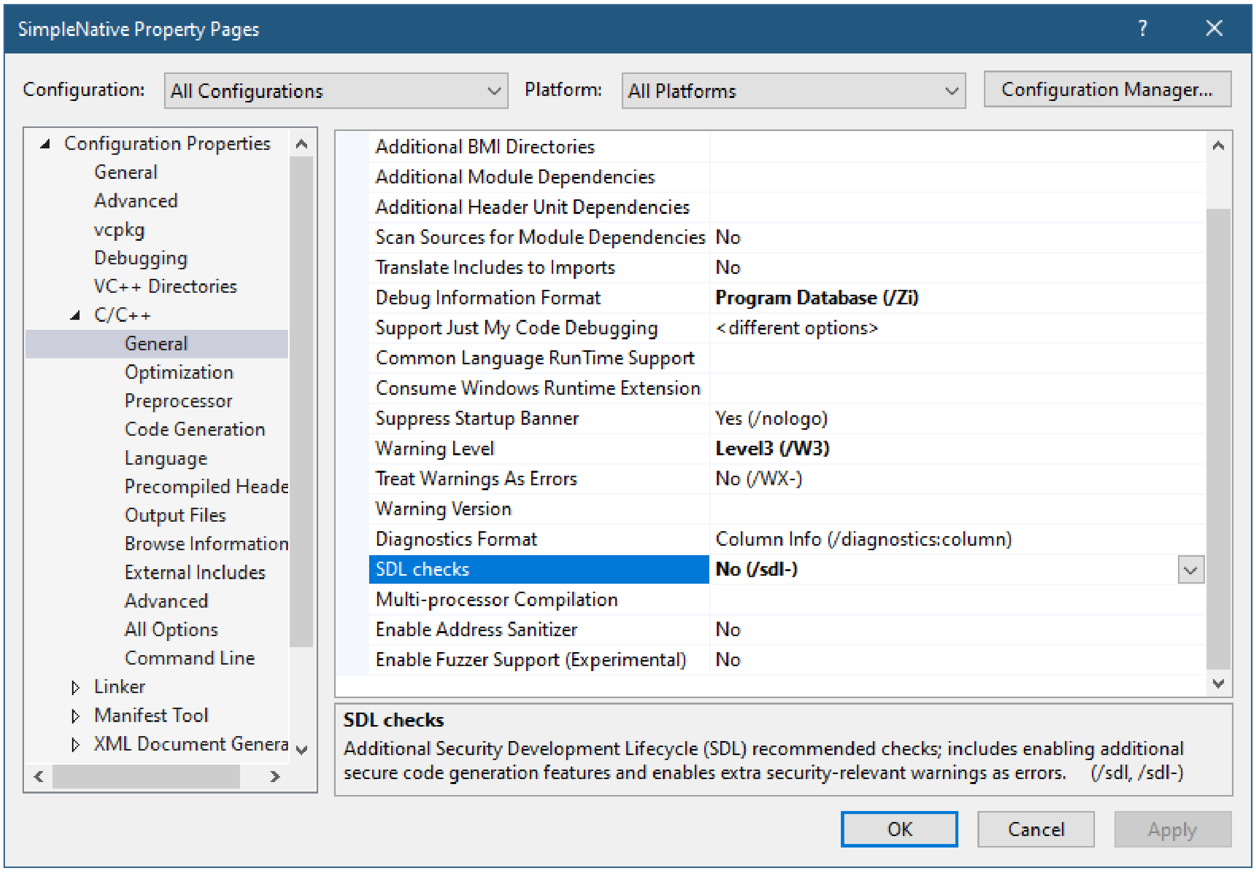


Figure 3-12: Compiler General options

This should be it. At this point, assuming the project has just an empty `main` function, it will compile but fail to link, complaining about an unresolved external called `NtProcessStartup`. As they say - we're not in Kansas anymore.

3.3: The Main Function

Standard console applications have a `main` function that can accept the number of arguments on the command line, an array of arguments, and even an array of environment variables:

```
int main(int argc, const char* argv[], const char* envp[]) {
```

Another supported variant is with Unicode strings:

```
int wmain(int argc, const wchar_t* argv[], const wchar_t* envp[]) {
```

A natural question to ask is who calls these functions with the correct arguments? It is the CRT. This is one reason the CRT provides the startup code for the process, with function names like `mainCRTStartup`; you can see that clearly when looking the the call stack if you set a breakpoint in the `main` function. In fact, the CRT source code is provided with Visual Studio, which means you can trace (and debug) these calls.

The CRT makes the call to the provided `main`/`wmain` with the expected semantics to adhere to the C/C++ standards. The CRT has other duties - for example, it's responsible for invoking constructors for global objects (before `main` is even invoked). This is necessary to comply with the C++ standard rules.

Now that we removed any dependency on the CRT (which we have to for native applications), there is no one that can call a standard `main` function, nor invoke constructors for global objects. The basic "main" for a Windows user-mode process looks like this:

```
NTSTATUS NtProcessStartup(PPEB peb);
```

The name of the function itself can be changed with a linker option, but the parameter and return type are as shown. The return value is conceptually the same as with the normal `main` function - the exit code of the process. The input, however, is fundamentally different - it's a pointer to the *Process Environment Block* (PEB).

We expect that information such as command line arguments to be present in the `PEB`, and they are. Specifically, they can be found in the `ProcessParameters` member, which is a large structure of type `RTL_USER_PROCESS_PARAMETERS`.

The `PEB` of the current process is always available with `NtCurrentPeb`.

3.4: Simple Native Application

Let's build a simple native application and register it with the *BootExecute* value. The application's name is *SimpleNative*, and it's configured with the steps in the previous section to make it a proper native application.

The application will display the Windows version on the boot screen, and pause for few seconds so we can see it. The main function will call `RtlGetVersion` to get some version information:

```
NTSTATUS NtProcessStartup(PPEB peb) {
    RTL_OSVERSIONINFOEXW osvi = { sizeof(osvi) };
    RtlGetVersion(&osvi);
```

`RtlGetVersion` is a simple API, defined like so:

```
NTSTATUS RtlGetVersion(_Out_ PRTL_OSVERSIONINFOW VersionInformation);
```

It expects one of two structures, `PRTL_OSVERSIONINFOW` or the extended `PRTL_OSVERSIONINFOEXW`, both defined in *<WinNt.h>*. The function itself, however, is not defined in the documented headers. Both structures have a first "size" member, that must be correctly initialized so the API knows how much information to provide:

```
typedef struct _OSVERSIONINFOW {
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    WCHAR szCSDVersion[ 128 ];
} OSVERSIONINFOW, *POSVERSIONINFOW, RTL_OSVERSIONINFOW, *PRTL_OSVERSIONINFOW;

typedef struct _OSVERSIONINFOEXW {
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    WCHAR szCSDVersion[ 128 ];
    WORD  wServicePackMajor;
    WORD  wServicePackMinor;
    WORD  wSuiteMask;
    BYTE  wProductType;
    BYTE  wReserved;
} OSVERSIONINFOEXW, *POSVERSIONINFOEXW, RTL_OSVERSIONINFOEXW, *PRTL_OSVERSIONINFOEXW;
```

Once we have the information, we need to build a string that uses some of the returned members. We can use the `swprintf_s` function that is exported from *NtDll.dll*, and thus available. Unfortunately, the *phnt* headers don't provide the prototypes for the CRT-like functions, so we need to define it ourselves. Fortunately, this is easy since the definition is known and documented for usage with the standard CRT:

```
extern "C" int swprintf_s(
    wchar_t* _Buffer, USHORT size,
    wchar_t const* _Format, ...);
```

> You may be temped to simply `#include` *<stdio.h>* to get the definition. Unfortunately, this won't compile, because the header has other C++ dependencies that get rejected by the compiler when there is no CRT. You can, however, copy the definition from the official headers.

Now we can build a string, and pass it to `NtDrawText` to show on the boot screen:

```
WCHAR text[256];
swprintf_s(text, ARRAYSIZE(text), L"Windows version: %d.%d.%d\n",
    osvi.dwMajorVersion, osvi.dwMinorVersion, osvi.dwBuildNumber);

UNICODE_STRING str;
RtlInitUnicodeString(&str, text);
NtDrawText(&str);
```

The string uses the major and minor versions of the OS, as well as the build number. Feel free to use other structure members of interest.

`NtDrawText` requires a `UNICODE_STRING`, so we initialize such a string with `RtlInitUnicodeString`, as what we have is a `NULL`-terminated string to begin with.

The last thing to do is to delay the thread for a few seconds before the process shuts down:

```
LARGE_INTEGER li;
li.QuadPart = -10000000 * 10;
NtDelayExecution(FALSE, &li);
```

`NtDelayExeution` is a rough native equivalent to the `SleepEx` Windows API. We'll discuss its full semantics in a later chapter, but for now the above code causes a 10 second sleep. Here is the full code for this sample:

```
#include <phnt_windows.h>
#include <phnt.h>

extern "C" int swprintf_s(
    wchar_t* _Buffer, USHORT size,
    wchar_t const* _Format, ...);

NTSTATUS NtProcessStartup(PPEB peb) {
    RTL_OSVERSIONINFOEXW osvi = { sizeof(osvi) };
    RtlGetVersion(&osvi);
    WCHAR text[256];
    swprintf_s(text, ARRAYSIZE(text), L"Windows version: %d.%d.%d\n",
        osvi.dwMajorVersion, osvi.dwMinorVersion, osvi.dwBuildNumber);

    UNICODE_STRING str;
    RtlInitUnicodeString(&str, text);
    NtDrawText(&str);

    LARGE_INTEGER li;
    li.QuadPart = -10000000 * 10;
```

```
    NtDelayExecution(FALSE, &li);

    return STATUS_SUCCESS;
}
```

Unfortunately, building the project fails with the linker complaining that `swprintf_s` is not found ("unresolved external"). How could this be?

As it turns out, the import library *NtDll.Lib* that is provided with the Windows SDK does not list the CRT-like functions that are implemented in *NtDll.Dll*, so the linker complains.

One way to resolve this is to bind to the function dynamically as discussed in chapter 2. This is certainly possible, but obviously we cannot use `GetModuleHandle` and `GetProcAddress`, as these are unavailable. The functions we need are `LdrGetDllHandle` and `LdrGetProcedureAddress`. We'll look at this in chapter 5. For now, there is another thing we can try.

We could build our own *NtDll.lib*. Remember, that an import library consists of two pieces of information: the DLL name (without any path information), and a list of exported symbols. There is no need to really implement anything, because it's implemented by the DLL itself. It's just about making the linker happy, so the binding to functions is deferred to runtime.

As it turns out, someone already did that. A Github project at https://github.com/Fyyre/ntdll provides such an "extended" *NtDll.lib*. To get the extended library, clone the repository (or download the code to some folder), open a Visual Studio command window, navigate to the project directory on your system, and type the following:

```
C:\Github\ntdll>nmake msvc
```

The result is two files, *NtDll64.lib* (for 64-bit) and *NtDll86.lib* (for 32-bit). Just copy the correct file (*NtDll64.lib* in our case, as only 64-bit executables work as native applications on 64-bit Windows), and use that file instead of the "standard" *NtDll.lib*. I have included this LIB file in the *SimpleNative* sample project.

Now the executable should compile and link successfully.

The next step, if we wish to run the executable on startup, is to copy the *SimpleNative.exe* file to the *System32* directory, and then to modify the *BootExecute* Registry value to include our executable. Figure 3-13 shows the change to the Registry value.

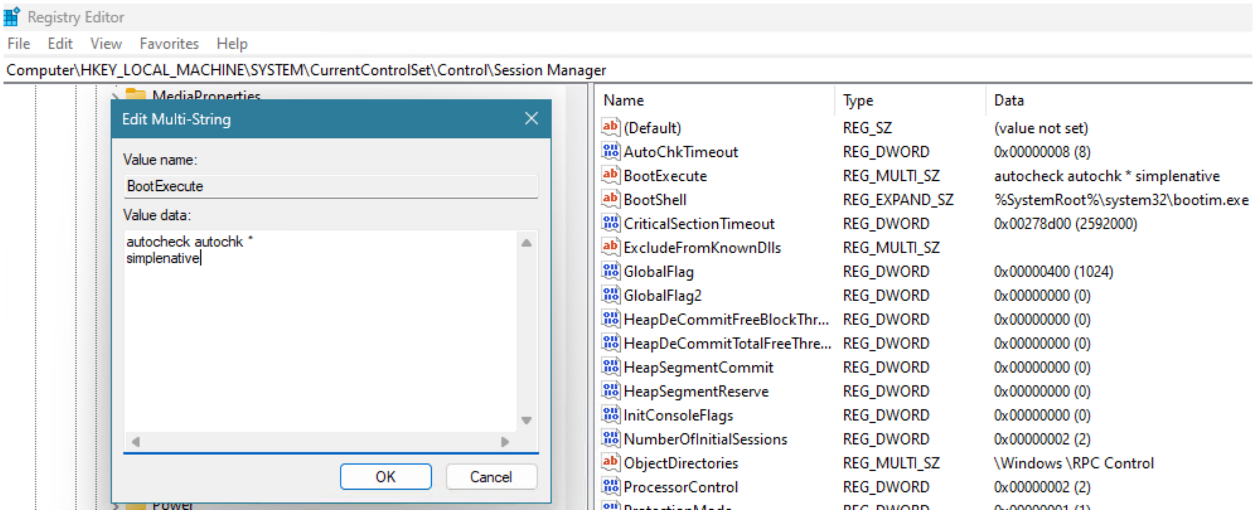


Figure 3-13: BootExecute with SimpleNative

Now reboot the machine (you can test on a Virtual Machine if you prefer), and you should see the Windows version string showing up for a few seconds (figure 3-14).

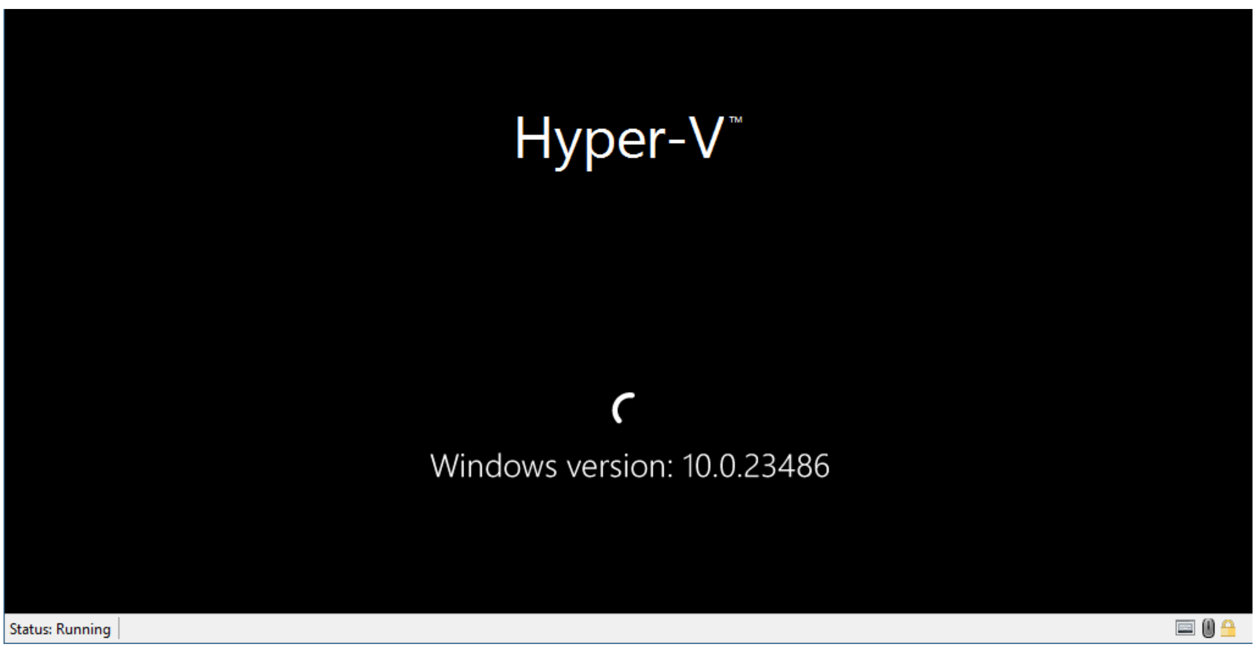


Figure 3-14: SimpleNative running

3.5: Launching Native Applications

There may be cases where we want to run a native application directly, without going through *BootExecute*. As we've seen already, trying to run such an application directly from a command window, Explorer, or similar means fails. In all these cases, the `CreateProcess` (or one of its variants like `CreateProcessAsUser`) is utilized, but these APIs cannot run native applications.

We'll create a standard Windows application that can run a native application, given its executable path and optional command line arguments.

Create a new C++ console application named *nativerun*. We'll add the usual *phnt* headers, since we'll have to use native APIs to accomplish launching native applications!

Let's start with a standard `wmain` function to get the native application path and any optional arguments to be passed to the executable:

```
int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 2) {
        printf("Usage: nativerun <executable> [arguments...]\n");
        return 0;
    }
```

Next, we need to build the command line arguments to the executable as a `UNICODE_STRING`. We'll use the `std::wstring` C++ class from the standard library for easy concatenation:

```
std::wstring args;
for (int i = 2; i < argc; i++) {
    args += argv[i];
    args += L" ";
}
UNICODE_STRING cmdline;
RtlInitUnicodeString(&cmdline, args.c_str());
```

We also need the executable path itself as a `UNICODE_STRING`:

```
UNICODE_STRING name;
RtlInitUnicodeString(&name, argv[1]);
```

Creating a process requires two steps: the first is creating a helper structure called `RTL_USER_PROCESS_PARAMETERS` by calling `RtlCreateProcessParameters` like so:

```
PRTL_USER_PROCESS_PARAMETERS params;
auto status = RtlCreateProcessParameters(&params, &name,
    nullptr, nullptr, &cmdline,
    nullptr, nullptr, nullptr, nullptr, nullptr);
```

There are quite a few parameters to this function, and we'll deal with these in more detail in chapter 5. For now, just note the name of the executable and the command line arguments passed to this function.

Next, we create the process by calling `RtlCreateUserProcess` with the parameters structure and other details like so:

```
RTL_USER_PROCESS_INFORMATION info;
status = RtlCreateUserProcess(&name, 0, params,
    nullptr, nullptr, nullptr, 0, nullptr, nullptr, &info);
RtlDestroyProcessParameters(params);
```

Detailed discussion of this API is deferred to chapter 5. The last parameter is an output structure that contains some details of the created process if successful. Part of that is provided in a `CreateProcess` call in the `PROCESS_INFORMATION` structure.

The created process has the first thread ready to go, but it's suspended initially. We need to resume the thread to kick the process into action:

```
auto pid = HandleToULong(info.ClientId.UniqueProcess);
printf("Process 0x%X (%u) created successfully.\n", pid, pid);

ResumeThread(info.ThreadHandle);
```

We can finally close the two handles we received back, even though in this case it's not a big deal, as our *NativeRun* process will soon exit:

```
CloseHandle(info.ThreadHandle);
CloseHandle(info.ProcessHandle);
```

Either `CloseHandle` or `NtClose` will do the job.

Here is the full code with error handling:

```
#include <phnt_windows.h>
#include <phnt.h>
#include <stdio.h>
#include <string>

#pragma comment(lib, "ntdll")

int Error(NTSTATUS status) {
    printf("Error (status=0x%08X)\n", status);
    return 1;
}

int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 2) {
        printf("Usage: nativerun <executable> [arguments...]\n");
        return 0;
```

```
    }

    //
    // build command line arguments
    //
    std::wstring args;
    for (int i = 2; i < argc; i++) {
        args += argv[i];
        args += L" ";
    }
    UNICODE_STRING cmdline;
    RtlInitUnicodeString(&cmdline, args.c_str());

    UNICODE_STRING name;
    RtlInitUnicodeString(&name, argv[1]);

    PRTL_USER_PROCESS_PARAMETERS params;
    auto status = RtlCreateProcessParameters(&params, &name,
        nullptr, nullptr, &cmdline,
        nullptr, nullptr, nullptr, nullptr, nullptr);
    if (!NT_SUCCESS(status))
        return Error(status);

    RTL_USER_PROCESS_INFORMATION info;
    status = RtlCreateUserProcess(&name, 0, params, nullptr,
        nullptr, nullptr, 0, nullptr, nullptr, &info);
    if (!NT_SUCCESS(status))
        return Error(status);

    RtlDestroyProcessParameters(params);

    auto pid = HandleToULong(info.ClientId.UniqueProcess);
    printf("Process 0x%X (%u) created successfully.\n", pid, pid);

    ResumeThread(info.ThreadHandle);
    CloseHandle(info.ThreadHandle);
    CloseHandle(info.ProcessHandle);

    return 0;
}
```

Let's test the application by invoking a native application like *SimpleNative* from the previous section. Open a command window, navigate to the location of the resulting *NativeRun.exe* file, and type the following:

```
C:\Chapter03\x64\Debug>nativerun.exe SimpleNative.exe
Error (status=0xC000003B)
```

We get an error, whose value means "Object Path Component was not a directory object". Well, maybe we need a full path:

```
C:\Chapter03\x64\Debug>nativerun.exe C:\Chapter03\x64\Debug\SimpleNative.exe
Error (status=0xC000003B)
```

Same result. The problem is that the native API expects paths to be in "NT" style rather than Win32 style. This means paths must be based on the Object Manager's namespace. It may seem that something like "C:" is very fundamental, but it's not. Drive letters get special treatment by standard Windows APIs.

One way to fix this is to prepend "??" before continuing with the normal path like so:

```
C:\Chapter03\x64\Debug>nativerun.exe \??\C:\Chapter03\x64\Debug\SimpleNative.exe
Process 0xAC48 (44104) created successfully.
```

What does this strange "\??\" mean? It's a directory in the Object Manager's namespace where symbolic links are stored, one of which is "C:". You can view all this with the *Sysinternals WinObj* tool, or my own *Object Explorer*. Figure 3-15 shows *WinObj* displaying the "\Global??\" directory (another name for "\??\"). It also shows the target of this symbolic link: "Device\Harddiskvolume3" (on your system it may be different).

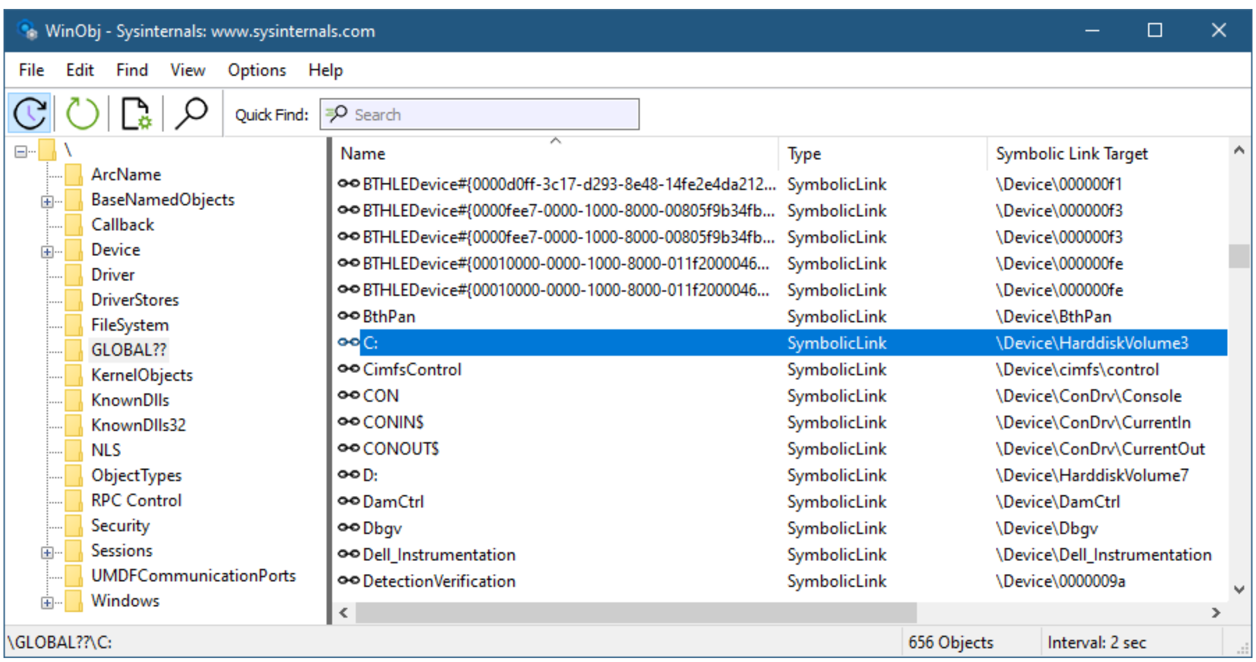


Figure 3-15: WinObj showing symbolic links

You can replace "\??\c:" with "\Device\Harddiskvolume3" and it would work just as well:

```
C:\Chapter03\x64\Debug>nativerun.exe \Device\harddiskvolume3\Chapter03\x64\Debug\Sim\
pleNative.exe
Process 0x93B0 (37808) created successfully.
```

3.6: Debugging Native Applications

Debugging a native application presents some challenges. Since `CreateProcess` and its variants are unable to launch a native application, standard debuggers like Visual Studio or WinDbg are unable to launch such executables.

However, once such an executable is launched, these debuggers are perfectly capable to attach to the running process. A simple tweak would be to add a call to `NtDelayExecution` at the beginning of the `NtProcessStartup` function to have a few seconds time in which to ask Visual Studio to attach to the process.

We can do better, however. When creating a native application process, the main thread does not start running, so we can take advantage of this and attach a debugger at this point, and then let the thread resume execution. It would be simple enough to add support for that in *NativeRun* like so:

```
int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 2) {
        printf("Usage: nativerun [-d] <executable> [arguments...]\n");
        return 0;
    }

    int start = 1;
    bool debug = _wcsicmp(argv[1], L"-d") == 0;
    if (debug)
        start = 2;

    std::wstring args;
    for (int i = start + 1; i < argc; i++) {
        args += argv[i];
        args += L" ";
    }
    UNICODE_STRING cmdline;
    RtlInitUnicodeString(&cmdline, args.c_str());

    UNICODE_STRING name;
    RtlInitUnicodeString(&name, argv[start]);

    PRTL_USER_PROCESS_PARAMETERS params;
    auto status = RtlCreateProcessParameters(&params, &name,
```

```
        nullptr, nullptr, &cmdline,
        nullptr, nullptr, nullptr, nullptr, nullptr);
    if (!NT_SUCCESS(status))
        return Error(status);

    RTL_USER_PROCESS_INFORMATION info;
    status = RtlCreateUserProcess(&name, 0, params, nullptr,
        nullptr, nullptr, 0, nullptr, nullptr, &info);
    if (!NT_SUCCESS(status))
        return Error(status);

    RtlDestroyProcessParameters(params);

    auto pid = HandleToULong(info.ClientId.UniqueProcess);
    printf("Process 0x%X (%u) created successfully.\n", pid, pid);

    if (debug) {
        printf("Attach with a debugger. Press ENTER to resume thread...\n");
        char dummy[3];
        gets_s(dummy);
    }
    ResumeThread(info.ThreadHandle);
    CloseHandle(info.ThreadHandle);
    CloseHandle(info.ProcessHandle);

    return 0;
}
```

3.6.1: Debugging *BootExeutable* Native Applications

What about debugging *BootExecute* applications? These run too early to use any user-mode debugger. For these, we have to use a kernel-mode debugger, such as *WinDbg*. The configuration of the debugger and target (where the native application will execute) is done in the same manner as any other kernel debugging setup.

> If you are not familiar with kernel-mode debugging, consult the *WinDbg* documentation or look online. Also, chapter 5 in my book “Windows Kernel Programming” has a good introduction to WinDbg for user-mode and kernel-mode debugging.

As a reminder, copy the executable’s PDB (symbols) file to the target’s *System32* directory (where the executable is) to make it easy for the debugger to locate the symbols.

Once the target restarts and the initial debugger breakpoint is hit, you can set an event to break when the native application executable is loaded:

```
!gflag +ksl
sxe ld:simplenative.exe
```

The first command instructs the debugger to load kernel symbols automatically rather than be lazy about it. It improves its ability to satisfy the second command which indicates that the first time the image name indicated is loaded, the debugger should break.

At the breakpoint, the process is (barely) created. In fact, the command `!process 0 0` will not show it. However, you can view the basics with `!process @$proc` or `!process -1`, because it's the current process:

```
2: kd> !process @$proc
PROCESS ffff820384609140
    SessionId: none  Cid: 0000    Peb: 00000000  ParentCid: 02bc
    DirBase: 132381000  ObjectTable: ffffe689c2516d40  HandleCount:   0.
    Image: SimpleNative.exe
    VadRoot ffff820382c75960 Vads 3 Clone 0 Private 12. Modified 0. Locked 0.
    DeviceMap 0000000000000000
    Token                             ffffe689c255a360
    ElapsedTime                       00:00:00.000
    UserTime                          00:00:00.000
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         4280
    QuotaPoolUsage[NonPagedPool]      408
    Working Set Sizes (now,min,max)  (12, 50, 345) (48KB, 200KB, 1380KB)
    PeakWorkingSetSize                4
    VirtualSize                       0 Mb
    PeakVirtualSize                   0 Mb
    PageFaultCount                    12
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      26

No active threads
```

Unfortunately, the debugger is not happy enough with the process, so any breakpoints set will fail or be ignored, for example:

```
bp simplenative!NtProcessStartup
bp /p ffff820384609140 simplenative!NtProcessStartup
```

Setting breakpoints in *NtDll.dll* similarly fails. The best way I have found to make it work is to artificially add `NtDelayExecution` code to the beginning of `NtProcessStartup` and then forcefully break into the debugger while the main thread is sleeping. Then we can set a breakpoint normally using commands or by opening the source file and pressing F9 where a breakpoint should hit.

3.7: Summary

In this chapter we looked at native applications that depend on *NtDll.Dll* only. These are mostly useful for running at system boot, by *Smss.exe*. Once we learn more about the native API you may come up with ideas of things to do at this early time in the Windows boot process.

In the next chapter, we'll look at system information that can be obtained and/or changed with the native API.

Chapter 4: System Information

In this chapter we'll focus on getting and setting various system-level details. Some of the native API capabilities have Windows API counterparts, but some do not. The chapter is not exhastive by any means - the sheer number of information classes used with `NtQueryInformationClass` is staggering. The chapter covers some of the more "interesting" or non-trivial system information details, that can be difficult to intuitively use.

In this chapter:

- **Querying and Setting Information**
- **General System Information**
- **Processes and Threads**
- **Handles and Objects**
- **The `KUSER_SHARED_DATA` Structure**
- **Miscellaneous Information**

4.1: Querying and Setting Information

There are two most useful functions for getting and setting system-related information: `NtQuerySystemInformation` and its complementary `NtSetSystemInformation`:

```
NTSTATUS NtQuerySystemInformation(
    _In_ SYSTEM_INFORMATION_CLASS SystemInformationClass,
    _Out_writes_bytes_opt_(SystemInformationLength) PVOID SystemInformation,
    _In_ ULONG SystemInformationLength,
    _Out_opt_ PULONG ReturnLength);

NTSTATUS NtSetSystemInformation(
    _In_ SYSTEM_INFORMATION_CLASS SystemInformationClass,
    _In_reads_bytes_opt_(SystemInformationLength) PVOID SystemInformation,
    _In_ ULONG SystemInformationLength);
```

Both functions are generic, based on a `SYSTEM_INFORMATION_CLASS` enumeration and some associated expected buffer. For query, there are cases where the expected buffer size is fixed, but in others the information retrieved is dynamic, so the size may not be known in advance. A typical pattern is to call `NtQuerySystemInformation` once with a `NULL` buffer and a size of zero to get back the required size (in the last argument) (the return value in this case is `STATUS_INFO_LENGTH_MISMATCH`). Next, the buffer is allocated, and the call is reissued, but this time with the real buffer and the allocated size. Care must be taken in some cases to allocate a bit more than the returned size indicates in case the information size has increased between getting the required size and issuing the second call. For example, if a list of processes is requested, new processes may have been added just after the required size is returned.

The `SYSTEM_INFORMATION_CLASS` enumeration is a large one, and keeps growing with newer versions of Windows. Since these APIs are generic, their prototypes don't need to change. We'll spend some time to look at some enumeration values and how to use them.

> The *phnt* definition of `SYSTEM_INFORMATION_CLASS` provides comments for most values and the expected structure, and whether the enumeration supports querying, setting, or both.

Another query variant exists which takes an input buffer (not just an output buffer):

```
NTSTATUS NtQuerySystemInformationEx(
    _In_ SYSTEM_INFORMATION_CLASS SystemInformationClass,
    _In_reads_bytes_(InputBufferLength) PVOID InputBuffer,
    _In_ ULONG InputBufferLength,
    _Out_writes_bytes_opt_(SystemInformationLength) PVOID SystemInformation,
    _In_ ULONG SystemInformationLength,
    _Out_opt_ PULONG ReturnLength);
```

The input buffer serves as an extra detail that affects the output. Only a subset of system information classes support this API. A `NULL` input buffer or an input length of zero are invalid - the function fails rather than call `NtQuerySystemInformation`.

4.1.1: General System Information

In this section, we'll examine some system information classes that provide various general details for the system as a whole.

4.1.1.1: `SystemBasicInformation` (0) (Query)

Using `SystemBasicInformation` is possible with query only, returning a `SYSTEM_BASIC_INFORMATION` structure:

```
typedef struct _SYSTEM_BASIC_INFORMATION {
    ULONG Reserved;
    ULONG TimerResolution;
    ULONG PageSize;
    ULONG NumberOfPhysicalPages;
    ULONG LowestPhysicalPageNumber;
    ULONG HighestPhysicalPageNumber;
    ULONG AllocationGranularity;
    ULONG_PTR MinimumUserModeAddress;
    ULONG_PTR MaximumUserModeAddress;
    ULONG_PTR ActiveProcessorsAffinityMask;
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION, *PSYSTEM_BASIC_INFORMATION;
```

Some of the above information is available with the Windows API functions `GetSystemInfo` and `GetNativeSystemInfo`.

Most members are self-explanatory, or are explained in the documentation of the `SYSTEM_INFO` structure. A few that may not be:

- `TimerResolution` - number of 100 nsec units used for the default timer ticks, used with thread scheduling. This value should be 156250 (15.625 msec) on most systems. This value is the maximum number, where the current timer resolution and the minimum one is available with another native API, `NtQueryTimerResolution`, discussed later in this chapter.
- `LowestPhysicalPageNumber`, `HighestPhysicalPageNumber` - lowest and highest physical page numbers supported on the system. The lowest is typically 1, meaning the first page in physical memory is not used.
- `NumberOfPhysicalPages` - total number of physical pages, indicating the amount of RAM supported on the system (multiply by `PAGE_SIZE` (4 KB) to get the number of bytes.
- `NumberOfProcessors` - this might seem obvious, but notice the type (`CCHAR`) - it's between 0 and 255. Windows supports more than 255 logical processors. This number is actually the number of processors in the current processor group, which can be between 1 and 64. The total number of processors is given by a different system information class (`SystemProcessorInformation`), discussed next.

4.1.1.2: `SystemProcessorInformation` (1) (Query)

This information class returns a structure of type `SYSTEM_PROCESSOR_INFORMATION`:

```c
typedef struct _SYSTEM_PROCESSOR_INFORMATION {
    USHORT ProcessorArchitecture;
    USHORT ProcessorLevel;
    USHORT ProcessorRevision;
    USHORT MaximumProcessors;
    ULONG ProcessorFeatureBits;
} SYSTEM_PROCESSOR_INFORMATION, *PSYSTEM_PROCESSOR_INFORMATION;
```

The total number of logical processors in the system is returned in `MaximumProcessors` (not just the current processor group). The other members are documented as part of the `SYSTEM_INFO` structure, except `ProcessorFeatureBits`, which provides CPU-specific feature bits.

4.1.1.3: `SystemPerformanceInformation` (2) (Query)

The returned structure is of type `SYSTEM_PERFORMANCE_INFORMATION`, and it's a huge one. Most members are self-explanatory.

4.1.1.4: `SystemTimeOfDayInformation` (3) (Query)

This information class returns a structure of type `SYSTEM_TIMEOFDAY_INFORMATION`:

```c
typedef struct _SYSTEM_TIMEOFDAY_INFORMATION {
    LARGE_INTEGER BootTime;
    LARGE_INTEGER CurrentTime;
    LARGE_INTEGER TimeZoneBias;
    ULONG TimeZoneId;
    ULONG Reserved;
    ULONGLONG BootTimeBias;
    ULONGLONG SleepTimeBias;
} SYSTEM_TIMEOFDAY_INFORMATION, *PSYSTEM_TIMEOFDAY_INFORMATION;
```

- `BootTime` is the local time the system booted (in the standard 100 nsec units since January 1, 1601).
- `CurrentTime` is the current local time.
- `TimeZoneBias` is the difference (in the usual 100 nsec units) from Universal Time.
- `BootBiasTime` and `SleepBiasTime` are typically zero.
- `TimeZoneId` is the index of the time zone configured.

The current time zone information is available with `RtlQueryTimeZoneInformation`:

```
typedef struct _RTL_TIME_ZONE_INFORMATION {
    LONG Bias;
    WCHAR StandardName[32];
    TIME_FIELDS StandardStart;
    LONG StandardBias;
    WCHAR DaylightName[32];
    TIME_FIELDS DaylightStart;
    LONG DaylightBias;
} RTL_TIME_ZONE_INFORMATION, *PRTL_TIME_ZONE_INFORMATION;
NTSTATUS RtlQueryTimeZoneInformation(
    _Out_ PRTL_TIME_ZONE_INFORMATION TimeZoneInformation);
```

The returned names may not be normal strings, but rather have a *Multiple User Interface* (MUI) format involving a DLL and an ID. For example: "@tzres.dll,-112". This means the string is a resource ID 112 in a DLL named *tzres.dll* (in one of the system directories, typically *System32*) or one based on the current locale. For example, if the locale is "en-US", then the file where the string is actually located is *System32\en-US\tzres.dll.mui".

Reaching into these DLLs to locate the actual string is tedious and error prone. Fortunately, the Shell provides an API that does the work:

```
HRESULT SHLoadIndirectString(
    _In_ PCWSTR pszSource,
    _Out_writes_(cchOutBuf) PWSTR pszOutBuf,
    _In_ UINT cchOutBuf,
    _Reserved_ void **ppvReserved);
```

Here is an example usage to bring the standard name of the current time zone:

```
RTL_TIME_ZONE_INFORMATION tzinfo;
RtlQueryTimeZoneInformation(&tzinfo);
WCHAR text[64];
SHLoadIndirectString(tzinfo.StandardName, text, _countof(text), nullptr);
printf("Time zone: %ws\n", text);
```

The full example is part of the *SysInfo* sample project for this chapter.

4.1.2: Timer Resolution

The kernel works with an internal timer, used to control all timing-related activities, such as wait times, scheduler wakeups, etc. The basic "tick" of this timer is queryable and controllable with the following APIs:

```
NTSTATUS NtQueryTimerResolution(
    _Out_ PULONG MaximumTime,
    _Out_ PULONG MinimumTime,
    _Out_ PULONG CurrentTime);

NTSTATUS NtSetTimerResolution(
    _In_ ULONG DesiredTime,
    _In_ BOOLEAN SetResolution,
    _Out_ PULONG ActualTime);
```

`NtQueryTimerResolution` returns the maximum, minimum and current timer tick interval, in the usual 100n sec units.

The *clockres Sysinternals* tool shows exactly these values.

Since multiple processes may want to affect the timer resolution, the kernel keeps track of each change done by which process. The highest resolution requested is the one used, but once a process exits, or removes its resolution effect (calling `NtSetTimerResolution` with `SetResolution` being `FALSE`), the kernel will reduce the resolution (increment the tick count) if other processes did not request that small tick.

The last argument to `NtSetTimerResolution` returns the actual value set, based on the hardware's capabilities and other process requests, in 100 nsec unites (the same units used for `DesiredTime`).

4.1.3: Processor Dynamic Information

Several information classes provide dynamic information on each processor in the current processor group or in a specific processor group if calling `NtQuerySystemInformationEx`. Table 4-1 lists the information classes, and their corresponding structures. The processor group number can be specified as the input buffer (`USHORT`), or the current processor group is used if `NtQuerySystemInformation` is called. Note that the special `ALL_PROCESSOR_GROUPS` value is invalid for these calls.

Table 4-1: Processor information classes

Information class	Structure type
`SystemProcessorPerformanceInformation` (8)	`SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION`
`SystemInterruptInformation` (23)	`SYSTEM_INTERRUPT_INFORMATION`
`SystemProcessorIdleInformation` (42)	`SYSTEM_PROCESSOR_IDLE_INFORMATION`
`SystemProcessorPowerInformation` (61)	`SYSTEM_PROCESSOR_POWER_INFORMATION`
`SystemProcessorIdleCycleTimeInformation` (83)	`SYSTEM_PROCESSOR_IDLE_CYCLE_TIME_INFORMATION`
`SystemProcessorPerformanceDistribution` (100)	`SYSTEM_PROCESSOR_PERFORMANCE_DISTRIBUTION`
`SystemProcessorCycleTimeInformation` (108)	`SYSTEM_PROCESSOR_CYCLE_TIME_INFORMATION`

Information class	Structure type
SystemProcessorPerformanceInformationEx (141)	SYSTEM_PROCESSOR_PERFORMANCE_-INFORMATION_EX
SystemLogicalProcessorInformation (73)	SYSTEM_LOGICAL_PROCESSOR_INFORMATION
SystemProcessorCycleStatsInformation (160)	SYSTEM_PROCESSOR_CYCLE_STATS_INFORMATION

The expected output buffer must be a multiple of a single structure, one instance for each processor information required. If zero size is provided for the output buffer, the API returns the needed size for all processors in the requested group.

The following example shows one way to get one of these processor details:

```
ULONG len;
USHORT group = 0;   // group 0
NtQuerySystemInformationEx(SystemProcessorPerformanceInformation,
    &group, sizeof(group), nullptr, 0, &len);
ULONG cpuCount = len / sizeof(SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION);
auto pi = std::make_unique<SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION[]>(cpuCount);
NtQuerySystemInformationEx(SystemProcessorPerformanceInformation,
    &group, sizeof(group), pi.get(), len, nullptr);
for(ULONG i = 0; i < cpuCount; i++) {
    // use pi[i] to access information for CPU i
}
```

Let's examine one of the basic structures, `SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION`:

```
typedef struct _SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION {
    LARGE_INTEGER IdleTime;
    LARGE_INTEGER KernelTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER DpcTime;
    LARGE_INTEGER InterruptTime;
    ULONG InterruptCount;
} SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION, *PSYSTEM_PROCESSOR_PERFORMANCE_INFORMATI\
ON;
```

Here is the breakdown of the members (all times are in 100 nsec units):

- `IdleTime` - the time the processor was idle.
- `KernelTime` - the time the processor spent executing code in kernel-mode. Note that this time includes `IdleTime`, which is considered kernel mode execution.
- `UserTime` - the time the processor spent executing in user-mode.

- `DpcTime` - the time the CPU spend executing *Deferred Procedure Calls* (DPCs). These routines are provided by kernel drivers and typically execute after *Interrupt Service Routines* (ISRs) execute. Full discussion of DPCs and ISRs is beyond the scope of this book.
- `InterruptTime` - the time the CPU spend executing ISRs.
- `InterruptCount` - the number of interrupts serviced by the processor.

> The *CpuInfo* sample provides a full example.

4.1.4: Module Information

The loaded kernel modules (the kernel itself, the HAL, and all drivers) can be obtained with two information classes, `SystemModuleInformation` and `SystemModuleInformationEx`. Here are the associated structures:

```
typedef struct _RTL_PROCESS_MODULE_INFORMATION {
    HANDLE Section;
    PVOID MappedBase;
    PVOID ImageBase;
    ULONG ImageSize;
    ULONG Flags;
    USHORT LoadOrderIndex;
    USHORT InitOrderIndex;
    USHORT LoadCount;
    USHORT OffsetToFileName;
    UCHAR FullPathName[256];
} RTL_PROCESS_MODULE_INFORMATION, *PRTL_PROCESS_MODULE_INFORMATION;

typedef struct _RTL_PROCESS_MODULES {
    ULONG NumberOfModules;
    RTL_PROCESS_MODULE_INFORMATION Modules[1];
} RTL_PROCESS_MODULES, *PRTL_PROCESS_MODULES;

typedef struct _RTL_PROCESS_MODULE_INFORMATION_EX {
    USHORT NextOffset;
    RTL_PROCESS_MODULE_INFORMATION BaseInfo;
    ULONG ImageChecksum;
    ULONG TimeDateStamp;
    PVOID DefaultBase;
} RTL_PROCESS_MODULE_INFORMATION_EX, *PRTL_PROCESS_MODULE_INFORMATION_EX;
```

Although the name "process" is part of these structure names, they just describe modules - kernel or user. In the context of the above information classes, it's the former.

The details of these members will be discussed in chapter 5. For kernel modules (as used here), the section handle and mapped base are always zero.

The following shows how to get module information using the extended information class:

```
ULONG size;
NtQuerySystemInformation(SystemModuleInformationEx, nullptr, 0, &size);
std::unique_ptr<BYTE[]> buffer;
for (;;) {
    buffer = std::make_unique<BYTE[]>(size);
    auto status = NtQuerySystemInformation(SystemModuleInformationEx,
        buffer.get(), size, &size);
    if (NT_SUCCESS(status))
        break;
}

auto mod = (RTL_PROCESS_MODULE_INFORMATION_EX*)buffer.get();
for (;;) {
    // do something with mod

    if (mod->NextOffset == 0)
        break;

    mod = (RTL_PROCESS_MODULE_INFORMATION_EX*)((PBYTE)mod + mod->NextOffset);
}
```

The full code is in the *ModList* sample.

4.2: Process and Thread Information

`NtQuerySystemInformation` has several information classes for providing details on processes and threads running on the system. The following lists the information class values and the associated structures.

- `SystemProcessInformation` (5) provides `SYSTEM_PROCESS_INFORMATION` (process) and `SYSTEM_THREAD_INFORMATION` array (threads), see figure 4-1.
- `SystemExtendedProcessInformation` (57) provides `SYSTEM_PROCESS_INFORMATION` (process) and `SYSTEM_EXTENDED_THREAD_INFORMATION` array (threads), see figure 4-2.
- `SystemFullProcessInformation` (148) provides `SYSTEM_PROCESS_INFORMATION` followed by `SYSTEM_EXTENDED_THREAD_INFORMATION` array (threads), and then `SYSTEM_PROCESS_INFORMATION_EXTENSION` (more information about the process), see figure 4-3. Using this information class requires admin rights.

Task Manager uses this API to enumerate processes, and *Process Explorer* uses these APIs to enumerate processes and threads.

For processes, the basic structure looks like so:

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    LARGE_INTEGER WorkingSetPrivateSize;
    ULONG HardFaultCount;
    ULONG NumberOfThreadsHighWatermark;
    ULONGLONG CycleTime;
    LARGE_INTEGER CreateTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER KernelTime;
    UNICODE_STRING ImageName;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    HANDLE InheritedFromUniqueProcessId;
    ULONG HandleCount;
    ULONG SessionId;
    ULONG_PTR UniqueProcessKey; // (requires SystemExtendedProcessInformation)
    SIZE_T PeakVirtualSize;
    SIZE_T VirtualSize;
    ULONG PageFaultCount;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    SIZE_T QuotaPeakPagedPoolUsage;
    SIZE_T QuotaPagedPoolUsage;
    SIZE_T QuotaPeakNonPagedPoolUsage;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
    SIZE_T PrivatePageCount;
    LARGE_INTEGER ReadOperationCount;
    LARGE_INTEGER WriteOperationCount;
    LARGE_INTEGER OtherOperationCount;
    LARGE_INTEGER ReadTransferCount;
    LARGE_INTEGER WriteTransferCount;
    LARGE_INTEGER OtherTransferCount;
    SYSTEM_THREAD_INFORMATION Threads[1];
} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;
```

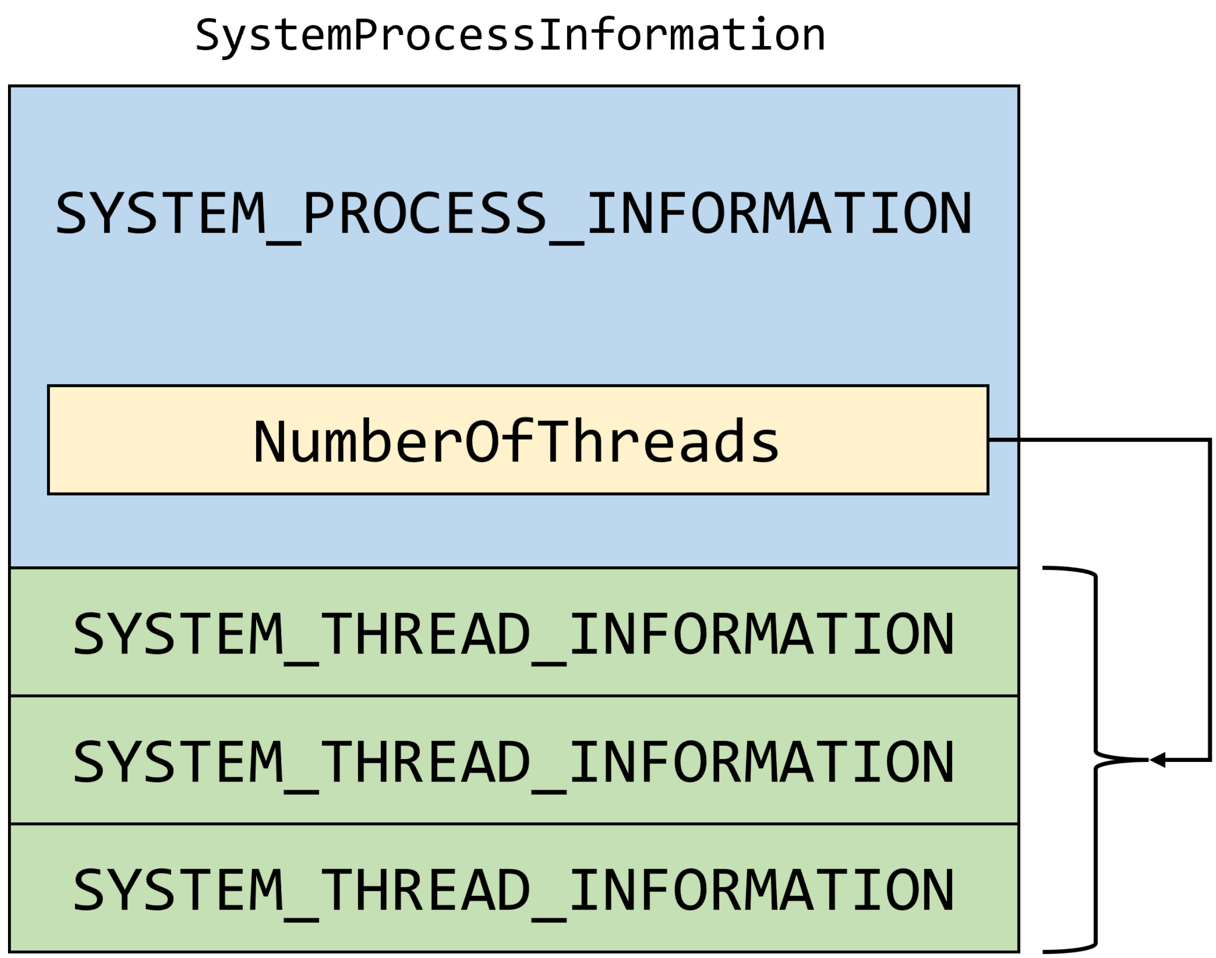


Figure 4-1: SystemProcessInformation details

Most of its members are self-explanatory. Here are a few that may be not obvious:

- `ImageName` is the executable path in NT device form ("\Device\HarddiskVolume…") rather than Win32 path. For kernel processes, the special name is provided: "System", "Secure System", "Registry", "Memory Compression". The *Idle* process (PID 0) is provided with no name.
- `InheritedFromUniqueProcessId` is the process parent ID.
- `NextEntryOffset` is the key to moving to the next process. This is the byte offset of the next process in the list. The enumeration should terminate when this value is zero.
- `NumberOfThreadsHighWatermark` is the highest number of threads that ever existed in this process.
- `UniqueProcessKey` is the same as the process ID, since the process uniqueness is defined by its ID for as long as the object is alive.
- `VirtualSize` describes the total usage of address space in the process (committed and reserved memory). All other memory counters don't consider reserved memory. More details on memory counters are provided in chapter 7.

The `*Pool` members represent kernel memory that is attributes to this process. For example, handle entries are allocated from the paged memory pool in the kernel.

The following example shows how to obtain process information and iterate through the process list:

```
ULONG size = 0;
auto status = NtQuerySystemInformation(SystemProcessInformation, nullptr, 0, &size);
std::unique_ptr<BYTE[]> buffer;
while(status == STATUS_INFO_LENGTH_MISMATCH) {
    size += 1024;   // in case some processes get created
    buffer = std::make_unique<BYTE[]>(size);
    status = NtQuerySystemInformation(SystemProcessInformation,
        buffer.get(), size, &size);
}

auto p = (SYSTEM_PROCESS_INFORMATION*)buffer.get();

for(;;) {
    //
    // use p...
    //
    if (p->NextEntryOffset == 0)
        break;

    //
    // move to the next process
    //
    p = (SYSTEM_PROCESS_INFORMATION*)((PBYTE)p + p->NextEntryOffset);
}
```

A full example is in the *ProcList* sample project.

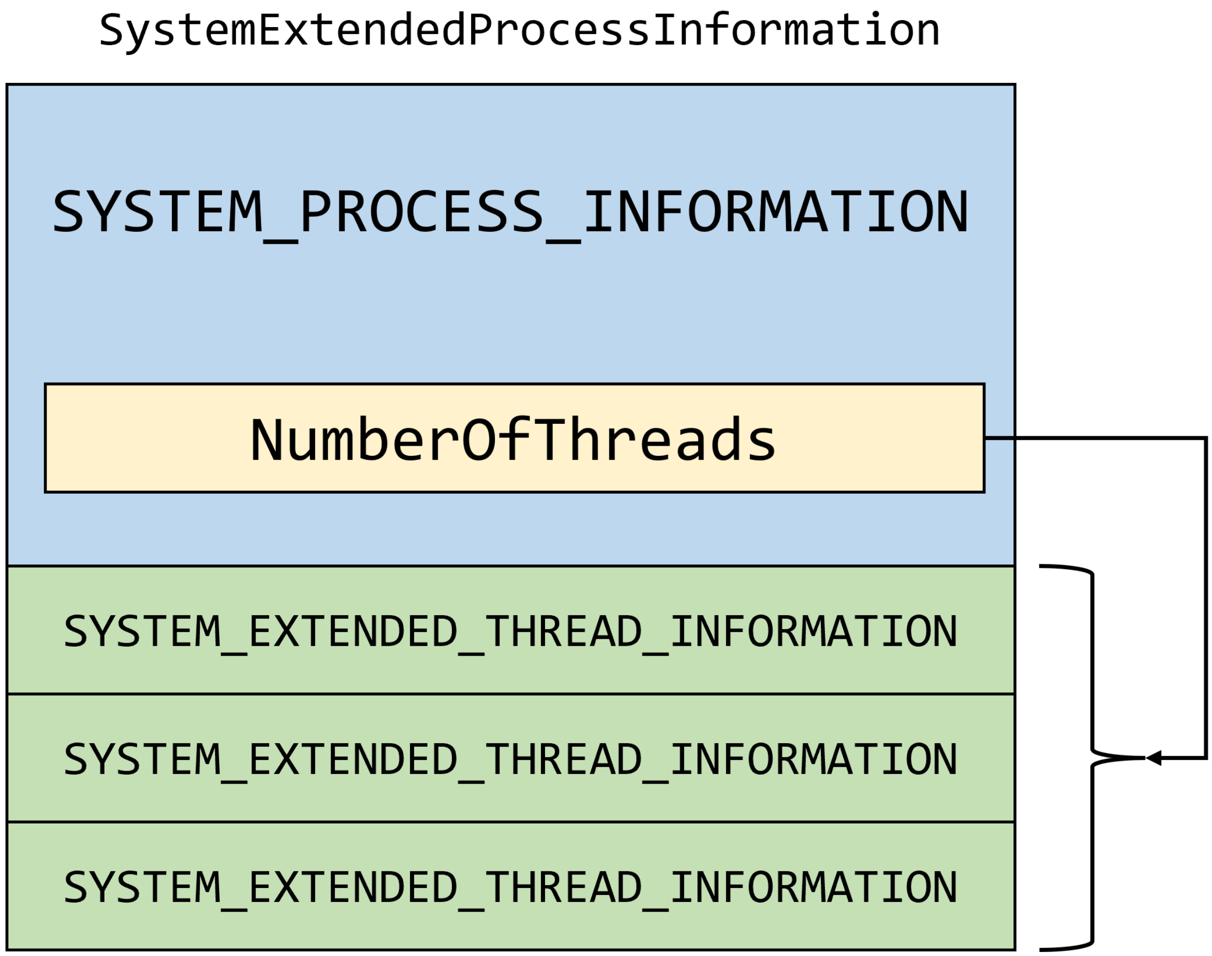


Figure 4-2: SystemExtendedProcessInformation details

Moving to the next process is accomplished by adding `NextEntryOffset` to the current `SYSTEM_-PROCESS_INFORMATION` pointer, just note the casting to a `BYTE` pointer so that the addition is done in units of bytes.

A list of threads for each process is provided in the `Threads` array. The basic structure follows:

```
typedef struct _SYSTEM_THREAD_INFORMATION {
    LARGE_INTEGER KernelTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER CreateTime;
    ULONG WaitTime;
    PVOID StartAddress;
    CLIENT_ID ClientId;
    KPRIORITY Priority;
    LONG BasePriority;
    ULONG ContextSwitches;
    KTHREAD_STATE ThreadState;
```

```
    KWAIT_REASON WaitReason;
} SYSTEM_THREAD_INFORMATION, *PSYSTEM_THREAD_INFORMATION;
```

Here is a description of the non-self explanatory members:

- `ClientId` is the thread and process IDs.
- `Priority` is the current (dynamic) priority of the thread, while `BasePriority` is the base priority. See chapter 6 for more on threads and priorities.
- `ThreadState` is the current thread state represented by the `KTHREAD_STATE` enumeration. Full explanation of thread states is deferred to chapter 6.
- `WaitReason` is the reason the thread is waiting (if it's in the `Waiting` state) of type `KWAIT_REASON`. Don't take any dependency on this value, as it's a hint only, and has no real meaning in practice.
- `StartAddress` is the thread's start address. For user-mode threads, it has the same value, pointing to the address of `RtlUserThreadStart` in *NtDll.Dll*. This is different from the start address provided to thread creation functions. That other address, referred to as "Win32 Address" is available in the extended structure, `SYSTEM_EXTENDED_THREAD_INFORMATION`, described shortly.

The thread description structures have fixed sizes, which means iteration is generally simpler. Given a pointer to a `SYSTEM_PROCESS_INFORMATION`, thread iteration could take this form (assuming the system info class is `SystemProcessInformation`):

```
void EnumThreads(SYSTEM_PROCESS_INFORMATION* p) {
    auto t = p->Threads;
    for (ULONG i = 0; i < p->NumberOfThreads; i++) {
        // do something with t
        t++;
    }
}
```

See the *ThreadList* sample for a full example.

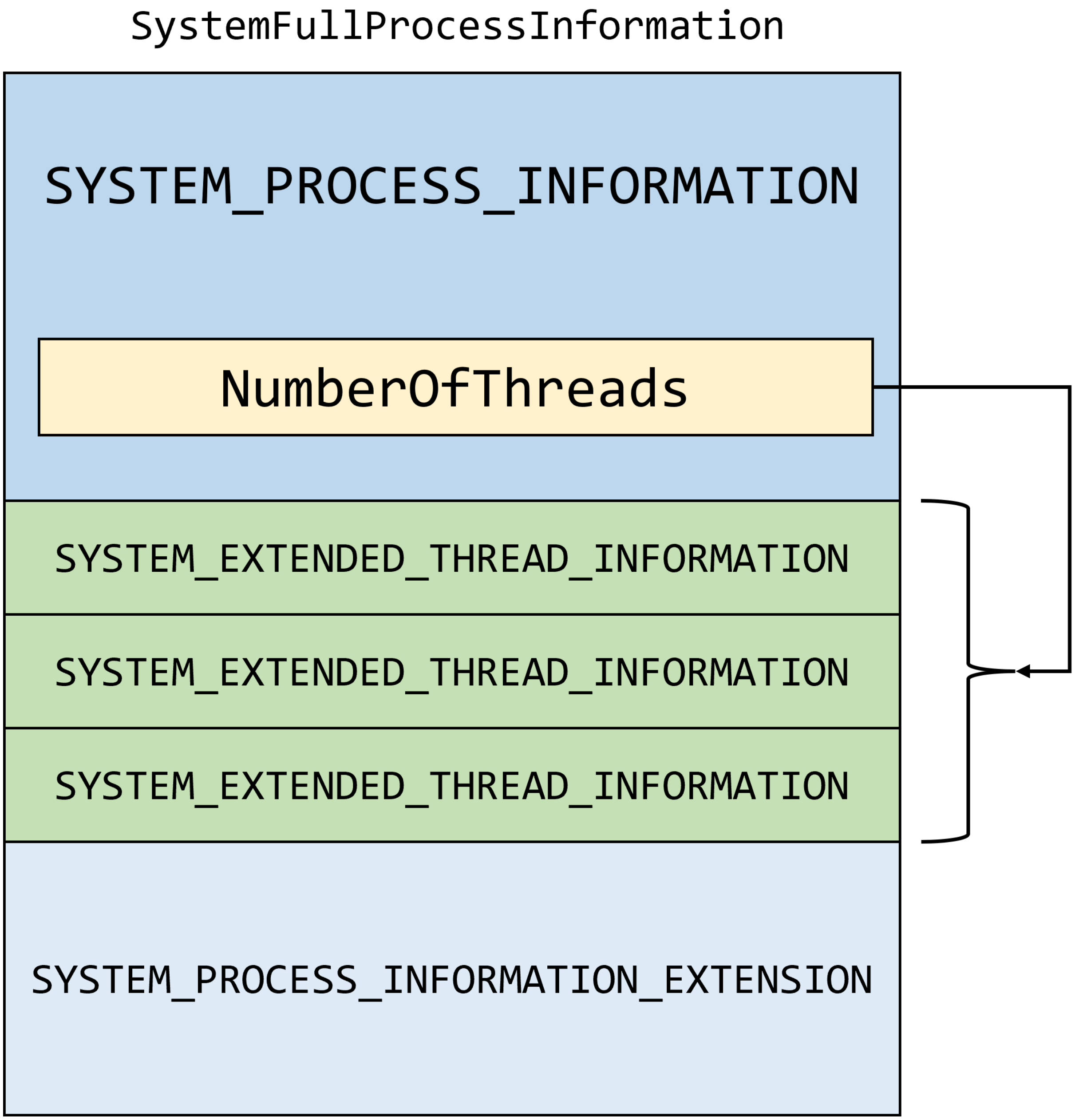


Figure 4-3: SystemFullProcessInformation details

The extension process information structure looks like so:

```
typedef enum _SYSTEM_PROCESS_CLASSIFICATION {
    SystemProcessClassificationNormal          = 0,
    SystemProcessClassificationSystem          = 1, // System
    SystemProcessClassificationSecureSystem    = 2, // Secure Kernel
    SystemProcessClassificationMemCompression  = 3, // Memory Compression
    SystemProcessClassificationRegistry        = 4, // Registry
    SystemProcessClassificationMaximum         = 5
} SYSTEM_PROCESS_CLASSIFICATION, *PSYSTEM_PROCESS_CLASSIFICATION;

typedef struct _PROCESS_DISK_COUNTERS {
    ULONGLONG BytesRead;
    ULONGLONG BytesWritten;
    ULONGLONG ReadOperationCount;
    ULONGLONG WriteOperationCount;
    ULONGLONG FlushOperationCount;
} PROCESS_DISK_COUNTERS, *PPROCESS_DISK_COUNTERS;

typedef struct _SYSTEM_PROCESS_INFORMATION_EXTENSION {
    PROCESS_DISK_COUNTERS DiskCounters;
    ULONGLONG ContextSwitches;
    union {
        ULONG Flags;
        struct {
            ULONG HasStrongId : 1;
            ULONG Classification : 4; // SYSTEM_PROCESS_CLASSIFICATION
            ULONG BackgroundActivityModerated : 1;
            ULONG Spare : 26;
        };
    };
    ULONG UserSidOffset;
    ULONG PackageFullNameOffset;          // since THRESHOLD
    PROCESS_ENERGY_VALUES EnergyValues; // since THRESHOLD
    ULONG AppIdOffset;                    // since THRESHOLD
    SIZE_T SharedCommitCharge;            // since THRESHOLD2
    ULONG JobObjectId;                    // since REDSTONE
    ULONG SpareUlong;                     // since REDSTONE
    ULONGLONG ProcessSequenceNumber;
} SYSTEM_PROCESS_INFORMATION_EXTENSION, *PSYSTEM_PROCESS_INFORMATION_EXTENSION;
```

Here is a description of the non-trivial members:

- `UserSidOffset` is an offset from the beginning of the `SYSTEM_PROCESS_INFORMATION_EXTENSION` structure to where the SID of the user running this process is stored in its binary format.

To convert to a string, you can call `RtlConvertSidToUnicodeString` like so:

```
std::wstring SidToString(PSID sid) {
    UNICODE_STRING str;
    if (NT_SUCCESS(RtlConvertSidToUnicodeString(&str, sid, TRUE))) {
        std::wstring result(str.Buffer, str.Length / sizeof(WCHAR));
        RtlFreeUnicodeString(&str);
        return result;
    }
    return L"";
}
```

Another option for converting a SID to a string is using the Windows API `ConvertSidToStringSid`.

- `PackageFullNameOffset` is an offset to a `NULL`-terminated string from the beginning of the `SYSTEM_PROCESS_INFORMATION_EXTENSION` object to a full package name of a UWP application this process is executing. If the process in question is not such a process, `PackageFullNameOffset` is zero. In addition, the `HasStrongId` member is `TRUE` if `PackageFullNameOffset` is non-zero and vice versa.
- `AppIdOffset` is an offset to a `NULL`-terminated string from the beginning of the `SYSTEM_PROCESS_INFORMATION_EXTENSION` object to an Application ID, used for some Shell features, such as Taskbar jump lists. It may be zero if the process is not associated with an Application ID.
- `JobObjectId` is the ID of a job that contains this process (if any). A job ID is used mainly with Windows Containers.
- `ProcessSequenceNumber` is a unique process identifier valid since Windows booted that is not reused (as opposed to the process ID that can be reused when the process object is destroyed).

To get a unique process ID on a system for any time, combine the process ID and the process creation time. Together, these are guaranteed to be unique at any time on the same machine.

Accessing the extended process information is a bit tricky if the `SYSTEM_PROCESS_INFORMATION` is declared as shown earlier. This is because the `Threads` first array index is included in the declaration. It would be somewhat easier to remove the `Threads` item from the declaration, and you can certainly do that. The declaration provided by the *phnt* header makes it easy to use `SystemProcessInformation`, but makes it clunkier to use `SystemFullProcessInformation`.

Nevertheless, this is possible as shown in the following code snippet:

```
// assume the NtQuerySystemInformation call succeeded
auto p = (SYSTEM_PROCESS_INFORMATION*)buffer.get();

for (;;) {
    auto px = (SYSTEM_PROCESS_INFORMATION_EXTENSION*)((PBYTE)p
        + sizeof(*p) - sizeof(SYSTEM_THREAD_INFORMATION)
        + p->NumberOfThreads * sizeof(SYSTEM_EXTENDED_THREAD_INFORMATION));
    //
    // use p and px as needed...
    //

    if (p->NextEntryOffset == 0)
        break;
    p = (SYSTEM_PROCESS_INFORMATION*)((PBYTE)p + p->NextEntryOffset);
}
```

> See the *ProcList* sample for a full example.

As can be seen from the code above, with `SystemFullProcessInformation` (and `SystemExtendedProcessInformation`), each thread information is now an `SYSTEM_EXTENDED_THREAD_INFORMATION`:

```
typedef struct _SYSTEM_EXTENDED_THREAD_INFORMATION {
    SYSTEM_THREAD_INFORMATION ThreadInfo;
    PVOID StackBase;
    PVOID StackLimit;
    PVOID Win32StartAddress;
    PTEB TebBase;
    ULONG_PTR Reserved2;
    ULONG_PTR Reserved3;
    ULONG_PTR Reserved4;
} SYSTEM_EXTENDED_THREAD_INFORMATION, *PSYSTEM_EXTENDED_THREAD_INFORMATION;
```

The basic structure is the first member, thus an "extended" structure, rather than "extension" in the process case which is distinct from the original structure.

Here is a quick rundown of the extra members:

- `StackBase` and `StackLimit` provide the beginning and end of the current thread stack. For user-mode threads, the addresses are in user-space, whereas for kernel-mode threads the addresses are in

kernel space. `StackBase` is greater than `StackLimit` on Intel/AMD platforms, as the stack grows down in memory.
- `Win32StartAddress` is the address provided to a user-mode thread creation function, as opposed to the `RtlUserThreadStart` address seen in the `SYSTEM_THREAD_INFORMATION` structure. For kernel threads, this value is zero.
- `TebBase` is the *Thread Environment Block* (TEB) address for this thread (zero for kernel threads). See chapter 6 for more on the TEB.

Extend the thread enumeration example to include the extended information.

4.3: Objects and Handles

Windows is an object-based operating system, which means that a lot of functionality is based around objects, such as processes, threads, sections, files, semaphores, and many more. Kernel objects are accessed in user-mode (and optionally kernel-mode) using handles, serving as an indirect "pointer" to kernel objects in system space, along with some handle-specific attributes.

> Full discussion of handles and objects is beyond the scope of this book.

Briefly, a handle entry has the following data associated with it:

- The address of the object it points to.
- The access mask used to obtain this handle, which specifies what "power" this handle has over the object.
- Three optional flags: inheritable handle ("I") - indicates if the handle should be inherited by a child process if such a process is created by the current process; protect from close ("P") - indicates the handle cannot be closed without first removing this flag. This serves as a precaution against accidentally closing the handle; audit on close ("A") - indicates closing of this handle should be audited in the event log.

4.3.1: Handles

`NtQuerySystemInformation` has information classes to retrieve the list of handles in the system: `SystemHandleInformation` (16) and `SystemExtendedHandleInformation` (64). The former should be considered deprecated, as the information it provides is more limited than the latter, but also because it uses `USHORT` as a data type for process IDs and handle values, which is too small. Process IDs and handle values are limited to 26 bits (and not just 16); therefore, we'll be using `SystemExtendedHandleInformation` only.

The structures associated with this information class are as follows:

```
typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX {
    PVOID Object;
    ULONG_PTR UniqueProcessId;
    ULONG_PTR HandleValue;
    ULONG GrantedAccess;
    USHORT CreatorBackTraceIndex;
    USHORT ObjectTypeIndex;
    ULONG HandleAttributes;
    ULONG Reserved;
} SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX, *PSYSTEM_HANDLE_TABLE_ENTRY_INFO_EX;

typedef struct _SYSTEM_HANDLE_INFORMATION_EX {
    ULONG_PTR NumberOfHandles;
    ULONG_PTR Reserved;
    SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX Handles[1];
} SYSTEM_HANDLE_INFORMATION_EX, *PSYSTEM_HANDLE_INFORMATION_EX;
```

`Object` is the object's address in kernel space. `UniqueProcessId` is the process ID handle table where this handle is part of, while `HandleValue` is the handle's numeric value. Handle values are multiples of 4, where the first valid handle is 4. `GrantedAccess` is the access mask this handle has towards the object. `ObjectTypeIndex` is the type index referring to the object type (e.g. process, thread, file, etc.). We'll see how to retrieve the type name based on that index in the next section. Finally, `HandleAttributes` consists of the optional handle flags mentioned earlier. The possible values are `OBJ_PROTECT_CLOSE` (1), `OBJ_INHERIT` (2), and `OBJ_AUDIT_OBJECT_CLOSE` (4). `CreateorBackTraceIndex` is always zero as far as I can tell.

The usual technique of calling `NtQuerySystemInformation` with zero size and getting back the required size does not work with `SystemExtendedHandleInformation`; more precisely, it will return the size of `SYSTEM_HANDLE_INFORMATION_EX` as the minimum required, which just provides the handle count.

One approach here is to allocate memory, check if it's large enough, and if not - double the allocation, and so on, until the allocation is large enough:

```
std::unique_ptr<BYTE[]> buffer;
ULONG size = 1 << 20;    // start with 1MB
NTSTATUS status;
do {
    buffer = std::make_unique<BYTE[]>(size);
    status = NtQuerySystemInformation(SystemExtendedHandleInformation,
        buffer.get(), size, nullptr);
    if (NT_SUCCESS(status))
        break;
    size *= 2;
} while (status == STATUS_INFO_LENGTH_MISMATCH);
```

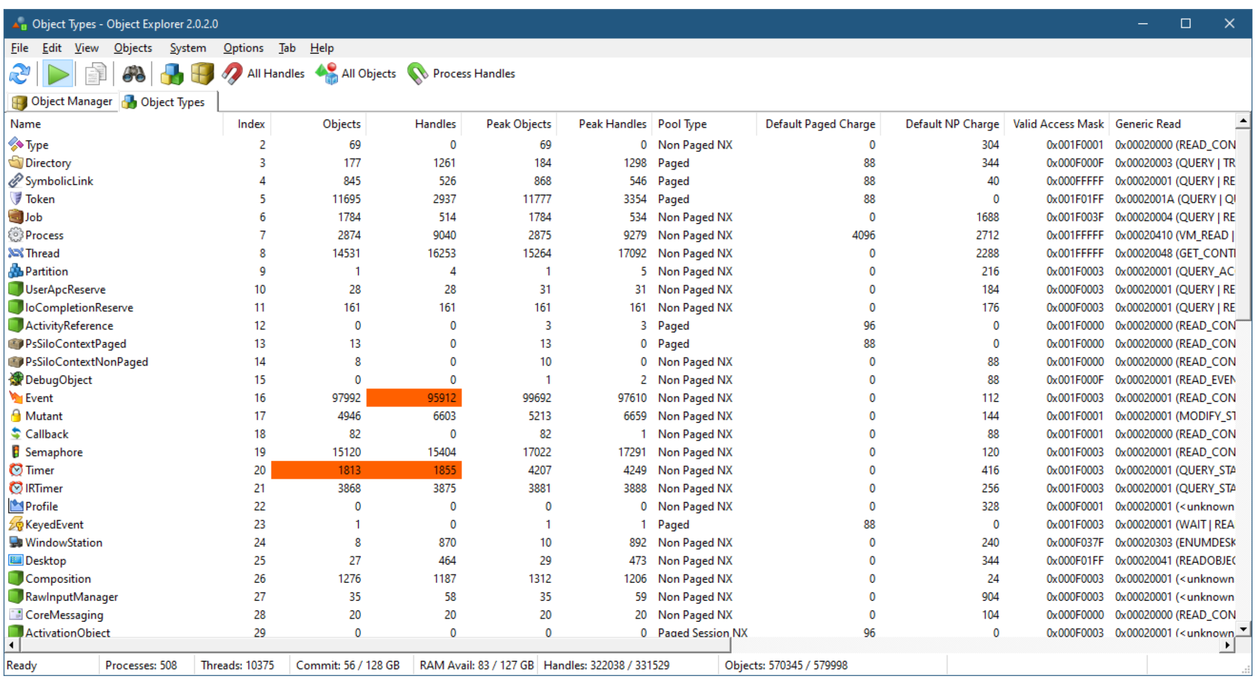
```
if(!NT_SUCCESS(status)) {
    // some error
}

auto p = (SYSTEM_HANDLE_INFORMATION_EX*)buffer.get();
for (ULONG_PTR i = 0; i < p->NumberOfHandles; i++) {
    auto& h = p->Handles[i];
    // do something with h
}
```

A full example is in the *Handles* sample project.

4.3.2: Object Types

Windows supports many object types; different Windows versions may support a slightly different set of object types. You can see details of object types in my tool *Object Explorer* (figure 4-4). How can we get to this information?



Name	Index	Objects	Handles	Peak Objects	Peak Handles	Pool Type	Default Paged Charge	Default NP Charge	Valid Access Mask	Generic Read
Type	2	69	0	69	0	Non Paged NX	0	304	0x001F0001	0x00020000 (READ_CON
Directory	3	177	1261	184	1298	Paged	88	344	0x000F000F	0x00020003 (QUERY | TR
SymbolicLink	4	845	526	868	546	Paged	88	40	0x000FFFFF	0x00020001 (QUERY | RE
Token	5	11695	2937	11777	3354	Paged	88	0	0x001F01FF	0x0002001A (QUERY | Q
Job	6	1784	514	1784	534	Non Paged NX	0	1688	0x001F003F	0x00020004 (QUERY | RE
Process	7	2874	9040	2875	9279	Non Paged NX	4096	2712	0x001FFFFF	0x00020410 (VM_READ |
Thread	8	14531	16253	15264	17092	Non Paged NX	0	2288	0x001FFFFF	0x00020048 (GET_CONTI
Partition	9	1	4	1	5	Non Paged NX	0	216	0x001F0003	0x00020001 (QUERY_AC
UserApcReserve	10	28	28	31	31	Non Paged NX	0	184	0x000F0003	0x00020001 (QUERY | RE
IoCompletionReserve	11	161	161	161	161	Non Paged NX	0	176	0x000F0003	0x00020001 (QUERY | RE
ActivityReference	12	0	0	3	3	Paged	96	0	0x001F0000	0x00020000 (READ_CON
PsSiloContextPaged	13	13	0	13	0	Paged	88	0	0x001F0000	0x00020000 (READ_CON
PsSiloContextNonPaged	14	8	0	10	0	Non Paged NX	0	88	0x001F0000	0x00020000 (READ_CON
DebugObject	15	0	0	1	2	Non Paged NX	0	88	0x001F000F	0x00020001 (READ_EVEN
Event	16	97992	95912	99692	97610	Non Paged NX	0	112	0x001F0003	0x00020001 (READ_CON
Mutant	17	4946	6603	5213	6659	Non Paged NX	0	144	0x001F0001	0x00020001 (MODIFY_ST
Callback	18	82	0	82	1	Non Paged NX	0	88	0x001F0001	0x00020000 (READ_CON
Semaphore	19	15120	15404	17022	17291	Non Paged NX	0	120	0x001F0003	0x00020001 (READ_CON
Timer	20	1813	1855	4207	4249	Non Paged NX	0	416	0x001F0003	0x00020001 (QUERY_STA
IRTimer	21	3868	3875	3881	3888	Non Paged NX	0	256	0x001F0003	0x00020001 (QUERY_STA
Profile	22	0	0	0	0	Non Paged NX	0	328	0x000F0001	0x00020001 (<unknown
KeyedEvent	23	1	0	1	1	Paged	88	0	0x001F0003	0x00020001 (WAIT | REA
WindowStation	24	8	870	10	892	Non Paged NX	0	240	0x000F037F	0x00020303 (ENUMDESK
Desktop	25	27	464	29	473	Non Paged NX	0	344	0x000F01FF	0x00020041 (READOBJEC
Composition	26	1276	1187	1312	1206	Non Paged NX	0	24	0x000F0003	0x00020001 (<unknown
RawInputManager	27	35	58	35	59	Non Paged NX	0	904	0x000F0003	0x00020001 (<unknown
CoreMessaging	28	20	20	20	20	Non Paged NX	0	104	0x000F0000	0x00020000 (READ_CON
ActivationObject	29	0	0	0	0	Paged Session NX	96	0	0x000F0003	0x00020001 (<unknown

Figure 4-4: Object Explorer showing object types

Object types information is available with `NtQueryObject`:

```
typedef enum _OBJECT_INFORMATION_CLASS {
    ObjectBasicInformation,      // OBJECT_BASIC_INFORMATION
    ObjectNameInformation,       // OBJECT_NAME_INFORMATION
    ObjectTypeInformation,       // OBJECT_TYPE_INFORMATION
    ObjectTypesInformation,      // OBJECT_TYPES_INFORMATION
    ObjectHandleFlagInformation, // OBJECT_HANDLE_FLAG_INFORMATION
    ObjectSessionInformation,
    ObjectSessionObjectInformation,
} OBJECT_INFORMATION_CLASS;

NTSTATUS NtQueryObject(
    _In_opt_ HANDLE Handle,
    _In_ OBJECT_INFORMATION_CLASS ObjectInformationClass,
    _Out_writes_bytes_opt_(ObjectInformationLength) PVOID ObjectInformation,
    _In_ ULONG ObjectInformationLength,
    _Out_opt_ PULONG ReturnLength);
```

This API normally requires a handle to an object in order to provide the requested detail. However, to get information about all existing object types, a `NULL` handle can be provided along with `ObjectTypesInformation`. This case also requires "guessing" the size of the required buffer, rather than getting the needed size. The returned information size is fixed, however, as it's about types and not any specific object.

The returned pointer is of type `OBJECT_TYPES_INFORMATION`:

```
typedef struct _OBJECT_TYPES_INFORMATION {
    ULONG NumberOfTypes;
} OBJECT_TYPES_INFORMATION, *POBJECT_TYPES_INFORMATION;
```

The following shows how to get object types information assuming some fixed allocation that is large enough (error handling omitted):

```
ULONG size = 1 << 14;
auto buffer = std::make_unique<BYTE[]>(size);
NtQueryObject(nullptr, ObjectTypesInformation, buffer.get(), size, nullptr);

auto p = (OBJECT_TYPES_INFORMATION*)buffer.get();
auto type = (OBJECT_TYPE_INFORMATION*)((PBYTE)p + sizeof(ULONG_PTR));
for (ULONG i = 0; i < p->NumberOfTypes; i++) {
    // do something with type...

    // move to the next type
    auto offset = sizeof(OBJECT_TYPE_INFORMATION) + type->TypeName.MaximumLength;
```

```
    if(offset % sizeof(ULONG_PTR))
        offset += sizeof(ULONG_PTR) - ((ULONG_PTR)type + offset) % sizeof(ULONG_PTR);

    type = (OBJECT_TYPE_INFORMATION*)((PBYTE)type + offset);
}
```

The code is complicated by the fact that every type object starts on an aligned address (4 bytes in 32-bit processes, or 8 bytes in 64-bit processes). The offset to the next type must take into account the type's name and add padding if the final address is not a multiple of the alignment size.

For every type we get the following information:

```
typedef struct _OBJECT_TYPE_INFORMATION {
    UNICODE_STRING TypeName;
    ULONG TotalNumberOfObjects;
    ULONG TotalNumberOfHandles;
    ULONG TotalPagedPoolUsage;
    ULONG TotalNonPagedPoolUsage;
    ULONG TotalNamePoolUsage;
    ULONG TotalHandleTableUsage;
    ULONG HighWaterNumberOfObjects;
    ULONG HighWaterNumberOfHandles;
    ULONG HighWaterPagedPoolUsage;
    ULONG HighWaterNonPagedPoolUsage;
    ULONG HighWaterNamePoolUsage;
    ULONG HighWaterHandleTableUsage;
    ULONG InvalidAttributes;
    GENERIC_MAPPING GenericMapping;
    ULONG ValidAccessMask;
    BOOLEAN SecurityRequired;
    BOOLEAN MaintainHandleCount;
    UCHAR TypeIndex;
    CHAR ReservedByte;
    ULONG PoolType;
    ULONG DefaultPagedPoolCharge;
    ULONG DefaultNonPagedPoolCharge;
} OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;
```

The following members are used (all others are always zero):

- `TypeName` is the type's name, e.g., "Process", "Event", "Desktop", etc.
- `TotalNumberOfObjects` is the current number of objects of this type.
- `TotalNumberOfHandles` is the current number of handles to objects of this type.

- `HighWatermarkNumberOfObjects` is the peak number of objects of this type since the most recent boot.
- `HighWatermarkNumberOfHandles` is the peak number of handles to objects of this type since the most recent boot.
- `InvalidAttributes` is the set of attribute flags (`OBJ_xxx`) that are not valid for handles of this type.
- `GenericMapping` is the mappings of standard right (`GENERIC_READ`, `GENERIC_WRITE`, `GENERIC_EXECUTE` and `GENERIC_ALL`) to specific rights for this type of object. See the `GENERIC_MAPPING` structure SDK documentation.
- `ValidAccessMask` is the set of valid access bits for this type.
- `SecurityRequired` indicates whether objects of this type must be created with a Security Descriptor.
- `MaintainHandleCount` indicates whether a list of handles to objects of this type is maintained within the type object. This flag is set mostly for UI-related kernel objects (Window Stations, Desktops, Composition, and a few others).
- `TypeIndex` is the index of this type. This allows correlation with the `ObjectTypeIndex` member in `SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX`.
- `PoolType` is the type of pool that should be used to allocate objects of this type. The following enumeration (not present in *phnt*) provides the values:

```
enum class PoolType {
    PagedPool = 1,
    NonPagedPool = 0,
    NonPagedPoolNx = 0x200,
    NonPagedPoolSessionNx = NonPagedPoolNx + 32,
    PagedPoolSessionNx = NonPagedPoolNx + 33
};
```

- `DefaultPagedPoolCharge` and `DefaultNonPagedPoolCharge` is the default sizes of paged/non-paged kernel memory required when allocating an object of this type.

The *ObjectTypes* sample provides a full example of showing the set of types with some of their members. The bulk of the code is as follows:

```
const char* PoolTypeToString(ULONG poolType) {
    switch ((PoolType)poolType) {
        case PoolType::PagedPool: return "Paged";
        case PoolType::NonPagedPool: return "Non Paged";
        case PoolType::NonPagedPoolNx: return "Non Paged NX";
        case PoolType::NonPagedPoolSessionNx: return "Session Non Paged NX";
        case PoolType::PagedPoolSessionNx: return "Session Paged NX";
    }
    return "";
}
```

```
void DisplayType(OBJECT_TYPE_INFORMATION* type) {
    printf("%2d %-34wZ H: %6u O: %6u PH: %6u PO: %6u Pool: %s\n",
        type->TypeIndex, &type->TypeName,
        type->TotalNumberOfHandles, type->TotalNumberOfObjects,
        type->HighWaterNumberOfHandles, type->HighWaterNumberOfObjects,
        PoolTypeToString(type->PoolType));
}

int main() {
    ULONG size = 1 << 14;
    auto buffer = std::make_unique<BYTE[]>(size);
    if (!NT_SUCCESS(NtQueryObject(nullptr, ObjectTypesInformation,
        buffer.get(), size, nullptr))) {
        return 1;
    }
    auto p = (OBJECT_TYPES_INFORMATION*)buffer.get();
    auto type = (OBJECT_TYPE_INFORMATION*)((PBYTE)p + sizeof(ULONG_PTR));
    long long handles = 0, objects = 0;
    for (ULONG i = 0; i < p->NumberOfTypes; i++) {
        DisplayType(type);
        handles += type->TotalNumberOfHandles;
        objects += type->TotalNumberOfObjects;

        auto offset = sizeof(OBJECT_TYPE_INFORMATION) + type->TypeName.MaximumLength;
        if (offset % sizeof(ULONG_PTR))
            offset += sizeof(ULONG_PTR) -
            ((ULONG_PTR)type + offset) % sizeof(ULONG_PTR);
        type = (OBJECT_TYPE_INFORMATION*)((PBYTE)type + offset);
    }
    printf("Types: %u Handles: %llu Objects: %llu\n",
        p->NumberOfTypes, handles, objects);
    return 0;
}
```

Here is an example output (trimmed) on a Windows 10 machine:

```
 2 Type                       H:     0 O:     69 PH:     0 PO:    69 Non Paged NX
 3 Directory                  H:  1234 O:    179 PH:  1325 PO:   197 Paged
 4 SymbolicLink               H:   529 O:    847 PH:   546 PO:   868 Paged
 5 Token                      H:  2969 O:  12009 PH:  3354 PO: 12506 Paged
 6 Job                        H:   495 O:   2083 PH:   534 PO:  2095 Non Paged NX
 7 Process                    H:  9324 O:   3372 PH:  9888 PO:  3378 Non Paged NX
 8 Thread                     H: 16679 O:  15504 PH: 17467 PO: 15995 Non Paged NX
...
68 CrossVmEvent               H:     0 O:      0 PH:     0 PO:     0 Non Paged NX
69 CrossVmMutant              H:     0 O:      0 PH:     0 PO:     0 Non Paged NX
70 VRegConfigurationContext  H:     0 O:     12 PH:     0 PO:    13 Paged
Types: 69 Handles: 317539 Objects: 629165
```

Check this on a Windows 11 machine. You'll discover some new and removed object types.

4.3.3: Object Names

Some kernel objects have string-based names associated with them. Using `QuerySystemInformation` with `SystemExtendedHandleInformation` does not provide the object's name each handle points to (if any). One reason is that it may be expensive and not strictly necessary. Furthermore, multiple handles to the same object might allocate the same name multiple times, or else it would be costly to maintain a list of objects whose names was already retrieved.

To get an object's name (pointed to with a handle), the native API offers `NtQueryObject` with the `ObjectNameInformation` information class. One issue that may be evident when enumerating handles is that handle values returned from the enumeration only have meaning in their respective process. This means that to get an object's name we first have to duplicate the handle into the current process before using `NtQueryObject`.

Here is an initial version to return an object's name (if any) as a `std::wstring` given a process ID and a handle that is valid in that process. The first step is to open a handle to the provided process for handling duplication purposes:

```
NTSTATUS GetObjectName(HANDLE h, DWORD pid, std::wstring& name) {
    OBJECT_ATTRIBUTES procAttr = RTL_CONSTANT_OBJECT_ATTRIBUTES(nullptr, 0);
    CLIENT_ID cid{};
    cid.UniqueProcess = ULongToHandle(pid);
    HANDLE hProcess;
    auto status = NtOpenProcess(&hProcess, PROCESS_DUP_HANDLE, &procAttr, &cid);
    if (!NT_SUCCESS(status))
        return status;
```

The next step is to invoke `NtDuplicateObject`, which (despite the name) is the equivalent of the Windows API `DuplicateHandle`. Full discussion of `NtDuplicateObject` is saved for chapter 7:

```
HANDLE hDup;
status = NtDuplicateObject(hProcess, h, NtCurrentProcess(),
    &hDup, READ_CONTROL, 0, 0);
```

We duplicate the handle to our process, so that `hDup` now has meaning. Assuming the call succeeds, we can continue like so:

```
        BYTE buffer[1024];  // hopefully large enough :)
        auto sname = (UNICODE_STRING*)buffer;
        status = NtQueryObject(hDup, ObjectNameInformation,
            sname, sizeof(buffer), nullptr);
        if (NT_SUCCESS(status))
            name.assign(sname->Buffer, sname->Length / sizeof(WCHAR));
        NtClose(hDup);
    }
    NtClose(hProcess);
    return status;
}
```

This works for most object types, except for many file objects - the call to `NtQueryObject` hangs. This is because many file objects cannot be accessed for name query without grabbing locks that are usually maintained by the file system in question.

The only reasonable way I have found to skip “problematic” files is to perform the query on a separate thread, and if it blocks for too long, terminate it, and move on.

The code becomes more complicated, but cannot be avoided. The following shows the full `GetObjectName` implementation, where a Boolean flag indicates the handle in question is for a file. Discussion of `NtCreateThreadEx`, `NtWaitForSingleObject`, `NtTerminateThread`, and `NtQueryInformationThread` is deferred to chapters 6 (“Threads”) and 7 (“Objects and Handles”).

```
NTSTATUS GetObjectName(HANDLE h, DWORD pid, std::wstring& name, bool file) {
    OBJECT_ATTRIBUTES procAttr = RTL_CONSTANT_OBJECT_ATTRIBUTES(nullptr, 0);
    CLIENT_ID cid{};
    cid.UniqueProcess = ULongToHandle(pid);
    HANDLE hProcess;
    auto status = NtOpenProcess(&hProcess, PROCESS_DUP_HANDLE, &procAttr, &cid);
    if (!NT_SUCCESS(status))
        return status;

    HANDLE hDup;
    status = NtDuplicateObject(hProcess, h, NtCurrentProcess(),
        &hDup, READ_CONTROL, 0, 0);
    if (NT_SUCCESS(status)) {
        static BYTE buffer[1024];   // hopefully large enough :)
        auto sname = (UNICODE_STRING*)buffer;
        if (file) {
            //
            // special case for files
            // get name on a separate thread
            // kill thread if not available after some waiting
            //
            HANDLE hThread;
            //
            // create the thread
            //
            status = NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS,
                &procAttr, NtCurrentProcess(),
                (PTHREAD_START_ROUTINE)[](auto param) -> DWORD {
                    auto h = (HANDLE)param;
                    auto status = NtQueryObject(h, ObjectNameInformation,
                        buffer, sizeof(buffer), nullptr);

                    //
                    // status becomes the exit code of the thread
                    //
                    return status;
                }, hDup, 0, 0, 0, 0, nullptr);
            if (NT_SUCCESS(status)) {
                LARGE_INTEGER interval;
                interval.QuadPart = -50 * 10000;    // 50 msec
                status = NtWaitForSingleObject(hThread, FALSE, &interval);
                if (status == STATUS_TIMEOUT) {
```

```
                NtTerminateThread(hThread, 1);
            }
            else {
                //
                // read the thread's exit code
                //
                THREAD_BASIC_INFORMATION tbi;
                NtQueryInformationThread(hThread, ThreadBasicInformation,
                    &tbi, sizeof(tbi), nullptr);
                status = tbi.ExitStatus;
            }
            NtClose(hThread);
        }
    }
    else {
        //
        // not a File object, perform simple query
        //
        status = NtQueryObject(hDup, ObjectNameInformation,
            sname, sizeof(buffer), nullptr);
    }
    //
    // assign the result to std::wstring
    //
    if (NT_SUCCESS(status))
        name.assign(sname->Buffer, sname->Length / sizeof(WCHAR));
    NtClose(hDup);
}
NtClose(hProcess);
return status;
}
```

The *Handles* sample contains the full code. More information on objects and handles is provided in chapter 7.

4.4: The `KUSER_SHARED_DATA` Structure

The `KUSER_SHARED_DATA` data structure is partly documented by Microsoft, and provides global system information. The structure is read-only from user-mode, and is mapped to the same virtual address in every process (`0x7ffe0000`). Thus, accessing the data is easy:

```
auto data = (KUSER_SHARED_DATA*)0x7ffe0000;
// access data...
```

Some of the details in that structure are provided by `NtQuerySystemInformation`. Regardless, it can be accessed directly. For example, the Windows version can be displayed like so:

```
auto data = (KUSER_SHARED_DATA*)0x7ffe0000;
printf("Version: %d.%d.%d\n",
    data->NtMajorVersion, data->NtMinorVersion, data->NtBuildNumber);
```

The *phnt* headers provide the macro `USER_SHARED_DATA` to access the structure without casting.

4.5: Summary

A lot of system information is available with just one function: `NtQuerySystemInformation`. I recommend an exploration mindset when looking at the various information classes, including trial and error. That said, this chapter is likely to be expanded in future versions of this book.

Chapter 5: Processes

Processes are probably the most recognizable of all kernel object types. In this chapter, we'll examine the native APIs related to process creation, enumeration, and manipulation.

In this chapter:

- **Creating Processes**
- **Process Information**
- **The Process Environment Block**
- **Suspending and Resuming Processes**
- **Enumerating Processes (Take 2)**
- **Jobs**

5.1: Creating Processes

The Windows API provides several functions to create processes, such as `CreateProcess` and `CreateProcessAsUser`. Although these APIs eventually invoke native APIs, they do a lot of work, making it difficult to reproduce the same behavior when creating processes that belong to the Windows subsystem (that is, non-native applications).

The native API provides the `RtlCreateUserProcess(Ex)` functions, with supporting structures and functions. We have used `RtlCreateUserProcess` in chapter 3 to run native applications, because `CreateProcess` is not implemented to handle this case. In this section, I'll describe some aspects of the native APIs, but I recommend they are only used for creating processes for native applications.

> There have been several attempts in the Infosec community to utilize `RtlCreateUserProcess` and/or `NtCreateUserProcess` to allow running Windows subsystem applications, with varying degrees of success. Part of the problem is the need to communicate with the Windows Subsystem process (*csrss.exe*) to notify it of the new process and thread. This turns out to be fragile, as different Windows versions may have somewhat different expectations. This may still be an interesting research project, but is out of scope for this book.

Here are the `RtlCreateUserProcess(Ex)` functions with the returned structure that contains information on the created process if successful:

```
typedef struct _RTL_USER_PROCESS_INFORMATION {
    ULONG Length;
    HANDLE ProcessHandle;
    HANDLE ThreadHandle;
    CLIENT_ID ClientId;
    SECTION_IMAGE_INFORMATION ImageInformation;
} RTL_USER_PROCESS_INFORMATION, *PRTL_USER_PROCESS_INFORMATION;

NTSTATUS RtlCreateUserProcess(
    _In_ PUNICODE_STRING NtImagePathName,
    _In_ ULONG AttributesDeprecated,
    _In_ PRTL_USER_PROCESS_PARAMETERS ProcessParameters,
    _In_opt_ PSECURITY_DESCRIPTOR ProcessSecurityDescriptor,
    _In_opt_ PSECURITY_DESCRIPTOR ThreadSecurityDescriptor,
    _In_opt_ HANDLE ParentProcess,
    _In_ BOOLEAN InheritHandles,
    _In_opt_ HANDLE DebugPort,
    _In_opt_ HANDLE TokenHandle, // used to be ExceptionPort
    _Out_ PRTL_USER_PROCESS_INFORMATION ProcessInformation);

NTSTATUS RtlCreateUserProcessEx(
    _In_ PUNICODE_STRING NtImagePathName,
    _In_ PRTL_USER_PROCESS_PARAMETERS ProcessParameters,
    _In_ BOOLEAN InheritHandles,
     _In_opt_ PRTL_USER_PROCESS_EXTENDED_PARAMETERS ExtendedParameters,
    _Out_ PRTL_USER_PROCESS_INFORMATION ProcessInformation);
```

It may be surprising that the extended function has less parameters than the original function. In this case, some of the less common parameters from the original function are bundled in an extended parameters structure defined like so:

```
typedef struct _RTL_USER_PROCESS_EXTENDED_PARAMETERS {
    USHORT Version;     // currently must be set to 1
    USHORT NodeNumber;
    PSECURITY_DESCRIPTOR ProcessSecurityDescriptor;
    PSECURITY_DESCRIPTOR ThreadSecurityDescriptor;
    HANDLE ParentProcess;   // for inheritance
    HANDLE DebugPort;
    HANDLE TokenHandle;     // requires SeAssignPrimaryTokenPrivilege
    HANDLE JobHandle;
} RTL_USER_PROCESS_EXTENDED_PARAMETERS, *PRTL_USER_PROCESS_EXTENDED_PARAMETERS;
```

At the time of this writing, the above structure is not defined by the *phnt* headers.

The non-optional process parameters structure is a big one, defined likes so:

```
typedef struct _RTL_USER_PROCESS_PARAMETERS {
    ULONG MaximumLength;
    ULONG Length;

    ULONG Flags;
    ULONG DebugFlags;

    HANDLE ConsoleHandle;
    ULONG  ConsoleFlags;
    HANDLE StandardInput;   // STARTUPINFO.hStdInput
    HANDLE StandardOutput;  // STARTUPINFO.hStdOutput
    HANDLE StandardError;   // STARTUPINFO.hStdError

    CURDIR CurrentDirectory;
    UNICODE_STRING DllPath;
    UNICODE_STRING ImagePathName;
    UNICODE_STRING CommandLine;
    PVOID Environment;

    ULONG StartingX;    // STARTUPINFO.dwX
    ULONG StartingY;    // STARTUPINFO.dwY
    ULONG CountX;       // STARTUPINFO.dwXSize
    ULONG CountY;       // STARTUPINFO.dwYSize
    ULONG CountCharsX;  // STARTUPINFO.dwXCountChars
    ULONG CountCharsY;  // STARTUPINFO.dwYCountChars
    ULONG FillAttribute; // STARTUPINFO.dwFillAttribute

    ULONG WindowFlags;          // STARTUPINFO.dwFlags
    ULONG ShowWindowFlags;      // STARTUPINFO.wShowWindow
    UNICODE_STRING WindowTitle; // STARTUPINFO.lpTitle
    UNICODE_STRING DesktopInfo;
    UNICODE_STRING ShellInfo;
    UNICODE_STRING RuntimeData;
    RTL_DRIVE_LETTER_CURDIR CurrentDirectories[RTL_MAX_DRIVE_LETTERS];

    ULONG_PTR EnvironmentSize;
```

```
    ULONG_PTR EnvironmentVersion;
    PVOID PackageDependencyData;
    ULONG ProcessGroupId;
    ULONG LoaderThreads;

    UNICODE_STRING RedirectionDllName; // REDSTONE4
    UNICODE_STRING HeapPartitionName; // 19H1
    ULONG_PTR DefaultThreadpoolCpuSetMasks;
    ULONG DefaultThreadpoolCpuSetMaskCount;
} RTL_USER_PROCESS_PARAMETERS, *PRTL_USER_PROCESS_PARAMETERS;
```

Some of the structure members correspond directly to the `STARTUPINFO` structure defined in the Windows API and used with `CreateProcess`. Other members are more obscure. After a process is created, this structure can be found in the `PEB.ProcessParamaters` member.

This is a variable-length structure, as there are some strings inside. The native API provides two functions to create and partly initialize an `RTL_USER_PROCESS_PARAMETERS`, and another one to free the structure:

```
NTSTATUS RtlCreateProcessParameters(
    _Out_ PRTL_USER_PROCESS_PARAMETERS *pProcessParameters,
    _In_ PUNICODE_STRING ImagePathName,
    _In_opt_ PUNICODE_STRING DllPath,
    _In_opt_ PUNICODE_STRING CurrentDirectory,
    _In_opt_ PUNICODE_STRING CommandLine,
    _In_opt_ PVOID Environment,
    _In_opt_ PUNICODE_STRING WindowTitle,
    _In_opt_ PUNICODE_STRING DesktopInfo,
    _In_opt_ PUNICODE_STRING ShellInfo,
    _In_opt_ PUNICODE_STRING RuntimeData);
NTSTATUS RtlCreateProcessParametersEx(
    _Out_ PRTL_USER_PROCESS_PARAMETERS *pProcessParameters,
    _In_ PUNICODE_STRING ImagePathName,
    _In_opt_ PUNICODE_STRING DllPath,
    _In_opt_ PUNICODE_STRING CurrentDirectory,
    _In_opt_ PUNICODE_STRING CommandLine,
    _In_opt_ PVOID Environment,
    _In_opt_ PUNICODE_STRING WindowTitle,
    _In_opt_ PUNICODE_STRING DesktopInfo,
    _In_opt_ PUNICODE_STRING ShellInfo,
    _In_opt_ PUNICODE_STRING RuntimeData,
    _In_     ULONG Flags);
```

```
NTSTATUS RtlDestroyProcessParameters(
    _In_ _Post_invalid_ PRTL_USER_PROCESS_PARAMETERS ProcessParameters);
```

The two creation functions are practically identical except for the extra `Flags` parameter. The most common flag is `RTL_CREATE_PROC_PARAMS_NORMALIZED` (1) that indicates the structure should be initialized as "normalized". Normalized means that the string pointers within are true pointers and can be readily used in subsequent calls that require this structure. The downside is that copying the structure for later use is not possible, as the pointers are absolute. Creating the structure as de-normalized stores offsets only in the various `UNICODE_STRING.Buffer` members so that copying the structure can be done by a simple `memcpy`-like call. The non-`Ex` function initializes the structure as de-normalized. `RtlCreateUserProcess` normalizes the structure if needed before using it.

The native API provides two functions to perform the normalization or de-normalization:

```
PRTL_USER_PROCESS_PARAMETERS RtlNormalizeProcessParams(
    _Inout_ PRTL_USER_PROCESS_PARAMETERS ProcessParameters);
PRTL_USER_PROCESS_PARAMETERS RtlDeNormalizeProcessParams(
    _Inout_ PRTL_USER_PROCESS_PARAMETERS ProcessParameters);
```

The `Flags` member of `RTL_USER_PROCESS_PARAMETERS` keeps track of the normalization state so that the operation is not performed if not required.

Here is more information on members of `RTL_USER_PROCESS_PARAMETERS`:

- `ShellInfo` - arbitrary string or buffer that is passed as is to the new process. Some console application use this internally.
- `DesktopInfo` is an optional string that points to another Window Station and/or Desktop in the format "winstaname\desktopname". This is the same parameter passed in `STARTUPINFO.lpDesktop`.
- `RuntimeData` - arbitrary strings or buffers that are passed as is to the new process. Remember that it does not have to be a string, since `UNICODE_STRING` has a length, meaning the buffer can point to any arbitrary data.
- `DllPath` - optional NT path that serves as another directory to look for DLLs by the loader for the new process.
- `CommandLine` - optional command line arguments to pass to the new process.
- `Environment` - optional environment block created by calling `RtlCreateEnvironment`. If `NULL`, copies the environment from the parent process. See the the documentation for the `lpEnvironment` argument to `CreateProcess`.
- `RedirectionDllName` - optional NT path DLL that instructs the loader to perform function redirection. If specified, the DLL must export a global variable named `__RedirectionInformation__` that is of type `REDIRECTION_DESCRIPTOR` defined like so:

```
typedef struct _REDIRECTION_FUNCTION_DESCRIPTOR {
    PCSTR DllName;
    PCSTR FunctionName;
    PVOID RedirectionTarget;
} REDIRECTION_FUNCTION_DESCRIPTOR, *PREDIRECTION_FUNCTION_DESCRIPTOR;
typedef const REDIRECTION_FUNCTION_DESCRIPTOR *PCREDIRECTION_FUNCTION_DESCRIPTOR;

typedef struct _REDIRECTION_DESCRIPTOR {
    ULONG Version;
    ULONG FunctionCount;
    PCREDIRECTION_FUNCTION_DESCRIPTOR Redirections;
} REDIRECTION_DESCRIPTOR, *PREDIRECTION_DESCRIPTOR;
```

Two optional functions can be further exported from that DLL named `__ShouldApplyRedirection__`- and `__ShouldApplyRedirectionToFunction__` whose expected prototypes are the following:

```
typedef BOOLEAN (*RedirectCbFunc)(PCWSTR);  // name
typedef BOOLEAN (*RedirectByFunctionCbFunc)(PWSTR, ULONG);  // name, index
```

5.2: Process Information

The first step in getting or setting process information is to obtain a handle to the requested process with an appropriate access mask by calling `NtOpenProcess`:

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

`DesiredAccess` is the access mask required, either one of the generic ones (e.g. `GENERIC_READ`) or better yet, a combination of specific ones for processes (e.g. `PROCESS_TERMINATE`, `PROCESS_VM_OPERATION`) - all which are officially documented.

`ObjectAttributes` does not need a name (as processes don't have names), so can be initialized very simply by using `RTL_CONST_OBJECT_ATTRIBUTES` like so:

```
OBJECT_ATTRIBUTES processAttributes = RTL_CONSTANT_OBJECT_ATTRIBUTES(nullptr, 0);
```

The most important piece is `ClientId`, where the process ID is specified. Note that the thread ID must be zero, or the call fails. Here is an example that returns a handle to a given process ID and access mask (if successful):

```
HANDLE OpenProcess(ULONG pid, ACCESS_MASK access) {
    OBJECT_ATTRIBUTES processAttributes = RTL_CONSTANT_OBJECT_ATTRIBUTES(nullptr, 0);
    CLIENT_ID cid {};
    cid.UniqueProcess = ULongToHandle(pid);
    HANDLE hProcess = nullptr;
    NtOpenProcess(&hProcess, access, &processAttributes, &cid);
    return hProcess;
}
```

With a process handle in hand, the primary APIs to get and set information are the following:

```
NTSTATUS NtQueryInformationProcess(
    _In_ HANDLE ProcessHandle,
    _In_ PROCESSINFOCLASS ProcessInformationClass,
    _Out_writes_bytes_(ProcessInformationLength) PVOID ProcessInformation,
    _In_ ULONG ProcessInformationLength,
    _Out_opt_ PULONG ReturnLength);
NTSTATUS NtSetInformationProcess(
    _In_ HANDLE ProcessHandle,
    _In_ PROCESSINFOCLASS ProcessInformationClass,
    _In_reads_bytes_(ProcessInformationLength) PVOID ProcessInformation,
    _In_ ULONG ProcessInformationLength);
```

The pattern should look familiar - `NtQuerySystemInformation` and `NtSetSystemInformation` use the same pattern.

The `PROCESSINFOCLASS` enumeration provides the various pieces of data that can be retrieved and modified. It's not shown here for brevity, but the following subsections list some of the more useful ones.

> Unless otherwise specified, the required access mask for the process handle is `PROCESS_QUERY_LIMITED_INFORMATION` or `PROCESS_QUERY_INFORMATION`.

5.2.1: `ProcessBasicInformation` (0, Query)

This information class provides some basic details about the process. One of the following structures are accepted for this query:

```
typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS ExitStatus;
    PPEB PebBaseAddress;
    ULONG_PTR AffinityMask;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    HANDLE InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION, *PPROCESS_BASIC_INFORMATION;

typedef struct _PROCESS_EXTENDED_BASIC_INFORMATION {
    SIZE_T Size;
    PROCESS_BASIC_INFORMATION BasicInfo;
    union {
        ULONG Flags;
        struct {
            ULONG IsProtectedProcess : 1;
            ULONG IsWow64Process : 1;
            ULONG IsProcessDeleting : 1;
            ULONG IsCrossSessionCreate : 1;
            ULONG IsFrozen : 1;
            ULONG IsBackground : 1;
            ULONG IsStronglyNamed : 1;
            ULONG IsSecureProcess : 1;
            ULONG IsSubsystemProcess : 1;
            ULONG SpareBits : 23;
        };
    };
} PROCESS_EXTENDED_BASIC_INFORMATION, *PPROCESS_EXTENDED_BASIC_INFORMATION;
```

A description of `PROCESS_BASIC_INFORMATION` members follows:

- `ExitStatus` is the exit code of the process (if exited). Otherwise, `STATUS_PENDING` is returned (0x103); this is called `STILL_ACTIVE` in Windows API headers.
- `PebBaseAddress` is the address of the *Process Environment Block* (PEB). See the next section for more details on the PEB.
- `AffinityMask` is a processor bitmask, indicating which processors can be used by threads in this process (for the current process group). A usable processor is represented with a `1` bit.
- `UniqueProcessId` is the process ID.
- `InheritedFromUniqueProcessId` is the parent process ID, from which certain properties are inherited (if not explicitly specified), such as current directory, environment variables, and more.

The extended structure has some additional flags for the process:

- `IsProtectedProcess` indicates the process is protected ("standard" protection or *Protected Process Light* (PPL).
- `IsWow64Process` indicates the process is 32-bit and running on a 64-bit system.
- `IsProcessDeleting` indicates the process is no longer running code, but is kept alive because it's still being referenced (such as having a handle open to it). This is sometimes referred to as a "Zombie Process". You can see such processes with my *Object Explorer* tool (*System / Zombie Processes* menu item).
- `IsCrossSessionCreate` indicates the process was created by a process running in a different session. The most common case is UWP processes, which are always launched by the *DCOM Launch* service, hosted in a standard *SvcHost.exe* running in session zero.
- `IsFrozen` indicates the process' threads are all suspended. This is typical for UWP processes whose windows are minimized.
- `IsBackground` indicates the process is in *Background Mode*. All the process' threads run at base priority of 4, the process I/O priority is "Very Low", and its memory priority is 1. See the official documentation for more on background mode. Chapter 6 provides more details.
- `IsStronglyNamed` indicates the process has some application identifier associated with it. The typical case is UWP processes having their full package name.
- `IsSecureProcess` indicates a process running in *Virtual Trust Level* (VTL 1), if *Virtualization Based Security* (VBS) is active. Typical secure processes are the Secure Kernel and *Lsaiso.exe*.
- `IsSubsystemProcess` indicates the process is running under the *Windows Subsystem for Linux* (WSL) version 1 - a process running a Linux application. This is also referred to as a *Pico Process*.

There is no need to set the `Size` member; it's filled in automatically by the API based on the passed in buffer size.

Here is an example that uses `ProcessBasicInformation` to display some process details:

```
std::string ProcessFlagsToString(PROCESS_EXTENDED_BASIC_INFORMATION const& ebi) {
    std::string flags;
    if (ebi.IsProtectedProcess)
        flags += "Protected, ";
    if (ebi.IsFrozen)
        flags += "Frozen, ";
    if (ebi.IsSecureProcess)
        flags += "Secure, ";
    if (ebi.IsCrossSessionCreate)
        flags += "Cross Session, ";
    if (ebi.IsBackground)
        flags += "Background, ";
    if (ebi.IsSubsystemProcess)
        flags += "WSL, ";
    if (ebi.IsStronglyNamed)
```

```
        flags += "Strong Name, ";
    if (ebi.IsProcessDeleting)
        flags += "Deleting, ";
    if (ebi.IsWow64Process)
        flags += "Wow64, ";

    if (!flags.empty())
        return flags.substr(0, flags.length() - 2);
    return "";
}

void DisplayInfo(HANDLE hProcess) {
    PROCESS_EXTENDED_BASIC_INFORMATION ebi;
    if (NT_SUCCESS(NtQueryInformationProcess(hProcess,
        ProcessBasicInformation, &ebi, sizeof(ebi), nullptr))) {
        auto& bi = ebi.BasicInfo;
        printf("PID: %6u PPID: %6u Pri: %2u PEB: 0x%p %s\n",
            HandleToULong(bi.UniqueProcessId),
            HandleToULong(bi.InheritedFromUniqueProcessId),
            bi.BasePriority, bi.PebBaseAddress,
            ProcessFlagsToString(ebi).c_str());
    }
}
```

5.2.2: `ProcessIoCounters` (2, Query)

This information class returns some I/O statistics for the process using the following structure (defined in *WinNt.h*):

```
typedef struct _IO_COUNTERS {
    ULONGLONG ReadOperationCount;
    ULONGLONG WriteOperationCount;
    ULONGLONG OtherOperationCount;
    ULONGLONG ReadTransferCount;
    ULONGLONG WriteTransferCount;
    ULONGLONG OtherTransferCount;
} IO_COUNTERS;
```

5.2.3: `ProcessVmCounters` (3, Query)

This information class provides various memory counters for the process. It accepts any of the following structures:

```c
typedef struct _VM_COUNTERS {
    SIZE_T PeakVirtualSize;
    SIZE_T VirtualSize;
    ULONG PageFaultCount;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    SIZE_T QuotaPeakPagedPoolUsage;
    SIZE_T QuotaPagedPoolUsage;
    SIZE_T QuotaPeakNonPagedPoolUsage;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
} VM_COUNTERS, *PVM_COUNTERS;

typedef struct _VM_COUNTERS_EX {
    SIZE_T PeakVirtualSize;
    SIZE_T VirtualSize;
    ULONG PageFaultCount;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    SIZE_T QuotaPeakPagedPoolUsage;
    SIZE_T QuotaPagedPoolUsage;
    SIZE_T QuotaPeakNonPagedPoolUsage;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
    SIZE_T PrivateUsage;    // same as PagefileUsage
} VM_COUNTERS_EX, *PVM_COUNTERS_EX;

typedef struct _VM_COUNTERS_EX2 {
    VM_COUNTERS_EX CountersEx;
    SIZE_T PrivateWorkingSetSize;
    SIZE_T SharedCommitUsage;
} VM_COUNTERS_EX2, *PVM_COUNTERS_EX2;
```

The various terms used for the memory counters are officially documented, so will not be detailed here.

5.2.4: `ProcessTimes` (4, Query)

This information class provides the process creation time, exit time (if exited), user-time and kernel-time execution using the following structure:

```
typedef struct _KERNEL_USER_TIMES {
    LARGE_INTEGER CreateTime;
    LARGE_INTEGER ExitTime;
    LARGE_INTEGER KernelTime;
    LARGE_INTEGER UserTime;
} KERNEL_USER_TIMES, *PKERNEL_USER_TIMES;
```

The Windows API provides the `GetProcessTimes` function to retrieved this same data.

`CreateTime` and `ExitTime` are in the usual 100 nsec units from January 1, 1601. `KernelTime` and `UserTime` are in the usual 100 nsec units.

5.2.5: `ProcessBasePriority` (5, Set) and `ProcessPriorityClass` (18, Query/Set)

These information classes allow changing the process base priority, which affects all thread priorities within a process. Both information classes require the process handle to have the `PROCESS_SET_INFORMATION` access mask bit.

The Windows API provides six possible values, referred to as *Priority Classes*: Low (4), Below Normal (6), Normal (8), Above Normal (10), Highest (13), and Realtime (24) (see the docs for `SetPriorityClass` API). `ProcessPriorityClass` allows querying and setting a priority class value based on the following structure:

```
#define PROCESS_PRIORITY_CLASS_UNKNOWN 0
#define PROCESS_PRIORITY_CLASS_IDLE 1
#define PROCESS_PRIORITY_CLASS_NORMAL 2
#define PROCESS_PRIORITY_CLASS_HIGH 3
#define PROCESS_PRIORITY_CLASS_REALTIME 4
#define PROCESS_PRIORITY_CLASS_BELOW_NORMAL 5
#define PROCESS_PRIORITY_CLASS_ABOVE_NORMAL 6

typedef struct _PROCESS_PRIORITY_CLASS {
    BOOLEAN Foreground;
    UCHAR PriorityClass;
} PROCESS_PRIORITY_CLASS, *PPROCESS_PRIORITY_CLASS;
```

The `Foreground` member indicates the process should be considered a foreground process, which means that on client machines the quantum of threads in this process are triple length (around ~90 msec by default instead of the standard ~30 msec). See chapter 6 for more on thread quantums.

`SystemBasePriority` provides a more flexible means to set a base priority that does not have to be one of the above values. The value provided is a `KPRIORITY`, which is just a `LONG`. The number translates directly to a base priority. The high (31) bit can be set to specify the Foreground bit.

Here is an example of setting the current base priority to 7:

```
KPRIORITY priority = 7;
NtSetInformationProcess(NtCurrentProcess(), ProcessBasePriority,
    &priority, sizeof(priority));
```

Curiously enough, increasing a process base priority (beyond its current level) with `ProcessBasePriority` requires the `SeIncreaseBasePriorityPrivilege` privilege to be in the target's process token (by default granted to administrators and not standard users), but using `ProcessPriorityClass` is allowed with all priority classes except Realtime without any special privileges.

5.2.6: `ProcessHandleCount` (20, Query)

This information returns the number of handles in the process, and optionally the highest number of handles to ever exist in the process. THe full expected structure is the following:

```
typedef struct _PROCESS_HANDLE_INFORMATION {
    ULONG HandleCount;
    ULONG HandleCountHighWatermark;
} PROCESS_HANDLE_INFORMATION, *PPROCESS_HANDLE_INFORMATION;
```

You can supply a pointer to a single `ULONG`, in which case the result would be the current handle count only.

5.2.7: `ProcessSessionInformation` (24, Query)

This information class returns the session this process is attched to (`ULONG`).

5.2.8: `ProcessImageFileName` (27) and `ProcessImageFileNameWin32` (43) (Query)

These information classes return the process executable image file name as a `UNICODE_STRING`. The caller must allocate a string big enough to accommodate the path. `ProcessImageFileName` returns the string in NT device form ((“\Device\HarddiskVolume…”), while `ProcessImageFileNameWin32` returns it in Win32 form (“C:\MyFolder\…”). Here is an example:

```
BYTE buffer[512];
NtQueryInformationProcess(hProcess, ProcessImageFileName,
    buffer, sizeof(buffer), nullptr);
auto exePath = (UNICODE_STRING*)buffer;
printf("NT: %wZ\n", exePath);
NtQueryInformationProcess(hProcess, ProcessImageFileNameWin32,
    buffer, sizeof(buffer), nullptr);
exePath = (UNICODE_STRING*)buffer;
printf("Win32: %wZ\n", exePath);
```

5.2.9: `ProcessImageInformation` (37, Query)

This information class returns a fixed structure of type `SECTION_IMAGE_INFORMATION` that provides some of the details part of the *Portable Executable* (PE):

```
typedef struct _SECTION_IMAGE_INFORMATION {
    PVOID TransferAddress;
    ULONG ZeroBits;
    SIZE_T MaximumStackSize;
    SIZE_T CommittedStackSize;
    ULONG SubSystemType;
    union {
        struct {
            USHORT SubSystemMinorVersion;
            USHORT SubSystemMajorVersion;
        };
        ULONG SubSystemVersion;
    };
    union {
        struct {
            USHORT MajorOperatingSystemVersion;
            USHORT MinorOperatingSystemVersion;
        };
        ULONG OperatingSystemVersion;
    };
    USHORT ImageCharacteristics;
    USHORT DllCharacteristics;
    USHORT Machine;
    BOOLEAN ImageContainsCode;
    union {
        UCHAR ImageFlags;
```

```
        struct {
            UCHAR ComPlusNativeReady : 1;
            UCHAR ComPlusILOnly : 1;
            UCHAR ImageDynamicallyRelocated : 1;
            UCHAR ImageMappedFlat : 1;
            UCHAR BaseBelow4gb : 1;
            UCHAR ComPlusPrefer32bit : 1;
            UCHAR Reserved : 2;
        };
    };
    ULONG LoaderFlags;
    ULONG ImageFileSize;
    ULONG CheckSum;
} SECTION_IMAGE_INFORMATION, *PSECTION_IMAGE_INFORMATION;
```

The various members correspond to various parts of a PE header. Look online for a full description.

The term "COMPlus" used in the above structure means ".NET". COM+ has a different meaning today, but the initial name for .NET was "COM+".

5.2.10: `ProcessHandleInformation` (51, Query)

This information class returns a snapshot of the handles in the given process with the following structures:

```
typedef struct _PROCESS_HANDLE_TABLE_ENTRY_INFO {
    HANDLE HandleValue;
    ULONG_PTR HandleCount;
    ULONG_PTR PointerCount;
    ULONG GrantedAccess;
    ULONG ObjectTypeIndex;
    ULONG HandleAttributes;
    ULONG Reserved;
} PROCESS_HANDLE_TABLE_ENTRY_INFO, *PPROCESS_HANDLE_TABLE_ENTRY_INFO;

// private
typedef struct _PROCESS_HANDLE_SNAPSHOT_INFORMATION {
    ULONG_PTR NumberOfHandles;
    ULONG_PTR Reserved;
    PROCESS_HANDLE_TABLE_ENTRY_INFO Handles[1];
} PROCESS_HANDLE_SNAPSHOT_INFORMATION, *PPROCESS_HANDLE_SNAPSHOT_INFORMATION;
```

The information is similar to the system-wide handles returned by `NtQuerySystemInformation` we've seen in chapter 4. Most notable is the missing object address.

5.2.11: `ProcessHandleTable` (58, Query)

This information class is supported in Windows 8.1 and later and returns an array of handle values only in the given process. Each value is a `ULONG`. The number of handles returned should be retrieved by the returned size. Divide it by `sizeof(ULONG)` to get the number of returned handles.

5.2.12: `ProcessCommandLineInformation` (60, Query)

This information class returns the full command line used to invoke the given process as a `UNICODE_STRING`. Here is an example:

```
BYTE buffer[2048];  // arbitrary
NtQueryInformationProcess(hProcess, ProcessCommandLineInformation,
    buffer, sizeof(buffer), nullptr);
auto cmdLine = (UNICODE_STRING*)buffer;
```

The longest possible command line can have 32767 characters (the longest string `UNICODE_STRING` can describe).

5.3: The Process Environment Block (PEB)

The `PEB` is a user-mode data structure each user-mode process has. One reason to have a `PEB` is to minimize user to kernel transitions when information about the process is required by the process itself. Another reason is that some of the information is not very "interesting" from the kernel's perspective if that information cannot be abused to hurt the kernel.

Gaining access to the `PEB` of the current process is done with the convenience function `NtCurrentPeb`. "Convenience" means it's not an actual native API, but rather a macro locating the `PEB` using the current *Thread Environment Block* (TEB):

```
#define NtCurrentPeb() (NtCurrentTeb()->ProcessEnvironmentBlock)
```

The current TEB (`NtCurrentTeb`) is available by accessing the `GS` register (x64) or `FS` register (x86) (Similarly, it's taken from certain registers on ARM/ARM64). Both `NtCurrentPeb` and `NtCurrentTeb` are defined in *WinNt.h*

Getting access to the `PEB` of another process requires calling `NtQueryInformationProcess` with `ProcessBasicInformation`. Of course, the retrieved `PEB` pointer is only valid in the target process. Getting to that data requires calling `ReadProcessMemory` / `WriteProcessMemory` (Windows API), or

`NtReadVirtualMemory` / `NtWriteVirtualMemory` (native API). The latter functions are described in chapter 8.

The `PEB` is a large structure - displaying it here would take too many pages. I recommend you look at the structure in the *phnt* headers (it's also available with the public Microsoft symbols) while various members are described.

> Not all members of the `PEB` are described as some are no longer used, and the list of members is quite long as it is.

- `InheritedAddressSpace` is `TRUE` if the process is cloned (`RtlCloneUserProcess`) and `FALSE` otherwise.
- `ReadImageFileExecOptions` seems to be always zero.
- `BeingDebugged` is `TRUE` if the process is running under a debugger. Note that changing this value to `FALSE` does not change that fact. This can be used to "trick" code to conclude that the process is not being debugged, when in fact it still is. The `IsDebuggerPresent` Windows API examines this member to report if the process is being debugged. The "correct" way of checking without relying on the `PEB` is by calling the Windows API `CheckRemoteDebuggerPresent`, which calls `NtQueryInformationProcess` with `ProcessDebugPort`, returning a valid handle to a *Debug* object if the process is being debugged.
- `ImageUsesLargePages` is set if the PE image is mapped using large pages.
- `IsProtectedProcess` is set if the process is protected (e.g. *Csrss.exe*, *Smss.exe*).
- `IsImageDynamicallyRelocated` is set if the PE image can be loaded into any address (usually `TRUE`).
- `SkipPatchingUser32Forwarders` is set for "safe" CD/DVD content; seems to be a legacy setting.
- `IsPackagedProcess` is set if process has been created based on an AppX package.
- `IsAppContainer` is set if the process runs inside an AppContainer.
- `IsProtectedProcessLight` is set if the process is PPL protected (compared to classic protection).
- `IsLongPathAwareProcess` is set if the process supports paths longer than `MAX_PATH` (260) characters.
- `ImageBaseAddress` is the base address the executable image is mapped to.
- `Ldr` is a pointer to `PEB_LDR_DATA` structure:

```
typedef struct _PEB_LDR_DATA {
    ULONG Length;
    BOOLEAN Initialized;
    HANDLE SsHandle;
    LIST_ENTRY InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
    PVOID EntryInProgress;
    BOOLEAN ShutdownInProgress;
    HANDLE ShutdownThreadId;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

The main ingredient is a set of three linked list that store information about the loaded modules in the process in three different ways:

- `InLoadOrderModuleList` stores the list in module load order.
- `InMemoryOrderModuleList` stored the list in the same order as the load order after all modules have been verified.
- `InInitializationOrderModuleList` stores the list in order of initialization. Naturally, this excludes the executable module, since it has no initialization like a DLL does (no `DllMain` function).

Each module entry is of type `LDR_DATA_TABLE_ENTRY`, containing quite a few details about a module:

```
typedef struct _LDR_DATA_TABLE_ENTRY {
    LIST_ENTRY InLoadOrderLinks;
    LIST_ENTRY InMemoryOrderLinks;
    union {
        LIST_ENTRY InInitializationOrderLinks;
        LIST_ENTRY InProgressLinks;
    };
    PVOID DllBase;
    PLDR_INIT_ROUTINE EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    union {
        UCHAR FlagGroup[4];
        ULONG Flags;
        struct {
            ULONG PackagedBinary : 1;
            ULONG MarkedForRemoval : 1;
            ULONG ImageDll : 1;
```

```
        ULONG LoadNotificationsSent : 1;
        ULONG TelemetryEntryProcessed : 1;
        ULONG ProcessStaticImport : 1;
        ULONG InLegacyLists : 1;
        ULONG InIndexes : 1;
        ULONG ShimDll : 1;
        ULONG InExceptionTable : 1;
        ULONG ReservedFlags1 : 2;
        ULONG LoadInProgress : 1;
        ULONG LoadConfigProcessed : 1;
        ULONG EntryProcessed : 1;
        ULONG ProtectDelayLoad : 1;
        ULONG ReservedFlags3 : 2;
        ULONG DontCallForThreads : 1;
        ULONG ProcessAttachCalled : 1;
        ULONG ProcessAttachFailed : 1;
        ULONG CorDeferredValidate : 1;
        ULONG CorImage : 1;
        ULONG DontRelocate : 1;
        ULONG CorILOnly : 1;
        ULONG ChpeImage : 1;
        ULONG ChpeEmulatorImage : 1;
        ULONG ReservedFlags5 : 1;
        ULONG Redirected : 1;
        ULONG ReservedFlags6 : 2;
        ULONG CompatDatabaseProcessed : 1;
    };
};
USHORT ObsoleteLoadCount;
USHORT TlsIndex;
LIST_ENTRY HashLinks;
ULONG TimeDateStamp;
struct _ACTIVATION_CONTEXT *EntryPointActivationContext;
PVOID Lock;         // SRW Lock
PLDR_DDAG_NODE DdagNode;
LIST_ENTRY NodeModuleLink;
struct _LDRP_LOAD_CONTEXT *LoadContext;
PVOID ParentDllBase;
PVOID SwitchBackContext;
RTL_BALANCED_NODE BaseAddressIndexNode;
RTL_BALANCED_NODE MappingInfoIndexNode;
ULONG_PTR OriginalBase;
```

```
    LARGE_INTEGER LoadTime;
    ULONG BaseNameHashValue;
    LDR_DLL_LOAD_REASON LoadReason;
    ULONG ImplicitPathOptions;
    ULONG ReferenceCount; // since WIN10
    ULONG DependentLoadFlags;
    UCHAR SigningLevel;     // since REDSTONE2
    ULONG CheckSum;         // since 22H1
    PVOID ActivePatchImageBase;
    LDR_HOT_PATCH_STATE HotPatchState;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

Here is an example of enumerating the three lists using a common function:

```
void ListModules(LIST_ENTRY* head, int linkOffset) {
    for (auto next = head->Flink; next != head; next = next->Flink) {
        auto data = (PLDR_DATA_TABLE_ENTRY)((PBYTE)next - linkOffset);
        printf("0x%p: %wZ \n", data->DllBase, &data->BaseDllName);
    }
}

int main(int argc, const char* argv[]) {
    auto peb = NtCurrentPeb();
    printf("Load Order\n");
    ListModules(&peb->Ldr->InLoadOrderModuleList,
        offsetof(LDR_DATA_TABLE_ENTRY, InLoadOrderLinks));

    printf("\nMemory Order\n");
    ListModules(&peb->Ldr->InMemoryOrderModuleList,
        offsetof(LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks));

    printf("\nInitialization Order\n");
    ListModules(&peb->Ldr->InInitializationOrderModuleList,
        offsetof(LDR_DATA_TABLE_ENTRY, InInitializationOrderLinks));

    return 0;
}
```

The trick is to pass the offset of the corresponding `LIST_ENTRY` embedded in the `LDR_DATA_TABLE_ENTRY` structure, so that the correct pointer is obtained. This code does the traversal on the current process, yielding output such as the following (Debug build):

```
Load Order
0x00007FF7696F0000: ModList.exe
0x00007FFC832D0000: ntdll.dll
0x00007FFC82590000: KERNEL32.DLL
0x00007FFC81030000: KERNELBASE.dll
0x00007FFC69000000: VCRUNTIME140D.dll
0x00007FFBD9C20000: ucrtbased.dll

Memory Order
0x00007FF7696F0000: ModList.exe
0x00007FFC832D0000: ntdll.dll
0x00007FFC82590000: KERNEL32.DLL
0x00007FFC81030000: KERNELBASE.dll
0x00007FFC69000000: VCRUNTIME140D.dll
0x00007FFBD9C20000: ucrtbased.dll

Initialization Order
0x00007FFC832D0000: ntdll.dll
0x00007FFC81030000: KERNELBASE.dll
0x00007FFC82590000: KERNEL32.DLL
0x00007FFBD9C20000: ucrtbased.dll
0x00007FFC69000000: VCRUNTIME140D.dll
```

Doing the same traversal in another process is trickier, since we would have to read memory that belongs to another process. This is left as an exercise to the interested reader. We'll see full code for that in chapter 8.

The Windows API provides the `EnumProcessModules` API to traverse the modules loaded in any process with a powerful-enough handle, which it does by reading the relevant details from the process' PEB. This function only returns the base addresses of the modules, which forces the caller to call more APIs such as `GetModuleBaseName` and `GetModuleInformation` to get more details, but the returned details are not as rich as the full `LDR_DATA_TABLE_ENTRY`.

The full member of list of `LDR_DATA_TABLE_ENTRY` will not be described here in the interest of time.

Back to the `PEB` structure members:

- `ProcessHeap` is the default process heap handle, available with the macro `RtlProcessHeap` or the Windows API `GetProcessHeap`. See chapter 8 for more in heaps.
- `FastPebLock` is a pointer to a critical section (`RTL_CRITICAL_SECTION`) that is used to synchronize access to various PEB members. The native APIs `RtlAcquirePebLock` and `RtlReleasePebLock` can be used to acquire and release it.

Next are several flags under stored in a union where `CreateProcessFlags` is available as the overarching member:

- `ProcessInJob` is set if the process is part of any job.
- `ProcessInitializing` is set while the process is being initialized by the loader.
- `ProcessUsingVEH` is set if the process is using *Vectored Exception Handlers*.
- `ProcessUsingVCH` is set if the process is using *Vectored Continuation Handlers*.

See the SDK documentation for more on VEH and VCH.

- `KernelCallbackTable` is a pointer to an array of callbacks used by *User32.Dll* that occur after a relevant system call is invoked. You can try the following in WinDbg attached to a process that uses *User32.Dll* (e.g. *Notepad.exe*):

```
0:014> dt nt!_peb @$peb
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
...
   +0x058 KernelCallbackTable : 0x00007ffc`82f4f070 Void
...
0:014> dqs 0x00007ffc`82f4f070 L30
00007ffc`82f4f070  00007ffc`82ee2780 USER32!_fnCOPYDATA
00007ffc`82f4f078  00007ffc`82f47ea0 USER32!_fnCOPYGLOBALDATA
00007ffc`82f4f080  00007ffc`82ee0c00 USER32!_fnDWORD
00007ffc`82f4f088  00007ffc`82ee6a60 USER32!_fnNCDESTROY
00007ffc`82f4f090  00007ffc`82eedac0 USER32!_fnDWORDOPTINLPMSG
00007ffc`82f4f098  00007ffc`82f486d0 USER32!_fnINOUTDRAG
00007ffc`82f4f0a0  00007ffc`82ee7f90 USER32!_fnGETTEXTLENGTHS
00007ffc`82f4f0a8  00007ffc`82f48370 USER32!_fnINCNTOUTSTRING
00007ffc`82f4f0b0  00007ffc`82f48430 USER32!_fnINCNTOUTSTRINGNULL
00007ffc`82f4f0b8  00007ffc`82ee9700 USER32!_fnINLPCOMPAREITEMSTRUCT
00007ffc`82f4f0c0  00007ffc`82ee2be0 USER32!__fnINLPCREATESTRUCT
00007ffc`82f4f0c8  00007ffc`82f484f0 USER32!_fnINLPDELETEITEMSTRUCT
00007ffc`82f4f0d0  00007ffc`82eefe50 USER32!__fnINLPDRAWITEMSTRUCT
00007ffc`82f4f0d8  00007ffc`82f48550 USER32!_fnINLPHELPINFOSTRUCT
...
```

Each callback has a simple, generic, prototype:

```
NTSTATUS KernelToUserCallback(
    PVOID InputBuffer,
    ULONG InputLength);
```

- `ApiSetMap` is a pointer to the API Set mappings, pointing API Set names to their implementation on the current system. Full discussion of API Sets is beyond the scope of this book, please refer to the book "Windows Internals, part 1". Here is the gist of it: API Sets allow separating sets of functions (think of each set as an interface) from the actual implementation binary.

The following example lists that mapping (see the *ApiSets* sample):

```
#define API_SET_SCHEMA_ENTRY_FLAGS_SEALED 1

auto apiSetMap = NtCurrentPeb()->ApiSetMap;
auto apiSetMapAsNumber = ULONG_PTR(apiSetMap);
auto nsEntry = PAPI_SET_NAMESPACE_ENTRY(apiSetMap->EntryOffset +
    apiSetMapAsNumber);

for (ULONG i = 0; i < apiSetMap->Count; i++) {
    auto isSealed = (nsEntry->Flags & API_SET_SCHEMA_ENTRY_FLAGS_SEALED) != 0;

    std::wstring name(PWCHAR(apiSetMapAsNumber + nsEntry->NameOffset),
        nsEntry->NameLength / sizeof(WCHAR));
    printf("%56ws.dll -> %s{", name.c_str(), (isSealed ? "s" : ""));

    auto valueEntry = PAPI_SET_VALUE_ENTRY(apiSetMapAsNumber +
        nsEntry->ValueOffset);
    for (ULONG j = 0; j < nsEntry->ValueCount; j++) {
        //
        // host name
        //
        name.assign(PWCHAR(apiSetMapAsNumber + valueEntry->ValueOffset),
            valueEntry->ValueLength / sizeof(WCHAR));
        printf("%ws", name.c_str());

        if ((j + 1) != nsEntry->ValueCount)
            printf(", ");

        //
        // If there's an alias
        //
        if (valueEntry->NameLength != 0) {
```

```
            name.assign(PWCHAR(apiSetMapAsNumber + valueEntry->NameOffset),
                valueEntry->NameLength / sizeof(WCHAR));
            printf(" [%ws]", name.c_str());
        }

        valueEntry++;
    }
    printf("}\n");
    nsEntry++;
}
```

- `TlsBitmap` is a pointer to `RTL_BITMAP` (see chapter 2 for more on bitmaps) that stores information about indices used with *Thread Local Storage* (TLS). A process guarantees at least 64 TLS indices, stored in the next member, `TlsBitmapBits` (an array of 2 32-bit values). If more than 64 TLS indices are needed, the `TlsExpansionBitmap` member provides that service, with its own bitmap in the following member (`TlsExpansionBitmapBits`), which is an array of 32 `ULONG` values, supporting 1024 (32 * 8) bits (indices).

> Detailed description of TLS is beyond the scope of this book, but it's use is fully documented in the Windows API. Look for the functions `TlsAlloc`, `TlsFree`, `TlsSetValue`, and `TlsGetValue`.

- `NtGlobalFlags` stores the *Image File Execution Options* flags read from the Registry that apply to this process based on the executable name. The key is **HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*exename***. The value name is *GlobalFlag*.

The following members are related to heaps:

- `HeapSegmentReserve` and `HeapSegmentCommit` indicate the initial defaults for reserved memory (in bytes) and committed bytes for a new heap if no override is specified.
- `HeapDeCommitTotalFreeThreshold` and `HeapDeCommitFreeBlockThreshold` are used as thresholds for decommitting memory from heaps when blocks are freed.

The above defaults are read from the Registry key *HKLM\System\CurrentControlSet\Control\Session Manager*.

- `NumberOfHeaps` stored the current number of heaps in the process.
- `MaximumNumberOfHeaps` is the maximum number of heaps the process can have.
- `ProcessHeaps` is a pointer to an array of heap pointers, their count is `NumberOfHeaps`. The `RtlGetProcessHeaps` API returns this array of heaps.

See chapter 8 for more on heaps.

- `GdiSharedHandleTable` is the shared handle table for GDI objects for this session (not just this process).
- `LoaderLock` is a critical section tasked with protecting certain loader operations from concurrent access.

This concludes the `PEB` members I will cover at this point. Newer versions of the book will likely cover more members.

5.4: Suspending and Resuming Processes

The native API provides functions to suspend and resume a process:

```
NTSTATUS NtSuspendProcess(_In_ HANDLE ProcessHandle);

NTSTATUS NtResumeProcess(_In_ HANDLE ProcessHandle);
```

There is no direct counterpart to these APIs in the Windows API. Suspending a process means suspending all the threads in the process, since threads are the ones actually executing.

The handle provided must have the `PROCESS_SUSPEND_RESUME` access mask for these functions to succeed.

5.5: Enumerating Processes (Take 2)

Chapter 4 showed how to enumerate processes with `NtQuerySystemInformation` with a few information classes providing varying levels of detail. There is another way to enumerate processes, receiving open handles to processes that can be accessed by the caller.

The API in question is `NtGetNextProcess`:

```
NTSTATUS NtGetNextProcess(
    _In_opt_ HANDLE ProcessHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ ULONG HandleAttributes,
    _In_ ULONG Flags,
    _Out_ PHANDLE NewProcessHandle);
```

The idea behind this function is to get the "next" process whose handle can be obtained with the requested access mask. If the next process in line is not accessible by the caller with that access masked, it will be skipped. This function is typically invoked in a loop until it fails, which means no more processes can be retrieved with the requested access.

To begin iteration, the input `ProcessHandle` is set to `NULL`. The output handle (if successful) is stored in `NewProcessHandle`. This handle should then passed as `ProcessHandle` in the next iteration. It's important not to forget to close each handle once you're doing using it - the API opens handles, and the client's job to close them at some point, usually after using a returned handle.

`ObjectAttributes` can be zero or a set of flags defined for the `OBJECT_ATTRIBUTES` structure discussed in chapter 2; a value of zero is typical.

The only flag currently supported in `Flags` has a value of 1, and if specified, traverses the processes in reverse order.

Here is an example that iterates through processes with a specific access mask (see the *ProcList* sample for full code):

```
int main() {
    HANDLE hProcess = nullptr;
    for (;;) {
        HANDLE hNewProcess;
        auto status = NtGetNextProcess(hProcess,
            PROCESS_QUERY_LIMITED_INFORMATION, 0, 0, &hNewProcess);
        //
        // close previous handle
        //
        if(hProcess)
            NtClose(hProcess);

        if (!NT_SUCCESS(status))
            break;

        PROCESS_EXTENDED_BASIC_INFORMATION ebi;
        if (NT_SUCCESS(NtQueryInformationProcess(hNewProcess,
            ProcessBasicInformation, &ebi, sizeof(ebi), nullptr))) {
            auto& bi = ebi.BasicInfo;
            printf("PID: %6u PPID: %6u Pri: %2u PEB: 0x%p %s\n",
                HandleToULong(bi.UniqueProcessId),
                HandleToULong(bi.InheritedFromUniqueProcessId),
                bi.BasePriority, bi.PebBaseAddress,
                ProcessFlagsToString(ebi).c_str());
        }

        hProcess = hNewProcess;
    }
    return 0;
}
```

ProcessFlagsToString is a simple function returning a textual description of the flags provided in PROCESS_BASIC_INFORMATION:

```
std::string ProcessFlagsToString(PROCESS_EXTENDED_BASIC_INFORMATION const& ebi) {
    std::string flags;
    if (ebi.IsProtectedProcess)
        flags += "Protected, ";
    if (ebi.IsFrozen)
        flags += "Frozen, ";
    if (ebi.IsSecureProcess)
        flags += "Secure, ";
    if (ebi.IsCrossSessionCreate)
        flags += "Cross Session, ";
    if (ebi.IsBackground)
        flags += "Background, ";
    if (ebi.IsSubsystemProcess)
        flags += "WSL, ";
    if (ebi.IsStronglyNamed)
        flags += "Strong Name, ";
    if (ebi.IsProcessDeleting)
        flags += "Deleting, ";
    if (ebi.IsWow64Process)
        flags += "Wow64, ";

    if (!flags.empty())
        return flags.substr(0, flags.length() - 2);
    return "";
}
```

5.6: Summary

This chapter dealt with several native APIs related to processes. In the next chapter, we'll take a look at native APIs related to threads.

Chapter 6: Threads

Threads are the entities scheduled by the Windows kernel to execute code on processors. This chapter explores the native APIs related to threads.

In this chapter:

- **Creating Threads**
- **Thread Information**
- **The Thread Environment Block (TEB)**
- **Asynchronous Procedure Calls (APC)**
- **Thread Pools**
- **More Thread APIs**

6.1: Creating Threads

The Windows API provides the following functions to create a thread: `CreateThread`, `CreateRemoteThread`, and `CreateRemoteThreadEx`. The native API provides two functions: `RtlCreateUserThread` and `NtCreateThreadEx`. Let's begin with the former (which eventually invokes the latter).

6.1.1: `RtlCreateUserThread`

This is the simpler of the two:

```
NTSTATUS RtlCreateUserThread(
    _In_ HANDLE Process,
    _In_opt_ PSECURITY_DESCRIPTOR ThreadSecurityDescriptor,
    _In_ BOOLEAN CreateSuspended,
    _In_opt_ ULONG ZeroBits,
    _In_opt_ SIZE_T MaximumStackSize,
    _In_opt_ SIZE_T CommittedStackSize,
    _In_ PUSER_THREAD_START_ROUTINE StartAddress,
    _In_opt_ PVOID Parameter,
    _Out_opt_ PHANDLE Thread,
    _Out_opt_ PCLIENT_ID ClientId);
```

This API mimics very closely the `CreateRemoteThread` Windows API. Here are the parameters:

- `Process` is a handle to the process where the thread should be created in. The handle must have the `PROCESS_CREATE_THREAD` access mask bit. If the handle is set to `NtCurrentProcess()`, the thread is created in the caller's process. Note that `NULL` is not a valid value for this parameter.
- `ThreadSecurityDescriptor` is an optional security descriptor to apply to the newly created thread.
- `CreateSuspended` is a Boolean flag that if set, indicates the thread should be created in suspended state. In that case, `NtResumeThread` should be called when appropriate to spring the thread into action.
- `ZeroBits` is typically set to zero. This gives the kernel full freedom as where to allocate the thread's stack.
- `MaximumStackSize` and `CommittedStack` are the maximum reserved and the initial commit stack size in bytes, respectively, rounded up to a page boundary if needed. If zero is specified, the default is taken from the PE header. Noe that the Windows APIs accept the initial commit or the maximum reserved, but not both.
- `StartAddress` is the thread's start address, that has essentially the same prototype as expected by the Windows API functions:

```
typedef NTSTATUS (__stdcall *PUSER_THREAD_START_ROUTINE)(_In_ PVOID ThreadParameter);
```

- `Parameter` is the value passed to the thread function.
- `Thread` is an optional handle pointer where the resulting handle should land upon a successful call. You should always retrieve it, so the handle can (at the very least) be closed.
- `ClientId` is an optional `CLIENT_ID` pointer, where the new thread's unique ID, and the process ID it's in are returned.

The following example creates a thread in the current process:

```
NTSTATUS __stdcall DoWork(PVOID param) {
    //...
}

void SomeFunction() {
    HANDLE hThread;
    status = RtlCreateUserThread(NtCurrentProcess(), nullptr,
        FALSE,              // not suspended
        0, 128 << 10,       // maximum size: 128 KB
        16 << 10,           // initial size: 16 KB
        DoWork, nullptr,    // function and argument
        &hThread, nullptr);
    //...
    NtClose(hThread);
}
```

6.1.2: NtCreateThreadEx

`NtCreateThreadEx` provides more flexibility than `RtlCreateUserThread`:

```
NTSTATUS NtCreateThreadEx(
    _Out_ PHANDLE ThreadHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ HANDLE ProcessHandle,
    _In_ PVOID StartRoutine, // PUSER_THREAD_START_ROUTINE
    _In_opt_ PVOID Argument,
    _In_ ULONG CreateFlags, // THREAD_CREATE_FLAGS_*
    _In_ SIZE_T ZeroBits,
    _In_ SIZE_T StackSize,
    _In_ SIZE_T MaximumStackSize,
    _In_opt_ PPS_ATTRIBUTE_LIST AttributeList);
```

`ThreadHandle` is the returned thread handle on successful invocation. `DesiredAccess` specifies the desired access mask for the returned handle, the typical value being `THREAD_ALL_ACCESS` (this is always the case with the Windows APIs). `ObjectAttributes` is an optional pointer to the standard `OBJECT_ATTRIBUTES`, typically passed as `NULL`.

`ProcessHandle`, `StartRoutine`, `Argument`, and `ZeroBits` serve the same purpose as in `RtlCreateUserThread`. `StackSize` is the initial committed stack size, and `MaximumStackSize` is the maximum reserved stack size.

`CreateFlags` is a set of optional flags that can be specified:

- `THREAD_CREATE_FLAGS_SUSPENDED` (1) - if set, creates the thread in a suspended state.
- `THREAD_CREATE_FLAGS_SKIP_ATTACH` (2) - if set, does not call `DllMain` for DLLs in the process with `THREAD_DLL_ATTACH` or `THREAD_DLL_DETACH` reasons.
- `THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER` (4) - if set, debuggers will not show this thread.
- `THREAD_CREATE_FLAGS_LOADER_WORKER` (0x10) - set if the thread is a loader worker thread.
- `THREAD_CREATE_FLAGS_SKIP_LOADER_INIT` (0x20) - if set, skips loader initialization entirely.
- `THREAD_CREATE_FLAGS_BYPASS_PROCESS_FREEZE` (0x40) - if set, thread is not suspended when process is frozen, such as with `PsSuspendProcess` or a UWP process is suspended.

Note that *phnt* has different defines for some of the above constants.

Finally, `AttributeList` is an optional attribute list to apply to the created thread. You can find more possible attributes by examining the documentation of the `UpdateProcThreadAttribute` Windows API, although the native uses a different way of building attribute lists.

Here are the associated structures:

```
typedef struct _PS_ATTRIBUTE {
    ULONG_PTR Attribute;
    SIZE_T Size;
    union {
        ULONG_PTR Value;
        PVOID ValuePtr;
    };
    PSIZE_T ReturnLength;
} PS_ATTRIBUTE, *PPS_ATTRIBUTE;

typedef struct _PS_ATTRIBUTE_LIST {
    SIZE_T TotalLength;
    PS_ATTRIBUTE Attributes[1];
} PS_ATTRIBUTE_LIST, *PPS_ATTRIBUTE_LIST;
```

An attributes list structure has room for just one attribute, but it can dynamically allocated based on the actual number needed, or event statically by introducing a union.

The following example uses a group affinity attribute to set a specific group affinity for a thread:

```
GROUP_AFFINITY affinity{};  // zero out the Reserved members
affinity.Group = 1;      // group 1
affinity.Mask = 0;       // all processors in the group
PS_ATTRIBUTE_LIST attributes;
attributes.TotalLength = sizeof(attributes);

auto& attribute = attributes.Attributes[0];
attribute.Attribute = PS_ATTRIBUTE_GROUP_AFFINITY;
attribute.Size = sizeof(affinity);
attribute.ValuePtr = &affinity;
attribute.ReturnLength = 0;

status = NtCreateThreadEx(..., &attributes);
```

The following example uses two attributes, showing a way to statically allocate the required structures:

```
union {
    PS_ATTRIBUTE_LIST List;
    //
    // force big enough stack allocation
    //
    BYTE Buffer[FIELD_OFFSET(PS_ATTRIBUTE_LIST, Attributes)
        + 2 * sizeof(PS_ATTRIBUTE)];
} attributes;

GROUP_AFFINITY affinity{};
affinity.Group = 0;
affinity.Mask = 0;

{
    //
    // first attribute
    //
    auto& attribute = attributes.List.Attributes[0];
    attribute.Attribute = PS_ATTRIBUTE_GROUP_AFFINITY;
    attribute.Size = sizeof(affinity);
    attribute.ValuePtr = &affinity;
    attribute.ReturnLength = 0;
}
PROCESSOR_NUMBER ideal{};
ideal.Group = 0;
ideal.Number = 5;   // set CPU 5 as the ideal CPU

{
    //
    // second attribute
    //
    auto& attribute = attributes.List.Attributes[1];
    attribute.Attribute = PS_ATTRIBUTE_IDEAL_PROCESSOR;
    attribute.Size = sizeof(ideal);
    attribute.ValuePtr = &ideal;
    attribute.ReturnLength = 0;
}
attributes.List.TotalLength = sizeof(attributes.Buffer);

status = NtCreateThreadEx(..., &attributes.List);
```

6.2: Thread Information

Just like with processes (and other object types), two main functions exist for getting and setting thread information:

```
NTSTATUS NtQueryInformationThread(
    _In_ HANDLE ThreadHandle,
    _In_ THREADINFOCLASS ThreadInformationClass,
    _Out_writes_bytes_(ThreadInformationLength) PVOID ThreadInformation,
    _In_ ULONG ThreadInformationLength,
    _Out_opt_ PULONG ReturnLength);

NTSTATUS NtSetInformationThread(
    _In_ HANDLE ThreadHandle,
    _In_ THREADINFOCLASS ThreadInformationClass,
    _In_reads_bytes_(ThreadInformationLength) PVOID ThreadInformation,
    _In_ ULONG ThreadInformationLength);
```

The `THREADINFOCLASS` is an enumeration used to specify the type of information to set or query. In this section, we'll take a look at some of these information classes. Unless otherwise specified, the thread handle used must have the `THREAD_QUERY_LIMITED_INFORMATION` or `THREAD_QUERY_INFORMATION` access mask for querying and `THREAD_SET_LIMITED_INFORMATION` or `THREAD_SET_INFORMATION` for setting.

6.2.1: `ThreadBasicInformation` (0, Query)

This value provides some basic details of a thread:

```
typedef struct _THREAD_BASIC_INFORMATION {
    NTSTATUS ExitStatus;
    PTEB TebBaseAddress;
    CLIENT_ID ClientId;
    ULONG_PTR AffinityMask;
    KPRIORITY Priority;
    LONG BasePriority;
} THREAD_BASIC_INFORMATION, *PTHREAD_BASIC_INFORMATION;
```

Here is a description of the members:

- `ExitStatus` is the exit code of the thread (if terminated). Otherwise, the value `STATUS_PENDING` (0x103) is returned. The Windows API defines this as `STILL_ACTIVE`.

- `TebBaseAddress` is the address of the *Thread Environment Block* (TEB). See the next section for more on the TEB.
- `ClientId` is the process and thread IDs.
- `AffinityMask` is the mask of processors that can be used by the thread, where "1" bits indicate valid processors. Normally, it's set to the full range of processors available in the current processor group.
- `Priority` is the thread's current priority (0 to 31).
- `BasePriority` is the base priority of the thread, which can be lower than `Priority` temporarily if because of priority boosting.

6.2.2: `ThreadTimes` (1, Query)

This information class returns the thread creation and exit times, kernel-mode and user-mode execution times. The structure used, `KERNEL_USER_TIMES`, is the same one used in `NtQueryInformationProcess` with `ProcessTimes` in chapter 5. The Windows API `GetThreadTimes` provides the same information.

6.2.3: Thread CPU Priorities

Several information classes are related to thread CPU priorities:

- `ThreadPriority` (2) sets the current thread priority. If the priority is in the real-time range (16-31), the `SeIncreaseBasePriorityPrivilege` is required.
- `ThreadBasePriority` (3) sets the base priority of the thread, on top of which a boost can be applied by the kernel and kernel drivers. *Csrss.exe* processes and processes running with the *Realtime* priority class can change to any priority without restriction.
- `ThreadPriorityBoost` (14) allows settings or querying whether priority boosts should apply to the thread (`ULONG` used with 0 to disable, 1 to enable). By default, boosts are enabled. Note, however, that boots are never applied in the Realtime range (16-31).
- `ThreadActualBasePriority` (25) allows setting the current thread priority similarly to `ThreadBasePriority`, but also changes the current priority to have the same value. Querying is supported as well.

Priority boots (if enabled) occur for the following reasons (not an exhaustive list):

- Thread released after a wait is satisfied on an event or semaphore gets a +1 boost by the kernel. Device drivers can specify their own boost when they call `KeSetEvent` or `KeReleaseSemaphore`.
- GUI threads that receive a message get a +2 boost.
- Threads in the foreground process (one of its windows are in focus) get a +2 boost as well. Boosts are cumulative, so a GUI thread in such a process may get a +4 boost.
- A starved thread (being in the Ready state for more than 4 seconds) might get a boost to priority 15 for a single quantum.

Except for the last boost that drops to the thread's base priority after one quantum of running, the other boosts cause a decay of 1 for the thread's priority after each quantum the thread executed, until the priority reaches the thread's base priority. For more information about thread priorities and scheduling, see chapter 4 in the "Windows Internals part 1" book.

6.2.4: `ThreadSystemThreadInformation` (40, Query)

This information class returns a `SYSTEM_THREAD_INFORMATION`, the same structure we met in chapter 4.

6.2.5: `ThreadNameInformation` (38, Query, Set)

This information class allows setting or getting a description for a thread (available in Windows 10 and later). The name is just a description that serves no "real" purpose, but is used to easily identify threads of interest in the debugger. Visual Studio's debugger supports setting and querying this value.

The expected structure is just a wrapper around a `UNICODE_STRING`:

```
typedef struct _THREAD_NAME_INFORMATION {
   UNICODE_STRING ThreadName;
} THREAD_NAME_INFORMATION, *PTHREAD_NAME_INFORMATION;
```

This functionality is available with the Windows API functions `SetThreadDescription` and `GetThreadDescription`.

6.2.6: `ThreadIoPriority` (22, Query, Set)

This information class queries or sets the I/O priority for the thread. The value provided or queried is from the `IO_PRIORITY_HINT` enumeration:

```
typedef enum _IO_PRIORITY_HINT {
    IoPriorityVeryLow = 0,  // Defragging, content indexing and other background I/O
    IoPriorityLow,          // Prefetching for applications.
    IoPriorityNormal,       // Normal I/O (default)
    IoPriorityHigh,         // Used by filesystems for checkpoint I/O
    IoPriorityCritical,     // Used by memory manager. Not available for applications
    MaxIoPriorityTypes
} IO_PRIORITY_HINT;
```

For a query operation, this is exactly what is returned. For a set operation, an extended structure can be specified:

```
typedef struct _IO_PRIORITY_HINT_INFORMATION_EX {
    IO_PRIORITY_HINT    PriorityHint;
    BOOLEAN             BoostOutstanding;
} _IO_PRIORITY_HINT_INFORMATION_EX;
```

If specified, `BoostOutstanding` indicates if existing I/O operations started from this thread should be boosted immediately (if higher than current).

6.2.7: `ThreadSuspendCount` (35, Query)

This information class returns the suspend count of a thread. Zero means the thread is not suspended, while any positive value indicates the thread is suspended. `NtSuspendThread` and `NtResumeThread` can be used to suspend/resume a thread. If a thread is suspended with calls to `NtSuspendThread`, the same number of calls to `NtResumeThread` must be made to resume the thread. Here are the APIs:

```
NTSTATUS NtSuspendThread(
    _In_ HANDLE ThreadHandle,
    _Out_opt_ PULONG PreviousSuspendCount);
NTSTATUS NtResumeThread(
    _In_ HANDLE ThreadHandle,
    _Out_opt_ PULONG PreviousSuspendCount);
```

The `THREAD_SUSPEND_RESUME` access mask is required for the handle used to suspend a thread.

6.3: Synchronization

Threads sometimes need to coordinate work by waiting on various kernel dispatcher (“waitable”) objects, which maintain a state of *signaled* or *non-signaled*. A thread can wait on one or more objects using the following basic APIs:

```
NTSTATUS NtWaitForSingleObject(
    _In_ HANDLE Handle,
    _In_ BOOLEAN Alertable,
    _In_opt_ PLARGE_INTEGER Timeout);
NTSTATUS NtWaitForMultipleObjects(
    _In_ ULONG Count,
    _In_reads_(Count) HANDLE Handles[],
    _In_ WAIT_TYPE WaitType,
    _In_ BOOLEAN Alertable,
    _In_opt_ PLARGE_INTEGER Timeout);
```

The Windows API (rough) equivalents are `WaitForSingleObject(Ex)` and `WaitForMultipleObjects(Ex)`.

Example dispatcher object types are: process, thread, mutex, event, semaphore, job, file. We’ll look at the APIs available for working with various object types in chapter 7.

Both functions require a `Timeout` parameter to indicate how long the wait should last at most. Specifying `NULL` means forever (as long as it takes). Negative values indicate relative time (in the usual 100 nsec units), while positive values indicate absolute time with the same units measured from January 1, 1601 at midnight UT.

The handles provided point to the object(s) to wait on. `NtWaitForSingleObject` waits for a single object, while `NtWaitForMultipleObjects` can wait on at most 64 objects. The `WaitType` indicates whether all objects must become signaled for the wait to succeed (`WaitAll`), or just one (`WaitAny`).

The `Aletrable` flag indicates if the wait should be alterable. If `TRUE`, then if APCs are queued to the thread while waiting, they execute, and the wait is satisfied. See the section *Asynchronous Procedure Calls* later in this chapter for more on APCs.

The common return values for both single and multiple waits are the following:

- `STATUS_TIMEOUT` - the timeout has elapsed, but the object(s) have not become signaled. This value cannot be returned if the the timeout is `NULL` (infinite wait).
- `STATUS_SUCCESS` - the object waited upon has become signaled before the timeout elapsed. This applies to a single object wait and a multiple object wait with `WaitAll`.
- `STATUS_USER_APC` - an alertable wait is satisfied because APC(s) executed.

For single waits, `STATUS_ABANDONED` may be returned if the wait was satisfied because a mutex was *abandoned*. An abandoned mutex is one that was not released by a thread before that thread terminated. The kernel forcefully released the mutex (making it signaled), but the next thread to successfully acquire the mutex receives a `STATUS_ABANDONED` to indicate the previous owner thread of that mutex terminated prematurely.

For a multiple wait with `WaitAny`, a return value in the range `STATUS_WAIT_0` (0) to `STATUS_WAIT_63` (63) indicates the index of the object which became signaled. In the rare case where an abandomed mutex is one of the object becoming signaled, `STATUS_ABANDONED` (0x80) to `STATUS_ABANDONED + 63` is returned.

See chapter 7 for more on dispatcher objects.

6.4: The Thread Environment Block (TEB)

Every user-mode thread has a user-mode `TEB` structure, that stores some interesting information for the thread. Just like the `PEB`, the `TEB` is pretty big and will not be reprinted here. Examine *ntpebteb.h* file from *phnt*. In this section, I'll describe of the structure's members.

> Inside Wow64 processes, there are two `TEB`s for a thread: a 64-bit and a 32-bit one. There are two variants of `TEB` - `TEB32` and `TEB64`. The differences are around sizes of pointers and some 32/64 bit size changes.

The first member of `TEB` named `NtTib`, and is of type `TIB`. It's small enough to repeat here:

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;
} EXCEPTION_REGISTRATION_RECORD;

typedef struct _NT_TIB {
    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
    PVOID StackBase;
    PVOID StackLimit;
    PVOID SubSystemTib;
    union {
        PVOID FiberData;
        DWORD Version;
    };
    PVOID ArbitraryUserPointer;
    struct _NT_TIB *Self;
} NT_TIB;
```

`ExceptionList` is an exception list created by this thread (implemented on x86 only). `StackBase` and `StackLimit` are the current used stack addresses for this thread. `SubSystemTib` does not seem to be used. `FiberData` contains something if the thread has been converted to fiber. In most cases, the `Version` member is the "used" one, which seems to store the value `0x1e00` (latest version of the support for OS2 Windows had). `ArbitraryUserPointer` seems like a location to store some thread-specific data, and it is, but not by developers. Some scenarios use this pointer, so don't use it yourself. There is *Thread Local Storage* for that. Finally, `Self` points to the beginning of `NT_TIB` which is also the beginning of the `TEB`.

Moving to other members of the `TEB`:

- `EnvironmentPointer` seems to always have the value `NULL`.
- `ClientId` is a `CLIENT_ID` structure, storing the unique process and thread IDs.
- `ActiveRpcHandle` doesn't seem to be used, always having the value `NULL`.
- `ProcessEnvironmentBlock` points to the `PEB` (discussed in chapter 5).
- `LastErrorValue` is the last value returned from a Windows API call, usually read by calling `GetLastError`. the Windows API `SetLastError` can be used to set it.
- `CountOfOwnedCriticalSections` is typically zero. It's used in Debug build of Windows to keep track of the number of critical sections owned by the thread.
- `CsrClientThread` is `NULL` in all processes except *Csrss.exe* processes, where it may point to a `CSR_THREAD` structure, maintained by *csrss.exe* for threds that registered with *Csrss.exe*. This structure is defined like so:

```
typedef struct _CSR_THREAD {
    LARGE_INTEGER CreateTime;
    LIST_ENTRY Link;
    LIST_ENTRY HashLinks;
    CLIENT_ID ClientId;
    struct _CSR_PROCESS *Process;
    HANDLE ThreadHandle;
    ULONG Flags;
    LONG ReferenceCount;
    ULONG ImpersonateCount;
} CSR_THREAD;
```

It points to a `CSR_PROCESS` shown below, along with `CSR_NT_SESSION` pointed to by the `CSR_PROCESS`:

```
typedef struct _CSR_PROCESS {
    CLIENT_ID ClientId;
    LIST_ENTRY ListLink;
    LIST_ENTRY ThreadList;
    PCSR_NT_SESSION NtSession;
    HANDLE ClientPort;
    PCH ClientViewBase;
    PCH ClientViewBounds;
    HANDLE ProcessHandle;
    ULONG SequenceNumber;
    ULONG Flags;
    ULONG DebugFlags;
    LONG ReferenceCount;
    ULONG ProcessGroupId;
    ULONG ProcessGroupSequence;
    ULONG LastMessageSequence;
    ULONG NumOutstandingMessages;
    ULONG ShutdownLevel;
    ULONG ShutdownFlags;
    LUID Luid;
    PVOID ServerDllPerProcessData[ANYSIZE_ARRAY];
} CSR_PROCESS;

typedef struct _CSR_NT_SESSION {
    LIST_ENTRY SessionLink;
    ULONG SessionId;
    LONG ReferenceCount;
    STRING RootDirectory;
```

```
} CSR_NT_SESSION;
```

These data structures can be obtained by loading *Csrss.exe* into a *WinDbg* instance and using the standard `dt` command.

Back to more TEB fields:

- `CurrentLocale` is the current locale used by the thread. For example, 0x409 is "en-US", equivalent to `MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US)`.
- `ExceptionCode` stores the last exception code that occurred on this thread.
- `LastStatusValue` stores the last status value returned from a system call on this thread.
- `SubProcessTag` is non-zero for threads that handle a service. The service tag information is maintained by the *Service Control Manager* (SCM, running in *services.exe*). The exported `I_QueryTagInformation` function from *AdvApi32.dll* can be used to obtain the mapping to a service name. Here is its definition with supporting structures and enums:

```
typedef enum _TAG_INFO_LEVEL {
    eTagInfoLevelNameFromTag = 1, // TAG_INFO_NAME_FROM_TAG
    eTagInfoLevelNamesReferencingModule, // TAG_INFO_NAMES_REFERENCING_MODULE
    eTagInfoLevelNameTagMapping, // TAG_INFO_NAME_TAG_MAPPING
    eTagInfoLevelMax
} TAG_INFO_LEVEL;

typedef enum _TAG_TYPE {
    eTagTypeService = 1,
    eTagTypeMax
} TAG_TYPE;

typedef struct _TAG_INFO_NAME_FROM_TAG_IN_PARAMS {
    DWORD dwPid;
    DWORD dwTag;
} TAG_INFO_NAME_FROM_TAG_IN_PARAMS, *PTAG_INFO_NAME_FROM_TAG_IN_PARAMS;

typedef struct _TAG_INFO_NAME_FROM_TAG_OUT_PARAMS {
    DWORD eTagType;
    LPWSTR pszName;
} TAG_INFO_NAME_FROM_TAG_OUT_PARAMS, *PTAG_INFO_NAME_FROM_TAG_OUT_PARAMS;

DWORD I_QueryTagInformation(
    _In_opt_ LPCWSTR pszMachineName,
    _In_ TAG_INFO_LEVEL eInfoLevel,
    _Inout_ PVOID pTagInfo);
```

You could use code like the following to get a service name based on a `SubProcessTag`:

```
std::wstring GetServiceNameByTag(DWORD pid, uint32_t tag) {
    static auto QueryTagInformation = (PQUERY_TAG_INFORMATION)GetProcAddress(
        GetModuleHandle(L"advapi32"), "I_QueryTagInformation");

    TAG_INFO_NAME_FROM_TAG info = { 0 };
    info.InParams.dwPid = pid;
    info.InParams.dwTag = tag;
    auto err = QueryTagInformation(nullptr, eTagInfoLevelNameFromTag, &info);
    if (err)
        return L"";
    return info.OutParams.pszName;
}
```

Reading the tag itself can be done by getting the `TEB` address and then reading from the process memory. For example:

```
uint32_t GetSubProcessTag(HANDLE hThread) {
    THREAD_BASIC_INFORMATION tbi;
    auto status = NtQueryInformationThread(hThread,
        ThreadBasicInformation, &tbi, sizeof(tbi), nullptr);
    if (!NT_SUCCESS(status))
        return 0;

    if (tbi.TebBaseAddress == 0)
        return 0;

    auto pid = GetProcessIdOfThread(hThread);
    auto hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION | PROCESS_VM_READ,
        FALSE, pid);
    if (!hProcess)
        return 0;

    uint32_t tag = 0;
    BOOL isWow = FALSE;
    //
    // assume a 64-bit OS
    //
    IsWow64Process(hProcess, &isWow);
    if (isWow) {
        auto teb = (TEB32*)tbi.TebBaseAddress;
```

```
        ReadProcessMemory(process->GetHandle(),
            (BYTE*)teb + offsetof(TEB32, SubProcessTag),
            &tag, sizeof(ULONG), nullptr);
    }
    else {
        auto teb = (TEB*)tbi.TebBaseAddress;
        ReadProcessMemory(process->GetHandle(),
            (BYTE*)teb + offsetof(TEB, SubProcessTag),
            &tag, sizeof(tag), nullptr);
    }
    return tag;
}
```

- `TlsSlots` is the array of Thread Local Storage used for this thread. The Windows APIs `TlsSetValue` and `TlsGetValue` use this array to store/query TLS values.
- `ReservedForOle` stores a pointer to a data structure used to manage the *Component Object Model* (COM) state for this thread. If the thread has not called `CoInitialize`(`Ex`), then this pointer is `NULL`. The structure type is `SOleThreadData`:

```
typedef enum {
    OLE_LOCALTID                = 0x01,    // TID is in current process
    OLE_UUIDINITIALIZED         = 0x02,
    OLE_INTHREADDETACH          = 0x04,
    OLE_CHANNELTHREADINITIALZED = 0x08,
    OLE_WOWTHREAD               = 0x10,
    OLE_THREADUNINITIALIZING = 0x20,    // thread in CoUninitialize
    OLE_DISABLE_OLE1DDE         = 0x40,    // thread can't use a DDE
    OLE_APARTMENTTHREADED       = 0x80,    // STA thread
    OLE_MULTITHREADED           = 0x100,   // MTA thread
    OLE_IMPERSONATING           = 0x200,   // impersonating
    OLE_DISABLE_EVENTLOGGER  = 0x400,
    OLE_INNEUTRALAPT            = 0x800,   // thread in NTA
    OLE_DISPATCHTHREAD          = 0x1000,
    OLE_HOSTTHREAD              = 0x2000, // host STA
    OLE_ALLOWCOINIT             = 0x4000,
    OLE_PENDINGUNINIT           = 0x8000,
    OLE_FIRSTMTAINIT            = 0x10000,
    OLE_FIRSTNTAINIT            = 0x20000,
    OLE_APTINITIALIZING         = 0x40000,
    OLE_UIMSGSINMODALLOOP       = 0x80000, // Thread has messages in modal loop
    OLE_MARSHALING_ERROR_OBJECT = 0x100000,
    OLE_WINRT_INITIALIZE        = 0x200000,
```

```
    OLE_APPLICATION_STA       = 0x400000, // Application STA (ASTA)
    OLE_IN_SHUTDOWN_CALLBACKS = 0x800000,
    OLE_POINTER_INPUT_BLOCKED = 0x1000000,
    OLE_IN_ACTIVATION_FILTER  = 0x2000000,
    OLE_ASTATOASTAEXEMPT_QUIRK = 0x4000000,
    OLE_ASTATOASTAEXEMPT_PROXY = 0x8000000,
    OLE_ASTATOASTAEXEMPT_INDOUBT = 0x10000000,
    OLE_DETECTED_USER_INITIALIZED = 0x20000000,
    OLE_BRIDGE_STA            = 0x40000000,
    OLE_NAINITIALIZING        = 0x80000000,
} OleTlsFlags;

typedef struct tagSOleTlsData {
    PVOID          pvThreadBase;
    CSmAllocator*  pSmAllocator;
    DWORD          dwApartmentID;
    OleTlsFlags    dwFlags;

    LONG           TlsMapIndex;
    void           *ppTlsSlot;
    DWORD          cComInits;
    DWORD          cOleInits;
    DWORD          cCalls;

    ServerCall     *pServerCall;
    ThreadCallObjectCache *pCallObjectCache;
    ContextStackNode* pContextStack;
    CObjServer*     pObjServer;
    DWORD           dwTIDCaller;
    PVOID           pCurrentCtxForNefariousReaders;

    CObjectContext *pCurrentContext;   // Current context
    CObjectContext *pEmptyCtx;

    ULONGLONG       ContextId;          // Uniquely identifies the current context
    CComApartment*  pNativeApt;        // Native apartment
    IUnknown*       pCallContext;      // call context
    CCtxCall*       pCtxCall;

    CPolicySet*     pPS;
    PVOID           pvPendingCallsFront;
    PVOID           pvPendingCallsBack;
```

```
CAptCallCtrl*  pCallCtrl;
CSrvCallState* pTopSCS;
HWND           hwndSTA;           // STA window
LONG           cORPCNestingLevel;
DWORD          cDebugData;
UUID           LogicalThreadId;    // logical thread id

HANDLE         hThread;            // used for cancellation
HANDLE         hRevert;            // Token before first impersonation
IUnknown       pAsyncRelease;

HWND           hwndDdeServer;
HWND           hwndDdeClient;
ULONG          cServeDdeObjects;
PVOID          pSTALSvrsFront;
HWND           hwndClip;          // Clipboard window
IDataObject*   pDataObjClip;      // Current Clipboard DataObject
DWORD          dwClipSeqNum;
DWORD          fIsClipWrapper;
IUnknown*      punkState;
DWORD          cCallCancellation;
DWORD          cAsyncSends;

CAsyncCall*    pAsyncCallList;
CSurrogatedObjectList* pSurrogateList;

PRWLOCKTLSEntry  pRWLockTlsEntry;
CallEntryBuffer  CallEntry;

InitializeSpyNode* pFirstSpyReg;
InitializeSpyNode* pFirstFreeSpyReg;

CVerifierTlsData*  pVerifierData;
DWORD              dwMaxSpy;
BYTE               cCustomMarshallerRecursion;
PVOID              pDragCursors;
IUnknown*          punkError;
ULONG              cbErrorData;

OutgoingCallData  outgoingCallData;
IncomingCallData  incomingCallData;
OutgoingActivationData outgoingActivationData;
```

```
    ULONG cReentrancyFromUserAPC;
    ModernSTAWaitContext* pModernSTAWaitContext;
    DWORD             dwCrossThreadFlags;
    DWORD             dwNestedRemRelease;
    ULONG             cIncomingTouchedASTACalls;
    PushLogicalThreadId* pTopPushedLogicalThreadId;
    ULONG             iXslockOwnerTableHint;
    OLETLS_PREVENT_RUNDOWN_MITIGATION currentPreventRundownMitigation;
    BOOL              fOweForcedBulkUpdateForCurrentMitigation;
    IUnknown*         pClipboardBroker;
    DWORD             dwActivationType;
    ULONG             cTouchedAstasInActiveCall;
    OXID*             pTouchedAstasInActiveCall;
    UnmarshalForQueryInterface* pTopmostUnmarshalForQueryInterface;
    CoGetStandardMarshalInProgress* pTopmostCoGetStandardMarshalInProgress;
    WireContainerThis* requestContainerPassthroughData;
    ULONG              requestContainerPassthroughDataSize;
    BOOL               freeRequestContainerPassthroughData;
    WireContainerThat* responseContainerPassthroughData;
    ULONG              responseContainerPassthroughDataSize;
    ComTlsFlags        comTlsFlags;
    ThreadLockOrderVerifier<ComLockOrder> lockOrderVerifier;
} SOleTlsData;
```

You can find this definition (and other referenced by it) by loading *ComBase.Dll* into *WinDbg* and using the `dt` command.

6.5: Asynchronous Procedure Calls (APC)

An *Asynchronous Procedure Call* (APC) allows attaching a callback to a thread, so that only that thread can run that callback. To actually execute the callback(s), the thread must be in an *alertable wait* (sometimes called "alertable state").

The native APIs to wait in an alertable state are `NtWaitForSingleObject` and `NtWaitForMultipleObjects` discussed in the section *Synchronization*, where `Alertable` is `TRUE`. Additionally, `NtDelayExecution` offers an alertable wait while sleeping:

```
NTSTATUS NtDelayExecution(
    _In_ BOOLEAN Alertable,     // alertable?
    _In_opt_ PLARGE_INTEGER DelayInterval);
```

Queuing an APC to a thread can be done with `NtQueueApcThread`:

```
typedef VOID (*PPS_APC_ROUTINE)(
    _In_opt_ PVOID ApcArgument1,
    _In_opt_ PVOID ApcArgument2,
    _In_opt_ PVOID ApcArgument3);

NTSTATUS NtQueueApcThread(
    _In_ HANDLE ThreadHandle,
    _In_ PPS_APC_ROUTINE ApcRoutine,
    _In_opt_ PVOID ApcArgument1,
    _In_opt_ PVOID ApcArgument2,
    _In_opt_ PVOID ApcArgument3);
```

The `ThreadHandle` must have the `THREAD_SET_CONTEXT` access mask for this to work. The target thread can be in a different process than the caller, but a 32-bit process cannot queue an APC to a thread that belongs to a 64-bit process. Regardless, the `ApcRoutine` must be valid in the target process. The APC routine itself accepts three arbitrary arguments specified by the caller.

What happens if the target thread never enters an alertable wait? The APCs will never execute. At some point, if more APCs are queued to the thread, queuing will fail. This means that normally the queuer needs to somehow "know" that the target thread goes into an alertable wait from time to time so the APCs can execute.

What happens if a thread waits in an alertable state, but no APCs ever arrive? If the wait is non-infinite, then it will eventually end when the timeout elapses. If the wait is infinite, the thread will wait forever. However, the native API provides ways to release the thread from an alertable wait regardless of APCs:

```
NTSTATUS NtAlertThread(_In_ HANDLE ThreadHandle);
NTSTATUS NtAlertThreadByThreadId(_In_ HANDLE ThreadId);
NTSTATUS NtAlertResumeThread(
    _In_ HANDLE ThreadHandle,
    _Out_opt_ PULONG PreviousSuspendCount);
```

If the thread is in a alertable waiting state, it will be alerted (woken up), and continues execution. The return value from the wait function (or `NtDelayExecution`) is `STATUS_ALERTED` (0x101), rather than `STATUS_USER_APC` (0xc0). The last variant (`NtAlertResumeThread`) also resumes the thread before alerting it. This could be useful if the target thread is known to be suspended.

Finally, `NtTestAlert` tests if the current thread has the "alerted" flag set. If so, `STATUS_ALERTED` is returned, and pending APCs execute.

```
NTSTATUS NtTestAlert();
```

Note that to get the `NtAlert*` APIs to work, the target thread must be waiting in an alertable with a native API (`NtWait...` or `NtDelayExecution`), rather than one of the rough equivalents from Windows APIs (this is one reason the equivalence is not exact).

6.6: Thread Pools

Creating threads explicitly with `NtCreateThread` and similar is fine in many cases. However, there are cases where short amount of work is needed, and the overhead of creating and managing a new thread is overkill. This is where thread pooling comes in. With a thread pool, requests are submitted, and handled by threads from the pool. This provides the following benefits:

- No need to explicitly create and manage threads.
- Execution is usually faster, as some threads in the pool are idle, waiting for work. Once work is received, a thread wakes up, executes the work, and goes back to waiting. This is faster than creating a new thread.
- Even with high load on the thread pool, the number of threads created can be limited. Too many threads cannot be utilized at the same time as there are limited processors on the system.
- The thread pool also supports running code as a result of timer expiring, or objects becoming signaled. This is more efficient than creating threads for these purposes.

The native API functions related to thread pooling begin with `Tp`. The documented Windows APIs are thin wrappers around the native APIs. The parameters are virtually identical, except the native API returns `NTSTATUS`, and any result is provided as an indirect first argument, while the Windows API returns the result directly (or a Boolean if there is no special result). For example, here are two equivalent APIs:

```
NTSTATUS TpAllocWork(       // native API
    _Out_ PTP_WORK *WorkReturn,
    _In_ PTP_WORK_CALLBACK Callback,
    _Inout_opt_ PVOID Context,
    _In_opt_ PTP_CALLBACK_ENVIRON CallbackEnviron);

PTP_WORK CreateThreadpoolWork(   // Windows API
    _In_ PTP_WORK_CALLBACK pfnwk,
    _Inout_opt_ PVOID pv,
    _In_opt_ PTP_CALLBACK_ENVIRON pcbe);
```

Table 6-1 lists the many of the Windows thread pool APIs and their native API equivalents. Refer to the Windows SDK documentation for the description of these APIs.

> Check out the *phnt* project for some more APIs.

Table 6-1: Thread pool APIs

Windows API	Native API
`TrySubmitThreadPoolCallback`	`TpSimpleTryPost`
`CreateThreadpoolWork`	`TpAllocWork`
`CloseThreadpoolWork`	`TpReleaseWork`
`SubmitThreadpoolWork`	`TpPostWork`
`CreateThreadpool`	`TpAllocPool`
`CloseThreadpool`	`TpReleasePool`
`SetThreadpoolThreadMaximum`	`TpSetPoolMaxThreads`
`SetThreadpoolThreadMinimum`	`TpSetPoolMinThreads`
`SetThreadpoolStackInformation`	`TpSetPoolStackInformation`
`QueryThreadpoolStackInformation`	`TpQueryPoolStackInformation`
`CreateThreadpoolCleanupGroup`	`TpAllocCleanupGroup`
`CloseThreadpoolCleanupGroup`	`TpReleaseCleanupGroup`
`SetThreadpoolCallbackCleanupGroup`	`TpSetCallbackCleanupGroup`
`CloseThreadpoolCleanupGroupMembers`	`TpReleaseCleanupGroupMembers`
`CreateThreadpoolTimer`	`TpAllocTimer`
`SetThreadpoolTimer`	`TpSetTimer`
`SetThreadpoolTimerEx`	`TpSetTimerEx`
`IsThreadpoolTimerSet`	`TpIsTimerSet`
`CreateThreadpoolWait`	`TpAllocWait`
`CreateThreadpoolIo`	`TpAllocIoCompletion`

6.7: More Thread APIs

In this section, we'll take a look at a few more thread-related native APIs.

A thread's context represents its execution state in terms of the CPU's registers. The structure used is called `CONTEXT` and is platform-specific. It is described in the Microsoft documentation. Reading or writing the context can be accomplish with the following APIs:

```
NTSTATUS NtGetContextThread(
    _In_ HANDLE ThreadHandle,    // requires THREAD_GET_CONTEXT
    _Inout_ PCONTEXT ThreadContext);
NTSTATUS NtSetContextThread(
    _In_ HANDLE ThreadHandle,    // requires THREAD_SET_CONTEXT
    _In_ PCONTEXT ThreadContext);
```

To safely set the context (or read it in some predictable manner), the thread should be suspended first by calling `NtSuspendThread`:

```
NTSTATUS NtSuspendThread(
    _In_ HANDLE ThreadHandle,
    _Out_opt_ PULONG PreviousSuspendCount);
```

Each call to `NtSuspendThread` must be matched eventually with `NtResumeThread`:

```
NTSTATUS NtResumeThread(
    _In_ HANDLE ThreadHandle,
    _Out_opt_ PULONG PreviousSuspendCount);
```

A suspend count is maintained for each thread, optionally returned by the above APIs. The maximum suspend count is 127.

Sometimes a handle to a thread needs to be obtained. This is where `NtOpenThread` can help:

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

The similarities to `NtOpenProcess` should be evident. The `ClientId` should contain the thread ID in `UniqueThread`, and zero for `UniqueProcess`.

Terminating a thread is possible with `NtTerminateThread` (only use in extreme cases):

```
NTSTATUS NtTerminateThread(
    _In_opt_ HANDLE ThreadHandle,    // requires THREAD_TERMINATE
    _In_ NTSTATUS ExitStatus);
```

Thread enumeration in a specific process is supported by the following API:

```
NTSTATUS NtGetNextThread(
    _In_ HANDLE ProcessHandle,  // requires PROCESS_QUERY_INFORMATION
    _In_opt_ HANDLE ThreadHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ ULONG HandleAttributes,
    _In_ ULONG Flags,           // must be zero
    _Out_ PHANDLE NewThreadHandle);
```

This is similar to `NtGetNextProcess` we encountered in chapter 5. The enumeration starts with `ThreadHandle` set to `NULL`, and ends when the returned status is unsuccessful. Don't forget to close each handle you receive to prevent a handle leak.

6.8: Summary

This chapter looked at thread-related native APIs. In the next chapter, we'll deal with kernel objects and handles.

Chapter 7: Objects and Handles

Much of the functionality in Windows is provided through kernel objects. Accessing kernel objects from user-mode is always performed indirectly through *handles*. This chapter deals with several common types of objects, as well as more generally working with objects and handles.

In this chapter:

- **Objects**
- **Enumerating Objects**
- **Object Manager Namespace**
- **Handles**
- **Enumerating Handles**
- **Specific Object Types**

7.1: Objects

The Windows kernel supports a set of object types, out of which objects can be created. Each object belongs to a type (in itself an object). The Windows API provides many functions to create objects or open handles to existing objects. Each object type has its own set of open and/or create functions, as well as specific functions for manipulating existing objects.

The objects themselves are data structures residing in kernel space, which is one reason user-mode cannot access objects directly, even if an object's address is known.

7.1.1: Object Types

The set of object types supported on a particular version of Windows can be viewed with a tool like *Object Explorer* (figure 7-1).

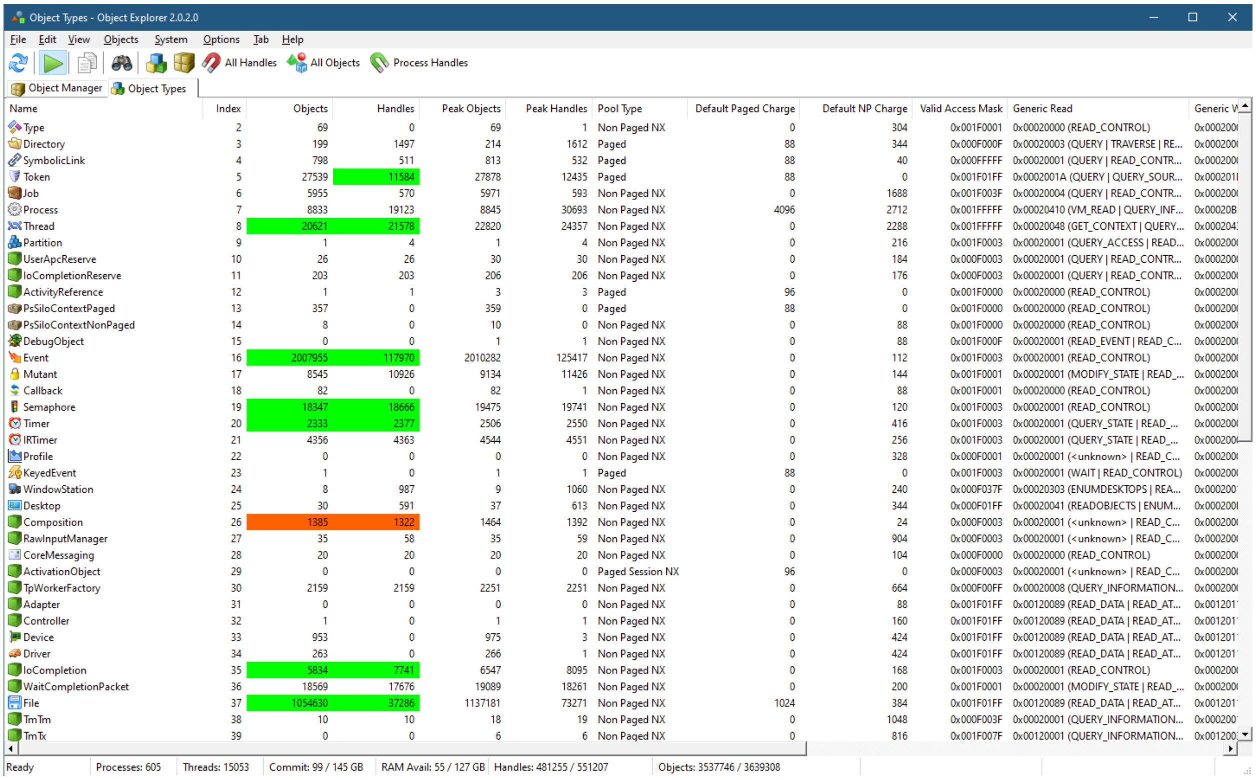


Figure 7-1: Object Explorer showing object types

For each object types, *Object Explorer* shows the total number of objects and handles, the peak number of objects and handles, and a few more details discussed shortly.

Getting the information seen in *Object Explorer* is a matter of calling `NtQueryObject`:

```
typedef enum _OBJECT_INFORMATION_CLASS {
    ObjectBasicInformation,          // OBJECT_BASIC_INFORMATION
    ObjectNameInformation,           // OBJECT_NAME_INFORMATION
    ObjectTypeInformation,           // OBJECT_TYPE_INFORMATION
    ObjectTypesInformation,          // OBJECT_TYPES_INFORMATION
    ObjectHandleFlagInformation,     // OBJECT_HANDLE_FLAG_INFORMATION
    ObjectSessionInformation,
    ObjectSessionObjectInformation,
    MaxObjectInfoClass
} OBJECT_INFORMATION_CLASS;

NTSTATUS NtQueryObject(
    _In_opt_ HANDLE Handle,
    _In_ OBJECT_INFORMATION_CLASS ObjectInformationClass,
    _Out_writes_bytes_opt_(ObjectInformationLength) PVOID ObjectInformation,
    _In_ ULONG ObjectInformationLength,
    _Out_opt_ PULONG ReturnLength);
```

Normally, calling `NtQueryObject` requires a valid handle, with one exception. Getting object types information is done by specifying a `NULL` handle with `ObjectTypesInformation` information class. The returned structure are as follows:

```
typedef struct _OBJECT_TYPE_INFORMATION {
    UNICODE_STRING TypeName;
    ULONG TotalNumberOfObjects;
    ULONG TotalNumberOfHandles;
    ULONG TotalPagedPoolUsage;
    ULONG TotalNonPagedPoolUsage;
    ULONG TotalNamePoolUsage;
    ULONG TotalHandleTableUsage;
    ULONG HighWaterNumberOfObjects;
    ULONG HighWaterNumberOfHandles;
    ULONG HighWaterPagedPoolUsage;
    ULONG HighWaterNonPagedPoolUsage;
    ULONG HighWaterNamePoolUsage;
    ULONG HighWaterHandleTableUsage;
    ULONG InvalidAttributes;
    GENERIC_MAPPING GenericMapping;
    ULONG ValidAccessMask;
    BOOLEAN SecurityRequired;
    BOOLEAN MaintainHandleCount;
    UCHAR TypeIndex;
    CHAR ReservedByte;
    ULONG PoolType;
    ULONG DefaultPagedPoolCharge;
    ULONG DefaultNonPagedPoolCharge;
} OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;

typedef struct _OBJECT_TYPES_INFORMATION {
    ULONG NumberOfTypes;
    // OBJECT_TYPE_INFORMATION follows
} OBJECT_TYPES_INFORMATION, *POBJECT_TYPES_INFORMATION;
```

Each object type has a different size because of the type name (`TypeName` member), so moving to the next object type is somewhat tricky, as it always starts on a pointer-aligned address. The first type starts on such an aligned address as well. The following code shows how to iterate over the types, displaying some key information (error handling omitted):

```
auto size = 1 << 16;
auto buffer = std::make_unique<BYTE[]>(size);
NtQueryObject(nullptr, ObjectTypesInformation, buffer.get(), size, nullptr);
auto types = (OBJECT_TYPES_INFORMATION*)buffer.get();
//
// first type starts at offset of pointer-size bytes
//
auto type = (OBJECT_TYPE_INFORMATION*)((PBYTE)types + sizeof(PVOID));
for (ULONG i = 0; i < types->NumberOfTypes; i++) {
    printf("%-33wZ (%2d) O: %7u H: %7u PO: %7u PH: %7u\n",
        &type->TypeName, type->TypeIndex,
        type->TotalNumberOfObjects, type->TotalNumberOfHandles,
        type->HighWaterNumberOfObjects, type->HighWaterNumberOfHandles);

    //
    // move to next type object
    //
    auto temp = (PBYTE)type + sizeof(OBJECT_TYPE_INFORMATION) +
        type->TypeName.MaximumLength;
    //
    // round up to the next pointer-size address
    //
    type = (OBJECT_TYPE_INFORMATION*)(((ULONG_PTR)temp +
        sizeof(PVOID) - 1) / sizeof(PVOID) * sizeof(PVOID));
}
printf("Total Types: %u\n", types->NumberOfTypes);
```

The full code is in the *ObjectTypes* project.

Here is an example output on a recent Windows 10 machine:

```
Type                      ( 2) O:      69 H:       0 PO:      69 PH:       1
Directory                 ( 3) O:     219 H:    1670 PO:     222 PH:    1680
SymbolicLink              ( 4) O:     789 H:     502 PO:     813 PH:     532
Token                     ( 5) O:   21626 H:   11868 PO:   27878 PH:   12435
Job                       ( 6) O:     656 H:     584 PO:    5971 PH:     595
Process                   ( 7) O:    1693 H:   12957 PO:    8845 PH:   30693
Thread                    ( 8) O:   19402 H:   24711 PO:   22820 PH:   25008
...
```

```
CrossVmEvent               (68) O:        0 H:        0 PO:        0 PH:        0
CrossVmMutant              (69) O:        0 H:        0 PO:        0 PH:        0
VRegConfigurationContext (70) O:       23 H:        0 PO:      358 PH:        0
Total Types: 69
```

Each object type provides the following details in `OBJECT_TYPE_INFORMATION`:

- `TypeName` is the name of the type.
- `TotalNumberOfObjects` is the current number of objects of this type.
- `TotalNumberOfHandles` is the current number of handles open to objects of this type.
- `HighWaterNumberOfObjects` and `HighWaterNumberOfHandles` are the peak number of objects and handles, respectively, since the system started.
- `InvalidAttributes` indicates the invalid attributes (`OBJ_xxx`) that cannot be applied to handles to objects of this type.
- `GenericMapping` stores the mapping of generic access to specific access. `GENERIC_MAPPING` is a documented structure defined like so (in *WinNt.h*):

```
typedef struct _GENERIC_MAPPING {
    ACCESS_MASK GenericRead;
    ACCESS_MASK GenericWrite;
    ACCESS_MASK GenericExecute;
    ACCESS_MASK GenericAll;
} GENERIC_MAPPING;
```

For each object type, the meaning of `GENERIC_READ`, `GENERIC_WRITE`, `GENERIC_EXECUTE`, and `GENERIC_ALL` is different - this mapping provides the details. For example, *Process* objects map `GENERIC_READ` to `PROCESS_QUERY_INFORMATION | PROCESS_VM_READ | READ_CONTROL`.

- `ValidAccessMask` indicates the valid access mask bits for handles to objects of this type (for example, for *Process* object type, this is `PROCESS_ALL_ACCESS`).
- `SecurityRequired` indicates if objects of this type must have a security descriptor if they are created with default security. Normally, objects that can be named have this set to `TRUE`.
- `MaintainHandleCount` is set if the number of handles in each process for this object type is maintained by the kernel.
- `TypeIndex` is the internal type index for this type.
- `PoolType` indicates the pool type used to allocate objects of this type. This is one of `PagedPool` (1), `NonPagedPool` (0), `NonPagedPoolNx` (0x200, non-paged pool with no Execute permissions), or `PagedPoolSession` (0x21). For example, *Process* objects are allocated from `NonPagedPoolNx`, which means these objects are always maintained in RAM.
- `DefaultPagedPoolCharge` and `DefaultNonPagedPoolCharge` are the default allocation charges to the creator process, if the objects of this type are allocated from paged pool or non-paged pool, respectively.

All other members are always zero.

7.2: Enumerating Objects

Can we get a list of all objects in the system? Looking at `NtQuerySystemInformation`, we see a promising `SystemObjectInformation` value. However, by default the kernel does not keep track of object creation in some list. Calling `NtQuerySystemInformation` with `SystemObjectInformation` returns `STATUS_UNSUCCESSFUL`.

To enable object tracking, a global flag must be set. The easiest way to set it is by running the *Gflags* utility from the Windows SDK, and check "Maintain a list of objects for each type" (figure 7-2). This changes a value in the Registry, and the system must be restarted for this flag to take effect. Once restarted, enumerating objects can work.

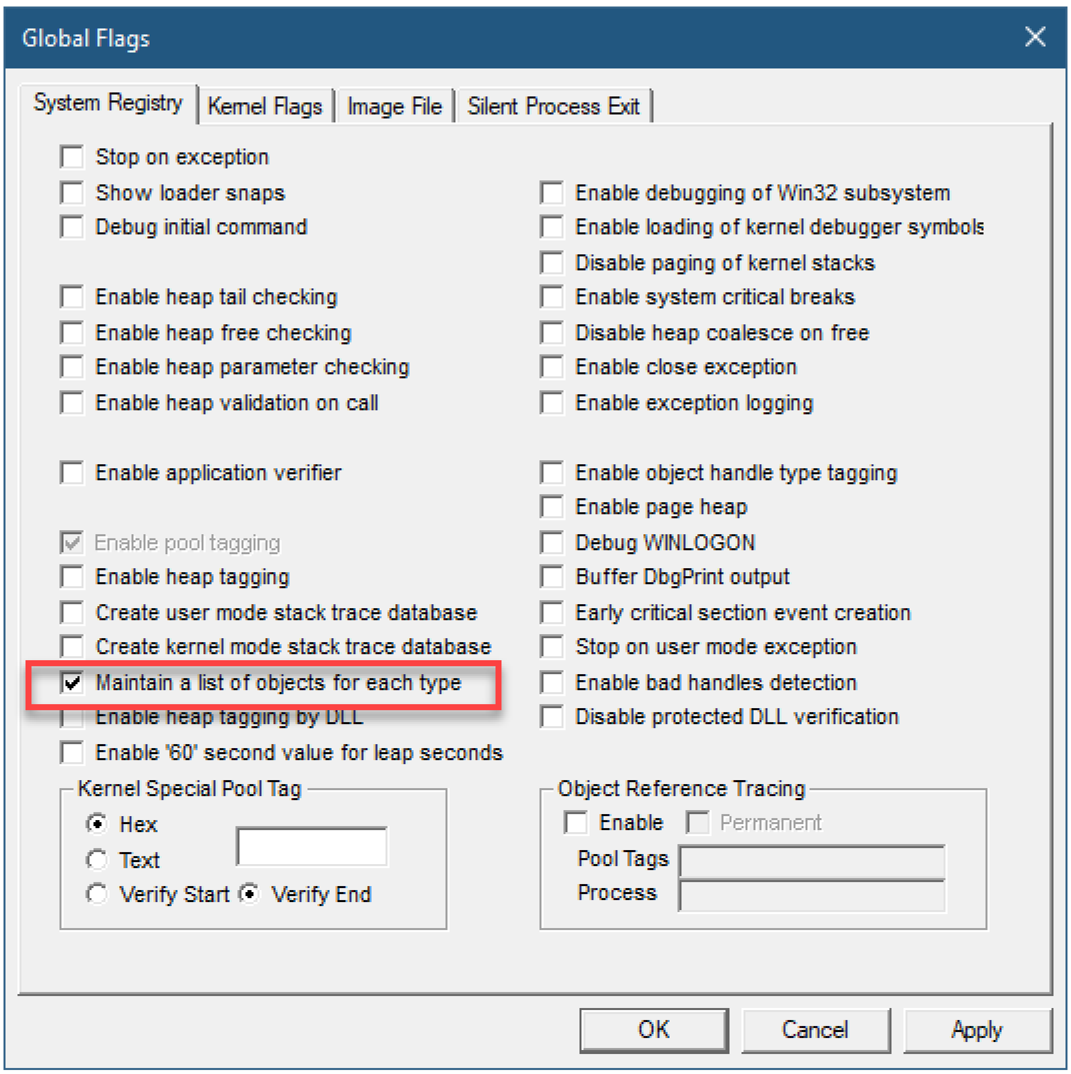


Figure 7-2: GFlags utility

> Why isn't this option enabled by default? Because of some overhead - this was especially important to avoid in the (distant) past, when hardware wasn't as powerful as it is now, and memory sizes were smaller. It's difficult to estimate the cost of this overhead, but it exists. More CPU time must be spent when objects are created, and more memory is needed as well to maintain this extra information.

> Setting "Maintain a list of objects for each type" causes the value 0x4000 to be added to the *GlobalFlag* value in the Registry key *HKLM\SYSTEM\CurrentControlSet\Control\Session Manager*. This value is read when the system boots.

The enumeration starts by allocating a large enough buffer. The API will not return the necessary buffer size if you pass zero. This is because calculating the number of bytes required requires going over all objects, which is inefficient. Instead, the API tries to fill the provided buffer with as many objects as possible, and if the buffer is not big enough, the call fails with a `STATUS_INFO_LENGTH_MISMATCH` status. It's your job to allocate more memory and try again. Here is one way to go about it:

```
ULONG size = 1 << 22;
auto buffer = std::make_unique<BYTE[]>(size);
NTSTATUS status;
while ((status = NtQuerySystemInformation(SystemObjectInformation,
    buffer.get(), size, nullptr)) == STATUS_INFO_LENGTH_MISMATCH) {
    size *= 2;
    buffer = std::make_unique<BYTE[]>(size);
}
if (!NT_SUCCESS(status)) {
    printf("Object tracking is not turned on.\n");
    return status;
}
```

The code starts with a 4MB (`1 << 22`) allocation, and doubles it every time the length is not enough. Expect at least 12MB of memory to be allocated for this call to succeed.

The above code also takes care of the case where the call fails for a different reason, which would mean the global flag is not set. Of course, the code ignores allocation failures for simplicity.

Now the enumeration can begin. The returned information is in two layers: the first is a `SYSTEM_OBJECTTYPE_INFORMATION` structure, providing information about a type, very similar (but not identical) to `OBJECT_TYPE_INFORMATION`. Then, the list of objects for this type follows as `SYSTEM_OBJECT_INFORMATION` structures. This is non-trivial, because the structures don't have fixed sizes, as there are names (for types, and possibly for objects).

`SYSTEM_OBJECTTYPE_INFORMATION` is defined like so:

```
typedef struct _SYSTEM_OBJECTTYPE_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfObjects;
    ULONG NumberOfHandles;
    ULONG TypeIndex;
    ULONG InvalidAttributes;
    GENERIC_MAPPING GenericMapping;
    ULONG ValidAccessMask;
    ULONG PoolType;
    BOOLEAN SecurityRequired;
    BOOLEAN WaitableObject;
    UNICODE_STRING TypeName;
} SYSTEM_OBJECTTYPE_INFORMATION, *PSYSTEM_OBJECTTYPE_INFORMATION;
```

The differences from `OBJECT_TYPE_INFORMATION` are as follows:

- The next type object is pointed `NextEntryOffset` bytes from this one.
- The `WaitableObject` flag is added, indicating whether objects of this type are waitable (dispatcher objects).
- There are some details that are missing compared to `OBJECT_TYPE_INFORMATION`.

For each object, we get a `SYSTEM_OBJECT_INFORMATION`:

```
typedef struct _SYSTEM_OBJECT_INFORMATION {
    ULONG NextEntryOffset;
    PVOID Object;
    HANDLE CreatorUniqueProcess;
    USHORT CreatorBackTraceIndex;
    USHORT Flags;
    LONG PointerCount;
    LONG HandleCount;
    ULONG PagedPoolCharge;
    ULONG NonPagedPoolCharge;
    HANDLE ExclusiveProcessId;
    PVOID SecurityDescriptor;
    UNICODE_STRING NameInfo;
} SYSTEM_OBJECT_INFORMATION, *PSYSTEM_OBJECT_INFORMATION;
```

Here is a description of its members:

- `NextEntryOffset` stores the number of bytes to move forward from the beginning of the result buffer to get to the next object.

- `Object` is the object's address in kernel space.
- `CreatorUniqueProcess` is the process ID of the creator of this object.
- `CreatorBackTraceIndex` seems to be always zero.
- `Flags` is the flags set in the object's header. The possible flags are shown below:

```
typedef enum {
    OBF_NEW_OBJECT              = 0x01,
    OBF_KERNEL_OBJECT           = 0x02,
    OBF_KERNEL_ONLY_ACCESS      = 0x04,
    OBF_EXCLUSIVE_OBJECT        = 0x08,
    OBF_PERMANENT_OBJECT        = 0x10,
    OBF_DEFAULT_SECURITY_QUOTA  = 0x20,
    OBF_SINGLE_HANDLE_ENTRY     = 0x40,
    OBF_DELETED_INLINE          = 0x80
} OBJECT_HEADER_FLAGS;
```

- `PointerCount` is the reference count of the object mixed with an inverted usage count (Windows 8.1 and later). This value should not be relied upon because of that usage count. See the "Windows Internals 7th edition, Part 2" book for more details.
- `HandleCount` is the number of handles open to this object (could be zero if it's being used by kernel code only).
- `PagedPoolCharge` and `NonPagedPoolCharge` are the memory allocation charges for this object. These are the same as the ones reported for their type.
- `ExclusiveProcessId` is usually zero, but is the process ID that owns the object if it's exclusive. An exclusive object cannot be used by any other process. This kind of power is available to kernel-mode code only.
- `SecurityDescriptor` is the security descriptor (if any) that is attached to the object.
- `NameInfo` is the name of the object (if any).

Given the above definitions, here is a full iteration over all types and all objects within each type:

```
auto type = (SYSTEM_OBJECTTYPE_INFORMATION*)buffer.get();
for(;;) {
    //
    // do something with type
    //
    printf("%wZ O: %u H: %u\n",
        &type->TypeName, type->NumberOfObjects, type->NumberOfHandles);

    //
    // get the first object
    //
```

```
    auto obj = (SYSTEM_OBJECT_INFORMATION*)
        ((PBYTE)type + sizeof(*type) + type->TypeName.MaximumLength);
    for (;;) {
        //
        // do something with the object
        //
        printf("0x%p (%wZ) P: 0x%X H: %u PID: %u EPID: %u\n",
            obj->Object, &obj->NameInfo, obj->PointerCount, obj->HandleCount,
            HandleToULong(obj->CreatorUniqueProcess),
            HandleToULong(obj->ExclusiveProcessId));

        //
        // if no more objects, break
        //
        if (obj->NextEntryOffset == 0)
            break;

        //
        // move to next object
        //
        obj = (SYSTEM_OBJECT_INFORMATION*)(buffer.get() + obj->NextEntryOffset);
    }
    //
    // if no more types, break
    //
    if (type->NextEntryOffset == 0)
        break;

    //
    // move to next type
    //
    type = (SYSTEM_OBJECTTYPE_INFORMATION*)(buffer.get() + type->NextEntryOffset);
}
```

Note that the type "ObjectType" and its objects are not provided by the enumeration as separate objects, because the type objects themselves are these objects.

The full code can be found in the *AllObjects* project.

7.3: Object Manager Namespace

Named objects are kept in a tree-like hierarchy by the kernel's Object Manager. This hierarchy can be viewed with tools like *WinObj* (from *Sysinternals*) and my own *Object Explorer*. Figure 7-3 shows *WinObj* (it's best to run it elevated to view the full directory object structure).

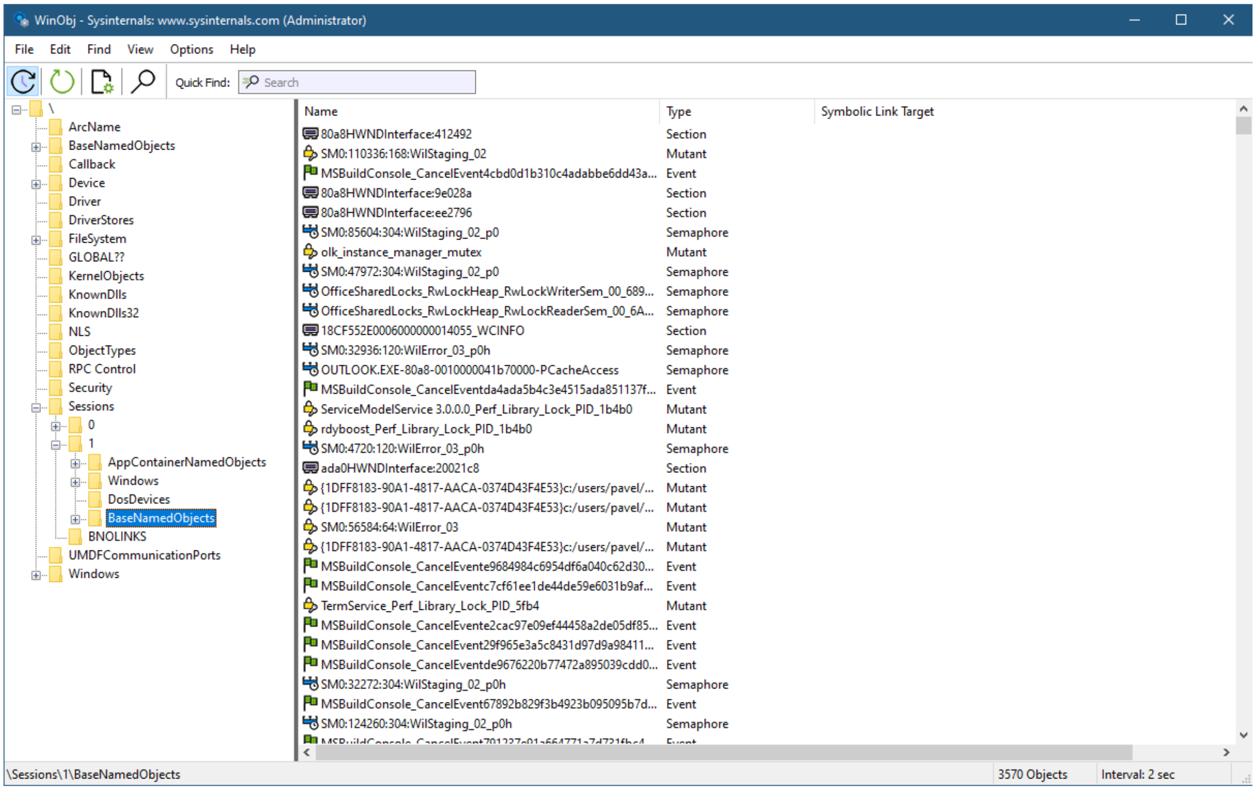


Figure 7-3: WinObj from Sysinternals

The "directories" shown in *WinObj* have nothing to do with the file system; rather, these are Directory kernel objects, which are containers for other kernel objects (including other directories). The following briefly describes some of the more "known" Directory objects:

- *BaseNamedObjects* contains session 0 named objects.
- *Device* contains device objects, representing physical or virtual devices.
- *Driver* contains driver objects, one for each kernel driver loaded into the system.
- *Global??* (also called *??*) contains symbolic links that are legal values to pass to the `CreateFile` Windows API (with a prefix of `\\.\`). The native API counterparts (`NtCreateFile` and `NtOpenFile`) are not limited to this directory.
- *KnownDlls* and *KnownDlls32* contain Section objects that map to "known DLLs" listed in the Registry key *HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs* early in Windows boot (by *Smss.exe*).
- *ObjectType* contains all type objects.
- *Callback* contains Callback kernel objects.

- *\Sessions\n\BaseNamedObjects* contains named kernel objects that were created by processes in session *n* (where *n* is a session number).

Querying the contents of a directory object requires opening a handle to the directory in question:

```
NTSTATUS NtOpenDirectoryObject(
    _Out_ PHANDLE DirectoryHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes);
```

The `ObjectAttributes` is where the directory name is specified with the usual `InitializeObjectAttributes`. The following code opens a handle to the root directory:

```
HANDLE hDirectory;
UNICODE_STRING name;
RtlInitUnicodeString(&name, L"\\");
OBJECT_ATTRIBUTES attr;
InitializeObjectAttributes(&attr, &name, 0, nullptr, nullptr);
status = NtOpenDirectoryObject(&hDirectory, DIRECTORY_QUERY, &attr);
if(!NT_SUCCESS(status)) {
    // do something with hDirectory
    NtClose(hDirectory);
}
```

The `AccessMask` parameter indicates the requested access to the directory. These can be a combination of the following access flags:

```
#define DIRECTORY_QUERY                 0x0001
#define DIRECTORY_TRAVERSE              0x0002
#define DIRECTORY_CREATE_OBJECT         0x0004
#define DIRECTORY_CREATE_SUBDIRECTORY   0x0008
#define DIRECTORY_ALL_ACCESS            (STANDARD_RIGHTS_REQUIRED | 0xf)
```

With a directory handle in hand, the contents of the directory can be obtained with `NtQueryDirectoryObject`:

```
typedef struct _OBJECT_DIRECTORY_INFORMATION {
    UNICODE_STRING Name;
    UNICODE_STRING TypeName;
} OBJECT_DIRECTORY_INFORMATION, *POBJECT_DIRECTORY_INFORMATION;

NTSTATUS NtQueryDirectoryObject(
    _In_ HANDLE DirectoryHandle,
    _Out_writes_bytes_opt_(Length) PVOID Buffer,
    _In_ ULONG Length,
    _In_ BOOLEAN ReturnSingleEntry,
    _In_ BOOLEAN RestartScan,
    _Inout_ PULONG Context,
    _Out_opt_ PULONG ReturnLength);
```

Using this function is non-intuitive. Here is description of the parameters:

- `DirectoryHandle` is the input handle to the directory of interest.
- `Buffer` is a caller-allocated buffer to receive the results, and `Length` is its size in bytes.
- `ReturnSingleEntry` indicates if a single entry is requested. If `FALSE`, as many entries as would fit in the buffer are returned.
- `RestartScan` indicates if the retrieval should start from the beginning. Typically, you would set it to `TRUE` for the first call (if retrieving multiple results), and `FALSE` on subsequent calls if the provided buffer is insufficient in size.
- `Context` is the index of the item requested, and the last one actually retrieved. Normally, you would set it to zero on the first call, but keep using it for subsequent calls (if the buffer is too small).
- `ReturnedLength` is the usual optional return value indicating the number of bytes actually used (or needed, if even a single entry was too big for the buffer).

To simplify the use of `NtQueryDirectoryObject`, one could pass a big buffer, where it's unlikely a second call would be required. But there is no good way to tell what size would be large enough, as it heavily depends on the directory in question and the system on which the code runs; so, it's best to be prepared for multiple calls.

The returned buffer is an array of `OBJECT_DIRECTORY_INFORMATION` objects, providing the object's name and its type name. Fortunately, simple array access works, because even though the name and type of the object are provided, the strings themselves are stored after the array itself so that no special calculation is needed.

Assuming `hDirectory` is a valid directory handle, the following example shows how to correctly get all objects in that directory with a (possibly) limited buffer:

```
ULONG index = 0;
bool first = true;
ULONG size = 1 << 12;    // example size (4KB)
auto buffer = std::make_unique<BYTE[]>(size);
auto data = (POBJECT_DIRECTORY_INFORMATION)buffer.get();

int start = 0;
for(;;) {
    status = NtQueryDirectoryObject(hDirectory, buffer.get(), size,
        FALSE, first, &index, nullptr);
    if (!NT_SUCCESS(status))
        break;
    first = false;

    //
    // loop number of entries from the last call
    //
    for (ULONG i = 0; i < index - start; i++) {
        //
        // do something with data[i].Name and data[i].TypeName
        //
    }
    start = index;
    if (status != STATUS_MORE_ENTRIES)
        break;
}
```

7.3.1: Symbolic Links

The *SymbolicLink* object type is a kernel object that points to another kernel object (similar in spirit to a shell shortcut). Given a symbolic link object name, its target can be retrieved by opening a handle to it (`NtOpenSymbolicLinkObject`) and then querying for the target (`NtQuerySymbolicLinkObject`):

```
NTSTATUS NtOpenSymbolicLinkObject(
    _Out_ PHANDLE LinkHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes);

NTSTATUS NtQuerySymbolicLinkObject(
    _In_ HANDLE LinkHandle,
    _Inout_ PUNICODE_STRING LinkTarget,
    _Out_opt_ PULONG ReturnedLength);
```

The parameters should be self-explanatory at this point. Here is an example of getting a symbolic link target given a C_string symbolic link object name:

```
std::wstring GetSymbolicLinkTarget(PCWSTR symolicLinkName) {
    UNICODE_STRING linkName;
    RtlInitUnicodeString(&linkName, symolicLinkName);
    OBJECT_ATTRIBUTES attr;
    InitializeObjectAttributes(&attr, &linkName, 0, nullptr, nullptr);

    //
    // open a handle to the symbolic link
    //
    HANDLE hLink;
    auto status = NtOpenSymbolicLinkObject(&hLink, GENERIC_READ, &attr);
    if (!NT_SUCCESS(status))
        return L"";

    WCHAR buffer[512]{};    // big enough buffer in practice
    UNICODE_STRING target;
    RtlInitUnicodeString(&target, buffer);
    target.MaximumLength = sizeof(buffer);

    //
    // query for the target
    //
    status = NtQuerySymbolicLinkObject(hLink, &target, nullptr);
    NtClose(hLink);
    return buffer;
}
```

You can also create a symbolic link, if you have enough permissions:

```
NTSTATUS NtCreateSymbolicLinkObject(
    _Out_ PHANDLE LinkHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ PUNICODE_STRING LinkTarget);
```

7.3.2: The *ObjDir* Sample

The *ObjDir* sample puts most of what we've seen in this section, by querying the contents of a specified directory on the command line, showing the objects names, type names, and symbolic link targets (for symbolic link objects only, of course).

The `main` function obtains a directory, and calls a helper function to get the results. If successful, it displays them:

```
int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 2) {
        printf("Usage: ObjDir <directory>\n");
        return 0;
    }

    NTSTATUS status;
    auto objects = EnumDirectoryObjects(argv[1], status);
    if (objects.empty() && !NT_SUCCESS(status)) {
        printf("Error: 0x%X\n", status);
    }
    else {
        printf("Directory: %ws\n", argv[1]);
        printf("%-20s %-50s Link Target\n", "Type", "Name");
        printf("%-20s %-50s -----------\n", "----", "----");
        for (auto& obj : objects) {
            printf("%-20ws %-50ws %ws\n",
                obj.TypeName.c_str(), obj.Name.c_str(),
                obj.LinkTarget.c_str());
        }
    }
    return 0;
}
```

The returned `vector` is of type `ObjectInfo`:

```
struct ObjectInfo {
    std::wstring Name;
    std::wstring TypeName;
    std::wstring LinkTarget;
};
```

The workhorse is `EnumDirectoryObjects`:

```cpp
std::vector<ObjectInfo> EnumDirectoryObjects(PCWSTR directory, NTSTATUS& status) {
    HANDLE hDirectory;
    UNICODE_STRING name;
    RtlInitUnicodeString(&name, directory);
    OBJECT_ATTRIBUTES attr;
    InitializeObjectAttributes(&attr, &name, 0, nullptr, nullptr);
    status = NtOpenDirectoryObject(&hDirectory, DIRECTORY_QUERY, &attr);
    if (!NT_SUCCESS(status))
        return {};

    ULONG index = 0;
    bool first = true;

    ULONG size = 1 << 12;
    auto buffer = std::make_unique<BYTE[]>(size);
    auto data = (POBJECT_DIRECTORY_INFORMATION)buffer.get();
    int start = 0;
    std::vector<ObjectInfo> objects;
    for(;;) {
        status = NtQueryDirectoryObject(hDirectory, buffer.get(), size, FALSE,
            first, &index, nullptr);
        if (!NT_SUCCESS(status))
            break;
        first = false;

        for (ULONG i = 0; i < index - start; i++) {
            ObjectInfo info;
            //
            // turn UNICODE_STRING into std::wstring
            //
            info.Name.assign(data[i].Name.Buffer,
                data[i].Name.Length / sizeof(WCHAR));
            info.TypeName.assign(data[i].TypeName.Buffer,
                data[i].TypeName.Length / sizeof(WCHAR));
            if (info.TypeName == L"SymbolicLink")
                info.LinkTarget = GetSymbolicLinkTarget(directory, info.Name);
            objects.push_back(std::move(info));
        }
        start = index;
        if (status != STATUS_MORE_ENTRIES)
            break;
    }
```

```
    NtClose(hDirectory);
    return objects;
}
```

The code is identical to the example shown earlier, except the results are accumulated in a `vector<ObjectInfo>`, and the status is returned as a second argument. The last piece is `GetSymbolicLinkTarget`, which is called only if the current object's type name is "SymbolicLink":

```
std::wstring
GetSymbolicLinkTarget(std::wstring const& directory, std::wstring const& name) {
    auto fullName = directory == L"\\" ? (L"\\" + name) : (directory + L"\\" + name);
    UNICODE_STRING linkName;
    RtlInitUnicodeString(&linkName, fullName.c_str());
    OBJECT_ATTRIBUTES attr;
    InitializeObjectAttributes(&attr, &linkName, 0, nullptr, nullptr);
    HANDLE hLink;
    auto status = NtOpenSymbolicLinkObject(&hLink, GENERIC_READ, &attr);
    if (!NT_SUCCESS(status))
        return L"";

    WCHAR buffer[512]{};
    UNICODE_STRING target;
    RtlInitUnicodeString(&target, buffer);
    target.MaximumLength = sizeof(buffer);
    status = NtQuerySymbolicLinkObject(hLink, &target, nullptr);
    NtClose(hLink);
    return buffer;
}
```

The only difference from the earlier example is that the directory and object names are provided, and the function is forced to concatenate them together, taking care of not duplicating a backslash for the root directory.

Here are a few examples when running *ObjDir* (some formatting applied to better fit a page):

```
c:\>objdir \
Directory: \
Type                 Name                                    Link Target
----                 ----                                    -----------
Event                DSYSDBG.Debug.Trace.Memory.530
Mutant               PendingRenameMutex
Directory            ObjectTypes
FilterConnectionPort storqosfltport
FilterConnectionPort MicrosoftMalwareProtectionRemoteIoPortWD
SymbolicLink         SystemRoot                              \Device\BootDevice\Wi\
ndows
...
Section              LsaPerformance
ALPC Port            SmApiPort
FilterConnectionPort CLDMSGPORT
FilterConnectionPort DtdSel03
FilterConnectionPort MicrosoftMalwareProtectionPortWD
SymbolicLink         OSDataRoot                              \Device\OSDataDevice
Event                SAM_SERVICE_STARTED
Directory            Driver
Directory            DriverStores

c:\>objdir \KernelObjects
Directory: \KernelObjects
Type                 Name                                    Link Target
----                 ----                                    -----------
SymbolicLink         MemoryErrors
Event                LowNonPagedPoolCondition
Session              Session1
Event                SuperfetchScenarioNotify
Event                SuperfetchParametersChanged
SymbolicLink         PhysicalMemoryChange
SymbolicLink         HighCommitCondition
Mutant               BcdSyncMutant
SymbolicLink         HighMemoryCondition
Event                HighNonPagedPoolCondition
Partition            MemoryPartition0
KeyedEvent           CritSecOutOfMemoryEvent
Event                SystemErrorPortReady
SymbolicLink         MaximumCommitCondition
SymbolicLink         LowCommitCondition
Event                HighPagedPoolCondition
```

```
SymbolicLink          LowMemoryCondition
Session               Session0
Event                 LowPagedPoolCondition
Event                 PrefetchTracesReady
```

You may notice in the above output that some symbolic links have no target. These are called *dynamic symbolic links*, and their target is set by a kernel callback. All the above non-target links actually point to Event objects (e.g. "HighMemoryCondition").

Some directories are inaccessible without elevated permissions (for example, "\ObjectTypes"). Run *ObjDir* elevated to get that extra access.

7.4: Handles

Accessing kernel objects from user-mode must be done through *handles*. A handle is an index into a handle table, maintained on a per-process basis. In this section, we'll look at native APIs related to handles in general, applicable to any (or many) object types.

Once a handle is no longer needed, it should be closed with `NtClose` that we have used many times before:

```
NTSTATUS NtClose(_In_ _Post_ptr_invalid_ HANDLE Handle);
```

7.4.1: Handle Duplication

It is sometimes useful to get another handle to the same object based on an existing handle. For example, a handle in a different process cannot be used by the calling process to access the same object unless the handle resides in the caller's process handle table. Handle duplication provides a solution.

The function name is `NtDuplicateObject`, which is the one invoked by the Windows API `DuplicateHandle` function:

```
NTSTATUS NtDuplicateObject(
    _In_ HANDLE SourceProcessHandle,
    _In_ HANDLE SourceHandle,
    _In_opt_ HANDLE TargetProcessHandle,
    _Out_opt_ PHANDLE TargetHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ ULONG HandleAttributes,
    _In_ ULONG Options);
```

The API name (`NtDuplicateObject`) is misleading - no object is duplicated, only a handle is duplicated. The Windows API name is accurate.

A handle from a source process is duplicated into the process handle of a target process. Both source and target processes may be the same one, which is sometimes useful if a different access mask is needed. `SourceHandleProcess` and `TargetHandleProcess` must have `PROCESS_DUP_HANDLE` access mask available (`NtCurrentProcess()` always has that). `SourceHandle` is the handle to duplicate, and the result is returned in `TargetHandle` (if successful).

`DesiredAccess` can be zero if `Options` includes the flag `DUPLICATE_SAME_ACCESS`; otherwise, it specifies the required access mask (if that can be obtained). If the access mask requested is a subset of the original access mask, this always succeed.

`HandleAttributes` can include the following optional flags:

- `OBJ_INHERIT` (2) - duplicated handle is marked as inheritable.
- `OBJ_PROTECT_CLOSE` (1) - duplicated handle cannot be closed until this flag is removed.

`Options` can be zero or a combination of the following:

- `DUPLICATE_SAME_ACCESS` (2) - as mentioned, uses the access mask of the source handle
- `DUPLICATE_CLOSE_SOURCE` (1) - the source handle is closed after duplication (essentially "moving" the handle from the source to the target process). If this flag is used, the source process handle and the target handle pointer can be `NULL`, which causes the source handle to be closed without any duplication actually occurring.

The target process gets the duplicated handle, but if the target process is not the caller process, it doesn't "know" that a new handle has been created in its handle table. This means that the duplicating process must somehow tell the target process about the new handle value (presumabely, these two processes would like to share access to the object). In such a case there must be some form of *Interprocess Communication* (IPC) mechanism in place between the caller process and the target process.

7.4.2: Querying Information

A general way to get information about an object (through a handle) and the handle entry itself is `NtQueryObject`, repeated here for convenience:

```
typedef enum _OBJECT_INFORMATION_CLASS {
    ObjectBasicInformation,          // OBJECT_BASIC_INFORMATION
    ObjectNameInformation,           // OBJECT_NAME_INFORMATION
    ObjectTypeInformation,           // OBJECT_TYPE_INFORMATION
    ObjectTypesInformation,          // OBJECT_TYPES_INFORMATION
    ObjectHandleFlagInformation,     // OBJECT_HANDLE_FLAG_INFORMATION
    ObjectSessionInformation,
    ObjectSessionObjectInformation,
```

```
    MaxObjectInfoClass
} OBJECT_INFORMATION_CLASS;

NTSTATUS NtQueryObject(
    _In_opt_ HANDLE Handle,
    _In_ OBJECT_INFORMATION_CLASS ObjectInformationClass,
    _Out_writes_bytes_opt_(ObjectInformationLength) PVOID ObjectInformation,
    _In_ ULONG ObjectInformationLength,
    _Out_opt_ PULONG ReturnLength);
```

7.4.2.1: Basic Information

`ObjectBasicInformation` returns an `OBJECT_BASIC_INFORMATION`:

```
typedef struct _OBJECT_BASIC_INFORMATION {
    ULONG Attributes;
    ACCESS_MASK GrantedAccess;
    ULONG HandleCount;
    ULONG PointerCount;
    ULONG PagedPoolCharge;
    ULONG NonPagedPoolCharge;
    ULONG Reserved[3];
    ULONG NameInfoSize;
    ULONG TypeInfoSize;
    ULONG SecurityDescriptorSize;
    LARGE_INTEGER CreationTime;
} OBJECT_BASIC_INFORMATION, *POBJECT_BASIC_INFORMATION;
```

Some of the members provide information on the handle entry maintained in the process handle table, and some details are for the object "pointed to" by the handle. Here is a description of the members:

- `Attributes` includes the handle and object attributes. It could be zero, or a combination of the following flags:
  - `OBJ_PROTECT_CLOSE` (1) - handle is not closable (unless this flag is removed).
  - `OBJ_INHERIT` (2) - handle is inheritable (applies if handle inheritance is used when creating a new process).
  - `OBJ_AUDIT_OBJECT_CLOSE` (4) - an audit event long entry is added to the security event log when the handle is closed.
  - `OBJ_PERMANENT` (0x10) - object is permanent and cannot be destroyed from user mode.
  - `OBJ_EXCLUSIVE` (0x20) - object is exclusive and cannot be opened from another process.
- `GrantedAccess` is the handle's "power" over the object.

- `HandleCount` is the number of open handles to the object.
- `PointerCount` is supposed to be the total reference count (including handle count) for the object. However, starting with Windows 8.1, the pointer count includes an inverse usage count for the handle (see the "Windows Internals" book (part 2) for the gory details. The bottom line is that this value is mostly useless.
- `PagedPoolCharge` and `NonPagedPoolQuota` are the paged and non-paged memory (respectively) used by the object and quotad to the creating process.
- `NameInfoSize` is the size (in bytes) of the object's name information (if any). This includes the size of `OBJECT_NAME_INFORMATION` (see next subsection) and the size of the name itself.
- `TypeInfoSize` is the size (in bytes) of the object's type information. This includes the size of `OBJECT_TYPE_INFORMATION` and the type name's size.
- `SecurityDescriptorSize` is the size (in bytes) of the object's *Security Descriptor* (if any).
- `CreationTime` looks promising, but is only applicable to Symbolic Link objects (indicating their creation time).

> `NameInfoSize` is returned as zero for objects that have no name, but also objects that are nowhere in the Object Manager's namespace (even though they have some form of name). Primary examples are file and desktop objects.

We'll see an example using this information in the next subsection.

7.4.2.2: Object Name

The object's name (if any) can be retrieved using the `ObjectNameInformation` information class. This returns an `OBJECT_NAME_INFORMATION`, which is a glorified `UNICODE_STRING`:

```
typedef struct _OBJECT_NAME_INFORMATION {
    UNICODE_STRING Name;
} OBJECT_NAME_INFORMATION, *POBJECT_NAME_INFORMATION;
```

The catch is that the buffer provided must be large enough to accomodate the name - the API will not allocate it for you. One way to get that information is with `ObjectBasicInformation` from the previous subsection. The following example displays an object's name given a handle and a process ID:

```
std::wstring GetObjectName(HANDLE hObject, ULONG pid) {
    static const WCHAR accessDeniedName[] = L"<access denied>";

    //
    // open handle to target process
    //
    HANDLE hProcess;
    CLIENT_ID cid{ ULongToHandle(pid) };
```

```
OBJECT_ATTRIBUTES procAttr = RTL_CONSTANT_OBJECT_ATTRIBUTES(nullptr, 0);
auto status = NtOpenProcess(&hProcess, PROCESS_DUP_HANDLE, &procAttr, &cid);
if (!NT_SUCCESS(status))
    return accessDeniedName;

HANDLE hDup = nullptr;
std::wstring name;
do {
    //
    // duplicate handle to our process so we can access object
    //
    status = NtDuplicateObject(hProcess, hObject, NtCurrentProcess(),
        &hDup, 0, 0, 0);
    if (!NT_SUCCESS(status)) {
        name = accessDeniedName;
        break;
    }

    //
    // get basic information
    //
    OBJECT_BASIC_INFORMATION info;
    status = NtQueryObject(hDup, ObjectBasicInformation,
        &info, sizeof(info), nullptr);
    if (!NT_SUCCESS(status)) {
        name = accessDeniedName;
        break;
    }

    if (info.NameInfoSize == 0)     // no name
        break;

    //
    // query the actual name
    //
    auto buffer = std::make_unique<BYTE[]>(info.NameInfoSize);
    status = NtQueryObject(hDup, ObjectNameInformation, buffer.get(),
        info.NameInfoSize, nullptr);
    if (!NT_SUCCESS(status))
        break;

    auto uname = (UNICODE_STRING*)buffer.get();
```

```
        name.assign(uname->Buffer, uname->Length / sizeof(WCHAR));
    } while (false);

    if (hProcess)
        NtClose(hProcess);
    if (hDup)
        NtClose(hDup);

    return name;
}
```

The above code retrieves object names for accessible objects (you can run it elevated to gain more access), which are within the object manager's namespace. File objects, for example, are not reported as having a name (`NameInfoSize` is set to zero).

How do we get names of such objects, like files and desktops, which have names, but are not in the hierarchy maintained by the Object Manager's namespace? We could just call `NtQueryObject` with `ObjectNameInformation`, disregarding the fact that `NameInfoSize` is zero.

File objects, in particular, pose an unexpected problem: querying a file object's name may hang the `NtQueryObject` call. The reason has to do with internal synchronization-related operations deep within the Executive's I/O system (file names can change at any time, file systems have their own caches, etc.).

The best approach I have found for this case is to spawn a thread to get the file object name, and if after a predefined time the name is unavailable (call is stuck), terminate the thread and move on. Here is some code that does that:

```
std::wstring GetFileObjectName(HANDLE hFile) {
    //
    // context data to pass to thread
    //
    struct Data {
        std::wstring Name;
        HANDLE hFile;
    } data;
    data.hFile = hFile;

    HANDLE hThread;
    OBJECT_ATTRIBUTES threadAttr = RTL_CONSTANT_OBJECT_ATTRIBUTES(nullptr, 0);

    //
    // create thread
    //
    auto status = RtlCreateUserThread(NtCurrentProcess(), nullptr, FALSE, 0, 0, 0,
        [](auto p) -> NTSTATUS {
```

```
            auto data = (Data*)p;
            //
            // allocate a buffer that should be large enough
            //
            auto buffer = std::make_unique<BYTE[]>(MAX_PATH * 4);
            auto status = NtQueryObject(data->hFile, ObjectNameInformation,
                buffer.get(), MAX_PATH * 4, nullptr);
            if (!NT_SUCCESS(status))
                return 1;

            //
            // extract the name
            //
            auto uname = (UNICODE_STRING*)buffer.get();
            data->Name.assign(uname->Buffer, uname->Length / sizeof(WCHAR));
            return 0;
        }, &data, &hThread, nullptr);
    if (!NT_SUCCESS(status))
        return L"";

    LARGE_INTEGER timeout;
    timeout.QuadPart = -10 * 10000; // 10 msec
    //
    // wait for the thread to terminate
    //
    if (STATUS_TIMEOUT == NtWaitForSingleObject(hThread, FALSE, &timeout)) {
        //
        // terminate forefully
        //
        NtTerminateThread(hThread, 1);
    }
    NtClose(hThread);
    return data.Name;
}
```

To make this work we need to know the object type for which the name is required. Getting this information get be done with `ObjectTypeInformation`, discussed next.

7.4.2.3: Object Type

Using the `ObjectTypeInformation` information class with a given handle returns `OBJECT_TYPE_INFORMATION`, which we looked at earlier in this chapter. Refer to the section *Object Types* for the details.

Write an application that display's an object's name given a handle value and a process ID. Make sure you handle file objects correctly.

7.4.2.4: Handle Information

The `ObjectHandleInformation` returns a simple structure indicating the state of the "protect from close" and "inherit" flags of the given handle:

```
typedef struct _OBJECT_HANDLE_FLAG_INFORMATION {
    BOOLEAN Inherit;
    BOOLEAN ProtectFromClose;
} OBJECT_HANDLE_FLAG_INFORMATION, *POBJECT_HANDLE_FLAG_INFORMATION;
```

As we've seen, these details are also available with `ObjectBasicInformation` as part of the `Attributes` member.

The last two members of `OBJECT_INFORMATION_CLASS` (`ObjectSessionInformation` and `ObjectSessionObjectInformation`) are for setting information only.

7.4.3: Setting Information

A complementary function to `NtQueryObject` exists:

```
NTSTATUS NtSetInformationObject(
    _In_ HANDLE Handle,
    _In_ OBJECT_INFORMATION_CLASS ObjectInformationClass,
    _In_reads_bytes_(ObjectInformationLength) PVOID ObjectInformation,
    _In_ ULONG ObjectInformationLength);
```

A subset of `OBJECT_INFORMATION_CLASS` values are valid for setting information.

`ObjectHandleFlagInformation` can be used to change the state of the "protect from close" and "inherit" flags using the `OBJECT_HANDLE_FLAG_INFORMATION` structure (equivalent to the Windows API `SetHandleInformation`).

`ObjectSessionInformation` only applied to Directory objects. It can be used to associate a Directory object to the caller's session. No buffer is needed in this case. However, it does require the *TCB* privilege, normally not given to any user. Services, however, are able to request it if so desired.

`ObjectSessionObjectInformation` does the same thing as "ObjectSessionInformation', but it only works for Directory objects not currently associated with any session. The upside is that no special privilege is needed.

7.5: Enumerating Handles

Tools such as *Process Explorer* know how to show a list of handles in any process. This is done by enumerating the handles, available with the native API. There are two variants for handle enumeration: the first is to enumerate all handles in the system. The second, enumerate handles in a specific process of interest.

To enumerate all handles in the system, a call to `NtQuerySystemInformation` is required with `SystemExtendedHandleInformation`. You'll find `SystemHandleInformation` as well, but this is deprectaed because the results it returns may be truncated. For example, process IDs are returned as `USHORT`, which is not big enough necessarily.

The returned value from `SystemExtendedHandleInformation` involves a to-level structure and then an array of structures, one for each handle in the system:

```
typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX {
    PVOID Object;
    ULONG_PTR UniqueProcessId;
    ULONG_PTR HandleValue;
    ULONG GrantedAccess;
    USHORT CreatorBackTraceIndex;
    USHORT ObjectTypeIndex;
    ULONG HandleAttributes;
    ULONG Reserved;
} SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX, *PSYSTEM_HANDLE_TABLE_ENTRY_INFO_EX;

typedef struct _SYSTEM_HANDLE_INFORMATION_EX {
    ULONG_PTR NumberOfHandles;
    ULONG_PTR Reserved;
    SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX Handles[1];
} SYSTEM_HANDLE_INFORMATION_EX, *PSYSTEM_HANDLE_INFORMATION_EX;
```

The members of `SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX` should be familiar based on previous sections. Note that the name of the object is not provided. Also, `CreatorBackTraceIndex` seems to be always zero.

Using this enumeration requires passing a buffer large enough to handle the returned data. As with previous examples, it's best to start with some size and double it every time it's too small, or use the returned value (but increase it a bit because new handles may be created before the second call). Here is an example:

```
ULONG size = 0;
std::unique_ptr<BYTE[]> buffer;
auto status = STATUS_SUCCESS;
do {
    if(size)
        buffer = std::make_unique<BYTE[]>(size);
    status = NtQuerySystemInformation(SystemExtendedHandleInformation,
        buffer.get(), size, &size);
    if (status == STATUS_INFO_LENGTH_MISMATCH) {
        size += 1 << 12;
        continue;
    }
} while (!NT_SUCCESS(status));

if (!NT_SUCCESS(status)) {
    printf("Error: 0x%X\n", status);
    return 1;
}

auto handles = (SYSTEM_HANDLE_INFORMATION_EX*)buffer.get();
for (ULONG i = 0; i < handles->NumberOfHandles; i++) {
    // do something with handles->Handles[i];
}
```

7.5.1: Process Handle Enumeration

Enumerating handles in a specific process can be done in one of two ways. One way is to use the previous system-wide handle enumeration and simply filter to the required process. The second option is to get handles of a the process of interest by calling `NtQueryInformationProcess` with ProcessHandleInformation‘ (51). This returns the follwing structures:

```
typedef struct _PROCESS_HANDLE_TABLE_ENTRY_INFO {
    HANDLE HandleValue;
    ULONG_PTR HandleCount;
    ULONG_PTR PointerCount;
    ULONG GrantedAccess;
    ULONG ObjectTypeIndex;
    ULONG HandleAttributes;
    ULONG Reserved;
} PROCESS_HANDLE_TABLE_ENTRY_INFO, *PPROCESS_HANDLE_TABLE_ENTRY_INFO;

typedef struct _PROCESS_HANDLE_SNAPSHOT_INFORMATION {
```

```
    ULONG_PTR NumberOfHandles;
    ULONG_PTR Reserved;
    PROCESS_HANDLE_TABLE_ENTRY_INFO Handles[1];
} PROCESS_HANDLE_SNAPSHOT_INFORMATION, *PPROCESS_HANDLE_SNAPSHOT_INFORMATION;
```

Although it seems good enough, there are a couple of issues when using the process-specific call:

- A process handle is required with `PROCESS_QUERY_INFORMATION`, which is not always possible to get. For example, protected processes can never be opened with this access mask.
- The `PROCESS_HANDLE_TABLE_ENTRY_INFO` structure is missing the object's address in kernel space. Although technically useless to user-mode directly, it may still be needed by the caller.

If these two issues are not a problem, then this method can certainly be used.

Tools like *Process Explorer* use the `NtQuerySystemInformation` call so they can enumerate handles for all processes - no need to obtain a handle to any process.

The *Handles* sample project shows how to perform a system-wide handle enumeration. It also shows object names. Feel free to add support for process filtering, show object types, filter based on object types or names, etc.

7.6: Specific Object Types

The native API has functions for creating, opening, and manipulating various kinds of kernel objects. In this section, we'll look at the most common object types that are useful for user-mode code.

7.6.1: Mutex

Mutex objects are used for thread synchronization. A mutex is either free (signaled), or owned by a thread (non-signaled). The wait functions discussed in chapter 6 apply to mutexes (as they do to all dispatcher objects). Creating a named mutex is achieved with the following:

```
NTSTATUS NtCreateMutant(
    _Out_ PHANDLE MutantHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ BOOLEAN InitialOwner);
```

The desired access for mutex creation is typically `MUTEX_ALL_ACCESS`. If the mutex is not named (using `ObjectAttributes`), a new mutex is created. If a name is provided, and the mutex with that name exists - the function fails with `STATUS_OBJECT_NAME_COLLISION`. The name (if specified) must be either a full name (such as "\KernelObjects\MyMutex" or be a relative name, if a Directory handle is provided in the object attributes. If an unnamed mutex is to be created, and no special attributes are required - `ObjectAttributes` can be `NULL`.

Finally, `InitialOwner` indicates whether the created mutex should be initially owned by the calling thread or not.

> The term "mutant" used by the native API is the original name for mutexes. The kernel object type is still called "Mutant" for compatibility reasons. For all practical purposes, these are one and the same.

The following example shows how to create a named mutex in the Directory "KernelObjects" (requires running elevated):

```
HANDLE hMutex;
UNICODE_STRING name;
RtlInitUnicodeString(&name, L"\\KernelObjects\\MySuperMutex");
OBJECT_ATTRIBUTES mutexAttr = RTL_CONSTANT_OBJECT_ATTRIBUTES(&name, 0);
auto status = NtCreateMutant(&hMutex, MUTEX_ALL_ACCESS, &mutexAttr, FALSE);
```

Opening a handle to an existing mutex is done with `NtOpenMutant`:

```
NTSTATUS NtOpenMutant(
    _Out_ PHANDLE MutantHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes);
```

`ObjectAttributes` must store a name for the mutex, otherwise the call fails. The call may also fail for security reasons - access may not be granted to the caller.

The following example opens a handle to the named mutex created above to allow waiting only:

```
HANDLE hMutex;
UNICODE_STRING name;
RtlInitUnicodeString(&name, L"\\KernelObjects\\MySuperMutex");
OBJECT_ATTRIBUTES mutexAttr = RTL_CONSTANT_OBJECT_ATTRIBUTES(&name, 0);
auto status = NtOpenMutant(&hMutex, SYNCHRONIZATION, &mutexAttr);
```

Once a mutex is successfully acquired, the owner thread must eventually release it:

```
NTSTATUS NtReleaseMutant(
    _In_ HANDLE MutantHandle,
    _Out_opt_ PLONG PreviousCount);
```

When a mutex is created unowned, it's internal count is once. If it's created owened, its count is zero. Each acuisition drops the count by one, and each release increments the count by one.

A mutex may be acquired recusrively more than once by the same thread. The number of waits must match the number of `NtRelaseMutant` calls to actually free the mutex. The optional `PreviousCount` returns the previous count of ownership. `NtReleaseMutant` fails if the caller thread is not the owner of the mutex.

Finally, a mutex state can be queried with `NtQueryMutant`:

```
typedef enum _MUTANT_INFORMATION_CLASS {
    MutantBasicInformation,
    MutantOwnerInformation
} MUTANT_INFORMATION_CLASS;

typedef struct _MUTANT_BASIC_INFORMATION {
    LONG CurrentCount;
    BOOLEAN OwnedByCaller;
    BOOLEAN AbandonedState;
} MUTANT_BASIC_INFORMATION, *PMUTANT_BASIC_INFORMATION;

typedef struct _MUTANT_OWNER_INFORMATION {
    CLIENT_ID ClientId;
} MUTANT_OWNER_INFORMATION, *PMUTANT_OWNER_INFORMATION;

NTSTATUS NtQueryMutant(
    _In_ HANDLE MutantHandle,
    _In_ MUTANT_INFORMATION_CLASS MutantInformationClass,
    _Out_writes_bytes_(MutantInformationLength) PVOID MutantInformation,
    _In_ ULONG MutantInformationLength,
    _Out_opt_ PULONG ReturnLength);
```

The members are mostly self-explanatory. `AbandonedState` indicates if the mutex is *abandoned* - a thread owned this mutex and terminated without releasing it. Since only the mutex owner can release a mutex, the kernel steps in and releases the mutex forcefully upon the thread's termination and changes the mutex state to abandoned - it's the same as free in the sense that the next thread to acquire it will do so, but the abandoned state is an indication of that abnormal situation that may hint some bug in the application. Once the mutex is owned again, its abandoned state is removed (the wait function returns `STATUS_ABANDONED` rather than `STATUS_SUCCESS`)

> Remember that querying the state of dispatcher objects (like mutexes) is not a safe operation, in the sense that its state may change immediately after the call by another thread; it's still useful for debugging purposes.

7.6.2: Sempahore

Sempahore objects maintain a maximum and a current count, initailized at creation time. A semaphore is signaled while its current count is above zero. Once it drops to zero, it becomes non-signaled. Every successful wait decrements its current count. Creating and opening a semaphore object are similar to a mutex:

```
NTSTATUS NtCreateSemaphore(
    _Out_ PHANDLE SemaphoreHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ LONG InitialCount,
    _In_ LONG MaximumCount);

NTSTATUS NtOpenSemaphore(
    _Out_ PHANDLE SemaphoreHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes);
```

> The purpose of a typical semaphore is to limit the size of something given that multiple threads work with that "something" (e.g. a queue).

Acquiring one of the semaphore's counts is done by calling a waiting function. Releasing a semaphore is done with `NtReleaseSemaphore`.

```
NTSTATUS NtReleaseSemaphore(
    _In_ HANDLE SemaphoreHandle,
    _In_ LONG ReleaseCount,
    _Out_opt_ PLONG PreviousCount);
```

A thread may release more than one "count" at a time (`ReleaseCount`). The optional `PreviousCount` returns the previous count of the semaphore before the current operation.

Just like a mutex, a semaphore's state may be queried (mostly for debugging purposes):

```
typedef enum _SEMAPHORE_INFORMATION_CLASS {
    SemaphoreBasicInformation
} SEMAPHORE_INFORMATION_CLASS;

typedef struct _SEMAPHORE_BASIC_INFORMATION {
    LONG CurrentCount;
    LONG MaximumCount;
} SEMAPHORE_BASIC_INFORMATION, *PSEMAPHORE_BASIC_INFORMATION;

NTSTATUS NtQuerySemaphore(
    _In_ HANDLE SemaphoreHandle,
    _In_ SEMAPHORE_INFORMATION_CLASS SemaphoreInformationClass,
    _Out_writes_bytes_(SemaphoreInformationLength) PVOID SemaphoreInformation,
    _In_ ULONG SemaphoreInformationLength,
    _Out_opt_ PULONG ReturnLength);
```

7.6.3: Event

Event objects are simple flags, that may be in the signaled state (set, `TRUE`) or non-signaled (reset, `FALSE`). However, there are two types of events:

- Notification (*Manual reset* in Windows API terminology)
- Synchronization (*Auto reset* in Windows API terminology)

A notification event releases any number of threads when set, while a synchronization event releases just one thread when set and goes back to the non-signaled (reset) state immediately; it must be set again to release another thread.

> If, at the time of setting a synchronization event, no thread is waiting on it, it will remain signaled until a thread waits, which would release it immediately and go to the non-signaled state.

Creating and opening an event is similar to mutexes and semaphores:

```
typedef enum _EVENT_TYPE {
    NotificationEvent,
    SynchronizationEvent
} EVENT_TYPE;

NTSTATUS NtCreateEvent(
    _Out_ PHANDLE EventHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ EVENT_TYPE EventType,
    _In_ BOOLEAN InitialState);

NTSTATUS NtOpenEvent(
    _Out_ PHANDLE EventHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes);
```

Setting and reseting an event is done with the following:

```
NTSTATUS NtSetEvent(
    _In_ HANDLE EventHandle,
    _Out_opt_ PLONG PreviousState);

NTSTATUS NtClearEvent(_In_ HANDLE EventHandle);
NTSTATUS NtResetEvent(
    _In_ HANDLE EventHandle,
    _Out_opt_ PLONG PreviousState);
```

`NtClearEvent` and `NtResetEvent` do the same thing, but the latter returns the previous state of the event (can be zero or one).

Another API is available for setting an event:

```
NTSTATUS NtSetEventBoostPriority(_In_ HANDLE EventHandle);
```

This function works only on synchronization events and increases a released thread to the priority of the current thread (if it's higher). The standard behavior of `NtSetEvent` is to provide a boost of +1 to released threads.

Yet another option is `NtPulseEvent`, having the same parameters as `NtSetEvent`. It sets and immediately resets the event. It has potential issues, though (search online), and thus is not recommended.

Finally, an event's information can be queried:

```
typedef enum _EVENT_INFORMATION_CLASS {
    EventBasicInformation
} EVENT_INFORMATION_CLASS;

typedef struct _EVENT_BASIC_INFORMATION {
    EVENT_TYPE EventType;
    LONG EventState;
} EVENT_BASIC_INFORMATION, *PEVENT_BASIC_INFORMATION;

NTSTATUS NtQueryEvent(
    _In_ HANDLE EventHandle,
    _In_ EVENT_INFORMATION_CLASS EventInformationClass,
    _Out_writes_bytes_(EventInformationLength) PVOID EventInformation,
    _In_ ULONG EventInformationLength,
    _Out_opt_ PULONG ReturnLength);
```

7.7: Other Object Types

Many other object types are supported by the native API shown in table 7-1 with common functions.

Table 7-1: Other object types

Object Type	Example APIs
Timer	`NtCreateTimer`, `NtOpenTimer`, `NtSetTimer(Ex)`, `NtCancelTimer`
TpWorkerFactory	`NtCreateWorkerFactory`, `NtWorkerFactoryWorkerReady`
Profile	`NtCreateProfile`, `NtStartProfile`, `NtStopProfile`
KeyedEvent	`NtCreateKeyedEvent`, `NtOpenKeyedEvent`, `NtWaitForKeyedEvent`
Transaction	`NtCreateTransaction`, `NtOpenTransaction`, `NtCommitTransaction`
Job	`NtCreateJobObject`, `NtOpenJobObject`, `NtSetInformationJobObject`

These object types and others will be covered in a future edition of the book.

7.8: Summary

Kernel objects wrap the core functionality in Windows including processes, threads, jobs, mutexes and many others. This chapter showed how to work in general with objects and handles, and focused on some useful object types, more of which we'll meet in future chapters.

Chapter 8: Memory (Part 1)

Memory is obviously crucial to any computer system. The Windows executive Memory Manager component is responsible for managing the various data structures required to allow the CPU to correctly translate virtual addresses to physical ones (RAM) by managing a set of page tables. Memory (from the CPU and Memory Manager's perspective) is always managed in chunks called *pages*.

In this chapter:

- **Introduction**
- **The *Virtual* Functions**
- **Querying Memory**
- **Reading and Writing**
- **Other Virtual APIs**
- **Heaps**
- **Heap Information**

8.1: Introduction

Table 8-1 shows the page sizes for each architecture supported by Windows. The default page size is 4 KB.

Table 8-1: Page Sizes

Architecture	Normal (Small)	Large	Huge	Remarks
x86	4 KB	2 MB	N/A	Not supported on Windows 11+
x64	4 KB	2 MB	1 GB	
ARM	4 KB	4 MB	N/A	Not supported on Windows 11+
ARM64	4 KB	2 MB	1 GB	

In a simplified manner, the term *Virtual Memory* means the following:

- There is some mapping from a virtual address to a physical address that is transparently followed by the processor. Code accesses memory using virtual addresses only.
- The page may not be in physical memory at all, in which case the CPU raises a *page fault* exception, handled by the Memory Manager. If the page happens to be in a file (such as a *page file*), the Memory Manager will allocate a free physical page, read the data in, fix the page tables and tell the processor to try again. This is completely transparent to the calling code.

8.1.1: Page States

Every page in a virtual address space of a process or the kernel can be in the following states:

- Free - there is nothing there, no mapping. Accessing such a page causes an Access Violation exception.
- Committed - the opposite of free. A page that is definitely accessible, even though it may reside in a file at various times. It may still be inaccessible because of page protection conflicts.
- Reserved - same as free from an access perspective - there is nothing there. However, the page has been reserved for some future use, and will not be considered for new allocations.

8.1.2: Memory APIs

There are several layers of memory APIs in user-mode. At the lowest layer are the so called *Virtual* functions. These are the most powerful, as they are closest to the Memory Manager. They provide all the power of the Memory Manager, such as committing and reserving memory, changing protection, using large pages, etc. Their downside is that they always work in chunks of pages. For example, if you allocate 100 bytes with one of these functions, you will get 4 KB (size of a page); if 5 KB are allocated, 8 KB will be returned.

The API layers are depicted in figure 8-1.

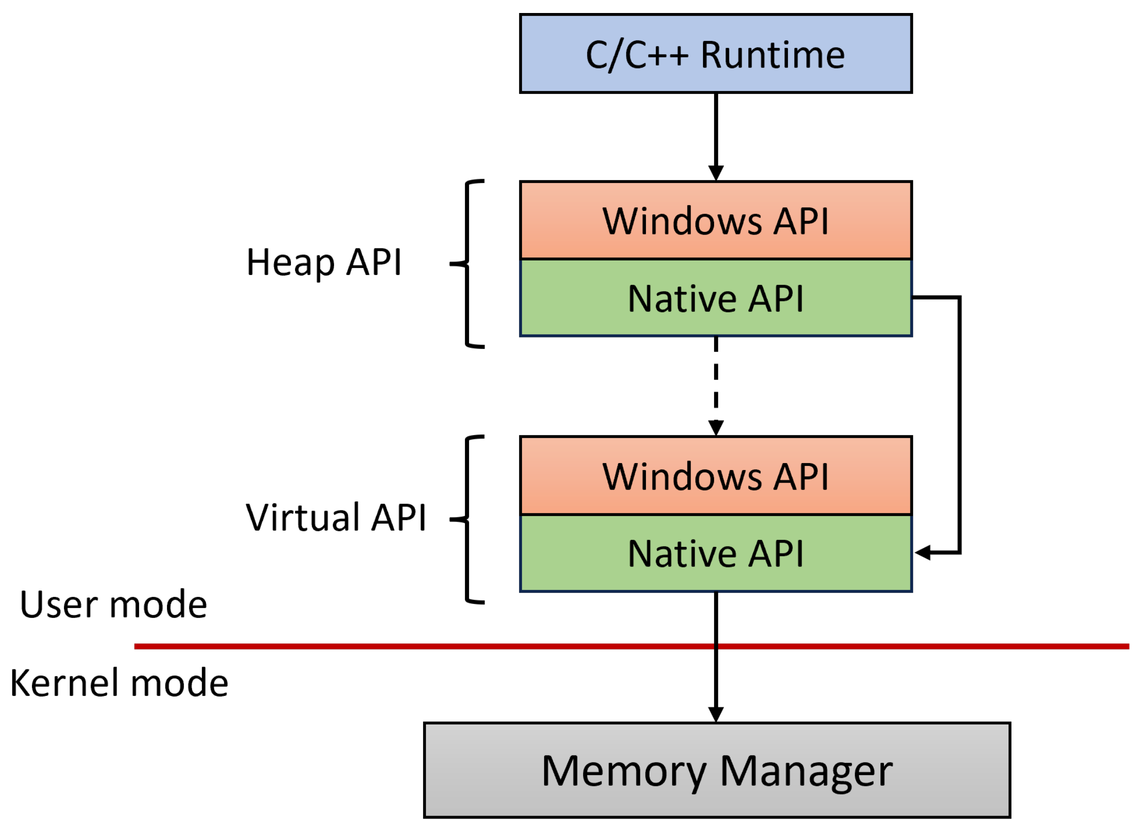


Figure 8-1: Memory API Layers

On top of the *Virtual* API is the *Heap* API. The purpose of this API is to manage small or non-page sized chunks efficiently. They call into the *Virtual* APIs when needed. Finally, the highest level of APIs are part of the C/C++ runtime - functions such as `malloc`, `free`, `operator new`, and similar. Their implementation is compiler-dependent, but Microsoft's compiler currently uses the heap API (working with the default process heap).

8.2: The *Virtual* Functions

The Windows API variants include `VirtualAlloc(Ex)`, `VirtualFree`, `VirtualProtect`, and others. In this section, we'll look at the underlying native APIs, starting with the basic allocation function, `NtAllocateVirtualMemory`:

```
NTSTATUS NtAllocateVirtualMemory(
    _In_ HANDLE ProcessHandle,
    _Inout_ PVOID *BaseAddress,
    _In_ ULONG_PTR ZeroBits,
    _Inout_ PSIZE_T RegionSize,
    _In_ ULONG AllocationType,
    _In_ ULONG Protect);
```

This function is actually documented in the WDK, and most of its parameters correspond to similar ones in the Windows API `VirtualAllocEx`. Here is a brief description:

- `ProcessHandle` is the process where the allocation should be made. The typical case is `NtCurrentProcess`, but it's possible to make an allocate in a different process if the handle has the `PROCESS_VM_OPERATION` access mask.
- `BaseAddress` provides the address to use. For new allocations, `*BaseAddress` is typically set to `NULL`, indicating to the Memory Manager there is no preferred address. If the region in question is reserved, then `*BaseAddress` can specify an address within this region for (say) committing memory. On a successful return of this function, `*BaseAddresd*` indicates the address actually used, which may be rounded down to the nearest page boundary from its input value, or if `NULL` was specified on input - returns the actual address selected by the Memory Manager.
- `ZeroBits` is typically zero, but can be used to affect the final address selected. See the description in chapter 6.
- `*RegionSize` is the size of the region affected, in bytes. It may be rounded up (depending on '*BaseAddress*) so as to align on page boundary.
- `AllocationType` is a set of flags (some mutually exclusive) indicating the operation to perform. The most common ones are:

  - `MEM_COMMIT` - commit the region.
  - `MEM_RESERVE` - reserve the region.
  - `MEM_COMMIT | MEM_RESERVED` - reserve and commit at the same time. Used for new allocations.

- `Protect` is the protection for the allocated pages, e.g. `PAGE_READWRITE`, `PAGE_READONLY`, `PAGE_-EXECUTE_READWRITE`, etc.

See the documentation of `VirtualAlloc` in the Windows SDK for more details on these parameters.

The inverse function to `NtAllocateVirtualMemory` is `NtFreeVirtualMemory`:

```
NTSTATUS NtFreeVirtualMemory(
    _In_ HANDLE ProcessHandle,
    _Inout_ PVOID *BaseAddress,
    _Inout_ PSIZE_T RegionSize,
    _In_ ULONG FreeType);
```

The first 3 parameters are the same as in `NtAllocateVirtualMemory`. However, `*BaseAddress` cannot be `NULL` - it must specify a valid address to de-commit or release (opposite of reserve). The address and region size will be rounded down and up, respectively, so the range is page aligned. Finally, the most common flags follow:

- `MEM_DECOMMIT` - de-commits the pages, making them reserved.
- `MEM_RELEASE` - releases the pages, making them free. The region size must be zero, and `*BaseAddress` must be the base address returned in the initial `NtAllocateVirtualMemory` allocation.

You can find more in the SDK documentation of `VirtualFree`.

8.3: Querying Memory

The `NtQueryVirtualMemory` API provides information on a region of virtual address space in a given process:

```
typedef enum _MEMORY_INFORMATION_CLASS {
    MemoryBasicInformation,           // MEMORY_BASIC_INFORMATION
    MemoryWorkingSetInformation,      // MEMORY_WORKING_SET_INFORMATION
    MemoryMappedFilenameInformation, // UNICODE_STRING
    MemoryRegionInformation,          // MEMORY_REGION_INFORMATION
    MemoryWorkingSetExInformation,    // MEMORY_WORKING_SET_EX_INFORMATION
    MemorySharedCommitInformation,    // MEMORY_SHARED_COMMIT_INFORMATION
    MemoryImageInformation,           // MEMORY_IMAGE_INFORMATION
    MemoryRegionInformationEx,        // MEMORY_REGION_INFORMATION
    MemoryPrivilegedBasicInformation, // MEMORY_BASIC_INFORMATION
    MemoryEnclaveImageInformation,    // MEMORY_ENCLAVE_IMAGE_INFORMATION
    MemoryBasicInformationCapped,     // ARM64 only
```

```
    MemoryPhysicalContiguityInformation, // MEMORY_PHYSICAL_CONTIGUITY_INFORMATION
    MemoryBadInformation,            // MEMORY_BAD_IDENTITY_INFORMATION
    MemoryBadInformationAllProcesses, // MEMORY_BAD_IDENTITY_INFORMATION
    MaxMemoryInfoClass
} MEMORY_INFORMATION_CLASS;

NTSTATUS NtQueryVirtualMemory(
    _In_ HANDLE ProcessHandle,
    _In_opt_ PVOID BaseAddress,
    _In_ MEMORY_INFORMATION_CLASS MemoryInformationClass,
    _Out_writes_bytes_(MemoryInformationLength) PVOID MemoryInformation,
    _In_ SIZE_T MemoryInformationLength,
    _Out_opt_ PSIZE_T ReturnLength);
```

As you can see, the function supports several information classes, described below.

8.3.1: MemoryBasicInformation (0)

The buffer is of type `MEMORY_BASIC_INFORMATION`, documented in the Windows SDK. In fact, using this information class is equivalent to the Windows API `VirtualQueryEx`. It returns information about a region of address space that has the same attributes in terms of page state, mapping type, and page protection. The access mask needed for the process handle is `PROCESS_QUERY_LIMITED_INFORMATION` or `PROCESS_QUERY_INFORMATION`. Consult the documentation of `VirtualQueryEx` for the details.

At the time of this writing, the documentation states that `PROCESS_QUERY_INFORMATION` access mask is needed, but this is incorrect.

This is the basic operation of the *Sysinternals VMMap* tool.

The following example shows how to scan a given process address:

```
void QueryVM(HANDLE hProcess) {
    MEMORY_BASIC_INFORMATION mbi;
    BYTE* address = nullptr;

    while (NT_SUCCESS(NtQueryVirtualMemory(hProcess, address,
        MemoryBasicInformation, &mbi, sizeof(mbi), nullptr))) {
        DisplayMemoryInfo(mbi);
        address += mbi.RegionSize;
    }
}
```

`NtQueryVirtualMemory` fails eventually when the address goes beyond the user-mode address space. `DisplayMemoryInfo` is a local function showing details of a `MEMORY_BASIC_INFORMATION` structure:

```
void DisplayMemoryInfo(MEMORY_BASIC_INFORMATION const& mi) {
    printf("%p %16zX %-10s %-8s %-15s %-15s\n",
        mi.BaseAddress, mi.RegionSize,
        StateToString(mi.State), TypeToString(mi.Type),
        ProtectionToString(mi.Protect).c_str(),
        ProtectionToString(mi.AllocationProtect).c_str());
}
```

The helper functions convert from numeric values to strings:

```
const char* StateToString(DWORD state) {
    switch (state) {
        case MEM_FREE: return "Free";
        case MEM_COMMIT: return "Committed";
        case MEM_RESERVE: return "Reserved";
    }
    return "";
}

const char* TypeToString(DWORD type) {
    switch (type) {
        case MEM_IMAGE: return "Image";
        case MEM_MAPPED: return "Mapped";
        case MEM_PRIVATE: return "Private";
    }
    return "";
}

std::string ProtectionToString(DWORD protect) {
    std::string text;
    if (protect & PAGE_GUARD) {
        text += "Guard/";
        protect &= ~PAGE_GUARD;
    }
    if (protect & PAGE_WRITECOMBINE) {
        text += "Write Combine/";
        protect &= ~PAGE_WRITECOMBINE;
    }
```

```
    switch (protect) {
        case 0: break;
        case PAGE_NOACCESS: text += "No Access"; break;
        case PAGE_READONLY: text += "RO"; break;
        case PAGE_READWRITE: text += "RW"; break;
        case PAGE_EXECUTE_READWRITE: text += "RWX"; break;
        case PAGE_WRITECOPY: text += "Write Copy"; break;
        case PAGE_EXECUTE: text += "Execute"; break;
        case PAGE_EXECUTE_READ: text += "RX"; break;
        case PAGE_EXECUTE_WRITECOPY: text += "Execute/Write Copy"; break;
        default: text += "<other>";
    }
    return text;
}
```

Here is a (very) trimmed-down output looking at an *Explorer.exe* process on my system:

```
0000000000000000    380000 Free                No Access
0000000000380000     10000 Committed  Mapped   RW                RW
0000000000390000      3000 Committed  Mapped   RO                RO
0000000000393000      D000 Free                No Access
00000000003A0000     1F000 Committed  Mapped   RO                RO
00000000003BF000      1000 Free                No Access
00000000003C0000      4000 Committed  Mapped   RO                RO
00000000003C4000      C000 Free                No Access
00000000003D0000      3000 Committed  Mapped   RO                RO
00000000003D3000      D000 Free                No Access
00000000003E0000      2000 Committed  Private  RW                RW
00000000003E2000      E000 Free                No Access
00000000003F0000      3000 Committed  Mapped   RO                RO
00000000003F3000      D000 Free                No Access
0000000000400000      2000 Committed  Private  RW                RW
0000000000402000     1E000 Reserved   Private                    RW
...
0000000000600000     6F000 Reserved   Private                    RW
000000000066F000      3000 Committed  Private  Guard/RW          RW
0000000000672000      E000 Committed  Private  RW                RW
0000000000680000     11000 Committed  Mapped   RO                RO
0000000000691000      F000 Free                No Access
00000000006A0000     11000 Committed  Mapped   RO                RO
...
00007FF8A392F000     14000 Committed  Image    RO                Execute/Write Copy
```

```
00007FF8A3943000      1000 Committed  Image    RW              Execute/Write Copy
00007FF8A3944000      5000 Committed  Image    RO              Execute/Write Copy
00007FFFDB164000      5000 Committed  Image    RO              Execute/Write Copy
00007FFFDB169000      1000 Committed  Image    RW              Execute/Write Copy
00007FFFDB16A000      4000 Committed  Image    RO              Execute/Write Copy
00007FFFDB16E000  1B742000 Free                No Access
00007FFFF68B0000      1000 Committed  Image    RO              Execute/Write Copy
00007FFFF68B1000      4000 Committed  Image    RX              Execute/Write Copy
00007FFFF68B5000      2000 Committed  Image    RO              Execute/Write Copy
00007FFFF68B7000      1000 Committed  Image    RW              Execute/Write Copy
00007FFFF68B8000      3000 Committed  Image    RO              Execute/Write Copy
00007FFFF68BB000     1A5000 Free                No Access
00007FFFF6A60000      1000 Committed  Image    RO              Execute/Write Copy
00007FFFF6A61000      5000 Committed  Image    RX              Execute/Write Copy
00007FFFF6A66000      3000 Committed  Image    RO              Execute/Write Copy
00007FFFF6A69000      1000 Committed  Image    RW              Execute/Write Copy
00007FFFF6A6A000      4000 Committed  Image    RO              Execute/Write Copy
00007FFFF6A6E000   9582000 Free                No Access
```

The full code is in the *QueryVM* project.

8.3.2: `MemoryWorkingSetInformation` (1)

This information class is about a process *Working Set* - memory that can be accessed without generating a page fault. The process handle must have the `PROCESS_QUERY_INFORMATION` access mask.

The data structures involved are shown below:

```
typedef struct _MEMORY_WORKING_SET_BLOCK {
    ULONG_PTR Protection : 5;
    ULONG_PTR ShareCount : 3;
    ULONG_PTR Shared : 1;
    ULONG_PTR Node : 3;
#ifdef _WIN64
    ULONG_PTR VirtualPage : 52;
#else
    ULONG VirtualPage : 20;
#endif
} MEMORY_WORKING_SET_BLOCK, *PMEMORY_WORKING_SET_BLOCK;
```

```
typedef struct _MEMORY_WORKING_SET_INFORMATION {
    ULONG_PTR NumberOfEntries;
    MEMORY_WORKING_SET_BLOCK WorkingSetInfo[1];
} MEMORY_WORKING_SET_INFORMATION, *PMEMORY_WORKING_SET_INFORMATION;
```

The main structure is just a container for entries describing working set ranges in the given process. The `BaseAddress` value has no effect. This also means the size of the provided data is not fixed, and may fail with `STATUS_INFO_LENGTH_MISMATCH`, in which case the buffer should be enlarged and the call repeated.

The members of `MEMORY_WORKING_SET_BLOCK` are described below:

- `Protection` describes the region protection. The values are not the same as the standard protection constants we've seen in the previous subsection.
- `ShareCount` holds the share count of this region (if shared); the maximum value is 7.
- `Shared` is set if this region can be shared with other processes.
- `Node` is the NUMA node where this physical memory page is located.
- `VirtualPage` is the virtual page number (shift 12 bits to the left to get to the actual address).

> The functionality provided by this information class is the same as with the `QueryWorkingSet` Windows API. Consult the SDK docs for more information on the data members.

The following example shows how to output all working set pages given a handle to a process:

```
bool QueryWS(HANDLE hProcess) {
    SIZE_T size = 1 << 17;      // arbitrary
    std::unique_ptr<BYTE[]> buffer;
    NTSTATUS status;

    do {
        buffer = std::make_unique<BYTE[]>(size);
        status = NtQueryVirtualMemory(hProcess, nullptr,
                MemoryWorkingSetInformation, buffer.get(), size, nullptr);
        if(NT_SUCCESS(status))
            break;
        size *= 2;  // size too small
    } while (status == STATUS_INFO_LENGTH_MISMATCH);

    if (!NT_SUCCESS(status))
        return false;

    auto ws = (MEMORY_WORKING_SET_INFORMATION*)buffer.get();
```

```
    for (ULONG_PTR i = 0; i < ws->NumberOfEntries; i++) {
        //
        // get single item
        //
        auto& info = ws->WorkingSetInfo[i];

        //
        // show some details
        //
        printf("%16zX Prot: %-17s Share: %d Shareable: %s Node: %d\n",
            info.VirtualPage << 12, ProtectionToString(info.Protection),
            (int)info.ShareCount, (int)info.Shared ? "Y" : "N", (int)info.Node);
    }
    return true;
}
```

The full code is in the *QueryWS* project.

8.3.3: `MemoryMappedFilenameInformation` (2)

This information class returns a `UNICODE_STRING` for a mapped file in a given committed memory region (if any). Teh following example returns a `std::wstring` for a mapped file in a given address:

```
std::wstring Details(HANDLE hProcess, void* address) {
    BYTE buffer[1 << 10];
    if (NT_SUCCESS(NtQueryVirtualMemory(hProcess, address,
        MemoryMappedFilenameInformation, buffer, sizeof(buffer), nullptr))) {
        auto us = (PUNICODE_STRING)buffer;
        return std::wstring(us->Buffer, us->Length / sizeof(WCHAR));
    }
    return L"";
}
```

The process handle must have `PROCESS_QUERY_LIMITED_INFORMATION` or `PROCESS_QUERY_INFORMATION` access mask. The address used cannot contain private memory, which means that the type of the page where the address points to must be `MEM_MAPPED` or `MEM_IMAGE`. Querying a `MEM_PRIVATE` or non-committed pages will fail, so best to save time and avoid such regions.

A complete example is in the *QueryVM* sample in the `Details` function.

8.3.4: `MemoryRegionInformation` (3) and `MemoryRegionInformationEx` (7)

These information classes provide additional details for a given allocation range. Both return a `MEMORY_REGION_INFORMATION` defined below:

```
typedef struct _MEMORY_REGION_INFORMATION {
    PVOID AllocationBase;
    ULONG AllocationProtect;
    union {
        ULONG RegionType;
        struct {
            ULONG Private : 1;
            ULONG MappedDataFile : 1;
            ULONG MappedImage : 1;
            ULONG MappedPageFile : 1;
            ULONG MappedPhysical : 1;
            ULONG DirectMapped : 1;
            ULONG SoftwareEnclave : 1;
            ULONG PageSize64K : 1;
            ULONG PlaceholderReservation : 1;
            ULONG MappedWriteWatch : 1;
            ULONG PageSizeLarge : 1;
            ULONG PageSizeHuge : 1;
            ULONG Reserved : 19;
        };
    };
    SIZE_T RegionSize;
    SIZE_T CommitSize;
    ULONG_PTR PartitionId;
    ULONG_PTR NodePreference;
} MEMORY_REGION_INFORMATION, *PMEMORY_REGION_INFORMATION;
```

`MemoryRegionInformation` can be called with a structure that excludes the last two members; it's best to use the extended version.

Here is a description of the members:

- `AllocationBase` is the base of the initial allocation, which could be different from the provided *BaseAddress* in the call if *BaseAddress* is the beginning of the allocation block.
- `AllocationProtect` is the protection given to the memory block in the initial allocation. Note that the current protection could be different - in fact, it could be anything for each page of this allocation region.

This also explains why `MEMORY_BASIC_INFORMATION` has `AllocationProtect` member (initial protection for the entire region) and `Protect` - the current protection for this (possibly subset) of the region.

The flags unified with `RegionType` are the following:

- `Private` is set if the memory region is private to the process (not shared.
- `MappedDataFile` is set if the region maps to a file as data (not as a PE image).
- `MappedImage` is set if the region is mapped to a PE image.
- `MappedPageFile` is set if this region is mapped to a page file.
- `MappedPhysical` is set if the region is mapped to non-paged memory (such as with large pages).
- `DirectMapped` is set if the region is mapped by a kernel driver to a device memory.
- `SoftwareEnclave` is set if the region is mapped to a *memory enclave* (such as *Virtualization Based Security* (VBS) protected enclave).
- `PageSize64K` is set if the region is mapped with 64 KB page size (possible when working with *section* objects).
- `PlaceholderReservation` is set if the region is reserved by the kernel.
- `MappedWriteWatch` is set if the memory is registered for write watch (see `InitializeProcessForWsWatch` Windows API or `NtSetInformationProcess` with `ProcessWorkingSetWatch`).
- `PageSizeLarge` is set if the region is mapped using large pages.
- `PageSizeHuge` is set if the region is mapped with huge pages.

The remaining members are:

- `RegionSize` is the size of the region in question.
- `CommitSize` is the size of the committed part of the region.
- `PartitionId` is the memory partition this region belongs to (normally zero).
- `NodePreference` is the NUMA node preference for this region. It's `(ULONG)-1` if there is no preference.

The process handle required must have either `PROCESS_QUERY_LIMITED_INFORMATION` or `PROCESS_QUERY_INFORMATION` access mask bits present.

The *QueryVM* sample has been extended to include region information as follows (`Details` function):

```
if (mbi.State == MEM_COMMIT) {
    MEMORY_REGION_INFORMATION ri;
    if (NT_SUCCESS(NtQueryVirtualMemory(hProcess, mbi.BaseAddress,
        MemoryRegionInformationEx, &ri, sizeof(ri), nullptr))) {
        details += MemoryRegionFlagsToString(ri.RegionType);
    }
```

The helper `MemoryRegionFlagsToString` builds a string based on the set flags:

```
std::wstring MemoryRegionFlagsToString(ULONG flags) {
    std::wstring result;
    static PCWSTR text[] = {
        L"Private", L"Data File", L"Image", L"Page File",
        L"Physical", L"Direct", L"Enclave", L"64KB Page",
        L"Placeholder Reserve", L"Write Watch",
        L"Large Page", L"Huge Page",
    };

    for(int i = 0; i < _countof(text); i++)
        if (flags & (1 << i)) {
            result += text[i];
            result += L", ";
        }

    if (!result.empty())
        return result.substr(0, result.length() - 2);
    return L"";
}
```

8.3.5: MemoryWorkingSetExInformation (4)

This information class is an extended version of `MemoryWorkingSetInformation` in terms of the details returned, but its usage is different. Instead of providing working set information for the entire process, the caller selects the virtual addresses it's interested in (could be just one) passed as part of the output buffer. The buffers are updated with the attributes of the address(es) provided. Each item is of type `MEMORY_WORKING_SET_EX_INFORMATION` that contains a `MEMORY_WORKING_SET_EX_BLOCK`:

```
typedef struct _MEMORY_WORKING_SET_EX_BLOCK {
    union {
        struct {
            ULONG_PTR Valid : 1;
            ULONG_PTR ShareCount : 3;
            ULONG_PTR Win32Protection : 11;
            ULONG_PTR Shared : 1;
            ULONG_PTR Node : 6;
            ULONG_PTR Locked : 1;
            ULONG_PTR LargePage : 1;
            ULONG_PTR Priority : 3;
            ULONG_PTR Reserved : 3;
            ULONG_PTR SharedOriginal : 1;
            ULONG_PTR Bad : 1;
```

```
#ifdef _WIN64
            ULONG_PTR Win32GraphicsProtection : 4;
            ULONG_PTR ReservedUlong : 28;
#endif
        };
        struct {
            ULONG_PTR Valid : 1;
            ULONG_PTR Reserved0 : 14;
            ULONG_PTR Shared : 1;
            ULONG_PTR Reserved1 : 5;
            ULONG_PTR PageTable : 1;
            ULONG_PTR Location : 2;
            ULONG_PTR Priority : 3;
            ULONG_PTR ModifiedList : 1;
            ULONG_PTR Reserved2 : 2;
            ULONG_PTR SharedOriginal : 1;
            ULONG_PTR Bad : 1;
#ifdef _WIN64
            ULONG_PTR ReservedUlong : 32;
#endif
        } Invalid;
    };
} MEMORY_WORKING_SET_EX_BLOCK, *PMEMORY_WORKING_SET_EX_BLOCK;

typedef struct _MEMORY_WORKING_SET_EX_INFORMATION {
    PVOID VirtualAddress;
    union {
        MEMORY_WORKING_SET_EX_BLOCK VirtualAttributes;
        ULONG_PTR Long;
    } u1;
} MEMORY_WORKING_SET_EX_INFORMATION, *PMEMORY_WORKING_SET_EX_INFORMATION;
```

`VirtualAddress` is the input address which must be set prior to the call. The adjacent `VirtualAttributes` is the result. `VirtualAttributes` has two parts of a `union`:

- If the address is valid (committed and present in RAM), the first structure should be consulted (`Valid` is 1).
- If the address is invalid, the second structure (`Invalid`) should be consulted. In this case, `Invalid.Valid` is zero.

Some of the flags were present in `MEMORY_WORKING_SET_BLOCK`. Here is a description of those which were not:

- `Valid` is set if the virtual address is valid, and clear otherwise.
- `Win32Protection` is the protection for this address, given in the standard protection constants (`PAGE_READWRITE`, `PAGE_READONLY`, etc.).
- `Locked` is set if the address is locked in physical memory.
- `Priority` is the page priority (0 to 7, 7 is the highest), which determines how likely this page is to remain in RAM if RAM is needed. See the "Windows Internals" book in chapter 5 for more details.
- `SharedOriginal` is set if the page is backed by a prototype PTE.
- `Bad` is set if the page is physically defective, or if it's part of an enclave.
- `PageTable` is set if this page stores a page table (not applicable to a user-mode address).
- `Location` is set to one of three values: 0 (invalid page), 1 (resident), 2 (page file).
- `ModifiedList` is set if the page is on the Modified page list maintained by the Memory Manager.
- `Win32GraphicsProtection` is set with Win32 protection flags if that memory is used by a graphics card.

Using `MemoryWorkingSetExInformation` requires the caller's token to have the *SeProfileSingleProcessPrivilege*, normally given to administrators.

The *QueryVM* project includes code that displays most of this information if the caller has the required privilege (see the `Details` function).

8.3.6: `MemorySharedCommitInformation` (5)

This information just returns the shared committed memory size in the given process (as `SIZE_T`), in pages (4KB chunks). The following example shows the shared committed memory in KB:

```
SIZE_T committed;
if (NT_SUCCESS(NtQueryVirtualMemory(hProcess, nullptr,
    MemorySharedCommitInformation, &committed, sizeof(committed), nullptr))) {
    printf("Shared commit: %llu KB\n", committed << 2);
}
```

This information class requires a `PROCSS_QUERY_LIMITED_INFORMATION` or `PROCESS_QUERY_INFORMATION` access mask. The `BaseAddress` value is not used.

8.3.7: `MemoryImageInformation` (6)

This information class gives details about an image that is mapped to the provided address (if any). The returned structure is the following:

```
typedef struct _MEMORY_IMAGE_INFORMATION {
    PVOID ImageBase;
    SIZE_T SizeOfImage;
    union {
        ULONG ImageFlags;
        struct {
            ULONG ImagePartialMap : 1;
            ULONG ImageNotExecutable : 1;
            ULONG ImageSigningLevel : 4;
            ULONG Reserved : 26;
        };
    };
} MEMORY_IMAGE_INFORMATION, *PMEMORY_IMAGE_INFORMATION;
```

`ImageBase` is the virtual address the image is mapped to in the process. `SizeOfImage` is the size of the image in bytes. ImagePartialMap`is set if only part of the image is mapped within this page.`ImageNotExecutable`is set if the mapping do not include execute permissions. Finally,`ImageSigningLevel‘ is one value from the following definitions found in <*WinNt.h*>:

```
#define SE_SIGNING_LEVEL_UNCHECKED       0x00000000
#define SE_SIGNING_LEVEL_UNSIGNED        0x00000001
#define SE_SIGNING_LEVEL_ENTERPRISE      0x00000002
#define SE_SIGNING_LEVEL_CUSTOM_1        0x00000003
#define SE_SIGNING_LEVEL_DEVELOPER       SE_SIGNING_LEVEL_CUSTOM_1
#define SE_SIGNING_LEVEL_AUTHENTICODE    0x00000004
#define SE_SIGNING_LEVEL_CUSTOM_2        0x00000005
#define SE_SIGNING_LEVEL_STORE           0x00000006
#define SE_SIGNING_LEVEL_CUSTOM_3        0x00000007
#define SE_SIGNING_LEVEL_ANTIMALWARE     SE_SIGNING_LEVEL_CUSTOM_3
#define SE_SIGNING_LEVEL_MICROSOFT       0x00000008
#define SE_SIGNING_LEVEL_CUSTOM_4        0x00000009
#define SE_SIGNING_LEVEL_CUSTOM_5        0x0000000A
#define SE_SIGNING_LEVEL_DYNAMIC_CODEGEN 0x0000000B
#define SE_SIGNING_LEVEL_WINDOWS         0x0000000C
#define SE_SIGNING_LEVEL_CUSTOM_7        0x0000000D
#define SE_SIGNING_LEVEL_WINDOWS_TCB     0x0000000E
#define SE_SIGNING_LEVEL_CUSTOM_6        0x0000000F
```

8.3.8: `MemoryPrivilegedBasicInformation` (8)

This information class provides the same details as `MemoryBasicInformation`, but applies to “secure” (*Isolated User Mode* - IUM) processes that are protected using *Virtualization Based Security* (VBS), such as

LsaIso.exe (if Credential Guard is running on the system). The secure kernel is the one providing the results, at its complete discretion.

For normal processes, it behaves the same as `MemoryBasicInformation`.

8.3.9: `MemoryEnclaveImageInformation` (9)

This information class provides information on images stored in enclaves. It doesn't seem to be accessible from user-mode.

8.3.10: `MemoryBasicInformationCapped` (10)

This information class is the same as `MemoryBasicInformation`, but only supported on ARM64. It doesn't seem to be much different from `MemoryBasicInformation`, except the region size provided determines the maximum address range even if rounding down occurrs.

8.3.11: `MemoryPhysicalContiguityInformation` (11)

This information class returns physical memory information for a range of addresses as a `MEMORY_PHYSICAL_CONTIGUITY_INFORMATION` structure:

```
typedef enum _MEMORY_PHYSICAL_CONTIGUITY_UNIT_STATE {
    MemoryNotContiguous,
    MemoryAlignedAndContiguous,
    MemoryNotResident,
    MemoryNotEligibleToMakeContiguous,
} MEMORY_PHYSICAL_CONTIGUITY_UNIT_STATE;

typedef struct _MEMORY_PHYSICAL_CONTIGUITY_UNIT_INFORMATION {
    union {
        ULONG AllInformation;
        struct {
            ULONG State : 2;    // MEMORY_PHYSICAL_CONTIGUITY_UNIT_STATE
            ULONG Reserved : 30;
        };
    };
} MEMORY_PHYSICAL_CONTIGUITY_UNIT_INFORMATION;

typedef struct _MEMORY_PHYSICAL_CONTIGUITY_INFORMATION {
    PVOID VirtualAddress;
    ULONG_PTR Size;
    ULONG_PTR ContiguityUnitSize;
    ULONG Flags;
```

```
    PMEMORY_PHYSICAL_CONTIGUITY_UNIT_INFORMATION ContiguityUnitInformation;
} MEMORY_PHYSICAL_CONTIGUITY_INFORMATION, *PMEMORY_PHYSICAL_CONTIGUITY_INFORMATION;
```

The call requires the *SeProfileSingleProcessPrivilege* to be in the caller's token.

On input, `VirtualAddress` must be set to the address being queried (must be aligned to `ContiguityUnitSize` - typically a page), `Size` must be set to the size of the range, which must be a multiple of `ContiguityUnitSize`. On return, the `State` member is set to the appropriate value. `Flags` can be zero or one (if one, the query is for valid pages only).

Unfortunately, I couldn't get this to work - `STATUS_INVALID_PARAMETER` was always returned.

8.3.12: `MemoryBadInformation` (12) and `MemoryBadInformationAllProcesses` (12)

`MemoryBadInformation` returns a process' bad pages list - pages that are either physically defective, or are used for memory enclaves. `BaseAddress` must be set to `NULL`, and the returned buffer contains an array of the following structure:

```
typedef struct _MEMORY_BAD_IDENTITY_INFORMATION {
    union {
        PVOID VirtualAddress;      // virtual memory
        ULONG_PTR PageFrameIndex;  // physical memory
    };

    union {
        struct {
            ULONG_PTR Poisoned : 1;    // should not be accessed
            ULONG_PTR Physical : 1;
        };
        ULONG_PTR AllInformation;
    };
} MEMORY_BAD_IDENTITY_INFORMATION, *PMEMORY_BAD_IDENTITY_INFORMATION;
```

The `Physical` flag indicates whether `PageFrameIndex` should be used (if set), or `VirtualAddress` (if clear).

`MemoryBadInformationAllProcesses` provides similar information for all processes, but is not accessible from user-mode.

8.4: Reading and Writing

Reading and writing memory in the current process does not require any special APIs - just use C-style memory functions (e.g. `memcpy`, `memset`) and normal code to read and write.

Reading and writing in another process address space requires the following APIs:

```
NTSTATUS NtReadVirtualMemory(
    _In_ HANDLE ProcessHandle,
    _In_opt_ PVOID BaseAddress,
    _Out_writes_bytes_(BufferSize) PVOID Buffer,
    _In_ SIZE_T BufferSize,
    _Out_opt_ PSIZE_T NumberOfBytesRead);

NTSTATUS NtWriteVirtualMemory(
    _In_ HANDLE ProcessHandle,
    _In_opt_ PVOID BaseAddress,
    _In_reads_bytes_(BufferSize) PVOID Buffer,
    _In_ SIZE_T BufferSize,
    _Out_opt_ PSIZE_T NumberOfBytesWritten);
```

These functions are the ones being invoked by the Windows API functions `ReadProcessMemory` and `WriteProcessMemory`. The parameters should be mostly self-explanatory; `BaseAddress` is the virtual address in the target process. `ProcessHandle` must have `PROCESS_VM_READ` access mask (for `NtReadVirtualMemory`), or `PROCESS_VM_WRITE` (for `NtWriteVirtualMemory`).

8.4.1: Injecting a DLL with Remote Thread

A "classic" example of writing to another process memory is injecting a DLL into a target process by creating a thread in that process, and pointing it to a `LoadLibrary` function in that process. Here is an example of how this could be done using the Windows API (error handling ommitted for brevity):

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

    auto pid = atoi(argv[1]);
    HANDLE hProcess = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_OPERATION |
        PROCESS_CREATE_THREAD, FALSE, pid);
    auto p = VirtualAllocEx(hProcess, nullptr, 1 << 12, MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE);
    WriteProcessMemory(hProcess, p, argv[2], strlen(argv[2]) + 1, nullptr);
    auto hThread = CreateRemoteThread(hProcess, nullptr, 0,
        (LPTHREAD_START_ROUTINE)GetProcAddress(
        GetModuleHandle(L"kernel32"), "LoadLibraryA"),
        p, 0, nullptr);
```

```
    // wait for the remote thread to exit
    WaitForSingleObject(hThread, INFINITE);

    // optionally, be nice: free the memory in the target process
    VirtualFreeEx(hProcess, p, 0, MEM_RELEASE);

    CloseHandle(hThread);
    CloseHandle(hProcess);

    return 0;
}
```

Let's see how we can create a "native" version of this technique. First we'll accept the process ID and DLL path just like in the Windows API version, but we'll switch to Unicode:

```
int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 3) {
        printf("Usage: InjectDllRemoteThread <pid> <dllpath>\n");
        return 0;
    }
```

This is not mandatory, but we'll use `LoadLibraryW` as the target function for the remote thread to execute, and in that case simplifies the code.

The next step is to open a handle to the target process with the required access mask:

```
HANDLE hProcess;
CLIENT_ID cid{ ULongToHandle(wcstol(argv[1], nullptr, 0)) };
OBJECT_ATTRIBUTES procAttr = RTL_CONSTANT_OBJECT_ATTRIBUTES(nullptr, 0);
auto status = NtOpenProcess(&hProcess,
    PROCESS_VM_WRITE | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD,
    &procAttr, &cid);
if(!NT_SUCCESS(status)) {
    printf("Error opening process (status: 0x%X)\n", status);
    return status;
}
```

Next, we need to allocate memory in the target process and write the full DLL path into that memory. First, the allocation:

```
PVOID buffer = nullptr;
SIZE_T size = 1 << 12;  // 4 KB should be enough
status = NtAllocateVirtualMemory(hProcess, &buffer, 0, &size,
    MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
if (!NT_SUCCESS(status)) {
    printf("Error allocating memory (status: 0x%X)\n", status);
    return status;
}
```

Next, we copy the DLL path to be loaded into the allocated remote buffer:

```
status = NtWriteVirtualMemory(hProcess, buffer,
    (PVOID)argv[2], sizeof(WCHAR) * (wcslen(argv[2]) + 1), nullptr);
if (!NT_SUCCESS(status)) {
    printf("Error writing to process (status: 0x%X)\n", status);
    return status;
}
```

Now we need to get the module handle for *kernel32* from which we can locate `LoadLibraryW`:

```
UNICODE_STRING kernel32Name;
RtlInitUnicodeString(&kernel32Name, L"kernel32");
PVOID hK32Dll;
status = LdrGetDllHandle(nullptr, nullptr, &kernel32Name, &hK32Dll);
if (!NT_SUCCESS(status)) {
    printf("Error getting kernel32 module (status: 0x%X)\n", status);
    return status;
}
```

`LdrGetDllHandle` is a rough equivalent to the Windows API `GetModuleHandle` function. Now we can locate `LoadLibraryW`:

```
ANSI_STRING fname;
RtlInitAnsiString(&fname, "LoadLibraryW");
PVOID pLoadLibrary;
status = LdrGetProcedureAddress(hK32Dll, &fname, 0, &pLoadLibrary);
if (!NT_SUCCESS(status)) {
    printf("Error locating LoadLibraryW (status: 0x%X)\n", status);
    return status;
}
```

You may be wondering why are we not going "all native" and using `LdrLoadDll` (the equivalent of `LoadLibrary`). The reason is that a thread function accepts one argument only, and that argument must be the DLL path. Unfortunately, `LdrLoadDll` accepts the DLL path in its third parameter, which we have no good way of providing:

```
NTSTATUS LdrLoadDll(
    _In_opt_ PWSTR DllPath,
    _In_opt_ PULONG DllCharacteristics,
    _In_ PUNICODE_STRING DllName,    // DLL path
    _Out_ PVOID *DllHandle);
```

For now we'll stick with `LoadLibraryW` which accepts a single parameter - the DLL path to load.

The final step is creating the actual thread to execute `LoadLibraryW` and pass the argument to be the allocated buffer in the target process:

```
HANDLE hThread;
status = RtlCreateUserThread(hProcess, nullptr, FALSE, 0, 0, 0,
    (PUSER_THREAD_START_ROUTINE)pLoadLibrary, buffer,
    &hThread, nullptr);
if (!NT_SUCCESS(status)) {
    printf("Error creating thread (status: 0x%X)\n", status);
    return status;
}
printf("Success!\n");
```

If this succeeds, then the thread is executing and will load the DLL based on its path. Of course that could fail if the path is bad, for example. From the injector's perspective all that's left is to clean up:

```
NtWaitForSingleObject(hThread, FALSE, nullptr);
size = 0;
NtFreeVirtualMemory(hProcess, &buffer, &size, MEM_RELEASE);

NtClose(hThread);
NtClose(hProcess);
```

Here is the complete main function (*InjectDllRemoteThread* project) with error handling removed:

```
int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 3) {
        printf("Usage: InjectDllRemoteThread <pid> <dllpath>\n");
        return 0;
    }

    HANDLE hProcess;
    CLIENT_ID cid{ ULongToHandle(wcstol(argv[1], nullptr, 0)) };
    OBJECT_ATTRIBUTES procAttr = RTL_CONSTANT_OBJECT_ATTRIBUTES(nullptr, 0);
    auto status = NtOpenProcess(&hProcess,
        PROCESS_VM_WRITE | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD,
        &procAttr, &cid);

    PVOID buffer = nullptr;
    SIZE_T size = 1 << 12;  // 4 KB should be more than enough
    status = NtAllocateVirtualMemory(hProcess, &buffer, 0, &size,
        MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

    status = NtWriteVirtualMemory(hProcess, buffer,
        (PVOID)argv[2], sizeof(WCHAR) * (wcslen(argv[2]) + 1), nullptr);

    UNICODE_STRING kernel32Name;
    RtlInitUnicodeString(&kernel32Name, L"kernel32");
    PVOID hK32Dll;
    status = LdrGetDllHandle(nullptr, nullptr, &kernel32Name, &hK32Dll);

    ANSI_STRING fname;
    RtlInitAnsiString(&fname, "LoadLibraryW");
    PVOID pLoadLibrary;
    status = LdrGetProcedureAddress(hK32Dll, &fname, 0, &pLoadLibrary);

    HANDLE hThread;
    status = RtlCreateUserThread(hProcess, nullptr, FALSE, 0, 0, 0,
        (PUSER_THREAD_START_ROUTINE)pLoadLibrary, buffer,
        &hThread, nullptr);

    printf("Success!\n");

    NtWaitForSingleObject(hThread, FALSE, nullptr);
    size = 0;
    NtFreeVirtualMemory(hProcess, &buffer, &size, MEM_RELEASE);
```

```
    NtClose(hThread);
    NtClose(hProcess);
    return 0;
}
```

8.5: Other Virtual APIs

In this section we'll examine some more APIs from the "Virtual" family.

The initial page protection for newly allocated region is set at `NtAllocateVirtualMemory` time. If memory protection needs to be changed, `NtProtectVirtualMemory` can be used:

```
NTSTATUS NtProtectVirtualMemory(
    _In_ HANDLE ProcessHandle,
    _Inout_ PVOID *BaseAddress,
    _Inout_ PSIZE_T RegionSize,
    _In_ ULONG NewProtect,
    _Out_ PULONG OldProtect);
```

Most of the parameters should be familiar by now. `ProcessHandle` can be `NtCurrentProcess()`, changing protection in the caller's process (always succeeds). If a different process is used, the handle must have the `PROCESS_VM_OPERATION` access mask. `NewProtect` is the requested protection using the same constants used with the Windows API which we met earlier (`PAGE_READWRITE`, `PAGE_READONLY`, etc.). The returned `OldProtect` contains the previous protection of the first page in the region.

This function is invoked by the Windows API `VirtualProtect(Ex)` APIs.

Memory can be "locked" into physical memory, prevent it from being paged out until "unlocked". The following APIs can be used:

```
NTSTATUS NtLockVirtualMemory(
    _In_ HANDLE ProcessHandle,
    _Inout_ PVOID *BaseAddress,
    _Inout_ PSIZE_T RegionSize,
    _In_ ULONG MapType);
NTSTATUS NtUnlockVirtualMemory(
    _In_ HANDLE ProcessHandle,
    _Inout_ PVOID *BaseAddress,
    _Inout_ PSIZE_T RegionSize,
    _In_ ULONG MapType);
```

`MapType` can be one of two values: `MAP_PROCESS` (1) or `MAP_SYSTEM` (2). If `MAP_SYSTEM` is specified, then the caller must have the *SeLockMemoryPrivilege* in its token. There doesn't seem to be a difference in operation between these two flags - but one must be specified.

The Windows API `VirtualLock` and `VirtualUnlock` use `MAP_PROCESS`.

Locking memory tells the memory manager to keep these pages in the process working set as long as there are threads running (not waiting) in the process.

8.6: Heaps

The "Virtual" APIs always work in chunks of pages (4 KB), which is not what's needed for most applications. Heaps provide fine grained memory management, on top of the Virtual APIs. The Heap Manager is the entity responsible for managing memory within heaps.

The Windows API has a bunch of functions related to heaps, such as `HeapAlloc`, `HeapFree`, and more. These are thin layers on top of the native heap APIs.

Every process starts with one heap known as the *Process Default Heap* (created by *NtDll* as part of process initialization). This heap can be retrieved with `RtlProcessHeap`:

```
#define RtlProcessHeap() (NtCurrentPeb()->ProcessHeap)
```

More heaps can be created, which could be useful for several reasons, briefly discussed later.

The Heap Manager is also implemented within the kernel, and some of the functions we'll look at next are documented in the WDK.

8.6.1: Basic Heap Management

Allocation from a heap is accomplished with `RtlAllocateHeap`:

```
PVOID RtlAllocateHeap(
    _In_ PVOID HeapHandle,
    _In_opt_ ULONG Flags,
    _In_ SIZE_T Size);
```

`HeapHandle` is a handle to a heap (`RtlProcessHeap()` is a valid value). `Flags` can be zero or a combination of the following values:

- `HEAP_NO_SERIALIZE` (1) - the allocation should not be synchronized with possibly other threads accessing the same heap.

- `HEAP_ZERO_MEMORY` (8) - the returned buffer is zero-initialized (without this flag, its contents are unchanged).
- `HEAP_GENERATE_EXCEPTION` (4) - if the allocation fails, `NULL` is normally returned. With this flag specified, a `STATUS_NO_MEMORY` exception is raised instead.
- `HEAP_SETTABLE_USER_VALUE` (0x100) - allows setting a user-defined value associated with the allocation (see next section for details).

A "missing" flag does not mean the opposite of the flag; it means that the heap defaults are used. For example, if `HEAP_NO_SERIALIZE` is not specified, the allocation may still be unsynchronized if the heap was created with that flag.

The default process heap is synchronized.

`Size` is the allocation size in bytes. The return value is the pointer to the allocated block, or `NULL` on failure (unless `HEAP_GENERATE_EXCEPTION` is specified in `Flags`, in which case an exception is raised on failure).

Once the memory is no longer needed, `RtlFreeHeap` can be called to free the block:

```
BOOLEAN RtlFreeHeap(
    _In_ PVOID HeapHandle,
    _In_opt_ ULONG Flags,
    _Frees_ptr_opt_ PVOID BaseAddress);
```

The only valid flag is `HEAP_NO_SERIALIZE`. `BaseAddress` must be an address previously returned from `RtlAllocateHeap`.

If an allocated block needs to be resized, `RtlReAllocateHeap` can be used:

```
PVOID RtlReAllocateHeap(
    _In_ PVOID HeapHandle,
    _In_ ULONG Flags,
    _Frees_ptr_opt_ PVOID BaseAddress,
    _In_ SIZE_T Size);
```

The new block can be larger or smaller than the existing block, specified with `BaseAddress`. If a larger block is requested, and cannot be specified within the existing block, the data is copied to the new block. Even with a smaller allocation, there is no guarantee the same block is returned, which means the return value is the one pointing to the block (whether moved or not).

The `Flags` accepted are the same as for `RtlAllocateHeap`.

8.6.2: Creating Heaps

Heaps can be created in the process. Here are some possible reasons to do so and not just use the default process heap:

- The default process heap is serialized, which reduces performance if there is no contention on the heap. Creating a separate heap provides control over synchronization. In general, creating a heap provides customization opportunities.
- A heap may become fragmented over time if different size blocks are requested a lot. One possible solution is to create a heap that caters to fixed sized allocations if these are common in the application.
- Bugs accessing the heap (such as heap corruptions) are easier to find if these happen accessing a specific heap.
- It's sometimes easier to destroy a heap in one swoop rather than freeing all individual allocations.

Windows supports two types of heaps. The first, referred to as the *NT Heap* is the original heap management implemented. Windows 8 introduced a new type of heap, *Segment Heap*, that provides a better implementation in terms of block management and security. Still, normal processes by default use the *NT Heap* because of compatibility concerns. UWP applications and certain system processes use the Segment heap.

System process image names that use the segment heap include: *Svchost.exe*, *Smss.exe*, *Csrss.exe*, *Lsass.exe*, *Services.exe*, *RuntimeBroker.exe*, *dwm.exe*, *WinInit.exe*, *WinLogon.exe*, *MsMpEng.exe*, *NisSrv.exe*, *SiHost.exe*, and others.

An executable can opt-in to use the segment heap by adding a value named *FrontEndHeapDebugOptions* to the *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\exename* key, with a value of 8 to use the segment heap, or a value of 4 to disable the segment heap.

`RtlHeapCreate` is the heap creation function, documented in the WDK (with a kernel bias):

```
PVOID RtlCreateHeap(
    _In_ ULONG Flags,
    _In_opt_ PVOID HeapBase,
    _In_opt_ SIZE_T ReserveSize,
    _In_opt_ SIZE_T CommitSize,
    _In_opt_ PVOID Lock,
    _In_opt_ PRTL_HEAP_PARAMETERS Parameters);
```

`Flags` can be zero or a combination of the following flags:

- `HEAP_NO_SERIALIZE` (1) - indicates the heap is not synchronized by default.
- `HEAP_GENERATE_EXCEPTION` (4) - the allocation function raise `STATUS_NO_MEMORY` instead of returning `NULL` on failure.

- `HEAP_GROWABLE` (2) - indicates the heap can grow to any size, limited by the process address space and available memory. This requires `HeapBase` to be set to `NULL`.
- `HEAP_REALLOC_IN_PLACE_ONLY` (0x10) - reallocations can only succeed if there is no need to allocate a new block.
- `HEAP_TAIL_CHECKING_ENABLED` (0x20) - the heap functions perform an additional check that ensures the process did not use more bytes than it was provided. This is done by filling a few extra bytes beyond an allocation request with a known fill pattern, and validating it when freeing it or when its size is queried. This is a debugging aid to find issues in the heap before corruptions occur.
- `HEAP_FREE_CHECKING_ENABLED` (0x40) - another debugging heap feature that aims to ensure free operations are operating on correct addresses.
- `HEAP_DISABLE_COALESCE_ON_FREE` (0x80) - does not attempt to automatically coalesce free regions. Such coalescing can be forced with `RtlCompactHeap`.

`HEAP_TAIL_CHECKING_ENABLED`, `HEAP_FREE_CHECKING_ENABLED`, and `HEAP_DISABLE_COALESCE_ON_FREE` can also be specified as NT Global Flags, and if so, are applied automatically to newly created heaps whether explicitly specified or not.

- `HEAP_CREATE_ALIGN_16` (0x10000) - indicates memory blocks should be 16-bytes aligned.
- `HEAP_CREATE_ENABLE_TRACING` (0x20000) - enables heap tracing.
- `HEAP_CREATE_ENABLE_EXECUTE` (0x40000) - indicates allocations should have protection `PAGE_EXECUTE_READWRITE`, meaning code can execute from the heap without causing an exception.
- `HEAP_CREATE_SEGMENT_HEAP` (0x100) - indicates a Segment heap should be created rather than an NT heap. For a segment heap, `HEAP_GROWABLE` must be specified as well. In addition, `HeapBase`, `ReservedSize`, `CommitSize`, and `Lock` must be zero/`NULL`.

> The various "debug" flags (e.g., `HEAP_TAIL_CHECKING_ENABLED`, `HEAP_FREE_CHECKING_ENABLED`) cause allocation functions to be slower; only use for debugging purposes.

`BaseAddress` is either `NULL` (`HEAP_GROWABLE` must be specified as flag), in which case the heap manager allocates a new region of memory to serve the heap, or if non-`NULL`, the address is used as the basis for the new heap. That address must have been allocated previously by the process (e.g., with `NtAllocateVirtualMemory`).

If `ReservedSize` is non-zero, it indicates the initial amount of memory to reserve for the heap, rounded up to pages. `CommitSize` indicates the initial committed size of the heap. The relationship of `ReservedSize` and `CommittedSize` is described in table 8-2.

Table 8-2: 'CommitSize' and 'ReservedSize'

`ReservedSize`	`CommitSize`	Result
Zero	Zero	64 pages are initially reserved for the heap, one page committed
Zero	non-Zero	`ReservedSize` is set to `CommitSize` and rounds to nearest 64KB size
non-Zero	Zero	One page is initially committed on the heap

ReservedSize	CommitSize	Result
non-Zero	non-Zero	CommitSize is reduced to ReservedSize if it's greater

Next, the Lock parameter is an optional pointer to an initialized CRITICAL_SECTION structure to serve as the lock to use to synchronize heap access. If the HEAP_NO_SERIALIZE flag is specified, Lock must be NULL, as the intent is to prevent heap serialization.

Finally, Parameters is an optional pointer to more customization options with this structure:

```
typedef NTSTATUS (NTAPI *PRTL_HEAP_COMMIT_ROUTINE)(
    _In_ PVOID Base,
    _Inout_ PVOID *CommitAddress,
    _Inout_ PSIZE_T CommitSize);

typedef struct _RTL_HEAP_PARAMETERS {
    ULONG Length;   // sizeof(RTL_HEAP_PARAMATERS)
    SIZE_T SegmentReserve;
    SIZE_T SegmentCommit;
    SIZE_T DeCommitFreeBlockThreshold;
    SIZE_T DeCommitTotalFreeThreshold;
    SIZE_T MaximumAllocationSize;
    SIZE_T VirtualMemoryThreshold;
    SIZE_T InitialCommit;
    SIZE_T InitialReserve;
    PRTL_HEAP_COMMIT_ROUTINE CommitRoutine;
    SIZE_T Reserved[2];
} RTL_HEAP_PARAMETERS, *PRTL_HEAP_PARAMETERS;
```

Table 8-3 summarizes the role of the members of RTL_HEAP_PARAMETERS.

Table 8-3: 'RTL_HEAP_PARAMETERS'

Member	Description	Value if zero specified
Length	must be set to the size of the structure before use	Must be specified
SegmentReserve	Reserved segment size when heap is expanded	PEB->HeapSegmentReserve
SegmentCommit	Commit segment size more committed memory is needed	PEB->HeapSegmentCommit
DeCommitFreeBlockThreshold	Block threshold to decommit when memory is freed	PEB->HeapDeCommitFreeBlockThreshold
DeCommitTotalFreeThreshold	Threshold to decommit all heap memory	PEB->DeCommitTotalFreeThreshold

Member	Description	Value if zero specified
`MaximumAllocationSize`	Maximum size allowed in a single allocation	One page less than the size of the process address space
`VirtualMemoryThreshold`	Used to decide size of internally created arrays	`0x7f000`
`InitialCommit`	Same meaning as `CommitSize` parameter, overrides it of none zero	Ignored
`InitialReserve`	Same meaning as `ReservedSize` parameter, overrides it of none zero	Ignored
`CommitRoutine`	Custom routine to perform allocation	Ignored
`Reserved`	Reserved	Must be cleared to zero

The `CommitRoutine` allows customizing allocation of blocks. This is only valid if the heap is not growable, and `BaseAddress` is not `NULL`. In such a case, the routine is invoked by the Heap Manager with the following arguments:

- `Base` is the base address of the heap (`BaseAddress` in `RtlCreateHeap`).
- `CommitAddress` is where the allocated address is to be stored.
- `CommitSize` is the requested size on input, which may be changed by the routine to the actual number of bytes allocated.

The return value from `RtlCreateHeap` is a heap handle. It's not a handle in the kernel object sense; rather, it's an opaque pointer to an undocumented structure. It could be one of two structures: `HEAP` for an NT heap or `SEGMENT_HEAP` for a Segment heap. Although undocumented, these structures are available as part of the public Microsoft symbols.

Here is the output from a WinDbg session attached to a user-mode process (any will do) that shows these two structures (truncated for brevity):

```
0:000> dt ntdll!_HEAP
   +0x000 Segment          : _HEAP_SEGMENT
   +0x000 Entry            : _HEAP_ENTRY
   +0x010 SegmentSignature : Uint4B
   +0x014 SegmentFlags     : Uint4B
   +0x018 SegmentListEntry : _LIST_ENTRY
   +0x028 Heap             : Ptr64 _HEAP
   +0x030 BaseAddress      : Ptr64 Void
   +0x038 NumberOfPages    : Uint4B
   +0x040 FirstEntry       : Ptr64 _HEAP_ENTRY
   +0x048 LastValidEntry   : Ptr64 _HEAP_ENTRY
   +0x050 NumberOfUnCommittedPages : Uint4B
   +0x054 NumberOfUnCommittedRanges : Uint4B
   +0x058 SegmentAllocatorBackTraceIndex : Uint2B
   +0x05a Reserved         : Uint2B
```

```
   +0x060 UCRSegmentList   : _LIST_ENTRY
...
   +0x238 Counters         : _HEAP_COUNTERS
   +0x2b0 TuningParameters : _HEAP_TUNING_PARAMETERS

0:000> dt ntdll!_SEGMENT_HEAP
   +0x000 EnvHandle        : RTL_HP_ENV_HANDLE
   +0x010 Signature        : Uint4B
   +0x014 GlobalFlags      : Uint4B
   +0x018 Interceptor      : Uint4B
   +0x01c ProcessHeapListIndex : Uint2B
   +0x01e AllocatedFromMetadata : Pos 0, 1 Bit
   +0x020 CommitLimitData  : _RTL_HEAP_MEMORY_LIMIT_DATA
   +0x020 ReservedMustBeZero1 : Uint8B
   +0x028 UserContext      : Ptr64 Void
   +0x030 ReservedMustBeZero2 : Uint8B
   +0x038 Spare            : Ptr64 Void
   +0x040 LargeMetadataLock : Uint8B
   +0x048 LargeAllocMetadata : _RTL_RB_TREE
   +0x058 LargeReservedPages : Uint8B
   +0x060 LargeCommittedPages : Uint8B
   +0x068 Tag              : Uint8B
   +0x070 StackTraceInitVar : _RTL_RUN_ONCE
   +0x080 MemStats         : _HEAP_RUNTIME_MEMORY_STATS
   +0x0d8 GlobalLockCount  : Uint2B
   +0x0dc GlobalLockOwner  : Uint4B
   +0x0e0 ContextExtendLock : Uint8B
   +0x0e8 AllocatedBase    : Ptr64 UChar
   +0x0f0 UncommittedBase  : Ptr64 UChar
   +0x0f8 ReservedLimit    : Ptr64 UChar
   +0x100 ReservedRegionEnd : Ptr64 UChar
   +0x108 CallbacksEncoded : _RTL_HP_HEAP_VA_CALLBACKS_ENCODED
   +0x140 SegContexts      : [2] _HEAP_SEG_CONTEXT
   +0x2c0 VsContext        : _HEAP_VS_CONTEXT
   +0x380 LfhContext       : _HEAP_LFH_CONTEXT
```

This means you can cast a heap handle to one of these structures and examine its member if so desired. How can you tell the type of heap given some heap you did not create? You can case to `SEGMENT_HEAP` and examine the `Signature` member. If it's `0xddeeddee` - it's a Segment heap. Here is an example assuming the above structures have been converted to their C declaration:

```
void PrintHeapType(PVOID heap) {
    auto segHeap = (SEGMENT_HEAP*)heap;
    if(segHeap->Signature == 0xddeeddee)
        printf("Segment Heap");
    else {
        auto ntHeap = (HEAP*)heap;
        printf("NT Heap, Base Address: 0x%p", ntHeap->BaseAddress);
    }
}
```

When using WinDbg, you can use the `!heap` command to get details for all heaps in the process or a specific heap of interest.

Once a heap is no longer needed, it should be destroyed:

```
PVOID RtlDestroyHeap(_In_ _Post_invalid_ PVOID HeapHandle);
```

If any allocations remain on the heap, they are "released" as the heap is completely destroyed. The return value is `NULL` on successful destruction, or `HeapHandle` otherwise.

The following example creates a new heap and performs an allocation against it, frees the allocation, and then destroys the heap. Obviously, in a real scenario the created heap would be used for many allocations.

```
auto heap = RtlCreateHeap(HEAP_GROWABLE | HEAP_NO_SERIALIZE,
    nullptr, 0, 0, nullptr, nullptr);
auto buffer = RtlAllocateHeap(heap, HEAP_ZERO_MEMORY, 30);  // 30 bytes

// use buffer...

RtlFreeHeap(heap, 0, buffer);
RtlDestroyHeap(heap);
```

8.6.3: Other Heap Functions

In this section, we'll look at other heap functions.

8.6.3.1: Heap Allocation Size

A heap allocation can be queried for its size:

```
SIZE_T RtlSizeHeap(
    _In_ PVOID HeapHandle,
    _In_ ULONG Flags,
    _In_ PVOID BaseAddress);
```

The returned size in bytes is for the original allocation size specified, and does not include any padding bytes or metadata bytes.

Sometimes for security purposes, it's useful to zero out any freed blocks because normally the data is still there until overwritten. This is the job of `RtlZeroHeap`:

8.6.3.2: Protection

```
NTSTATUS RtlZeroHeap(
    _In_ PVOID HeapHandle,
    _In_ ULONG Flags);
```

The memory of a heap is normally protected with `PAGE_READWRITE` or `PAGE_EXECUTE_READWRITE` (if `HEAP_CREATE_ENABLE_EXECUTE` is specified at heap creation time). It's possible to change the protection to read-only with `RtlProtectHeap`:

```
VOID RtlProtectHeap(
    _In_ PVOID HeapHandle,
    _In_ BOOLEAN MakeReadOnly);
```

It can be reverted later by specifying calling the function again with `MakeReadOnly` set to `FALSE`.

8.6.3.3: Locking

It's possible to lock access to the heap to the current thread, preventing other threads from accessing the heap. This could be more performant than making multiple calls, each causing a lock/unlock internally. It's also necessary if heap walking is attempted, as any changes from other threads will corrupt the state of the walk. The following allow locking and unlocking a heap:

```
BOOLEAN RtlLockHeap(_In_ PVOID HeapHandle);
BOOLEAN RtlUnlockHeap(_In_ PVOID HeapHandle);
```

These work recursively, which means that multiple Lock calls require the same number of Unlock calls to actually unlock the heap for normal access.

8.6.3.4: User Data

It's possible to attach a user-defined value with heap entries. The allocation itself must specify the `HEAP_SETTABLE_USER_VALUE` flag for this to work:

```
BOOLEAN RtlSetUserValueHeap(
    _In_ PVOID HeapHandle,
    _In_ ULONG Flags,
    _In_ PVOID BaseAddress,
    _In_ PVOID UserValue);
```

It's also theoretically possible to set 2 bits of flags by calling `RtlSetUserFlagsHeap`, but this is too fragile as it does not work with the Segment heap nor if the *Low Fragmentation Heap* (LFH) layer is active.

To retrieve the value back given a heap allocation, call `RtlGetUserInfoHeap`, which returns the user-defined value and/or the flags:

```
BOOLEAN RtlGetUserInfoHeap(
    _In_ PVOID HeapHandle,
    _In_ ULONG Flags,
    _In_ PVOID BaseAddress,
    _Out_opt_ PVOID *UserValue,
    _Out_opt_ PULONG UserFlags);
```

Here is an example of setting the value `0x1234` as a user-defined value, and retrieving it back:

```
auto buffer = RtlAllocateHeap(heap, HEAP_SETTABLE_USER_VALUE, 40);
auto ok = RtlSetUserValueHeap(heap, 0, buffer, (PVOID)0x1234);
ok = RtlSetUserFlagsHeap(heap, 0, buffer, 0, RTL_HEAP_SETTABLE_FLAG1);

// read back

PVOID value{ nullptr };
ok = RtlGetUserInfoHeap(heap, 0, buffer, &value, nullptr);
assert(value == (PVOID)0x1234);
```

8.6.3.5: More Allocations

It's possible to make multiple allocations of the same size in a single call:

```
ULONG RtlMultipleAllocateHeap(
    _In_ PVOID HeapHandle,
    _In_ ULONG Flags,
    _In_ SIZE_T Size,
    _In_ ULONG Count,
    _Out_ PVOID* Array);
```

`Size` is the size in bytes for each allocation. `Count` is the number of allocations. `Array` is the returned array of pointers. The return value is `Count` on success. If smaller, it indicates the number of successful allocations.

As you might expect, it's possible to free multiple allocations as well:

```
ULONG RtlMultipleFreeHeap(
    _In_ PVOID HeapHandle,
    _In_ ULONG Flags,
    _In_ ULONG Count,
    _In_ PVOID *Array);
```

8.7: Heap Information

Several native APIs exist to get information about heaps. Let's start with the getting all the heaps in the calling process. There are two ways to go about it - either call `RtlGetProcessHeaps` to get an array of heap handles, or enumerate heaps with a callback using `RtlEnumProcessHeaps`:

```
ULONG RtlGetProcessHeaps(
    _In_ ULONG NumberOfHeaps,
    _Out_ PVOID *ProcessHeaps);

typedef NTSTATUS (NTAPI *PRTL_ENUM_HEAPS_ROUTINE)(
    _In_ PVOID HeapHandle,
    _In_ PVOID Parameter);
NTSTATUS RtlEnumProcessHeaps(
    _In_ PRTL_ENUM_HEAPS_ROUTINE EnumRoutine,
    _In_ PVOID Parameter);
```

`RtlGetProcessHeaps` accepts the number of heap handles to retrieve, and an array to place the results in. It returns the actual number of heaps in the process. This means that if the return value is larger than `NumberOfHeaps`, not all heap handles have been retrieved.

`RtlEnumProcessHeaps` invokes a callback (`EnumRoutine`) for each heap in the process, which makes it easier perhaps, since no prior allocation is necessary nor "guessing" the number of heaps.

8.7.1: Heap Walking

Given a heap handle, it's possible to "walk" the heap, meaning the various blocks the heap is composed of can retrieved, mostly useful for debugging purposes. `RtlWalkHeap` is the function to use, with entries returned as `RTL_HEAP_WALK_ENTRY`:

```
#define RTL_HEAP_BUSY                (USHORT)0x0001
#define RTL_HEAP_SEGMENT             (USHORT)0x0002
#define RTL_HEAP_SETTABLE_VALUE      (USHORT)0x0010
#define RTL_HEAP_SETTABLE_FLAG1      (USHORT)0x0020
#define RTL_HEAP_SETTABLE_FLAG2      (USHORT)0x0040
#define RTL_HEAP_SETTABLE_FLAG3      (USHORT)0x0080
#define RTL_HEAP_SETTABLE_FLAGS      (USHORT)0x00e0
#define RTL_HEAP_UNCOMMITTED_RANGE   (USHORT)0x1000
#define RTL_HEAP_PROTECTED_ENTRY     (USHORT)0x2000
#define RTL_HEAP_LARGE_ALLOC         (USHORT)0x4000
#define RTL_HEAP_LFH_ALLOC           (USHORT)0x8000

typedef struct _RTL_HEAP_WALK_ENTRY {
    PVOID DataAddress;
    SIZE_T DataSize;
    UCHAR OverheadBytes;
    UCHAR SegmentIndex;
    USHORT Flags;   // see flags above
    union {
        struct {
            SIZE_T Settable;
            USHORT TagIndex;
            USHORT AllocatorBackTraceIndex;
            ULONG Reserved[2];
        } Block;
        struct {
            ULONG CommittedSize;
            ULONG UnCommittedSize;
            PVOID FirstEntry;
            PVOID LastEntry;
        } Segment;
    };
} RTL_HEAP_WALK_ENTRY, *PRTL_HEAP_WALK_ENTRY;

NTSTATUS RtlWalkHeap(
    _In_ PVOID HeapHandle,
    _Inout_ PRTL_HEAP_WALK_ENTRY Entry);
```

Heap walking begins with a `RTL_HEAP_WALK_ENTRY` structure where `DataAddress` is set to `NULL`. Subsequent calls should use the last address returned to get the next item. Each returned item is either a block or a segment, based on the flag `RTL_HEAP_SEGMENT` in the `Flags` member. A Segment is a management entity, containing multiple blocks, each block representing a chunk of memory (not to be confused with the

Segment Heap).

A simple walk of the default process heap could be done as follows:

```
RTL_HEAP_WALK_ENTRY entry{};
while (NT_SUCCESS(RtlWalkHeap(RtlProcessHeap(), &entry))) {
    printf("Addr: 0x%p Size: 0x%08X Flags: 0x%04X\n",
        entry.DataAddress, entry.DataSize, (int)entry.Flags);
}
```

`DataSize` is the size of the block or segment. For a block, if busy (used, `RTL_HEAP_BUSY`), it's the size of the allocation handed to a client. The `OverheadBytes` is the extra bytes used to maintain this allocation. `SegmentIndex` is a segment index in this heap, starting with zero.

For a Segment (`RTL_HEAP_SEGMENT` flag set), the `Segment` part of the union has the following members:

- `CommittedSize` is the number of committed bytes in the segment (always multiple of page size).
- `UncommittedSize` is the number of bytes not currently committed (always multiple of page size).
- `FirstEntry` is the first entry in the segment.
- `LastEntry` is the last valid entry in the segment.

For a Block (no `RTL_HEAP_SEGMENT` flag), the following members are available in the `Block` union:

- `Settable` is the value attached to this block (if `RTL_HEAP_SETTABLE_VALUE` flag is set).
- `TagIndex` is the tag index of the block is tagged.
- `AllocatorBackTraceIndex` is used for debugging purposes (outside the scope of this chapter).

8.7.2: More Heap Information

Heap information can be retrieved with `RtlQueryHeapInformation`, and some heap details can be changed with `RtlSetHeapInformation`:

```
NTSTATUS RtlQueryHeapInformation(
    _In_ PVOID HeapHandle,
    _In_ HEAP_INFORMATION_CLASS HeapInformationClass,
    _Out_opt_ PVOID HeapInformation,
    _In_opt_  SIZE_T HeapInformationLength,
    _Out_opt_ PSIZE_T ReturnLength);
NTSTATUS RtlSetHeapInformation(
    _In_ PVOID HeapHandle,
    _In_ HEAP_INFORMATION_CLASS HeapInformationClass,
    _In_opt_ PVOID HeapInformation,
    _In_opt_ SIZE_T HeapInformationLength);
```

The patterns used by these functions is very similar to other query/set functions we've seen. This is the information classes supported (some are defined in *<WinNt.h>*):

```
typedef enum _HEAP_INFORMATION_CLASS {
    HeapCompatibilityInformation        = 0,
    HeapEnableTerminationOnCorruption   = 1
    HeapExtendedInformation             = 2
    HeapOptimizeResources               = 3
    HeapTaggingInformation              = 4,
    HeapStackDatabase                   = 5,
    HeapMemoryLimit                     = 6,
    HeapTag                             = 7,
    HeapDetailedFailureInformation      = (int)0x80000001,
    HeapSetDebuggingInformation         = (int)0x80000002
} HEAP_INFORMATION_CLASS;
```

8.7.3: `HeapCompatibilityInformation` (0)

Querying with this information class returns `ULONG` describing some aspects of the heap in question. For a segment heap, the value is always 2, which means the Low Fragmentation Heap (LFH) layer is active. For a standard heap, this could be zero (LFH not applied) or 2 (LFH applied).

For set operations, LFH can be enabled for standard heaps only. This also requires the heap to be growable and created without `HEAP_NO_SERIALIZE`. There is no need to enable the LFH as it's enabled automatically if the Heap Manager determines it's beneficial. The following enumeration from *phnt* can be used:

```
typedef enum _HEAP_COMPATIBILITY_MODE {
    HEAP_COMPATIBILITY_STANDARD = 0UL,
    HEAP_COMPATIBILITY_LAL = 1UL,    // N/A
    HEAP_COMPATIBILITY_LFH = 2UL,
} HEAP_COMPATIBILITY_MODE;
```

8.7.4: `HeapEnableTerminationOnCorruption` (1)

This information class allows querying/setting the option that if a heap corruption is detected, the process should terminate immediately rather than raise an exception which may be handled in some way, and is unlikely to be fruitful.

The expected type is `ULONG`, with non-zero meaning termination on corruption is enabled. It's currently enabled by default for all heaps.

8.7.5: `HeapExtendedInformation` (2)

This information class returns more details about the heap in question or all heaps in a process using the `HEAP_EXTENDED_INFORMATION` structure and friends:

```
typedef struct _PROCESS_HEAP_INFORMATION {
    ULONG_PTR ReserveSize;
    ULONG_PTR CommitSize;
    ULONG NumberOfHeaps;
    ULONG_PTR FirstHeapInformationOffset;
} PROCESS_HEAP_INFORMATION, *PPROCESS_HEAP_INFORMATION;
typedef struct _HEAP_INFORMATION {
    ULONG_PTR Address;
    ULONG Mode;
    ULONG_PTR ReserveSize;
    ULONG_PTR CommitSize;
    ULONG_PTR FirstRegionInformationOffset;
    ULONG_PTR NextHeapInformationOffset;
} HEAP_INFORMATION, *PHEAP_INFORMATION;

typedef struct _HEAP_EXTENDED_INFORMATION {
    HANDLE Process;
    ULONG_PTR Heap;
    ULONG Level;
    PVOID CallbackRoutine;
    PVOID CallbackContext;
    union {
        PROCESS_HEAP_INFORMATION ProcessHeapInformation;
        HEAP_INFORMATION HeapInformation;
    };
} HEAP_EXTENDED_INFORMATION, *PHEAP_EXTENDED_INFORMATION;
```

On input, the heap handle to `RtlQueryHeapInformation` is not used. Instead, the first five members of `HEAP_EXTENDED_INFORMATION` determine what information is requested. One of the powers of this information class is the ability to get heap details from another process, passed as a handle in the `Process` member. This member can be `NtCurrentProcess()` if the calling process is the target. If it's a different process, the handle to that process has to be pretty powerful; it requires `PROCESS_QUERY_INFORMATION`, `PROCESS_CREATE_THREAD`, `PROCESS_VM_OPERATION`, `PROCESS_VM_READ`, `PROCESS_DUP_HANDLE`, and `PROCESS_VM_WRITE`. Sometimes, even this is not enough, and you may need `PROCESS_SET_QUOTA` and `PROCESS_SET_INFORMATION` as well. This is because in the cross-process case, a thread is created in the target process to get the information, and a Section object is shared with the target process.

> I find that for this kind of query, `PROCESS_ALL_ACCESS` is easier to ask for, or use `MAXIMUM_ALLOWED`, and see if that gives you access.

The `Level` member indicates the details required by the call. The most generic level is `HEAP_INFORMATION_LEVEL_PROCESS` (1), which fills the `ProcessHeapInformation` member. Here is an example (error handling omitted):

```
#define HEAP_INFORMATION_LEVEL_PROCESS 1

void DisplayProcessHeapsTotals(HANDLE hProcess) {
    HEAP_EXTENDED_INFORMATION info{};
    info.Process = hProcess;
    info.Level = HEAP_INFORMATION_LEVEL_PROCESS;

    RtlQueryHeapInformation(nullptr, HeapExtendedInformation,
        &info, sizeof(info), nullptr);
    printf("Total heaps: %u Commit: 0x%zX Reserved: 0x%zX\n",
        info.ProcessHeapInformation.NumberOfHeaps,
        info.ProcessHeapInformation.CommitSize,
        info.ProcessHeapInformation.ReserveSize);
}
```

> The *phnt* headers have constants like `HeapExtendedInformation` defined as macros instead of an enum, because such enum exists from *<WinNt.h>* but does not include the full list. I've simplified the code not to include a hard cast to `HEAP_INFORMATION_CLASS`, although I would have preferred that cast to be part of the macros, or alternatively define a slightly different enum name. I'll leave that to the interested reader.

The next `Level` is `HEAP_INFORMATION_LEVEL_HEAP` (2). This can be used to get details for each heap in a process, and also includes the total process heaps details shown above. If you need both, it's best to make a single call to get that instead of two separate calls. The returned result consists of a `ProcessHeapInformation` being filled up just like the previous case. However, `NextHeapInformationOffset` stores the offset to the first heap details (with `HEAP_INFORMATION_LEVEL_PROCESS` it's just zero).

Using that offset, the next items are all `HEAP_INFORMATION` structures, each describing a single heap, where `NextHeapInformationOffset` points to the next heap information. The following code snippet uses a single call to get the general details for the process and all heaps:

```
// hProcess is a handle to the process of interest
// assume no more than 128 heaps for simplicity
auto size = 128 * sizeof(HEAP_INFORMATION) + sizeof(HEAP_EXTENDED_INFORMATION);
auto buffer = std::make_unique<BYTE[]>(size);
auto info = (HEAP_EXTENDED_INFORMATION*)buffer.get();
info->Process = hProcess;
info->Level = HEAP_INFORMATION_LEVEL_HEAP;
RtlQueryHeapInformation(nullptr, HeapExtendedInformation,
    info, size, nullptr);
//
// get to the first heap
```

```
//
auto hi = (HEAP_INFORMATION*)(buffer.get() +
    info->ProcessHeapInformation.FirstHeapInformationOffset);
auto heaps = info->ProcessHeapInformation.NumberOfHeaps;
printf("Total heaps: %u Commit: 0x%zX Reserved: 0x%zX\n",
    heaps, info->ProcessHeapInformation.CommitSize,
    info->ProcessHeapInformation.ReserveSize);

for (ULONG i = 0; i < heaps; i++) {
    // heap details
    printf("Heap %2d: Addr: 0x%p Commit: 0x%08zX Reserve: 0x%08zX %s\n", i + 1,
        (PVOID)hi->Address, hi->CommitSize, hi->ReserveSize,
        hi->Mode ? "(LFH)" : "");

    // move to next heap
    hi = (HEAP_INFORMATION*)(buffer.get() + hi->NextHeapInformationOffset);
}
```

Here is some example output from the previous code (full source part of the *Heaps* sample):

```
Total heaps: 16 Commit: 0x1DE20000 Reserved: 0x1F418000
Heap  1: Addr: 0x0000000000C50000 Commit: 0x1C287000 Reserve: 0x1CE01000 (LFH)
Heap  2: Addr: 0x0000000000810000 Commit: 0x00001000 Reserve: 0x00010000
Heap  3: Addr: 0x0000000001210000 Commit: 0x01A01000 Reserve: 0x01F9D000 (LFH)
Heap  4: Addr: 0x00000000011F0000 Commit: 0x00051000 Reserve: 0x001D1000 (LFH)
Heap  5: Addr: 0x000000000C030000 Commit: 0x0004A000 Reserve: 0x001D1000 (LFH)
Heap  6: Addr: 0x000000000CBA0000 Commit: 0x00004000 Reserve: 0x00004000
Heap  7: Addr: 0x0000000016AB0000 Commit: 0x00004000 Reserve: 0x00004000
Heap  8: Addr: 0x0000000016AD0000 Commit: 0x00004000 Reserve: 0x00004000
Heap  9: Addr: 0x000000001CBD0000 Commit: 0x000D0000 Reserve: 0x001D1000 (LFH)
Heap 10: Addr: 0x000000001C7F0000 Commit: 0x00004000 Reserve: 0x00004000
Heap 11: Addr: 0x000000001C810000 Commit: 0x00004000 Reserve: 0x00004000
Heap 12: Addr: 0x000000001A880000 Commit: 0x00004000 Reserve: 0x00004000
Heap 13: Addr: 0x000000001A8A0000 Commit: 0x00004000 Reserve: 0x00004000
Heap 14: Addr: 0x00000000196A0000 Commit: 0x00004000 Reserve: 0x00004000
Heap 15: Addr: 0x00000000196C0000 Commit: 0x00004000 Reserve: 0x00004000
Heap 16: Addr: 0x0000000014E00000 Commit: 0x00008000 Reserve: 0x000D3000 (LFH)
```

The `Heap` member of `HEAP_EXTENDED_INFORMATION` can be used to get details for a specific heap rather than all heaps in the process (if zero). The value is the heap handle (in the context of the target process).

The next level of details has to do with memory regions within a heap, which provide another structure like so:

```
#define HEAP_INFORMATION_LEVEL_REGION 3

typedef struct _HEAP_REGION_INFORMATION {
    ULONG_PTR Address;
    SIZE_T ReserveSize;
    SIZE_T CommitSize;
    ULONG_PTR FirstRangeInformationOffset;
    ULONG_PTR NextRegionInformationOffset;
} HEAP_REGION_INFORMATION, * PHEAP_REGION_INFORMATION;
```

The `HEAP_INFORMATION` structure has a `FirstRegionInformationOffset` member pointing to the first region descriptor for this heap, beginning from the original buffer. The following is an example of dumping regions for a heap (full code in the *Heaps* sample):

```
void DisplayRegions(PBYTE buffer, HEAP_INFORMATION* hi) {
    if (hi->FirstRegionInformationOffset == 0)  // no region info
        return;

    auto region = (HEAP_REGION_INFORMATION*)(buffer +
        hi->FirstRegionInformationOffset);
    for (;;) {
        printf(" Region Addr: 0x%p Commit: 0x%08zX Reserve: 0x08%zX\n",
            (PVOID)region->Address, region->CommitSize, region->ReserveSize);
        if (region->NextRegionInformationOffset == 0)   // no more regions
            break;
        region = (HEAP_REGION_INFORMATION*)(buffer +
            region->NextRegionInformationOffset);
    }
}
```

The next level of detail is a range, described with another level value and structure:

```
#define HEAP_INFORMATION_LEVEL_RANGE 4

#define HEAP_RANGE_TYPE_COMMITTED 1
#define HEAP_RANGE_TYPE_RESERVED  2

typedef struct _HEAP_RANGE_INFORMATION {
    ULONG_PTR Address;
    SIZE_T Size;
    ULONG Type;
    ULONG Protection;
    ULONG_PTR FirstBlockInformationOffset;
    ULONG_PTR NextRangeInformationOffset;
} HEAP_RANGE_INFORMATION, *PHEAP_RANGE_INFORMATION;
```

Here is an example enumerating ranges:

```
void DisplayRanges(PBYTE buffer, HEAP_REGION_INFORMATION* region) {
    if (region->FirstRangeInformationOffset == 0)
        return;

    auto range = (HEAP_RANGE_INFORMATION*)(buffer + region->FirstRangeInformationOff\
set);
    for (;;) {
        printf("  Addr: 0x%p Size: 0x%08zX Type: %s Prot: %s\n",
            (PVOID)range->Address, range->Size,
            RangeTypeToString(range->Type),
            ProtectionToString(range->Protection).c_str());
        if (range->NextRangeInformationOffset == 0)
            break;
        range = (HEAP_RANGE_INFORMATION*)(buffer +
            range->NextRangeInformationOffset);
    }
}
```

`RangeTypeToString` returns "Committed" or "Reserved" based on the range type. `ProtectionToString` is a function we've seen before that converts protection constants (e.g. `PAGE_READWRITE`) to a string.

The last information detail (within a range) is blocks. Here too we have another level value and an associated structure:

```
#define HEAP_INFORMATION_LEVEL_BLOCK 5

#define HEAP_BLOCK_BUSY              1
#define HEAP_BLOCK_EXTRA_INFORMATION 2
#define HEAP_BLOCK_LARGE_BLOCK       4
#define HEAP_BLOCK_LFH_BLOCK         8

typedef struct _HEAP_BLOCK_INFORMATION {
    ULONG_PTR Address;
    ULONG Flags;    // see flags above
    SIZE_T DataSize;
    SIZE_T OverheadSize;
    ULONG_PTR NextBlockInformationOffset;
} HEAP_BLOCK_INFORMATION, *PHEAP_BLOCK_INFORMATION;
```

The pattern should be pretty clear by now. A range points to its first block via `FirstBlockInformationOffset`. Here is an example for enumerating blocks given a range:

```
void DisplayBlocks(PBYTE buffer, HEAP_RANGE_INFORMATION* range) {
    if (range->FirstBlockInformationOffset == 0)
        return;

    auto block = (HEAP_BLOCK_INFORMATION*)(buffer +
        range->FirstBlockInformationOffset);
    for (;;) {
        printf("    Block Addr: 0x%p Size: 0x%04zX Overhead: %2zd %s\n",
            (PVOID)block->Address, block->DataSize, block->OverheadSize,
            BlockFlagsToString(block->Flags).c_str());
        if (block->NextBlockInformationOffset == 0)
            break;

        block = (HEAP_BLOCK_INFORMATION*)(buffer +
            block->NextBlockInformationOffset);
    }
}
```

`BlockFlagsToString` is a little function to convert the flags to a string:

```
std::string BlockFlagsToString(ULONG flags) {
    static const struct {
        ULONG flag;
        PCSTR text;
    } data[] = {
        { HEAP_BLOCK_BUSY, "Busy" },
        { HEAP_BLOCK_EXTRA_INFORMATION, "Extra" },
        { HEAP_BLOCK_LARGE_BLOCK, "Large" },
        { HEAP_BLOCK_LFH_BLOCK, "LFH" },
    };

    std::string text;
    for (auto& d : data) {
        if ((d.flag & flags) == d.flag)
            (text += d.text) += ", ";
    }
    return text.empty() ? "" : text.substr(0, text.length() - 2);
}
```

The block level is the last, but if the `HEAP_BLOCK_EXTRA_INFORMATION` is set, it means there is extra information for the block in question, immediately following `HEAP_BLOCK_INFORMATION`. Its header looks like this:

```
typedef struct _HEAP_BLOCK_EXTRA_INFORMATION {
    BOOLEAN Next;
    ULONG Type;
    SIZE_T Size;
    // data follows
} HEAP_BLOCK_EXTRA_INFORMATION, *PHEAP_BLOCK_EXTRA_INFORMATION;
```

The `Next` member indicates whether there is another extra information block following this one. `Type` is the extra information type which currently can be only 1. The extra information structure itself is as follows:

```
typedef struct _HEAP_BLOCK_SETTABLE_INFORMATION {
    SIZE_T Settable;
    USHORT TagIndex;
    USHORT AllocatorBackTraceIndex;
} HEAP_BLOCK_SETTABLE_INFORMATION, *PHEAP_BLOCK_SETTABLE_INFORMATION;
```

We've seen similar information in the discussion of `RtlWalkHeap`.

8.7.5.1: Callback

`HEAP_EXTENDED_INFORMATION` has a `CallbackRoutine` and `CallbackContext` members. If `CallbackRoutine` is non-`NULL`, it points to a callback invoked for every item enumerated. This means the buffer provided to `RtlQueryHeapInformation` needs to be `sizeof(HEAP_EXTENDED_INFORMATION)` and no more regardless of the value of `Level`. No further allocations are needed, as the callback is invoked for each and every item requested (heap, range, region, block) based on `Level`.

The callback prototype looks like the following with the given structure:

```
typedef struct _HEAP_INFORMATION_ITEM {
    ULONG Level;    // reported level
    SIZE_T Size;
    union {
        PROCESS_HEAP_INFORMATION ProcessHeapInformation;
        HEAP_INFORMATION HeapInformation;
        HEAP_REGION_INFORMATION HeapRegionInformation;
        HEAP_RANGE_INFORMATION HeapRangeInformation;
        HEAP_BLOCK_INFORMATION HeapBlockInformation;
        ULONG_PTR DynamicStart;
    };
} HEAP_INFORMATION_ITEM, * PHEAP_INFORMATION_ITEM;

NTSTATUS HeapInformationCallback(
    _In_ PHEAP_INFORMATION_ITEM item,
    _In_ PVOID context);
```

Returning `STATUS_SUCCESS` from the callback continues enumeration, while returning a failure status aborts further calls and that status is returned from `RtlQueryHeapInformation`.

8.7.6: `HeapOptimizeResources` (3)

This information class is only valid with `RtlSetHeapInformation` and requires the following structure:

```
typedef struct _HEAP_OPTIMIZE_RESOURCES_INFORMATION {
    ULONG Version;
    ULONG Flags;
} HEAP_OPTIMIZE_RESOURCES_INFORMATION, *PHEAP_OPTIMIZE_RESOURCES_INFORMATION;
```

Currently, `Version` must be set to 1, and `Flags` must be set to zero.

The call attempts to optimize the heap by decommitting regions that are currently unused. if the provided heap handle to `RtlSetHeapInformation` is `NULL`, all heaps in the process are so optimized. If a specific heap handle is provided, only that heap is dealt with.

8.7.7: `HeapMemoryLimit` (6)

This information class is available `RtlSetHeapInformation` to set some limits on the specified heap. The information structure must be the following:

```
typedef struct _RTL_HEAP_MEMORY_LIMIT_DATA {
    SIZE_T CommitLimitBytes;
    ULONG_PTR CommitLimitFailureCode;
    SIZE_T MaxAllocationSizeBytes;
    ULONG_PTR AllocationLimitFailureCode;
} RTL_HEAP_MEMORY_LIMIT_DATA, *PRTL_HEAP_MEMORY_LIMIT_DATA;

typedef struct _RTL_HEAP_MEMORY_LIMIT_INFO {
    ULONG Version;  // must be set to 1
    RTL_HEAP_MEMORY_LIMIT_DATA Data;
} RTL_HEAP_MEMORY_LIMIT_INFO, *PRTL_HEAP_MEMORY_LIMIT_INFO;
```

The above members should be self explanatory. The sizes must be in bytes, but multiples of a page size.

The remaining information classes either don't work or require more research.

8.8: Summary

This chapter covered a lot of ground as it related to memory. From the *Virtual* APIs to heaps. This is not all that is available, however. In chapter 13, we'll cover more memory related APIs, including Section objects, memory zones, and lookaside lists.

Chapter 9: I/O

In this chapter:

- **Files and Devices**
- **File and Device API**
- **File Information**
- **NTFS Streams**
- **Extended Attributes**
- **Accessing Devices**
- **I/O Completion Ports**
- **Miscellaneous Functions**

9.1: Files and Devices

File objects are an abstraction over connections with device objects. Device objects can represent physical devices, such as a hard disk or a keyboard, or virtual devices, such as a pipe. The device object is responsible for the actual reading and writing of data. The file object provides a convenient interface for the user to interact with the device object.

Creating a file object is always about connecting to a device. Device objects cannot be directly opened as such from user-mode. The Windows API function `CreateFile` is the main interface to creating a file object. The "create" in `CreateFile` is not about creating files in the file system necessarily - files in a file system derive from a some disk device; it's about creating the file object itself - a kernel structure that represents a connection to a device object. Any access to the device object is performed through the file object. Figure 9-1 shows the relationship between file objects and device objects. Note that multiple file objects can talk to the same device object.

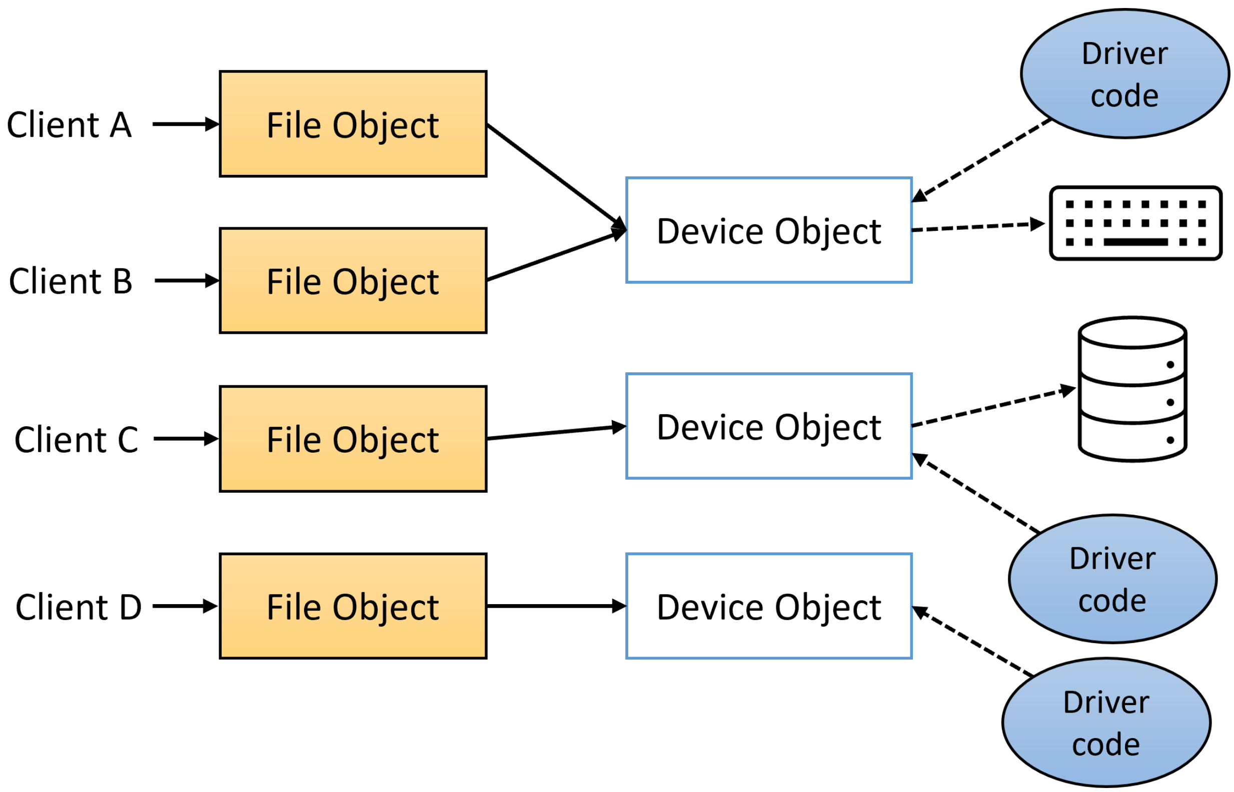


Figure 9-1: File and device objects

The well-known drive names users are accustomed to, such as "C:", "D:", etc. are nothing more than *symbolic links* - an object type we looked at in chapter 7. You can use the Windows API `QueryDosDevice` to get the target device name given a symbolic link. Here is an example:

```
WCHAR buffer[256];
QueryDosDevice(L"c:", buffer, _countof(buffer));
```

On my system I get "\Device\HarddiskVolume3" as the result. This is the device object that represents the C: drive. The symbolic links that are directly accessible by the `CreateFile` Windows API can be viewed with tools like *WinObj* from *Sysinternals* or my own *Object Explorer*. Figure 9-2 shows the "Global??" Object Manager directory in *Object Explorer*, which contains the symbolic links that can be used with `CreateFile`.

Except for drive letters and certain old DOS names (like `CON`), symbolic links passed to `CreateFile` must be prefixed with "\\.\" so they are interpreted as symbolic links and not as file names.

`QueryDosDevice` uses `NtQuerySymbolicLinkObject` under the covers.

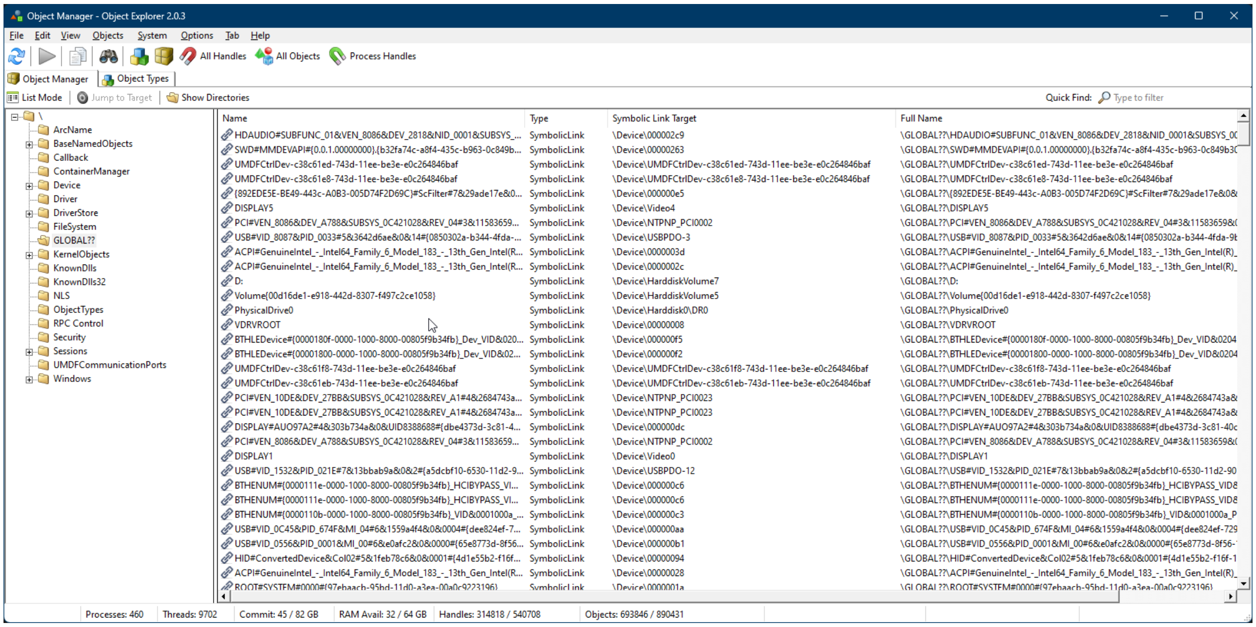


Figure 9-2: Global?? Object Manager directory

Contrary to `CreateFile`, the native API allows direct access to any object in the Object Manager namespace.

9.2: File and Device API

The native APIs to use for accessing devices are `NtCreateFile` and `NtOpenFile`:

```
NTSTATUS NtCreateFile(
    _Out_ PHANDLE FileHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_opt_ PLARGE_INTEGER AllocationSize,
    _In_ ULONG FileAttributes,
    _In_ ULONG ShareAccess,
    _In_ ULONG CreateDisposition,
    _In_ ULONG CreateOptions,
    _In_reads_bytes_opt_(EaLength) PVOID EaBuffer,
    _In_ ULONG EaLength);
NTSTATUS NtOpenFile(
    _Out_ PHANDLE FileHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
```

```
    _In_ ULONG ShareAccess,
    _In_ ULONG OpenOptions);
```

Fortunately, these APIs are officially documented in the WDK (under `ZwCreateFile` and `ZwOpenFile`) since device driver developers need these APIs for I/O work. `NtOpenFile` is the simpler API, providing a subset of the capabilities of `NtCreateFile`. Internally, both call a common function in the kernel (`IopCreateFile`), where `NtOpenFile` passes some defaults that are configurable with `NtCreateFile`.

The `CreateFile` Windows API calls `NtCreateFile` internally; this can be easily observed with a user-mode debugger.

Most of `NtOpenFile` parameters should be familiar by now. `ObjectAttributes` is where the device name is specified. `IoStatusBlock` is an output parameter filled with the following details:

```
typedef struct _IO_STATUS_BLOCK {
    union {
        NTSTATUS Status;
        PVOID Pointer;
    };
    ULONG_PTR Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

`Information` contains one of the following values indicating the type of operation that was performed. This is relevant for true files/directories in a file system, but not for devices (devices must always exist before an attempt to open them can succeed):

- `FILE_CREATED` (2) - a new file was created.
- `FILE_OPENED` (1) - an existing file was opened.
- `FILE_OVERWRITTEN` (3) - an existing file was overwritten.
- `FILE_SUPERSEDED` (0) - an existing file was superseded, which means the file was replaced.
- `FILE_EXISTS` (4) - an existing file was found.
- `FILE_DOES_NOT_EXIST` (5) - an existing file was not found.

`DesiredAccess` is the access mask requested. This can include specific access masks for file objects (e.g. `FILE_READ_DATA`), file generic ones (`FILE_GENERIC_READ`), and even object-generic ones (e.g. `GENERIC_READ`); this is no different in principle than any other object type.

`ShareAccess` is the share mode to open/create the file; it only has meaning if a new file is actually created. Values include `FILE_SHARE_READ`, `FILE_SHARE_WRITE`, and `FILE_SHARE_DELETE`. Finally, `OpenOptions` allows specifying more flags to customize the operation. Many such flags exist, documented in the WDK under `ZwCreateFile`.

The following example opens the *Kernel32.dll* file from the *System32* directory:

```
UNICODE_STRING path;
RtlInitUnicodeString(&path, L"\\SystemRoot\\System32\\kernel32.dll");
OBJECT_ATTRIBUTES attr = RTL_CONSTANT_OBJECT_ATTRIBUTES(&path, 0);
HANDLE hFile;
IO_STATUS_BLOCK ioStatus;
status = NtOpenFile(&hFile, FILE_READ_DATA, &attr, &ioStatus,
    FILE_SHARE_READ, 0);
```

Why does that even work? *SystemRoot* is a symbolic link that points to the root directory of where Windows is installed, typically *C:\Windows* from a user's perspective, but not quite what that looks like underneath. Looking at *Object Explorer* or the *ObjDir* tool we developed in chapter 7, we can see that *SystemRoot* is a symbolic link pointing to *Device\BootDevice\Window*. What is *Device\BootDevice*? Going to the *Device* directory and locating *BootDevice* shows that *BootDevice* is yet another symbolic link, pointing to *Device\HarddiskVolume3*. Looking at *\Global??\C:* shows it points to the same device, representing the C: drive on this machine.

Run *WinObj* or *ObjExp* with admin privileges to see the above symbolic link target.

> To use common user-mode paths, like "c:\Something", prepend "\??\" to the path. For example, "c:\Something" should be written as "\??\c:\Something"; "\??\" is the same as "\Global??\".

Of course we could get to the *C:\Windows* directory directly with *Device\HarddiskVolume3\Windows*, but that device may not be the same on every Windows system. That's why using the symbolic links is beneficial - they are guaranteed to be the same on every Windows system.

The example for opening the file above seems simple enough. Unfortunately, it's not *that* simple. Suppose we want to read the first 1024 bytes from the file. We can use the `NtReadFile` API:

```
typedef void (NTAPI *PIO_APC_ROUTINE)(
    _In_ PVOID ApcContext,
    _In_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_ ULONG Reserved);

NTSTATUS NtReadFile(
    _In_ HANDLE FileHandle,
    _In_opt_ HANDLE Event,
    _In_opt_ PIO_APC_ROUTINE ApcRoutine,
    _In_opt_ PVOID ApcContext,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
```

```
    _Out_writes_bytes_(Length) PVOID Buffer,
    _In_ ULONG Length,
    _In_opt_ PLARGE_INTEGER ByteOffset,
    _In_opt_ PULONG Key);
```

The `FileHandle` parameter is the handle returned from `NtOpenFile` or `NtCreateFile`. `Event` is an optional event handle to signal when the operation completes. `ApcRoutine` is an optional APC callback that is used to construct an APC on the calling thread when the I/O operation completes. The thread must enter an altertable state if the APC is to be eventually executed. `ApcContext` is an optional context value to pass to the APC routine. `Event`, `ApcRoutine`, and `ApcContext` only have meaning if the operation is asynchronous. We'll get to that in a moment.

`Buffer` is the buffer to read the data into, and `Length` is the number of bytes to read. `ByteOffset` is an optional offset to read from. Typically, `NULL` is provided which indicates to read from the current file position. `Key` is an optional key value to use for the operation, which has no use in user-mode.

Here is simple code to read the first 1KB from the previously opened file:

```
BYTE buffer[1024];
status = NtReadFile(hFile, nullptr, nullptr, nullptr, &ioStatus,
    buffer, sizeof(buffer), nullptr, nullptr);
```

The call mysteriously fails with `STATUS_INVALID_PARAMETER`. What's going on? The problem is that the file object was opened for asynchronous access. We didn't specify anything special in the call to `NtCreateFile`, but the default is asynchronous access. This is in contrast to the `CreateFile` function, which creates a file object for synchronous access by default.

To create a file object for synchronous access, we need to specify the `FILE_SYNCHRONOUS_IO_NONALERT` or `FILE_SYNCHRONOUS_IO_ALERT` flag in the `OpenOptions` parameter of `NtOpenFile`. Here is the correct call to `NtOpenFile`:

```
status = NtOpenFile(&hFile, FILE_READ_DATA | SYNCHRONIZE, &attr, &ioStatus,
    FILE_SHARE_READ, FILE_SYNCHRONOUS_IO_NONALERT);
```

Notice the access mask includes the `SYNCHRONIZE` flag. This is necessary because this flag is required to wait on the file object, which is what happens internally in for synchronous operations. Now the call to `NtReadFile` can succeed and data is available when the call returns.

What do we need to do to make an asynchronous call work? In this case, the call to `NtReadFile` must specify a file position in the `ByteOffset` parameter. This is because a file object does not keep track of a file position if opened for asynchronous access. This makes sense, since multiple operations could start concurrently, so there is no meaning to a single file position. Optionally, an event handle should be provided (or an APC callback) so that a thread can tell when the operation is complete and data is available. The status returned for a successful initiation of an async operation is `STATUS_PENDING` (0x103).

Here is a simple example that waits immediately after the asynchronous call:

```
HANDLE hEvent;
NtCreateEvent(&hEvent, EVENT_ALL_ACCESS, nullptr, NotificationEvent, FALSE);
LARGE_INTEGER pos{};    // pos is set to zero (beginning of file)
status = NtReadFile(hFile, hEvent, nullptr, nullptr, &ioStatus,
    buffer, sizeof(buffer), &pos, nullptr);
if (status != STATUS_PENDING) {
    // some real error occurred
}
else {
    NtWaitForSingleObject(hEvent, FALSE, nullptr);
    // data is available in buffer
}
NtClose(hEvent);
```

In a real scenario, the waiting would not occur immediately after initiation, as that would defeat the purpose of asynchronous I/O. Instead, the thread would do other work and periodically check the status of the operation, or another thread would wait on the event, or the thread pool would be used to wait on the event and execute a callback when the operation completes.

Keep in mind that for asynchronous operations, the `IO_STATUS_BLOCK` and buffer provided should be kept alive until the operation completes. If these are stack-allocated, then the thread must not exit the current function until the operation completes.

Writing to a file object is similar to reading, with `NtWriteFile`:

```
NTSTATUS NtWriteFile(
    _In_ HANDLE FileHandle,
    _In_opt_ HANDLE Event,
    _In_opt_ PIO_APC_ROUTINE ApcRoutine,
    _In_opt_ PVOID ApcContext,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_reads_bytes_(Length) PVOID Buffer,
    _In_ ULONG Length,
    _In_opt_ PLARGE_INTEGER ByteOffset,
    _In_opt_ PULONG Key);
```

The function is virtually identical to `NtReadFile`, except for SAL annotations and the fact that the `Buffer` parameter can be declared as `const`.

For more information about these APIs, including `NtCreateFile`, consult the WDK documentation.

9.3: File Information

File and directory information is first and foremost available with the `NtQueryInformationFile` and `NtSetInformationFile` functions, which follow the classic model we've seen before:

```
NTSTATUS NtQueryInformationFile(
    _In_ HANDLE FileHandle,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _Out_writes_bytes_(Length) PVOID FileInformation,
    _In_ ULONG Length,
    _In_ FILE_INFORMATION_CLASS FileInformationClass);
NTSTATUS NtSetInformationFile(
    _In_ HANDLE FileHandle,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_reads_bytes_(Length) PVOID FileInformation,
    _In_ ULONG Length,
    _In_ FILE_INFORMATION_CLASS FileInformationClass);
```

A related query function works on a file name instead of a handle:

```
NTSTATUS NtQueryInformationByName(
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _Out_writes_bytes_(Length) PVOID FileInformation,
    _In_ ULONG Length,
    _In_ FILE_INFORMATION_CLASS FileInformationClass);
```

As you might expect, there is a long list of file information classes. Some apply to files only, some to directories only, and others to both. In the following subsections we'll examine some of them.

> Some of the information classes and associated structures are documented in the WDK and/or SDK. Look for `ZwQueryInformationFile` in the WDK and `GetFileInformationByHandleEx` in the SDK.

9.3.1: `FileBasicInformation` (4)

This information class returns the basic details for a file or directory:

```
typedef struct _FILE_BASIC_INFORMATION {
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    ULONG FileAttributes;
} FILE_BASIC_INFORMATION, *PFILE_BASIC_INFORMATION;
```

The members are mostly self-explanatory, where times are expressed in 100nsec units since January 1, 1601. `FileAttributes` is a set of file attributes, such as `FILE_ATTRIBUTE_DIRECTORY`, `FILE_ATTRIBUTE_HIDDEN`, `FILE_ATTRIBUTE_READONLY`, etc. The difference between `ChangeTime` and `LastWriteTime` is that `ChangeTime` can mean change in file name or file attributes, whereas `LastWriteTime` is only the last time the file contents were modified.

The same information class can be used to set these attributes. The file handle must include the `FILE_WRITE_ATTRIBUTES` access mask (`FILE_GENERIC_WRITE` and `GENERIC_WRITE` include it).

The following example sets a file's creattion time to the given value:

```
NTSTATUS SetCreateTime(PCWSTR filename, LARGE_INTEGER const& createTime) {
    FILE_BASIC_INFORMATION fbi;
    UNICODE_STRING name;
    RtlInitUnicodeString(&name, filename);
    OBJECT_ATTRIBUTES attr = RTL_CONSTANT_OBJECT_ATTRIBUTES(&name, 0);
    HANDLE hFile;
    IO_STATUS_BLOCK ioStatus;
    auto status = NtOpenFile(&hFile, FILE_READ_ATTRIBUTES | FILE_WRITE_ATTRIBUTES,
        &attr, &ioStatus, 0, 0);
    if (!NT_SUCCESS(status))
        return status;

    status = NtQueryInformationFile(hFile, &ioStatus, &fbi,
        sizeof(fbi), FileBasicInformation);
    if (!NT_SUCCESS(status))
        return status;

    fbi.CreationTime = createTime;
    status = NtSetInformationFile(hFile, &ioStatus, &fbi,
        sizeof(fbi), FileBasicInformation);
    NtClose(hFile);
    return status;
}
```

The following example sets the creation time of a file to 24 hours in the future:

```
LARGE_INTEGER time;
NtQuerySystemTime(&time);
time.QuadPart += 10000000ULL * 60 * 60 * 24;
SetCreateTime(L"\\??\\C:\\Temp\\myfile.txt", time);
```

9.3.2: FileStandardInformation (5)

This information class returns the following structure:

```
typedef struct _FILE_STANDARD_INFORMATION {
    LARGE_INTEGER AllocationSize;
    LARGE_INTEGER EndOfFile;
    ULONG NumberOfLinks;
    BOOLEAN DeletePending;
    BOOLEAN Directory;
} FILE_STANDARD_INFORMATION, *PFILE_STANDARD_INFORMATION;
```

`AllocationSize` is the file allocation size, which is typically a multiple of the underlying sector size of the physical device. `EndOfFile` is the file size (zero for directories). `NumberOfLinks` is the number of hard links to the file. `DeletePending` indicates whether the file is pending deletion. For example, if it was opened with the `FILE_DELETE_ON_CLOSE` flag. `Directory` indicates whether the file is a directory.

9.3.3: FileNameInformation (9)

This information returns the full name of the file or directory in a `FILE_NAME_INFORMATION` structure:

```
typedef struct _FILE_NAME_INFORMATION {
    ULONG FileNameLength;    // in bytes
    WCHAR FileName[1];
} FILE_NAME_INFORMATION, *PFILE_NAME_INFORMATION;
```

`FileNameLength` is the length of the filename in bytes. `FileName` is the filename itself, which is not necessarily null-terminated. If the supplied buffer is not large enough to receive the full name, the function returns `STATUS_BUFFER_OVERFLOW` and `FileNameLength` is set to the required size (at least this is what file system drivers and filters suppose to do).

The returned file name always starts with a backslash. It does not include a drive letter or volume name. For example, the file name returned for "C:\Windows\System32\kernel32.dll" is "\Windows\System32\kernel32.dll". The same file name is returned if the file was opened by the name "\SystemRoot\System32\kernel32.dll".

How do you get the volume name? See next information class.

9.3.4: FileVolumeNameInformation (58)

This information class returns the volume device name in a `FILE_VOLUME_NAME_INFORMATION` structure:

```
typedef struct _FILE_VOLUME_NAME_INFORMATION {
    ULONG DeviceNameLength;
    WCHAR DeviceName[1];
} FILE_VOLUME_NAME_INFORMATION, *PFILE_VOLUME_NAME_INFORMATION;
```

It's identical in format to `FILE_NAME_INFORMATION`. Here is an example of using it:

```
BYTE buffer[1 << 11];
status = NtQueryInformationFile(hFile, &ioStatus, buffer,
    sizeof(buffer), FileVolumeNameInformation);
auto info = (FILE_VOLUME_NAME_INFORMATION*)buffer;
```

The return value is something like "\Device\HarddiskVolume3". Combining it with the `FileNameInformation` information class produces the full path name in device form. If a drive letter is desired, you'll need to lookup drive letters as symbolic link names and see which points to the device name. I'll leave that as an exercise for the interested reader. Note that this may fail - not all volumes are mapped to drive letters - there is no such requirement. A quick way to see what is mapped is to the *fltmc.exe* command line tool, which must be run from an elevated command prompt. For example (formatted for clarity):

```
C:\Users\Pavel> fltmc volumes
Dos Name              Volume Name                       FileSystem
----------------  --------------------------------    ----------
                  \Device\Mup                          Remote
D:                \Device\HarddiskVolume7              NTFS
C:                \Device\HarddiskVolume3              NTFS
                  \Device\NamedPipe                    NamedPipe
                  \Device\Mailslot                     Mailslot
                  \Device\HarddiskVolume1              FAT
                  \Device\HarddiskVolume5              NTFS
F:                \Device\HarddiskVolume8              NTFS
                  \Device\HarddiskVolume4              NTFS
                  \Device\HarddiskVolumeShadowCopy4    NTFS
                  \Device\HarddiskVolumeShadowCopy6    NTFS
                  \Device\HarddiskVolumeShadowCopy7    NTFS
```

The *fltmc.exe* tool is in the *System32* directory.

9.3.5: `FileRenameInformation` (10)

This information class can be used with `NtSetInformationFile` to rename or move a file or directory. The structure provided is `FILE_RENAME_INFORMATION`:

```
typedef struct _FILE_RENAME_INFORMATION {
    BOOLEAN ReplaceIfExists;
    HANDLE RootDirectory;
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_RENAME_INFORMATION, *PFILE_RENAME_INFORMATION;
```

`ReplaceIfExists` indicates whether to replace an existing file if the new file name exists. `RootDirectory` is a handle to the directory where the new file should reside. If `RootDirectory` is `NULL`, then the file is renamed in the same directory. `FileNameLength` is the length of the new name in bytes and `FileName` is the new name, which is not necessarily null-terminated.

Here is an example of renaming a file in the same directory:

```
NTSTATUS RenameFile(PCWSTR path, PCWSTR newName) {
    UNICODE_STRING name;
    RtlInitUnicodeString(&name, path);
    OBJECT_ATTRIBUTES attr = RTL_CONSTANT_OBJECT_ATTRIBUTES(&name, 0);
    HANDLE hFile;
    IO_STATUS_BLOCK ioStatus;
    auto status = NtOpenFile(&hFile, DELETE, &attr, &ioStatus, 0, 0);
    if (NT_SUCCESS(status)) {
        BYTE buffer[1 << 10];
        auto info = (FILE_RENAME_INFORMATION*)buffer;
        info->ReplaceIfExists = FALSE;
        info->RootDirectory = nullptr;
        info->FileNameLength = (ULONG)wcslen(newName) * sizeof(WCHAR);
        memcpy(info->FileName, newName, info->FileNameLength);

        status = NtSetInformationFile(hFile, &ioStatus, info,
            sizeof(buffer), FileRenameInformation);
        NtClose(hFile);
    }
    return status;
}
```

Notes:

- The file must be opened with the `DELETE` access mask. `CreateOptions` should be zero (not something like `FILE_SYNCHRONOUS_IO_NONALERT`).
- The buffer size provided to `NtSetInformationFile` in the above code is larger than needed, but that's not an issue. Still, the buffer should be allocated dynamically based on the length of the new name.

Here is an example of usage:

```
RenameFile(L"\\??\\C:\\Temp\\myfile.txt", L"somefile.txt");
```

We can extend the example to support moving a file to a different directory by specifying a root directory handle. A directory handle can be opened for this purpose like so:

```
HANDLE OpenDirectoryForMoving(PCWSTR path) {
    IO_STATUS_BLOCK ioStatus;
    UNICODE_STRING name;
    RtlInitUnicodeString(&name, path);
    HANDLE hDir = nullptr;
    OBJECT_ATTRIBUTES dirAtt = RTL_CONSTANT_OBJECT_ATTRIBUTES(&name, 0);
    NtOpenFile(&hDir, FILE_GENERIC_WRITE, &dirAtt, &ioStatus,
        FILE_SHARE_READ | FILE_SHARE_WRITE, FILE_DIRECTORY_FILE);
    return hDir;
}
```

The target directory must be in the same volume as the file being moved.

9.3.6: `FileDispositionInformation` (13)

This information class is used with `NtSetInformationFile` to delete a file or directory. The structure required is a simple `BOOLEAN`:

```
typedef struct _FILE_DISPOSITION_INFORMATION {
    BOOLEAN DeleteFile;
} FILE_DISPOSITION_INFORMATION, *PFILE_DISPOSITION_INFORMATION;
```

`DeleteFile` is set to `TRUE` to delete the file. The file must be opened with the `DELETE` access mask. The following example deletes a file:

```
NTSTATUS NtDeleteFile(PCWSTR path) {
    UNICODE_STRING name;
    RtlInitUnicodeString(&name, path);
    OBJECT_ATTRIBUTES attr = RTL_CONSTANT_OBJECT_ATTRIBUTES(&name, 0);
    HANDLE hFile;
    IO_STATUS_BLOCK ioStatus;
    auto status = NtOpenFile(&hFile, DELETE, &attr, &ioStatus, 0, 0);
    if (NT_SUCCESS(status)) {
        FILE_DISPOSITION_INFORMATION info{ TRUE };
```

```
        status = NtSetInformationFile(hFile, &ioStatus, &info,
            sizeof(info), FileDispositionInformation);
        NtClose(hFile);
    }
    return status;
}
```

The file is actually deleted when the handle is closed. A simple way to delete a file is to open it with the flag `FILE_DELETE_ON_CLOSE` (as part of `CreateOptions`), and closing the handle.

9.3.7: `FilePositionInformation` (14)

This information class can be used to query or set the file pointer as a `LARGE_INTEGER`:

```
typedef struct _FILE_POSITION_INFORMATION {
    LARGE_INTEGER CurrentByteOffset;
} FILE_POSITION_INFORMATION, *PFILE_POSITION_INFORMATION;
```

9.3.8: `FileModeInformation` (16)

This information class returns the mode the file was opened with. The structure returned is `FILE_MODE_INFORMATION`, which is a glorified set of flags:

```
typedef struct _FILE_MODE_INFORMATION {
    ULONG Mode;
} FILE_MODE_INFORMATION, *PFILE_MODE_INFORMATION;
```

The `Mode` includes flags like `FILE_WRITE_THROUGH`, `FILE_SEQUENTIAL_ONLY`, `FILE_NO_INTERMEDIATE_BUFFERING`, `FILE_SYNCHRONOUS_IO_ALERT`, and others. There are the ones that can be used with `NtCreateFile` and `NtOpenFile` as part of `CreateOptions`. See the WDK docs for more information.

This information class is also partially supported with `NtSetInformationFile`, especially contradicting the way the file was opened in relation to synchronous/asynchronous open.

9.3.9: `FileAllInformation` (18)

This information class returns many of the structures we've seen before as one:

```
typedef struct _FILE_ALL_INFORMATION {
    FILE_BASIC_INFORMATION BasicInformation;
    FILE_STANDARD_INFORMATION StandardInformation;
    FILE_INTERNAL_INFORMATION InternalInformation;
    FILE_EA_INFORMATION EaInformation;
    FILE_ACCESS_INFORMATION AccessInformation;
    FILE_POSITION_INFORMATION PositionInformation;
    FILE_MODE_INFORMATION ModeInformation;
    FILE_ALIGNMENT_INFORMATION AlignmentInformation;
    FILE_NAME_INFORMATION NameInformation;
} FILE_ALL_INFORMATION, *PFILE_ALL_INFORMATION;
```

This is more efficient than making multiple calls, assuming most of these details are of interest. Note that this is a variable length structure, as the last member is a variable-length string.

9.4: Directory-Only Information

Some information classes are only applicable to directories only, when directory contents is involved. The `NtQueryDirectoryFile` function is used to get this information:

```
NTSTATUS NtQueryDirectoryFile(
    _In_ HANDLE FileHandle,
    _In_opt_ HANDLE Event,
    _In_opt_ PIO_APC_ROUTINE ApcRoutine,
    _In_opt_ PVOID ApcContext,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _Out_writes_bytes_(Length) PVOID FileInformation,
    _In_ ULONG Length,
    _In_ FILE_INFORMATION_CLASS FileInformationClass,
    _In_ BOOLEAN ReturnSingleEntry,
    _In_opt_ PUNICODE_STRING FileName,
    _In_ BOOLEAN RestartScan);
```

`FileHandle` must be a directory handle (`FILE_DIRECTORY_FILE` specified in a call to `NtOpenFile` or `NtCreateFile`).

The last three parameters are new. `ReturnSingleEntry` indicates whether to return a single entry or all entries (remember this is a directory). `FileName` can be used to filter the returned items. If `NULL` is specified, no filtering is performed. If non-`NULL`, it indicates the items of interest, supporting wildcards. For example, specifying "Hello.*" returns all files/directories whos name is "hello" with any extension. `RestartScan` indicates whether to restart the scan from the beginning or continue from the last entry. This is useful when the buffer provided is not large enough to receive all entries. The first time `NtQueryDirectoryFile` is called with a particular handle, `RestartScan` is ignored. It is respected on subsequent calls.

The next subsections examine the directory-contents information classes.

9.4.1: `FileDirectoryInformation` (1), `FileFullDirectoryInformation` (2), `FileBothDirectoryInformation` (3)

These information classes are similar, with the difference being the details returned. The provided handle must be of a directory (not a file). The `FileDirectoryInformation` class returns the following structure:

```
typedef struct _FILE_DIRECTORY_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_DIRECTORY_INFORMATION, *PFILE_DIRECTORY_INFORMATION;
```

It's a variable-size structure, as the filename appears as the last member. Moving to the next structure is done using `NextEntryOffset`, which indicates the number of bytes to move forward to the next entry. When `NextEntryOffset` is zero, there are no more entries. The structure is described in the WDK documentation. Here is a brief description of the members:

- `FileIndex` is a byte offset of the file from the parent directory. It should normally be ignored, as it's not used by NTFS, for example.
- `CreationTime`, `LastAccessTime`, `LastWriteTime`, and `ChangeTime` should be familiar by now.
- `EndOfFile` is the size of the file in bytes.
- `AllocationSize` is the file allocation size, which is typically a multiple of the underlying sector size of the physical device.
- `FileAttributes` is a set of file attributes, such as `FILE_ATTRIBUTE_DIRECTORY`, `FILE_ATTRIBUTE_HIDDEN`, `FILE_ATTRIBUTE_READONLY`, etc.
- `FileNameLength` is the length of the filename in bytes.
- `FileName` is the filename itself, which is not necessarily null-terminated.

The following example lists the items in a given directory:

```
NTSTATUS ListDirectory(PCWSTR path, PCWSTR filter) {
    //
    // init directory name
    //
    UNICODE_STRING name;
    RtlInitUnicodeString(&name, path);
    //
    // init filter (if any)
    //
    UNICODE_STRING filterName;
    if (filter)
        RtlInitUnicodeString(&filterName, filter);
    OBJECT_ATTRIBUTES attr = RTL_CONSTANT_OBJECT_ATTRIBUTES(&name, 0);
    HANDLE hDir;
    IO_STATUS_BLOCK ioStatus;
    //
    // open directory
    //
    auto status = NtOpenFile(&hDir, FILE_LIST_DIRECTORY | SYNCHRONIZE,
        &attr, &ioStatus, FILE_SHARE_READ | FILE_SHARE_WRITE,
        FILE_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT);
    if (!NT_SUCCESS(status))
        return status;

    BYTE buffer[1 << 10];
    for(;;) {
        //
        // retriueve as many items as would fit in the buffer
        //
        status = NtQueryDirectoryFile(hDir, nullptr, nullptr, nullptr,
            &ioStatus, buffer, sizeof(buffer), FileDirectoryInformation,
            FALSE, filter ? &filterName : nullptr, FALSE);
        if (!NT_SUCCESS(status))
            break;

        auto info = (FILE_DIRECTORY_INFORMATION*)buffer;
        for (;;) {
            printf("%.*ws", (int)(info->FileNameLength / sizeof(WCHAR)),
                info->FileName);
            //
            // add <DIR> in case of a directory
            // otherwise, add file size in KB
```

```
            //
            if(info->FileAttributes & FILE_ATTRIBUTE_DIRECTORY)
                printf(" <DIR>");
            else
                printf(" [%llu KB]", info->EndOfFile.QuadPart >> 10);

            printf("\n");
            if (info->NextEntryOffset == 0)
                break;
            info = (FILE_DIRECTORY_INFORMATION*)((BYTE*)info +
                info->NextEntryOffset);
        }
    }

    NtClose(hDir);
    return status;
}
```

`FileFullDirectoryInformation` returns an extended structure:

```
typedef struct _FILE_FULL_DIR_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    ULONG EaSize;
    WCHAR FileName[1];
} FILE_FULL_DIR_INFORMATION, *PFILE_FULL_DIR_INFORMATION;
```

`EaSize` is the size of the extended attributes in bytes. See the subsection "Extended Attributes" later in this chapter for more information.

Finally, `FileBothDirectoryInformation` returns an even extended structure that contains the "short name" (in the old DOS format 8.3):

```
typedef struct _FILE_BOTH_DIR_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    ULONG EaSize;
    CCHAR ShortNameLength;
    WCHAR ShortName[12];
    WCHAR FileName[1];
} FILE_BOTH_DIR_INFORMATION, *PFILE_BOTH_DIR_INFORMATION;
```

9.5: NTFS Streams

The NTFS file system supports *alternate streams*, which are additional data items stored with a file. A file can have multiple alternate streams, each with its own name. The main stream is unnamed, and is the one that is used by default as the file contents. Streams can be created/opened just like regular files, where the stream name follows the file name, separated by a colon. The following example creates an alternate stream and writes some data to it (error handling omitted):

```
HANDLE hFile;
UNICODE_STRING name;
IO_STATUS_BLOCK ioStatus;
OBJECT_ATTRIBUTES attr = RTL_CONSTANT_OBJECT_ATTRIBUTES(&name, 0);
RtlInitUnicodeString(&name, L"\\??\\C:\\Temp\\myfile.txt:mystream");
NtCreateFile(&hFile, GENERIC_WRITE | SYNCHRONIZE,
    &attr, &ioStatus, nullptr, 0,
    FILE_SHARE_WRITE, FILE_OPEN_IF, FILE_SYNCHRONOUS_IO_NONALERT, nullptr, 0);
char data[] = "Hello, stream!";
NtWriteFile(hFile, nullptr, nullptr, nullptr, &ioStatus,
    data, (ULONG)strlen(data), nullptr, nullptr);
NtClose(hFile);
```

Note that `NtCreateFile` must be used to create alternate streams. If a stream exists, `NtOpenFile` works as well. The size of the file as reflected in standard tools like *Windows Explorer* show the size of the main stream only. This means you can create a file with zero size, but have alternate streams with data of any size, practically invisible by standard tools. The *Streams* tool from *Sysinternals* will list the streams in a file:

```
C:\temp>streams -nobanner myfile.txt
C:\temp\myfile.txt:
        :mystream:$DATA 14
```

My own *NTFS Streams* tool can show the contents of streams as well as their name and size (figure 9-3).

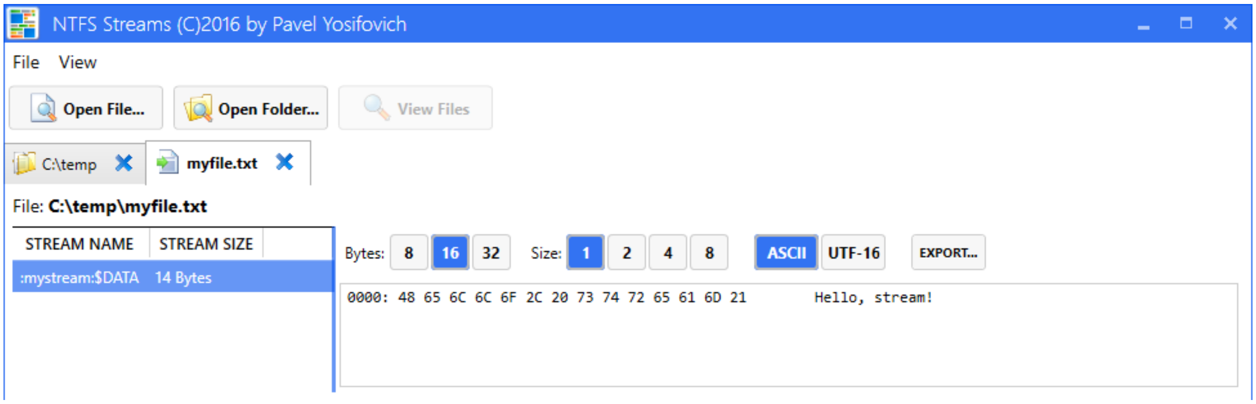


Figure 9-3: NTFS Streams

As you can see, reading and writing to alternate streams is no different than "normal" file. Enumerating streams in a file is a different story. The `NtQueryInformationFile` function can be used with the `FileStreamInformation` (22) information class to enumerate the streams in a file. The expected structure is `FILE_STREAM_INFORMATION`:

```
typedef struct _FILE_STREAM_INFORMATION {
    ULONG NextEntryOffset;
    ULONG StreamNameLength;
    LARGE_INTEGER StreamSize;
    LARGE_INTEGER StreamAllocationSize;
    WCHAR StreamName[1];
} FILE_STREAM_INFORMATION, *PFILE_STREAM_INFORMATION;
```

The following example shows how to enumerate the streams in a file (see full example in the *Streams* sample):

```
NTSTATUS EnumStreams(PCWSTR filename) {
    std::wstring path(filename);
    // deal with a drive letter
    if (filename[1] == L':')
        path = L"\\??\\" + path;

    UNICODE_STRING name;
    RtlInitUnicodeString(&name, path.c_str());
    HANDLE hFile;
    IO_STATUS_BLOCK ioStatus;
```

```
    OBJECT_ATTRIBUTES attr = RTL_CONSTANT_OBJECT_ATTRIBUTES(&name, 0);
    auto status = NtOpenFile(&hFile, FILE_READ_ACCESS | SYNCHRONIZE,
        &attr, &ioStatus, FILE_SHARE_READ | FILE_SHARE_WRITE,
        FILE_SYNCHRONOUS_IO_NONALERT);
    if (!NT_SUCCESS(status))
        return status;

    BYTE buffer[1 << 10];
    status = NtQueryInformationFile(hFile, &ioStatus,
        buffer, sizeof(buffer), FileStreamInformation);
    if (!NT_SUCCESS(status))
        return status;

    auto info = (FILE_STREAM_INFORMATION*)buffer;
    for (;;) {
        printf("Name: %.*ws Size: %llu bytes\n",
            int(info->StreamNameLength / sizeof(WCHAR)), info->StreamName,
            info->StreamSize.QuadPart);
        if (info->NextEntryOffset == 0)
            break;

        info = (FILE_STREAM_INFORMATION*)((PBYTE)info + info->NextEntryOffset);
    }

    NtClose(hFile);
    return status;
}
```

Here is what the output looks like for an "empty" *myfile.txt* from the earlier example:

```
C:\winnativeapibooksamples\Chapter09\>Streams.exe c:\temp\myfile.txt
Name: ::$DATA Size: 0 bytes
Name: :mystream:$DATA Size: 14 bytes
```

9.6: Extended Attributes

Extended attributes (EA) are a way to store additional information about a file or directory. They are not supported by all file systems, but NTFS supports them. One way to look at EA is as another way to associate custom information with a file.

There are a couple of information information classes that deal with EA. The first is `FileEaInformation` (7), which returns the size of the EA in bytes as a `ULONG`. The second, and more interesting, is `FileFullEaInformation` (15), which returns the EA in a (possibly) chain of `FILE_FULL_EA_INFORMATION` structures:

```
typedef struct _FILE_FULL_EA_INFORMATION {
    ULONG NextEntryOffset;
    UCHAR Flags;
    UCHAR EaNameLength;
    USHORT EaValueLength;
    CHAR EaName[1];
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

`NextEntryOffset` indicates the number of bytes to move to the next attribute, where zero means there are no more attributes. `Flags` is either zero or `FILE_NEED_EA` (0x80) to indicate that the EA is required for proper operation of the file. `EaNameLength` is the length of the EA name in bytes, and the name itself (`EaName`) consists of ASCII characters stored internally as uppercase. `EaValueLength` is the length of the EA value in bytes. The EA value follows the structure. This is depicted in figure 9-4.

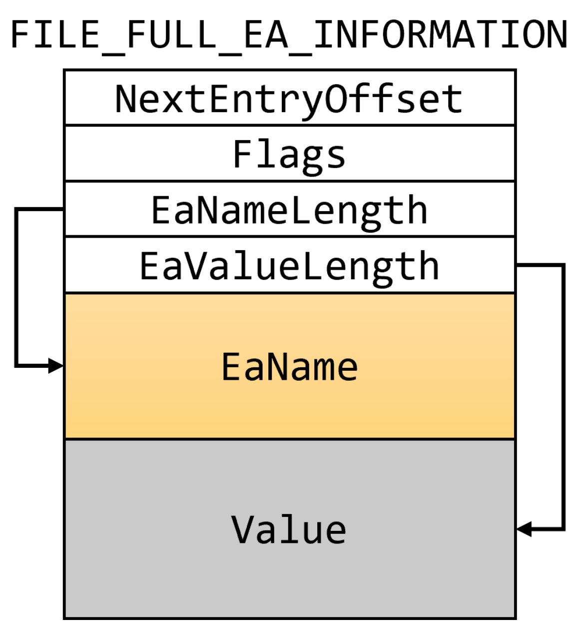


Figure 9-4: Layout

There are two ways to query the EA of a file. The first is calling `NtQueryInformationFile` with `FileFullEaInformation` and interpreting the returned buffer as a chain of `FILE_FULL_EA_INFORMATION` structures. This works, but is limited in flexibility. The second is calling `NtQueryEaFile`, which returns the EA, which provides more flexibility as the expense of complexity:

```
NTSTATUS NtQueryEaFile(
    _In_ HANDLE FileHandle,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _Out_writes_bytes_(Length) PVOID Buffer,
    _In_ ULONG Length,
    _In_ BOOLEAN ReturnSingleEntry,
    _In_reads_bytes_opt_(EaListLength) PVOID EaList,
    _In_ ULONG EaListLength,
    _In_opt_ PULONG EaIndex,
    _In_ BOOLEAN RestartScan);
```

The first four parameters should be familiar. `ReturnSingleEntry` indicates whether to return a single EA or all EAs. `EaList` is a list of EA names to return. If `NULL` is specified, all EAs are returned. `EaListLength` is the length of the list in bytes. `EaIndex` indicates the index of the EA to return. If `NULL` is specified, the first EA or all EAs are returned (based on `ReturnSingleEntry`).

`RestartScan` indicates whether to restart the scan from the beginning or continue from the last entry. This is useful when the buffer provided is not large enough to receive all entries. The first time `NtQueryEaFile` is called with a particular handle, `RestartScan` is ignored. It is respected on subsequent calls.

The following example shows how to retrieves all EAs of a file:

```
BYTE buffer[1 << 10];
// hFile is an open handle to a file
for (;;) {
    status = NtQueryEaFile(hFile, &ioStatus, buffer, sizeof(buffer), FALSE,
        nullptr, 0, nullptr, FALSE);
    if (!NT_SUCCESS(status))
        break;

    auto info = (FILE_FULL_EA_INFORMATION*)buffer;
    for (;;) {
        //
        // assume for demonstration purposes that the
        // EA value is a Unicode string
        printf("Name: %.*s Value: %.*ws\n",
            info->EaNameLength, info->EaName,
            (int)(info->EaValueLength / sizeof(WCHAR)),
            (PCWSTR)(info->EaName + info->EaNameLength + 1));
        if (info->NextEntryOffset == 0)
            break;

        info = (PFILE_FULL_EA_INFORMATION)((PBYTE)info + info->NextEntryOffset);
```

```
    }
}
```

To write an EA, use `NtSetEaFile`:

```
NTSTATUS NtSetEaFile(
    _In_ HANDLE FileHandle,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_reads_bytes_(Length) PVOID Buffer,
    _In_ ULONG Length);
```

The expected buffer is a chain of `FILE_FULL_EA_INFORMATION` structures, as described earlier. Each one must be formatted as shown in figure 9-4. The following example shows how to set a single EA (assuming the value is a Unicode string):

```
NTSTATUS WriteEaToFile(HANDLE hFile, PCSTR name, PCWSTR value) {
    //
    // use a trick to allow FILE_FULL_EA_INFORMATION to extend
    // beyond its static bounds without generating memory access violation
    //
    union {
        FILE_FULL_EA_INFORMATION info{};
        BYTE buffer[1 << 10];
    };

    //
    // copy name
    //
    strcpy_s(info.EaName, info.EaNameLenth + 1, name);
    info.NextEntryOffset = 0;
    info.EaValueLength = USHORT(1 + info.EaNameLength) * sizeof(WCHAR);
    wcscpy_s((PWSTR)(info.EaName + info.EaNameLength + 1),
        info.EaValueLength / sizeof(WCHAR), value);
    return NtSetEaFile(hFile, &ioStatus, &info, sizeof(buffer));
}
```

It's ok to indicate the buffer is bigger than it actually is, because the file system driver will follow the chain based on `NextEntryOffset`.

> The full sample is in the *ea* project.

9.7: Accessing Devices

The `NtCreateFile` and `NtOpenFile` functions can be used to open device objects that are not necessarily file system files or volumes. Any named *Device* object can be potentially opened. The open operation may fail because of access rights, but all the Object Manager's namespace is accessible.

For example, the `Beep` Windows API makes a sound based on a frequency and duration. Internally, it uses a device called *Beep* to do so. The `Beep` function is synchronous - that is, the thread waits until the sound is complete. What if we wanted to play a sound asynchronously? We can access the *Beep* device directly by opening a handle to it:

```
HANDLE hFile;
UNICODE_STRING name;
RtlInitUnicodeString(&name, L"\\Device\\Beep");
OBJECT_ATTRIBUTES attr = RTL_CONSTANT_OBJECT_ATTRIBUTES(&name, 0);
IO_STATUS_BLOCK ioStatus;
status = NtOpenFile(&hFile, FILE_WRITE_DATA | SYNCHRONIZE, &attr,
    &ioStatus, FILE_SHARE_WRITE | FILE_SHARE_READ, FILE_SYNCHRONOUS_IO_NONALERT);
```

The path "\Device\Beep" can be viewed with *WinObj* or *Object Explorer* (figure 9-5).

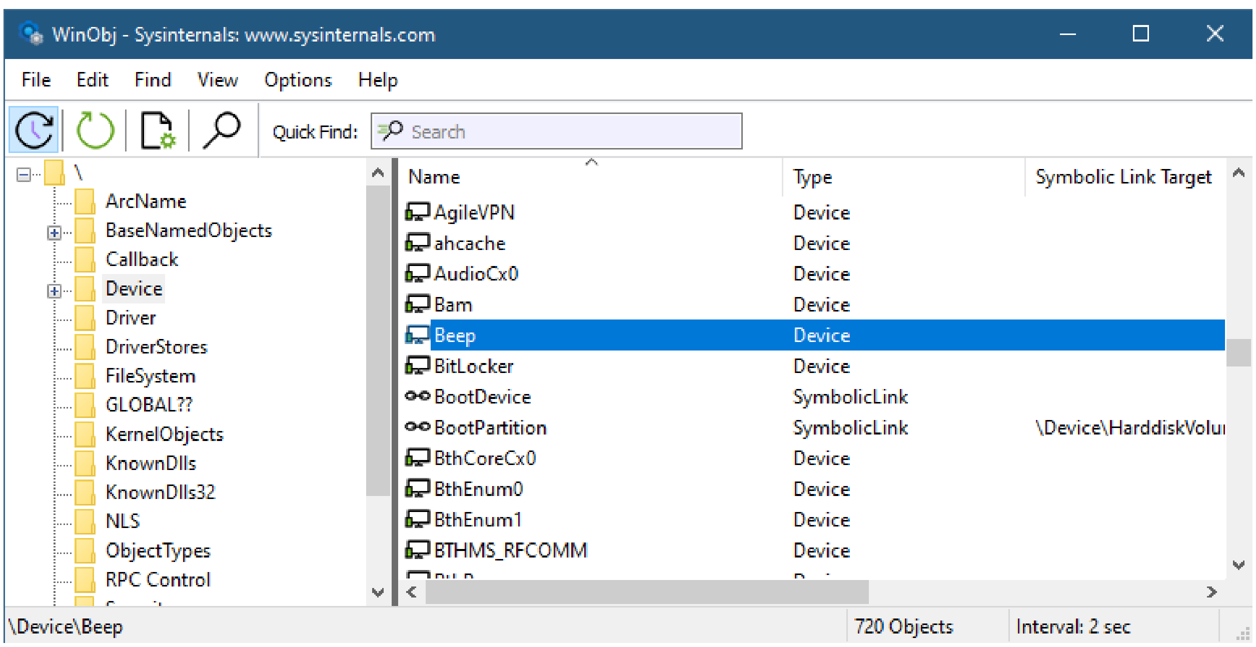


Figure 9-5: Beep device in WinObj

We've seen this kind of code multiple times. Once we have a valid handle, how do we "tell" the *Beep* device to play a sound? With files in a file system we called `NtReadFile` or `NtWriteFile`. With devices, it depends on the driver for that device. It may expect some call to `NtReadFile` or `NtWriteFile` with some formatted data that it understands, but more often than not a third API is used, `NtDeviceIoControlFile`:

```
NTSTATUS NtDeviceIoControlFile(
    _In_ HANDLE FileHandle,
    _In_opt_ HANDLE Event,
    _In_opt_ PIO_APC_ROUTINE ApcRoutine,
    _In_opt_ PVOID ApcContext,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_ ULONG IoControlCode,
    _In_reads_bytes_opt_(InputBufferLength) PVOID InputBuffer,
    _In_ ULONG InputBufferLength,
    _Out_writes_bytes_opt_(OutputBufferLength) PVOID OutputBuffer,
    _In_ ULONG OutputBufferLength);
```

`NtDeviceIoControlFile` is the native (rough) equivalent of the `DeviceIoControl` Windows API, just like `NtReadFile` and `NtWriteFile` are the native equivalents of `ReadFile` and `WriteFile`.

`NtDeviceIoControlFile` provides more flexibility when talking to devices. The `IoControlCode` is a 32-bit value that is interpreted by the driver as an operation to perform. Of course, if the value is unrecognized by the driver, the call will fail. The `InputBuffer` optional buffer can be used to send input data to the driver, while `OutputBuffer` may be used to receive results. This all depends on the driver and the way it expects to be communicated with.

How do we know the correct way to communicate with the *Beep* device? Except for reverse engineering its operation, we need documentation. Fortunately, for the *Beep* device, Microsoft provides the *Ntddbeep.h* header with the information needed.

Within that file, we find the following definitions (slightly edited for clarity):

```
#define DD_BEEP_DEVICE_NAME    "\\Device\\Beep"
#define DD_BEEP_DEVICE_NAME_U L"\\Device\\Beep"

#define IOCTL_BEEP_SET \
    CTL_CODE(FILE_DEVICE_BEEP, 0, METHOD_BUFFERED, FILE_ANY_ACCESS)

typedef struct _BEEP_SET_PARAMETERS {
    ULONG Frequency;
    ULONG Duration;
} BEEP_SET_PARAMETERS, *PBEEP_SET_PARAMETERS;

#define BEEP_FREQUENCY_MINIMUM 0x25
#define BEEP_FREQUENCY_MAXIMUM 0x7FFF
```

The header includes the device name in ASCII and Unicode formats. `BEEP_SET_PARAMETERS` is the expected input buffer to the driver. `IOCTL_BEEP_SET` is the expected control code; no need to guess anything. The following example plays a single sound with a frequency of 600Hz for 2 seconds:

```
BEEP_SET_PARAMETERS params;
params.Duration = 2000;     // msec
params.Frequency = 600;     // Hz

status = NtDeviceIoControlFile(hFile, nullptr, nullptr, nullptr, &ioStatus,
    IOCTL_BEEP_SET, &params, sizeof(params), nullptr, 0);
```

Although the device was opened for synchronous access, as it happens, the *Beep* driver works asynchronously; the call to `NtDeviceIoControlFile` returns immediately, while the sound is played in the background. We can "prove" that by waiting for the sound to complete while displaying something from time to time:

```
LARGE_INTEGER ticks, delay;
NtQuerySystemTime(&ticks);
delay.QuadPart = -250000;   // 25msec
// calculate future time based on the duration
auto target = ticks.QuadPart + params.Duration * 10000;

while (ticks.QuadPart < target) {
    printf(".");
    // wait a bit
    NtDelayExecution(FALSE, &delay);
    NtQuerySystemTime(&ticks);
}
```

The full sample is in the *Beep* project.

In addition to `NtDeviceIoControlFile`, there is `NtFsControlFile`, which is used to communicate with file system drivers. It's similar to `NtDeviceIoControlFile`, having the exact same arguments, but the expected control codes are different, starting with `FSCTL_`. For example, the *NTFS* file system driver supports the `FSCTL_GET_NTFS_VOLUME_DATA` control code, which returns information about the volume the file handle was opened to. It returns a `NTFS_VOLUME_DATA_BUFFER` structure:

```
typedef struct {
    LARGE_INTEGER VolumeSerialNumber;
    LARGE_INTEGER NumberSectors;
    LARGE_INTEGER TotalClusters;
    LARGE_INTEGER FreeClusters;
    LARGE_INTEGER TotalReserved;
    DWORD BytesPerSector;
    DWORD BytesPerCluster;
    DWORD BytesPerFileRecordSegment;
    DWORD ClustersPerFileRecordSegment;
    LARGE_INTEGER MftValidDataLength;
    LARGE_INTEGER MftStartLcn;
    LARGE_INTEGER Mft2StartLcn;
    LARGE_INTEGER MftZoneStart;
    LARGE_INTEGER MftZoneEnd;
} NTFS_VOLUME_DATA_BUFFER, *PNTFS_VOLUME_DATA_BUFFER;
```

Here is an example:

```
NTFS_VOLUME_DATA_BUFFER data;
status = NtFsControlFile(hFile, nullptr, nullptr, nullptr, &ioStatus,
    FSCTL_GET_NTFS_VOLUME_DATA, nullptr, 0, &data, sizeof(data));
```

The above file handle can be to any file or directory for the requested volume. Curiously enough, calling `NtDeviceIoControlFile` with the same control code fails with `STATUS_INVALID_PARAMETER`. However, since `NtFsControlFile` is not part of the documented Windows API, calling `DeviceIoControl` (the documented API) with the same control code works just fine.

All standard control codes and structures that may be supported by file systems are defined in <*WinIoCtl.h*>, many of which are officially documented.

9.8: I/O Completion Ports

I/O completion (port) objects are a way to invoke handlers when asynchronous I/O operations complete. There are multiple ways to know when an asynchronous I/O operation completes, such as using the (optional) event handle provided to many I/O functions. I/O completion objects provide the most efficient way to do so, by allowing multiple threads to respond to I/O completions, and optionally by utilizing a thread pool, from which threads can pick up completed operations and run handlers.

An I/O completion object can be associated with multiple file objects. It can also be used independently of any I/O operations as a mechanism to post operations that can be handled by multiple threads. Figure 9-6 shows the main ingredients of an I/O completion object.

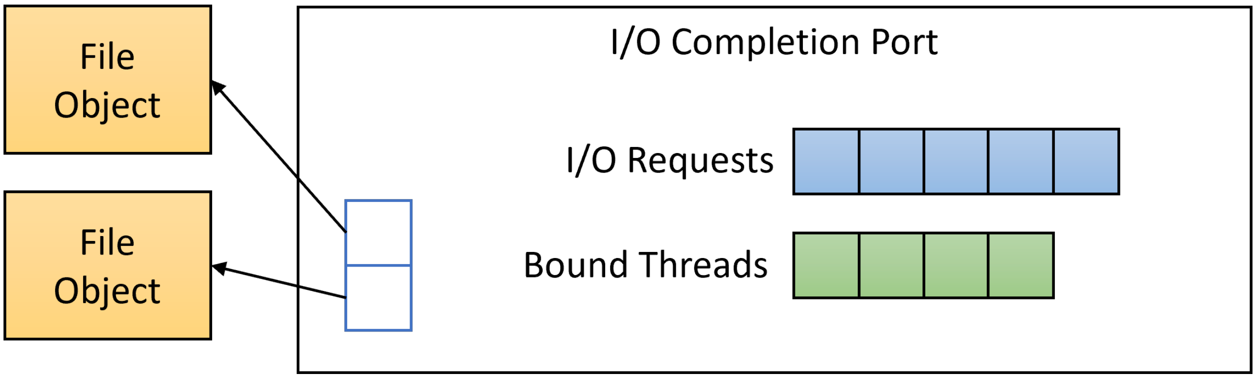


Figure 9-6: I/O Completion (port) object

Creating an I/O completion object is done with `NtCreateIoCompletion`:

```
NTSTATUS NtCreateIoCompletion(
    _Out_ PHANDLE IoCompletionHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_opt_ ULONG Count);
```

`DesiredAccess` is the desired access for the new object, typically `IO_COMPLETION_ALL_ACCESS` (defined in <*WinNt.h*>). `ObjectAttributes` can be provided and can contain a name. Finally, `Count` indicates how many threads at most can handle I/O completions. Zero is interpreted as the number of logical processors on the system.

Curiously enough, the Windows API `CreateIoCompletionPort` does not allow specifying a name. This makes some sense, as sharing I/O completion objects between processes is not very practical, so the value of having a name is diminished; a name is supported nonetheless.

Creating named objects outside the session namespace (such as the global namespace) requires the `SeCreateGlobalPrivilege` privilege, which is granted by default to the Administrators group.

Since a name is allowed, it's possible to open a handle to an existing I/O completion object with `NtOpenIoCompletion`:

```
NTSTATUS NtOpenIoCompletion(
    _Out_ PHANDLE IoCompletionHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes);
```

In this case, `ObjectAttributes` is not optional.

To associate a file handle with an I/O completion object, use `NtSetInformationFile` with `FileCompletionInformation` (30), where the expected buffer is `FILE_COMPLETION_INFORMATION`:

```
typedef struct _FILE_COMPLETION_INFORMATION {
    HANDLE Port;
    PVOID Key;
} FILE_COMPLETION_INFORMATION, *PFILE_COMPLETION_INFORMATION;
```

The call can be made multiple times with different file handles but the same port. `Port` is the port handle returned from `NtCreateIoCompletion` or `NtOpenIoCompletion`. `Key` is an arbitrary value that can be used to identify the specific file.

When an I/O operation is performed, any number of threads can call `NtRemoveIoCompletion` to wait for an operation to complete. Once completed, the thread is unblocked and can do whatever is needed. The maximum number of threads released from `NtRemoveIoCompletion` is based on what has been specified in the I/O completion object creation. Here is `NtRemoveIoCompletion` and its extended version, `NtRemoveIoCompletionEx`:

```
NTSTATUS NtRemoveIoCompletion(
    _In_ HANDLE IoCompletionHandle,
    _Out_ PVOID *KeyContext,
    _Out_ PVOID *ApcContext,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_opt_ PLARGE_INTEGER Timeout);
NTSTATUS NtRemoveIoCompletionEx(
    _In_ HANDLE IoCompletionHandle,
    _Out_writes_to_(Count, *Removed) PFILE_IO_COMPLETION_INFORMATION Information,
    _In_ ULONG Count,
    _Out_ PULONG Removed,
    _In_opt_ PLARGE_INTEGER Timeout,
    _In_ BOOLEAN Alertable);
```

The simpler function waits until an asynchronous I/O operation completes (or the timeout elapses). If completed, it can run any code needed at that time. The information returned for the completed operation is the `*KeyContext` (the key supplied to `NtSetInformationFile`), `*ApcContext` (value passed to the asynchronous I/O operation), and `*IoStatusBlock`, indicating the result of the operation.

The extended function can wait for multiple operations to complete, up to `Count` operations. The `*Removed` parameter indicates how many operations have completed. The information for each completed operation is returned in the `Information` array, of the following type:

```
typedef struct _FILE_IO_COMPLETION_INFORMATION {
    PVOID               KeyContext;
    PVOID               ApcContext;
    IO_STATUS_BLOCK     IoStatusBlock;
} FILE_IO_COMPLETION_INFORMATION, *PFILE_IO_COMPLETION_INFORMATION;
```

This basically bundles the same information returned individually by `NtRemoveIoCompletion`. Finally, `Alertable` indicates whether the wait should be alertable or not.

The return values for these functions can be `STATUS_SUCCESS` (if an operation completed), `STATUS_TIMEOUT` (if the timeout elapsed), or `STATUS_USER_APC` (if the wait was alertable and APC(s) executed).

As mentioned earlier, I/O completion objects can be used to post operations that can be waited on by multiple threads regardless of (or in addition to) any I/O operation. This is done with `NtSetIoCompletion` or its extended version, `NtSetIoCompletionEx`:

```
NTSTATUS NtSetIoCompletion(
    _In_ HANDLE IoCompletionHandle,
    _In_opt_ PVOID KeyContext,
    _In_opt_ PVOID ApcContext,
    _In_ NTSTATUS IoStatus,
    _In_ ULONG_PTR IoStatusInformation);
NTSTATUS NtSetIoCompletionEx(
    _In_ HANDLE IoCompletionHandle,
    _In_ HANDLE IoCompletionPacketHandle,
    _In_opt_ PVOID KeyContext,
    _In_opt_ PVOID ApcContext,
    _In_ NTSTATUS IoStatus,
    _In_ ULONG_PTR IoStatusInformation);
```

`NtSetIoCompletion` queues an item to the I/O completion object that can be picked up by one of threads calling `NtRemoveIoCompletion(Ex)`. The meaning of the parameters other than the completion object handle are up to the caller. The extended function uses a I/O Reserve object handle, which is out of scope for this chapter.

Finally, a thread pool can be used to handle I/O completion object's "completions". This requires creating a thread pool I/O with `TpAllocIoCompletion`. This is left as an exercise to the interested reader.

9.9: Miscellaneous Functions

The following subsection briefly discuss other I/O related APIs.

9.9.1: Drivers

A kernel driver can be loaded into the system by calling `NtLoadDriver`:

```
NTSTATUS NtLoadDriver(_In_ PUNICODE_STRING DriverServiceName);
```

Where `DriverServiceName` is the name of the driver's Registry key stored under *\HKLM\System\CurrentControlSet\Service*
Figure 9-7 shows this key and some of its subkey in *RegEdit.exe*.

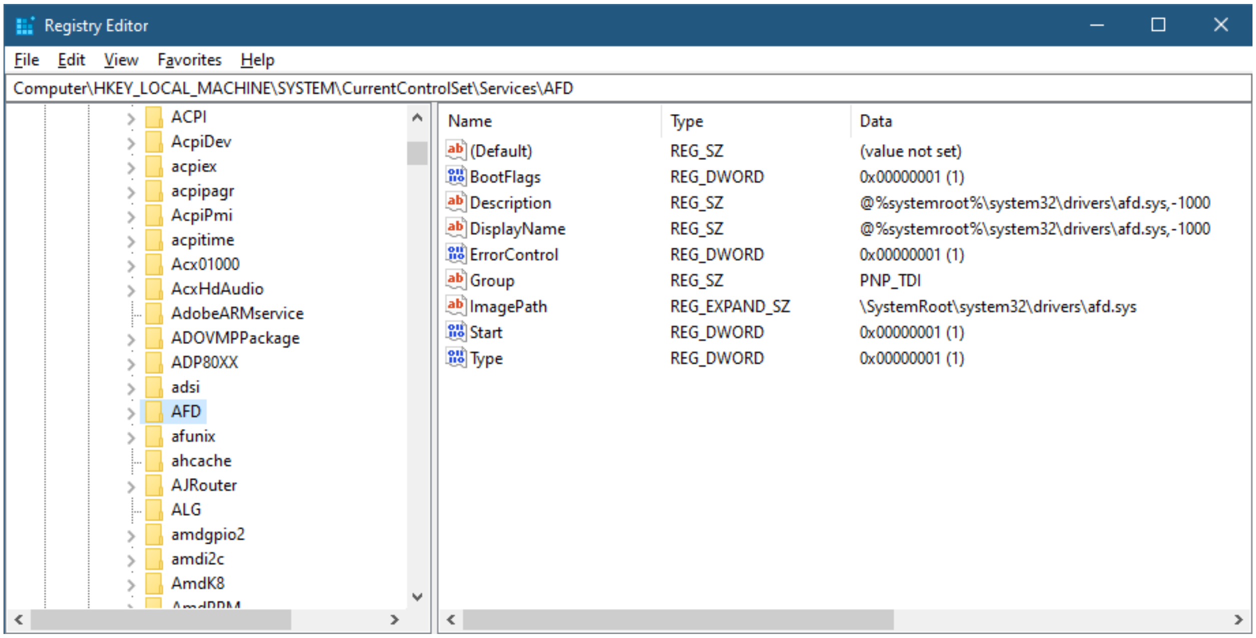


Figure 9-7: Services Registry Key

Drivers are typically installed using *INF* files, or command line tools like *Sc.Exe*. Loading a driver is a privilege operation, granted by default to the Administrators group.

After a driver is loaded, its Registry key can be deleted without any adverse effect.

The opposite function, `NtUnloadDriver`, can be used to unload (stop) a driver:

```
NTSTATUS NtUnloadDriver(_In_ PUNICODE_STRING DriverServiceName);
```

9.9.2: Locking Files

A range of bytes in a file can be locked using `NtLockFile`:

```
NTSTATUS NtLockFile(
    _In_ HANDLE FileHandle,
    _In_opt_ HANDLE Event,
    _In_opt_ PIO_APC_ROUTINE ApcRoutine,
    _In_opt_ PVOID ApcContext,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_ PLARGE_INTEGER ByteOffset,
    _In_ PLARGE_INTEGER Length,
    _In_ ULONG Key,
```

```
    _In_ BOOLEAN FailImmediately,
    _In_ BOOLEAN ExclusiveLock);
```

The first five parameters should be familiar by now. `*ByteOffset` is the offset to start locking from. Note that `NULL` is not a valid value, and will cause an access violation. `*Length` is the number of bytes to lock (`NULL` is illegal). `Key` is an arbitrary value used to identify the lock. `FailImmediately` indicates whether to fail if the lock cannot be acquired immediately - even if the lock operation is specified as asynchronous. `ExclusiveLock` indicates whether to acquire an exclusive lock (`TRUE`) or a shared lock (`FALSE`).

Once the lock is no longer needed, `NtUnlockFile` can be used to release it:

```
NTSTATUS NtUnlockFile(
    _In_ HANDLE FileHandle,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_ PLARGE_INTEGER ByteOffset,
    _In_ PLARGE_INTEGER Length,
    _In_ ULONG Key);
```

9.9.3: Change Notifications

Changes in a file system can be detected by calling `NtNotifyChangeDirectoryFile` or `NtNotifyChangeDirectoryFileEx`:

```
NTSTATUS NtNotifyChangeDirectoryFile(
    _In_ HANDLE FileHandle,
    _In_opt_ HANDLE Event,
    _In_opt_ PIO_APC_ROUTINE ApcRoutine,
    _In_opt_ PVOID ApcContext,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _Out_writes_bytes_(Length) PVOID Buffer, // FILE_NOTIFY_INFORMATION
    _In_ ULONG Length,
    _In_ ULONG CompletionFilter,
    _In_ BOOLEAN WatchTree);
NTSTATUS NtNotifyChangeDirectoryFileEx(
    _In_ HANDLE FileHandle,
    _In_opt_ HANDLE Event,
    _In_opt_ PIO_APC_ROUTINE ApcRoutine,
    _In_opt_ PVOID ApcContext,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _Out_writes_bytes_(Length) PVOID Buffer,
    _In_ ULONG Length,
    _In_ ULONG CompletionFilter,
```

```
    _In_ BOOLEAN WatchTree,
    _In_opt_ DIRECTORY_NOTIFY_INFORMATION_CLASS DirectoryNotifyInformationClass);
```

`NtNotifyChangeDirectoryFile` is a shortcut that calls `NtNotifyChangeDirectoryFileEx` with `DirectoryNotifyInformationClass` set to `DirectoryNotifyInformation`. `DirectoryNotifyInformationClass` indicates the data buffer to return:

```
typedef enum _READ_DIRECTORY_NOTIFY_INFORMATION_CLASS {
    ReadDirectoryNotifyInformation        = 1,    // FILE_NOTIFY_INFORMATION
    ReadDirectoryNotifyExtendedInformation,       // FILE_NOTIFY_EXTENDED_INFORMATION
#if (NTDDI_VERSION >= NTDDI_WIN10_NI)
    ReadDirectoryNotifyFullInformation,           // FILE_NOTIFY_FULL_INFORMATION
#endif
} READ_DIRECTORY_NOTIFY_INFORMATION_CLASS;
```

The associated structures are defined in <*WinNt.h*>. `FileHandle` must be a directory handle (rather than a file handle). The operation completes when a change is detected. `Buffer` is where the information is returned when the operation completes, and `Length` is the size of the buffer in bytes. `CompletionFilter` is a set of flags that indicate the type of changes to detect (defined in <*WinNt.h*> as well). For example: `FILE_NOTIFY_CHANGE_FILE_NAME` indicates to detect file name changes, `FILE_NOTIFY_CHANGE_DIR_NAME` indicates to detect directory name changes, and `FILE_NOTIFY_CHANGE_LAST_WRITE` indicates to detect changes to the last write time. `WatchTree` indicates whether to watch the directory tree recursively.

If called synchronously, the call blocks until a change is detected. A simple way to use this function is to call it synchronously in a loop, handling each change as they come in.

9.10: Summary

I/O APIs are about working with files and devices. This chapter looked at the most common native I/O APIs, some of which are officially documented in the WDK. The I/O system supports synchronous and asynchronous access, where the requested mode is provided in the `NtOpenFile` or `NtCreateFile` call.

In the next chapter, we'll examine one of Windows undocumented features - *Asynchronous Local Procedure Calls* (ALPC).